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

STÉFANO DRIMON KURZ MÓR

Escalonamento On-Line Eficiente de Programas Fork-Join Recursivos do Tipo Divisão e Conquista em MPI

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

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, Março de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Mór, Stéfano Drimon Kurz

Escalonamento On-Line Eficiente de Programas Fork-Join Recursivos do Tipo Divisão e Conquista em MPI / Stéfano Drimon Kurz Mór. – Porto Alegre: PPGC da UFRGS, 2010.

104 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Orientador: Nicolas Maillard.

1. MPI-2. 2. Escalonamento. 3. Tarefas. 4. Dinamismo. I. Maillard, Nicolas. II. Título.

"It is a mistake to think you can solve any major problems just with potatoes."
— Douglas Adams

AGRADECIMENTOS

- Agradeço a Deus.
- Agradeço aos meus pais e toda a minha família.
- Agradeço ao meu orientador, Nicolas Maillard.
- Agradeço aos meus amigos, em especial os que dividem a sala comigo: Alexandre Almeida, Claúdio Schepke, Cristian Casteñeda, Fernando Afonso e João Vicente Lima.
- Agradeço à Jamile Moroszczuk.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

DAG	*Directed Acyclic Graph*
DFWS	*Deque-free Workstealing*
EAU	Escalonador de Alta Utilização
EMU	Escalonador de Máxima Utilização
EQD	Equiparticionamento Dinâmico
FPGA	*Field-Programmable Gate Array*
GPU	*Graphical Processor Unit*
HWS	*Hierarquical Workstealing*
LAN	*Local Area Network*
MIMD	*Multiple Instruction, Multiple Data*
MPI	*Message Passing Interface*
NT	*Nomadic Threads*
PAD	Programação para Alto Desempenho
RFB	Recebedor Faz Bem
SMP	*Simultaneous Multi Processor*
SPMD	*Single Program, Multiple Data*
TBB	*Thread Building Blocks*
UMA	*Uniform Memory Access*
WS	*Work Stealing*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE TEOREMAS E COROLÁRIOS

RESUMO

Esta Dissertação de Mestrado propõe dois novos algoritmos para tornar mais eficiente o escalonamento *on-line* de tarefas com dependências estritas em agregados de computadores que usam como *middleware* para troca de mensagens alguma implementação da MPI (até a versão 2.1). Esses algoritmos foram projetados tendo-se em vista programas construídos no modelo de programação *fork/join*, onde a operação de *fork* é usada sobre uma chamada recursiva da função. São eles:

1. O algoritmo RatMD, implementado através de uma biblioteca de primitivas do tipo *map-reduce*, que funciona para qualquer implementação MPI, com qualquer versão da norma. Utilizado para minimizar o tempo de execução de uma computação paralela; e

2. O algoritmo RtMPD, implementado através de um sistema distribuído sobre *daemons* gerenciadores de processos criados dinamicamente com a implementação MPICH2 (que implementa a MPI-2). Utilizado para permitir execuções de instâncias maiores de programas paralelos dinâmicos.

Ambos se baseiam em roubo de tarefas, que é a estratégia de balanceamento de carga mais difundida na literatura. Para ambos os algoritmos apresenta-se modelagem téorica de custos. Resultados experimentais obtidos ficam dentro dos limites teóricos calculados. RatMD provê uma redução no tempo de execução de até 80% em relação ao algoritmo usual (baseado em *round-robin*), com manutenção do *speedup* próximo ao linear e complexidade espacial idêntica à popular implementação com *round-robin*. RtMPD mantém, no mínimo, o mesmo desempenho que a implementação canônica do escalonamento em MPICH2, dobrando-se o limite físico de processos executados simultaneamente por cada nó.

Palavras-chave: MPI-2, Escalonamento, Tarefas, Dinamismo.

Efficent On-line Scheduling of Recursive Fork-Join Programs on MPI

ABSTRACT

This Master's Dissertation proposes two new algorithms for improvement on on-line scheduling of dynamic-created tasks with strict dependencies on clusters of computers using MPI (up to version 2.1) as its middleware for message-passing communication. These algorithms were built targeting programs written on the fork-join model, where the fork operation is always called over an recursive function call. They are:

1. RatMD, implemented as a map-reduce library working for any MPI implementation, on whatever norm's version. Used for performance gain; and
2. RtMPD, implemented as a distributed system over dynamic-generated processes manager daemons with MPICH2 implentation of MPI. Used for executing larger instances of dynamic parallel programs.

Both algorithms are based on the (literature consolidated) work stealing technique and have formal guarantees on its execution time and load balancing. Experimental results are within theoretical bounds. RatMD shows an improvement on the performance up to 80% when paired with more usual algorithms (based on round-robin strategy). It also provides near-linear speedup and just about the same space-complexity on similar implementations. RtMPD keeps, at minimum, the very same performance of the canonical MPICH2 implementation, near doubling the physical limit of simultaneous program execution per cluster node.

Keywords: MPI, scheduling, dynamic, fork-join, recursive.

1 INTRODUÇÃO

A exploração eficiente de máquinas paralelas depende da definição do que cada recurso de processamento executa. Em outras palavras, o aproveitamento do paralelismo está diretamente ligado à definição de *tarefa* a ser executada. Na mesma proporção, uma computação paralela eficiente depende de um bom escalonamento, *i.e.*, a definição de quais recursos executarão quais tarefas.

O conceito de tarefa, em programação paralela, é a maneira oferecida para que o programador expresse que partes do programa podem executar paralelamente e que partes do programa devem ser executadas em sequência. Tarefas que devam executar sequencialmente possuem *dependências*. Um conjunto de tarefas sem nenhuma dependência é uma "sacola de tarefas" e seus componentes podem ser executados em paralelo ou sequencialmente, em qualquer ordem. Um sistema paralelo qualquer possui criação dinâmica de tarefas quando uma tarefa pode originar sub-tarefas em tempo de execução. Estas sub-tarefas geradas dinamicamente podem ou não possuir dependências com outras tarefas.

Informalmente, uma tarefa é um fluxo sequencial de instruções de processamento associado a uma entrada específica. Uma tarefa é consumida quando o fluxo de instruções processa a entrada e gera uma saída. Uma variedade de *middlewares* para programação paralela implementa a noção de tarefa usando abstrações distintas. *v.g.*,

Intel's *Thread Building Blocks* (TBB) , direcionada a *chips multi-core*, é uma biblioteca C++ que implementa paralelismo baseado em tarefas criadas dinamicamente de duas maneiras: (1) uma tarefa é um objeto que herda, de uma classe especial do tipo *container*, (fornecida pela biblioteca), um método para a execução, parametrizado com a respectiva entrada; (2) uma tarefa é uma iteração de um método fornecido pela biblioteca que executa um laço sobre variáveis locais. Em ambos os casos, tarefas sem dependências executam paralelamente através da alocação desses fluxos de instruções em *threads* concorrentes, associadas ao programa através do sistema operacional. O escalonamento das tarefas sobre as *threads* é feito através da técnica de roubo de tarefas. As dependências entra as tarefas no caso (1) respeitam o modelo estrito de dependências, apresentado mais à frente.

OpenMP3 , uma extensão para compiladores C/C++ que implementa paralelismo de tarefas de maneira análoga à TBB: (1) uma tarefa é uma chamada de função C delimitada por diretivas ao compilador; (2) uma tarefa é uma iteração de um laço `for`, também delimitado por diretivas ao compilador. O paralelismo

é dado pelo uso de *threads* POSIX (BUTENHOF, 1997) em *chips multi-core*. O caso (1) também segue o modelo estrito de dependências. O escalonamento também é feito via roubo de tarefas.

Nvidia's CUDA , implementa paralelismo de tarefas como a execução de uma função C (chamada "*kernel*"), com sintaxe restrita, em GPUs *multi-core*. Não permite a criação dinâmica de tarefas. O escalonamento é definido manualmente no código-fonte.

Este trabalho preocupa-se com a implementação eficiente de paralelismo de tarefas e seu escalonamento em agregados de computadores (NAVAUX; DEROSE, 2003).

A corrida pelo aumento da frequência de *clock* nos microprocessadores comerciais disputada pelos principais fabricantes nas décadas de 1980 e 1990 deu lugar à corrida pela introdução de uma quantidade elevada de núcleos de processamento em um mesmo circuito integrado (ASANOVIC et al., 2009), ligados por uma rede intra-*chip*, nomeados *chips many-core* (*v.g.*, SEILER et al. 2008). Esse movimento da indústria sugere que as técnicas utilizadas para a obtenção de alto desempenho em *agregados* (*clusters*) *de computadores* serão integradas à computação cotidiana, em computadores pessoais. Subitamente, todo o trabalho desenvolvido sobre a programação em *clusters* de computadores ganha mais relevância.

A Tabela 1.1, retirada do site `www.top500.org`, que se dedica a classificar as quinhentas máquinas mais poderosas do mundo, mostra que aproximadamente 83% dessas máquinas (em Novembro/2009) são *clusters* de computadores.

Arq.	Cont.	Porc. (%)	S. Rmax (GF)	S. Rpico (GF)	S. do Proc
Const.	2	0,40%	94.970	112.947	17.648
MPP	81	16,20%	10.939.467	13.614.091	2.090.251
Cluster	417	**83,40%**	16.943.065	27.223.084	2.556.728
Totais	500	100%	27.977.501,79	40.950.122,01	4.664.627

Tabela 1.1: Tabela de arquiteturas disponíveis no TOP500. Dados de Novembro de 2009. De acordo com (Top 500 Team, 2009), o *benchmark* utilizado é o LINPACK, com 1.000 equações. "Arq." é o tipo de arquitetura considerada (destacado o tipo *cluster*). "Cont." é a quantidade de máquinas construídas em uma dada arquitetura no ranqueamento, enquanto "Porc." é a proporção da quantidade de máquinas de uma arquiteturas em relação ao total. "GF" (*GigaFloats/s*) é o número de 10^9 operações em ponto flutuante por segundo. "S. Rmax" é a soma, em GF, dos maiores desempenhos por cada máquina de uma dada arquitetura, com o LINPACK. "S. Rpico" é a soma das performances teóricas de cada máquina de uma dada arquitetura, com o LINPACK. "S. do Proc." é a soma da quantidade de ciclos de *clock* que os processadores de cada máquina de uma dada arquitetura consumiram no uso do LINPACK.

Agregados de computadores operam com memória principal fisicamente distribuída (PATTERSON; HENNESSY, 2005), que induz ao programador o uso de um modelo de comunicação via troca de mensagens (TANENBAUM, 2007).

Desde 1996 *Message Passing Interface* (MPI) é o padrão *de facto* para comunicação via troca de mensagens em máquinas *Multiple Instruction, Multiple Data*

(MIMD) (GROPP; LUSK; SKJELLUM, 1999), permitindo que processos em diferentes recursos comuniquem-se por meios de primitivas do tipo *send/receive* especificadas em C/C++/Fortran. Essas primitivas abstraem o uso da rede de comunicação. Em outras palavras, implementações MPI são o *middleware* mais popular para programação paralela em agregados de computadores.

MPI implementa paralelismo de tarefas. Cada tarefa MPI é definida tendo seu próprio espaço de endereçamento e dependências são estabelecidas através de operações bloqueantes para trocas de mensagens. Na primeira versão da norma, MPI-1 (FORUM, 1994), todas as tarefas são criadas de uma só vez, antes do início da computação, *i.e.*, não há criação dinâmica de tarefas. A norma é omissa quanto à implementação de uma tarefa do ponto de vista do sistema operacional, embora sua redação (e as principais implementações existentes) sugiram que uma tarefa MPI deva ser implementada como um processo.

Em 1997 a norma MPI ganhou uma extensão, a MPI-2 (FORUM, 1997). Dentre outros recursos, essa extensão introduziu uma família de primitivas que fornecem criação dinâmica de tarefas através da execução de programas (processos) comunicantes durante a execução de um programa MPI. Como será visto mais a frente, o escalonamento das tarefas geradas por essas primitivas é ineficiente nas principais implementações MPI quando se consideram programas paralelos no modelo *fork/join* (OLIVEIRA; CARíSSIMI; TOSCANI, 2008).

1.1 Desafio e Motivação

Um dos fatores-chave para o aproveitamento de um *cluster* é o balanceamento eficiente da distribuição de tarefas entre seus recursos (PACHECO, 1997).

MPI não oferece mecanismos para balanceamento automático de carga trabalho (*i.e.*, o número de tarefas executadas por cada recurso); o controle eficiente para a minimização da ociosidade de um dado recurso fica a cargo do programador, que deve definir suas tarefas e distribuí-las.

Programar manualmente o balanceamento de carga no sistema é uma missão problemática; podem surgir "gargalos" de desempenho. Essa deficiência fica evidente quando se usam algoritmos cuja carga de trabalho em cada nó varia de maneira imprevisível durante a execução, como é o caso de algoritmos paralelos do tipo *fork/join*.

O modelo de programação paralela *fork/join* consiste de duas fases: a divisão do fluxo de execução sequencial em fluxos de execução paralelos (instrução *fork*) e a espera até que todos os fluxos paralelos previamente lançados atinjam um ponto de convergência, para o prosseguimento sequencial da computação (instrução *join*). Uma computação *fork/join* equivale a um grafo do tipo *Directed Acyclic Graph* (DAG), como o visto na Fig. 1.1. Como a figura evidencia, um primitiva *fork* pode ser invocada mesmo que uma primitiva *join* correspondente a um *fork* mais externo ainda não tenha sido atingida.

Fork/join recursivo é uma variação mais específica deste modelo de programação paralela. Nessa variação, cada vez que o fluxo de execução é paralelizado através de uma primitiva *fork* os fluxos paralelos resultantes são execuções do mesmo procedimento, possivelmente invocados com parâmetros distintos do original. O algoritmo procede, então, recursivamente, até que uma condição de parada termine a recursão, alcance o *join* mais interno e, com isso, os *joins* mais externos vão sendo alcançados

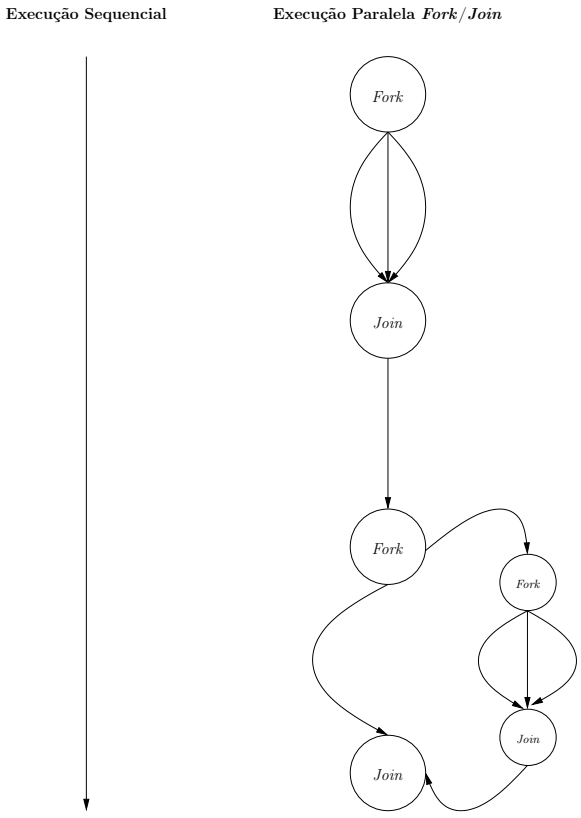


Figura 1.1: DAG para exemplo do modelo *Fork/Join*. Setas representam fluxo sequencial. Nós representam as primitivas *fork* e *join*. Setas lado-a-lado são fluxos de execução paralelos.

em cascata.

Embora pareça complexo, o modelo *fork/join* recursivo é imediato para a paralelização de algoritmos recursivos, que progridem em forma de árvore; cada chamada de função é feita em paralelo, com *fork*, e a computação só progride quando a primitiva *join* desbloqueia, ao receber os resultados da aplicação das funções disparadas em paralelo.

Tarefas no modelo *fork/join* recursivo são funções associadas aos seus parâmetros. A primitiva *fork* origina sub-tarefas que podem ser executadas em paralelo (não possuem dependências entre si). A tarefa criadora possui dependência com as tarefas criadas, já que estas devem retornar valores a serem usados posteriormente. Este modelo de dependências é chamado de *modelo estrito* (BLUMOFE; LEISERSON, 1994).

Uma classe de algoritmos importante que pode ser trivialmente paralelizada via *fork/join* recursivo são os algoritmos recursivos do tipo *Divisão e Conquista* (D&C). Esse tipo de programa torna-se interessante do ponto de vista do escalonamento de tarefas, visto que certos problemas (*v.g.*, otimização combinatorial) são implementados de maneira mais eficiente quando o algoritmo pode abdicar de executar alguma tarefa cujo valor de retorno é sabido, *a priori*, ser não ótimo. Dessa maneira, um programa paralelo consegue poupar recursos, que podem ser utilizados no processamento de tarefas que possam contribuir efetivamente na computação.

Um algoritmo paralelo que empregue paralelismo de tarefas no modelo *fork/join* recursivo é *processor-oblivious* (CUNG et al., 2006), *i.e.*, o número de recursos (processadores) disponíveis não é um parâmetro para a execução do algoritmo. Esse fato introduz facilidades no escalonamento de tarefas em ambientes onde o número de recursos varie ao longo da computação.

A granularidade de uma dada tarefa (*i.e.*, a quantidade de trabalho feita por cada recurso ao executar o fluxo de instruções de uma tarefa) não é conhecida em tempo de compilação; processadores cujas tarefas tenham um número elevado de sub-tarefas tendem a ficar sobrecarregados enquanto outros processadores, beneficiados pela decisão que o algoritmo pode ter tomado de não executar algumas tarefas (no caso de D&C), ficam ociosos. Isso produz desbalanceamento da carga de trabalho mesmo em uma máquina MIMD de recursos homogêneos, onde o a disparidade das cargas de trabalho devido às diferenças de velocidade entre os processadores tende a ser menor. Essa característica peculiar é um obstáculo para o projeto de escalonadores de tarefas eficientes, visto que a previsão do número de sub-tarefas que cada tarefa gera é, por si, um problema $\mathcal{NP}$-Completo.

A especificação MPI-1 (FORUM, 1994) não oferece mecanismos nativos para a criação dinâmica de tarefas e balanceamento automático da carga computacional.

Uma tarefa dinâmica, em MPI-2, é um processo MPI. MPI-2 não especifica uma política de escalonamento canônica para processos recém disparados; cada implementação MPI-2 pode incluir seu próprio método para assinalar tarefas a recursos. Dado o fato que as mais utilizadas implementações MPI-2 (*e.g.*, LAM-MPI, OpenMPI, MPICH2) oferecem um simples escalonamento por lista local ordenada (*round-robin*), programas *fork/join* podem gerar grande desbalanceamento e perda de desempenho ao se utilizarem delas (os problemas dessa estratégia são discutidos no Capítulo 2). Ao mesmo tempo, não há na literatura algum consenso sobre o melhor tipo de algoritmo para se utilizar no escalonador de tarefas dessas implementações.

1.2 Proposta e Objetivos

Esta Dissertação de Mestrado propõe dois novos algoritmos para tornar mais eficiente o escalonamento *on-line* de tarefas com dependências estritas em agregados de computadores que usam como *middleware* para troca de mensagens alguma implementação da MPI (até a versão 2.1). Esses algoritmos foram projetados tendo-se em vista programas construídos no modelo de programação *fork/join*, onde a operação de *fork* é usada sobre uma chamada recursiva da função. São eles:

1. O algoritmo RatMD, implementado através de uma biblioteca de primitivas do tipo *map-reduce*, que funciona para qualquer implementação MPI, com qualquer versão da norma. Utilizado para minimizar o tempo de execução de uma computação paralela; e

2. O algoritmo RtMPD, implementado através de um sistema distribuído sobre *daemons* gerenciadores de processos criados dinamicamente com a implementação MPICH2 (que implementa a MPI-2). Utilizado para permitir execuções de instâncias maiores de programas paralelos dinâmicos.

Ambos se baseiam em roubo de tarefas, que é a estratégia de balanceamento de carga mais difundida na literatura para este modelo de programa paralelo. Para ambos os algoritmos apresentam-se predições teóricas.

Este Capítulo conceitualizou o uso da técnica de *fork/join* recursivo como um modelo central na definição de tarefas em uma computação paralela e contextualizou esse abordagem no uso de agregados de computadores. O restante do trabalho é disposto como segue. O Capítulo 2 mostra o Estado da Arte do escalonamento *on-line* nos cenários de memória compartilhada e memória distribuída. O Capítulo 3 apresenta o algoritmo RatMD, limites teóricos e noções de sua implementação como uma biblioteca *map-reduce* para MPI. Do mesmo modo, o Capítulo 4 descreve o algoritmo RtMPD, seus limites teóricos e sua implementação como um sistema distribuído sobre *daemons* gerenciadores de processos. O Capítulo 5 mostra as medições realizadas e como elas confirmam os limites teóricos previamente calculados. Por fim o Capítulo 6 traça conclusões sobre as medições realizadas e apresenta os primeiros esboços de trabalhos a serem desenvolvidos no futuro, tanto a curto prazo quanto a longo prazo.

2 ESTADO DA ARTE E CONTEXTO CIENTÍFICO

Este capítulo apresenta conceitos e notações utilizados no restante da Dissertação e alguns trabalhos relevantes encontrados na literatura sobre os tópicos abordados no Cap. 1. O objetivo desta parte do trabalho, portanto, é definir e apresentar como a noção de tarefa gerada por um algoritmo *fork/join* recursivo é discutida e implementada por produções relevantes para a área. A notação apresentada será sistematicamente empregada nos Cap. 3 e 4.

2.1 Conceitos Fundamentais e Notações

O modelo de programação *fork/join* recursivo, conforme observado na Introdução (Cap. 1), é a abordagem natural para a implementação de algoritmos recursivos paralelos. Em especial, é uma excelente abordagem para paralelizar algoritmos D&C recursivos. Considere-se, primeiro, o Alg. 2.1 (função DCRec), aplicado sobre uma entrada E com n elementos ($E^{(n)}$) que ilustra o D&C recursivo considerado. Neste algoritmo, AVAL é uma função que pode ou não modificar a entrada $E^{(n)}$ e determina a condição de parada da recursão. DIVIDIR é uma função que divide a entrada $E^{(n)}$ em δ segmentos de tamanho n/a, retornando o conjunto D composto pelos segmentos originados dessa divisão. Por fim, CONQUISTAR é uma função que recebe como entrada um conjunto de resultados sub-ótimos retornados pelas chamadas recursivas e os compõem em um resultado ótimo. a e δ não são passados como parâmetro por dois motivos: (1) são valores constantes, não se alterando a cada chamada recursiva da função e (2) na maior parte das implementações seus valores, em geral pequenos, são implícitos no código; ao invés do laço apresentado na linha 4 o programador faz chamadas recursivas "por extenso". (*v.g.*, ordenação por intercalação).

Algoritmo 2.1 DCRec($E^{(n)}$)

Requer: $\delta \in \mathbb{N}_+$ e $a \in \mathbb{N}_+$ definidos pelo programador no código fonte.

1: **função** DCRec($E^{(n)}$)
2: AVAL($E^{(n)}$) ▷ Inclui condição de parada da recursão.
3: $D \leftarrow$ DIVIDIR($E^{(n)}, a, \delta$) ▷ A partir daqui, $D = \{E_1^{(n/a)}, \cdots, E_\delta^{(n/a)}\}$
4: **para** $i \leftarrow 1$ **até** δ **faça**
5: $\phi_i \leftarrow$ DCRec($E_i^{(n/a)}$) ▷ $E_i^{(n/a)} \in D$
6: **retorne** CONQUISTAR($\{\phi_1, \ldots, \phi_\delta\}$)

A Fig. 2.1 apresenta um grafo da execução do Alg. 2.1 com $\delta = a = 3$ e introduz a notação empregada. Uma árvore de execução D&C recursiva é notada como H, tem N nós, altura h e grau δ ($\delta \geq 2$). Na Fig. 2.1 cada nó (círculo) representa uma chamada de DCRec, indicando o tamanho da entrada passada como parâmetro àquela invocação em relação ao tamanho original n. A condição de parada da recursão é satisfeita nos nós retangulares, onde o tamanho da entrada é $n/3^k$ ($k \in \mathbb{N}$), com k arbitrado pelo programador. Ao atingir um nó retangular o algoritmo encontra um caso básico, resolvendo-o iterativamente.

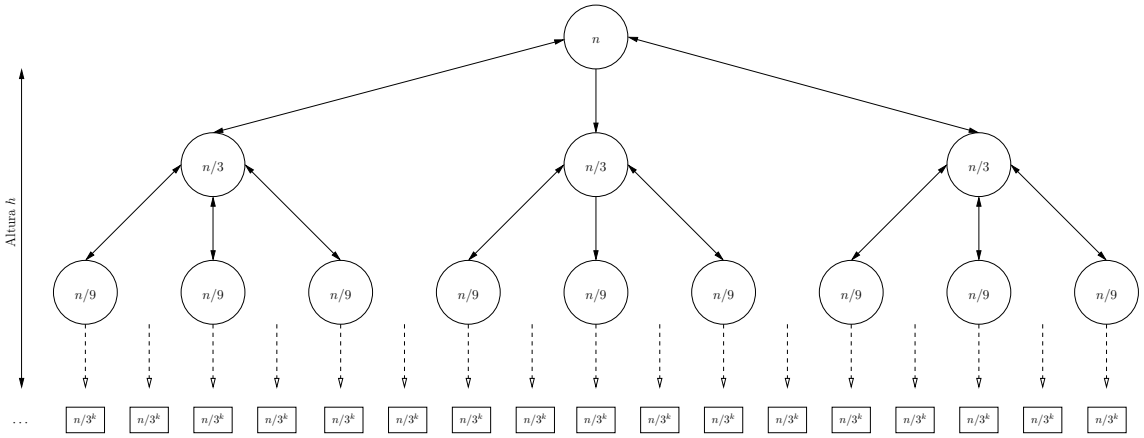


Figura 2.1: Exemplo de árvore de execução para um algoritmo D&C recursivo. Uma árvore H têm altura (distância da raiz às folhas) h, N nós (representados por círculos quando são nós intermediários e como retângulos em nós folha) e grau máximo (número máximo de filhos que um nó intermediário pode possuir) δ. n é o tamanho inicial da entrada (na raiz) e a é um fator de divisão sobre a entrada n passada na chamada recursiva das funções. Nesse exemplo $\delta = a = 3$. A recursão cessa quando a entrada atinge tamanho $n/3^k$, onde $k \in \mathbb{N}$ é arbitrado pelo programador.

Este trabalho visa otimizar o desempenho de um programa nos moldes do Alg. 2.1 ao fazer as chamadas recursivas de função em paralelo. Neste caso é imperativo que a complexidade temporal seja limitada pelo tamanho da árvore de recursão e não pelas funções AVAL, DIVIDIR e CONQUISTAR (*i.e.*, que o trabalho esteja centrado na árvore de recursão). Em termos quantitativos, considera-se a equação de recorrência apresentada em (CORMEN et al., 2001), que descreve o número de operações realizadas por um algoritmo D&C recursivo (utilizada de maneira a concordar com as notações da Fig. 2.1):

$$R(E^{(n)}) = \delta R(E^{(n/a)}) + d(E^{(n)}, a, \delta) + c(\{\phi_1, \ldots, \phi_\delta\}) \tag{2.1}$$

onde

$R(E^{(n)})$	=	complexidade temporal de DCRec$(E^{(n)})$
$d(E^{(n)}, a, \delta)$	=	complexidade temporal de DIVIDIR$(E^{(n)}, a, \delta)$
$c(\{\phi_1, \ldots, \phi_\delta\})$	=	complexidade temporal de CONQUISTAR$(\{\phi_1, \ldots, \phi_\delta\})$

Tomando-se a Equ. (2.1), "o trabalho estar concentrado na árvore de recursão" significa que $\delta R(E^{(n/a)}) > d(E^{(n)}, a, \delta) + c(\{\phi_1, \ldots, \phi_\delta\})$, ou, de uma maneira mais genérica, $d + c \in \mathrm{o}(\delta R)$. Como AVAL testa uma condição de parada, requer-se

que sua complexidade seja O(1). Isso significa que, embora AVAL possa realizar outras operações sobre E, a complexidade dessas operações não pode depender do tamanho de E. Na prática, busca-se garantir que a complexidade de AVAL$(E^{(n)})$ seja $\ll d(E^{(n)}, a, \delta) + c(\{\phi_1, \ldots, \phi_\delta\})$

Uma vez que tem-se um algoritmo D&C recursivo como o Alg. 2.1, que cumpra as restrições acima sobre a complexidade temporal, pode-se inserir paralelismo através das primitivas *fork/join*. O Alg. 2.1 (DCREC) modificado é mostrado no Alg. 2.2 (DCRECFJ).

Algoritmo 2.2 DCRECFJ$(E^{(n)})$

Requer: $\delta \in \mathbb{N}_+$ e $a \in \mathbb{N}_+$ definidos pelo programador no código fonte.

1: **função** DCRECFJ$(E^{(n)})$
2: AVAL$(E^{(n)})$
3: $D \leftarrow$ DIVIDIR$(E^{(n)}, a, \delta)$
4: **para** $i \leftarrow 1$ **até** δ **faça**
5: $\phi_i \leftarrow$ *fork* DCRECFJ$(E_i^{(n/a)})$ ▷ *fork*; a função executa em paralelo.
6: *join* ▷ Bloqueia até que todos os ϕ_i estejam aptos.
7: **retorne** CONQUISTAR$(\{\phi_1, \ldots, \phi_\delta\})$

A semântica de *fork* é diferente da primitiva UNIX `fork`; enquanto *fork* faz com que a chamada de função posterior execute em paralelo, `fork` faz com que um novo processo, idêntico ao anterior, seja criado e a execução seja retomada do ponto de invocação, retornando um número inteiro que serve para que o programa possa testar se ele mesmo é o processo original ou é o processo recém criado. No entanto, o comportamento de *fork* pode ser facilmente replicada usando o `fork` UNIX, modificando-se levemente a semântica do Alg. 2.2.

Para analisar o comportamento paralelo do algoritmo, considere-se uma máquina que possui um conjunto $P = \{p_1, \ldots, p_{|P|}\}$ de processadores homogêneos onde cada núcleo acessa a memória com a mesma latência esperada, configurando uma máquina com *Uniform Memory Access* (UMA). (Como um abuso de notação, há vezes, ao longo do trabalho, em que P será usado onde o correto seria $|P|$, que denota o número total de processadores na máquina. Isso é feito para limpeza visual e apenas quando o seu uso é inequívoco.)

Considerando a Fig. 2.1 (árvore de execução para um algoritmo D&C) e o uso de *fork/join*, cada nó passa a ser uma tarefa paralela. Em termos do escalonamento dessas tarefas nos P processadores a situação ideal é quando $P > N$, porque cada nó (tarefa) pode executar em paralelo com qualquer outro nó, desde que não hajam dependências. Nesse caso, o escalonamento de cada tarefa em cada um dos P processadores é o mais eficiente e simples possível. Na prática, no entanto, frequentemente $P \ll N$, dado o custo monetário de um recurso computacional (em comparação ao custo praticamente nulo de um programador invocar primitivas para a criação dinâmica de tarefas). Nesse caso, precisa-se otimizar a noção de escalonamento. Para isso usam-se notações clássicas.

Conforme mencionado no Cap 1, um escalonamento é a definição de que tarefas executam em quais recursos em um dado momento. Essa definição é escrita como um algoritmo de decisão e será notada por $\mathcal{A}$ (BRUCKER, 2001). Com isso em mente, precisa-se de uma notação para designar o custo que utilizar $\mathcal{A}$ produz. O

tempo que a execução de um algoritmo D&C (árvore) H leva para executar usando *fork/join* com escalonamento $\mathcal{A}$ é notado por $T[\mathcal{A}](H)$.

Considera-se a árvore de execução (agora) paralela da Fig. 2.1. Uma execução sequencial do algoritmo demora um tempo $T[H] = N$, pois exatamente N nós são processados sequencialmente. No contexto paralelo, por outro lado, o trabalho de cada processador $p \in P$, ao processar um nó, é expandi-lo (criar sub-tarefas) até que nós folha sejam encontrados ou até que um limite pré-estabelecido seja atingido. Neste contexto, um $\mathcal{A}$ ideal (chamado *min*) produz um tempo de execução para P processadores $T_P[min](H) = \lceil N/P \rceil$ (WU; KUNG, 1991).

Dada a notação para expressar o custo e um limite inferior para um escalonamento $\mathcal{A}$, em termos de complexidade temporal, apresenta-se uma notação para expressar a quantidade de comunicação de uma computação paralela com escalonamento $\mathcal{A}$. Entende-se quantidade de comunicação em um algoritmo *fork/join* recursivo como o número total de ocorrências de nós cruzados na computação (WU; KUNG, 1991). Um *nó cruzado* é um nó que é gerado em um processador p_a, mas é expandido em um processador p_b, tal que $a \neq b$. Nota-se $M_P[\mathcal{A}](H)$ o número de nós cruzados produzidos por um escalonamento $\mathcal{A}$ sobre uma árvore D&C H com P processadores. De acordo com (WU; KUNG, 1991), o limite inferior para a quantidade de comunicação de uma árvore D&C H com P processadores é $M_P[min](H) = P\delta h$.

2.1.1 *Lazy Task Creation*

O Alg. 2.2 faz com que o processador executando DCRecFJ($E^{(n)}$) bloqueie à espera de resultados quando chama a primitiva *join*. *A priori* esse fato não é um problema; pode-se supor que uma tarefa bloqueada é enfileirada e outra tarefa apta executa no processador. No entanto, esse enfileiramento deve ser garantido na prática, o que é um obstáculo em um cenário de memória distribuída, devido ao seu custo proibitivo em termos de *overhead*. Uma variante não bloqueante é apresentada no Alg. 2.3, usando a noção de *lazy task creation* vista em (MOHR; KRANZ; R. H. HALSTEAD, 1990); ao invés de δ chamadas de DCRecFJ($E^{(n)}$) em paralelo, faz-se $\delta - 1$ chamadas em paralelo e executa-se a última chamada sequencialmente, eliminando a necessidade do enfileiramento da tarefa bloqueada e potencialmente contribuindo para um ganho de desempenho, pois a primitiva *join* só fará a tarefa bloquear caso as outras requisições não tenham sido atendidas no intervalo de tempo da chamada sequencial.

Algoritmo 2.3 DCRecFJnb($E^{(n)}$)

Requer: $\delta \in \mathbb{N}_+$ e $a \in \mathbb{N}_+$ definidos pelo programador no código fonte.

```
1: função DCRecFJnb(E^(n))
2:     aval(E^(n))
3:     D ← dividir(E^(n), a, δ)
4:     para i ← 1 até δ − 1 faça
5:         φ_i ← fork DCRecFJnb(E_i^(n/a))
6:     φ_δ ← DCRecFJnb(E_i^(n/a))        ▷ Prossegue em H antes de bloquear.
7:     join
8:     retorne conquistar({φ_1, ..., φ_δ})
```

Para o restante do trabalho serão considerados os dois cenários (bloqueante e

não-bloqueante). Quando não mencionado o cenário padrão será o bloqueante.

