Algoritmos probabilísticos de *list ranking* para máquinas paralelas com memória distribuída

Fabiana Soares Santana

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇAO DO GRAU DE MESTRE

MATEMATICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Siang Wun Song**

Durante a elaboração deste trabalho a autora recebeu apoio financeiro da FAPESP - Proc. No. 96/1251-0

- São Paulo, dezembro de 1997 -

Algoritmos probabilísticos
de *list ranking*
para máquinas paralelas
com memória distribuída

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Fabiana Soares Santana
e aprovada pela comissão julgadora.

São Paulo, 18 de janeiro de 1998.

Banca examinadora:

- Prof. Dr. Siang Wun Song (orientador) - IME-USP
- Prof. Dr. José Augusto R. Soares - IME-USP
- Prof. Dr. Edson Norberto Cáceres - DCT-UFMS

Para μ, π e υ.

Agradecimentos

Aos meus pais, por terem cumprido seu papel de forma tão brilhante, fazendo-me honrá-los sobre todas as coisas e ter orgulho do meu nome.

Aos meus irmãos, por terem me mostrado que era possível.

Ao meu irmão Gero, *in memoriam*, pela coragem e talento.

Ao Prof. Dr. Siang Wun Song, pela orientação e apoio durante todo o curso.

Ao Prof. Dr. Yoshiharu Kohayakawa, pela amizade e por sua colaboração na realização deste trabalho.

Ao Flávio Keidi Miyazawa e ao Orlando Lee, pelo risco e por permanecer ao meu lado quando as coisas ficaram difíceis.

Aos amigos e familiares Bina, Léo, Renato, Victor, Bininha, Rose, Guga, Cassiana, Johnny, Carol, Gabi, Clara, Otávio, Pão, Sylvio (*in memoriam*), Rô, Rafa, Renata, Rico, July, Maurinho, Carlos, Fabíola, Alice, Cruz, Júlio, Susana, Fernando, Lu, Mané, Sandro, Sayuri, Sérgio, e Zeza, por serem o que são.

Ao pessoal do IME, pelos bons papos e pelo apoio técnico.

À FAPESP, pelo apoio financeiro.

À Capes, pelo apoio financeiro.

À Idéias e Ideais Consultoria S/C Ltda., pelo apoio financeiro.

Ao Dr. Luiz Fernando Carrijo da Cunha, pelo seu excelente trabalho e por me transformar na mulher que sou hoje.

Índice

Introdução

O problema principal em computação paralela é o bem conhecido *software bottleneck*, ou *gargalo de software*. Atualmente, aplicações comerciais em máquinas paralelas estão restritas a problemas trivialmente paralelizáveis, onde os requisitos de comunicação são baixos. Por outro lado, existe uma grande quantidade de literatura sobre algoritmos paralelos para problemas não-triviais. Entretanto, os resultados deixam a desejar quando estes algoritmos são executados em máquinas reais, devido a inexistência de um modelo de computação paralela adequado e suficientemente próximo das máquinas existentes para permitir uma predicção razoável do desempenho.

Essa discrepância é óbvia para algoritmos no modelo *PRAM* (*Parallel Random Access Machine*, apresentado no *apêndice A*), mas mesmo em outros modelos o problema não está satisfatoriamente resolvido.

Portanto, é imperativo projetar modelos e algoritmos onde as análises teóricas de complexidade estão próximas aos tempos observados na execução real.

No *capítulo 2*, fazemos uma breve discussão sobre essa questão e apresentamos os modelos *BSP* (*Bulk-Synchronous Parallel*) e *CGM*(*Coarse-Grained Multicomputer*). Estes modelos, relativamente recentes, propõem soluções para máquinas com memória distribuída. (A validade deles ainda está sendo verificada e nossos resultados podem contribuir para isso, mas o nosso objetivo é muito mais restrito.)

No mesmo capítulo, enunciamos o problema estudado neste trabalho: realizar o *list ranking* para uma lista ligada no modelo *CGM*, problema inicialmente estudado por F. Dehne e S. W. Song em [12].

Uma máquina paralela baseada no modelo *CGM* consiste de um conjunto de p processadores, cada um com memória local muito maior que O(1), interligados através de uma rede de comunicação arbitrária. Em geral, se n é o tamanho da entrada, $p << n$. Neste modelo, o desempenho de um algoritmo é dado principalmente pelo número de *rodadas de comunicação.*

No *capítulo 3*, apresentamos dois algoritmos probabilísticos para o problema. O primeiro resolve o *list ranking* em $O(n/p)$ operações de computação local por rodada e $O(\log p + \log\log n)$ rodadas de comunicação, com alta probabilidade. O segundo algoritmo resolve o LR em um número esperado de $O(n/p)$ operações de computação local por rodada e $O(\log p)$ rodadas de comunicação[1].

O primeiro algoritmo é apresentado em [12]. No mesmo artigo, os autores descrevem outro algoritmo probabilístico que requer $O(k \log p)$ rodadas, com alta probabilidade, onde $k < \ln^* n$. Note que, apesar de k ser extremamente pequeno para instâncias reais, a complexidade deste algoritmo ainda depende do tamanho da lista.

O segundo algoritmo do *capítulo 3* é original e consiste na principal contribuição deste trabalho. Ao contrário dos anteriores, resolve o *list ranking* em um número que independe do tamanho da entrada. Por este motivo, optamos por omitir o segundo algoritmo de F. Dehne e S. W. Song. O primeiro foi incluído por considerarmos o nosso como uma evolução deste.

Os algoritmos aqui descritos foram implementados utilizando a máquina paralela *Parsytec PowerXplorer 16/32 Parallel Computing System.* A máquina em questão apresenta 16 nós sob uma topologia 2-D (*grade*). Cada nó consiste de um processador de aplicação *PowerPC601* (80 *MHz*) e um processador de comunicação *T805*, com memória local de 32 *MBytes.* A taxa de transferência de dados entre os nós é da ordem de milisegundos. Os resultados encontram-se no final do *capítulo 3.* Uma breve descrição da implementação do primeiro algoritmo pode ser encontrada no *apêndice B.*

Finalmente, apresentamos no *capítulo 4* as nossas conclusões sobre algoritmos probabilísticos para *list ranking* no modelo *CGM*, incluindo uma breve análise dos resultados experimentais obtidos com as nossas implementações. Acrescentamos a esse capítulo uma pequena lista com algumas questões levantadas no decorrer do trabalho.

[1] A notação O é a usual [20].

Capítulo 2

List ranking no modelo *CGM*

2.1 Modelos de computação

Em [31], um sistema de computação moderno é definido como uma coleção de um ou mais processadores, algumas memórias, relógios, terminais, discos e interfaces de rede, além de outros dispositivos de entrada e saída de dados. Controlar todos estes componentes de forma adequada na resolução de um dado problema a fim de obter o melhor desempenho possível do sistema é uma tarefa extremamente difícil e nem sempre está diretamente relacionada ao problema. Os sistemas paralelos têm esta complexidade muitas vezes multiplicada pelo aumento do número de variáveis a considerar.

No entanto, da mesma forma que existem os sistemas operacionais, os compiladores e os editores para simplificar o uso prático dos sistemas de computação, existem abstrações teóricas para caracterizar tais sistemas, que são os chamados *modelos de computação*. Um modelo de computação deve descrever as operações primitivas que podem ser executadas em um sistema, associando-as aos respectivos custos [22, 29, 27].

Um bom modelo deve procurar fornecer hipóteses para a resolução dos problemas e adequar-se ao projeto e análise de algoritmos. Deve permitir ao usuário avaliar e comparar soluções alternativas para escolher a mais adequada às suas necessidades. Deve também captar todas as capacidades e restrições do sistema. É extremamente importante que as hipóteses fornecidas permitam fazer previsões acuradas sobre o desempenho real dos algoritmos para ele projetados [22, 29].

No entanto, o modelo não deve se perder em detalhes. Ao contrário, precisa ser tão simples quanto possível para que os algoritmos sejam facilmente implementáveis [18, 22, 29]. Em [29] recomenda-se *transparência de desempenho*, o que significa que o modelo deve permanecer exposto para a programação, permitindo que sejam acrescentadas às suas primitivas novas operações e estruturas de dados para as quais as análises dos algoritmos continuem válidas. Também é

recomendável que o modelo seja tão independente da máquina quanto possível para permitir o intercâmbio de soluções.

Em modelos de computação paralela devem ser acrescentados os fatores críticos intrínsecos ao paralelismo para permitir resultados satisfatórios [29, 32, 34, 35, 36]. Tais fatores incluem a concorrência computacional, a alocação de dados e de processadores, a comunicação e o sincronismo.

O modelo *RAM* [7] fornece toda a abstração necessária para a computação seqüencial, por sua proximidade com as *máquinas de von Neumman* [22, 29]. Ele não fornece previsões extremamente precisas para a execução prática dos programas, mas é bastante adequado para o projeto de algoritmos[1]. Permite comparar entre as soluções antes de sua execução e, até mesmo, decidir quais problemas são tratáveis e quais têm um custo computacional proibitivo.[2] O desenvolvimento prático da computação paralela, de certa forma, está amarrado à definição de um modelo semelhante ao *RAM* [29].

Para a avaliação dos modelos, destaca-se o quesito de *previsões acuradas* para a execução de programas reais e a necessidade de verificação experimental, opinião compartilhada em diversos trabalhos [9, 10, 12, 29, 32, 34, 35, 36]. Em [36], acrescenta-se ao modelo a observância das restrições físicas e tecnológicas. Além dos citados, muitos outros trabalhos têm proposto soluções práticas eficientes e modelos para a computação paralela. No entanto, ainda não se sabe se este modelo existe para resolver problemas em geral ou se classes distintas de problemas adaptam-se melhor a modelos paralelos distintos.

