Melhorando o Desempenho de Algoritmos do Tipo Branch & Bound em MPI via Escalonador com Roubo Aleatório de Tarefas *

Stéfano D. K. Mór, Nicolas Maillard
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, RS, Brasil
{sdkmor, nicolas}@inf.ufrgs.br

Resumo

Nossa principal contribuição é a integração de um modelo de escalonamento distribuído por roubo de tarefas para computação em MPI capaz de otimizar o desempenho de programas do tipo Branch & Bound. Esse escalonador é introduzido em tempo de compilação e é independente da distribuição MPI usada. Resultados experimentais mostram que se pode obter um ganho de até 80% no desempenho, mantendo o speedup *próximo ao linear e sem a perda do consumo linear de memória. Esses ganhos se confirmam mesmo em um ambiente de processadores homogêneos, que tendem a produzir um menor desbalanceamento da carga de trabalho.*

1. Introdução

Desde 1996 *Message Passing Interface* (MPI) é o padrão *de facto* para comunicação via troca de mensagens em máquinas *Multiple Instruction, Multiple Data* (MIMD) [8]. MPI não oferece mecanismos para balanceamento automático da carga computacional; o controle eficiente da ociosidade 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 dos algoritmos do tipo *Branch and Bound* (B&B).

Algoritmos B&B são um subgrupo dos algoritmos de Divisão e Conquista e consistem de duas fases: a divisão de uma tarefa em subtarefas (*branch*) e a eliminação de subtarefas que não possam retornar uma resposta ótima (*bound*) [6]. São de especial interesse à área de algoritmos para Programação de Alto Desempenho (PAD) pois a granularidade de suas tarefas (*i.e.*, a quantidade de trabalho sequencial feita por cada recurso) não é conhecida em tempo de compilação; processadores cujas tarefas tenham um número elevado de *bounds* tendem a ficar ociosos enquanto outros processadores ficam sobrecarregados, mesmo em uma máquina MIMD de recursos homogêneos, onde o desbalanceamento da carga de trabalho devido às diferenças de velocidades 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 *bounds* por tarefa é, por si, um problema $\mathcal{NP}$-Completo.

Este artigo propõe o uso da técnica de Roubo Aleatório de Tarefas (RAT) para realizar um balanceamento de carga eficiente em algoritmos paralelos do tipo B&B em um ambiente MPI através da implementação de um escalonador distribuído. Esse escalonador é baseado na discussão proposta em [1], sobre escalonadores para ambientes multithread e no trabalho desenvolvido em [18] sobre a linguagem para programação multithread CILK. Nossa proposta é desenvolver um escalonador nos mesmos moldes, mas integrado ao modelo MPI. Essa adaptação não é trivial, dadas as diferenças nas limitações entre um cenário de memória compartilhada (*threads*) e memória distribuída (processos em uma máquina MIMD).

A base para a introdução do nosso algoritmo de escalonamento é um modelo de processamento MPI para B&B já existente chamado *Árvore Paralela de Pesquisa*. Utilizando-o, esperamos acomodar as limitações de escopo de variáveis que agregados de computadores [16] (*clusters*) apresentam em relação às máquinas multithread.

2. Árvore Paralela de Pesquisa

Em geral, um problema B&B é resolvido, em paralelo, através do emprego de uma Árvore Paralela de Pesquisa (APP), apresentada –na versão MPI–, em [17]. APP é um modelo para a paralelização de algoritmos com progresso em árvore n-ária projetado 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.

*Agradecimentos ao CNPq pelo financiamento que tornou este trabalho possível.

No modelo APP todos os processadores têm uma lista de dados independentes a serem processados (tarefas), potencialmente fora de ordem. Para uma máquina MIMD multicomputador com P processadores o algoritmo consiste em:

1. Um processador p_i ($1 \leq i \leq P$) recebe o nó raiz da árvore (tarefa) e a expande até um número δ de nós (subtarefas) 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 subtarefas, fazendo busca em profundidade. p_i reserva algumas tarefas para si e, após distribuir as demais subtarefas 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, o principal pergunta que resta é:

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

A resposta para essa pergunta impacta diretamente na quantidade de comunicação realizada pela computação; 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 da execução), apresentam um desbalanceamento de carga que prejudica o tempo de execução.