2.1.2 *Threshold*

Em virtude do custo de comunicações em agregados de computadores ser alto, é comum que se utilize a técnica de *degradação sequencial* em algoritmos paralelos (PACHECO, 1997); em um dado momento da recursão –em geral quando a entrada é pequena o suficiente– as chamadas recursivas em paralelo são substituídas por chamadas sequenciais, pois o custo de comunicação associado é maior do que o custo de processamento da tarefa em si.

Com relação ao tamanho da entrada, o limite que que determina o fim da execução paralela é chamado de *threshold* (*e.g.*, PARHAMI 1996). O uso adequado do *threshold* permite o balanceamento entre a melhora do tempo de execução e a execução de entradas com tamanhos que excedem a memória física de um nó.

2.1.3 Notação para Algoritmos de Escalonamento

A notação usada para expressar algum algoritmo de escalonamento $\mathcal{A}$ é próxima à apresentada em (TEL, 2001) para algoritmos distribuídos, cuja principal estrutura, a condição, é explicada a seguir. Considera-se que $\mathcal{A}$ é replicado em todos os processadores que compõem a execução.

Uma *condição* $\mathbf{C}_i : \{\langle predicados \rangle\}$ é habilitada a qualquer instante da execução de $\mathcal{A}$ caso seus *predicados* se cumpram. Um predicado é um teste intrínseco à declaração de uma condição. Um exemplo do uso de condições pode ser visto na Fig. 2.4 que implementa um algoritmo de envio de *token* em anel. Nesse exemplo o envio do *token* é feito explicitamente pelo uso de SEND enquanto o recebimento e tratamento é feito implicitamente, pelo uso de condições. No algoritmo cada processador p_i tem um identificador $id = i$.

Caso alguma condição seja habilitada enquanto outra já está executando, as condições habilitadas são enfileiradas por ordem temporal e a primeira da fila executa apenas quando a atual acabar sua execução. Mesmo que várias condições sejam satisfeitas ao mesmo tempo e de uma só vez, apenas uma (escolhida aleatoriamente) é executada, sendo as outras enfileiradas e executadas *a posteriori*. O critério de enfileiramento, nesse caso, é aleatório.

Algoritmo 2.4 TOKENEMANEL()

Requer: Cada processador p_i com um identificador $id = i$ usado como endereço para envio e recebimento de mensagens.

1: $\mathbf{C}_1$: { $id = 1$ } ▷ Inicia; lança o *token*.
2: SEND($token, 2$)
3: $\mathbf{C}_2$: { Recebe *token* de $id - 1$ }
4: **se** $id = |P|$ **então**
5: SEND($token, 1$)
6: **senão**
7: SEND($token, id + 1$)
8: $\mathbf{C}_3$: { Recebe *token* de $|P|$ } ▷ Término; p_1 recebe o *token*.
9: **fim**

2.1.4 Árvore Paralela de Pesquisa

O conceito apresentado de tratar um programa recursivo paralelo como a distribuição dos nós da árvore de execução entre P processadores é formalizado por (WU; KUNG, 1991) e (WU, 1991). A porção final da discussão sobre como representar algoritmos de escalonamento de tarefas consiste na sistematização desse conceito, de modo a apresentar as primitivas básicas de manipulação das tarefas paralelas nesses algoritmos.

Conforme visto em (PACHECO, 1997) a implementação da distribuição dos nós de uma árvore de execução entre os processadores disponíveis é chamada de Árvore Paralela de Pesquisa (APP).

Uma APP é projetada para correção *on-line* do desbalanceamento produzido pela paralelização do tipo Mestre-Escravo (MS). Em paralelizações MS (implementadas com *round-robin*) alguns processadores param a computação após o processamento de sub-árvores pequenas mesmo que outros estejam sobrecarregados com sub-árvores maiores. O objetivo da implementação de uma APP é minimizar o *makespan*, que é o tempo máximo que um trabalho (programa, algoritmo, conjunto de tarefas paralelas) leva para ser executado.

No modelo APP, empregado no restante deste trabalho, todos os processadores têm uma lista de dados independentes a serem processados (tarefas paralelas), potencialmente fora de ordem. Para uma máquina MIMD multi-computador com P processadores o modelo consiste em:

1. Um processador $p_i \in P$ $(1 \leq i \leq |P|)$ recebe o nó raiz da árvore (tarefa) e a expande até um número δ de nós (sub-tarefas) usando *busca em largura*. Então, os nós expandidos são distribuídos entre os outros P_i processadores $(P_i = \{p_1, ..., p_P\} - \{p_i\})$ e cada processador $p_j \in P_i$ expande as sub-tarefas, fazendo busca em profundidade. p_i reserva algumas tarefas para si e, após distribuir as demais sub-tarefas geradas pela raiz, começa, também, a realizar busca em profundidade.

2. A busca em profundidade local (em cada p_j) continua até que todas as sub-árvores do processador sejam expandidas ou até que um limite de profundidade seja atingido.

3. Quando uma tarefa é concluída, p_j, antes de executar a próxima tarefa, atende a todas as requisições que tiver recebido neste meio-tempo com uma fração de sua lista ou uma mensagem de que não há trabalho a ser entregue.

4. Quando a lista está vazia, o processador p_j solicita mais tarefas de outro processador, recebendo mais trabalho ou uma mensagem de que não há mais trabalho a ser feito.

5. Quando uma das δ tarefas iniciais é concluída, o processador p_j, que a concluiu, manda o resultado a p_i. Quando p_i recebe δ resultados, os combina e a computação para.

Nesse contexto, a pergunta que resta é:

> "*Na etapa (4), como p_j escolhe o processador-alvo para solicitar novas tarefas?*"

A resposta para essa pergunta é a essência do algoritmo de escalonamento; uma boa escolha de p_j resulta em carga de trabalho balanceada, implicando em baixa comunicação e consequente aumento de performance. O inverso também é verdade; *e.g.*, critérios de escolha baseados em *round-robin*, apesar de não possuírem um custo de comunicação elevado quando comparados a algoritmos distribuídos (não existem verificações de estado global do algoritmo), apresentam um desbalanceamento de carga que tende a prejudicar na otimização do *makespan*.

As notações que seguem representam os elementos de uma APP ainda não definidos.

Uma tarefa é notada por Γ. A lista de tarefas de um processador p é notada por $Q[p]$. Quando uma tarefa Γ está sendo processada em (não necessariamente sendo consumida) um processador p nota-se $\Gamma[p]$. $bottom[Q[p]]$ denota o primeiro elemento de $Q[p]$ enquanto $top[Q[p]]$ denota o último. $Q[p]$ é um conjunto ordenado e $Q[p] = \emptyset$ representa a ausência de tarefas na lista do processador p.

As operações suportadas sobre $Q[p]$ são $\text{PUSH}(top[Q[p]], \Gamma)$ que coloca a tarefa Γ no topo de $Q[p]$ (ou no fundo, caso o primeiro parâmetro seja $bottom[Q[p]]$) e não retorna valores; e $\text{POP}(top[Q[p]])$ remove a tarefa que está no topo de $Q[p]$ (ou no fundo, caso o primeiro parâmetro seja $bottom[Q[p]]$) e a retorna.

Por fim, duas funções auxiliares são úteis na representação dos cálculos sobre escalonamentos. $\text{EXEC}(\Gamma)$ faz com que o processador onde a primitiva foi disparada passe a executar a tarefa Γ no seu próximo ciclo de execução (não retorna valores) e $\text{SORTEIO}(C)$ que para um conjunto qualquer C retorna um de seus elementos, escolhidos aleatoriamente.

Devido à ampla possibilidade de implementação para uma tarefa paralela escolheu-se dividir o restante do capítulo em três seções principais:

1. Seção 2.2, que trata de tarefas paralelas implementadas como *threads* escalonadas em processos e processadores *multi-core*.;

2. Seção 2.3, que trata de tarefas paralelas implementadas como processos escalonadas em nós de um *cluster*; e

3. Seção 2.4, que trata de sistemas que podem escolher implementar suas tarefas como processos ou *threads* em tempo de compilação e/ou execução.

Ainda que as implementações MPI, por trabalharem com processos, favoreçam amplamente uma abordagem que siga os tópicos no item (2), o trabalho com tarefas implementadas como *threads* fornece uma variedade de algoritmos de escalonamento que, feitas as devidas adaptações, formam a base dos algoritmos RatMD e RtMPD, apresentados nos Capítulos 3 e 4. Também é relevante considerar que os tópicos vistos no item (3), mesmo que não sirvam diretamente à construção dos algoritmos apresentados, são úteis em relação ao projeto de longo prazo deste trabalho, conforme detalhado no Capítulo 6, mais precisamente na Subseção 6.3.2.

2.2 Escalonamento de *Threads* em Processos

Esta seção mostra alguns trabalhos que implementam tarefas paralelas como *threads* de sistema operacional. Logo, Γ é uma *thread* em uma máquina MIMD do tipo *Simultaneous Multi Processor* (SMP) (*multi-core*) *monothread* (um *core* executa

exatamente uma *thread* por vez) com um conjunto $P = \{p_1, \ldots, p_{|P|}\}$ de processadores homogêneos onde cada núcleo acessa a memória com a mesma latência esperada, configurando uma máquina com *Uniform Memory Access* (UMA)(NAVAUX; DEROSE, 2003).

O objetivo dos trabalhos apresentados é especificar um escalonador de *threads* dinâmico (CASAVANT; KUHL, 1994). Ainda, é requerido que este escalonador seja distribuído; soluções centralizadas são fracamente escaláveis e não são adequadas para sistemas paralelos, por criarem gargalos (PACHECO, 1997). A definição mais precisa do que se considera uma *thread* é vista na próxima subseção.

2.2.1 Roubo Aleatório de Tarefas

O escalonamento baseado em roubo de tarefas é um algoritmo distribuído clássico onde os processadores ociosos são quem roubam tarefas dos processadores que têm trabalho a ser feito. Esse modelo contrasta com a abordagem *work-pushing*, onde os processadores que criam novas tarefas são quem determinam onde estas novas tarefas executarão. *Work-stealing* remonta a (BURTON; SLEEP, 1981), (R. H. HALSTEAD, 1984). Outros trabalhos evidenciaram se tratar de uma abordagem mais eficiente que *work-pushing*, *e.g.*, (BLUMOFE; LEISERSON, 1994), (FELDMANN; MYSLIWIETZ; MONIEN, 1991), (JOERG; HALBHERR; ZHOU, 1994), (LEISERSON; KUSZMAUL, 1994), (MOHR; KRANZ; R. H. HALSTEAD, 1990).

Em (WU; KUNG, 1991) propõe-se um algoritmo de concentração de *tokens* para escalonamento de tarefas do tipo D&C; cada processador recebe e repassa esses *tokens*, de modo que um processador, ao ficar sem tarefas, tenta roubá-las de processadores que tenham mais *tokens*. Em (WU, 1991) o algoritmo é modificado para realizar escalonamento eficiente quando a interconexão física não é do tipo "*all-to-all*", mas sim do tipo *mesh* k-dimensional, hipercubo e *perfect shuffle* (*i.e.*, múltiplos *switches* para conectar processadores em $\log_2 n$ níveis e *switches*/nível). Embora menos eficientes que o algoritmo original, essas variações são ótimas no que se considera a variação na camada de *hardware*. Ainda assim o critério baseado em *tokens* tende a introduzir um *overhead* considerável devido ao gerenciamento de *tokens*.

(BLUMOFE; LEISERSON, 1994) propõe um escalonador de *threads* baseado em roubo de tarefas (*Work Stealing* – WS), que é apresentado a seguir. A grande vantagem deste sobre o algoritmo que manipula *tokens* descrito acima é que este usa um critério aleatório na escolha das tarefas a serem roubadas; espera-se que, em média, a complexidade de gerenciamento desses roubos seja $\Theta(1)$. Esse escalonador é a base para a linguagem Cilk, apresentada mais à frente.

O modelo proposto em (BLUMOFE; LEISERSON, 1999) baseia-se em DAGs para definir uma computação *multithread*: uma *thread* Γ é um ordenamento sequencial de tarefas que executam em uma unidade de tempo ("instrução" v) em um espaço de memória nomeado *frame de ativação*. Uma *Computação Multithread* é, então, uma execução dessas tarefas, onde é permitido criar novas threads ("*spawn*") e sincronizar sua execução com outras *threads* já executando ("*sync*"). As operações *spawn* e *sync* tem o mesmo significado que as operações *fork* e *join* apresentadas anteriormente.

O uso de *spawn* e *sync* permitem definir os estados que uma *thread* Γ pode assumir em termos destas primitivas. Uma *thread* criada por um *spawn* está *viva*. Quando a última instrução da *thread* executa, ela suspende sua execução, quando se diz que a *thread morre*. Uma *thread* pode, ainda, estar *bloqueada*, caso em que

bloqueia até que outra *thread* realiza a operação de *sync* sobre esta.

Os conceitos introduzidos podem ser observados no DAG da Fig. 2.2. Essa figura se compara com a Fig. 2.1 (árvore para D&C) pois cada *frame* de ativação da primeira corresponde um nó da segunda. Ambas diferem pois o DAG apresentado na Fig. 2.2 é mais geral, não se limitando à condições fixas para paralelização com *fork/join*, *i.e.*, δ não é constante entre os nós.

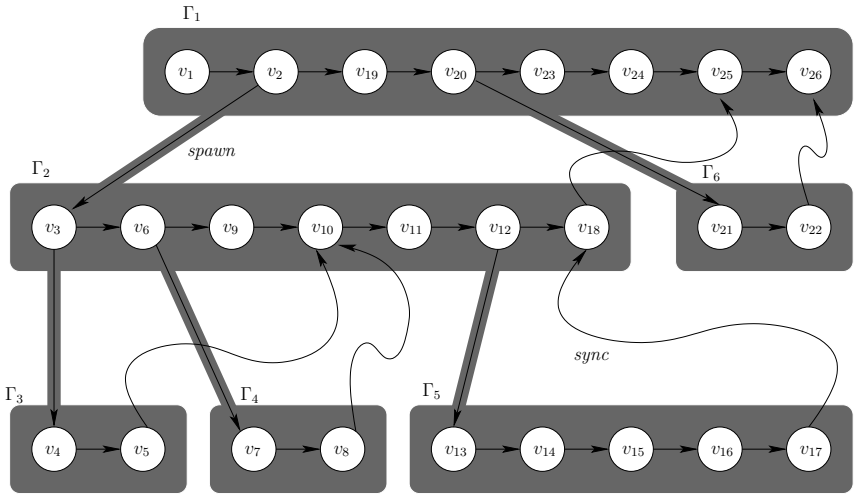


Figura 2.2: (BLUMOFE; LEISERSON, 1994) DAG para computação *multithread*. Cada instrução v_i é representada por um círculo e cada dependência sequencial é representada por uma seta normal, num estilo produtor-consumidor. Setas hachuradas representam uma operação de *spawn* e setas curvilíneas representam operações *sync*. Uma instrução só pode ser executada quando todas as suas dependências são satisfeitas, *i.e.*, quando todas as instruções que originam as setas que chegam até a instrução atual já foram executadas. Cada *frame* de ativação é representado por um retângulo de cantos arredondados e uma *thread* Γ_i é o retângulo anterior preenchido com instruções e dependências entre estas instruções.

De acordo com (BLUMOFE; LEISERSON, 1999) para que uma execução seja considerada válida, algumas restrições devem ser impostas ao modelo, em referência à Fig. 2.2:

1. Cada instrução têm, no máximo, um número constante de dependências a serem satisfeitas (setas que chegam); isso garante um modelo em que cada instrução leva um múltiplo de unidades básicas de tempo para executar.

2. Todas as operações de *sync* de uma thread Γ_i são feitas sobre seu pai Γ_{i-1} (execução *totalmente estrita*). Isso significa que uma sub-rotina não pode ser executada até que todos os seus argumentos estejam disponíveis, embora esses possam ser calculados em paralelo.

O item (1) equivale a dizer que o tamanho do conjunto Φ do Alg. 2.2 (algoritmo D&C com *fork/ join* recursivo) é limitado por uma constante.

Utiliza-se a mesma notação definida na seção anterior no que diz respeito ao tempo de execução com P processadores. Para fins de clareza nomeia-se a árvore H de DAGMT, uma vez que se trabalha com um DAG gerado por uma computação *multithread*, que é mais geral que o modelo de árvore D&C já definido para H. Nesse

mesmo contexto, $S_P[\mathcal{A}](\text{DAGMT})$ é o espaço (soma dos tamanhos de todos os *frames* de ativação) consumido por uma computação *multithread* com P processadores usando o algoritmo $\mathcal{A}$.

Como, nesse cenário, δ não é um número constante, a semântica da altura h da árvore D&C ilustrada na Fig. 2.1 como o número mínimo de passos sequenciais de H, se perde. Logo, introduz-se a notação empregada por (JAJA, 1992), que possui a mesma semântica que h para um DAG como o da Fig. 2.2: $T_\infty(\text{DAGMT})$ denota o tempo de execução de um DAG *multithread* com um número de processadores tão grande quanto se queira. $T_\infty(\text{DAGMT})$ também é chamado de *profundidade* do DAG ou *caminho crítico*.

Da mesma maneira que no parágrafo anterior, utiliza-se corriqueiramente $T_1(\text{DAGMT})$ como o tempo da execução com um processador ou (carga total de) *trabalho* em uma computação *multithread*. Tempo sequencial já apresentado, $T(\text{DAGMT})$, não é utilizado, pois, de acordo com (JAJA, 1992), o algoritmo sequencial pode diferir da versão com um processador do algoritmo paralelo, o que elimina a semântica desejada de "carga total de trabalho a ser distribuída". Esse raciocínio é análogo para $S_1(\text{DAGMT})$.

Nesse cenário, o algoritmo Roubo Aleatório de Tarefas (RAT) é um algoritmo de escalonamento guloso (BRENT, 1974) descrito no Alg. 2.5.

Algoritmo 2.5 RAT()

```
 1: C_0: { p é o processador inicial. }
 2:     PUSH(top[Q[p]], Γ_init)        ▷ Γ_init (tarefa inicial) em apenas um processador.
 3: C_1: { Γ_1[p] habilita Γ_2[p] }    ▷ Γ_1[p] sempre está executando em p.
 4:     PUSH(top[Q[p]], Γ_2)
 5: C_2: { Γ_1[p] faz spawn de Γ_2[p] }
 6:     PUSH(top[Q[p]], Γ_1)
 7:     EXEC(p, Γ_2)
 8: C_3: { Γ[p] morre ou bloqueia }
 9:     se Q[p] = ∅ então
10:         repita
11:             p' ← SORTEIO(P)
12:         até Q[p'] ≠ ∅
13:         EXEC(POP(top[Q[p']]))      ▷ Aqui acontece o roubo de uma tarefa remota.
14:     senão
15:         EXEC(POP(bottom[Q[p]]))
```

Dado o fato que a lista de tarefas Q é manipulada adicionando-se elementos somente no topo e removendo-os do topo ou do fundo, (BLUMOFE; LEISERSON, 1994) utiliza uma estrutura de dados otimizada do tipo *Double Ended Queue* (*deque*).

Os seguintes teoremas versam sobre a ocupação em espaço e tempo de uma execução do RAT. Eles são apresentados em (BLUMOFE; LEISERSON, 1999).

Teorema 2.1 (Complexidade Espacial do RAT)

$$S_P[\text{DAGMT}](\text{RAT}) \leq P \times S_1(\text{DAGMT}). \tag{2.2}$$

Prova. Cada processador pode, no máximo, ocupar um espaço equivalente a $S_1(\text{DAGMT})$, por definição. Dessa maneira, no pior cenário, todos ocupam a sua capacidade máxima e tem-se um espaço de $P \times S_1(\text{DAGMT})$. □

Este teorema mostra que a memória consumida é proporcional somente ao tamanho da entrada, limitando a alocação de memória dinâmica intrínseca a um *spawn* à carga de trabalho total.

As provas dos Teoremas 2.2 e 2.3 são omitidas, pois fogem ao escopo desse trabalho.

Teorema 2.2 (Complexidade Temporal do RAT)

$$T_P[\text{RAT}](\text{DAGMT}) \in \text{O}\left(\frac{T_1(\text{DAGMT})}{P} + T_\infty(\text{DAGMT})\right) \tag{2.3}$$

é o tempo médio esperado de execução de RAT *sobre um DAG multithread* DAGMT.

Como o limite inferior para o tempo de execução de DAGMT é $T_1(\text{DAGMT})/P$, o Teo. 2.2 enuncia que o *overhead* introduzido por RAT é, em média, $T_\infty(\text{DAGMT})$. Como, em uma árvore balanceada, esse valor é igual a h, pode-se esperar um *overhead* que atinja no máximo um valor logarítmico, em função da entrada. Esse resultado é bastante significativo quando comparado ao limite inferior do problema.

O teorema seguinte estabelece um limite para a comunicação empregada no RAT.

Teorema 2.3 (Comunicação do RAT)

$$M_P[\text{DAGMT}](\text{RAT}) \in \text{O}\left(P \times T_\infty(\text{DAGMT}) \times S_{max}(\text{DAGMT})\right) \tag{2.4}$$

onde S_{max} é o tamanho do maior frame de ativação na computação.

Ao analisar-se o caso de computações do tipo *fork/join*, um caso específico de paralelismo de tarefas no modelo estrito, observa-se que a quantidade de comunicação empregada é ótima em $\Omega(PT_\infty(\text{DAGMT})S_{max}(\text{DAGMT}))$ para qualquer escalonamento, conforme teorema de (WU; KUNG, 1991). O Teo. 2.3 confirma este limite. Além disso, o número de roubos esperado por processador é $\Theta(T_\infty(\text{DAGMT}))$; se $T_\infty(\text{DAGMT})$ é pequeno, o que se espera, ocorrerão poucos roubos, minimizando o *overhead*. De fato, o limite inferior apresentado anteriormente para $M_P[\mathcal{A}](H)$, $P\delta h$ equivale, em um DAG *multithread*, a $PT_\infty S_{max}$; em um algoritmo D&C, $T_\infty(H) = h$ e $S_{max}(H) = \delta$.

O algoritmo atinge *speedup* linear quando $P = T_1(\text{DAGMT})/T_\infty(\text{DAGMT})$, possuindo comunicação no limite de $\text{O}(T_1(\text{DAGMT}) \times S_{max}(\text{DAGMT}))$. Como, geralmente, $P \ll T_1(\text{DAGMT})/T_\infty(\text{DAGMT})$, a comunicação total fica bem abaixo de $T_1(\text{DAGMT}) \times S_{max}(\text{DAGMT})$, que é a comunicação mínima necessária a algoritmos do tipo *work-pushing* (onde quem determina em que processador uma nova tarefa executará é o processador que a criou).

As próximas subseções refinam o modelo, tratando de duas extensões: o uso de processadores heterogêneos na computação e a introdução de escalonamento de processos.

2.2.2 Processadores Heterogêneos

O problema de se fazer um escalonamento distribuído e eficiente em um ambiente de processadores heterogêneos, via RAT, foi abordado por (BENDER; RABIN, 2000). O modelo anterior é estendido para um cenário onde cada processador possui uma estimativa (que pode não ser precisa) de sua própria velocidade.

Não é admitido que os processadores possam ser *arbitrariamente erráticos*, significando que não podem, eventualmente, reduzir tanto sua velocidade que não executem nada em um ciclo de relógio.

Este cenário admite a introdução de um *Escalonador de Máxima Utilização* (EMU); sempre que um processador p_1 com velocidade π_1 está ocioso e um processador p_2 com velocidade π_2 está processando e $\pi_1 > \pi_2$, então o processador p_1 rouba a tarefa de p_2, tornando p_2 ocioso.

EMU é um algoritmo de escalonamento ótimo, mas não se conhecem implementações viáveis, pois é um problema $\mathcal{NP}$-difícil. O algoritmo RAT adaptado a uma máquina onde os processadores têm velocidades variáveis usa um *Escalonador de Alta Utilização* (EAU); sempre que um processador p_1 com velocidade π_1 está ocioso e um processador p_2 com velocidade π_2 está processando e $\pi_1 > \beta\pi_2$ ($\beta \in \mathbb{N}$, constante e definido antes da execução), então o processador p_1 rouba a tarefa de p_2, tornando pi_2 ocioso.

Ao algoritmo RAT adaptado para tornar-se um EAU dá-se o nome de RAT-EAU. RAT-EAU é apresentado no Alg. 2.6. As provas sobre esse algoritmo baseiam-se no uso de um adversário, que pode alterar as velocidades relativas dos processadores como achar conveniente (mas não pode ser arbitrariamente errático). Como realizar um escalonamento ótimo, em qualquer cenário, é $\mathcal{NP}$-difícil, RAT-EAU é um algoritmo sub-ótimo (CASAVANT; KUHL, 1994). Seu objetivo é diminuir o *makespan*.

Deve-se definir duas novas operações e um novo atributo no uso do RAT-EAU. MUG(p) faz com que o processador que invocou a primitiva roube a *thread* que o processador p está executando para si, deixando p ocioso e retornando a *thread* roubada. PROCESSING(p) retorna *falso* se o processador p estiver ocioso ou retorna *verdadeiro*, caso contrário. Por fim, $\pi[p]$ denota a velocidade do processador p.

Sendo o RAT um escalonador guloso, é natural que se aproxime de outros algoritmos que fazem escalonamento guloso em um cenário homogêneo. No entanto, em um cenário de processadores heterogêneos, diferentes escalonamentos gulosos podem ser muito díspares no que se refere ao *makespan*. Para um algoritmo de escalonamento guloso alcançar um bom (*i.e.*, pequeno) *makespan* é necessário que processadores mais rápidos tenham assinalados ramos mais longos do DAG.

Embora o *makespan* teórico do algoritmo EMU possa ser levemente inferior ao *makespan* de algoritmos EAU, é esperado que ocorra bem menos roubos neste que no primeiro, tornando o sistema mais eficiente em comunicação e como um todo. (BENDER; RABIN, 2000) justifica isso através da propriedade de RAT-EAU de, em face a processadores de velocidades homogêneas, comportar-se exatamente como o RAT, que tem um tempo de execução ótimo.

A prova sobre a complexidade espacial de RAT-EAU é equivalente à prova do RAT. Quanto ao desempenho e à comunicação, apresenta-se o seguinte Teorema, com a prova omitida:

Algoritmo 2.6 RAT-EAU()

```
 1: C0: { p é o processador inicial. }
 2:     PUSH(top[Q[p]], Γ_init)        ▷ Γ_init (tarefa inicial) em apenas um processador.
 3: C1: { Γ1[p] habilita Γ2[p] }
 4:     PUSH(top[Q[p]], Γ2)
 5: C2: { Γ1[p] faz spawn de Γ2[p] }
 6:     PUSH(top[Q[p]], Γ1)
 7:     EXEC(p, Γ2)
 8: C3: { Γ[p] morre ou bloqueia ou é migrada para outro processador }
 9:     se Q[p] = ∅ então
10:         l ← falso
11:         repita
12:             p' ← SORTEIO(P)
13:             se Q[p']| ≠ ∅ então
14:                 EXEC(POP(bottom[Q[p']]))
15:                 l ← verdadeiro
16:             senão
17:                 se π[p] ≥ βπ[p'] e PROCESSING(p') então
18:                     EXEC(p, MUG(p'))        ▷ faz a migração de p' para p
19:                     l ← verdadeiro
20:         até l = verdadeiro
21:     senão
22:         EXEC(POP(bottom[Q[p]]))
```

Teorema 2.4 (Complexidade Espacial e Temporal do RAT-EAU)

$$T_P[\text{RAT-EAU}](\text{DAGMT}) \leq \frac{T_1(\text{DAGMT})}{P\pi_{med}} + \text{O}\left(\frac{T_\infty(\text{DAGMT})}{\pi_{med}}\right) \tag{2.5}$$

onde π_{med} é a velocidade média de processamento definida como $\pi_{med} = \sum_{i=1}^{P}(\pi_i/P)$.

O número de tentativas de roubo e migrações (limitante superior da comunicação), nas mesmas condições, é $O(\beta \times T_\infty(\text{DAGMT}) \times P)$.

O Teo 2.4 mostra que a complexidade de RAT-EAU equivale à complexidade do RAT, ponderada pelo *overhead* introduzido pelos roubos originados de processadores com as velocidade díspares por um fator β. Esse tipo de redução de uma prova à outra será aproveitado no Cap. 3, onde é feito o mesmo com um algoritmo RAT com as ordens de empilhamento de tarefas modificada.

2.2.3 Escalonamento de *Threads* em Processos

Os modelos propostos em (BLUMOFE; LEISERSON, 1994), (BLUMOFE; LEISERSON, 1999) e (BENDER; RABIN, 2000) consideram que *threads* são escalonadas diretamente em processadores. No entanto, essa não é a realidade na maioria dos sistemas; em geral, *threads* são escalonadas em processos e um *kernel* escalona esses processos em processadores. (ARORA; BLUMOFE; PLAXTON, 1998) propõe uma extensão aos modelos previamente apresentados para adaptá-los a esse cenário. Vale ressaltar que o *kernel* pode usar qualquer algoritmo/ambiente para o escalonamento dos processos.

Em um cenário onde *threads* são escalonadas em processos não é possível esperar um *speedup* proporcional a P, pois o *kernel* pode alocar menos de P processadores para a computação. Ao invés disso, quer-se um *speedup* proporcional a P_{med}, número médio de processadores disponível para a computação ao longo do tempo.

A primeira modificação necessária é tornar todas as operações sobre estruturas de dados (principalmente as sobre a *deque*) não-bloqueantes. Com isso, se evita o risco do *kernel* desalocar o processo em que uma *thread* realiza operações de exclusão mútua e causar um *deadlock* (LYNCH, 1997) permanente no sistema.

A segunda modificação necessária é que a inclusão da operação de YIELD; permite-se a uma *thread* Γ fazer com que o processo onde está alocada possa abrir mão de executar em algum processador no próximo turno. Um *turno* é o intervalo entre dois escalonamentos consecutivos feitos pelo *kernel* de processos em processadores.

Os teoremas e provas apresentados baseiam-se em mostrar que as propriedades do RAT se mantém mesmo na presença de um *kernel* que se comporta como um adversário (sempre escolhe a pior opção para o escalonador de *threads*), cuja única limitação é ter de admitir a primitiva YIELD. Em cada passo da execução, o *kernel* escolhe qualquer subconjunto dos P processos para executar a instrução correspondente àquele turno.

O algoritmo RATTP que faz escalonamento em dois níveis está descrito no Alg. 2.7. Com relação ao Alg. 2.5 a principal modificação é a inserção de um YIELD entre duas tentativas de roubo seguidas.

Algoritmo 2.7 RATTP()

```
 1: C_0: { p é o processador inicial. }
 2:     PUSH(top[Q[p]], Γ_init)        ▷ Γ_init (tarefa inicial) em apenas um processador.
 3: C_1: { Γ_1[p] habilita Γ_2[p] }    ▷ Γ_1[p] sempre está executando em p.
 4:     PUSH(top[Q[p]], Γ_2)
 5: C_2: { Γ_1[p] faz spawn de Γ_2[p] }
 6:     PUSH(top[Q[p]], Γ_1)
 7:     EXEC(p, Γ_2)
 8: C_3: { Γ[p] morre ou bloqueia }
 9:     se Q[p] = ∅ então
10:         repita
11:             YIELD()        ▷ Este processo não executa a seguir.
12:             p' ← SORTEIO(P)
13:         até Q[p'] ≠ ∅
14:         EXEC(POP(top[Q[p']]))
15:     senão
16:         EXEC(POP(bottom[Q[p]]))
```

Outro ponto importante é tornar os métodos da *deque* não bloqueantes. Como apenas o próprio processador executa o método PUSH($top[Q[p]], \Gamma$), não há necessidade de este execute em exclusão mútua. Resta apenas modificar os métodos POP($top[Q[p]], \Gamma$) e POP($bottom[Q[p]], \Gamma$).

Para qualquer mecanismo de sincronização que se use (*e.g.*, Semáforos, Monitores, Mutex, *etc.*), basta que a *thread*, ao invés do bloquear, aborte imediatamente o roubo caso alguma outra *thread* esteja com modificações na sua *deque* em curso.

Para isso, precisa-se de uma instrução do tipo *test-and-set* e uma variável de controle embutida em cada *deque*; a maioria dos processadores atuais dispõe de tal tipo de instrução, sendo simples a solução desse entrave.

Ainda há o problema de alvo de escalonamento; ainda é permitido ao *kernel* continuamente reescalonar *threads* cuja *deque* está vazia em um processo que é alvo de tentativas de roubo, prolongando *ad infinitum* a execução do programa. Para resolver isso, limita-se a atividade do *kernel* para que ele possa apenas escalonar em *rodadas* ao invés de passos; uma rodada é um conjunto de passos/instruções que varia de $2k$ a $3k$, onde k é um número grande o suficiente (*i.e.*, um limite superior) para executar uma instrução ou completar uma chamada à função $\text{POP}(top[Q[p]], \Gamma)$.

O rigor da aplicação dos métodos vistos até aqui pode variar. Dependendo do tipo de *kernel* considerado mais ou menos medidas devem ser tomadas.

Um *kernel* do tipo *adversário abstraído* pode escolher o número p_i de processos e quais p_i processos são escalonados numa rodada i, mas deve fazer esse escalonamento de maneira *estática*. Um *kernel* do tipo *adversário adaptativo* difere do anterior por poder realizar o escalonamento de maneira *dinâmica*.

(ARORA; BLUMOFE; PLAXTON, 1998) mostra que para um *kernel* do tipo adversário abstraído a primitiva $\text{YIELD}(r)$ (quando chamada pelo processo q na rodada i significa que o *kernel*, ao fim da rodada, não poderá escalonar q novamente até que o processo r seja escalonado), menos rigorosa, é suficiente; pode-se escolher sempre o processo que será alvo da tentativa de roubo, garantindo-se que ele não seja substituído por algum processo que eventualmente possua a *deque* vazia. Como o escalonamento é feito de maneira *off-line*, o escalonador não pode frustrar esta estratégia ao não escalonar q até o fim da execução; todos os processos que tem q como alvo também não mais executarão e, conforme já definido, o *kernel* não pode abrir mão de executar algum processo.

Já para um *kernel* do tipo adversário adaptativo, YIELD deve comportar-se de maneira que quando chamada pelo processo q na rodada i signifique que o *kernel*, ao fim da rodada, não poderá escalonar q novamente até que *todos os outros* processos tenham sido escalonados. Neste caso, (ARORA; BLUMOFE; PLAXTON, 1998) apresenta o seguinte teorema:

Teorema 2.5 (Complexidade Temporal do RatTP) *Se* YIELD *está disponível na sua versão mais rigorosa, então*

$$T_P[\text{RatTP}](\text{DAGMT}) \in \text{O}\left(\frac{T_1(\text{DAGMT})}{P_{med}} + \frac{T_\infty(\text{DAGMT}) \times P}{P_{med}}\right)$$

é o tempo de execução esperado, em média, para qualquer tipo de kernel que se comporte como um adversário.

Apesar de omitida, a prova desse teorema baseia-se no fato de que a chamada de YIELD antes de cada tentativa de roubo força o *kernel* a escalonar todos os outros processos antes de q; caso todos ou muitos processos estejam na fase de roubo, pelo menos um de todos os possíveis devem executar.