Nesse contexto, o antigo padrão para a computação paralela, o modelo *PRAM*[3], tem sido alvo de diversas críticas e adaptações devido à freqüente discrepância entre a complexidade teórica e os resultados experimentais obtidos. Essa discrepância é ditada sobretudo pelo alto custo de acesso à memória compartilhada, pelo tratamento da concorrência e da sincronização e pela distância do modelo às máquinas reais existentes[8, 22, 32].

O modelo *BSP*, proposto por L. G. Valiant em [32], pode simular otimamente alguns algoritmos em *PRAM* sobre sistemas paralelos com memória distribuída [33].

No entanto, o próprio L. G. Valiant afirma que pode ser interessante construir algoritmos sobre o modelo *BSP* que privilegiem a computação local e minimizem as operações globais, pois existem circunstâncias em que a simulação do *PRAM* não pode ser efetuada eficientemente [16]. Entre outras, se o fator g (= velocidade de computação local/largura da banda de comunicação)

[1] Em [22] encontra-se uma breve discussão sobre as suas diferenças para as máquinas reais.

[2] É preciso destacar a correção desta afirmação e que o modelo *RAM* possibilitou o desenvolvimento de toda uma teoria de complexidade de algoritmos.

[3] As vantagens e desvantagens do *PRAM* são discutidas no *Apêndice A*.

for alto[4], o que é verdade para a maioria das máquinas disponíveis atualmente [12]. A cada mensagem trocada ainda deve ser acrescentado um custo extra s, definido como a *latência do sistema.* Este custo pode ser tão grande a ponto de o o custo total extra da latência para todas as mensagens trocadas ter um impacto considerável sobre o *speedup* (que representa o ganho na execução do algoritmo) resultante da utilização de paralelismo[32].

Partindo de todas estas observações, F. Dehne e seus colaboradores utilizam em diversos trabalhos [4, 9, 10, 11, 12, 8] uma versão adaptada do modelo *BSP.* Esta versão, conhecida como *modelo CGM*, considera todos os parâmetros anteriormente discutidos e que afetam o *speedup* final observado sem, contudo, fazer hipóteses sobre g. O modelo *BSP*, bem como uma breve discussão sobre as suas vantagens e desvantagens, encontra-se na próxima seção. A sua descrição é necessária para a compreensão do modelo *CGM*, apresentado em seguida.

Além deste e dos trabalhos já citados, o modelo *CGM* também foi utilizado (explícita ou implicitamente) em algoritmos paralelos construídos para resolver diversos outros problemas, com destaque para [3, 13, 21, 28]. Alguns bons resultados práticos foram obtidos.

2.2 O modelo BSP

O modelo *BSP*, ou *Bulk-Synchronous Parallel*, foi proposto por L. G. Valiant em [32] como um modelo de propósito geral e um candidato a se tornar o padrão para a computação paralela, com resultados eficientes tanto para o projeto e execução prática de algoritmos quanto para implementações físicas em *hardware.*

O autor afirma que o sucesso do modelo de von Neumann para a computação seqüencial deve-se ao fato de este ser uma ponte eficiente entre o *software* e o *hardware*, dado que linguagens de alto nível podem ser facilmente compiladas neste modelo e ele pode ser facilmente implementado em *hardware.* Sugere-se que o *BSP* é o equivalente paralelo ao modelo de von Neumman [32, 35].

Em linhas gerais, o modelo *BSP* é formado por uma combinação de três atributos [32, 16, 34, 35], a saber:

- Um número p de *componentes*, onde cada componente é dada por um processador e uma memória local, de dimensões não especificadas pelo modelo;
- Um *roteador* para permitir a troca de mensagens *ponto-a-ponto* entre todos os pares de componentes;
- Primitivas para sincronizar um subconjunto (possivelmente completo) de componentes a

[4]Isto equivale a dizer que a largura da banda de comunicacão é pequena em relação à quantidade de dados a transmitir gerados pela intensa computação local.

intervalos regulares de unidades de tempo. A este intervalo dá-se o nome de *período* e é o tempo mínimo entre operações de sincronização sucessivas, onde cada unidade de tempo corresponde ao tempo necessário para se realizar uma operação local.

Uma *computação* no modelo *BSP* consiste de uma seqüência de *supersteps*, ou *superpassos*. Nestes, alocam-se tarefas às componentes constituídas por operações locais e trocas de mensagens com as demais componentes. A cada período de, digamos, L unidades de tempo, é feita uma verificação para determinar se um superpasso foi concluído por todas as componentes. Em caso afirmativo, executa-se o próximo superpasso. Se pelo menos uma das componentes não terminou sua tarefa, são alocadas mais L unidades de tempo ao superpasso, para que ela o faça. De acordo com este tratamento da sincronização, o autor admite uma *variação* para o modelo, que permite o término do superpasso em qualquer instante após decorridas as L primeiras unidades de tempo. Doravante vamos nos referir a ela simplesmente como *variant BSP*.

Esta forma de *sincronismo* ditada pela existência de L, onde as componentes trabalham assincronamente apenas entre intervalos de tempo determinados, é comumente denominada *barreira de sincronização*. Valiant sugere que ela seja preferencialmente fornecida por *hardware*, a fim de melhorar o desempenho e não onerar o programador. No entanto, o *BSP* permite que o valor de L seja controlado pelo programa, mesmo em tempo de execução. Nesse caso, a escolha de L deve considerar restrições de *software* e de *hardware*.

É interessante notar que o modelo não impõe restrições para o número de componentes às quais a *barreira de sincronização* deve ser aplicada, permitindo, inclusive, que pares de componentes se sincronizem independentes das demais. Isso pode ser extremamente benéfico, pois processos seqüenciais independentes não precisam ser retardados desnecessariamente.

Para analisar o desempenho de um computador *BSP* deve-se considerar o número de componentes, o período L e o tratamento da sincronização (definindo se é *variant BSP* ou não), o desempenho do roteador e a latência do sistema.

A análise da *latência do sistema*, ou *startup cost*, diz respeito ao tempo de resposta dos equipamentos envolvidos em cada operação. Para avaliar o desempenho do roteador[5] define-se uma *h-relação* como a fase do superpasso em que cada componente envia e recebe, no máximo, h mensagens. Seja b a largura da banda de comunicação[6] e seja $g = 1/b$. Cada *h-relação*, que no modelo *CGM* foi chamada de *all-to-all broadcast*[7], tem o custo de $gh + s$ unidades de tempo, onde s é a latência do sistema.

[5] A avaliação do desempenho do roteador é importantíssima, pois os maiores custos paralelos estão na comunicação entre as componentes do sistema.

[6] A largura da banda de comunicação é o parâmetro que define o fluxo de transferência de dados entre as componentes quando o roteador está em uso contínuo

[7] Protocolo de comunicação em que todos os processadores enviam e recebem dados de todos os outros processadores.

Considere um superpasso onde um algoritmo realiza, em cada componente, x operações locais a cada período de sincronização L e o tempo gasto para uma *h-relação* é $hg + s$. O custo do superpasso é dado por $\lceil \frac{\max(L, hg+s)}{L} \rceil$.

Usando *variant BSP* e os mesmos dados, o custo do superpasso é dado por $\max(L, hg + s)$.

Variant BSP é utilizada em [16] para a apresentação de diversos resultados interessantes. Destacam-se a simulação do modelo *PRAM* (à qual Valiant já havia se referido na apresentação do modelo) e diversos algoritmos, incluindo ordenação, eliminação de Gauss-Jordan e *parallel prefix*. O algoritmo de *parallel prefix* consiste no processamento de uma árvore em duas fases, sendo que na primeira percorre-se a árvore da raiz para as folhas e na segunda, das folhas para a raiz.

Uma vez avaliado o custo de cada superpasso, o número assintótico de superpassos conclui a análise do algoritmo.

Os resultados até agora apresentados não são conclusivos sobre a força prática do modelo, mas a sua importância é indiscutível. Por não se deter a detalhes de *hardware* e de *software*, o modelo facilita o projeto de algoritmos enquanto universaliza os resultados para diversas arquiteturas e não amarra o desenvolvimento de um ao desenvolvimento do outro. Também são louváveis a proposta de realismo do modelo e a tentativa cuidadosa de quantificação dos custos, sobretudo os de comunicação, um dos grandes problemas reais enfrentados pela computação paralela.

Os objetivos do *BSP* são pouco modestos, pois pretende fornecer um padrão para a computação paralela e simultaneamente permitir a resolução de problemas de qualquer tipo. Satisfazendo as expectativas, pode certamente causar uma revolução na ciência da computação. No entanto, para fazer calar os críticos, entre eles J. Wiedermann [36], que fala sobre os atuais resultados práticos desanimadores e a *parallel computing crisis*, por ele definida, ainda é necessário muito tempo e trabalho. Por enquanto, ser um grande referencial para pesquisa ainda é uma das principais qualidades do modelo *BSP*.

2.3 O modelo *CGM*

A definição do modelo *CGM*, ou *Coarse-Grained Multicomputer*, para computadores paralelos com memória distribuída pode ser encontrada em [9, 10, 11, 12, 8, 28]. O modelo foi proposto por F. Dehne inspirado no *BSP* [32], segundo afirma o próprio autor. Sua construção se baseia nas observações de L. G. Valiant (autor do *BSP*) sobre algoritmos que privilegiem a computação local e minimizem as operações globais, conforme discutimos anteriormente. O *CGM* se propõe a ser para o *BSP* o mesmo que o modelo *RAM* é para as *máquinas de von Neumman.*

Seja n o tamanho da entrada e considere um computador paralelo com p processadores, diga-

mos $P_1, P_2, \ldots, P_p$, dispostos em uma rede de comunicação arbitrária como na figura 2.1. Cada processador possui uma memória local de tamanho muito superior a $O(1)$. Em PRAM, freqüentemente requer-se $p = O(n)$. Aqui, consideramos $p << n$, que é o caso usual para as máquinas reais existentes. Argumenta-se que a tendência é que a granularidade das máquinas paralelas com memória distribuída aumente, mesmo para as MPP (*Massively Parallel Processors*) [24]. O termo *granularidade* refere-se à quantidade de operações locais executadas por um processador sem se comunicar com os demais[8].