A definição de que tarefas executam em quais recursos em um dado momento é chamada de *escalonamento de execução*, denotado por $\mathcal{A}$ [4]. Para uma dada APP com N nós, altura h e grau δ ($\delta \geq 2$), seguindo parcialmente a notação estabelecida em [22], o *custo de computação paralela* $T_P[\mathcal{A}](H)$ para um escalonamento de execução $\mathcal{A}$ em uma árvore de Divisão e Conquista (D&C) H com P processadores é o número máximo de nós que cada processador $p \in \{1, ..., P\}$ pode expandir. Como existem N nós e P processadores, o limite inferior de $T_P[\mathcal{A}](H)$ é $T_P[min](H) = \lceil N/P \rceil$. Nesse sentido, para uma APP H, se o critério de escolha de p_j (*i.e.*, $\mathcal{A}$) for bom, então $T_P[\mathcal{A}](H)$ se aproxima de $T_P[min]$.

O *custo de comunicação* $M_P[\mathcal{A}](H)$ é o número de nós-cruzados que o algoritmo $\mathcal{A}$ gera ao executar sobre uma APP H usando P processadores. Um *nó cruzado* é um nó que é gerado em um processador p_a, mas expandido em um processador p_b, tal que $a \neq b$.

Uma bom algoritmo de escalonamento é capaz de balancear a carga do sistema e aproximar $T_P[\mathcal{A}](H)$ de $T_P[min](H)$. Para atingir esse objetivo no cenário proposto emprega-se um algoritmo de roubo aleatório de tarefas, como descrito na próxima seção.

3. Roubo Aleatório de Tarefas

O escalonamento baseado em roubo de tarefas é um algoritmo distribuído clássico especificado em alto nível de abstração para o escalonamento de tarefas no modelo *work-stealing*, 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 [5, 9]. Outros trabalhos evidenciaram se tratar de uma abordagem mais eficiente que *work-pushing*, *e.g.*, [1, 7, 10, 12, 15].

A escolha que o processador p_j faz de qual outro processador roubar o trabalho (o escalonamento $\mathcal{A}$) é essencial na construção do algoritmo. O objetivo é normalizar algum $\mathcal{A}$ em razão de $T_P[\mathcal{A}](H)$ e $M_P[\mathcal{A}](H)$. Sabe-se, no entanto, que o trabalho realizado por um processador p e a profundidade δ de H são critérios frequentemente antagônicos [20]; quanto menor a granularidade de uma tarefa, maior tende a ser $M_P[\mathcal{A}](H)$, o que aumenta a latência da aplicação e distancia $T_P[\mathcal{A}](H)$ de $T_P[min](H)$.

Em [22] 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*. No algoritmo proposto, $T_P[\mathcal{A}](H) = \lceil N/P \rceil$ e $M_P[\mathcal{A}](H) = P\delta h$, que são limites inferiores para qualquer APP [22]. Em [21] 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*.

O resultado mais importante para esta discussão é apresentado em [1] e revisado em [2]. Nesses trabalhos mostra-se um escalonamento de *threads* por roubo de tarefas para árvores D&C com escolha aleatória da "vítima", que nomeamos *Roubo Aleatório de Tarefas* (RAT). É apresentada prova formal de que a escolha aleatória do alvo atinge, com alta probabilidade, os limites teóricos mostrados anteriormente [22, 21], com a vantagem da simplicidade de

programação.

Seja $T_1(H)$ o tempo de execução do algoritmo com 1 (um) processador e $T_\infty(H)$ o tempo de execução do mesmo algoritmo com tantos processadores quanto necessário. Obtém-se $T_P[RAT](H) = \mathcal{O}(T_1(H)/P + T_\infty(H))$, $M_P[RAT](H) = \mathcal{O}(P\delta T_\infty(H))$, que verificam os resultados anteriores, haja visto que $T_\infty(H) = h$ [3]. Adicionalmente, o espaço em memória S utilizado pela computação com P processadores satisfaz $S \leq S_1 P$, onde S_1 é a quantidade de memória ocupada pela execução sequencial do algoritmo.