2.2.4 Cilk

A linguagem Cilk (FRIGO; LEISERSON; RANDALL, 1998) é uma implementação de um escalonador de *threads* para processadores SMP baseada no Alg. 2.7

(RatTP). Cilk foi criada no *Massachusetts Institute of Technology*, originando vários trabalhos relevantes ao longo dos anos, *e.g.*, (HALBHERR; ZHOU; JOERG, 1994), (JOERG; HALBHERR; ZHOU, 1994), (FENG; LEISERSON, 1997), (BLUMOFE; LEISERSON, 1998), (CHENG et al., 1998), (DAILEY; LEISERSON, 2002), (BENDER; RABIN, 2002), (DANAHER; LEE; LEISERSON, 2005), (AGRAWAL; HE; LEISERSON, 2006a), (AGRAWAL et al., 2006), (AGRAWAL; HE; LEISERSON, 2006b), (AGRAWAL; LEISERSON; SUKHA, 2006), (DANAHER; LEE; LEISERSON, 2006) e (AGRAWAL; FINEMAN; SUKHA, 2008). Além disso, foi a base para a criação de bibliotecas como TBB.

Cilk é implementada como uma modificação da linguagem C, com adição de palavras-chave que expressam paralelismo. A linguagem está na sua versão 5 e é usada, *v.g.*, nos programas que jogam Xadrez: ∗Socrates e Cilkchess.

Se as palavras-chave Cilk forem retiradas de um programa, o que resta deve ser um código C plenamente compilante e válido. Essa variante é chamada de *elision*. Ainda, Cilk é uma linguagem *fiel* a C, pois a semântica do programa permanece funcional na versão *elision*.

A estratégia do Cilk é baseada no *princípio de priorizar o trabalho*, que consiste em sempre tentar mover os *overheads* para fora da carga total de trabalho (T_1(DAGMT)) e para dentro do *caminho crítico* (T_∞[DAGMT]), por que se espera que poucos roubam ocorram e, portanto, que o sistema executará, na maior parte das vezes, sem *overhead* do roubo de tarefas. Uma das principais características dessa decisão de projeto é o uso de dois "clones" para cada função executada como uma *thread*. Um deles é uma versão "rápida", minimamente modificada em relação à versão sequencial, que é executada quando uma tarefa é consumida localmente. O outro, a versão "lenta", munida dos *overheads* de controle para a execução paralela, é invocada sempre que a função é executada remotamente. Desta maneira todos os *overheads* de paralelização contribuem apenas com o caminho crítico (T_∞[DAGMT]) e uma execução sequencial ocorre livre de *overheads* mesmo que utilize o Cilk. Como se espera uma quantidade pequena de roubos ao longo da computação esse princípio tende a contribuir para uma queda no *makespan*.

O compilador Cilk é chamado `cilk2c` e gera um código C válido para ser compilado por outro programa, como o `gcc`, usando a biblioteca Cilk.

As *deques* de cada *thread* são uma zona de exclusão mútua, que é implementada usando-se uma versão levemente modificada do protocolo de exclusão mútua de Dijkstra (DIJKSTRA, 1965) chamado de THE, cujo *overhead* contribui maximamente para o caminho crítico e minimamente para o trabalho realizado.

A Fig. 2.3 mostra um exemplo de código Cilk. Observa-se que o programa seria um programa C comum caso as palavras `cilk`, `spawn` e `sync` fossem removidas.

O emprego do Rat dentro do Cilk é a base para outras implementações populares para escalonamento por roubo de tarefas. *v.g.*, *Intel's Thread Building Blocks* e OpenMP3. O primeiro implementa o mesmo algoritmo sobre uma biblioteca C/C++, enquanto o segundo, ao invés de utilizar um compilador próprio, trabalha como uma extensão de outros compiladores.

A próxima subseção abordará o ferramental técnico necessário para se fazer a migração das *threads* em processos, através de bibliotecas e *frameworks* que permitem a migração de *threads*. Este tipo de ferramental é fundamental quando da aplicação do Alg. 2.7 (RatTP).

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

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

cilk int main(int argc, char *argv[])
{
    int n;
    int result;
    n = atoi(argv[1]);
    result = spawn fib(n);
    sync;
    printf("Resultado: %d\n", result);
    return 0;
}
```

Figura 2.3: (FRIGO; LEISERSON; RANDALL, 1998) Programa Cilk simples para computar o n-ésimo número de Fibonacci em paralelo. A palavra-chave `cilk` identifica `fib` como sendo um *procedimento Cilk*, versão paralela do código em C. A invocação de `spawn` diz que a função que segue, tomados seus argumentos, deve ser executada em paralelo em relação ao resto do código. `x` e `y` não podem ser referenciados novamente até o uso de uma primitiva `sync`, sob pena de estarem inconsistentes; condições de corrida no paralelismo podem afetar o valor das variáveis. No caso, `sync` age como uma sincronização do tipo barreira em relação a todas as threads disparadas no código até o momento.

2.2.5 MigThread

MigThread (JIANG; CHAUDHARY, 2004) é um pacote que provê migração e *checkpointing* de múltiplas threads para diferentes máquinas e sistemas de arquivos simultaneamente.

O princípio básico empregado é implementar um núcleo de *threads* em nível de usuário, permitindo que computações sejam tratadas em um nível mais alto: os estados de um processo são abstraídos para construções presentes na linguagem, ao invés da visão clássica de sistema operacional. Como manipula a pilha/*heap* de um processo em nível do usuário, a *MigThread* independe do sistema de *threads* usado na camada mais de baixo para se comunicar com o sistema operacional (*e.g.*, Pthreads). Pelo mesmo motivo, independe do sistema operacional utilizado. *MigThread* é uma biblioteca para a linguagem C. Um esquema do funcionamento da MigThread é apresentado na Figura 2.4.

A abstração proporcionada pela biblioteca advém da representação de um estado da computação. Para isso, utiliza um esquema de conversão de dados chamado "Recebedor Faz Bem" (RFB), que consegue detectar tipos de dados, gerar marcações, e converter os dados apenas no lado do recebedor da(o) *thread*/processo migrada(o). RFB trata tipos de dados compostos como um bloco indivisível. Como outras ferramentas de migração, a Eficiência (JAJA, 1992) aumenta conforme o volume de dados aumenta.

Um problema decorrente de implementação é que linguagens fracamente tipadas oferecem um complicador, visto que uma variável pode conter um dado de qualquer tamanho. *MigThread* considera como fatores de insegurança advindos desse fato *casting* de ponteiros, ponteiros em *unions*, chamadas a bibliotecas de terceiros e

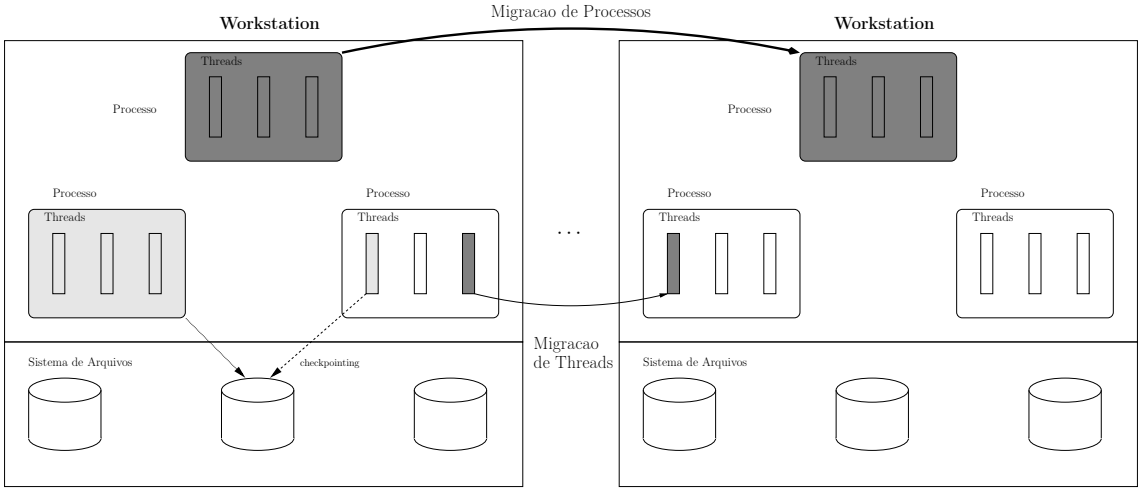


Figura 2.4: (JIANG; CHAUDHARY, 2004) Migração/ *checkpointing* de processos/*thread* em *MigThread*.

conversão de dados não-compatível. O trabalho apresentado é capaz de lidar com essas limitações (JIANG; CHAUDHARY, 2004).

Pode-se rodar a biblioteca em uma grade de computadores (NAVAUX; DEROSE, 2003). *MigThread* consiste de duas partes: um pré-processador e um módulo de suporte em tempo de execução, diferente do Cilk (Subseção 2.2.4), que possui apenas o primeiro. O pré-processador é encarregado de transformar o código-fonte do usuário em código entendível para o módulo de suporte.

Uma visão geral do processo pode ser vista na Figura 2.5.

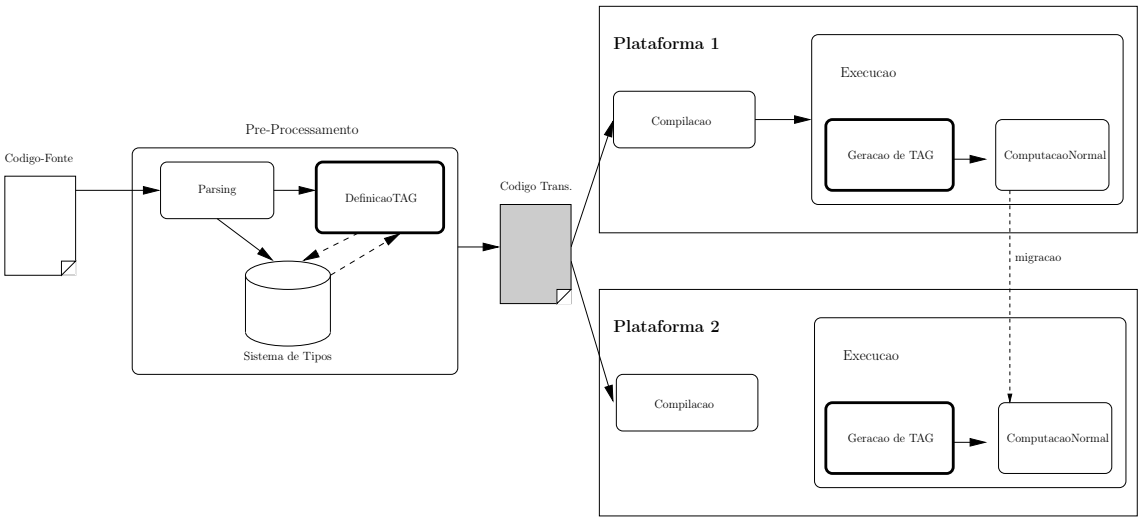


Figura 2.5: (JIANG; CHAUDHARY, 2004) Funcionamento dos componentes da *MigThread*.

2.2.6 Nomadic Threads

Nomadic Threads (NT) (JENKS, 2004) é uma ferramenta para computação multithread com acesso à memória remota com *cache* e migração de *threads* para explorar a localidade de arranjos de memória com *Myrinet*. Essa ferramenta roda em MPI sobre *clusters* Linux.

Nomadic Threads foca migração de *threads* para aproveitamento de características de localidade espacial. É importante perceber que NT não é capaz de migrar processos, apenas *threads*. NT é baseada em *frames*, *i.e.*, porções de memória alocadas para a execução de *threads*. Como *MigThread* (Sub-seção 2.2.5) e Cilk (Sub-seção 2.2.4), baseia-se no uso de um compilador e na existência de um núcleo próprio, em nível de usuário, de *threads*. Nesse contexto, *threads* são geradas pelo compilador, que procura minimizar as situações de troca de *thread*. Os *frames* são reservatórios de dados distribuídos pelas máquinas participantes da computação e possuem dados das máquinas relativos às *threads* que rodam sobre eles em um dado momento. Assim, a execução de uma *thread* em um *frame* é chamada de *ativação*.

NT é executada sobre uma implementação MPI, com uma arquitetura de comunicação curiosa. Ao invés de enviar os dados a nós que precisam deles, uma ativação ou migra para o nó que contém os dados, ou, dinamicamente, faz uma busca em memória remota, se o dado não é local. Em ambos os casos o tamanho da mensagem é pequeno (de 64 *bytes* a 512 *bytes*) e tamanhos ainda menores são usados para mensagens de coordenação e requisições. NT usa uma biblioteca de troca de mensagens chamada CMMD por baixo da implementação MPI.

O sistema de troca de mensagens empregado é extremamente assíncrono e menos coordenado que a maioria dos programas MPI. Além disso, o sistema de escalonamento de *threads* é empregado da seguinte maneira:

1. Roda todas as *threads* presentes na ativação atual.

2. Somente quando não há mais *threads*, o escalonador rouba outra ativação com *threads* aptas a executar.

Enquanto o escalonador se encontra no passo descrito no item (2), ele faz *pooling* para tratar todas as mensagens que chegarem nesse período. A transferência de mensagens que chegam endereçadas à *thread* antiga para a eventual nova *thread* é feita de maneira transparente ao programador. Como algumas implementações MPI populares (*v.g.*, LAM-MPI) não são *thread-safe*, NT elege uma *thread* única especialmente para fazer comunicações e redistribuí-las a outras *threads* (via memória compartilhada), realizando multiplexação na comunicação. Isso é feito através de chamadas de comunicação não-bloqueantes que existem em MPI, como `MPI_Irecv` (receber mensagens de modo não-bloqueante) e `MPI_Test_any` (testa se qualquer dos *receives* não-bloqueantes recebeu alguma mensagem).

A migração é realizada à *frames*; isso simplifica bastante o trabalho de pré-processamento a ser feito pelo *framework*. Uma importante constatação sobre a migração de fluxos de execução em (JENKS, 2004) é que embora a migração ofereça localidade espacial, ela elimina a *localidade temporal*; qualquer algoritmo que se baseia no uso mais recente de algum dado não funciona em um ambiente desse tipo.

A migração de *threads* em NT é feita entre diferentes máquinas, em um modelo de *cluster*. NT usa como base o CM5 (JENKS; GAUDIOT, 2003), que é um escalonador para sistemas SMP e SMT. Nesse aspecto, o CM5 usa os mesmos algoritmos que NT, mas não opera sobre um ambiente MPI.

2.3 Escalonamento de Processos em Processadores

Para computação com SMP (que possui visão lógica de comunicação *all-to-all*), o algoritmo RAT é sabidamente ótimo. No entanto, em ambientes com processadores

dispostos hierarquicamente, essa abordagem falha; alguns processadores, geograficamente próximos, estão conectados por barramentos muito rápidos, enquanto outros, geograficamente distantes, têm uma latência de comunicação devido ao barramento utilizado nas *Wide-Area Networks* (*WANs*). Como o RAT não têm capacidade de ponderar seus alvos de roubo pela velocidade de comunicação entre os processadores, acaba por produzir resultados ruins nesse cenário. Sobretudo, RAT tem sua eficiência comprovada apenas quando todos os acessos à *deque* se fazem em tempo O(1); quando os recursos estão distribuídos na rede não há como aplicar tal garantia.

Essa seção apresenta soluções relevantes encontradas na literatura para lidar com a migração de processos (fluxos de execuções pesados) entre recursos de máquinas MIMD do tipo *cluster* e grade (NAVAUX; DEROSE, 2003).

2.3.1 SATIN

O SATIN (NIEUWPOORT; KIELMANN; BAL, 2000) é uma biblioteca Java para paralelização de programas D&C (via *spawn*/*sync*) em grades e agregados de *clusters* que emprega um algoritmo chamado Roubo Aleatorizado Cluster-ciente (RACC), altamente eficiente. Como nas bibliotecas anteriormente apresentadas, o SATIN necessita de um compilador específico para gerar código-fonte compatível com suas especificações. No caso, acaba estendendo o compilador Manta para Java.

SATIN suporta o uso de *threads* Java e RMI de maneira concomitante com suas próprias primitivas. Além disso, sua sintaxe e comandos são muito próximos aos usados na linguagem Cilk, como visto na Sub-seção 2.2.4. Um exemplo de código SATIN que implementa o cálculo do n-ésimo termo de Fibonacci é mostrado na Fig. 2.6. É de especial interesse a comparação desse código com o da Fig. 2.3 (código-exemplo para a linguagem Cilk), para se constatar a semelhança das duas abordagens.

```
class Fibonacci {
    SATIN int fib(int n) {
        if ( n < 2 )
            return n;
        int x = SPAWN fib(n-1);
        int y = SPAWN fib(n-2);
        SYNC;

        return x + y;
    }
    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        int result = f.fib(10);
        System.out.println("Fib 10 = " + result);
    }
}
```

Figura 2.6: (NIEUWPOORT; KIELMANN; BAL, 2001) Um exemplo de código SATIN: Fibonacci. As primitivas `SPAWN` e `SYNC` tem o mesmo significado que `spawn` e `sync` da Fig. 2.3
.

Devido ao fato de que o SATIN roda em máquinas de memória distribuída, objetos não-replicados que são passados como parâmetros em uma chamada de *spawn* não estarão disponíveis na máquina remota. Por isso, SATIN usa uma semântica de *call-by-value* na passagem de parâmetros quando o *runtime* decide que um método

vai ser executado remotamente. Quando a chamada é local a semântica é *call-by-reference*. Os algoritmos de roubo do SATIN são implementados sobre a biblioteca de comunicação Panda.

O RACC, implementado no SATIN, visa oferecer uma minimização do *overhead* em nós próximos (LAN) ao mesmo tempo em que procura reduzir a comunicação WAN e o tempo que um processador fica ocioso. RACC usa o algoritmo RAT dentro de *clusters* (*i.e.*, em um ambiente LAN). No entanto, outra abordagem é usada quando do roubo de trabalhos em ambiente WAN; não há um coordenador local para cada *cluster* (em comparação com outros algoritmos hierárquicos). A comunicação é feita diretamente nó-a-nó. A Fig. 2.7 mostra um diagrama dos canais de comunicação do RACC (observa-se que não há restrição na topologia dos nós). O pseudo-código baseado em Java para o algoritmo RACC pode ser visto na Figura 2.8.

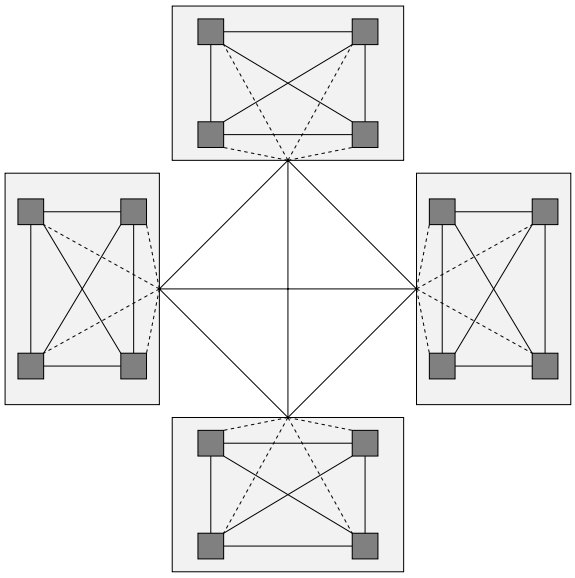


Figura 2.7: Diagrama de conexões para um algoritmo do tipo RACC. Retângulos cinza-escuro são processadores. Retângulos cinza-claro representam agregados de computadores. Linhas que ligam retângulos cinza-escuro são barramentos LAN, enquanto linhas que ligam retângulos cinza-claro são barramentos WAN.

RACC é de implementação simples. Sempre que algum nó fica sem tarefas a processar, envia, de forma não-bloqueante, uma requisição de roubo para um nó escolhido aleatoriamente em um aglomerado de nós escolhido aleatoriamente. Até que a mensagem chegue, o nó fica executando o algoritmo RAT no aglomerado de nós local. Se tarefas remotas chegarem após o processador ter conseguido tarefas locais, ele armazena essas tarefas em sua pilha, prontas a serem roubas local ou remotamente. Como resultado, tem-se um paralelismo nas comunicações que compensa o *overhead* de se recuperar mensagens à longa distância.

2.3.2 *Workstealing* Hierárquico sobre MPI-2

Seguindo a idéia do SATIN e ciente do fato de que MPI-2 é extensão da norma MPI que, dentre outras coisas, permite a criação dinâmica de tarefas, (PEZZI et al., 2006) apresenta um esquema de algoritmo de *Workstealing* Hierárquico (*Hierarquical Workstealing* – HWS), semelhante ao do SATIN, que roda sobre uma distribuição

```
void cluster_aware_random_stealing(void) {
    while ( NOT exiting ) {
       job = queue_get_from_head();
       if ( job ) {
           execute(job);
       } else {
           if ( nr_clusters > 1 AND NOT stealing_remotely ) {
               /* no wide-area messages in transit */
               stealing_remotely = true;
               send_async_steal_request(remote_victim());
           }
           /* do a synchronous steal in my cluster */
           job = send_steal_request(local_victim());
           if ( job )
               queue_add_to_tail(job);
       }
    }
}

void handle_wide_area_reply(Job job) {
    if ( job )
        queue_add_to_tail(job);
    stealing_remotely = false;
}
```

Figura 2.8: (NIEUWPOORT; KIELMANN; BAL, 2000) Pseudo-código estilo Java para o algoritmo RACC.

MPI-2 (LAM-MPI) e permite que se faça o escalonamento de tarefas em recursos ociosos e balanceamento de carga. O objetivo desse trabalho é prover um mecanismo eficiente de alocação/escalonamento de tarefas criadas dinamicamente, especificação não fornecida pela norma (GROPP; LUSK; THAKUR, 1999).

Ao contrário do SATIN, o algoritmo é dependente de topologia lógica; a comunicação na LAN dever ser estruturada como uma árvore e a raiz de cada uma dessas árvores é um gerente. Esse gerente contém um número inicial de tarefas e as processa. Ao gerente estão conectados sub-nós que também têm sua pilha de tarefas local. O último nível de cada árvore são folhas e representam a noção de tarefa no algoritmo. A topologia lógica empregada no algoritmo é um dos modos suportados pelo SATIN e pode ser visto na Figura 2.9.

A vantagem do algoritmo HWS é que ele independe da topologia física dos nós; conquanto que exista uma lista global e sincronizada de nós ociosos e nós ocupados, ele consegue, intrinsecamente, trabalhar com todos os nós como se formassem uma estrutura de árvore, graças a habilidade do MPI de criar tarefas dinamicamente e alocá-las em novos nós em tempo de execução. Dessa maneira, o algoritmo alcança boa eficiência e balanceamento de carga *on-line*, tanto em ambientes SMP/SMT como em *clusters* de *clusters*.

Há a desvantagem do algoritmo não ser totalmente transparente ao programador MPI-2; é necessário estruturar a criação de novas tarefas levando em consideração se o atual nó é uma folha ou um gerente, visto que gerentes possuem a função de roteamento associada.

A Tabela 2.1 mostra um comparativo entre os tempos conseguidos pelo HWS e o SATIN com algoritmo RACC.

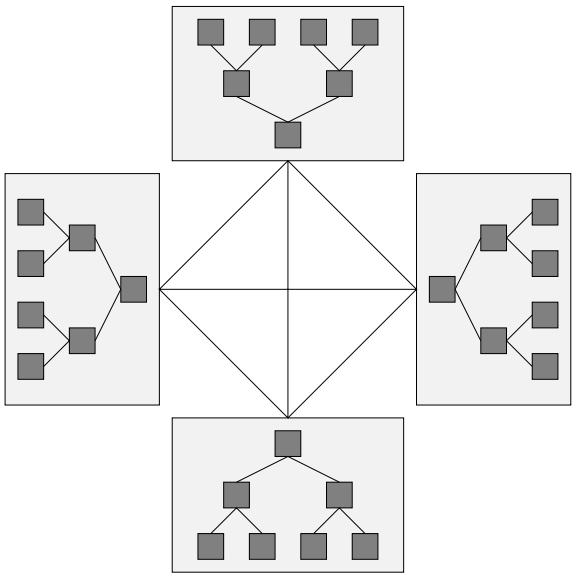


Figura 2.9: Diagrama de conexões para um algoritmo do tipo HWS. O significado dos retângulos e retas é o mesmo da Fig. 2.7.

	Execução MPI (seg.)		Execução SATIN (seg.)	
N	Média	Des. Pad.	Média	Des. Pad.
16	11,31	0,01	15,74	0,01
17	76,79	0,28	108,75	0,06
18	552,88	0,14	790,75	0,42
19	4252,88	0,57	6085,02	0,38

Tabela 2.1: (PEZZI et al., 2006) Programa N-Queens com MPI-2 versus SATIN (média dos tempos de execução e desvio padrão).

2.3.3 *Workstealing* sem Deque

O algoritmo de *Workstealing* sem Deque (*Deque-free Workstealing* – DFWS) (TRAORé et al., 2008) é uma modificação do RAT, que opera no topo da biblioteca KAAPI (GAUTIER; BESSERON; PIGEON, 2007). O KAAPI é uma *interface* para programação paralela que opera em nível de *threads* e processos e será vista na Seção 2.4.

Com o roubo de tarefas clássico o trabalho é definido por uma *deque* de tarefas. No DFWS, ele é definido por um *container* (no sentido aplicado em STL/C++) que provê iteradores para executá-la sequencialmente e particioná-la. O iterador usado para particionar o *container* é chamado `extract-par`. O iterador `extract-seq` extrai um bloco constante de sub-tarefas de alguma tarefa indicada.

Um esquema de funcionamento do DFWS pode ser visto na Figura 2.10.

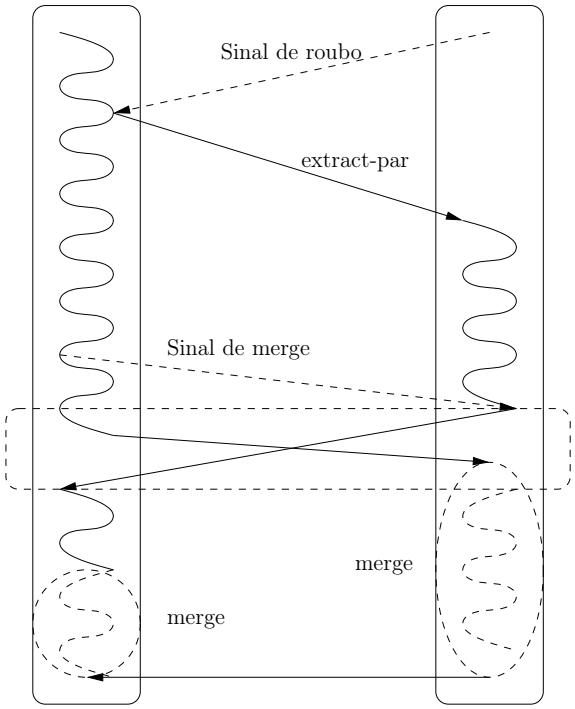


Figura 2.10: (TRAORé et al., 2008) Esquema de funcionamento do algoritmo DFWS. Os retângulos de cantos arrendondados são processadores e as linhas onduladas, fluxos de execução.

DWFS foi aplicado à uma paralelização da Biblioteca Padrão de *Templates* (*Standard Template Library* – STL) da linguagem C++, atingindo desempenhos melhores em relação a bibliotecas como *Intel Thread Building Blocks* (TBB) e *Multi-core STL* (MCSTL).

2.3.4 Escalonador Centralizado para LAM-MPI

Conforme mencionado anteriormente, a norma MPI-2 não esclarece como processos criados dinamicamente devem ser escalonados. Sendo um padrão *de facto*, é de grande interesse que a biblioteca a ser empregada usasse, garantidamente, um sistema eficiente de escalonamento de tarefas criadas dinamicamente. O único trabalho visto até agora capaz de fazer isso é visto na Sub-seção 2.3.2 (HWS) (PEZZI et al., 2006, 2007), mas o faz de maneira não-transparente ao programador. (CERA et al.,

Ambiente	nó 1	nó 2	nó 3	nó 4	nó 5
Escalonador-padrão LAM	39	0	0	0	0
Escalonador modificado	8	8	8	8	7

Tabela 2.2: Comparação dos escalonadores LAM-MPI nativo e o fornecido por (CERA et al., 2006; CERA; MAILLARD; NAVAUX, 2007)

.

2006; CERA; MAILLARD; NAVAUX, 2007) apresenta um trabalho no intento de abstrair esse escalonamento do programador, inserindo um *daemon* responsável por realizar esse escalonamento.

Basicamente, a distribuição MPI LAM (BURNS; DAOUD; VAIGL, 1994) é capaz de escalonar processos criados dinamicamente em processadores com um algoritmo *round-robin*, mas apenas quando uma chamada faz o *spawn* de vários processos; a especificação não é capaz de manter um estado consistente de *round-robin* entre todas as chamadas individuais e reinicia a fila de escalonamento a cada chamada. Um esquema de funcionamento do escalonador para tarefas dinâmicas da LAM-MPI pode ser visto na Figura 2.11.

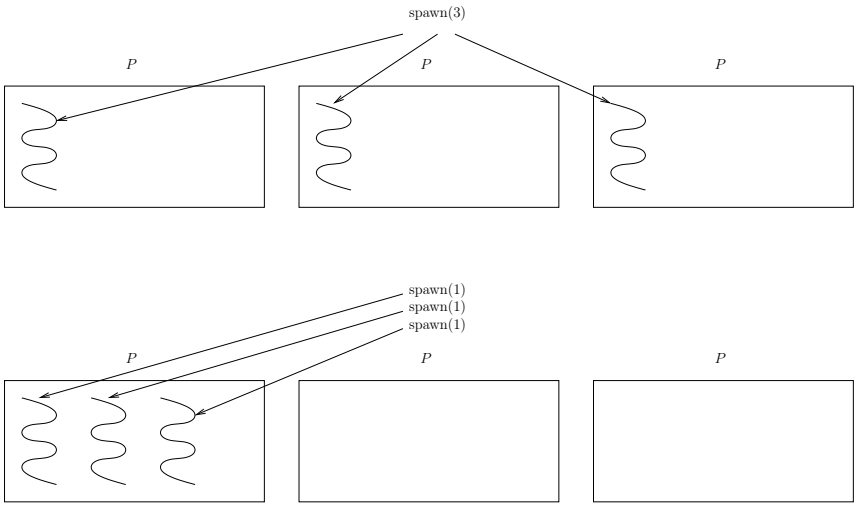


Figura 2.11: (CERA et al., 2006) Esquema de funcionamento do comportamento do escalonador de tarefas criadas dinamicamente da LAM-MPI. Cada retângulo representa um recurso e cada linha curvilínea um fluxo de execução.

Ao inserir um escalonador centralizado, consegue-se manter o estado do algoritmo de *round-robin* através de múltiplas chamadas de primitivas do tipo *spawn*. A Fig. 2.12 mostra o funcionamento do escalonador nas mesmas duas etapa mostradas na Figura 2.11.

Essa estratégia consegue atingir o balanceamento de carga ideal, conforme amostra a Tabela 2.2, que compara o comportamento após 39 chamadas de primitivas do tipo *spawn*(1) em 5 nós.

2.4 Escalonamento Simultâneo

Essa seção mostrará alguns trabalhos relacionados que fazem escalonamento em dois níveis: no nível de *threads* em processos e no nível de processos em proces-

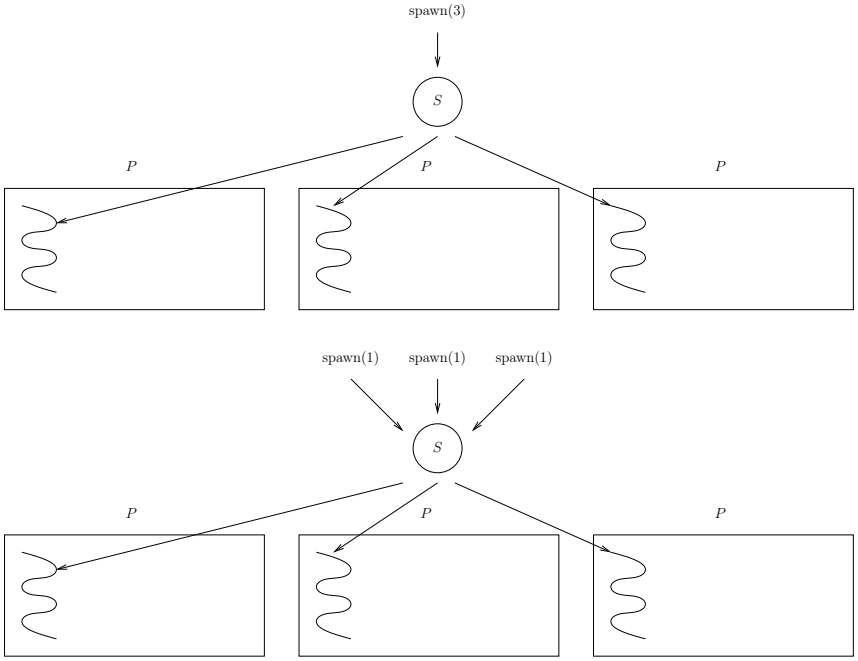


Figura 2.12: (CERA et al., 2006) Esquema de funcionamento do comportamento do escalonador centralizado de tarefas criadas dinamicamente com um *daemon* centralizado. O significado dos elementos é o mesmo da Fig. 2.11.

sadores. Primeiramente, examinar-se-á (HE; HSU; LEISERSON, 2006), com os fundamentos teóricos de algoritmos adaptativos de roubo de tarefas em dois níveis. Após, ver-se-á duas bibliotecas de computação paralela que fazem o mesmo: KAAPI (GAUTIER; BESSERON; PIGEON, 2007) e *Adaptive MPI* (AMPI) (HUANG; LAWLOR; KALé, 2004; HUANG et al., 2006).

2.4.1 Escalonamento Multi-nível Adaptativo

Essa seção é fundamentada no artigo (HE; HSU; LEISERSON, 2006). Este descreve dois algoritmos:

AGDEQ usa o Equiparticionamento Dinâmico (EQD) para escalonar os processos e um algoritmo adaptativo guloso (*Adaptive Greed Algorithm* – A-Greedy) como o escalonador de *threads*. Projetado para ambientes onde o escalonamento é centralizado.

ASDEQ usa o algoritmo DEQ para escalonar os processos e um algoritmo de roubo de tarefas adaptativo (*Adaptive Workstealing Algorithm* – A-Steal) para escalonar as *threads*. Desenhado para ambientes onde o escalonamento é distribuído.

Enquanto o algoritmo AGDEQ pode ser usado em ambientes como o apresentado na Seção 2.3.4 (CERA et al., 2006; CERA; MAILLARD; NAVAUX, 2007), não é adequado a um ambiente de alto paralelismo; decorrem gargalos da abordagem centralizada. Logo, este trabalho analisa apenas o algoritmo ASDEQ. Tal algoritmo é dito adaptativo pois consegue obter desempenho ótimo mesmo em um cenário onde processadores entram e saem da computação a todo instante. Nesse caso, pode-se

inverter a visão de que processos são alocados a processadores, admitindo-se que processadores participantes é que são alocados a processos.

A-STEAL, o algoritmo do ASDEQ responsável pelo escalonamento de *threads*, é uma variação do algoritmo RAT, apresentado na Subseção 2.2.1. Cada processador alocado a um processo cujas *threads* são escalonadas pelo A-STEAL possui uma *deque* (exatamente como a do RAT). Além dessa, A-STEAL cria uma nova *deque* para cada novo processador que é alocado ao processo corrente. Quando os respectivos processadores são desalocados do processo, cada uma dessas deques é marcada como uma *deque roubável*. Um processador alocado trabalha em apenas uma *thread* de cada vez. O algoritmo age como o RAT, apresentado no Alg. 2.5, com algumas diferenças.

Quando o processador se torna um ladrão, ele primeiro procurará, aleatoriamente, dentre as *deques* roubáveis. Podem haver, aí, um dos dois casos:

1. Se uma deque roubável é encontrada, então a deque é roubada e *todo* o seu conteúdo é passado ao ladrão.

2. Caso contrário, prossegue com a execução normal do RAT.

A-STEAL é usado ao invés do RAT pois o algoritmo que escalona processos em processadores, o DEQ, faz partição e redistribuição de processadores à processos de maneira dinâmica. Não há suporte há dinamismo do número de processadores disponíveis a um processo no RAT. DEQ não é coberto aqui, mas pode ser visto com mais detalhes em (NARLIKAR; BLELLOCH, 1999). ASDEQ é um exemplo de como a propriedade de manutenção do tempo de execução esperado do RAT se mantém em sistemas que disponibilizam de primitivas do tipo *yield*, conforme observado na Subseção 2.2.3 (escalonamento de *threads em processos*).

Ainda não há implementação do algoritmo ASDEQ e, portanto, seu desempenho não pode ser verificado na prática.

2.4.2 KAAPI

O KAAPI (GAUTIER; BESSERON; PIGEON, 2007) é uma plataforma para o escalonamento de *threads* em um ambiente de cluster multiprocessadores. O KAAPI é capaz de escalonar tanto *threads* quanto processos e possui suporte à criação dinâmica de tarefas. Na verdade, como o Cilk (FRIGO; LEISERSON; RANDALL, 1998), o KAAPI tem a noção de tarefa definida, sendo o mapeamento entre *threads* e processos transparente ao programador.

A sintaxe de um programa KAAPI estende a sintaxe Cilk (*i.e.*, uma extensão da linguagem C). A Fig. 2.13 mostra um exemplo de código que implementa o algoritmo de Fibonacci em paralelo. A sintaxe é da *interface* Athapascan, sob o qual o KAAPI é implementado. Com isso, é feito o controle de sincronização de escritas e leituras em um nível que fica abstraído do programador.

As *threads* KAAPI são implementadas via *interface* de POSIX Threads (*pthreads*) (BUTENHOF, 1997), mas sobre um núcleo de *threads* próprio. KAAPI pode operar em mais baixo nível, se for esse o desejo do programador. Existem primitivas como `kaapi_create` e `kaapi_mutex`, que têm exatamente o mesmo significado que seus correspondentes em *pthreads*.

O escalonamento KAAPI é feito por um algoritmo em dois níveis (*threads* em processos e processos em processadores). Cada processo é chamado de "processador