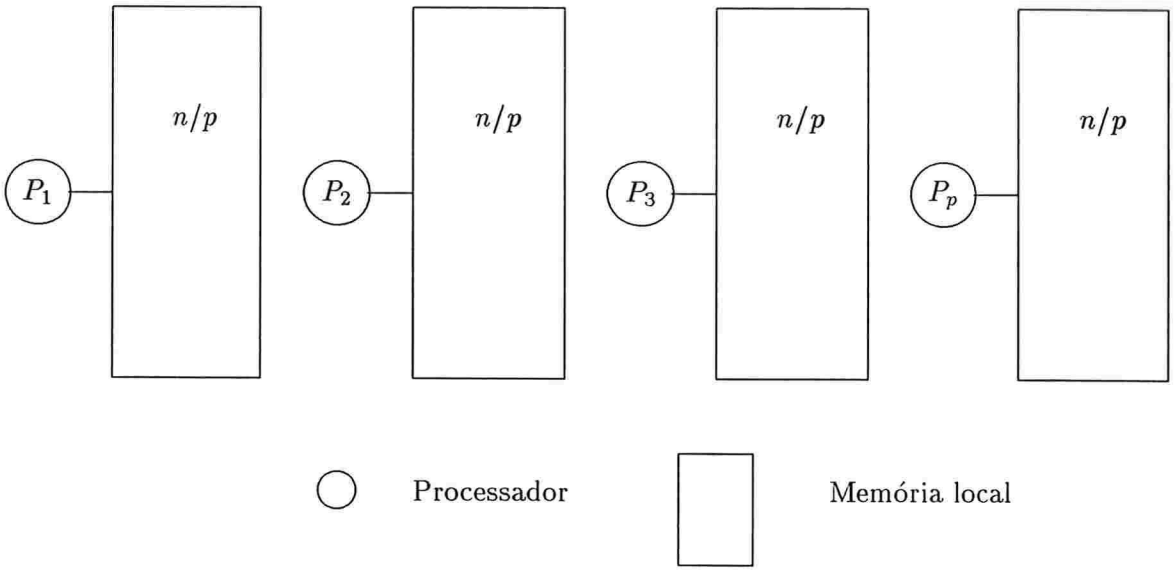

Figura 2.1: Um multiprocesador com memória distribuída

Os algoritmos no modelo *CGM* consistem da alternância entre uma rodada de computação local e uma rodada de comunicação. Se R é o número de rodadas, sugere-se que os algoritmos obedeçam ao seguinte esquema genérico [28]:

[8] *Granularidade grossa*, no caso, significa que a quantidade de computação local deve ser *grande*.

para $i = 1$ **até** R **faça**

{

(**rodada de computação local**)$_i$

(**rodada de comunicação**)$_i$

{

Uma *rodada de computação local* no modelo *CGM* equivale à computação de um superpasso no modelo *BSP* [28]. Devemos utilizar o melhor algoritmo seqüencial possível em cada processador e tratar os seus dados localmente.

Uma *rodada de comunicação*, por sua vez, equivale à uma *h-relação* no modelo *BSP*, com $h \leq O(n/p)$. Isto significa que cada processador pode enviar e receber $O(n/p)$ dados a cada rodada. Adicionalmente, o modelo*CGM* exige que todos os dados enviados de um dado processador para outro em uma rodada sejam *empacotados* em uma mensagem.

A complexidade de um algoritmo paralelo é dada principalmente pelo número de rodadas, de forma que o esforço para reduzi-la consiste em reduzir o número destas rodadas. Assim, o objetivo do modelo *CGM* é construir algoritmos que requerem uma pequena quantidade de rodadas, minimizando R. Em muitos algoritmos para problemas geométricos [9, 10, 8] obteve-se R constante ou $O(\log p)$. Como p é usualmente pequeno comparado a n, podemos obter resultados práticos eficientes [12]. Além das vantagens evidentes, este critério aumenta a portabilidade entre diferentes arquiteturas paralelas [12, 14, 32, 33].

Finalmente, temos que o modelo *CGM* também se adequa ao projeto de algoritmos paralelos escaláveis [12, 28], o que significa que deve produzir resultados eficientes para um grande intervalo de valores de n/p [28], pois a velocidade da computação local é consideravelmente maior que a velocidade da comunicação entre os processadores nas máquinas atuais.

2.4 *List ranking* no modelo *CGM*

Encontrar algoritmos paralelos para *list ranking* no modelo *CGM* foi um problema originalmente proposto em [12]. Definido o modelo *CGM* na seção anterior, passemos ao problema cujo estudo é o objetivo deste trabalho.

Uma *lista ligada* é um conjunto ordenado e finito de elementos denominados *nós*, onde cada nó contém um apontador para outro, que é o seu *sucessor* na lista [19]. O *apontador* pode ser o endereço de memória do nó ou alguma outra informação que permita localizá-lo [19, 30]. Para este trabalho, suponha que o último nó, denominado λ, aponta para si mesmo.

Seja L uma lista ligada como na figura 2.2 e seja $x \in L$ um nó. A *distância* de x a λ para

$x \neq \lambda$, denotada por $dist(x, \lambda)$, é dada pelo número de nós sucessores de x em L, incluindo λ. Por definição, $dist(\lambda, \lambda) = 0$. Se L tem n nós e x_1 é o primeiro nó de L, $dist(x_1, \lambda) = n - 1$.

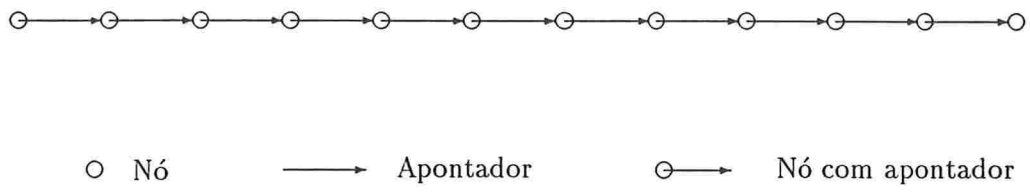

Figura 2.2: Uma lista ligada

Na literatura, $dist(x, \lambda)$ algumas vezes é chamada de *posto* ou *rank* de x. Daí, o problema de determinar a distância de todos os nós de uma lista ligada L a λ ter ficado conhecido como *list ranking*, ou LR, como será futuramente referenciado.[9] Formalmente, o LR consiste em determinar $dist(x, \lambda)$ para cada nó x de L.

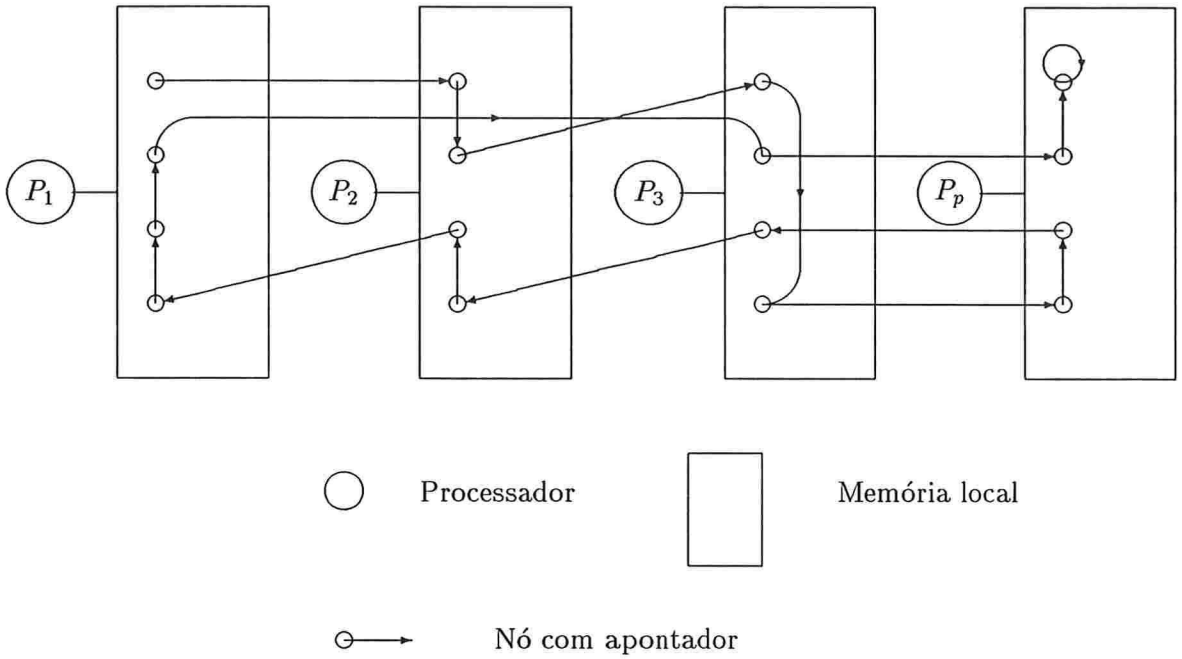

Figura 2.3: Uma lista ligada armazenada em um multiprocessador de memória distribuída

Um algoritmo seqüencial trivial resolve o problema em $O(n)$, simplesmente percorrendo a

[9]Tentamos encontrar uma tradução adequada para *list ranking*, mas após ouvirmos a sugestão do Marcelo Gomes de Queiroz de chamá-lo de *problema de empostamento de uma lista*, desistimos...

lista. Vários algoritmos em *PRAM* foram propostos [6, 18, 23].