4. Proposta

Nossa proposta se consolida em implementar um escalonador MPI que empregue uma APP onde o critério de escolha do processador p_j para escolher o processador-alvo p é aleatório. Nossa implementação é uma versão em memória distribuída do que é apresentado em [2], diferenciando-se pela ordem em que as tarefas são empilhadas; a sincronização e salvamento de contexto em um cenário de memória distribuída é mais onerosa do que no de memória compartilhada. Dessa maneira, o algoritmo muda a ordem de empilhamento de modo a eliminar a necessidade de salvamento de contexto e diminuir a comunicação. Nosso algoritmo RAT modificado recebe o nome de *Roubo Aleatório de Tarefas para Memória Distribuída* (RATMD).

RATMD é como segue. No lugar de uma lista, cada processador possui uma *double ended queue* (*deque*) de tarefas aptas, *i.e.*, uma estrutura de dados do tipo *container*, onde tarefas aptas a executar podem ser inseridas na base e retiradas da base ou do topo. Cada processador também possui uma fila de tarefas desbloqueadas (*i.e.*, tarefas que em determinado momento dispararam subtarefas, bloquearam esperando o resultado da computação e agora estão aptas a executar novamente).

A execução começa com todos os processadores sem executar e com as *deque* vazias, a exceção de uma, que contém apenas a tarefa raiz da APP. A execução é como descrita anteriormente, adicionando o tratamento a quatro possíveis eventos:

Criação de nova tarefa. Se a tarefa Φ_a gera uma nova tarefa Φ_b, então Φ_b é colocada no topo da *deque* e o processador continua o trabalho em Φ_a.

Bloqueio/Término. Se a tarefa Φ_a bloqueia ou termina, o processador verifica a fila de tarefas desbloqueadas e executa a primeira tarefa apta. Caso não existam tarefas aptas na fila, busca-se na *deque*. Nesse caso, duas situações podem ocorrer:

- **A deque possui tarefas.** Nesse caso, o processador tira da base a tarefa apta e começa a executá-la.
- **A deque não possui tarefas.** Nesse caso, seleciona outro processador de maneira aleatória e rouba a tarefa do topo da outra *deque*, executando-a diretamente. Se o processador selecionado também não contiver tarefas, repete-se o passo, até que se receba um sinal de término do algoritmo.

Desbloqueio. Caso Φ_a esteja executando e Φ_b desbloqueie, Φ_b é colocada no final da fila de tarefas desbloqueadas após a execução de Φ_a e do enfileiramento de todas as tarefas que desbloquearam antes de Φ_b durante a execução de Φ_a.

A execução termina quando o algoritmo B&B realiza uma detecção distribuída de término e sinaliza aos processadores. O modelo de comunicação é o apresentado em [14]; 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.

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*. Essa troca de mensagens é feita de maneira não bloqueante; enquanto espera pela resposta um processador pode atender outras requisições. O Corolário 5.2, apresentado na Seção 5 mostra que esse modelo é suficiente para que o escalonamento não resulte em *deadlock*.

5. Propriedades

Essa seção apresenta três propriedades importantes do algoritmo RATMD, no cenário de memória distribuída: ausência de *deadlock*, ausência de postergação indefinida e bom desempenho. Além disso, reforçamos o fato de o algoritmo ser não-determinístico, o que auxilia na depuração da implementação.

5.1. Ausência de *Deadlock*

Começamos enunciando o Lema 5.1 [19] para, em seguida, mostrar que decorre trivialmente dessa definição que o sistema não entra em *deadlock* (Corolário 5.2):

Lema 5.1. *Em um sistema distribuído, são necessárias quatro condições para que exista a possibilidade de que este entre em deadlock:*

1. *exclusão mútua bloqueante;*
2. *uma vez alocado, um recurso em exclusão mútua não pode ser preemptado por outro processo.*
3. *um processo que tem um recurso alocado pode requerer outro; e*
4. *poder haver dependências cíclicas entre os processos.*

Corolário 5.2. RATMD *não entra em* deadlock.