```
#include <athapascan-1>

void sum(shared_r int res1, shared_r int res2, shared_w int res)
{
    res = res1 + res2;
}

void fibonacci(int n, shared_w int res)
{
    if ( n < 2 )
        res = n;
    else {
        shared int res1;
        shared int res2;
        fork fibonacci(n-1, res1);
        fork fibonacci(n-1, res2);
        fork sum(res1, res2, res);
    }
}
```

Figura 2.13: (GAUTIER; BESSERON; PIGEON, 2007) Exemplo de código KAAPI: algoritmo de Fibonacci. A operação *fork* corresponde a uma operação do tipo *spawn* e não é necessário o uso de *sync*; cada variável utilizada na computação paralela possui um tipo `shared_w` (para escrita) ou `shared_r` (para a leitura) quando são descrições de parâmetros e possuem o tipo único `shared` quando declaradas.

virtual" e, basicamente, roda-se o algoritmo RACC, apresentado quando o SATIN foi abordado.

KAAPI mostrou-se mais eficiente que algumas plataformas de uso geral para memória distribuída (GAUTIER; BESSERON; PIGEON, 2007), como o MPI (MPICH) (GROPP, 2002).

2.4.3 AMPI

Adaptive MPI (AMPI) (HUANG; LAWLOR; KALé, 2004; HUANG et al., 2006) é uma implementação do padrão MPI que suporta virtualização de processadores e migração de *threads* e processos. Na verdade a especificação MPI é usada como uma espécie de *frontend* para uma linguagem/sistema que já suportava virtualização e migração chamada CHARM++ (KALE; KRISHNAN, 1993). AMPI implementa "Processos Virtuais MPI" (PVMPI), que podem ser mapeados para um ou muitos processadores. Um esquema de relacionamento entre esses processos e processadores físicos pode ser visto na Figura 2.14.

O *runtime* AMPI é o grande responsável pelo gerenciamento e pela maleabilidade que os PVMPI possuem. Graças a ele, AMPI fornece vários recursos, dentre os quais:

controle de sobreposição automático entre processos e comunicação : seu algum processador virtual está bloqueado em uma primitiva do tipo *receive*, outro PVMPI, *cpu-intensive*, pode preemptar o processador físico.

balanceamento de carga automático : se algum processador físico está sobrecarregado, o *runtime* pode migrar alguns PVMPI para outros processadores que estejam ociosos ou com uma carga de tarefas menor.

checkpointing : pode ser feito de maneira automática, mediante chegada a um *milestone*, de forma transparente ao usuário.

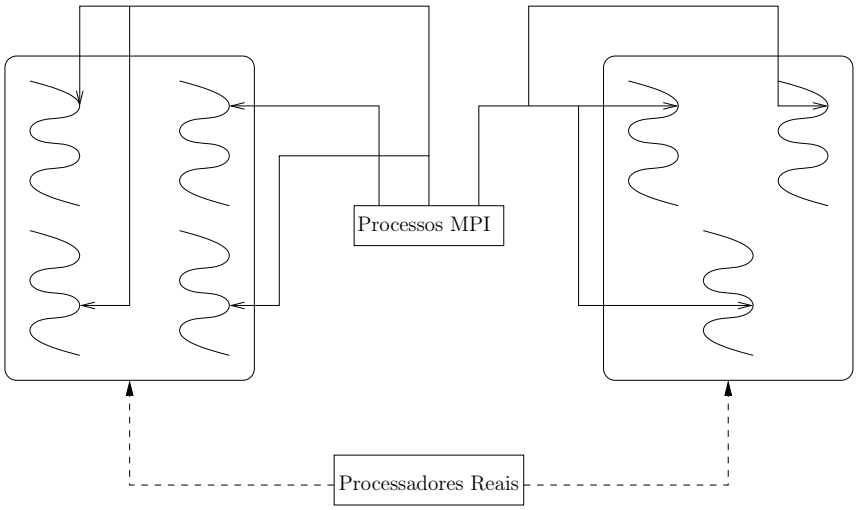


Figura 2.14: (HUANG; LAWLOR; KALé, 2004) Diagrama de implementação do relacionamento entre processos e processadores AMPI.

variação do número de processadores em tempo de execução : se processadores físicos forem adicionados ou removidos da computação ao longo dessa, AMPI é capaz de balancear as cargas dinamicamente e se adaptar a esse contexto.

A idéia central do AMPI é separar questões de escalonamento de questões de paralelização. Idealmente, um programador divide seu programa MPI em P programas, um para cada processador, enquanto o programador AMPI divide seu programa em V processadores virtuais, enquanto o *runtime* se encarrega de prover um escalonamento eficiente, com balanceamento de carga e migração, de forma transparente ao programador.

AMPI implementa os VP como *threads* CHARM++ de nível de usuário, ligadas a objetos comunicantes CHARM++. A troca de mensagens MPI é implementada como métodos de comunicação entre esses objetos. CHARM++ é capaz de migrar seus objetos, suportando, inclusive, roteamento.

A programação com AMPI é quase totalmente transparente ao programador MPI. De fato, um programa MPI qualquer é, também, um programa AMPI válido. Uma questão, no entanto, deve ser tratada adequadamente: variáveis globais. MPI permite que variáveis globais tenham valores diferentes em cada processador, admitindo que existam espaços de endereçamento relativos. AMPI, no entanto, trata todos os PVMPI em um processador físico como tendo um único espaço de endereçamento (são implementados via *threads*), o que implica que um PVMPI pode modificar as variáveis globais de outro. Ainda, quando da migração de PVMPIs, as variáveis globais são deixadas para trás. Por esses motivos, variáveis globais não são admitidas em AMPI.

AMPI mostra *benchmarks* convincentes sobre as características que enumera mas, infelizmente, não é comparado com outras distribuições MPI, para fins de desempenho, mesmo nos cenários onde claramente levaria vantagem.

2.5 Panorama

A técnica de roubo de tarefas é a mais eficiente e difundida para balanceamento da carga de trabalho em máquinas MIMD. Este capítulo apresentou o algoritmo RAT e seus derivados, que oferecem roubo de tarefas altamente eficiente em ambientes de memória compartilhada (*chips multicore*). Este mesmo algoritmo é usado como base para implementações famosas, como a linguagem Cilk e a biblioteca para C/C++ *Intel's Thread Building Blocks*.

O cenário de memória compartilhada oferece facilidades para a programação com roubo de tarefas. O tempo de comunicação e acesso às *deque* é $\Theta(1)$ e dados compartilhados não precisam ser migrados; o empilhamento de uma *thread* depende apenas de um *checkpoiting* cujo *overhead* é praticamente desprezível. Um cenário com memória distribuída, por outro lado, não oferece quaisquer garantias desse tipo.

Diversas implementações de *middleware* para a programação em *clusters* de computadores com troca de mensagens oferecem suporte ao escalonamento por roubo de tarefas. *v.g.*, KAAPI, SATIN, MigThread. Essas implementações, no entanto, contornam essas limitações estabelecendo padrões de programação e restrições próprios. A comunidade científica e a indústria endereçaram suas necessidades na construção da norma MPI, que é o padrão *de facto* para a troca de mensagens em agregados de computadores. Devido à estrutura aberta e geral da norma, que obsta restrições ao modelo de programação, existem poucos trabalhos na implementação de um escalonador eficiente por roubo de tarefas paralelas.

Os trabalhos que implementam escalonamento dinâmico dentro da MPI (*v.g.*, *nomadic threads*, AMPI) não suportam a criação dinâmica de tarefas paralelas, que, conforme já mencionado, é vital para a execução de algoritmos paralelos no modelo *fork/join*. O tema é alvo de pesquisa no grupo.

Tanto o escalonador centralizado para LAM-MPI (CERA et al., 2006) quanto o algoritmo HWS (PEZZI et al., 2007) buscam prover escalonamento dinâmico ao MPI. O primeiro é bem sucedido para instâncias pequenas de programas, mas apresenta problemas de escalabilidade com o crescimento do número de processos. O segundo consegue bons resultados quando comparado ao SATIN, mas é pouco genérico; o algoritmo deve ser implementado caso-a-caso. Além disso, não oferece uma base teórica consistente com a do algoritmo RAT, visto que a escolha da tarefa a ser roubada é determinística.

Em suma, a literatura oferece uma lacuna quando se busca uma implementação MPI (que seja comprovadamente eficiente) com escalonamento *on-line* por roubo de tarefas paralelas criadas dinamicamente. Este trabalho busca preencher essa lacuna.

O próximo capítulo introduz um algoritmo de escalonamento de roubo de tarefas eficiente sobre qualquer implementação da MPI. Esse algoritmo é programado sobre uma biblioteca de primitivas do tipo *map-reduce* (YEUNG et al., 2008), onde o programador tem disponível uma *interface* na linguagem C para definir suas tarefas e uma estrutura de dados do tipo *deque* para inseri-las e retirá-las; um *framework* cuida ativamente para que essas pilhas estejam sempre balanceadas. O Cap. 4 apresenta um algoritmo capaz de realizar escalonamento por roubo de tarefas de maneira transparente e distribuída, a medida que estas vão sendo criadas. Esse algoritmo é programado dentro da implementação MPICH2 e insere apenas uma nova primitiva na norma, de modo que tudo o que o programador precisa fazer é notificar o sistema, através dessa primitiva, toda a vez que uma tarefa está esperando a conclusão de suas sub-tarefas.

3 UMA BIBLIOTECA DO TIPO *MAP-REDUCE* PARA ESCALONAMENTO ON-LINE DE TAREFAS NO MODELO *FORK/JOIN* EM MPI

Este capítulo propõe o uso de uma adaptação do Alg. 2.5 (RAT) para realizar um balanceamento de carga eficiente em agregados de computadores . Este balanceamento é feito sobre algoritmos paralelos D&C do tipo *fork/join* em um ambiente MPI através da implementação de um escalonador distribuído. Esse escalonador é baseado na discussão proposta em (BLUMOFE; LEISERSON, 1994), sobre escalonadores para ambientes *multithread* e no trabalho desenvolvido em (RANDALL, 1998) sobre a linguagem para programação *multithread* Cilk.

A adaptação entre os cenários não é trivial, dadas as diferenças nas limitações entre um cenário de memória compartilhada (*threads*) e memória distribuída (processos em uma máquina MIMD).

Neste contexto, a tarefa Γ é uma função associada a uma entrada que, quando consumida, é executada como um processo MPI.

3.1 RatMD

A proposta desse capítulo se consolida em implementar um escalonador MPI que empregue uma APP onde o critério de escolha do processador a ser roubado é aleatório. Essa implementação é uma versão em memória distribuída do que é apresentado em (BLUMOFE; LEISERSON, 1999). A idéia principal é contornar as limitações de um cenário com memória distribuída mudando a ordem com que as tarefas paralelas são empilhadas na *deque*.

A sincronização e salvamento de contexto de um processo em um cenário de memória distribuída é mais onerosa do que no de memória compartilhada. Dessa maneira, o empilhamento do processo ativo e execução do processo recém criado (do alg. RAT) é custoso e complexo.

O algoritmo proposto, *Roubo Aleatório de Tarefas para Memória Distribuída* (RATMD), muda a ordem de empilhamento das tarefas na *deque* de modo a eliminar a necessidade de salvamento de contexto e diminuir a comunicação. Cada processador também possui uma fila de tarefas desbloqueadas (*i.e.*, tarefas que em determinado momento dispararam sub-tarefas, bloquearam esperando o resultado da computação e agora estão aptas a executar novamente).

Teoricamente essas tarefas poderiam ser inseridas no fundo da *deque*. No entanto, isso poderia resultar em um cenário onde uma tarefa desbloqueada é inserida na *deque* e, antes de ser processada, é roubada por outro processador. Isso introduz

um *overhead* desnecessário na computação. Com duas listas, todas as tarefas aptas a serem processadas são consumidas antes de todas as outras tarefas, movendo o *overhead* para o caminho crítico, priorizando a carga de trabalho. Simplesmente marcar uma tarefa desbloqueada que esteja na *deque* como "não-roubável" obrigaria o processador que fez uma requisição de roubo a vasculhar a *deque* alheia, em busca de uma tarefa roubável; isso introduz uma variabilidade no tempo de roubo que não é desejável.

A execução termina quando o algoritmo D&C realiza uma detecção distribuída de término e sinaliza aos processadores. Escolheu-se omitir essa detecção do pseudo-código para fins de simplicidade.

O modelo de comunicação é o apresentado em (LIU; AIELLO; BHATT, 1993); se mais de uma requisição é feita à *deque* em um instante de tempo, então uma é atendida e as outras são enfileiradas por um adversário e atendidas na sequência. Se alguma requisição é feita enquanto outras são atendidas, então elas são enfileiradas para atendimento após a execução da próxima tarefa ou após o envio da requisição de tarefas, no caso de não existirem mais tarefas a serem realizadas.

RatMD é apresentado no Alg. 3.1. Para denotar a fila de tarefas desbloqueadas usa-se $R[p]$.

Algoritmo 3.1 RatMD()

```
 1: C0: { p é o processador inicial. }
 2:     PUSH(top[Q[p]], Γ_init)        ▷ Γ_init (tarefa inicial) em apenas um processador.
 3: C1: { Γ1[p] habilita Γ2[p] }       ▷ Γ1[p] sempre está executando em p.
 4:     PUSH(top[R[p]], Γ2)
 5: C2: { Γ1[p] faz spawn de Γ2[p] }
 6:     PUSH(top[Q[p]], Γ2)            ▷ Difere do Rat; thread criada é empilhada.
 7: C3: { Γ[p] morre ou bloqueia }
 8:     se R[p] ≠ ∅ então              ▷ Primeiro executa todas as tarefas da fila de aptas.
 9:         EXEC(POP(bottom[R[p]]))
10:     senão
11:         se Q[p] = ∅ então
12:             repita
13:                 p' ← SORTEIO(P)
14:             até Q[p'] ≠ ∅
15:             EXEC(POP(top[Q[p']]))
16:         senão
17:             EXEC(POP(bottom[Q[p]]))
```

A mudança em relação ao Rat é, conforme já mencionado, a mudança da ordem de empilhamento das tarefas; aqui, a tarefa recém-criada é empilhada ao invés da tarefa criadora (Alg. 4.4, linha 6). Com isso, evita-se fazer o *checkpoiting* da tarefa criadora e ter de migrá-la entre máquinas, evitando os problemas que surgem no cenário de memória distribuída e que inexistem no de memória compartilhada.

Uma requisição de tarefa consiste em uma mensagem de solicitação ao processador-alvo e a espera de resposta, que pode ser uma nova tarefa ou a mensagem de que não há tarefas naquela *deque*. No Alg. 3.1 a requisição é representada pela condição de parada da linha 14. Essa troca de mensagens é feita de maneira não bloqueante; enquanto espera pela resposta um processador pode atender outras requisições. Esse

modelo é suficiente para que o escalonamento não resulte em *deadlock*, como será visto na próxima seção.

3.2 Propriedades

Discorre-se sobre algumas propriedades do Alg. 3.1 (RatMD).

No RatMD os nós fazem apenas requisições não-bloqueantes. De acordo com (TANENBAUM, 2007) isso garante a ausência de *deadlock*.

Usando-se o RatMD também não há risco de ocorrência de postergação indefinida. No contexto apresentado, postergação indefinida é uma tarefa que, embora eventualmente roubada, nunca é executada, *i.e.*, a tarefa é transferida entre pilhas *ad infinitum*.

Para mostrar que uma tarefa eventualmente é executada basta apresentar um limite superior finito para o tempo de espera dela antes de ser executada. De acordo com o Alg. 3.1 uma tarefa é (1) executada na pilha do processador que a originou depois que todas as outras tarefas na sua frente foram executadas; ou (2) roubada e executada imediatamente. Logo, há um limite superior finito para que a tarefa Γ seja consumida: é o número máximo de tarefas que estão à frente de Γ durante a computação. Como uma tarefa roubada é executada imediatamente, não há risco de que ela fique sendo repassada entre pilhas infinitamente.

Para avaliar o desempenho do RatMD contra estratégias tradicionais para roubo de tarefas em agregados de computadores, este é comparado com uma APP que usa como critério de escolha o *round-robin*. *Round-Robin* é um critério comum (e popular) em sistemas de escalonamento de tarefas. A escolha de qual processador roubar é feita como no Alg. 3.2, que mostra um escalonamento estilo APP.

Algoritmo 3.2 APPRoundRobin()

```
 1: C0: { p é o processador inicial. }
 2:     PUSH(top[Q[p]], Γ_init)        ▷ Γ_init (tarefa inicial) em apenas um processador.
 3: C1: { Γ1[p] habilita Γ2[p] }       ▷ Γ1[p] sempre está executando em p.
 4:     PUSH(top[Q[p]], Γ2)
 5: C2: { Γ1[p] faz spawn de Γ2[p] }
 6:     PUSH(top[Q[p]], Γ2)
 7: C3: { Γ[p] morre ou bloqueia }
 8:     se R[p] ≠ ∅ então
 9:         EXEC(POP(bottom[R[p]]))
10:     senão
11:         se Q[p] = ∅ então
12:             i ← 1          ▷ Rouba em round-robin.
13:             repita
14:                 i ← i + 1
15:                 se i > |P| então
16:                     i ← 1
17:             até Q[p_i] ≠ ∅
18:             EXEC(POP(top[Q[p_i]]))
19:         senão
20:             EXEC(POP(bottom[Q[p_i]]))
```

No melhor caso, a APP atinge o o estado eficiente (sem necessidade de comunicação) em tempo $T_1/P + \mathrm{O}\left(\sqrt{P}\right)$. Esse resultado é apresentado no Corolário 3.1, que é formulado a partir do Teorema 3.1:

Teorema 3.1 *Seja uma APP onde apenas um processador possui a tarefa inicial empregando como critério de escolha de roubo o algoritmo round-robin. Após i rodadas t_i processadores terão tarefas aptas a serem executadas, onde*

$$t_i = \begin{cases} 1, & \text{se } i = 1 \\ t_{i-1} + i - 1, & \text{caso contrário} \end{cases} \tag{3.1}$$

Prova. A prova é por indução no número de rodadas. A base de indução é $i = 1$, onde, trivialmente, $t_i = 1$. Supondo que $t_i = t_{i-1} + i - 1$ é verdadeiro (hipótese de indução), resta provar que $t_i = t_{i-1} + i - 1 \Rightarrow t_{i+1} = t_i + i$.

Sem perda de generalidade, supõe-se um processador p_a cujo processador vizinho p_{a+1} obteve uma tarefa na rodada anterior $i - 1$ a partir do processador p_b. De acordo com o Alg. 3.2, na rodada i, p_a aponta para p_b (onde apontava p_{a+1}), *i.e.*, p_a garantidamente ganhará uma tarefa na rodada i. Também de acordo com o algoritmo, todos os processadores entre p_{a+1} e p_b também ganharam tarefas na rodada anterior.

Conforme a hipótese de indução, a quantidade de processadores entre p_{a+1} e p_b é $i-1$. Logo, exatamente $i-1$ novas tarefas em novos processadores estarão disponíveis para serem roubadas, além das tarefas disponíveis em p_b. *Ergo*, na próxima rodada serão roubadas i tarefas e, portanto, $t_i + i$ processadores terão tarefas disponíveis na rodada t_{i+1}. □

Corolário 3.1 *A complexidade temporal na execução de uma APP com o critério* Round-Robin *é*

$$\frac{T_1}{P} + \mathrm{O}\left(\sqrt{P}\right) \tag{3.2}$$

Prova. A prova é direta. Tomando por base o Teorema 3.1:

$$\begin{aligned} t_i &= t_{i-1} + i - 1 \\ &= ((t_{i-3} + i - 3) + i - 2) + i - 1 && \text{(Expandiu-se } t_{i-1} \text{ até } t_{i-3}.) \\ &= 1 + i^2 - \left(\tfrac{(1+i)i}{2}\right) && \text{(Removeu-se o ponto fixo.)} \end{aligned}$$

Ao isolar a variável i obtém-se a equação $i = (1 + \sqrt{8t_i - 7})/2$, pois valores negativos de tempo não fazem sentido. Como t_i corresponde ao número de processadores usados,

$$i = (1 + \sqrt{8P - 7})/2$$

o que implica em $i \in \mathrm{O}\left(\sqrt{P}\right)$. Tendo-se estimado o tempo que se leva até atingir o estado ótimo, o tempo de execução no melhor caso para o critério *Round-Robin* em uma APP é o tempo ideal de processamento paralelo T_1/P mais o tempo $i \in \mathrm{O}(\sqrt{P})$ que se leva até atingir o estado eficiente. □

Apesar de desempenho atingido com o criteŕio *Round-Robin* não ser ruim, o desempenho esperado de RatMD é melhor, conforme discutido na sequência.

Para o cálculo da complexidade temporal de melhor caso do RatMD precisa-se definir dois estados possíveis a serem atingidos pelo algoritmo. RatMD está em *estado eficiente* em um instante de tempo t quando todos os P processadores estão executando alguma tarefa. RatMD está em *estado de sincronização* quando não está em um estado eficiente.

Enquanto RatMD permanece em um estado eficiente, $M_P[\mathcal{A}](H) = 0$. Nesse caso, $T_P[\text{RatMd}](H) = T_P[min](H)$. Logo, quanto mais tempo o algoritmo ficar em estado eficiente, tanto melhor.

Usa-se $T_P^{syn}[\text{RatMD}](H)$ para denotar o tempo que o algoritmo RatMD permanece no estado de sincronização. Dessa maneira, RatMD sempre executa em tempo

$$T_P[\text{RatMD}](H) = \frac{T_1[\text{RatMD}](H)}{P} + T_P^{syn}[\text{RatMD}](H)$$

Após alcançar o estado eficiente, cada vez que um processador esvazia sua pilha o sistema entra em estado de sincronização. Assim que o processador rouba uma tarefa e esta tarefa é expandida, o sistema volta ao estado eficiente. O sistema pode nunca atingir o estado eficiente. O limite superior para a execução do RatMD é $T_1(H)$; no pior caso, o processador inicial p_i nunca é roubado, eliminando qualquer paralelismo da execução.

O limite inferior para $T_P^{syn}[\text{RatMD}](H)$ é $\log_2(P)$; o melhor tempo possível para o algoritmo é quando todos os processadores ganham tarefas após o processador inicial expandir a raiz, o que acontece, na melhor das hipóteses, em tempo $\log_2(P)$. A cada rodada no máximo um processador p_a rouba uma tarefa de algum processador p_b (conforme descrito no modelo de ordenamento de requisições em (LIU; AIELLO; BHATT, 1993)), a expande e preenche sua *deque*.

Surge, então, o problema de descobrir o valor de $T_P^{syn}[\text{RatMD}](H)$. Em (BLUMOFE; LEISERSON, 1999) é apresentado um jogo chamado "bolas e caixas", que trata do problema de arremessar pequenas bolas de metal em caixas cilíndricas, de modo que a todo arremesso corresponde um consumo de uma bola pela outra extremidade da caixa e as esferas vão aleatoriamente para cada caixa.

A modelagem da quantidade de arremessos necessários para que todas as bolas sejam consumidas é exatamente o problema de determinar $T_P^{syn}[\text{RatMD}](H)$ e, de acordo com o mesmo trabalho, $T_P^{syn}[RAT](H) \in \text{O}\left(T_\infty(H)\right)$.

Com isso, tem-se o Corolário 3.2:

Corolário 3.2 *A expectativa do tempo de execução do* RatMD *é*

$$T_P[\text{RatMD}](H) \in \text{O}\left(\frac{T_1[\text{RatMD}](H)}{P} + T_\infty[\text{RatMD}](H)\right) \tag{3.3}$$

Prova. Decorre de $T_P^{syn}[\text{RatMD}](H) \in \text{O}\left(T_\infty(H)\right)$. □

O resultado acima é importante, pois o valor de $T_\infty(H)$ é logarítmico, o que demonstra que o algoritmo executa, em média, com uma eficiência muito mais próxima ao melhor caso que ao pior caso. O Corolário 3.2 reforça o ponto de que, apesar das primeiras rodadas do algoritmo trazerem grande quantidade de comunicação não-aproveitável (requisições não atendidas) em função da unicidade do nó com a tarefa-raiz, os processadores ociosos rapidamente ganham tarefas.

Além do ganho de eficiência proposto, esse escalonamento mantém o consumo de memória proporcional somente ao tamanho da entrada n, conforme o Corolário 3.3.

Corolário 3.3 *A quantidade de memória consumida pelo algoritmo apresentado é* $S_P \in O(S_1 P)$.

Prova. Seja $\Delta = S_P - S_1$ ($S_P \geq S_1$). Se cada processador executar o algoritmo para toda a entrada (processando, inclusive, as sub-tarefas que lançar) em paralelo tem-se que $S_P = S_1 P \Delta$. Como no RatMD cada processador executa o algoritmo sobre uma parcela da entrada (característica intrínseca de um algoritmo D&C), tem-se que $S_P < S_1 P \Delta$ e, portanto, $S_P \in \mathrm{O}(S_1 P)$. □

Ressalta-se que os valores para o tempo de execução são valores médios; a escolha aleatória faz o algoritmo possuir um caráter não-determinístico. Ainda assim, o desvio padrão associado tende a zero (BLUMOFE; LEISERSON, 1999).

3.3 Implementação do Escalonador

RatMD foi implementado através de uma biblioteca para MPI para programação no modelo *map-reduce* (*v.g.*, YEUNG et al. 2008.) É oferecida ao programador MPI uma visão como a da Fig. 3.1. Essa figura representa as estruturas e alocações de dados seguindo um ciclo básico:

1. O programa decompõe uma estrutura de dados em sub-estruturas, analogamente à decomposição de uma tarefa em sub-tarefas.

2. As sub-estruturas (sub-tarefas) de dados são colocadas no topo da *deque* e uma lista de dependências de tarefas é atualizada com as novas dependências de sub-tarefas.

3. O programa retira uma sub-tarefa do fundo da *deque*. Volta para (1).

Esse ciclo de execução ocorre continuamente. O balanceamento da quantidade de sub-estruturas de dados em cada *deque* é feito através do algoritmo RatMD. Quando uma tarefa atinge um caso trivial, o programa deve resolvê-lo e marcar a tarefa como resolvida na lista de dependências. A biblioteca se encarrega de enviar o resultado parcial aos processadores que originaram a dependência, resolvendo novas dependências sucessivamente.

Em um nível de abstração mais baixo, o escalonador implementado oferece uma interface de *callbacks* para que o programador defina sua tarefa (estrutura de dados) e como particioná-la, além de suporte a *benchmarks*, (*v.g.*, tempo de execução, consumo de memória, quantidade de comunicação) através do (e integrado ao) mesmo recurso.

A biblioteca oferece uma implementação de *broadcast* não bloqueante que construída devido à falta de suporte na norma MPI. Como exemplo, na Fig. 3.2, segue a estrutura de dados a ser reescrita pelo programador e um preâmbulo de uma *callback* que usa essa estrutura.

O suporte através de *callbacks* também fornece uma *Aplication Program Interface* (API) para envio de mensagens de término distribuído e difusão de melhor resultado local, recurso frenquentemente utilizado em programas D&C no estilo *backtracking* e *branch and bound*.

Nossa API está fora da especificação MPI, mas segue sua convenção de nomenclatura, facilitando o aprendizado do programador. A programação é feita através

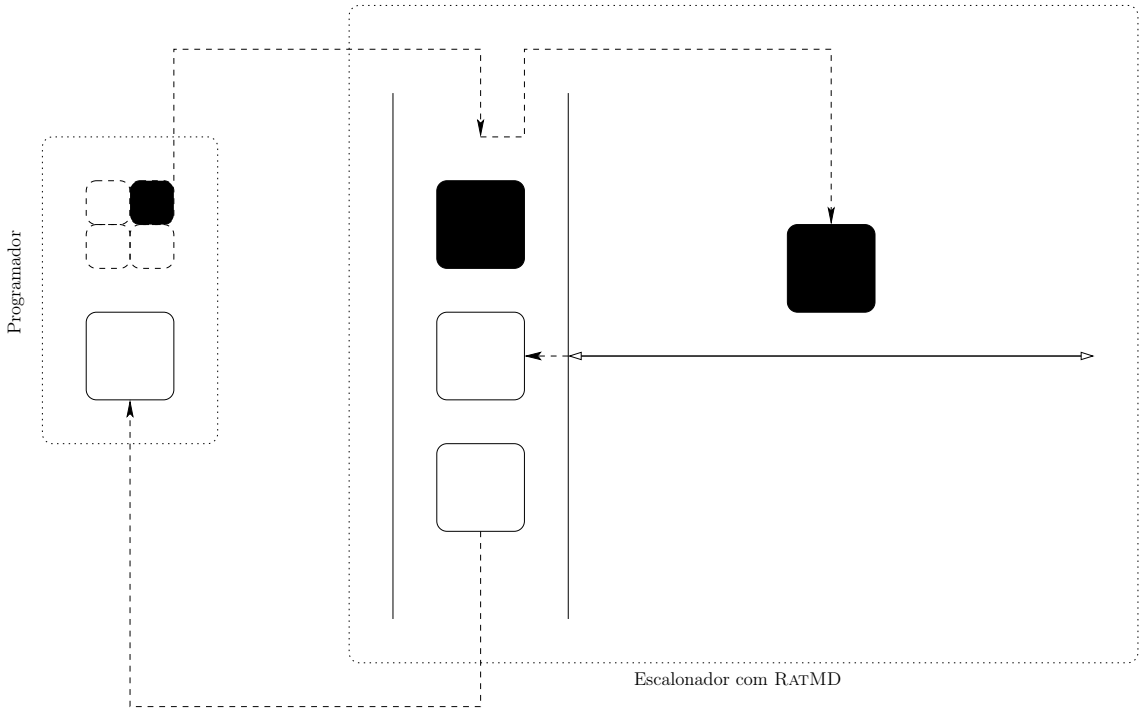


Figura 3.1: Representação de interações no escalonador com RatMD. Os retângulos pontilhados indicam a visão dos participantes (programador e escalonador). Os quadrados brancos são tarefas obtidas de outros processos e os quadrados pretos são tarefas oriundas da decomposição de outras tarefas pelo programador. A decomposição de tarefas em tarefas menores está representada mais à esquerda. A estrutura central é a *deque* de tarefas. A linha sólida que sai da *deque* é a comunicação com outros processos. As linhas tracejadas mostram a movimentação de algumas tarefas participantes.

```
/**
 * Basic structure for the programmer to define a task at
 * scheduler's context.
 */

typedef struct mpi_task_t {
    int active; /* Should not be setted or modified by the
                   programmer, system-use only */
    <código do usuário>
} MPI_Task;

/**
 * Gets next task from the local or remote deque, in that
 * order. [...]
 */

MPI_Task*
MPI_Get_task(MPI_Deque* deque);
```

Figura 3.2: Exemplo de código a ser reescrito pelo programador.

da implementação, pelo usuário, de protótipos de funções previamente definidos e o preenchimento de estruturas de dados que representam tarefas. A estrutura de dados mais importante nesse contexto é a que define uma tarefa (`mpi_task_t`) e um exemplo de implementação é mostrado na Figura 3.3.

```
/**
 * Basic structure for the programmer to define a task at
 * scheduler's context.
 */

typedef struct mpi_task_t {
    int active;       /* should not be setted or modified by the
                         programmer, system-use only. */

    int value[];       /* vector of values. */
    int (*compare)();  /* comparsion function. */
    int (*part)();     /* split function. */
    int is_valid;      /* boolean telling if this task is valid.*/
} MPI_Task;
```

Figura 3.3: Exemplo de estrutura que define uma tarefa. A variável `active` é usada internamente e não deve ser alterada.

A Fig. 3.4 mostra uma versão em pseudo-código próximo à linguagem C de uma implementação de um programa *branch and bound* (CORMEN et al., 2001) genérico em MPI (à esquerda na figura) e o mesmo código implementado no modelo induzido pelo escalonador com RatMD (à direita na figura); não há necessidade de se determinar para qual processo MPI a tarefa será enviada e nem comparar explicitamente o valor encontrado pela resolução da tarefa com o melhor valor global. Essas tarefas são abstraídas pelo escalonador.

3.3.1 Conclusão

Esse capítulo apresentou um algoritmo para escalonamento de tarefas paralelas em um ambiente de memória distribuída baseado em roubo de tarefas, derivado do Rat. Apresentou-se alguns pontos de interesse na implementação desse algoritmo, RatMD, através de uma biblioteca *map-reduce* compatível com qualquer implementação da MPI. Essa abordagem com uma biblioteca dedicada impõe um modelo rígido ao programador, tornando-a pouco portável em relação a códigos já programados. O próximo passo é incluir o escalonador no modelo de criação dinâmica de processos da MPI-2, minimizando esse problema.

O capítulo seguinte discute as modificações introduzidas na implementação MPICH2 para o uso de uma variação do RatMD para a distribuição dos processos criados dinamicamente entre os processadores. Em especial, apresenta-se uma nova primitiva análoga ao *join*, cuja a inserção em apenas um ponto-chave do código é suficiente para prover escalonamento de tarefas paralelas eficiente a um programa MPI D&C no modelo *fork/join* recursivo.