Suponha o modelo *CGM* com p processadores e a lista ligada L com n nós, arbitrariamente distribuídos entre os p processadores, como na figura 2.3. Para o *LR*, cada nó $x \in L$ localiza seu sucessor através do apontador, que deve informar em qual processador este se encontra, bem como sua localização na memória do mesmo. Para os algoritmos aqui apresentados, suponha também que a distribuição dos nós pelos processadores faz parte da entrada do problema e que cada processador inicialmente possui $O(n/p)$ dos nós da lista L.

Vale notar que uma solução eficiente para o *LR* permite a resolução de vários outros problemas. Em particular, pode ser interessante associá-lo à técnica de *Circuito de Euler* [1, 2, 4], produzindo resultados interessantes envolvendo teoria dos grafos. No próximo capítulo, são apresentados os algoritmos probabilísticos para *LR* no modelo *CGM*.

CAPÍTULO 3

Algoritmos probabilísticos

A escolha de amostras ao acaso, ou *random sampling*, é uma técnica poderosa para a obtenção de limites assintóticos eficientes para algoritmos paralelos, apresentando também bons resultados práticos [17]. Pela sua importância neste trabalho, iniciamos o capítulo por sua breve descrição.

Em seguida, apresentamos dois algoritmos probabilísticos para LR no modelo CGM baseados na escolha de amostras ao acaso. Seja n o tamanho da lista e seja p o número de processadores. O primeiro algoritmo foi publicado por F. Dehne e S. W. Song em [12] e, com alta probabilidade, resolve o LR em $O(\log p + \log\log n)$ rodadas e $O(n/p)$ operações de computação local por rodada. O segundo algoritmo resolve o LR em um número esperado de $O(n/p)$ operações de computação local por rodada e $O(\log p)$ rodadas de comunicação. Este algoritmo é original e consiste na principal contribuição desta dissertação. O *Prof. Dr. Yoshiharu Kohayakawa*[1] colaborou na sua construção.

3.1 Escolha de amostras ao acaso para a construção de algoritmos

Esta seção foi inspirada em [17].

Diversos resultados recentes na teoria de algoritmos paralelos mostram que a *escolha de amostras ao acaso*, ou *random sampling*, é uma técnica poderosa para a obtenção de limites assintóticos eficientes na análise destes algoritmos. Em [17] mostra-se *experimentalmente* a eficiência prática desta técnica.

A técnica consiste na escolha ao acaso de uma amostra relativamente pequena da entrada sobre a qual se executa um algoritmo possivelmente ineficiente (freqüentemente são utilizados algoritmos do tipo *força bruta*, em que todas as combinações possíveis são testadas). As infor-

[1]Professor Associado do Instituto de Matemática e Estatística da Universidade de São Paulo.

mações resultantes desta execução servem para subdividir a entrada e, utilizando agora as partes obtidas como entrada, pode-se aplicar um novo algoritmo ou fazer chamadas recursivas até que o problema esteja resolvido[2].

Intuitivamente, pode-se observar facilmente que os algoritmos assim construídos são eficientes. Independente do algoritmo aplicado à amostra, o problema inicial tem um tamanho *pequeno* e, através de análises probabilísticas garante-se que a aplicação do algoritmo inicial subdivide a entrada em partes cujo tamanho esperado é igualmente *pequeno*. O resultado é uma análise de complexidade favorável para o algoritmo como um todo. Além disso, como a amostra é escolhida ao acaso, os valores esperados calculados valem independentemente das hipóteses sobre a distribuição da entrada. Deve-se notar que, no entanto, sem meios de se determinar uma amostra *adequada* a técnica não é realmente útil. Os algoritmos probabilísticos apresentados neste trabalho são bons exemplos desta afirmação.

Um exemplo seqüencial clássico utilizando chamadas recursivas é o algoritmo probabilístico *quicksort* para ordenação de dados [7]. Em [17] encontra-se uma longa lista de publicações que se utilizam da escolha de amostras aleatórias como método de projeto de algoritmos.

Vale notar que subdividir o problema em partes *pequenas* é extremamente interessante para o projeto de algoritmos paralelos, sobretudo em modelos de computação com memória distribuída como o *CGM*, pois fornece aos problemas um paralelismo natural facilmente explorável. No entanto, a técnica não se restringe a estes modelos. Em [23], por exemplo, é apresentado um algoritmo ótimo para *LR* no modelo *CRCW PRAM*. Este algoritmo serviu como inspiração para o *algoritmo 1* apresentado na *seção 3.2*.

A possibilidade de fornecer também resultados práticos satisfatórios torna esta técnica ainda mais atraente, sendo recomendável uma atenção especial dos pesquisadores da área a ela e ao respectivo ferramental matemático para seu uso.

3.2 Algoritmo 1

O algoritmo para *LR* apresentado nesta seção consiste, inicialmente, na escolha ao acaso de uma amostra de nós da lista ligada através de um sorteio independente e local aos processadores. Esta amostra vai dividir a lista em partes *pequenas*, chamadas de *sublistas*, onde cada parte será resolvida utilizando *pointer jumping*. Na etapa final, os resultados serão combinados. O algoritmo e a respectiva análise foram descritos por F. Dehne e S. W. Song em [12], trabalho a partir do qual esta seção foi escrita.

[2] Em [7], técnica semelhante é apresentada com o nome de *divisão e conquista*.

Pointer jumping, *standard recursive doubling* ou *salto de ponteiros* é uma técnica bastante simples que consiste em, a cada iteração, fazer com que o sucessor de cada nó passe a apontar para o sucessor desse sucessor. Considerando x um nó de uma lista ligada qualquer e *next(x)* o seu sucessor, a técnica pode ser descrita pelo seguinte esquema:

Se $next(x) \neq x$

entao $next(x) = next(next(x))$.

Mais referências a técnica ou a algoritmos para LR que dela se utilizem, inclusive em outros modelos paralelos, podem ser encontradas em [7, 18, 22, 23].

Suponha o modelo CGM com p processadores e uma uma lista ligada L com n nós arbitrariamente distribuídos entre os processadores, como no capítulo anterior. Adote a seguinte convenção: um nó y está *à direita* de x se $dist(y, \lambda) < dist(x, \lambda)$, *i. e.*, se x está mais próximo do fim da lista.

Seja $L' \subset L$ uma amostra escolhida ao acaso a partir dos nós de L através de um sorteio independente. A cada nó $x \in L'$ dá-se o nome de *pivô*, portanto a amostra L' também pode ser vista como o *conjunto dos pivôs*. λ é sempre escolhido como pivô e, para os demais nós de L, a probabilidade de cada um ser escolhido como pivôs é igual a $1/p$.

Para cada nó $x \in L$, seja $nextPivot(x, L')$ o pivô mais próximo encontrado à direita de x em L. $nextPivot(\lambda, L') = \lambda$ (pela nossa definição de lista ligada) e para $x \neq \lambda$, $nextPivot(x, L') \neq x$.

Ainda considerando cada nó $x \in L$, $distToPivot(x, L')$ é a *distância* entre x e $nextPivot(x, L')$. Seja m a *distância máxima* entre os nós $x \in L$ e os seus respectivos $nextPivot(x, L')$, $m = \max_{x \in L} distToPivot(x, S')$. Note que os pivôs dividem a lista L em sublistas que terminam em um pivô (tal como a lista original) e que estas sublistas têm comprimento limitado por n, *i. e.*, $m \leq n$.

O *list ranking modificado*, ou LRM, para os nós da lista L com respeito aos pivôs de L' consiste em determinar, para cada nó $x \in L$, o pivô à direita de x, $nextPivot(x, L')$ e a respectiva distância $distToPivot(x, L')$. Observe que as estruturas de entrada e saída de dados para o LRM podem ser as mesmas que as utilizadas para o LR original.

Resolvido o LRM, cada nó $x \in L$ (inclusive os próprios pivôs $x \in L'$) aponta para $nextPivot(x, L')$. Combinando-se os resultados obtidos, constrói-se uma lista ligada apenas com estes pivôs.

Com alta probabilidade, esta lista tem tamanho $O(n/p)$, portanto pode ser colocada em um único processador e resolvida localmente com um algoritmo seqüêncial trivial. Um possível algoritmo consiste em percorrer a lista duas vezes, sendo a primeira para armazenar as distâncias relativas entre os nós consecutivos e descobrir o λ. Na segunda passada as distâncias relativas entre cada pivô e λ devem ser atualizadas, resolvendo o LR para os pivôs $x \in L'$.

A partir daí, a resolução do LR para a lista L segue trivialmente, pois cada nó que não é pivô ($x \in L - L'$) conhece a distância $distToPivot(x, L')$ e aponta para o respectivo $nextPivot(x, L')$ que, por sua vez, conhece a distância $distToPivot(x, \lambda)$ e aponta para λ.

O algoritmo para LR é apresentado a seguir.

Algoritmo 1

(1) Ao acaso, é selecionada uma amostra $L' \subset L$ com $O(n/p)$ pivôs (com alta probabilidade) como segue: cada processador faz, para cada nó $x \in L$ nele armazenado, um sorteio independente que o escolhe como pivô com probabilidade igual a $1/p$. (Se o sucessor do nó for ele mesmo, então esse nó é o λ e deve ser escolhido.)

(2) Utilizando *pointer jumping*, os processadores resolvem coletivamente o LRM, determinando $nextPivot(x, L')$ e $distToPivot(x, L')$ relativos aos pivôs de L' (e não a λ), para todo nó $x \in L$.