Demonstração. Como um nó faz uma requisição de conexão de maneira não-bloqueante, então (1) do Lema 5.1 não é verdadeiro. Adicionalmente, como um nó atende requisições enquanto faz alguma requisição, então (4) também não é verdade. □

5.2. Ausência 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. Apresentamos, então, o Corolário 5.3, que mostra que o tempo de execução de qualquer tarefa apta em uma dada *deque* é proporcional à carga da *deque* e executa em tempo finito.

Seja uma tarefa Φ presente na *deque* do processador p. Suponha que uma mensagem de solicitação de roubo, seu atendimento e envio da tarefa roubada demoram no máximo α instantes de tempo atômico. Suponha também que o tempo máximo de execução de cada tarefa seja β instantes de tempo atômicos e que em um instante de tempo t_i, existem σ_{t_i} tarefas na *deque* que estão à frente de Φ (possivelmente $\sigma_{t_i} = 0$).

Corolário 5.3. *Se Φ está na deque de p em t_i, então Φ é executada no instante t, onde*

$$t < t_i + \alpha + \beta\sigma_{t_i} \tag{1}$$

Demonstração. A observação fundamental é que uma tarefa roubada é sempre imediatamente executada (nunca entra na *deque*). Existem duas possibilidades para que uma tarefa originada na *deque* de p tenha sido executada:

1. ser executada por p: nesse caso, executará em tempo $t_1 = t_i + \beta\sigma_{t_i}$
2. ser executada por outro processador: nesse caso, ela foi roubada em algum instante exatamente α unidades de tempo antes de ser executada, sendo executada em $t_2 = t_i + \alpha + \beta(\sigma_{t_i} - \sigma_t)$.

Como $max(t_1, t_2) < t_i + \alpha + \beta\sigma_{t_i}$, então a tarefa é sempre executada em $t < t_i + \alpha + \beta\sigma_{t_i}$. □

5.3. Desempenho

Dizemos que RATMD está em *estado eficiente* em um instante de tempo t quando todos os P processadores estão executando alguma tarefa. Enquanto $\mathcal{A}$ permanece em um estado eficiente, $M_P[\mathcal{A}](H) = 0$. Dizemos que a computação está em um *estado de sincronização* quando contrário.

Enquanto a computação está em estado eficiente, $T_P[\mathcal{A}](H)$ é o mais próximo possível de $T_P[min](H) = \lceil N/P \rceil$. Como a execução sequencial do algoritmo processa exatamente N nós, então $N = T_1(H)$ e, portanto, $T_P[min](H) = T_1(H)/P$. Logo, quanto mais tempo nosso algoritmo ficar em estado eficiente, tanto melhor. Vamos chamar de $T_P^{syn}[A](H)$ o tempo que o algoritmo permanece no estado de sincronização.

Dessa maneira, temos o seguinte lema:

Lema 5.4. *Nosso algoritmo sempre executa em tempo*

$$T_P[\text{RATMD}](H) = \frac{T_1(H)}{P} + T_P^{syn}[\text{RATMD}](H) \tag{2}$$

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. Por conseguinte, 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 $\log_2(P)$, pois a cada rodada no máximo um processador p_a rouba uma tarefa de algum processador p_b (conforme descrito na Seção 3), a expande e preenche sua *deque*. No pior caso, o processador inicial p_i nunca é roubado e o tempo de execução é $T_1(H)$.