```
BranchAndBound ( input, glob bestValue )
{
    a = SplitInput ( input ); // Branch
    b = SplitInput ( input ); // Branch
    c = SplitInput ( input ); // Branch
    if not Bound ( bestValue, a ) then
        i = MPI_Send(pid_a, a, tag);
    else
        i = -INFINITY;
    if not Bound ( bestValue, b ) then
        j = MPI_Send(pid_b, b, tag);
    else
        j = -INFINITY
    if not Bound ( bestValue, c ) then
        k = BranchAndBound ( c );
    else
        k = -INFINITY
    i = MPI_Recv(pid_a);
    j = MPI_Recv(pid_b);
    r = Max ( i + x, j + y, k + z );
    if r > bestValue then
        MPI_Broadcast ( bestValue = r );
    return r;
}
```

```
BranchAndBound ( deque, blockinglist )
{
    MPI_Task t = MPI_Get_task ( deque );
    [...]
    if not Bound ( MPI_Retrieve_data(), t ) then
        a = SplitInput ( t )
    [...]
    MPI_Register_task ( a, blockinglist );
    [...]
    MPI_Push_task ( deque, a );
    [...]
    v = MPI_Enable_unblocking ( blockinglist );
    MPI_Update_data ( v );
}
```

Figura 3.4: Programa do tipo *branch and bound* em linguagem pseudo-C com MPI (à esquerda) *vs.* com MPI e escalonador baseado em RatMD (à direita). No código à esquerda, o modificador `glob` indica que uma variável é global e mantém seu valor mesmo quando a função é chamada recursivamente. `-INFINITY` é o valor de infinito negativo. `SplitInput` divide a entrada original de acordo com algum critério pré-estabelecido e `Bound` determina se aquela uma porção da entrada pode alcançar um resultado ótimo. No código à direita, apenas linhas relevantes a tarefa `a` são mostradas. `MPI_Get_task` retorna a próxima tarefa na *deque*; `MPI_Retrieve_data()` retorna o melhor valor global encontrado pelo programa de acordo com o protocolo do escalonador; `MPI_Register_task` informa ao escalonador que há alguma tarefa ainda não-resolvida; `MPI_Push_task` põe a tarefa no topo da *deque*; `MPI_Enable_unblocking` coleta as soluções de tarefas que já chegaram ao escalonador nesse meio-tempo e `MPI_Update_data` atualiza o melhor valor encontrado globalmente caso o novo valor seja melhor que o antigo.

4 ESCALONAMENTO DE TAREFAS INTEGRADO EM MPI-2

Esse capítulo estende as ideias anteriormente apresentadas para prover ao MPI-2 uma solução de escalonamento integrada no uso da primitiva de criação dinâmica de processos `MPI_Comm_spawn`. Espera-se aumentar a escalabilidade e eficiência de uma aplicação *fork/join* MPI-2 ao permitir que um número maior de processos seja criado devido ao balanceamento de carga eficiente implementado.

MPI-2, dentre outros recursos, introduziu uma família de primitivas que fornecem criação dinâmica (*i.e.*, em tempo de execução) de processos MPI (que podem ou não ser distintos do processo criador). A principal sub-rotina dedicada a fazer isso é

```
int
MPI_Comm_spawn ( char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm,
                 MPI_Comm *intercomm, int array_of_errcodes[] )
```

cujos parâmetros são (com informação de tipo e se é parâmetro de entrada ou saída):

1. `command` nome do programa a ser disparado (*string*, entrada).
2. `argv` argumentos para o programa disparado (vetor de *strings*, entrada).
3. `maxprocs` número máximo de processos inicializados (inteiro, entrada).
4. `info` conjunto de pares chave-valor descrevendo onde e como iniciar os processos (gerenciador, entrada).
5. `root` *rank* do processo no qual os argumentos anteriores são examinados (inteiro).
6. `comm` intracomunicador do processo designado por `root` (comunicador, entrada).
7. `intercomm` intercomunicador entre o grupo original de processos e o novo grupo (comunicador, saída).
8. `array_of_errcodes` vetor de códigos de erro por processo disparado (vetor de inteiros, saída).

A criação dinâmica de processos diferencia os comunicadores MPI em dois tipos:

intracomunicador. é o comunicador onde estão os processos que foram criados com o mesmo comando (seja a inicialização MPI clássica, sejam processos criados dinamicamente).

intercomunicador. é o comunicador onde estão processos criados com comandos distintos, através de `MPI_Comm_spawn` e similares. É usado para que o processo criador possa se comunicar com os processos criados e vice-versa. O intercomunicador associa o *rank* zero ao processo criador e os demais *ranks* aos processos criados, de modo que ficam visíveis o criador e os criados, em ambas as extremidades da computação.

Uma tarefa em MPI-2 é um executável associado a seus parâmetros e sempre é lançada como um processo. MPI-2 falha em especificar uma política de escalonamento canônica para tarefas recém criadas; cada implementação MPI-2 deve construir seu próprio algoritmo de assinalamento de tarefas em processadores.

Duas das mais usadas implementações, LAM-MPI e MPICH2, usam uma estratégia de *round-robin* estático na criação dinâmica de tarefas; a cada criação de n processos, o processador p que criou a tarefa a assinala aos primeiros n processadores (p_0 a p_{n-1}) de uma lista circular. Esta abordagem possui dois problemas:

1. distribuição equilibrada da carga de trabalho depende do programador saber o número de processadores e sua topologia em tempo de compilação ou por realizar trabalho extra em tempo de execução; e

2. grande desbalanceamento de carga de trabalho quando múltiplas criações com $n = 1$ ocorrem.

Ambos os métodos referenciados no item (1) também supõem que o número de processadores é constante e, no caso de se obter a informação em tempo de execução, que esse número é passado como parâmetro. Ambos os casos limitam a escalabilidade da aplicação.

Para ilustrar os problemas de balanceamento de carga em uma implementação *round-robin* para o escalonamento dos processos disparados pelo `MPI_Comm_spawn`, mostra-se um exemplo prático. Para o que segue, considera-se

$$esquerda[p_i] = \begin{cases} p_{i-1}, & \text{quando } i > 1 \\ p_{|P|}, & \text{caso contrário.} \end{cases}$$

$$direita[p_i] = \begin{cases} p_{i+1}, & \text{quando } i < |P| \\ p_1, & \text{caso contrário.} \end{cases}$$

Na implementação do escalonador nativo da MPICH2 (base para as implementações MPI da *Intel*, *Microsoft* .NET e *Windows*) quando um processador p chama `MPI_Comm_spawn` para criar apenas um novo processo ele é alocado no processador $direita[p]$. Quando chamado mais de uma vez a alocação segue uma lista circular. *v.g.*, quando `MPI_Comm_spawn` é chamado para a criação de 3 sub-tarefas, elas são alocadas, em ordem, em $direita[p]$, $direita[direita[p]]$ e $direita[direita[direita[p]]]$. Como visto abaixo, essa abordagem conduz a um cenário crítico no desbalanceamento dos processadores em um modelo *fork/join* recursivo em uma árvore D&C.

Considerando-se um modelo paralelo baseado em recursividade, é impossível gerar todos os processos necessários à execução de uma só vez; a medida que a árvore de recursão avança, novas primitivas `MPI_Comm_spawn` são necessárias. Relembrando as definições do Cap. 2 (Estado da Arte e Contexto Científico) δ é o número máximo de filhos que um nó da árvore D&C H possui, *i.e.*, o número máximo de sub-tarefas que uma dada tarefa pode gerar. A Fig. 4.1 mostra dois exemplos da

execução de uma árvore H (δ constante) com `MPI_Comm_spawn` usando o escalonador da implementação MPICH2 no modelo *fork/join* recursivo para pelos menos seis processadores.

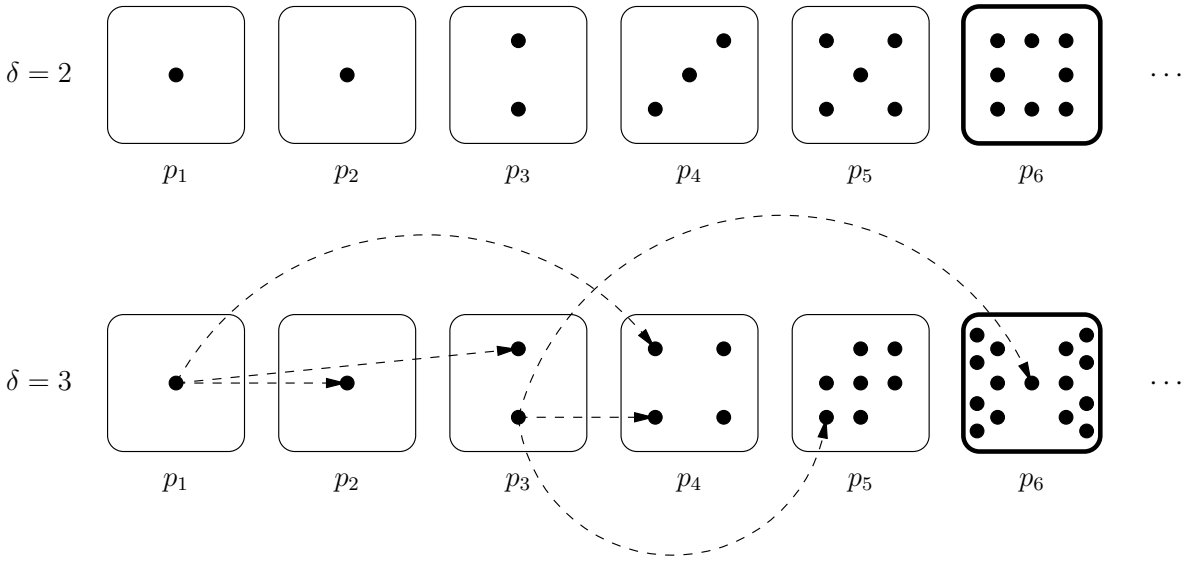


Figura 4.1: Exemplo de funcionamento nativo do MPICH2. A primeira linha é para $\delta = 2$ e a segunda para $\delta = 3$. Os quadrados com linha mais fina representam processadores onde cada uma de suas tarefas já lançou todas as suas sub-tarefas. Quadrados com a linha mais grossa representam processadores onde nenhuma de suas tarefas lançou alguma sub-tarefa. Setas tracejadas são exemplos da geração de sub-tarefas por uma tarefa escalonadas pelo escalonador MPICH2.

Na Fig. 4.1, segunda linha, p_1, com a tarefa original, aloca uma sub-tarefa em p_2, uma sub-tarefa em p_3 e uma sub-tarefa em p_4. Por sua vez, a tarefa de p_3 aloca uma sub-tarefa em p_4, uma sub-tarefa em p_5 e uma sub-tarefa em p_6 e assim sucessivamente. Se todas as setas tracejadas possíveis estivessem representadas, no máximo δ setas partiriam de algum ponto.

O exemplo da primeira linha da Fig. 4.1 mostra uma árvore D&C H onde $\delta = 2$ (*v.g.*, execução do cálculo do i-ésimo termo da série de Fibonacci). O desbalanceamento introduzido faz com que o processador p_6 tenho oito vezes mais processos que os processadores p_1 e p_2. O exemplo da segunda linha da Fig. 4.1 é feito com $\delta = 3$ (*v.g.*, Fig 2.1); o processador p_6 tem treze vezes mais processos que o processador p_1 e o processador p_2.

Generalizando os problemas apontados acima, uma política *round-robin* para o `MPI_Comm_spawn` produz grande desbalanceamento de carga em um algoritmo que siga o mesmo modelo de tarefas que produzem sub-tarefas. Mesmo que *round-robin* estático seja genérico o suficiente para a maior parte das aplicações, é uma política desastrosa quando se considera programas D&C, que podem fazer um bom e intensivo uso de processos criados dinamicamente. Como máquinas paralelas normalmente têm menos recursos do que o número de tarefas envolvidas em uma dada computação *fork/join*, o escalonamento eficiente de recursos é um ponto-chave para o balanceamento da carga de trabalho e desempenho como um todo.

A solução proposta como alternativa para o problema do desbalanceamento é agregar uma versão modificada do algoritmo RatMD à um escalonador com roubo

de tarefas em memória distribuída dentro da implementação MPICH2 de modo a tornar eficiente o escalonamento de processos criados dinamicamente em MPI-2. Além de ser uma das mais utilizadas implementações, escolheu-se MPICH2 em função de que

1. oferece um suporte consistente ao uso de *threads*, que podem ser utilizadas para otimizar o acesso a estruturas de dados; e

2. as outras implementações majoritárias, LAM-MPI e OpenMPI não estão em boa situação nesse momento; a primeira foi descontinuada em favor da segunda e esta última ainda é muito instável.

3. é a base para implementações MPI de grandes empresas, como a Intel e a Microsoft.

A arquitetura do gerenciador de processos no MPICH2 força a adoção de mudanças no algoritmo RatMD usado para instrumentalizar a biblioteca MPI-1. MPICH2 gerencia processos através de *daemons* que formam um sistema distribuído com uma topologia lógica em anel, independente da topologia física do *hardware*. A Fig. 4.2 mostra a relação entre a topologia física e lógica através da representação por grafos.

A próxima seção detalha fatores relevantes na implementação do *Message Passing Daemon* (MPD), responsável pela gerência da criação dinâmica de processos no MPICH2.

4.1 Message Passing Daemon na Implementação MPICH2

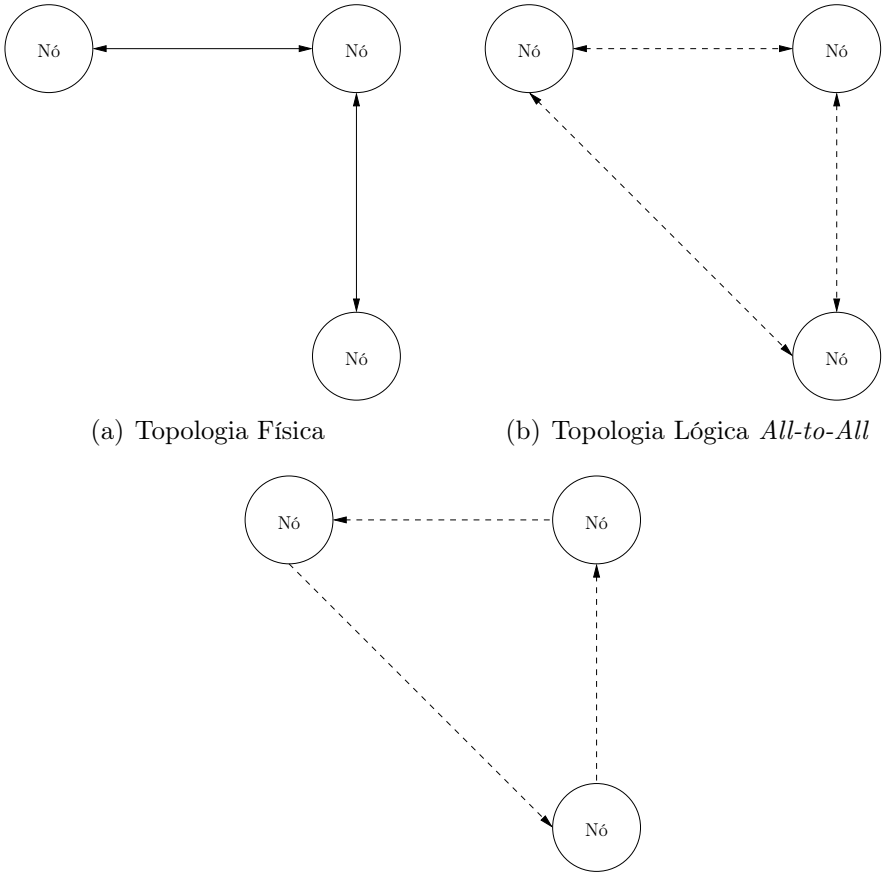
MPD é o *daemon* gerenciador de processos da MPICH2. É implementado em Python. Cada nó participante da computação possui um MPD, responsável por inicializar e gerenciar a criação dos processos, seja na inicialização ou na criação dinâmica de processos. Também é o MPD quem arbitra em que nó um processo criado dinamicamente executará. As subseções que seguem descrevem o funcionamento do sistema, obtido pelo processo de engenharia reversa.

4.1.1 Componentes

MPD é implementado como uma classe Python que possui três componentes principais:

MPDRing Uma subclasse de comunicação em anel. Toda a comunicação entre os diversos MPDs que estão executando é feita sob uma topologia lógica de anel. Cada MPD pode comunicar-se, através de métodos de envio de mensagem, com outro MPD (que recebe a mensagem através da mesma interface) que esteja "à direita" ou "à esquerda" na topologia de anel. A classe também oferece métodos próprios para tolerância à falhas e reconexões. Além disso, possui métodos que permitem a um MPD que não esteja inserido em um anel fazê-lo com o mínimo de parâmetros tomados do programador, através de técnicas como Zeroconf (GEBREMICHAEL; VAANDRAGER; ZHANG, 2006).

MPDMan Uma subclasse para gerenciamento de processos em C/C++/Fortran (doravante referenciado como processo-alvo). Cada processo-alvo comunica-se com o respectivo MPD através de um outro processo MPDMan (cardinalidade



(a) Topologia Física

(b) Topologia Lógica *All-to-All*

(c) Topologia Lógica em Anel

Figura 4.2: Grafos representando topologia física e topologia lógica em um agregado de computadores. A Fig. 4.2(a) representa a topologia física; cada vértice é um nó e cada aresta é uma conexão física direta ou ligada por um roteador dedicado. A Fig. 4.2(b) representa uma topologia lógica do tipo *all-to-all*. A Fig. 4.2(c) representa uma topologia lógica do tipo anel, como no MPICH2. Em ambos os exemplos de topologia lógica cada vértice é um nó e cada aresta um caminho virtual para troca de mensagens, que opera através de roteamento realizado pelo *middleware* de comunicação sobre a topologia física.

1:1). Cada MPDMan é criado via *fork* do MPD, mantendo informações sobre o ambiente possuídas pelo processo-pai. Cada MPD tem um vetor de referências para processos MPDMan, um para cada processo-alvo rodando. Este vetor de referências é representado pela variável V no Alg. 4.2.

MPDStreamHandler Uma classe para processar mensagens. De tempos em tempos esse módulo verifica que alguma mensagem chegou via MPDRing ou alguma outra *stream* previamente designada (*e.g.*, terminal, MPDMan). Responsável por fazer o tratamento de todas as mensagens recebidas pelo MPDMan, via *callbacks*.

.

A Figura 4.3 mostra um esquema da disposição dos MPDs em relação aos *hosts* e à topologia física da máquina. MPDStreamHandler é um módulo intrinsecamente ligado ao MPD, não estando representada. MPDRing é uma estrutura de dados também interna ao MPD, mas foi representado como uma caixa separada para melhor visualizar a comunicação existente entre este e outros módulos. Cada MPDMan roda como um processo separado, comunicando-se com os outros módulos via troca de mensagens. Toda a comunicação real é feita via *Sockets* TCP/IP. Cada MPDMan possui uma réplica do MPDRing do processo-pai (pois foi criado via *fork*), podendo usar sua própria referência para se comunicar com outros processos e, portanto, não necessita se comunicar com o processo-pai para tanto.

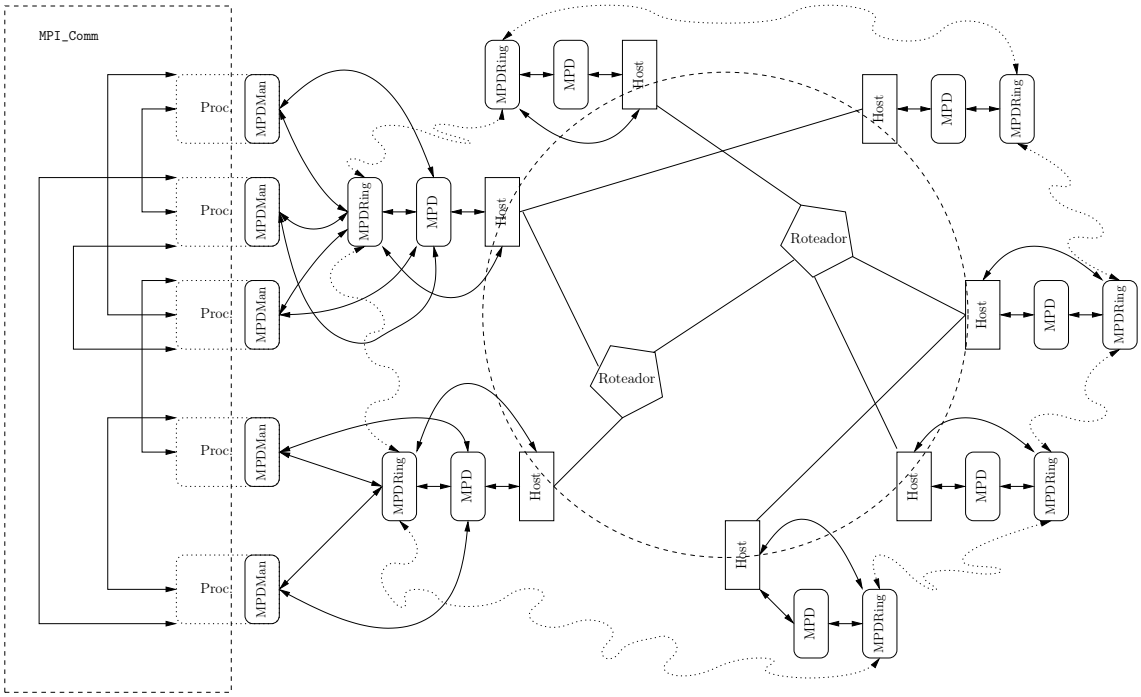


Figura 4.3: Topologia dos principais componentes do MPD. Retas pretas significam um caminho físico entre *hosts*, com emprego ou não de um roteador. Setas com linhas sólidas significam comunicação direta. Setas pontilhadas mostram a topologia lógica da comunicação entre os MPDRing. Linhas tracejadas indicam o agrupamento de componentes na visão do projeto; o retângulo mostra a visão do ponto de vista do programador MPI, enquanto o círculo representa a abstração da topologia física sob a topologia lógica em anel.

Na Figura 4.3 também se vê que os processos-alvo, mesmo estando em um mesmo comunicador MPI podem estar em *hosts* distintos. A comunicação entre processos-alvo (*e.g.*, `MPI_Send`, `MPI_Recv`) é sempre direta, via *sockets*; a estrutura anelar dos MPDs é usada apenas para a gerência de processos (criação estática, criação dinâmica, informações meta-execução, propagação de mensagens de erro, *etc.*).

4.1.2 Notação e Operações Básicas

Os algoritmos dessa seção empregam a noção de *sockets*; um *socket* é uma ligação lógica entre dois processos para a troca de *mensagens* bidirecional entre esses. A ordem de entrega das mensagens é a ordem de envio.

Sobre *sockets*, BindSocketMPD($sock, maq$) ao ser invocado cria um *socket* de comunicação na variável *sock* entre o processo que invocou a primitiva e o processo MPD na máquina *maq*. O *socket* criado na MPD remoto tem necessariamente o mesmo nome que *sock*. Se mais de um processo utilizar essa primitiva para um *socket* cujo nome já está ativo em algum MPD, então este MPD, que já tem o *socket* ativo, passa a receber mensagens de todas as fontes de maneira indistinta e a mandar a mensagem para todos, em *multicast*. Essa semântica não corresponde exatamente ao comportamento de um *socket* MPD como programado, mas seu comportamento mais simples ofusca partes do código do MPD sem relevância para este trabalho, ao mesmo tempo que entrega um comportamento final consistente com o da realidade. BindSocketMAN($sock, maq$) ao ser invocado cria um *socket* de comunicação na variável *sock* entre o processo que invocou a primitiva e o processo MPDMan na máquina *maq*. Quando maq = fork, usa-se uma semântica relacionada a primitiva `fork` UNIX; um novo processo MPDMan é criado na mesma máquina e o *socket* *sock* comunica-se com este processo.

Na notação empregada não há necessidade da máquina-alvo *maq* sincronizar-se de alguma maneira, estando o *socket* imediatamente disponível nas duas máquinas, com o mesmo nome. Caso *maq* queira enviar uma mensagens através de um *socket* onde a primitiva ainda não foi usada, considera-se que a mensagem fica armazenada em um *buffer* de tamanho tão grande quanto necessário até que seja consumida. Se, por outro lado, quiser-se receber uma mensagem de um *socket* ainda não foi inicializado, bloqueia-se até que este seja inicializado e receba sua primeira mensagem.

Há a necessidade de uma primitiva que periodicamente verifique os *sockets* e trate as mensagens recebidas adequadamente.
HandleStreams($A = \{sock_1, \ldots, sock_n\}, t$) durante t instantes de tempo realiza continua e sequencialmente o seguinte: para cada *socket* $sock_i \in A$ ($1 \leq i \leq n$), se há uma ou mais mensagens disponíveis para serem recebidas através daquele *socket*, recebe exatamente uma mensagem. Para cada mensagem recebida desta maneira, existem um tratamento em forma de condição $\mathbf{C}_i$ apropriado que é executado imediatamente após o recebimento da mensagem. Para os *sockets* $sock_i$ e $sock_{i+1}$, o segundo só é verificado quando a condição que trata o primeiro encerrar.

A troca de mensagens entre *sockets* é feita através de duas primitivas. SocketSend($sock, mem$) envia a mensagem *mem* ao processo remoto associado a *sock*. Não retorna valores. SocketReceive($sock$) retorna a mensagem recebida por *sock*. É bloqueante.

No contexto apresentado, mensagens podem ter campos lidos, alterados, removidos ou adicionados arbitrariamente em tempo de execução. $x[mem]$ denota o campo x da mensagem *mem*. $mem \leftarrow (x_1, \ldots, x_n)$ denota a criação da mensagem *mem*,

contendo os campos x_1 a x_n. Os campos de *mem* pode ser inicializados na criação da mensagem. Se $x[m] = \perp$, então $x[m]$ não tem um valor associado (valor nulo, *null* ou *nil*). Por fim, $mem_1 \cdot ... \cdot mem_n$ é a concatenação das mensagens mem_1 até mem_n. A mensagem resultante possui todos os campos de todas as mensagens na ordem de concatenação, com os respectivos valores associados. (A operação de concatenação é usada de maneira idêntica em variáveis do tipo *string*.)

Os *daemons* MPD usam uma topologia lógica de comunicação em anel. Um *anel* é representado, em cada *daemon*, como uma estrutura de dados que contém dois *sockets*: *direita*[*anel*], que comunica-se com o anel do *daemon* MPD à sua direita, e *esquerda*[*anel*], que faz o mesmo, mas à esquerda. Uma mensagem enviada pelo *socket direita*[*anel*] no processador p é recebida pelo *socket esquerda*[*anel*] no processador *direita*[p] e vice-versa. CREATERING($A = \{maq_1, \ldots, maq_n\}, s, list$) cria um anel de tamanho n composto pelas n primeiras máquinas do conjunto ordenado A e retorna uma referência a este. O *socket list*, passa a receber mensagens relevantes sobre o estado da estrutura de dados anel (falhas, desconexões, modificações, *etc.*) ENTERRING(*anel*) procura na rede por um anel já formado, e insere *anel* neste, formando um novo anel, maior, visíveis por todas as máquinas dos dois anéis anteriores. Se nenhum anel for encontrado, retorna *anel*. Se um ou mais anéis forem encontrados retorna um anel resultado da composição entre *anel* e o maior dos anéis encontrados.

A implementação de ENTERRING em Python possui alguns detalhes adicionais sem relevância para a análise, mas importantes para a compreensão do protocolo utilizado para formar múltiplos anéis. *v.g.*, a implementação Python usa uma "palavra secreta" passada como parâmetro e a composição de anéis só é permitida para os que tenham a mesma palavra secreta, possibilitando a formação de anéis disjuntos.

4.1.3 Execução

A inicialização de uma computação é feita utilizando-se o comando `mpdboot`, passando-se como parâmetros a lista de nós (nome na rede ou endereço IP) alocados pelo sistema para a execução e quantos processos MPD serão distribuídos por esses nós. Esse comando dispara tantos processos quanto se queira na máquina local e distribui os restantes entre os outros nós via *round-robin*.

Após inicializado o ambiente, um programa MPI-2 previamente compilado é executado via comando `mpiexec`. O Algoritmo 4.1 descreve simplificadamente o processo de disparo do programa MPI-2 na máquina que invoca o `mpdexec`. Nesse contexto, *prog* é o caminho completo para um executável MPI a ser disparado nos nós previamente alocados por `mpdboot` e parâmetros específicos para a chamada desse programa. *Maq* é uma lista $\{maq_1, \ldots, maq_{|Maq|}\}$ de identificações na rede de máquinas remotas onde se disparar o MPD. *Par* é uma lista de parâmetros do usuário e do sistema operacional repassados ao MPD. *ntotalproc* é o número total de processos a serem executados no anel do MPD.

Após a execução de MPIEXEC, O MPD, ao receber a mensagem de execução, executa a condição $\mathbf{C}_{run}$, conforme visto na próxima sub-seção Alg. 4.2.

4.1.4 Algoritmo de Escalonamento MPICH2

O Alg. 4.2 mostra uma versão simplificada da implementação do *daemon* MPD utilizando a notação presente em (TEL, 2001) vista no Cap. 2. Os parâmetros do Alg. 4.2 são como segue. À exceção do parâmetro *Maq* e *Par* (parcialmente),

Algoritmo 4.1 MPIEXEC($prog, Maq, ntotalproc, Par$)

1: $m \leftarrow (cmd \leftarrow$ `mpdrun`$, prog, ntotalproc, Par)$
2: BindSocketMPD(con, maq_1)
3: SocketSend(con, m)
4: $ack \leftarrow$ SocketReceive(con)
5: **se** $sucesso[ack] = falso$ **então retorne** erro
6: **senão**
7: HandleStreams($\{con\}, \infty$)

todos os outros são passados de maneira transparente ao usuário. *Maq* é uma lista $\{maq_1, \ldots, maq_{|Maq|}\}$ de identificações na rede de máquinas remotas a que participam da computação. *localhost* é a identificação, na rede, da máquina onde o MPD está rodando. *pid* é um identificador de processo para aquele MPD, provido pelo sistema operacional. $nproc_{maq}$ é o número de processos de usuário por máquina por solicitação de criação de processos (antes ou durante a execução). *Par* é um conjunto parâmetros globais de execução passados ao `mpdboot` combinados, de maneira transparente ao usuário, com dados do sistema operacional. Todas as operações definidas sobre mensagens podem ser utilizadas sobre *Par*. Por fim, vale destacar que Q e V são as listas de tarefas de cada processador e a lista de gerenciadores de processos ativos (*managers*, MPDMan), respectivamente.

Embora um *daemon* MPD seja escrito com forte uso de Orientação a Objetos, apresenta-se o algoritmo em um estilo imperativo e simplificado, sem, *v.g.*, controle de erros, *logs* e algumas etapas de sincronização com *acks*, salvo quando relevante. Este formato evidencia a lógica e a matemática por trás da implantação, de modo que se encaixa melhor aos propósitos do trabalho.

Este algoritmo mostra que de tempos em tempos mensagens dos sistema são processadas e tratadas sequencialmente, dispensando a sincronização nas estruturas de dados.

A condição $\mathbf{C}_{init}$ inicializa as estruturas de dados a serem utilizadas, além de determinar se o algoritmo roda na máquina local ou não. As principais estruturas de dados usadas neste trabalho são Q, lista de tarefas disparadas dinamicamente por este processo, e V, lista de processos C/C++ gerenciados por aquele MPD, vista com detalhes mais à frente. O Alg. 4.2 usa uma combinação das primitivas EnterRing e CreateRing. Sempre se cria um anel de tamanho unitário e tenta-se juntar o anel criado a algum anel preexistente; caso seja o primeiro a fazer tal operação a semântica de EnterRing faz com que a referência retornada seja para o anel já criado, enquanto os subsequentes irão se compor com este. Para procurar por anéis já formados na rede, o MPD utiliza o protocolo Zeroconf (GEBREMICHAEL; VAANDRAGER; ZHANG, 2006). $\mathbf{C}_{init}$ habilita o *loop* principal do MPD, $\mathbf{C}_{loop}$, que fica em um ciclo perpétuo lendo as mensagens recebidas pelos *sockets* do inicializados.

$\mathbf{C}_{run}$ descreve as ações do sistema ao receber uma mensagem para um execução MPI (MPIEXEC, descrito no Alg. 4.1); basicamente a mensagem é passada para o MPD que está à direita no anel, onde é tratada pela condição $\mathbf{C}_{exec}$, que dispara $nproc_{maq}$ processos e repassa a mensagem para um MPD à direita, onde as mesmas ações ocorrem. Processos são disparados até que um limite, passado no corpo da mensagem, seja atingido. O mensagem é, então, repassada em anel até seu emissor,

Algoritmo 4.2 MPD($Maq, localhost, pid, nproc_{maq}, Par$)

1: $\mathbf{C}_{init}$: { MPD acaba de ser inicializado }
2: $anel \leftarrow$ CreateRing($\{localhost\}, 1, list$) ▷ Cria anel unitário.
3: EnterRing($anel$) ▷ Via Zeroconf.
4: $S \leftarrow \{con, list, direita[anel], esquerda[anel]\}$ ▷ Conjunto de *sockets* a serem verificados.
5: $Q \leftarrow \emptyset$ ▷ Lista de tarefas.
6: $V \leftarrow \emptyset$ ▷ Lista de processos gerenciados.
7: $inprogress \leftarrow falso$ ▷ Diz se um roubo de tarefas está em andamento.
8: $id \leftarrow localhost \cdot pid$ ▷ identificador desta máquina.
9: habilita *loop* principal
10: $\mathbf{C}_{run}$: { Recebe m pelo *socket con* onde $cmd[m] =$ `mpdrun`. }
11: $m \leftarrow m \cdot (sender \leftarrow id, procdisp \leftarrow 0)$
12: SocketSend($direita[anel], m$)
13: $\mathbf{C}_{exec}$: { Recebe m pelo *socket* $esquerda[anel]$ onde $cmd[m] =$ `mpdrun` ou $cmd[m] =$ `spawn` }.
14: **se** $cmd[m] =$ `spawn` **então**
15: **se** $targethost[m] \neq$ `any` **e** $targethost[m] \neq localhost$ **então**
16: SocketSend($direita[anel], m$)
17: **sai de** $\mathbf{C}_{exec}$
18: **se** $procdisp[m] = proctotal[m]$ **então**
19: **se** $sender[m] = id$ **então**
20: **se** $cmd[m] =$ `spawn` **então**
21: $inprogress \leftarrow falso$
22: **senão**
23: SocketSend(con, m)
24: **senão**
25: SocketSend($direita[anel], m$)
26: **senão**
27: **para** $i \leftarrow 1$ até $nproc_{maq}$ **faça**
28: **se** $procdisp[m] \leq proctotal[m]$ **então**
29: BindSocketMan($man,$ `fork`)
30: $m' \leftarrow m \cdot (Par_{mpd} \leftarrow Par)$
31: SocketSend(man, m')
32: push($top[V], man$)
33: $procdisp[m] \leftarrow procdisp[m] + 1$
34: **senão**
35: **sai laço**
36: SocketSend($direita[anel], m$)
37: $\mathbf{C}_{loop}$: { *Loop* principal é habilitado. }
38: **enquanto** *verdadeiro* **faça**
39: **se** $Q \neq \emptyset$ **e** $inprogress \neq verdadeiro$ **então**
40: SendSocket($direita[anel]$, pop($bottom[Q]$))
41: **continuar**
42: HandleStreams($S, 0.8s$)
43: $\mathbf{C}_{spawn}$: { Recebe m pelo *socket man* onde $cmd[m] =$ `spawn`. }
44: $inprogress \leftarrow verdadeiro$
45: $m \leftarrow m \cdot (sender \leftarrow id, procdisp \leftarrow 0)$
46: push($top[Q], m$) ▷ Tratado em $\mathbf{C}_{loop}$

que atualiza a variável *inprogress* para *falso*, sinalizando que a criação de processos chegou ao seu fim.

$\mathbf{C}_{spawn}$ trata o recebimento de uma mensagem de criação dinâmica de processos originada por algum dos processos gerenciados na lista V. A mensagem é avaliada e o MPD que trata essa mensagem ajusta o valor de *inprogress* para *verdadeiro* e põe a mensagem no topo de Q para que a execução da condição $\mathbf{C}_{loop}$ do mesmo MPD envie-a para o MPD à direita, que a tratará com a condição $\mathbf{C}_{run}$, descrita acima.

4.1.5 Criação Dinâmica de Tarefas

A criação dinâmica de tarefas ocorre quando algum processo-alvo faz uma chamada à `MPI_Comm_spawn` ou similar. Primeiramente o algoritmo 4.3 mostra a implementação da estrutura MPDMan usando a notação de (TEL, 2001). O Algoritmo 4.3 usa uma nova primitiva, BindSocketProc(*sock*, `run` *prog Par*) que ao ser invocada cria um *socket* de comunicação na variável *sock* entre o processo que invocou a primitiva e o processo resultante da execução do programa *prog* com os parâmetros *Par* na mesma máquina. Não retorna valores.

Algoritmo 4.3 MPDMan(MPD, *man*)

1: $\mathbf{C}_{init}$: { MPDMan acaba de ser inicializado }
2: $m \leftarrow$SocketReceive(*man*)
3: BindSocketProc(*proc*, `run` *prog* $Par_{prog} \cdot Par$)
4: HandleStreams($\{proc\}, \infty$)
5: $\mathbf{C}_{spawn}$: { Recebe m do *socket proc*, onde $cmd[m] =$ `spawn`. }
6: **se** $targethost[m] = \perp$ **então**
7: $Par_{mpd}[m] \leftarrow Par_{mpd}[m] \cdot (targethost \leftarrow$ `any`)
8: SocketSend(*man*, m)

A criação dinâmica de processos se dá pelo cumprimento do predicado da condição $\mathbf{C}_{spawn}$ no MPDMan (Alg. 4.3) e, posteriormente, no MPD (Alg. 4.2).

O desbalanceamento causado pelo comportamento padrão do MPD pode ser corrigido pela aplicação do RatMD, visto no Capítulo 3. No entanto, a escolha aleatória do alvo a ser roubado fica comprometida pela topologia lógica do MPD. Mudar essa topologia lógica implicaria em reescrever grande parte do MPD, fazendo uma derivação da distribuição MPICH2, o que foge do escopo do trabalho. Ao invés disso, apresenta-se o critério de Roubo de Tarefas para MPD (RtMPD), uma modificação do RatMD para uma topologia de comunicação em anel.

4.2 RtMPD

Devido à topologia física em forma de anel, um processador p, no RtMPD, não pode enviar requisições de roubo e receber tarefas de qualquer outro processador; deve-se limitar a escolha à $direita[p]$ ou à $esquerda[p]$. No entanto, pode-se ampliar o espectro de processadores endereçáveis em troca de um ganho no *overhead* no envio e recebimento da mensagem. Para isso, usa-se uma difusão em anel do tipo coleta (LYNCH, 1997): cada processador p recebe mensagens, uma mensagem por canal ($direita[p]$ ou $esquerda[p]$), a processa e, se uma resposta for necessária, a envia para o canal dual ao de recebimento. Nesse contexto, o roubo de tarefas pode acontecer

aleatoriamente, fazendo-se uma mensagem circular pelo anel e voltar ao seu emissor, onde cada processador que receber essa mensagem pode ou não preenche-la com uma fração da sua *deque* de tarefas. RTMPD é apresentado no Alg. 4.4; todas as condições presentes no Alg. 4.2 estão presentes no Alg. 4.4, com as seguintes modificações, para fins de economia de espaço:

1. Se alguma condição do Alg. 4.2 tem o corpo alterado, ela aparece integralmente no Alg. 4.4.

2. Se há uma nova condição no Alg. 4.4, então ela aparece integralmente.

3. Uma condição que não tenha sido alterada em relação ao Alg. 4.2 não aparece.

A execução termina quando o algoritmo D&C realiza uma detecção distribuída de término e sinaliza aos processadores, não representado para fins de simplicidade.

A condição $\mathbf{C}_{init}$ é como no Alg. 4.2, adicionando a inicialização de duas variáveis: *collect*, uma variável booleana que diz se o algoritmo está, no momento, realizando uma coleta de trabalho, e *notf* que é inicializado com o número máximo de MPDs por recurso do *cluster* e é usada para manter esse valor constante durante a computação.

$\mathbf{C}_{run}$, não representada, é exatamente como no Alg. 4.2. $\mathbf{C}_{exec}$ e $\mathbf{C}_{spawn}$ são omitidas pelo mesmo motivo.

$\mathbf{C}_{loop}$ adiciona o tratamento a roubo de tarefas; caso o MPD se encontre sem processar tarefas não-bloqueadas, envia uma mensagem de coleta de trabalho em anel, que é tratada pela condição $\mathbf{C}_{coleta}$; se recebe uma mensagem sem tarefas, põe a tarefa do topo de Q na mensagem, se a $Q \neq \emptyset$. Repassa a mensagem para o MPD à direita. Quando a mensagem chega até o emissor este executa a tarefa que veio da mensagem ou, se a mensagem veio sem tarefas, faz outra coleta, até conseguir novas tarefas. Caso a mensagem recebida por um MPD intermediário já tenha alguma tarefa anexada ou, caso $Q = \emptyset$, a mensagem é simplesmente passada para o próximo MPD.

Por fim, $\mathbf{C}_{notf}$ é acionada quando algum processo em V bloqueia, comunicando ao MPD. Dessa maneira, a variável *notf* é incrementada, o que permite que a condição $\mathbf{C}_{loop}$ dispare a execução de uma tarefa em Q ou execute a coleta de tarefas.

No Algoritmo 4.4, se *parprocs* $= 1$, então n tarefas podem executar ao mesmo tempo se, e somente se, $n - 1$ tarefas estiverem bloqueadas; se n tarefas estão executando então a Linha 19 tem que ter sido executada n vezes, o que só acontece se a condição da linha 14 for satisfeita n vezes, já que *parprocs* foi inicializado com o valor 1. Desse modo, a condição $\mathbf{C}_{notf}$ deve ter sido satisfeita $n - 1$ vezes e, como ela só é chamada quando alguma tarefa bloqueia, $n - 1$ tarefas estão bloqueadas. Esse raciocínio pode ser generalizado quando *parprocs* > 1, atribuindo a *parprocs* a semântica de ser o número de processos que podem executar em um recurso simultaneamente no algoritmo RTMPD. Variar esse valor é interessante quando os recursos são processadores *multicore*, já que RTMPD garante que processos não bloqueados ocupam as CPUs.

4.3 Propriedades

Essa seção apresenta propriedades importantes do algoritmo RTMPD, no cenário de memória distribuída.

Algoritmo 4.4 RtMPD($Maq, localhost, pid, nproc_{maq}, Par, parprocs$)

1: $\mathbf{C}_{init}$: { MPD acaba de ser inicializado }
2: $anel \leftarrow$ CreateRing($\{localhost\}, 1, list$) ▷ Cria anel unitário.
3: EnterRing($anel$) ▷ Via Zeroconf.
4: $S \leftarrow \{con, list, direita[anel], esquerda[anel]\}$ ▷ Conjunto de *sockets* a serem verificados.
5: $Q \leftarrow \emptyset$ ▷ Lista de tarefas.
6: $V \leftarrow \emptyset$ ▷ Lista de processos gerenciados.
7: $collect \leftarrow falso$ ▷ Diz se este processo está realizando uma coleta de trabalho.
8: $notf \leftarrow parprocs$ ▷ Número de notificações de bloqueio recebidas.
9: $inprogress \leftarrow falso$ ▷ Diz se algum roubo de tarefa está em progresso.
10: $id \leftarrow localhost \cdot pid$ ▷ Identificador deste processo.
11: habilita *loop* principal
12: $\mathbf{C}_{loop}$: { *Loop* principal é habilitado. }
13: **enquanto** *verdadeiro* **faça**
14: **se** $inprogress = falso$ **e** ($V = \emptyset$ **ou** $notf > 0$) **então**
15: **se** $Q \neq \emptyset$ **então**
16: $inprogress \leftarrow verdadeiro$
17: **se** $notf > 0$ **então**
18: $notf \leftarrow notf - 1$
19: $m \leftarrow$ pop($bottom[Q]$)
20: **senão**
21: **se** $collect = falso$ **então**
22: $collect \leftarrow verdadeiro$
23: $m \leftarrow (cmd \leftarrow$ `collect`$, sender \leftarrow id, work \leftarrow \perp)$
24: SocketSend($direita[anel], m$)
25: HandleStreams($S, 0.8s$)
26: $\mathbf{C}_{coleta}$: { Recebe m pelo *socket* $esquerda[anel]$ onde $cmd[m] =$ `collect`. }
27: **se** $sender[m] = id$ **então**
28: $collect \leftarrow falso$ ▷ Acabou a coleta de trabalho.
29: **se** $work[m] \neq \perp$ **então** ▷ Roubando alguma tarefa, executa.
30: $m \leftarrow work[m]$
31: **senão**
32: **se** $Q \neq \emptyset$ **e** $work[m] = \perp$ **então**
33: $work[m] \leftarrow$ pop($top[Q]$) ▷ Rouba tarefa.
34: SocketSend($direita[anel], m$)
35: $\mathbf{C}_{notf}$: { Recebe m através do *socket man* onde $cmd[m] =$ `notf`. }
36: $notf \leftarrow notf + 1$ ▷ Recebe notificação de que um processo bloqueou; outro pode rodar no seu lugar.

RtMPD não entra em *deadlock*. Em qualquer instante de tempo, se *parprocs* foi inicializado com $\alpha \in \mathbb{N}_+$, então existem no máximo α tarefas executando em cada nó. Então, uma das quatro acontece:

Criação de nova tarefa. Não há bloqueio, não conduz à *deadlock*.

Bloqueio. A tarefa bloqueia e $parprocs \leftarrow parprocs + 1$, logo, uma das duas coisas acontece ao incrementar *parprocs*: ou outra tarefa começa a ser executada no mesmo nó ou a tarefa que bloqueou é uma folha na árvore de execução e termina, possibilitando que outra tarefa seja executada em algum nó. *Ergo*, não conduz à *deadlock*.

Término. A única possibilidade de bloqueio é se o processador fizer uma coleta de trabalho. Nesse caso, ele bloqueia por exatamente P processamentos sequenciais de m (topologia em anel) e recomeça a executar sobre o resultado recebido, fazendo ou não outra coleta. O desbloqueio ocorre em tempo finito e portanto o sistema não entra em *deadlock*.

Resumidamente, se a todas as tarefas, em todos os processadores, estão bloqueadas, então algum processador bloqueado dará lugar uma tarefa de alguma das *deque*, que iniciará desbloqueada. Caso não hajam tarefas nas *deque*, então alguma tarefa é uma folha e desbloqueará em tempo finito, fazendo com que o sistema nunca entre em *deadlock*.

Usando-se o RtMPD também não há risco de ocorrência de postergação indefinida. No contexto apresentado, postergação indefinida é uma tarefa que, embora eventualmente roubada, nunca é executada, *i.e.*, a tarefa é transferida entre pilhas *ad infinitum*.

Para mostrar que uma tarefa eventualmente é executada basta apresentar um limite superior finito para o tempo de espera dela antes de ser executada. De acordo com o Alg. 4.4 uma tarefa é (1) executada na pilha do processador que a originou depois que todas as outras tarefas na sua frente foram executadas; ou (2) roubada e executada imediatamente. Logo, há um limite superior finito para que a tarefa Γ seja consumida: é o número máximo de tarefas que estão à frente de Γ durante a computação. Como uma tarefa roubada é executada imediatamente, não há risco de que ela fique sendo repassada entre pilhas infinitamente.

4.3.1 Balanceamento de Carga

O desempenho do algoritmo de escalonamento de tarefas dinâmicas padrão do MPD, além de produzir desbalanceamento de carga indesejável, faz *work-pushing*, o que produz comunicação e *overhead* em excesso.

No escalonador MPICH2, quando uma tarefa é crida, ela passa a executar imediatamente em algum processador, dividindo o tempo de execução com quaisquer tarefas que já estejam executando naquele processador. Este não é o caso no RtMPD. Durante a execução do algoritmo RtMPD, em um dado momento, se todas as *deque* possuem tarefas então o sistema está em balanceamento ótimo, mesmo que essas *deque* possuam um número diferente de tarefas; apenas *parprocs* tarefas não bloqueadas estão executando por processador em um dado momento. Um processador que não esteja executando ao menos uma tarefa está esperando o resultado de uma coleta de trabalho e portanto necessariamente sua *deque* está vazia.

Resta analisar o papel da coleta de trabalho no balanceamento de carga.

Quando um processador que fica sem tarefas manda uma mensagem de coleta de trabalho três situações podem estar em curso, de maneira exclusiva:

1. Há tarefas em alguma *deque*.

2. Não há tarefas em alguma *deque*; os $P-1$ processadores restantes estão processando folhas da árvore de execução.

3. Não há tarefas em alguma *deque*; alguns dos $P-1$ processadores restantes estão processando nós intermediários da árvore de execução. O restante está processando nós-folha.

Caso o sistema esteja como no item (1) então estará novamente balanceado em tempo $\mathrm{O}(P)$, que é o tempo de se realizar uma coleta de trabalho (mensagem dá uma volta no anel). Caso o sistema esteja como no item (2), então está balanceado e rumo ao término distribuído. Por fim, caso ocorra o item (3) é necessário estimar o tempo máximo que o sistema fica desbalanceado.

Tomando-se a notação para uma árvore D&C H mostrada no Cap. 2 (Estado da Arte e Contexto Científico), se o sistema está como no item (3) então pelo menos um nó está executando o trecho de código de uma tarefa correspondente às funções DIVIDIR ou CONQUISTAR (considera-se que a complexidade de AVAL, $\Theta(1)$, é somada à da função DIVIDIR, sendo absorvida por essa). O mesmo Cap. 2 diz que o trabalho está concentrado na árvore de recursão, *i.e.*, $d+c \in \mathrm{o}(\delta R)$. Seja l a latência de processar uma mensagem que chega por *esquerda*[*anel*] e fazê-la prosseguir no anel por *direita*[*anel*]. Em qualquer algoritmo onde $d+c \leq lP$ (lP é o tempo total que uma coleta de trabalho leva) o tempo de espera no caso do item (3) é $2lP$ (pois precisa-se de mais coleta para colher trabalho do algoritmo). No caso onde $d+c \in \mathrm{O}(P)$, então o tempo de espera é $\mathrm{O}(P)$.

O raciocínio acima mostra que o sistema está a maior parte do tempo balanceado e, mesmo que eventualmente apresente algum desbalanceamento, demora no máximo $\mathrm{O}(P)$ para voltar ao estado balanceado. Dessa maneira, espera-se que o sistema esteja constantemente próximo a um estágio balanceado, otimizando o balanceamento de carga.

4.4 Implementação do Escalonador em MPICH2

A implementação do escalonador consiste na modificação do MPD para que passe a agir de acordo com o RtMPD. As modificações inseridas devem induzir a modificações mínimas em códigos já existentes. A implementação apresentada do cálculo do e-ésimo termo da série de Fibonacci em MPI-2 é mostrada na Figura 4.4. O código invoca dinamicamente `FibTask`, que pode ser visto na Figura 4.5. Didaticamente, a entrada é enviada ao processo através de um `MPI_Send`, explícito, mas poderia ser serializada no `argv` e passada quando da criação do processo dinâmico. O uso de `MPI_Send`, no entanto, pode ser visto como uma aplicação da limitação imposta pelo DAG de execução Cilk 2.4 de que uma tarefa criada via *spawn* só pode ser executada depois que a primeira instrução depois do *spawn* original já foi executada. Na prática, no entanto, usa-se a passagem serializada dos parâmetros com finalidade de ganhar desempenho ao diminuir o *overhead* de interação com a rede.