(3) Utilizando *all-to-all broadcast*, os valores $nextPivot(x, L')$ e $distToPivot(x, L')$ de todos os pivôs $x \in L'$ são enviados a todos os processadores.

(4) Com os dados recebidos na etapa anterior, cada processador resolve seqüencialmente o LR para os nós $x \in L$ nele armazenados (em tempo $O(n/p)$).

Note que pelo menos o último nó sempre é escolhido como pivô e portanto o LRM está bem definido. Note também que, na *etapa (4)*, todos os processadores têm localmente disponíveis os dados necessários para construir a lista L' de pivôs.

No *apêndice B* apresentamos uma sugestão para a implementaçao deste algoritmo[3].

Para a análise do algoritmo, adotamos as seguintes notações:

- se X é uma variável aleatória, $E(X)$ é a *esperança* de X.
- se A é um evento, Prob$\{A\}$ é a probabilidade de A ocorrer.

[3] A implementação de algoritmos paralelos no modelo CGM, em geral, constitui um problema à parte, dado o alto grau de complexidade pelo número de fatores envolvidos.

- L_i representa a sublista i e $|L_i|$ = número de nós em L_i.
- $\max_{1 \leq i \leq n} |L_i|$ é o tamanho da maior sublista L_i, para todo $1 \leq i \leq n$.
- $\ln n$ é o logaritmo *natural* de n (na base e) e $\log n$ é o logaritmo de n na base 2.

Seja A um algoritmo cuja complexidade é dada por $O(f(n))$ rodadas *com alta probabilidade*. Se R é o número de rodadas observado, isso equivale a dizer que, para todo $c > c_0 > 1$, $\text{Prob}\{R \geq f(n)\} \leq \frac{1}{n^{g(c)}}$, onde c_0 é uma constante fixa e $g(c)$ é um polinômio em c com $g(c) \to \infty$ quando $c \to \infty$ [12].

A *etapa (1)*, com alta probabilidade, deve escolher $O(n/p)$ pivôs dos nós de L e segue do lema abaixo [25]. No lema, uma *tentativa* aplica-se ao sorteio independente para selecionar os pivôs e um *sucesso* é obtido quando um pivô é escolhido.

Lema 3.2.1 *Considere uma variável aleatória X com distribuição binomial. Seja n o número de tentativas, q a probabilidade de sucesso em cada tentativa e $c > 1$ real. Então $E(X) = nq$ e $Prob\{X > cnq\} \leq e^{-\frac{1}{2}(c-1)^2 nq}$.*

O lema a seguir mostra que, se escolhemos n/p pivôs de L, então, com alta probabilidade, estes pivôs dividem L em sublistas cujo tamanho máximo é limitado por $3p \ln n$.

● Nó escolhido ao acaso, chamado pivô

Figura 3.1: Uma lista ligada linear com pivôs escolhidos.

Lema 3.2.2 *$xk \leq n$ pivôs escolhidos independentemente e ao acaso na lista L dividem-na em sublistas L_i tais que o tamanho da maior sublista é limitado por n/x com probabilidade pelo menos $1 - 2x(1 - \frac{1}{2x})^{xk}$.*

Prova. (Análoga a [3].)

Suponha que os nós de L estão ordenados pelo *rank*, *i. e.*, pela posição na lista.

A lista ordenada pode ser vista como $2x$ segmentos de tamanho $n/2x$. Se todo segmento contém pelo menos um pivô, então $\max_{1 \leq j \leq xk} |L_j| \leq \frac{n}{x}$.

Considere um segmento.

Desde que os pivôs são escolhidos ao acaso, a probabilidade de um pivô específico não estar no segmento é $(1 - \frac{1}{2x})$.

Como a escolha é independente e temos xk pivôs, a probabilidade de nenhum dos pivôs estar no segmento é $(1 - \frac{1}{2x})^{xk}$.

Portanto, mesmo supondo exclusão mútua, a probabilidade de que exista um segmento que não contém pivôs é no máximo $2x(1 - \frac{1}{2x})^{xk}$. Então todo segmento contém pelo menos um pivô com probabilidade pelo menos $1 - 2x(1 - \frac{1}{2x})^{xk}$. ■

Corolário 3.2.3 *$xk \leq n$ pivôs escolhidos ao acaso dividem a lista L em $xk + 1$ sublistas L_i tais que existe uma sublista L_i de tamanho maior que $c\frac{n}{x}$ com probabilidade no máximo $\frac{2x}{c}(1 - \frac{c}{2x})^{xk} \leq \frac{2x}{c}e^{-\frac{1}{2}ck}$.*

Lema 3.2.4 *Considere $xk \leq n$ pivôs escolhidos ao acaso e que dividem a lista L em $xk + 1$ sublistas L_i. Seja $m = \max_{0 \leq i \leq xk} |L_i|$. Se $k \geq \ln(x) + 2\ln(n)$, então $Prob\{m > c\frac{n}{x}\} \leq \frac{1}{n^c}$, $c > 2$.*

Prova. O *corolário 3.2.3* implica que $\text{Prob}\{m > c\frac{n}{x}\} \leq \frac{2x}{c}\text{e}^{-\frac{1}{2}ck}$.

Observamos que $\ln(x) + 2\ln(n) \leq k \Rightarrow \frac{2}{c}\ln(\frac{2x}{c}) + 2\ln(n) \leq k, para c > 2$.

$\Rightarrow \ln(\frac{2x}{c}) + c\ln(n) \leq \frac{ck}{2} \Rightarrow \frac{2x}{c}n^c \leq \text{e}^{\frac{ck}{2}} \Rightarrow \text{Prob}\{m > c\frac{n}{x}\} \leq n^{-c}$ ■

Teorema 3.2.5 *n/p pivôs escolhidos ao acaso através de um sorteio independente dividem L em $n/p + 1$ sublistas L_j com $m = \max_{0 \leq j \leq p} |L_j|$ e tal que $Prob\{m \geq c3p\ln(n)\} \leq \frac{1}{n^c}, c > 2$.*

Prova. Seja $x = \frac{n}{3p\ln(n)}$ e $k = \ln(x) + 2\ln(n) = 3\ln(n) - \ln(3p\ln(n))$.

Então $xk = \frac{n}{p}\frac{3\ln(n) - \ln(3p\ln(n))}{3\ln(n)} \leq \frac{n}{p}$ e o *teorema 3.2.5* segue do *lema 3.2.4*. ■

Os sorteios realizados na *etapa (1)* são independentes, logo é trivial que os resultados valem para uma lista ligada no modelo *CGM*.

Na *etapa (2)*, a cada iteração do *LRM* o *pointer jumping* faz com que a distância entre $x \in L$ e $nextPivot(x, L')$ dobre, o que significa que o número de rodadas fica limitado por $\log m$, onde m é a maior distância esperada entre dois pivôs consecutivos. Do *teorema 3.2.5*, com alta probabilidade, temos que $m \leq 3p \ln n$. Então a *etapa (2)* requer do *algoritmo 1* requer, com alta probabilidade, no máximo $\log(3p \ln n) = \log(3p) + \log \ln n$ rodadas.

A *etapa (3)* requer uma rodada de comunicação e a *etapa (4)* pode ser resolvida localmente. O teorema abaixo conclui a análise do algoritmo.

Teorema 3.2.6 *Seja n o tamanho da lista e seja p o número de processadores.* O algoritmo 1 *resolve o* LR *no modelo* CGM, *com alta probabilidade, em* $1 + \log(3p) + \log \ln n = O(\log p + \log \ln n)$ *rodadas e* $O(n/p)$ *operações de computação local por rodada.*

3.3 Algoritmo 2

A análise do *algoritmo 1* mostra que a sua complexidade é determinada pela maior distância esperada entre dois pivôs consecutivos. Essa distância, por sua vez, é determinada pela escolha da amostra. Logo, para melhorar essa complexidade é preciso obter uma nova amostra onde as distâncias entre os pivôs não sejam muito *grandes*.

Seja n o tamanho da lista e seja p o número de processadores. Sem ferir a restrição de que o número total de pivôs deve ser $O(n/p)$, obtemos uma nova amostra acrescentando alguns pivôs à amostra escolhida pelo *algoritmo 1*. Com essa amostra, o algoritmo seguinte resolve o *LR* em um número esperado de $O(n/p)$ operações de computação local por rodada e $O(\log p)$ rodadas de computação.

Este resultado merece destaque pois permite que o problema seja resolvido independentemente do tamanho da lista (resultado esperado). Este algoritmo é original e consiste em uma importante contribuição da dissertação.

Suponha L, L', as notações e as demais definições da seção anterior. A convenção de que um nó y está *à direita* de x se $dist(y, \lambda) < dist(x, \lambda)$, continua válida. Analogamente, um nó y está *à esquerda* de x se $dist(y, \lambda) > dist(x, \lambda)$.

Preto, *branco* e *vermelho* indicam os diferentes estados dos nós, sendo que o *preto* é atribuído aos pivôs, o *branco* aos não-pivôs e o *vermelho* é uma cor auxiliar utilizada no primeiro instante para compor a amostra e, em seguida, para diferenciar os demais nós dos pivôs.

Para marcar os nós *vermelhos* na *etapa (2)* utilizando *pointer jumping*, cada nó *preto* ou

vermelho deve ordenar a marcação do sucessor em *vermelho* até a penúltima rodada, propagando as cores adequadamente pela lista.

O algoritmo é apresentado a seguir.