Surge, então, o problema de descobrir o valor de $T_P^{syn}[\text{RATMD}](H)$. Em [2] 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. 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,

Lema 5.5. *A expectativa de tempo em que o sistema passa em estado de sincronização é*

$$T_P^{syn}[RAT](H) \in \mathcal{O}\left(T_\infty(H)\right) \tag{3}$$

O Lema 5.5 é importante, pois o valor de $T_\infty(H)$ é logarítmico, o que demonstra que nosso algoritmo executa, com alta probabilidade, com uma eficiência muito mais próxima ao melhor caso que ao pior caso. Com isso, temos o Corolário 5.6:

Corolário 5.6. *A expectativa do tempo de execução do* RATMD *é*

$$T_P[\text{RATMD}](H) \in \mathcal{O}\left(\frac{T_1(H)}{P} + T_\infty(H)\right) \tag{4}$$

Demonstração. Decorre dos Lemas 5.4 e 5.5. □

O Corolário 5.6 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. Não será apresentada demonstração formal, mas observamos que é altamente improvável (probabilidade inversamente proporcional a P) que o processador que contenha as subtarefas iniciais não seja roubado por pelo menos um dos outros processadores. Progressivamente, é cada vez menos provável que uma requisição de roubo seja infrutífera, visto que o número de processadores com tarefas tende a dobrar a cada rodada e, portanto, o tempo de espera para se atingir um

estado eficiente, a partir do estado inicial, é logarítmico ($T_\infty(H)$ é logarítmico).

Além do ganho de eficiência proposto, esse escalonamento mantém o consumo de memória proporcional somente ao tamanho da entrada n, conforme o Corolário 5.7. Considerando S_1 como o espaço ocupado pela execução sequencial do algoritmo:

Corolário 5.7. *A quantidade de memória consumida pelo algoritmo apresentado é $S_P \in \mathcal{O}(S_1 P)$.*

Demonstração. Seja $\Delta = S_P - S_1$ ($S_P \geq S_1$). Se cada processador executar o algoritmo para toda a entrada (processando, inclusive, as subtarefas 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 B&B), tem-se que $S_P < S_1 P \Delta$ e, portanto, $S_P \in \mathcal{O}(S_1 P)$. □

O Corolário seguinte, embora não verse sobre a eficiência do algoritmo, mostra que sua execução é não-determinística. É útil para a verificação da correção da implementação.

Corolário 5.8. *O algoritmo apresentado é não-determinístico.*

Demonstração. Como o processador escolhe seu alvo aleatoriamente, a cada execução a quantidade de mensagens trocadas varia com estas escolhas. □

6. Resultados Experimentais

Nossos experimentos foram realizados utilizando-se MPI-1.2 e uma versão paralela do Problema da Mochila, detalhado (em versão sequencial) no Algoritmo 6.1. As primitivas de escalonamento foram inseridas em tempo de compilação.

6.1. Problema da Mochila

Nossa aplicação é uma solução B&B para o Problema da Mochila Ilimitado (MBB), descrito na Definição 6.1 [11] e sua versão sequencial iterativa mostrada no Algoritmo 6.1.

Definição 6.1 (Problema da Mochila Ilimitado)**.** 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

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

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

Algoritmo 6.1 MBB(V, W, C)

Requer: V e W ordenados por valor de densidade dos elementos ($\forall i \in \{1, ..., n-1\} \mid v_i/w_i \geq v_{i+1}/w_{i+1}$).

```
1:  x_1 ← ⌊C/w_1⌋
2:  para j ← 2, n faça
3:      x_j ← ⌊(C − Σ_{i=1}^{j}(w_i × x_i))/w_j⌋
4:  X* ← X
5:  v* ← Σ_{i=1}^{n}(v_i × x_i)
6:  enquanto VERDADEIRO faça
7:      k ← n
8:      enquanto (k ≥ 1) ∧ (x_k = 0) faça
9:          k ← k − 1
10:     se k = n então
11:         x_k ← 0
12:         continue
13:     se (k ≥ 1) então
14:         x_k ← x_k − 1
15:         v' ← Σ_{i=1}^{k}(v_i × x_i)
16:         r' ← (Σ_{i=1}^{n}(w_i × x_i)) − (Σ_{i=1}^{k}(w_i × x_i))
17:         se v' + ((v_{k+1}/w_{k+1}) × r') ≤ v* então
18:             continue
19:         senão
20:             para j ← k + 1 até n faça
21:                 x_j ← ⌊(C − Σ_{i=1}^{j}(w_i × x_i))/p_j⌋
22:             v'' ← Σ_{i=1}^{n}(v_i × x_i)
23:             se v'' > v* então
24:                 X* ← X
25:                 v* ← v''
26:             k ← n
27:     senão
28:         quebra
29: X ← X*
```