```
1 int
2 fib ( int e )
3 {
4
5   MPI::Intercomm comm1, comm2;
6   MPI::Request req[2];
7
8   int a, b;
9   int aux;
10
11   if ( e == 1  || e == 2 )
12     return 1;
13
14   comm1 = MPI::COMM_WORLD.Spawn ( "./FibTask", MPI::ARGV_NULL, 1,
                                     MPI::INFO_NULL, 0, MPI_ERRCODES_IGNORE );
15
16   aux = e-1;
17   comm1.Send ( &aux, 1, MPI::INT, 0, 10 );
18
19   req[0] = comm1.Irecv ( &a, 1, MPI::INT, 0, 10 );
20
21   comm2 = MPI::COMM_WORLD.Spawn ( "./FibTask", MPI::ARGV_NULL, 1,
                                     MPI::INFO_NULL, 0, MPI_ERRCODES_IGNORE );
22
23   aux = e-2;
24   comm2.Send( &aux, 1, MPI::INT, 0, 10);
25
26   req[1] = comm2.Irecv( &b, 1, MPI::INT, 0, 10 );
27
28   /*--------------------------------------------------------------------*/
29   /**
30    * Notifies process manager that a irecv is made for allowing it to spawn
31    * one more process from its deque.
32    */
33
34   MPI_Block_notf();
35
36   /*--------------------------------------------------------------------*/
37
38   while ( ! MPI::Request::Testall(2, req) )
39     sched_yield();
40
41   return a+b;
42 }
```

Figura 4.4: Cálculo do e-ésimo termo de Fibonacci com MPI-2.

```
#include "Fib.h"

int
main ( int argc, char **argv )
{
  MPI::Init();
  MPI::Intercomm parentComm;
  MPI::Intercomm comm1, comm2;
  MPI::Status status;
  int e, a;

  parentComm = MPI::COMM_WORLD.Get_parent();
  parentComm.Recv ( &e, 1, MPI::INT, 0, 10, status );

  a = fib(e);

  parentComm.Send( &a, 1, MPI::INT, 0, 10 );

  MPI::Finalize();
  return 0;
}
```

Figura 4.5: Tarefa criada dinamicamente no cálculo da série de Fibonacci em MPI-2.

4.4.1 Primitiva de Notificação

Um processo precisa notificar explicitamente o MPD que está bloqueado a fim de que este execute o próximo processo que está na *deque* de tarefas, pois o processo corrente acabou de se bloquear. Com essa modificação garantiu-se que o sistema não entra em *deadlock*, visto que quando uma tarefa bloqueia outra imediatamente assume seu lugar, dando prosseguimento ao *caminho crítico* da computação.

Incorporar a notificação em um `MPI_Recv` ou `MPI_Irecv` faria com que a notificação fosse feita δ (ou $\delta - 1$ para *lazy task creation*) vezes por processo, o que eliminaria as garantias de paralelismo oferecidas pela inicialização da variável *parprocs*. Se, ao invés, a notificação fosse incorporada ao `MPI_Testall` a situação seria a mesma, já que a primitiva é chamada reiteradas vezes enquanto o processo está bloqueado. Escolheu-se implementar uma nova primitiva especial para fazer a notificação, `MPI_Block_notf` (linhas 28-36 da Fig. 5.5(b)), fora da norma.

Com essa modificação, introduz-se um elemento que faz com que códigos baseados nesse escalonador não sejam portáveis em relação a uma implementação MPI-2 diferente. A modificação, no entanto, é pequena e facilmente aplicável a um código D&C *fork/join* recursivo com MPI-2, ao contrário das modificação drásticas no modelo de programação introduzidas pelo uso da biblioteca com o escalonador baseado em RatMD.

O uso dessa primitiva de notificação é o ponto central na interação entre o programador e o escalonador programado. É seu uso que permite o roubo de tarefas em memória distribuída com MPI-2.

4.4.2 Comunicação Não-Bloqueante

O escalonador baseado em RtMPD, além do uso de `MPI_Comm_spawn`, necessita que o programador use recebimento não-bloqueante, por dois motivos:

1. Um processo que apenas aguarda que seu processo-filho compute deve, tão logo consiga ser escalonado pelo sistema operacional, abrir mão do uso do processador.

2. É necessário que o processo notifique ao MPD, explicitamente, quando está bloqueado.

Quando um processo bloqueia, abre mão do uso do processador; passa a corresponder ao estado bloqueado do RtMPD, que só espera até que uma resposta o habilite para ser executado novamente. Se o processo ficar bloqueado no `MPI_Recv`, não há garantias de que ele não realiza *busy waiting* (TANENBAUM, 2007). Ao mesmo tempo, quando um processo recebe uma resposta é porque necessariamente seu filho está morto ou prestes a morrer, o que garante que quando ele for reabilitado ainda respeitará o paralelismo especificado através da variável *parprocs*. As linhas 38-39 da Fig 5.5(b) mostram esse processo, de verificar a completude das tarefas enviadas pelos filhos e, enquanto elas não vêm, usar a primitiva UNIX `sched_yield`, que faz um processo abrir mão do uso do processador. Quando todos os dados estão disponíveis, quebra-se o laço e o restante da execução é continuado.

4.4.3 Alterações

Para fins de execução do escalonador distribuído, deve-se passar o grau de paralelismo (número de processadores por nó) para o MPD como parâmetro na inicia-

lização. Quanto à estrutura de *deque*, este já provia uma estrutura de dados FIFO para o ordenamento de tarefas a serem disparadas. Alterou-se essa estrutura de modo a funcionar como a *deque*. RTMPD foi implementado seguindo a filosofia de implementação do próprio MPD, fazendo que com todo o processamento fosse síncrono, não havendo disputa paralela pelo uso de alguma estrutura de dados. Desse modo, arcou-se com a perda de desempenho inerente a essa abordagem, onde o MPD com o escalonador distribuído tem de esperar até $2Pl$ até que um roubo seja bem-sucedido antes de enviar outras mensagens relevantes no sistema. O fato de um MPD só roubar processos se não tem mais tarefas a serem disparadas contribui para amortizar esse impacto. As Figuras 4.6 e 4.7 mostram, respectivamente, o trabalho do *loop* principal do MPD e o tratamento de mensagens de coleta de trabalho.

```
def runmainloop(self):
  # Main Loop
  while 1:
    if not self.spawnInProgress and ( (len(self.activeJobs) < par_level) or (self.recvNotf > 0) ):
      if self.spawnQ :
        self.spawnInProgress = 1
        if self.recvNotf > 0 :
          self.recvNotf -= 1
        self.do_mpdrun(self.spawnQ.pop(0))
        continue
      # Begins work stealing!
      # Sends collect message for work.
      elif not self.collectInProgress :
        self.collectInProgress = 1
        self.ring.rhsSock.send_dict_msg({'cmd':'wscollect', 'sender_id':self.myId, 'work':None})
```

Figura 4.6: Tratamento de processos dinâmicos no MPD – escalonador distribuído. Código Python rodando no *loop* principal .

```
# Treats work-stealing case.
elif msg['cmd'] == 'wscollect' :
  if msg['sender_id'] == self.myId :
    self.collectInProgress = 0
    if msg['work'] != None :
      # Execute task directly, no putting it on deque, because it
      # prevents starvation.
      self.do_mpdrun(msg['work'])
    else :
      # Its not the sender, msg must continue its course. If it has no
      # work already on it, gets top of the spawnQ, put on it and send
      # it around.
      if self.spawnQ and msg['work'] == None :
        # Pops from deque's top
        msg['work'] = self.spawnQ.pop()
        # Send it around.
        self.ring.rhsSock.send_dict_msg(msg)
```

Figura 4.7: Tratamento de mensagens de coleta de trabalho.

5 RESULTADOS

Este capítulo mostra resultados compatíveis com as garantias de qualidade obtidas teoricamente.

5.1 Biblioteca para MPI com RatMD

Os experimentos foram realizados utilizando-se MPI-1 e uma versão paralela do Problema da Mochila. As primitivas de escalonamento foram inseridas em tempo de compilação.

5.1.1 Problema da Mochila

A aplicação é uma solução B&B para o Problema da Mochila Ilimitado (MBB), descrito abaixo (KELLER; PFERSCHY; PISINGER, 2005).

Seja n a quantidade de tipos de itens disponíveis ($n \in \mathbb{N}$). Cada $x_i \in X = \{x_1, \ldots, x_n\}$ ($X \subset \mathbb{N}$) significa o número de itens do tipo i, onde cada tipo tem associado um valor $v_i \in V = \{v_1, \ldots, v_n\}$ ($V \subset \mathbb{R}_+$) e peso $w_i \in W = \{w_1, \ldots, w_n\}$ ($W \subset \mathbb{R}_+$). O peso máximo que a mochila considerada suporta é $C \in \mathbb{R}_+$. O problema é, então, achar valores de X, tal que

$$\text{maximizem} \quad \sum_{i=1}^{n} (x_i \times v_i)$$

$$\text{sujeitos a} \quad \left(\sum_{i=1}^{n} (x_i \times w_i) \right) \leq C$$

A complexidade temporal de pior caso é

$$C_p[\text{MBB}](n) \in O\left(\left(\frac{C}{min\,(W)} \right)^n \right) \tag{5.1}$$

MBB é um problema pertencente à classe $\mathcal{NP}$-Completo. Além disso, a resolução B&B do problema apresenta alta ocorrência de *bounds* (linha 17), o que causa grande desbalanceamento de carga, mesmo em ambientes com processadores homogêneos.

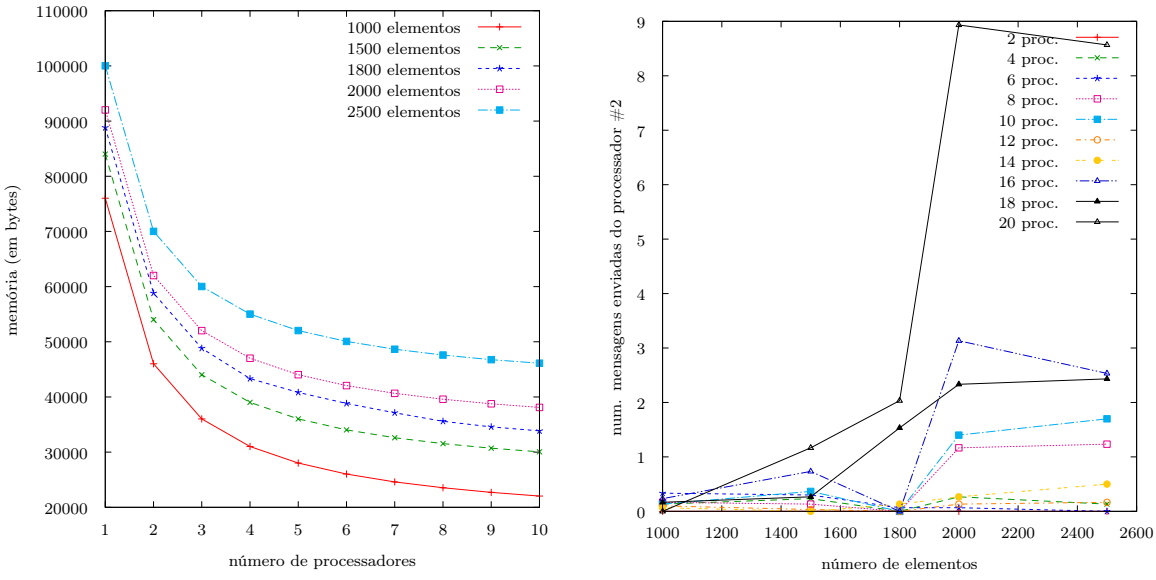
Usa-se, nos experimentos, a paralelização de uma solução recursiva para o problema vista em (KELLER; PFERSCHY; PISINGER, 2005), com a biblioteca MPI *map-reduce* baseada no escalonador RatMD (detalhada no Cap. 3).

5.1.2 Medições

Os experimentos foram conduzidos no *cluster* Labtec do Grupo de Processamento Paralelo e Distribuído (GPPD) da Universidade Federal do Rio Grande do Sul (UFRGS), usando a distribuição LAM-MPI e as seguintes configurações:

- uso de 10 nós, cada um com 2 processadores ($1 \leq P \leq 20$).
- cada nó é composto de dois processadores PENTIUM^{tm} III 1.266MHz, com 1 GB de memória RAM e barramento *Fast Ethernet*.
- 30 execuções para cada configuração, tomada a média aritmética do critério avaliado (*e.g.*, memória, tempo de execução). O desvio padrão foi sempre menor que $0,001$, salvo na avaliação da quantidade de mensagens enviadas por um dado processador, onde o desvio padrão permaneceu elevado.
- 1000, 1500, 1800, 2000 e 2500 tipos de itens (tamanho da entrada).
- o trabalho sequencial em cima da entrada é feito apenas quando sobra 1 (um) elemento no vetor de elementos considerados, lançando-se tarefas em paralelo quando contrário (*i.e.*, grão pequeno).

Primeiramente, a Figura 5.1(a) mostra resultados compatíveis com o Corolário 3.3. O crescimento de consumo de memória é linear, variando apenas com o tamanho n da entrada; a introdução de pilhas e filas e a estrutura do algoritmo tendem a aumentar a eficiência paralela sem aumentar significativamente o consumo de memória.



(a) Crescimento do consumo de memória linear em n.

(b) Comportamento variável na troca de mensagens.

Figura 5.1: Crescimento do consumo de memória linear em n (Fig. 5.1(a)) e comportamento variável na troca de mensagens (Fig 5.1(b)).

Devido ao caráter aleatório na escolha do processador a ser roubado, o número de mensagens trocadas por um processador específico não possui dependência funcional com n ou P. De fato, a Figura 5.1(b) mostra o caso específico do processador

#2, cuja comunicação é bastante caótica. Resultados em outros processadores mostraram um modelo de comunicação semelhante. Esse comportamento foi percebido em um sistema com processadores homogêneos, que induzem um menor nível de desbalanceamento no sistema. Nesse cenário não houve convergência que produzisse o desvio padrão desejado, haja visto que os valores tomados são muito discrepantes para convergirem em 30 execuções.

A Figura 5.2(a) mostra o ganho de desempenho referenciado na Seção 3.2. Essa implementação, no melhor caso, consegue ser 80% mais rápida que o mesmo algoritmo que usa *round-robin* (*work-pushing*) para distribuir as tarefas da APP. A ilustração de que em alguns casos o algoritmo supera o limite teórico mostrado advém do fato que a notação O esconde constantes multiplicativas relativas ao desempenho no Corolário 3.2.

A Figura 5.2(b) mostra que o consumo de memória do RatMD é praticamente idêntico ao consumo de memória da execução do Mbb sem o escalonador baseado em RatMD. Para ambas as medições foram considerados 2500 tipos de itens. Por fim, a Figura 5.2(c) mostra que o *speedup* obtido é próximo ao linear.

5.2 Extensão do MPICH2 com RtMPD

Os experimentos foram realizados utilizando-se MPICH2 1.1 (implementando a norma MPI-2) com os algoritmos de teste de carga, cálculo do i-ésimo termo da série de Fibonacci e ordenação por intercalação *Mergesort* (CORMEN et al., 2001) (este último aplicando o conceito de *lazy task creation*). O objetivo dos testes é verificar as observações contidas no Capítulo 4, obtendo um maior balanceamento de carga e permitindo o tratamento de instâncias maiores dos problemas apresentados sem comprometer o desempenho.

Os experimentos foram realizados no *cluster* ICE do Grupo de Processamento Paralelo e Distribuído (GPPD) da UFRGS, componente do *Grid 5000* (https://www.grid5000.fr). A configuração do *cluster* é:

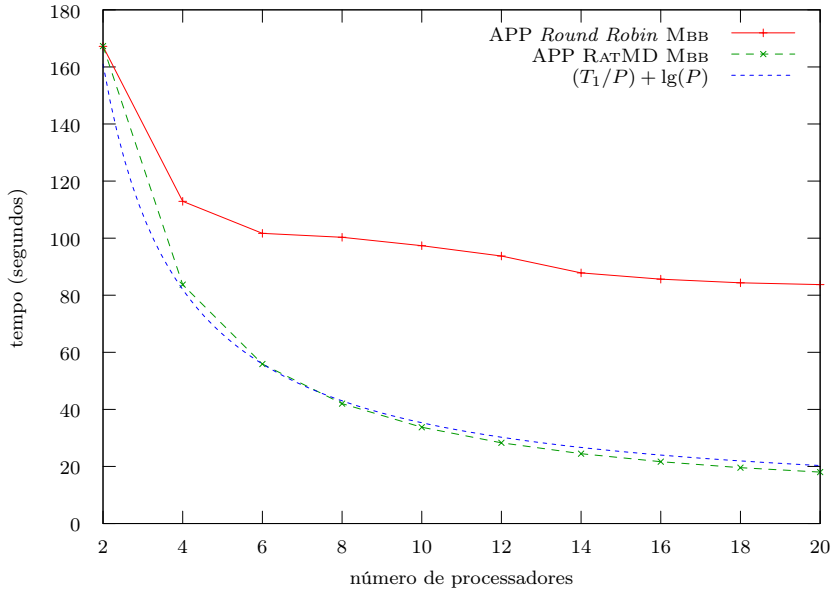
- 14 nós Dell PowerEdge 1950.
- Cada nó tem dois processadores Intel Xeon E5310 Quad Core de 1.60 GHz.
- Cada processador tem 2×4MB de memória *cache* (1066 *Front Side Bus*).
- A interconexão dos nós é feita por um *switch* 3Com modelo 2816.

Todas as execuções foram feitas com 12 nós (96 processadores) com 30 conjuntos de entradas distintas geradas aleatoriamente, onde cada entrada é executada 5 vezes (total de 150 execuções). O único algoritmo a fugir desse padrão é o algoritmo de teste de carga de trabalho, apresentado a seguir, executado trinta vezes, já que não há variação de entrada.

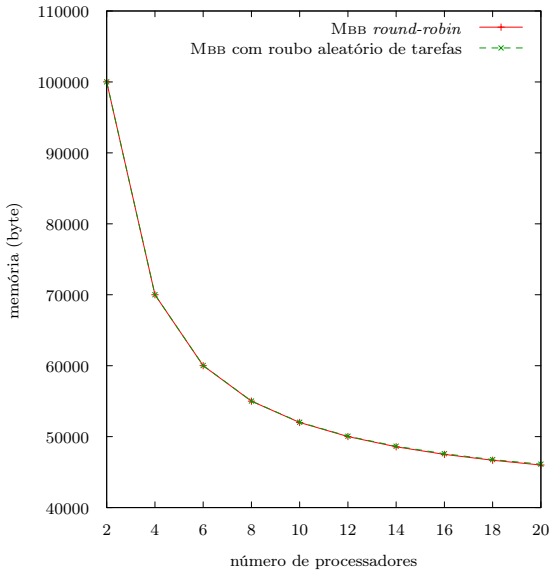
Todas as implementações foram feitas em C++ com MPICH2 1.1 usando o compilador g++ 4.1 com otimizações do conjunto O2.

5.2.1 Teste de Carga de Trabalho

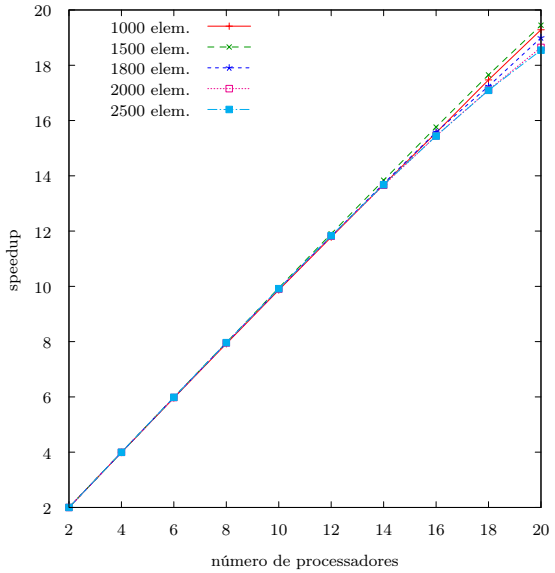
O teste de carga de trabalho é um *benchmark* sintético (*i.e.*, utilizado para evidenciar um desempenho de pico, sem resolver de maneira eficiente um problema real)



(a) Execução do MBB com roubo de tarefas (--×--) contra um MBB *round-robin* (—|—), tomados 2500 elementos. Comparação com o limite teórico apresentado (----).



(b) Alocação de memória para um processo MBB (—×—) contra um MBB *round-robin* (—|—), tomados 2500 elementos.



(c) Speedup próximo ao linear atingido pelo MBB com roubo de tarefas.

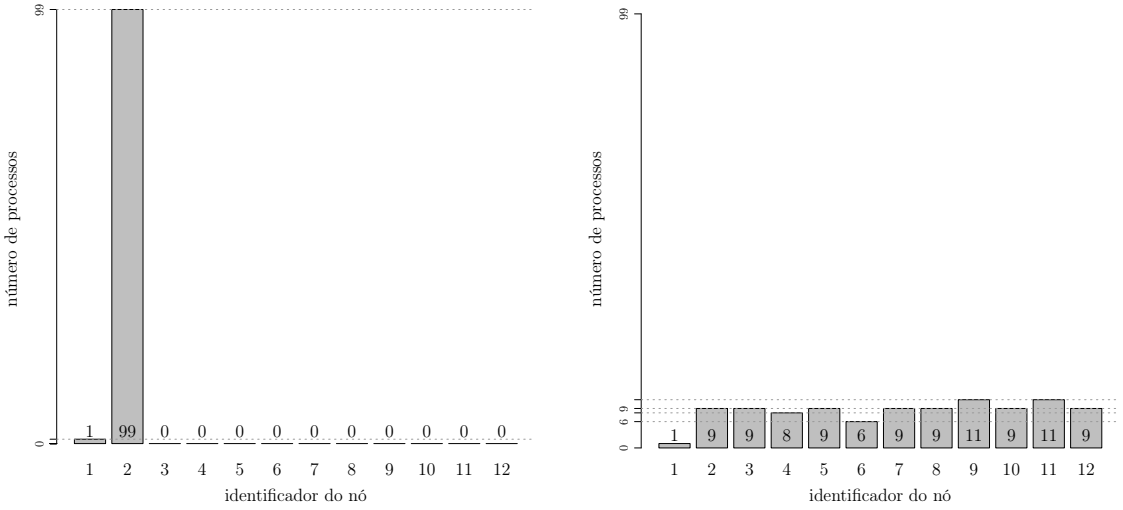
Figura 5.2: Execução do MBB com roubo de tarefas (--×--) contra um MBB *round-robin* (—|—), tomados 2500 elementos. Tempo de execução na Fig. 5.2(a) e consumo de memória na Fig. 5.2(b). *Speedup* na Fig. 5.2(c).