Algoritmo 2

(1) Execute a *etapa (1)* do *algoritmo 1* e defina L''. Marque como *pretos* os nós $x \in L''$ e como *brancos* os demais nós da lista (*i. e.*, os nós $x \in L \wedge x \notin L''$). Marque o primeiro nó da lista como *preto.*

(2) Cada processador P_i faz as seguintes operações, para cada nó $x \in L$ armazenado em P_i:

(2a) Se x é um nó *preto*, então todos os nós *brancos* à direita de x e com distância $\leq p \ln p - 1$ são marcados como *vermelhos.* Os nós que eram anteriormente *pretos* permanecem *pretos.*

(2b) Os nós que permanecem *branco* após a *etapa (2a)* são marcados como *pretos.*

(3) Defina L' como o subconjunto de nós *pretos* em L obtidos após o término da *etapa (2).* Os nós pretos são agora denominados pivôs. Os nós *vermelhos* passam a representar os demais nós da lista ($x \in L \wedge x \notin L'$). Continue com as *etapas (2) a (4)* do *algoritmo 1.*

Considerando que o número de pivôs não diminui e as observações feitas na seção anterior, temos que o número esperado de pivôs permanece $O(n/p)$, o que é necessário para permitir que as duas etapas finais (*etapas (3) e (4)* do *algoritmo 1*) sejam resolvidas em um número esperado de $O(n/p)$ operações de computação local por rodada.

Sejam X, Y e Z variáveis aleatórias definidas como segue:

- X é o número de nós em L'' escolhidos na *etapa (1)*;
- Y é o número de nós que permanecem *brancos* após a *etapa (2a)*;
- $Z(= X + Y)$ é o número total de pivôs escolhidos na *etapa (2).*

Pelo *lema 3.2.1* da seção anterior, $E(X) = n/p$.

Vamos estimar $E(Y)$.

Na *etapa (1)*, os pivôs são escolhidos ao acaso e a probabilidade de um nó ser escolhido é $1/p$. Então a probabilidade de um nó não ser escolhido inicialmente como pivô é $(1 - 1/p)$.

Fixado o conjunto L'' dos pivôs, executa-se a *etapa (2a)*. Um nó permanece *branco* se não foi escolhido como pivô e se nenhum dos $p \ln p - 1$ nós à sua esquerda pertencem a L''. Então, a probabilidade de um nó permanecer *branco* é $(1 - 1/p)^{p \ln p}$.

Fato 3.3.1 *Para x real, temos que* $1 - x \leq e^{-x}$.

Da *fato 3.3.1*, segue que:

$(1 - 1/p)^{p \ln p} \leq e^{-\ln p} = p^{-1} = 1/p$.

Se a probabilidade de um nó permanecer *branco* é $\leq 1/p$, então, como a lista tem n nós, $E(Y) \leq n/p$. O número total esperado de pivôs ao término da *etapa (2)* é $E(Z) = E(X + Y) = E(X) + E(Y) \leq n/p + n/p = 2n/p$, portanto $E(Z) = O(n/p)$, como queríamos.

O teorema a seguir conclui a análise do *algoritmo 2*.

Teorema 3.3.2 *Seja n o tamanho da lista e seja p o número de processadores.* O algoritmo 2 *resolve o* LR *no modelo* CGM *em um número esperado de $O(n/p)$ operações de computação local por rodada e* $\log(p \ln p) = O(\log p)$ *rodadas de comunicação.*

Prova.

Trivialmente, a realização da *etapa (2)* pode ser feita em $O(n/p)$ operações de computação local por rodada.

Na seção anterior e pelo número de pivôs após a *etapa (2)*, temos que as *etapas (1) e (4)* do *algoritmo 1* são realizadas localmente em um número esperado de $O(n/p)$ operações de computação local por rodada e a *etapa (3)* custa uma rodada de comunicação.

Por construção, após a *etapa (2)* deste segundo algoritmo, a distância entre os pivôs estará limitada a $p \ln p$. Como o *LRM* (*etapa (2)* do *algoritmo 1*) pode ser resolvido utilizando *pointer jumping*, este passo custa $\log(p \ln p) = \log p + \log \ln p = O(\log p)$.

Da mesma forma, a *etapa (2)* do *algoritmo 2* pode ser resolvida em $O(\log p)$ rodadas. ■

3.4 Resultados obtidos

Todos os dados constantes dessa seção foram obtidos utilizando a máquina paralela *Parsytec PowerXplorer 16/32 Parallel Computing System.* A máquina em questão apresenta 16 nós sob uma topologia 2-D (*grade*). Cada nó consiste de um processador de aplicação *PowerPC601* (80 *MHz*) e um processador de comunicação *T805*, com memória local de 32 *MBytes.* A taxa de transferência de dados entre os nós é da ordem de milisegundos.

No *apêndice 2*, apresentamos uma implementação deste algoritmo, baseada em [11][4]. As tabelas a seguir mostram os resultados obtidos com a execução do programa referente ao *algoritmo 1.*

Nas tabelas, p é o número de processadores, n é o número de elementos e n^o é o número da execução. Para cada valor de n, ao acaso geramos e distribuímos listas ligadas entre os processadores. Essas listas foram utilizadas como entrada para a obtenção dos resultados. Os valores tabelados representam o número de rodadas necessárias para resolver o *LR* no modelo *CGM.* A cada valor apresentado está somado um número fixo de duas rodadas extras, necessárias para a implementação (descrita no *apêndice 2*). Vale lembrar que essas rodadas não alteram a complexidade do algoritmo.

Observe os resultados obtidos com a execução do algoritmo para quatro e oito processadores.

p = 4

n^o \ n	4	8	16	32	64	128	256	512	1024	2048	4096
1	5	7	8	8	8	8	9	9	9	9	9
2	6	7	7	7	8	8	8	8	10	10	10
3	5	7	7	7	8	9	8	9	9	9	10
4	5	7	7	7	7	9	9	9	9	10	9
5	5	7	7	7	8	9	9	9	9	10	10

n^o \ n	8192	16384	32768	65536	131072	262144
1	9	9	10	10	10	10
2	10	10	10	10	10	10
3	10	10	10	11	10	10
4	9	10	10	11	10	10
5	10	10	10	10	10	10

[4] A implementação deste algoritmo, feita em parceria com Mário Y. Yamauti, recebeu *menção honrosa* no *II IME SOFTEST*, concurso de *software* do Instituto de Matemática e Estatística da Universidade de São Paulo, realizado em outubro de 1996.

p = 8

n^o \ n	8	16	32	64	128	256	512	1024	2048	4096
1	7	7	8	8	10	10	10	10	10	11
2	7	7	8	8	10	11	11	11	11	11
3	6	8	8	8	10	10	10	10	10	11
5	7	8	8	9	9	10	10	10	10	11
6	7	8	8	9	10	10	10	10	11	11

n^o \ n	8192	16384	32768	65536	131072	262144
1	11	11	11	11	11	11
2	11	11	11	11	11	11
3	11	11	11	11	11	11
5	11	11	11	11	11	11
6	11	11	11	11	11	12

As tabelas *melhores resultados* e *piores resultados* mostram, respectivamente, o menor e o maior número de rodadas obtidas nas tabelas anteriores para os diversos valores de n. Para cada n fixo podemos observar a pequena variação entre estes valores.

Melhores Resultados

p \ n	4	8	16	32	64	128	256	512	1024	2048	4096
4	5	7	7	7	7	8	8	8	9	9	9
8	-	6	7	8	8	9	10	10	10	10	11

p \ n	8192	16384	32768	65536	131072	262144
4	9	9	10	10	10	10
8	11	11	11	11	11	11

Piores Resultados

p \ n	4	8	16	32	64	128	256	512	1024	2048	4096
4	6	7	8	8	8	9	9	9	10	10	10
8	-	7	8	8	9	10	11	11	11	11	11

p \ n	8192	16384	32768	65536	131072	262144
4	10	10	10	11	10	10
8	11	11	11	11	11	12

Para o *algoritmo 2*, fizemos os mesmos testes para $p = 4$ e obtivemos um número de rodadas constante e igual a 7 (utilizamos o teto de $p \ln p$), para n variando de 4 a 262144, incluídas as rodadas intrínsecas à implementação. Nesse caso os resultados são óbvios, pois o número de rodadas é especificado na implementação.

Ao executar os algoritmos paralelos percebemos, mesmo sem efetuar medidas precisas, que os tempos de execução são extremamente *grandes*, sobretudo se comparados aos obtidos com uma implementação seqüencial. Por exemplo, para os dois maiores valores de n tabelados (131072 e 262144) a resolução do LR em paralelo leva um tempo da ordem de *minutos* (pelo menos 5), enquanto o seqüencial fornece a resposta *quase* imediatamente (o tempo de execução não é perceptível). Observamos, contudo, que o tempo de execução do *algoritmo 2* diminui em relação ao do *algoritmo 1*, conforme aumenta o tamanho da entrada.

Precisamos dizer que as otimizações para realizar a comunicação entre os nós, usuais em implementações paralelas, não estão sendo efetuadas, pois o objetivo da implementação é apenas validar os resultados de acordo com o modelo *CGM*, *i. e.*, mensurar o número de rodadas de comunicação.

Contudo, dada a discrepância entre os tempos paralelo e seqüencial observados, não cremos que tais otimizações pudessem produzir resultados suficientemente satisfatórios para obter um *speedup* maior que 1, se comparado o algoritmo paralelo ao seqüencial. Isto porque na *Parsytec* a taxa de transferência de dados entre os nós é considerada muito baixa, sobretudo para aplicações semelhantes ao LR, onde a necessidade de comunicação freqüente é ditada pelo alto grau de dependência entre os dados.