O Algoritmo 6.1, MBB, após seu término, tem X preenchido com a distribuição ótima, de acordo com a Definição 6.1. A complexidade de pior caso é

$$C_p[\text{MBB}](n) \in \mathcal{O}\left(\left(\frac{C}{min\,(W)} \right)^n \right) \tag{5}$$

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.

6.2. Implementação do Escalonador

Nosso escalonador oferece uma interface de *callbacks* para que o programador defina sua tarefa e como particioná-la, além de suporte a *benchmarks* (para medir os aspectos avaliados nos gráficos desse artigo; *e.g.*, tempo de execução, consumo de memória, quantidade de comunicação) através do (e integrado ao) mesmo recurso. Além disso, há uma

implementação de *broadcast* não bloqueante que precisamos construir devido à falta de suporte na norma MPI. Como exemplo, na Figura 1, segue a estrutura de dados a ser reescrita pelo programador e um preâmbulo de uma *callback* que usa essa estrutura.

```
/**
 * 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 1. Exemplo de código a ser reescrito pelo programador.

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 B&B. Nossa API está fora da especificação MPI, mas segue sua convenção de nomenclatura, facilitando o aprendizado do programador. A programação é feita através da implementação, pelo usuário, de protótipos de funções previamente definidos e o preenchimento de estruturas de dados que representam tarefas.

6.3. Medições

Esta seção mostra resultados práticos baseado em nossas observações anteriores.

Nossos 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 1266MHz, 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 não convergiu.
- 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 2(a) mostra indícios da correção do Corolário 5.7. 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.

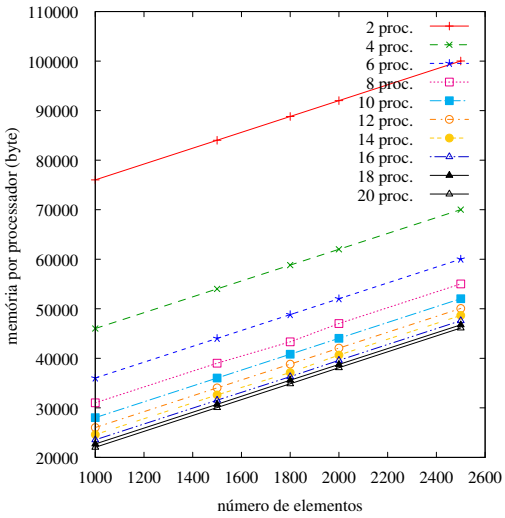
O Corolário 5.8 implica que o número de mensagens trocadas por um processador específico não possui dependência funcional com n ou P, pois a execução do algoritmo é não-determinística. De fato, a Figura 2(b) mostra o caso específico do processador #2, cuja comunicação é bastante caótica. Resultados em 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. Ainda assim, usamos a Figura 2(b) para ilustrar a caoticidade em relação à troca de mensagens.

A Figura 3(a) mostra o ganho de desempenho referenciado na Seção 5. Nossa 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. Além disso, a Figura 3(b) mostra que o consumo de memória do nosso algoritmo é praticamente idêntico ao consumo de memória da execução do MBB sem nosso escalonador. Para ambas as medições foram considerados 2500 tipos de itens. Por fim, a Figura 4 mostra que o *speedup* obtido é próximo ao linear.