```
#include <cstring>                  // string and vector fast manipulation
#include <iostream>                 // basic stream manipulation
#include <sched.h>                  // for yielding processor on non-blocking
#include <unistd.h>                 // for "sleep()" calling.
#include "mpi.h"                    // message-passing primitives

#define __SPAWN_TASK            0   // tag for spawn parameter
#define __HOSTNAME_MAX_SIZE   1024  // Max size of the hostname in bytes.

using namespace std;                // C++ standard features namespace

int
benchmarkLoadBalance( int numOfSpawns=99, int sleepingTimeInSec=1 )
{

  MPI::Intercomm comm;

  for ( int i=0 ; i < numOfSpawns ; ++ i ) {
    comm = MPI::COMM_WORLD.Spawn (
      "./LoadTask",           // executable name
      MPI::ARGV_NULL,         // argv for exec.
      1,                      // # spawned procs.
      MPI::INFO_NULL,         // spawn param (incl. sched.)
      0,                      // rank of parent
      MPI_ERRCODES_IGNORE );  // error codes
    sleep( sleepingTimeInSec );
  }
  return i;
}

/*--------------------------------------------------------------------------*/

int
main( int argc, char **argv )
{
  MPI::Init();
  printHostAndSleep();
  MPI::Finalize();
  return 0;
}

/*--------------------------------------------------------------------------*/

int
printHostAndSleep ( int sleepingTimeInSec=1 )
{

  char hostname[HOSTNAME_MAX_SIZE];
  gethostname( hostname, HOSTNAME_MAX_SIZE );
  cout << hostname << endl;
        sleep( sleepingTimeInSec );
  return 1;
}
```

Figura 5.3: *Benchmark* sintético para mediar balanceamento em C++. Versão simplificada. `gethostname` retorna no primeiro parâmetro o endereço IP da máquina. O critério é a contagem de impressões de um respectivo endereço IP. A função `main` é a executada por cada processo criado dinamicamente. A tarefa inicial é a função `benchmarkLoadBalance`.

que visa mostrar a correção de desbalanceamento *on-line* fornecido pelo RTMPD. A Fig. 5.3 descreve sucintamente o programa utilizado.

No contexto deste algoritmo não faz sentido falar no modelo de criação de tarefas (*v.g., lazy task creation*); não existem chamadas recursivas. Usando o escalonador nativo do MPICH2 (doravante referenciado como "escalonador MPICH2") o resultado é um desbalanceamento total, como visto na Figura 5.4(a). Ao usar o escalonador com RTMPD, o resultado aproxima-se do balanceamento ótimo, como na Figura 5.4(b).



(a) Sem RTMPD (b) Com RTMPD

Figura 5.4: Balanceamento de carga de processos por nó usando o escalonador MPICH2. 1 processo raiz cria 99 processos dinâmicos. Cada processo dinâmico é criado com intervalo de $1s$ em relação ao anterior e imprime em `stdout` a URL do nó que o está executando. O número sobre cada coluna é a replicação do valor correspondente no eixo vertical.

5.2.2 Série de Fibonacci

Apresenta-se uma versão paralela do algoritmo D&C que calcula o i-ésimo termo na série de Fibonacci. O modelo seguido é o modelo bloqueante. Uma visão simplificada da implementação desse algoritmo pode ser vista nas Fig. 4.4 e 4.5, na Seção 4.4.

i-ésimo termo de Fibonacci funciona como um algoritmo pseudo-sintético; embora existam implementações mais eficientes (CORMEN et al., 2001), o cálculo não é apenas uma maneira de mostrar a performance de pico do escalonador com RTMPD como a Subseção 5.2.1 foi.

O *threshold* paralelo, *i.e.*, o tamanho da entrada para o qual há a troca do algoritmo paralelo para o algoritmo sequencial, é quando $n = 0$ ou $n = 1$. Isso implica que o algoritmo produz 2^i processos em uma execução. A Fig. 5.5(a) mostra as medições para o escalonador MPICH2.

Como o escalonador MPICH2 é determinístico e opera com intervalos fixos de tempo para operações internas com os processos, há baixa variação de medidas de tempo entre 2^1 e 2^9 processos. Para 2^{10} processos, há uma variação de medidas maior

e o tempo é destoante. Em várias execuções houveram travamentos por excesso de processos disparados e resultados inconsistentes, de modo que as medidas para 2^{10} processos devem ser entendidas como uma operação limiar quanto ao número de processos do escalonador MPICH2.

A Fig. 5.5(b) mostra uma execução para o mesmo conjunto de entradas com o escalonador baseado em RTMPD. Ao contrário do cenário anterior, o escalonador consegue atingir até 2^{11} processos ativos sem entrar na situação limiar vista na Figura 5.5(a) (ainda que ambos os escalonadores não consigam executar com um número de processos maior que 2^{11}). Como o esperado, o intervalo de variação das amostras é maior, dado que o escalonador com RTMPD é não-determinístico, ainda que o tempo esperado de execução gire em torno do mesmo valor que o escalonador MPICH2.

Por fim, a Figura 5.5(c) compara os tempos de execução dos dois escalonadores, que se assemelham, a exceção do tempo para 2^{10} processos, por causa da situação limiar descrita anteriormente. O escalonador RTMPD consegue dobrar o número máximo de processos disparados sem comprometimento do desempenho; de fato, para algumas instâncias, o desempenho do RTMPD consegue ser melhor que o do escalonador padrão. A diferença do código utilizado para a implementação MPI-2 usual é apenas o uso da primitiva `MPI_Block_notf`.

5.2.3 Ordenamento por Intercalação (Mergesort)

A Fig 5.6 apresenta uma versão paralela do algoritmo D&C que ordena um vetor de números por intercalação (*Mergesort*). O modelo seguido é o de *lazy task creation*. Por conveniência de análise, supõe-se que o vetor de números que serve como entrada tem 2^d elementos, onde $d \in \mathbb{N} - \{0, 1\}$.

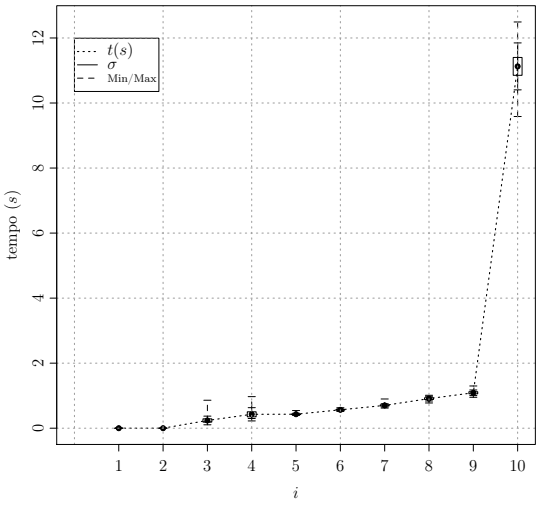
O *threshold* é $d = 8$; se $d < 8$ então executa-se uma versão sequencial. Para uma entrada A com $|A| = 2^d$ o número de processos criados será 2^{d-7} para a versão bloqueante do algoritmo. Considerando que se usa *lazy task creation*, reduz-se o número de processos para 2^{d-8} (a prova é omitida).

A Figura 5.7(a) mostra as medições para o escalonador MPICH2. Como o escalonador MPICH2 é determinístico e opera com intervalos fixos de tempo para operações internas com os processos, há baixa variação de medidas de tempo entre 2^1 e 2^8 processos. A função que mapeia o número de processos dado o valor de n é

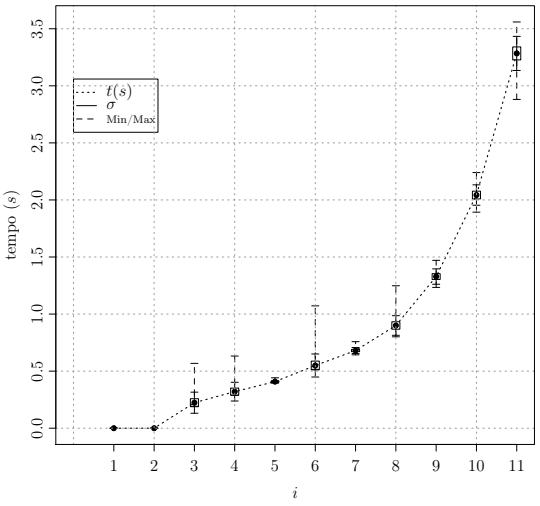
$$2^{(\log_2(n))-8}$$

o que resulta num limite superior para o número de processos próximo à avaliação de Fibonacci, apresentada anteriormente. Nesse caso, não foi atingido o estado limiar de operação como no supracitado *benchmark*. A explicação mais plausível para a redução do número de processos suportados é que a quantidade de processos disponível é diretamente proporcional à quantidade de memória existente para ser alocada. A complexidade espacial do algoritmo *mergesort* $\in \mathrm{O}(|A|)$, enquanto a memória do algoritmo que calcula o i-ésimo termo da série de Fibonacci $\in \mathrm{O}(1)$.

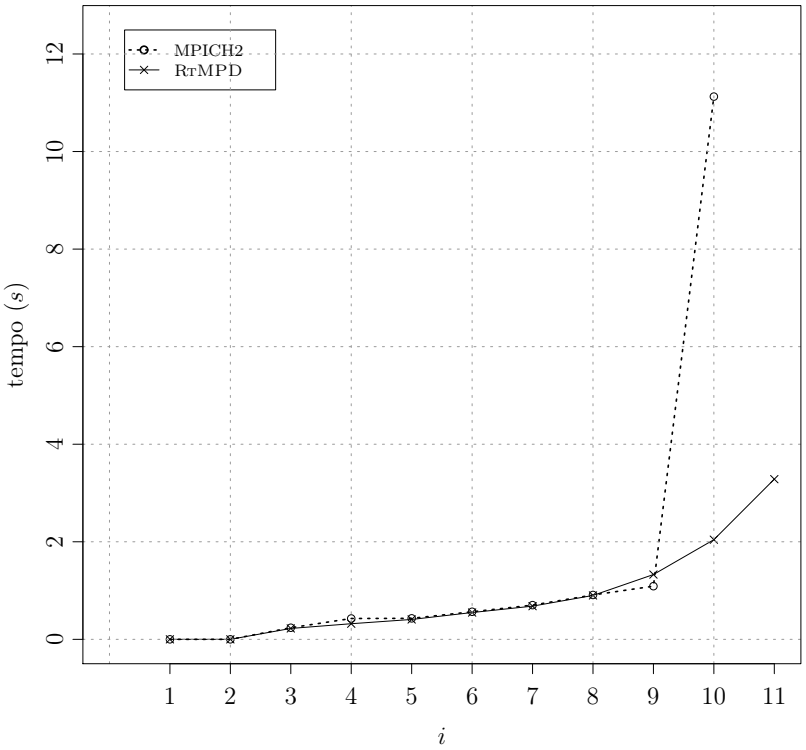
A Figura 5.7(b) mostra a execução do *Mergesort* para o mesmo conjunto de entradas com o escalonador baseado em RTMPD. Ao contrário do cenário anterior, o escalonador consegue atingir até 2^9 processos ativos (*i.e.*, consegue disparar o dobro de processos). Como o esperado, o intervalo de variação das amostras é maior, dado que o escalonador com RTMPD é não-determinístico, ainda que o tempo esperado de execução gire em torno do mesmo valor que o escalonador MPICH2.



(a) Sem RtMPD.



(b) Com RtMPD.

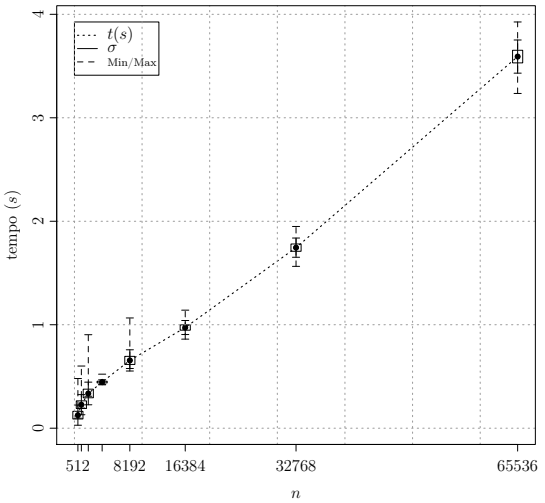


(c) Sobreposição em escala.

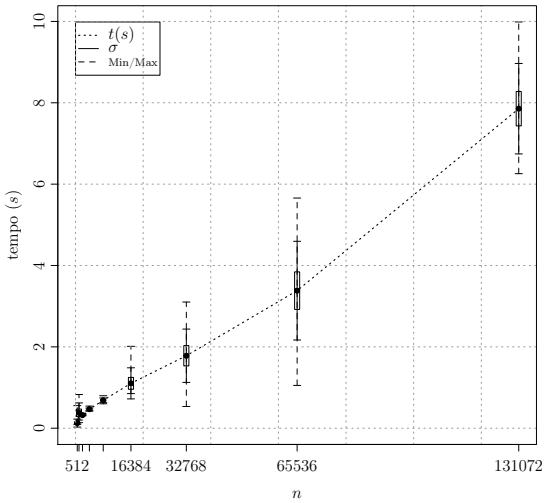
Figura 5.5: Medições para Fibonacci. Tempo de execução t *vs.* índice i para cálculo do i-ésimo elemento da série de Fibonacci. Os pontos pretos representam o tempo de execução médio. A altura dos retângulos delimita o intervalo de valores com 95% de confiança para uma distribuição normal. σ é o desvio padrão e $t(s)$ é a interpolação do tempo de execução. Min/Max são os valores mínimos e máximos do tempo de execução para um dado número de elementos. Resultados para 30 execuções em 96 processadores realizadas com entradas distintas.

```
void Mergesort ::
Mergesort( num_t A[], num_t X[], index_t begin, index_t end )
{
  MPI::Intercomm childComm; // spawned intercommunicator
  MPI::Status status;       // status for Recv

  size_t dataSize;          // sent parameter's size
  int middle;               // auxiliar index for merging

  MPI::Request req[1];      // request for irecv.

  middle = (begin + end - 1)/2;

  /// main recursive calls
  if ( end-begin+1 > 256   ) { // threshold.
    // creates new remote task
    childComm = MPI::COMM_WORLD.Spawn( "./MergesortTask", MPI::ARGV_NULL, 1,
                                       MPI::INFO_NULL, 0, MPI_ERRCODES_IGNORE );
    // size of data being sent as parameter
    dataSize = (end - begin + 1)/2; // size is half of the vector
    // send parameters to start task
    childComm.Send( &(A[begin]), dataSize, MPI::INT, 0, SPAWN_TASK );

    /// does recursive work
    Mergesort(A, X, middle+1, end);

    req[0] = childComm.Irecv( &(A[begin]), dataSize, MPI::NUM, 0, SPAWN_TASK );


    //-------------------------------------------------------------------------
    /// Notifies scheduler of blocking.

    MPI_Block_notf();

    //-------------------------------------------------------------------------

    /// Non-blocking waiting.

    while ( ! MPI::Request::Testall(1, req) )
      sched_yield();

    //-------------------------------------------------------------------------

    // merge data
    merge(A, X, begin, middle, end);

  } else {
    seqMergesort( A, X, begin, end );
  }

  return;
}
```

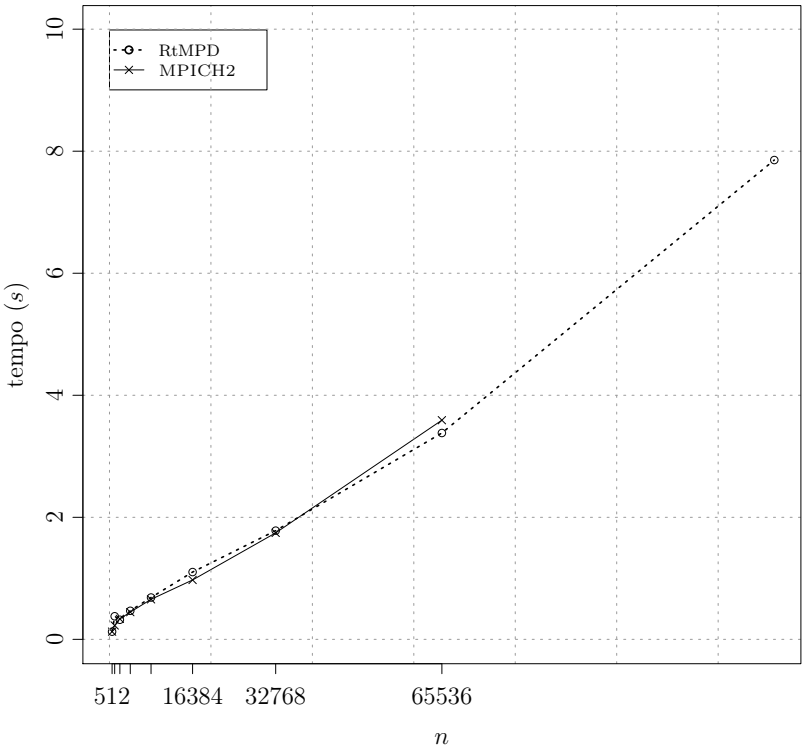
Figura 5.6: Código em C++ para a implementação do algoritmo *Mergesort*. O executável `MergesortTask` é uma função `main` com chamada ao método `Mergesort`. `seqMergesort` realiza o mesmo trabalho, mas sequencialmente. Usa-se *lazy task creation* com degradação sequencial.



(a) Sem RtMPD.



(b) Com RtMPD.



(c) Sobreposição em escala.

Figura 5.7: Medições para *Mergesort*. Tempo de execução t *vs.* número de elementos n a serem ordenados. Os pontos pretos representam o tempo de execução médio. A altura dos retângulos delimita o intervalo de valores com 95% de confiança para uma distribuição normal. σ é o desvio padrão e $t(s)$ é a interpolação do tempo de execução. Min/Max são os valores mínimos e máximos do tempo de execução para um dado número de elementos. Resultados para 30 execuções em 96 processadores, realizadas com entradas distintas.

Por fim, a Figura 5.7(c) compara os tempos de execução dos dois escalonadores, que se assemelham. O comportamento é o mesmo que o observado nas medições para o cálculo do i-ésimo elemento da série de Fibonacci, onde o tempo de execução dos escalonadores é aproximadamente o mesmo, enquanto o tamanho da entrada suportado pelo escalonador com RTMPD é o dobro do escalonador MPICH2. As modificações necessárias são, conforme observado anteriormente, a inserção de `MPI_Block_notf` e a espera não-bloqueante na entrega de resultados.

6 CONCLUSÕES E TRABALHOS FUTUROS

Esse capítulo trata de conclusões sobre os resultados obtidos e apresenta as possibilidades de trabalhos futuros a partir dos limites da Dissertação de Mestrado.

6.1 Conclusões Sobre Resultados

Os resultados atingidos correspondem aos limites teóricos calculados.

A biblioteca no estilo *map-reduce* (CORMEN et al., 2001) para MPI provê escalonamento eficiente para computações D&C (em especial *Branch & Bound*). Houve uma redução no tempo de execução de até 80% em relação ao algoritmo usual para a realização de D&C com MPI, com manutenção do *speedup* próximo ao linear e complexidade espacial idêntica a versão comum.

Esse resultado mostra que o algoritmo utilizado, RatMD, é uma solução de compromisso entre a eficiência do algoritmo Rat e as limitações do emprego deste em memória distribuída. O ganho é observado mesmo em um ambiente de processadores homogêneos, que tendem a provocar um desbalanceamento de carga de trabalho inferior ao de processadores heterogêneos.

Um código D&C MPI qualquer pode ser escalonado pelo RatMD com a inserção de código em tempo de compilação, sendo independente da implementação MPI utilizada. No entanto, a inserção das primitivas de escalonamento não é totalmente transparente ao programador; fica a cargo deste especificar o que é uma tarefa, sua granularidade e sua segmentação em sub-tarefas.

A modificação do escalonador MPICH2 utilizado no tratamento aos processos originados pelo `MPI_Comm_spawn` também satisfaz as observações de predição da execução apresentada.

O fator crucial que possibilita o ganho observado é a adição, ao programador, da responsabilidade de alertar ao MPD quando o programa entra em estado de bloqueio, via `MPI_Block_notf`. Essa notificação é feita de maneira não bloqueante para evitar que processos fizessem *busy waiting*.

Mantendo-se basicamente o mesmo desempenho conseguiu-se dobrar o limite de processos executados simultaneamente pelo gerente de processos MPD. Através do ajuste do *threshold* pode-se regular a execução de uma dado algoritmo para transformar o ganho na complexidade espacial em ganho no tempo de execução, embora essa transformação possivelmente não vá ocorrer sem perdas.

6.2 Conclusões em Relação ao Contexto Científico

O trabalho apresentado é inédito na literatura ao promover uma biblioteca com escalonamento eficiente para D&C em MPI-1, embora escalonamento D&C em ambientes de memória distribuída não sejam inéditos. O KAAPI (GAUTIER; BESSERON; PIGEON, 2007) escalona processos em memória distribuída de maneira eficiente, com roubo de tarefas, baseado em critérios que já haviam se provado eficiente na biblioteca Java SATIN (NIEUWPOORT; KIELMANN; BAL, 2001). Mesmo o Cilk (RANDALL, 1998) possuiu propostas de extensões para memória distribuída (BLUMOFE; LISIECKI, 1996), embora essas extensões sejam voltadas à tolerância a falhas e jamais tenham sido implementadas de maneira confiável. Ainda assim, a arquitetura aberta e o algoritmo com critério diverso ao de localidade (KAAPI e SATIN), além do fato de ser a única implementação genérica para a especificação MPI, conservam o ineditismo e valor acadêmico do trabalho.

Da mesma maneira que a biblioteca apresentada anteriormente, a modificação feita junto ao MPICH2, com os resultados obtidos, preenche lacunas deixadas por outros trabalhos. AMPI (HUANG; LAWLOR; KALé, 2004) provê escalonamento de processos MPI em tempo de execução mas, ao contrário deste trabalho, não trata processos criados dinamicamente por `MPI_Comm_spawn`. A implementação OpenMPI (GABRIEL et al., 2004), aguardada com ansiedade pela comunidade acadêmica, promete um escalonador eficiente para os processos, mas, até a presente data, não há implementação deste escalonador, tampouco uma especificação com o nível de formalidade como a apresentada neste trabalho.

Com relação ao trabalho presente no GPPD, a implementação apresentada de RtMPD pode ser considerada a versão distribuída do *daemon* escalonador proposto em (CERA et al., 2006), com uma eficiência próxima em um número pequeno de processos e mais eficiente quando o número de processos cresce, visto que não possui um gargalo centralizado. Com relação ao trabalho desenvolvido em (PEZZI et al., 2007), o diferencial apresentado é que a abordagem desenvolvida desatrela o algoritmo de roubo de tarefas da definição de tarefa, valendo-se apenas da inserção de `MPI_Block_notf` e do uso de recepção de mensagens não bloqueante; ao contrário da WS hierárquico, o programador não precisa reescrever todo o código em função da árvore de gerentes utilizada, já que a estrutura da árvore de gerentes é implicitamente substituída pelo anel do MPICH2. Além disso, o ganho apresentado refere-se à complexidade espacial do algoritmo em C++, enquanto o ganho observado com o HWS é no tempo de execução quando comparado à implementação Java SATIN.

6.3 Trabalhos Futuros

Esta sessão traz uma coleção de tópicos derivados desta Dissertação que estão em um estágio inicial e devem dar origem a novos trabalhos sobre o tema.

6.3.1 Questões em Aberto

Existem pelo menos quatro questões em aberto no presente trabalho pertinentes para o escalonamento eficiente em memória distribuída usando MPI:

1. modelagem formal de programas MPI dinâmicos;

2. uso de *threads* para tratamento assíncrono de mensagens;

3. sincronização do acesso às *deque*; e

4. migração de processos e *checkpointing*.

6.3.1.1 Modelagem Formal de Programas MPI Dinâmicos

A modelagem da primitiva *fork* baseada na primitiva `MPI_Comm_spawn` mostrou-se suficiente para a construção do escalonador, ainda que deficiente em uma análise mais formal do modelo MPI. O que é mostrado a seguir é uma proposta de modelagem a ser desenvolvida futuramente.

A observação-chave é que cada processo precisa executar apenas um par de *send*/*receive* por filho; a entrada sobre o qual o filho trabalhará e a saída produzida por esse filho. A primeira abordagem para modelar um *spawn* seria considerar que essa primitiva, além de criar um novo processo, também envia entradas para esse. Nesse caso, a invocação da primitiva deve ser $spawn(\mu)$, onde μ é a entrada a ser enviada.

A abordagem referida acima funcionaria na criação do filho, mas não é o suficiente; um filho não pode enviar dados de volta ao pai através de outro $spawn(\mu)$. Dessa maneira, pode-se modelar a primitiva $send(\mu, i)$, que envia os dados μ para o processo Γ_i e é não-bloqueante.

Dado que o modelo D&C apresentado permite apenas comunicação entre pais e filhos, é desejável que o i (índice do processo-destino dos dados) acima tenha algumas restrições. Ao invés de ser um identificador global de processos, i deve referir-se apenas a um processo previamente criado por *spawn* e esse deve ser todo conhecimento que um processo deve ter do restante do ambiente de execução. Isso pode ser alcançado substituindo-se a primitiva *spawn* com uma primitiva *cspawn*, cuja única diferença para a primitiva original é que esta retorna um *comunicador* em ambos o processo pai e o processo filho. O comunicador é uma estrutura abstrata de dados que identifica um dado processo. Desta maneira, i deve ser necessariamente um comunicador para que *send* funcione.

No cenário proposto, quando um filho acaba de computar, este deve enviar os dados computados para o processo-pai. Para obter esses dados deve-se chamar $receive(\mu, i)$, significando o recebimento dos dados μ do processo identificado pelo comunicador i. *Receive* é bloqueante e substitui o uso de *sync*. Como um *receive* deve saber para quem enviar os dados, todas as chamadas a *cspawn* devem enviar, junto com os dados a serem mandados, o comunicador que representa o processo pai, a ser usado pelo *send* do processo filho. A Figura 6.1 mostra um DAG modificado com apenas dois processos a usarem as primitivas descritas até aqui. As linha tracejadas inseridas representam modificações em relação ao DAG apresentado na Figura 2.2, com as sub-instruções que compõem as novas primitivas apresentadas explicitadas.

6.3.1.2 Uso de Threads para o Tratamento Assíncrono de Mensagens

No Capítulo 4 foi mencionado que MPICH2 era uma boa escolha como base de implementação pois suportava nativamente *threads* no seu fluxo de execução; *threads* podem otimizar o acesso às estruturas de dados que representam tarefas. No entanto, esse fato não é explorado pelo escalonador com RtMPD, que usa como uma de suas bases o fato de o acesso às principais estruturas de dados ser síncrono. A seguir apresentam-se os conceitos iniciais para a modificação futura do escalonador com o uso de *threads*.

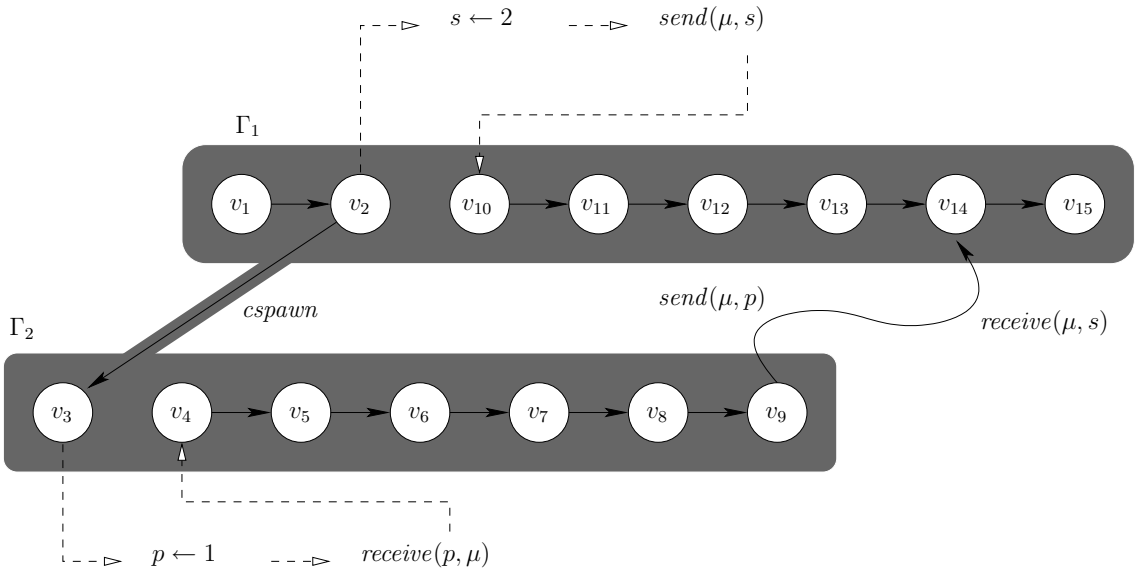


Figura 6.1: DAG modificado com primitivas MPI modeladas para programas D&C (versão simplificada).

Enquanto um ambiente com memória compartilhada permite o compartilhamento passivo de dados, uma máquina com memória distribuída não o faz; um *receive* explícito deve ser usado toda a vez em que algum processo precise de algum dado que pertence a outro processo. Isso implica em qualquer tentativa de roubo contra um processo ter de parar a atual atividade atual do alvo ou esperar que este acabe de processar. Ambos os casos induzem a uma sincronização desnecessária, pois as tarefas não seriam mais "roubadas".

Como solução, estende-se o modelo anteriormente proposto de exatamente um processo por processador para que se permitam exatamente dois processos por processador; um processo executa o caminho crítico da aplicação enquanto o outro age como um "gerente de processo". Cada gerente comunica-se com outros gerentes através do uso de primitivas *send*/*receive*. Embora o *send* permaneça o mesmo, *receive* deve tornar-se universal, o que significa que deve aceitar mensagens de qualquer outro processo. Para diferenciar esse *receive* do normal, esse será chamado *ureceive*. Destaca-se que *ureceive* retém a informação de quem foi o processo que enviou a mensagem de roubo e armazena isso em τ, porque o protocolo demanda que alguma resposta seja enviada ao requisitante em tempo finito. O gerente pode acessar a *deque* de tarefas e executa continuamente RtMPD (ou mesmo RatMD). Além disso, o gerente tem uma *thread* "sentinela" que trata os casos de processos que terminam ou que bloqueiam, fazendo *spawn* e desbloqueando, enquanto outra *thread* faz o gerenciamento de tentativas de roubo. Como ambas compartilham a *deque* e memória compartilhada, não há necessidade de interromper a *thread* que trabalha no caminho crítico e, portanto, o roubo de tarefas passa a se comportar como no caso de memória compartilhada.

Destaca-se que a solução proposta funciona apenas em máquinas *multi-core*, capaz de processamento paralelo total. A Figura 6.2 representa esta solução.

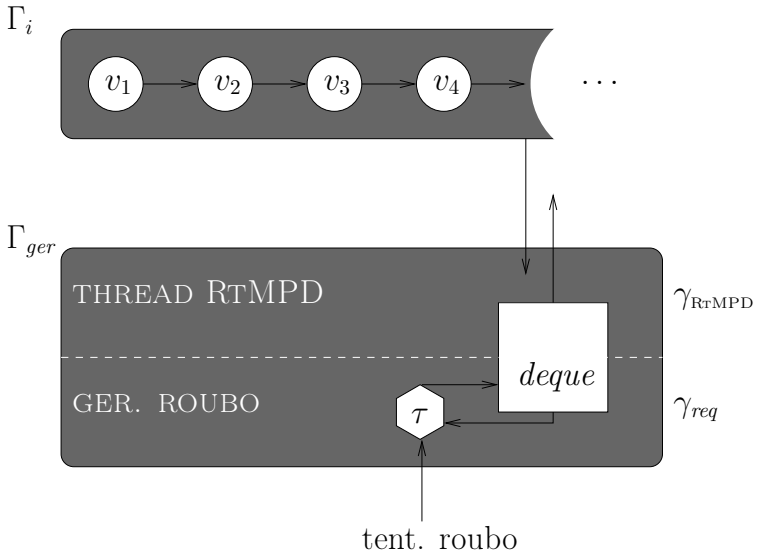


Figura 6.2: Uso otimizado de *threads* para garantia de desempenho. O octágono branco age como *ureceive*/*send*. τ guarda a identidade do requisitante. Γ_i representa o processo i e Γ_{ger} representa o processo gerente correspondente. γ_{RTMPD} representa a *thread* de caminho crítico, enquanto γ_{req} representa a *thread* que atende requisições. Agora, Γ é um processo e γ uma *thread*, alterando-se a notação estabelecida até o momento.

6.3.1.3 Sincronização do Acesso às Deque

O trabalho na linguagem Cilk (BLUMOFE et al., 1995) traz uma questão interessante sobre tentativas de roubo de tarefas; para que haja garantias de performance do RAT cada requisição de roubo deve entregar uma resposta em tempo constante, sendo irrelevante se essa requisição frutífera ou não. Com técnicas de sincronização de dados clássicas (*v.g.*, mutex, semáforo, monitor, *etc.*) não há como se garantir esse tempo constante.

Como solução, (BLUMOFE et al., 1995) propõe um protocolo chamado THE que, em resumo, não bloqueia quando do acesso a variáveis sincronizadas; na presença de um *lock*, uma resposta é retornada imediatamente, indicando que a requisição falhou. O protocolo THE pode ser emulado no cenário de memória distribuída, quando se usa sincronização para a solução de compartilhamento da *deque* apresentado na Sub-subseção apresentada logo acima.

6.3.1.4 Migração de Processos e Checkpointing

A primeira clausula do RAT diz que quando quer que uma nova tarefa seja criada, a *thread* atual Γ_a vai para o topo da *deque* e a tarefa recém-criada Γ_b começa a executar no processador. Passando de *threads* para processos, fazer a migração (ou *checkpointing*) de um processo é bastante complexo e oneroso em termos de desempenho. Os contornos iniciais desse problema são mostrados a seguir.

Uma abordagem para superar o problema é usar o conceito de *processos de continuidade*. Considerando o Algoritmo 2.1 modificado com notação para uso de *send*/*receive*/*cspawn* apresentado anteriormente, para uma ramificação de três processos, apresenta-se o Algoritmo 6.1:

No RAT, v_1 e v_2 executam sequencialmente e após, na linha 3, RATDISTMEN

Algoritmo 6.1 RatDistMen()