Até poderíamos tentar utilizar outras máquinas e realizar implementações otimizadas a fim de buscar melhores resultados, mas cremos que esta tarefa escapa ao contexto do trabalho. De qualquer forma, as observações acima indicam a existência de variáveis que interferem nos tempos de execução dos algoritmos e que não são consideradas pelo modelo *CGM*. Uma vez que todas as restrições teóricas foram observadas, os dados obtidos para as rodadas são perfeitamente confiáveis e as tabelas fornecem uma visão adequada do desempenho dos algoritmos, embora as nossas implementações sejam simples. Considerando a avaliação de complexidade do modelo *CGM*, os dados mostram um desempenho satisfatório para os algoritmos, tendo sido o LR resolvido de acordo com número de rodadas esperado.

CAPÍTULO 4

Conclusão

Algoritmos paralelos teoricamente eficientes freqüentemente apresentam resultados práticos que deixam a desejar. Em parte, isso se deve à inexistência de um modelo conveniente de paralelismo que esteja suficientemente próximo das máquinas reais para permitir uma predicção razoável dos desempenhos dos algoritmos.

Pelos motivos discutidos no *apêndice A*, que incluem a dificuldade de se implementar o sincronismo, o custo de acesso à memória compartilhada e a distância do modelo às maquinas existentes, o problema é obvio para algoritmos em *PRAM*. Lembramos que, até bem pouco tempo atrás, o *PRAM* era o modelo padrão para o projeto de algoritmos paralelos e foi amplamente estudado.

No entanto, a freqüente discrepância entre a complexidade teórica e os tempos experimentais obtidos com o *PRAM* nos últimos anos, aliada ao igualmente freqüente desapontamento face aos valores absolutos desses tempos (comparados aos seqüenciais) fizeram com que a questão passasse a ser amplamente discutida, resultando no surgimento de novos modelos paralelos.

Há quase um consenso a respeito das características que um bom modelo paralelo deve ter. A lista de tais características inclui desde previsões acuradas para o desempenho dos algoritmos até a portabilidade do modelo para diferentes máquinas e a facilidade de adaptação por parte dos usuários, como vimos no *capítulo 2*. A necessidade da validação experimental é a característica mais citada.

Dos novos modelos, merecem destaque o *BSP* de L. G. Valiant e o *CGM* de F. Dehne (estudados no *capítulo 2*). O *BSP* é um modelo para máquinas paralelas com memória distribuída mais detalhado, podendo ser utilizado também para simular o *PRAM*. O *CGM* pode ser visto como um caso particular do *BSP* e é voltado para a construção de algoritmos. A memória é sempre distribuída no *CGM* e pode armazenar uma quantidade *grande* de dados. Note que esta é a realidade para a maioria das máquinas existentes. Em ambos os modelos a complexidade dos algoritmos é dada pelo número de rodadas necessárias para que o problema seja resolvido,

devendo o custo de cada rodada ser mensurado.

Quando proposto a partir do modelo *CGM*, como fizeram F. Dehne e S. W. Song em [12], o *LR* mostra-se um problema bastante interessante. Dedicamo-nos ao seu estudo e obtivemos resultados bastante interessantes envolvendo o projeto de algoritmos paralelos teoricamente eficientes no modelo *CGM*. A formalização do *LR* no modelo *CGM* encontra-se no final do *capítulo 2*.

Em [12] foram propostos dois algoritmos probabilísticos para o problema. O primeiro resolve o *LR* em $O(\log p + \log\log n)$ rodadas e o segundo em $O(k \log p)$, onde $k < \ln^* n$. Em ambos a complexidade é de $O(n/p)$ operações para a computação local (por rodada) e os resultados valem com alta probabilidade. O primeiro algoritmo de [12] é apresentado no *capítulo 3*.

No mesmo capítulo, apresentamos outro algoritmo probabilístico que resolve o *LR* no modelo *CGM* em um número esperado de $O(n/p)$ operações de computação local por rodada e $O(\log p)$ rodadas de comunicação.

Este algoritmo é original e consiste na principal contribuição deste trabalho. Até a época de sua construção[1], nenhum outro algoritmo havia sido proposto que resolvesse o *LR* em um número de rodadas independente do tamanho da lista. Posteriormente, foi apresentado um algoritmo determinístico em [4] com a mesma complexidade. A chave deste algoritmo consiste no sorteio determinístico dos pivôs, feito de acordo com [5]. Também construímos um algoritmo determinístico que resolve o *LR* em $O(\log p)$ rodadas e $O(n/p)$ operações de computação local por rodada que é diferente dos demais porque não utiliza pivôs na solução do problema. Nenhum deles foi apresentado porque algoritmos determinísticos não estão no contexto deste trabalho.

Resolvido satisfatoriamente o problema teórico, implementamos os algoritmos do *capítulo 2* para observar o comportamento da sua execução. Os resultados foram obtidos utilizando a máquina paralela *Parsytec PowerXplorer 16/32 Parallel Computing System*. A máquina em questão apresenta 16 nós sob uma topologia 2-D (*grade*). Cada nó consiste de um processador de aplicação *PowerPC601* (80 *MHz*) e um processador de comunicação *T805*, com memória local de 32 *MBytes*. A taxa de transferência de dados entre os nós é da ordem de milisegundos. Esta taxa é considerada muito baixa, sobretudo para aplicações em problemas semelhantes ao *LR*, onde a necessidade de comunicação freqüente é ditada pelo alto grau de dependência entre os dados.

A nossa implementação é bastante simples, buscando apenas obedecer as especificações do modelo *CGM* e determinar o número de rodadas que cada algoritmo precisa para resolver o *LR*. As otimizações para realizar a comunicação entre os nós, usuais em implementações paralelas, não estão sendo efetuadas. Somando isto à baixa taxa de comunicação entre os nós da *Parsytec*, não dispomos de resultados animadores para o tempo de execução dos algoritmos, como podemos

[1] Este algoritmo foi concluído em julho de 1996.

observar no *capítulo 3*.

Note que o modelo *CGM* não considera para a análise dos algoritmos variáveis como a *latência do sistema* e o fator g, definidos no *BSP*. Os resultados mostram que estes fatores nem sempre podem ser desprezados. Em uma primeira avaliação, poderíamos utilizar este fato para concluir que o modelo *CGM* não é válido como ferramenta de predicção para o tempo de execução de certos tipos de algoritmos. No entanto, considerando que se trata de um modelo para algoritmos (e não para máquinas) e que a maioria dos outros fatores envolvidos na determinação do tempo de execução são fixos para cada máquina paralela, o número de rodadas pode até ser considerado um bom critério para a escolha de soluções alternativas. Favorável a esta afirmação, temos o fato de o tempo de execução do *algoritmo 2* diminuir em relação ao do *algoritmo 1*, conforme aumenta o tamanho da entrada.

De qualquer forma, embora as nossas implementações sejam simples, todas as restrições teóricas dos algoritmos foram observadas. Os dados obtidos para o número de rodadas são perfeitamente confiáveis e fornecem uma visão adequada do desempenho dos algoritmos pelo critério do modelo *CGM*. Os dados mostram o desempenho satisfatório dos algoritmos, que resolvem o *LR* no modelo *CGM* no número de rodadas esperado.

Julgamos que a construção do algoritmo probabilístico (*algoritmo 2*) é, sem dúvida, a principal contribuição desta dissertação, seguida pelo interessante estudo sobre técnicas de projeto de algoritmos e modelos de computação paralela para máquinas com memória distribuída para aplicações práticas. No entanto, formulamos diversas questões que gostaríamos de ver resolvidas em outros trabalhos.

As aplicações do *LR* no modelo *CGM* e a resolução de outros problemas envolvendo grafos e árvores nesse modelo são um assunto bastante interessante. Alguns desses problemas com as respectivas soluções estão apresentados em [4], mas obviamente existem muitos outros em aberto.

Outra questão diz respeito à avaliação dos algoritmos pelo número de rodadas. Não estamos completamente convencidos de que este é o método mais adequado para problemas em geral. Gostaríamos de ver comparações de *speedup* entre algoritmos que exigem uma grande quantidade de comunicação quando implementados no modelo *CGM* e quando implementados em outros modelos distribuídos utilizando outros protocolos de comunicação. Por exemplo, suponha um modelo em que as mensagens são *empacotadas* somente até que este *pacote* tenha o mesmo tamanho da *largura da banda de comunicação*[2], quando são imediatamente transmitidos. Uma outra comparação interessante poderia ser feita utilizando um modelo *assíncrono*. O *LR* é um bom problema para se fazer esse estudo.

Por último, embora não seja esta a questão menos importante, gostaríamos de ver resultados a respeito da definição de critérios para decidir quais os tipos de problemas e de máquinas

[2]Isto é o proposto por L. G. Valiant na apresentação do modelo *BSP* em [32].

onde a computação paralela efetivamente pode garantir a obtenção de resultados práticos muito superiores aos seqüenciais. Por exemplo, em [9] comprova-se a adequação prática do modelo *CGM* para algoritmos geométricos. Nesse caso, gostaríamos de saber se isto é suficiente para definir uma *classe* de problemas aos quais o modelo *CGM* é adequado e, em caso afirmativo, seria interessante caracterizar essa classe. Sabemos que esta questão é bastante complexa e recai parcialmente na de descobrir um modelo *padrão* para a computação paralela e na da existência de um tal modelo único mas, considerando como objetivo justificar o investimento necessário para difundir o uso de paralelismo em aplicações comerciais em larga escala, acreditamos que tal método pode ser mais eficaz, até por ser menos ambicioso.

APÊNDICE A

O modelo de computação PRAM

O *modelo de memória compartilhada* é um modelo de computação paralela composto por um número de processadores, digamos p, cada um dos quais possuindo sua própria memória e executando seu próprio programa local. A comunicação entre os processadores se dá através da leitura e escrita em uma mesma memória compartilhada [18].