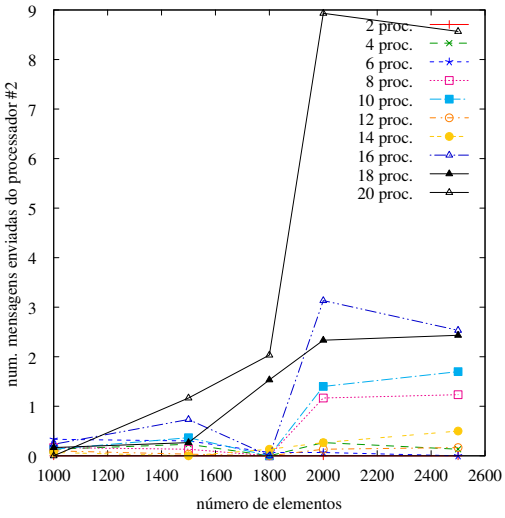
7. Conclusões & Trabalhos Futuros

O algoritmo para escalonamento MPI de tarefas B&B mostrou-se eficiente, alcançando os limites teóricos calculados. Em nossas medições pode-se obter um ganho de até 80% no desempenho, mantendo o *speedup* próximo ao linear e sem a perda do consumo linear de memória, mesmo em um ambiente de processadores homogêneos, que tendem a provocar um desbalanceamento de carga de trabalho inferior ao de processadores heterogêneos. Um código B&B MPI qualquer pode ser escalonado pelo nosso algoritmo com a inserção de código em tempo de compilação, sendo independente da distribuição 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 subtarefas.

Nosso próximo passo é incluir o escalonador dentro de alguma implementação MPI, dando total transparência

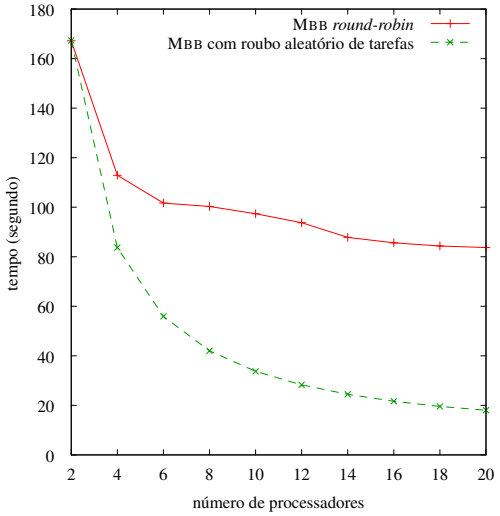


(a) Crescimento do consumo de memória linear em n. Evidência da correção do Corolário 5.7.

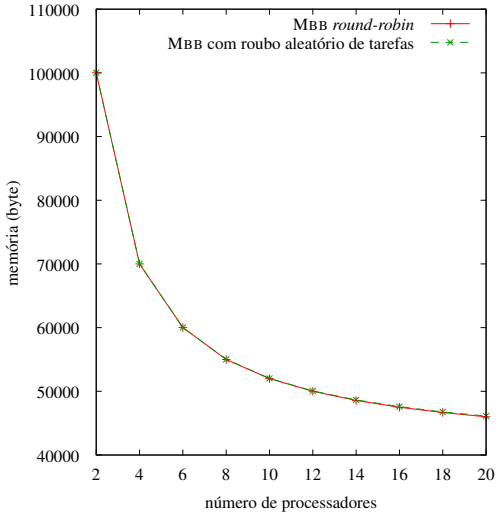


(b) Troca de mensagens independente de n e P. Evidência da correção do Corolário 5.8.

Figura 2. Evidência do crescimento de memória linear e padrão de comunicação aleatório em n.



(a) Execução do MBB com roubo de tarefas (-×-) contra um MBB *round-robin* (-┼-), tomados 2500 elementos. Evidência da correção do Corolário 5.6.



(b) Alocação de memória para um processo MBB (-×-) contra um MBB *round-robin* (-┼-), tomados 2500 elementos. Evidência da correção do Corolário 5.7.

Figura 3. MBB com roubo de tarefas *vs.* MBB *round-robin*: tempo de execução e alocação de memória para 2500 elementos.

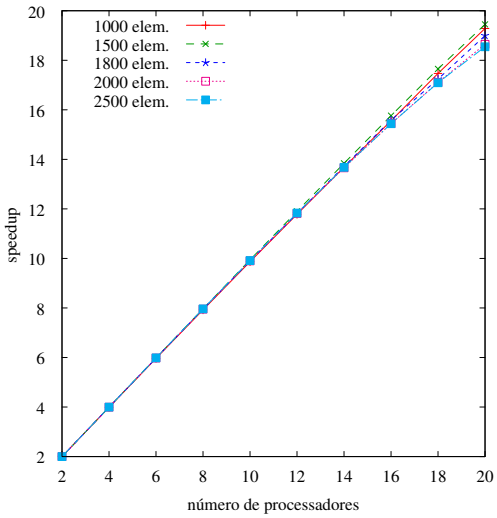


Figura 4. Speedup próximo ao linear atingido pelo MBB com roubo de tarefas.

ao programador. MPI-2, em especial, provê a primitiva `MPI_Comm_spawn`, destinada a criar novas tarefas, mas seus algoritmos de escalonamento baseiam-se em implementações de *round-robin* (*work-pushing*) que, conforme apresentado na Seção 6.3 são ineficientes para os problemas tratados. Embora dependente de uma distribuição específica, essa solução é genérica e portável. Até o final de 2009 esperamos disponibilizar uma versão modificada da distribuição MPICH2 (que possui um escalonador modular, topologia lógica em anel e suporte ao emprego de *threads*) com escalonamento B&B eficiente.

O passo final será integrar essa solução MPI com escalonador *work-stealing* distribuído em algum sistema de controle de granularidade de tarefas (*e.g.*, [13]), obtendo-se um sistema capaz de escalonar uma tarefa do tamanho e forma (*thread*, processo) mais eficiente em um *cluster*, independente da arquitetura de processadores paralelos empregada (multithread, multicore, manycore).

Referências