```
1: v_1
2: v_2
3:  a ← cspawn t_1
4:  b ← cspawn t_2
5:  c ← cspawn t_3
6: para i ← a to c faça
7:     receive(μ_i, i)
8:  ProcessaResultados(μ_a, μ_b, μ_c)
```

vai para o topo da *deque* e t_1 começa a executar. Quando RatDistMen começa a executar novamente (no processador local ou remoto), deve começar executando a linha 4.

A seguinte estratégia pode ser desenvolvida em trabalhos futuros:

1. Tudo que sucede o *receive* é tratado como uma única tarefa.
2. Após a linha 3 se faz um *spawn* do resto do código como uma única tarefa, que é a execução sequencial das demais tarefas.
3. Funciona recursivamente para os *cspawn* restantes.

A última tarefa, criada artificialmente, será chamada de "tarefa de continuação" (RANDALL, 1998). Ela difere das outras porque não pode executar até que todas as outras tarefas já tenham executado. Desse modo, sempre que essa tarefa é executada, ela bloqueia e espera até que esteja apta a continuar. Essa tarefa não pode ser alvo de roubo, devido às suas dependências locais ligadas aos processos MPI (NEVES et al., 2007). Contornar as limitações de migrações de processos comunicantes deve ser o ponto de partida desta ramificação do trabalho.

6.3.2 Escalonamento em Dois Níveis

O Capítulo 2 mostra técnicas para escalonamento em dois níveis (*threads* e processos). O trabalho desenvolvido em (LIMA; MAILLARD, 2008) apresenta ganhos de desempenho expressivos quando há um sistema de controle de granularidade de tarefas em MPI. Graças às tecnologias *Simultaneous Multiprocessors* (SMP) e *Simultaneous Multithread* (SMT) (NAVAUX; DEROSE, 2003) tem-se (em conformidade com (FLYNN, 1972)) multi-computadores MIMD (clusters) de multiprocessadores MIMD (*chip multi-core*) onde, possivelmente, cada processador é também uma máquina MISD (*multithread*, memória compartilhada).

A teoria clássica de escalonamento trata da alocação de *tarefas* em *recursos* (1 nível) (CASAVANT; KUHL, 1994). O cenário atual, todavia, evidencia a necessidade de escalonamento multi-nível. *v.g.*, a Figura 6.3 ilustra um possível cenário de escalonamento multi-nível.

Ao utilizar as ideias para escalonamento D&C eficiente apresentadas nesse trabalho, pode-se prover a função $\mathcal{X}_2$ da Figura 6.3 para esses problemas bem definidos. Uma vez que seja provida a função $\mathcal{X}_1$ integrada ao modelo MPI prove-se uma implementação do padrão com alta escalabilidade e desempenho. Em geral a função $\mathcal{X}_0$ é uma função identidade, visto que não há na literatura uma diferenciação clara

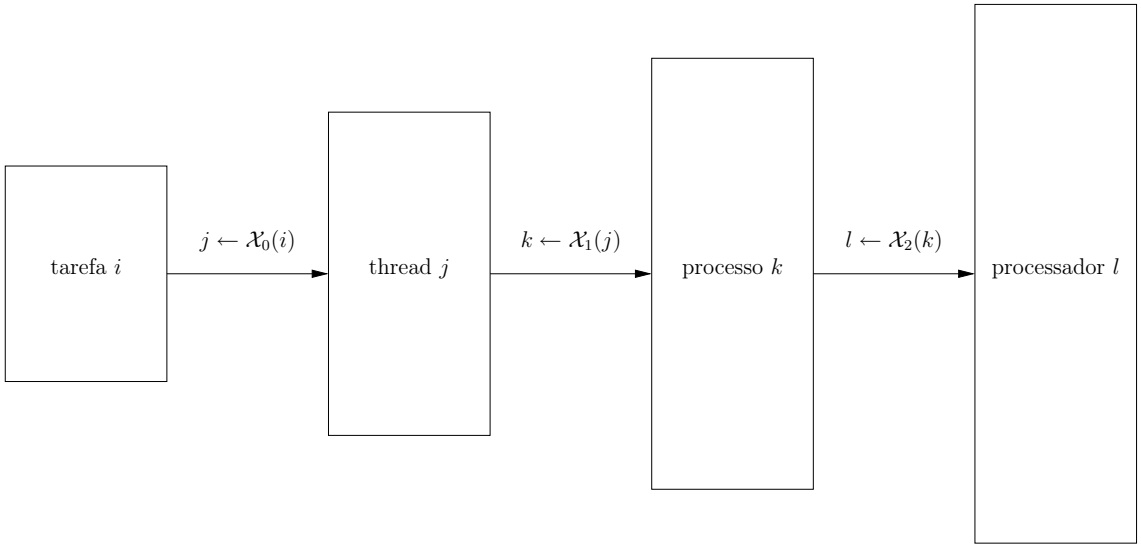


Figura 6.3: Escalonamento multi-nível. As setas significam "é escalonado em". O tamanho dos blocos é proporcional à granularidade de um componente em relação ao próximo. $\mathcal{X}_d$ é uma função de escalonamento, onde índice d representa o nível onde o escalonamento é feito na hierarquia. Todos os relacionamentos são do tipo 1:1.

da melhor abordagem para se adaptar uma tarefa a uma *thread* ou processo. O propósito prático em apresentar esse tipo de combinação é prover suporte total do compilador às diferentes configurações que um *cluster* oferece A Figura 6.4 apresenta a pilha arquitetural a ser desenvolvida e implementada em um trabalho futuro.

6.3.3 Escalonamento MPI como um *Backend*

Na subseção anterior, a Figura 6.4 apresentou uma estrutura para controle de granularidade do escalonamento entre *threads* e processos em um *cluster multi-core*. Esse conceito pode ser estendido, de modo a se planejar uma arquitetura de computação paralela onde se abstraia o *hardware* em tempo de compilação. A Figura 6.5 ilustra esse conceito. Nesse contexto, o trabalho desenvolvido com MPI atua como um *backend* de compilação, uma linguagem-alvo a ser compilada de acordo com o *hardware* com o qual o compilador opera, de maneira transparente ao programador.

Esse trabalho é desenvolvido a longo prazo pelo GPPD/UFRGS. Os próximos passos são definir que tipo de modelo, biblioteca e linguagem as camadas superiores empregarão. Existem trabalhos em andamento para o uso de refatoração na construção de uma linguagem intermediária (chamada, como protótipo, de Linguagem Δ) capaz de expressar com robustez e genericamente o modelo de computação paralela com troca de mensagens, que acredita-se ser o mais adequado para emprego de paralelismo. Também existem trabalhos para o uso de *auto-tunners*, *i.e.*, pequenos *benchmarks* realizados em tempo de compilação para determinar qual o *backend* mais adequado ao *hardware* sendo utilizado. Essa organização é a visão de mais alto-nível sobre os rumos que o trabalho tomará no futuro. Com os resultados obtidos pela pesquisa a ser desenvolvida apresentada na Subseção 6.3.2, espera-se montar um *backend* altamente eficiente nesse contexto.

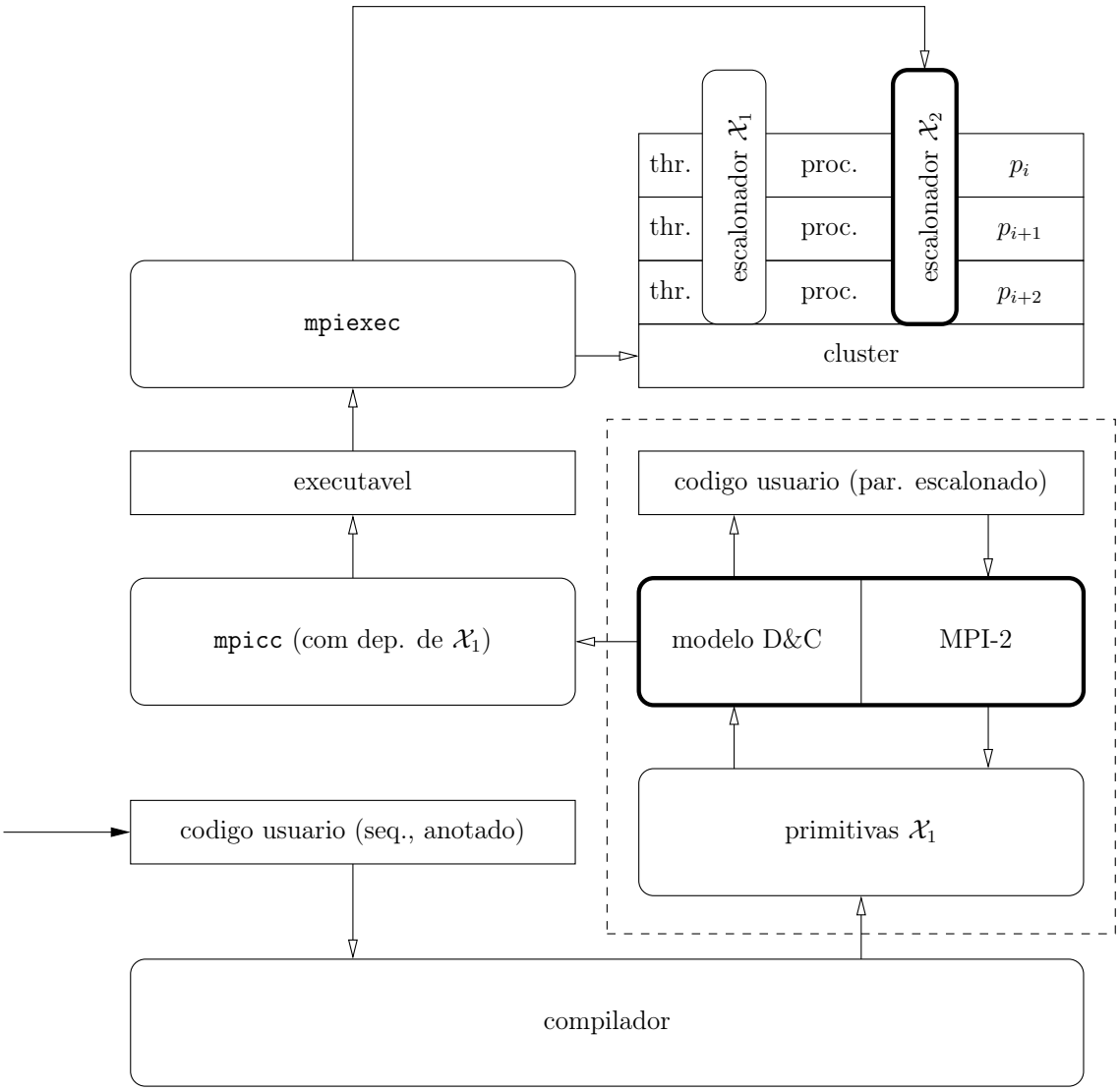


Figura 6.4: Arquitetura para escalonamento em dois níveis. Seta com ponta preta representa a inserção de código anotado pelo usuário, enquanto as setas com pontas brancas representam as interações entre os componentes da arquitetura. Retângulos com ângulos retos são componentes opacos ao usuário, enquanto retângulos com cantos arrendondados são componentes cuja construção é transparente ao usuário. O retângulo de linha mais grossa representa o trabalho desenvolvido nesta dissertação.

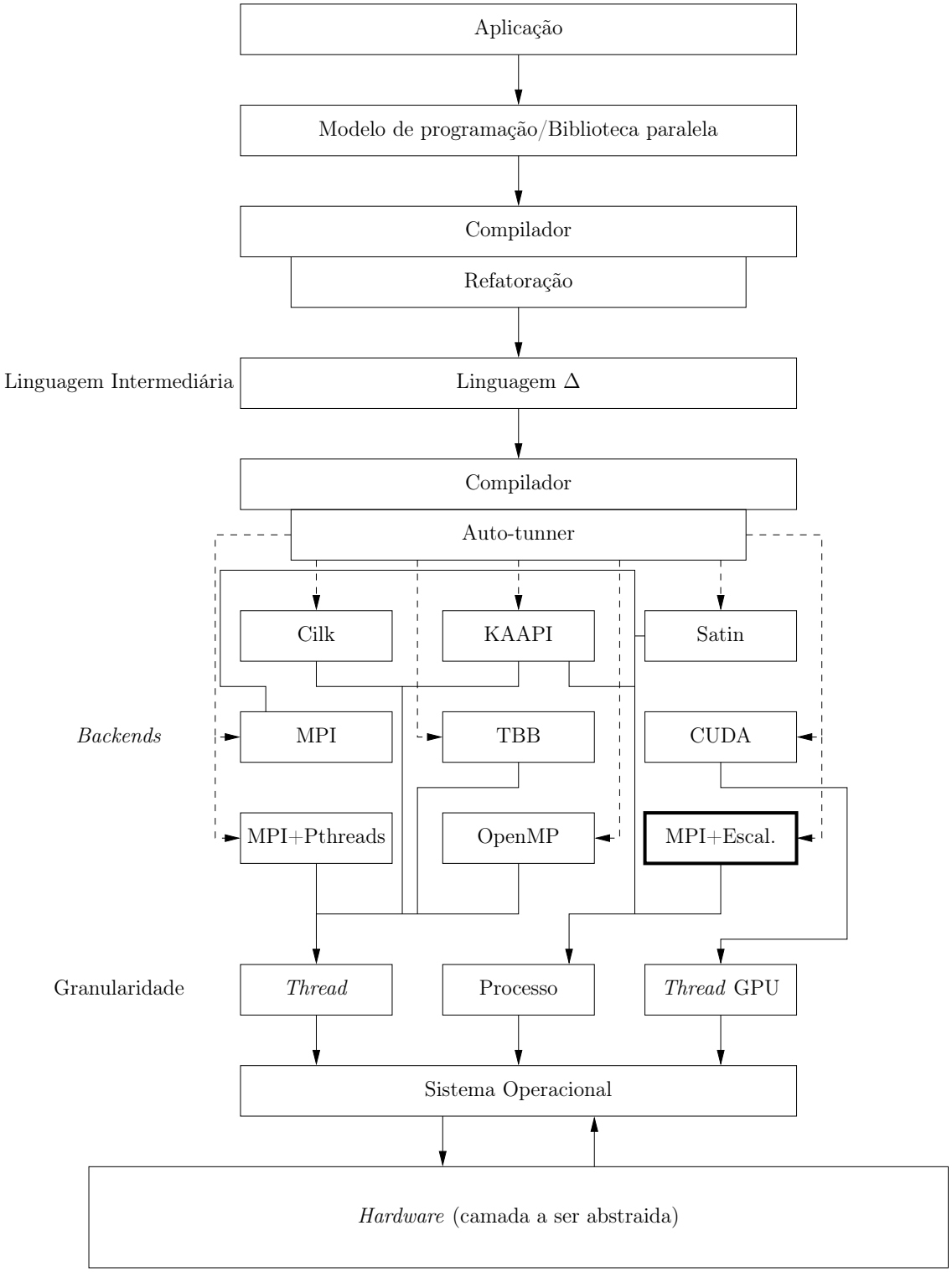


Figura 6.5: Esquema de abstração de arquitetura para Programação Paralela. Cada retângulo representa uma entidade. Retângulos adjuntos representam a relação componente (retângulo maior) – subcomponente (retângulo menor) relevantes para a discussão. A seta com linha sólida representa que a saída de um retângulo é processada como a entrada de outro ou uma estrutura derivada obtida por processamento anterior. A seta com linha tracejada representa um conjunto de testes que algum componente realiza sobre outro. O retângulo de linha mais grossa representa o trabalho desenvolvido nesta dissertação.

REFERÊNCIAS

AGRAWAL, K.; FINEMAN, J. T.; SUKHA, J. Nested Parallelism in Transactional Memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP), 13., 2008, Salt Lake City, Utah, USA. **Proceedings...** [S.l.: s.n.], 2008.

AGRAWAL, K.; HE, Y.; HSU, W. J.; LEISERSON, C. E. Adaptive Task Scheduling with Parallelism Feedback. In: ANNUAL ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP), 2006, New York City, NY, USA. **Proceedings...** [S.l.: s.n.], 2006.

AGRAWAL, K.; HE, Y.; LEISERSON, C. E. Adaptive Work Stealing with Parallelism Feedback. In: SUBMITTED FOR PUBLICATION, 2006. **Anais...** [S.l.: s.n.], 2006.

AGRAWAL, K.; HE, Y.; LEISERSON, C. E. An Empirical Evaluation of Work Stealing with Parallelism Feedback. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (ICDCS), 2006. **Proceedings...** [S.l.: s.n.], 2006.

AGRAWAL, K.; LEISERSON, C. E.; SUKHA, J. Memory Models for Open-Nested Transactions. In: ACM SIGPLAN WORKSHOP ON MEMORY SYSTEMS PERFORMANCE AND CORRECTNESS (MSPC), 2006. **Proceedings...** [S.l.: s.n.], 2006.

ARORA, N. S.; BLUMOFE, R. D.; PLAXTON, C. G. Thread Scheduling for Multiprogrammed Multiprocessors. In: SPAA '98: PROCEEDINGS OF THE TENTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 1998, New York, NY, USA. **Anais...** ACM, 1998. p.119–129.

ASANOVIC, K.; BODIK, R.; DEMMEL, J.; KEAVENY, T.; KEUTZER, K.; KUBIATOWICZ, J.; MORGAN, N.; PATTERSON, D.; SEN, K.; WAWRZYNEK, J.; WESSEL, D.; YELICK, K. A View of the Parallel Computing Landscape. **Communications of the ACM**, [S.l.], v.52, n.10, Outubro 2009.

BENDER, M. A.; RABIN, M. O. Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds. In: SPAA '00: PROCEEDINGS OF THE TWELFTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 2000, New York, NY, USA. **Anais...** ACM, 2000. p.13–21.

BENDER, M. A.; RABIN, M. O. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. **Theory of Computing Systems Special Issue on SPAA00**, [S.l.], v.35, p.289–304, 2002.

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

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling Multithreaded Computations by Work Stealing. In: IN PROCEEDINGS OF THE 35TH ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE (FOCS), 1994. **Anais...** [S.l.: s.n.], 1994. p.356–368.

BLUMOFE, R. D.; LEISERSON, C. E. Space-Efficient Scheduling of Multithreaded Computations. **SIAM Journal on Computing**, [S.l.], v.27, n.1, p.202–229, Feb. 1998.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling Multithreaded Computations by Work Stealing. **J. ACM**, New York, NY, USA, v.46, n.5, p.720–748, 1999.

BLUMOFE, R. D.; LISIECKI, P. A. Adaptive and Reliable Parallel Computing on Networks of Workstations. In: USENIX 1997 ANNUAL TECHNICAL SYMPOSIUM, 1996. **Proceedings...** [S.l.: s.n.], 1996.

BRENT, R. P. The Parallel Evaluation of General Arithmetic Expressions. **J. ACM**, New York, NY, USA, v.21, n.2, p.201–206, 1974.

BRUCKER, P. **Scheduling Algorithms**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

BURNS, G.; DAOUD, R.; VAIGL, J. LAM: An Open Cluster Environment for MPI. In: SUPERCOMPUTING SYMPOSIUM, 1994. **Proceedings...** [S.l.: s.n.], 1994. p.379–386.

BURTON, F. W.; SLEEP, M. R. Executing Functional Programs on a Virtual Tree of Processors. In: FPCA '81: PROCEEDINGS OF THE 1981 CONFERENCE ON FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, 1981, New York, NY, USA. **Anais...** ACM, 1981. p.187–194.

BUTENHOF, D. R. **Programming with POSIX Threads**. [S.l.]: Addison-Wesley Professional, 1997.

CASAVANT, T. L.; KUHL, J. G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. In: **Readings in Distributed Computing Systems**. Los Alamitos, CA: IEEE Computer Society Press, 1994.

CERA, M. C.; MAILLARD, N.; NAVAUX, P. O. A. A Centralized and On-line Scheduling Solution to Dynamic MPI Programs. **V Workshop de Processamento Paralelo e Distribuído (WSPPD 2007)**, Porto Alegre, Brasil, p.25–30, 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. **Lecture Notes in Computer Science - 13h European PVM/MPI Users Group Meeting**, Bonn, Germany, v.4192/2006, p.247–255, Sept. 2006.

CHENG, G.; FENG, M.; LEISERSON, C. E.; RANDALL, K. H.; STARK, A. F. Detecting Data Races in Cilk Programs that Use Locks. In: TENTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES (SPAA '98), 1998, Puerto Vallarta, Mexico. **Proceedings...** [S.l.: s.n.], 1998. p.298–309.

CORMEN, H. T.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms, Second Edition**. [S.l.]: The MIT Press, 2001.

CUNG, V. D.; DANJEAN, V.; DUMAS, J.; GAUTIER, T.; HUARD, G.; RAFFIN, B.; RAPINE, C.; ROCH, J.; TRYSTRAM, D. **Adaptive and Hybrid Algorithms**: classification and illustration on triangular system solving. Granada, Spain, 2006. 131–148p.

DAILEY, D.; LEISERSON, C. E. Using Cilk to Write Multiprocessor Chess Programs. **The Journal of the International Computer Chess Association**, [S.l.], 2002.

DANAHER, J. S.; LEE, I. A.; LEISERSON, C. E. The JCilk Language for Multithreaded Computing. In: SYNCHRONIZATION AND CONCURRENCY IN OBJECT-ORIENTED LANGUAGES (SCOOL), 2005, San Diego, California. **Anais...** [S.l.: s.n.], 2005.

DANAHER, J. S.; LEE, I. A.; LEISERSON, C. E. Programming with Exceptions in JCilk. **Science of Computer Programming (SCP)**, [S.l.], v.63, n.2, p.147–171, Dec. 2006.

DIJKSTRA, E. W. Solution of a Problem in Concurrent Programming Control. **Commun. ACM**, New York, NY, USA, v.8, n.9, p.569, 1965.

FELDMANN, R.; MYSLIWIETZ, P.; MONIEN, B. A Fully Distributed Chess Program. **Advances in Computer Chess VI**, [S.l.], p.1–27, 1991.

FENG, M.; LEISERSON, C. E. Efficient Detection of Determinacy Races in Cilk Programs. In: NINTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES (SPAA), 1997, Newport, Rhode Island. **Proceedings...** [S.l.: s.n.], 1997. p.1–11.

FLYNN, M. J. Some Computer Organization and Their Effectiveness. In: **IEEE Transactions on Computers**. [S.l.: s.n.], 1972. v.C-21, n.9.

FORUM, M. P. I. **MPI**: A Message-Passing Interface Standard. Knoxville, Tennessee: University of Tennessee, 1994. (CS-94-230).

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

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. **SIGPLAN Not.**, New York, NY, USA, v.33, n.5, p.212–223, 1998.

GABRIEL, E.; FAGG, G. E.; BOSILCA, G.; ANGSKUN, T.; DONGARRA, J. J.; SQUYRES, J. M.; SAHAY, V.; KAMBADUR, P.; BARRETT, B.; LUMSDAINE, A.; CASTAIN, R. H.; DANIEL, D. J.; GRAHAM, R. L.; WOODALL, T. S. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. **Lecture Notes in Computer Science - 13h European PVM/MPI Users Group Meeting**, Bonn, Germany, v.3241/2004, p.97–104, Nov. 2004.

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

GEBREMICHAEL, B.; VAANDRAGER, F.; ZHANG, M. Analysis of The ZeroConf Protocol Using UPPAAL. In: EMSOFT '06: PROCEEDINGS OF THE 6TH ACM & IEEE INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, 2006, New York, NY, USA. **Anais...** ACM, 2006. p.242–251.

GROPP, W. MPICH2: A New Start for MPI Implementations. **Lecture Notes in Computer Science - 9th European PVM/MPI Users Group Meeting**, Linz, Austria, v.2474/2002, p.37–42, Sept. 2002.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI - Portable Parallel Programming with Message-Passing Interface**. 2.ed. Massachusetts Institue of Technology - Cambridge, Massachusetts 02142: The MIT Press, 1999. (Scientific and Engineering Computation Series).

GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2**: advanced features of the message-passing interface. Massachusetts Institue of Technology - Cambridge, Massachusetts 02142: The MIT Press, 1999. (Scientific and Engineering Computation Series).

HALBHERR, M.; ZHOU, Y.; JOERG, C. F. MIMD-Style Parallel Programming with Continuation-Passing Threads. In: INTERNATIONAL WORKSHOP ON MASSIVE PARALLELISM: HARDWARE, SOFTWARE, AND APPLICATIONS, 2., 1994, Capri, Italy. **Proceedings...** [S.l.: s.n.], 1994.

HE, Y.; HSU, W.; LEISERSON, C. E. Provably Efficient Two-Level Adaptive Scheduling for Multithreaded Jobs on Parallel Computers. In: IN JSSPP, SAINT-MALO, 2006. **Anais...** JSSPP, 2006.

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

HUANG, C.; ZHENG, G.; KUMAR, S.; KALé, L. V. Performance Evaluation of Adaptive MPI. In: PPOPP '06: PROCEEDINGS OF THE ELEVENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Anais...** ACM Press, 2006. p.12–21.

JAJA, J. **An Introduction to Paralell Algorithms**. [S.l.]: Addinson-Wesley, 1992.

JENKS, S. Multithreading and Thread Migration Using MPI and Myrinet. In: PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS CONFERENCE, 2004, Cambridge, MA. **Proceedings...** [S.l.: s.n.], 2004.

JENKS, S.; GAUDIOT, J. A Multithreaded Runtime System With Thread Migration for Distributed Memory Parallel Computing. In: HIGH PERFORMANCE COMPUTING SYMPOSIUM, 2003, Orlando, FL. **Anais...** [S.l.: s.n.], 2003.

JIANG, H.; CHAUDHARY, V. Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems. In: HICSS '04: PROCEEDINGS OF THE PROCEEDINGS OF THE 37TH ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS'04) - TRACK 9, 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.90282.2.

JOERG, C. F.; HALBHERR, M.; ZHOU, Y. MIMD-Style Parallel Programming Based on Continuation-Passing Threads. In: MASSACHUSETTS INSTITUTE OF TECHNOLOGY, LABORATORY FOR COMPUTER SCIENCE, 1994. **Anais...** [S.l.: s.n.], 1994. p.pp.

KALE, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on c. In: IN PROCEEDINGS OF THE CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 1993. **Anais...** ACM Press, 1993. p.91–108.

KELLER, H.; PFERSCHY, U.; PISINGER, D. **Knapsack Problems**. [S.l.]: Springer, 2005. 546p.

LEISERSON, C. E.; KUSZMAUL, B. C. **Synchronized MIMD Computing**. [S.l.]: Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1994.

LIMA, J. V. F.; MAILLARD, N. Controle de Granularidade com Threads em Programas MPI Dinâmicos. In: WSCAD-SCC 2008, 2008. **Anais...** [S.l.: s.n.], 2008.

LIU, P.; AIELLO, W.; BHATT, S. An Atomic Model for Message-passing. In: SPAA '93: PROCEEDINGS OF THE FIFTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 1993, New York, NY, USA. **Anais...** ACM, 1993. p.154–163.

LYNCH, N. A. **Distributed Algorithms**. 1st.ed. [S.l.]: Morgan Kaufmann, 1997.

MOHR, E.; KRANZ, D. A.; R. H. HALSTEAD, J. Lazy Task creation: a technique for increasing the granularity of parallel programs. In: LFP '90: PROCEEDINGS

OF THE 1990 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, 1990, New York, NY, USA. **Anais...** ACM, 1990. p.185–197.

NARLIKAR, G. J.; BLELLOCH, G. E. Space-Efficient Scheduling of Nested Parallelism. **ACM Transactions on Programming Languages and Systems**, [S.l.], v.21, 1999.

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

NEVES, M. V.; RIGHI, R. R.; MAILLARD, N.; NAVAUX, P. O. A. Impacto da Migração de Máquinas Virtuais de Xen na Execução de Programas MPI. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO 2007 - WSCAD, 2007, 2007. **Proceedings...** [S.l.: s.n.], 2007.

NIEUWPOORT, R.; KIELMANN, T.; BAL, H. E. Satin: efficient parallel divide-and-conquer in java. In: EURO-PAR '00: PROCEEDINGS FROM THE 6TH INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, 2000, London, UK. **Anais...** Springer-Verlag, 2000. p.690–699.

NIEUWPOORT, R.; KIELMANN, T.; BAL, H. E. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In: PPOPP '01: PROCEEDINGS OF THE EIGHTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICES OF PARALLEL PROGRAMMING, 2001, New York, NY, USA. **Anais...** ACM, 2001. p.34–43.

OLIVEIRA, R.; CARíSSIMI, A.; TOSCANI, S. **Programação Concorrente**. [S.l.]: Editora Bookman, 2008. n.14.

PACHECO, P. S. **Paralell Programming With MPI**. 1.ed. [S.l.]: Morgan Kaufmann Publishers, 1997.

PARHAMI, B. Parallel Threshold Voting. **Comput. J.**, [S.l.], v.39, n.8, p.692–700, 1996.

PATTERSON, D. A.; HENNESSY, J. L. **Organização e Projeto de Computadores**: a interface hardware software. 3.ed. [S.l.]: Editora Campus, 2005.

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

PEZZI, G. P.; CERA, M.; MATHIAS, E.; MAILLARD, N. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. **Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on**, [S.l.], p.247–254, Oct. 2007.

R. H. HALSTEAD, J. Implementation of Multilisp: lisp on a multiprocessor. In: LFP '84: PROCEEDINGS OF THE 1984 ACM SYMPOSIUM ON LISP AND FUNCTIONAL PROGRAMMING, 1984, New York, NY, USA. **Anais...** ACM, 1984. p.9–17.

RANDALL, K. H. **Cilk**: efficient multithreaded computing. 1998. Tese (Doutorado em Ciência da Computação) — MIT.

SEILER, L.; CARMEAN, D.; SPRANGLE, E.; FORSYTH, T.; ABRASH, M.; DUBEY, P.; JUNKINS, S.; LAKE, A.; SUGERMAN, J.; CAVIN, R.; ESPASA, R.; GROCHOWSKI, E.; JUAN, T.; HANRAHAN, P. Larrabee: a many-core x86 architecture for visual computing. In: SIGGRAPH '08: ACM SIGGRAPH 2008 PAPERS, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.1–15.

TANENBAUM, A. S. **Modern Operating Systems**. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

TEL, G. **Introduction to Distributed Algorithms**. New York, NY, USA: Cambridge University Press, 2001.

Top 500 Team, T. **TOP500 Description**. 2009.

TRAORé, D.; ROCH, J.; MAILLARD, N.; GAUTIER, T.; BERNARD, J. Deque-Free Work-Optimal Parallel STL Algorithms. In: EURO-PAR, 2008. **Anais...** Springer, 2008. p.887–897. (Lecture Notes in Computer Science, v.5168).

WU, I. Efficient Parallel Divide-and-Conquer for a Class of Interconnection Topologies. In: ISA '91: PROCEEDINGS OF THE 2ND INTERNATIONAL SYMPOSIUM ON ALGORITHMS, 1991, London, UK. **Anais...** Springer-Verlag, 1991. p.229–240.

WU, I. chen; KUNG, H. T. Communication Complexity for Parallel Divide-and-Conquer. In: IN PROCEEDINGS OF THE 32ND ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 1991. **Anais...** [S.l.: s.n.], 1991. p.151–162.

YEUNG, J. H. C.; TSANG, C. C.; TSOI, K. H.; KWAN, B. S. H.; CHEUNG, C. C. C.; CHAN, A. P. C.; LEONG, P. H. W. Map-reduce as a Programming Model for Custom Computing Machines. In: FCCM '08: PROCEEDINGS OF THE 2008 16TH INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 2008, Washington, DC, USA. **Anais...** IEEE Computer Society, 2008. p.149–159.