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. MIT Laboratory for Computer Science, 1994.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of ACM*, 46(5):720–748, September 1999.

[3] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[4] P. Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[5] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[7] R. Feldmann, P. Mysliwietz, and B. Monien. A fully distributed chess program. *Advances in Computer Chess VI*, pages 1–27, 1991.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with Message-Passing Interface*. Scientific and Engineering Computation Series. The MIT Press, Massachusetts Institue of Technology - Cambridge, Massachusetts 02142, 2nd edition, 1999.

[9] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17, New York, NY, USA, 1984. ACM.

[10] C. F. Joerg, M. Halbherr, and Y. Zhou. MIMD-style parallel programming based on continuation-passing threads. In *Massachusetts Institute of Technology, Laboratory for Computer Science*, page pp., 1994.

[11] H. Keller, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2005.

[12] C. E. Leiserson and B. C. Kuszmaul. Synchronized MIMD computing. Technical report, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1994.

[13] J. V. F. Lima and N. Maillard. Controle de granularidade com threads em programas mpi dinâmicos. In SBC, editor, *IX Simpósio em Sistemas Computacionais - WSCAD-SSC 2008*, pages 117–124, Campo Grande, Brasil, Nov 2008.

[14] P. Liu, W. Aiello, and S. Bhatt. An atomic model for message-passing. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 154–163, New York, NY, USA, 1993. ACM.

[15] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. 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*, pages 185–197, New York, NY, USA, 1990. ACM.

[16] P. O. A. Navaux and C. A. F. D. Rose. *Arquiteturas Paralelas*. Number 15 in Série Livros Didáticos. Editora Sagra Luzzato, 1st edition, 2003.

[17] P. S. Pacheco. *Paralell Programming With MPI*. Morgan Kaufmann Publishers, 1st edition, 1997.

[18] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT, Jun 1998.

[19] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[20] D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel STL algorithms. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 887–897, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] I.-C. Wu. Efficient parallel divide-and-conquer for a class of interconnection topologies. In *ISA '91: Proceedings of the 2nd International Symposium on Algorithms*, pages 229–240, London, UK, 1991. Springer-Verlag.

[22] I.-C. Wu and H. T. Kung. Communication complexity for parallel divide-and-conquer. In *In Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, 1991.