Neste modelo, dois modos de operação são possíveis. Em um deles, chamado *síncrono*, todos os processadores processam sincronamente em um mesmo relógio, ou *clock*. No outro modo, chamado *assíncrono*, cada processador pode operar sob *clocks* separados. O nome padrão para o *modelo de memória compartilhada* operando em modo síncrono é *PRAM*, *Parallel Random Access Machine* ou *Máquina Paralela de Acesso Aleátorio.*

O *PRAM* foi proposto como uma extensão do modelo seqüencial *RAM* e adequa-se ao desenvolvimento de algoritmos paralelos [22]. Devido ao sucesso inicial do modelo dispomos de vasta literatura a respeito. Os algoritmos em *PRAM* são freqüentemente do tipo *SIMD*, ou *Single Instruction Multiple Data*, segundo a caracterização de Flynn [15]. Isto significa que todos os processadores ativos estão executando a mesma instrução em cada *clock*, porém com dados usualmente diferentes. Contudo, embora facilite o projeto de algoritmos, esta não é uma restrição do modelo e é importante que não seja entendida como tal [18].

Dado um algoritmo paralelo em *PRAM*, usualmente medimos o seu desempenho em termos da análise do pior caso. Se n é o tamanho da instância, $T(n)$ é o tempo de execução do algoritmo e $P(n)$ é o número de processadores necesssários à sua execução, o custo $C(n)$ do algoritmo é dado por $C(n) = T(n) * P(n)$. Neste modelo, o algoritmo é considerado *ótimo* quando tiver o seu custo $C(n)$ minimizado [18].

O *trabalho* realizado por um algoritmo em *PRAM* é definido pelo número total de operações realizadas, independente do grau de paralelismo. Através do uso do *paradigma de trabalho-tempo* este método também pode ser utilizado para a avaliação de algoritmos. O *paradigma de trabalho-tempo* [18] sugere que o algoritmo seja descrito através de uma seqüência de unidades de tempo,

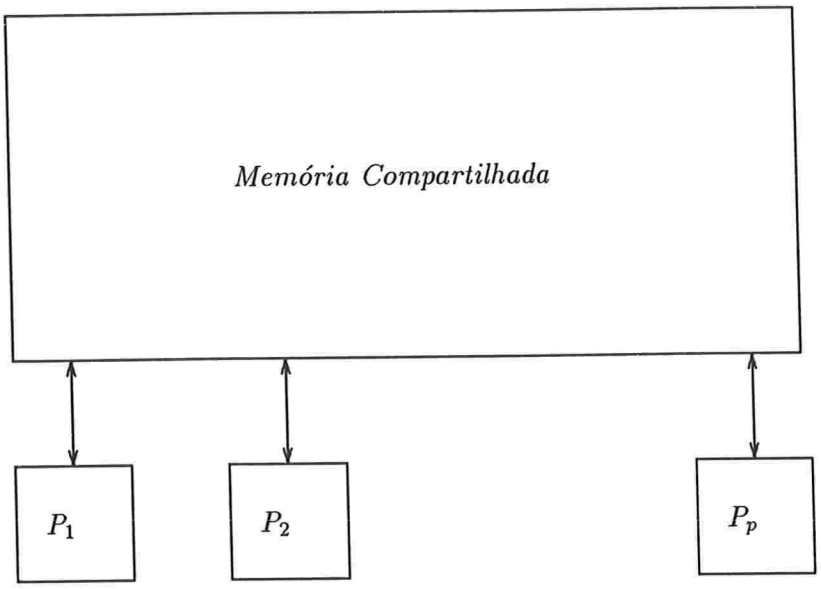

Figura A.1: O modelo de memória compartilhada para p processadores.

cada uma delas incluindo um número qualquer de operações concorrentes,

Em [18], podem ser encontradas todas as definições e métodos de avaliação de algoritmos em *PRAM*. Restringi-mo-nos aos mais utilizados, pois fornecer maiores detalhes está fora do contexto deste trabalho. Precisamos apenas acrescentar que, a exemplo do que acontece no modelo *CGM*, a otimalidade dos algoritmos em *PRAM* também deve ser avaliada em termos de comparações com os algoritmos seqüênciais ótimos que resolvam os mesmos problemas.

Dado que a memória é compartilhada, de acordo com a concorrência de acesso, podemos subdividir o modelo *PRAM* em:

- *EREW PRAM (Exclusive Read Exclusive Write)*: não permite acesso concorrente para leitura ou para escrita de dados;
- *CREW PRAM (Concurrent Read Exclusive Write)*: aceita apenas que vários processadores leiam a mesma posição de memória ao mesmo tempo;
- *CRCW PRAM (Concurrent Read Concurrent Write)*: tanto a leitura quanto a escrita concorrente na memória compartilhada são aceitas;
- *ERCW PRAM (Exclusive Read Concurrent Write)*: aceita a escrita concorrente mas não a leitura. É pouco comum porque o problema de escrita concorrente é muito mais difícil de tratar do que o de leitura. Este modelo não é citado por alguns autores.

O modelo mais restritivo, *EREW PRAM*, consegue simular o menos restritivo, *CRCW PRAM*, com perda de apenas $O(\log n)$ no tempo, onde n é o tamanho da instância [7].

Ao apresentar um algoritmo no modelo, os autores devem especificar também a concorrência de acesso pois esta influi na determinação do grau de paralelismo do algoritmo. Para resolver o *LR*, por exemplo, temos o algoritmo de [6], que o faz com $O(n)$ operações e tempo $O(\log n)$ em *CRCW PRAM*. Já em [23], temos outro algoritmo para o problema, com os mesmos resultados para tempo e número de operações só que para o modelo *EREW PRAM*. Este último é um algoritmo comparativamente mais forte.

Em [22] é feita uma análise comparativa dos modelos de computação paralela em geral e o trabalho destaca como vantagem do *PRAM* a sua *simplicidade*. Por ser uma extensão natural do modelo *RAM*, é menor a dificuldade de adaptação dos usuários ao *PRAM* do que aos outros modelos paralelos, pois ele permite uma fácil estruturação do problema e torna relativamente mais simples a análise dos algoritmos para ele projetados. Outra vantagem do modelo [26, 18] é que o comportamento do *EREW PRAM* reflete um sistema real quando ignoradas as conexões de rede e os caches[1]. Em [26], sugere-se que esta é uma arquitetura *ideal* e que mensurar o *speedup* assintótico para *EREW PRAM* é o mesmo que medir o paralelismo inerente ao algoritmo apresentado.

Por outro lado, a lista de desvantagens para o uso do modelo começa pela discrepância entre o desempenho teórico e o desempenho obtido em uma execução real [18, 22]. Esta discrepância é dada parcialmente pela necessidade de se satisfazer a exigência de sincronismo, devido a fatores como o sistema operacional e as diferentes velocidades de acesso dos meios de armazenamento de dados e de execução das instruções. Um outro problema sério consiste na distância existente entre o modelo e as máquinas reais existentes, já que a maioria dos algoritmos para *PRAM* supõe recursos ilimitados de processador e memória, o que é impraticável.

Finalmente, não é razoável supor que cada acesso a memória compartilhada em uma máquina real ocorra em um único ciclo [8, 22, 32]. O custo de acesso à memória pode ser efetivamente muito alto e comprometer as previsões de desempenho do modelo *PRAM* para implementações reais. A necessidade de tratar acessos concorrentes, a existência de contenção no barramento e a distância física entre o módulo de memória a ser acessado e o processador são alguns dos fatores a serem considerados na justificativa dessa afirmação.

No entanto, esta lista de desvantagens não significa que os algoritmos e todo o trabalho envolvendo o modelo *PRAM* não sejam importantes. Ao contrário, servem como fonte de idéias e soluções para novos problemas. Além disso, existem pesquisadores empenhados em adequar

[1]Em termos práticos, particularmente, consideramos ignorar estas variáveis um absurdo, pois são justamente as responsáveis pelos maiores custos durante a execução dos algoritmos, como veremos. Porém, esta é uma motivação para o estudo do modelo *PRAM* que não pode ser desprezada.

o modelo *PRAM* à realidade. Em [36], por exemplo, são feitas referências a resultados sobre as limitações físicas das máquinas paralelas e são propostos modelos de máquina para a simulação de *PRAM* em que o custo extra, inerente ao modelo, pode ser medido. Medir estes custos inclui o *PRAM* na categoria dos modelos *realistas*, tão defendidos pela comunidade especializada atualmente, contra-argumentando os que se utilizam deste mote para relegá-lo a uma categoria inferior de modelo de computação.

APÊNDICE B

A implementação do algoritmo 1

Dado que os algoritmos no modelo *CGM* são descritos em *alto nível*, as respectivas implementações usualmente não são triviais. Por isso, baseados em [11], julgamos interessante apresentar este apêndice para descrever a implementação do *algoritmo 1*.

Considere o modelo *CGM* com p processadores e seja L uma lista ligada com n nós. Cada nó de L deve conter as seguintes informações:

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem

onde:

Elem representa o conteúdo de cada nó da lista ligada.

IsPivot valor booleano (TRUE/FALSE) que indica se o nó foi ou não escolhido como pivô.

NewNumber número atribuído ao pivô antes de se iniciar a *etapa (2)* para a definiçao de L'.
Lembramos que L' é o conjunto de pivôs escolhidos na *etapa (1)* do algoritmo.

NextPivot armazena o pivô que sucede cada nó na resolução do $LRM(etapa\ (2))$.

DistToPivot armazena a distância do nó ao pivô que o sucede na lista.

NextProc processador onde está o nó sucessor.

NextElem localização do nó sucessor dentro do processador que o contém.

A implementação de uma lista ligada usualmente considera dois campos em cada nó: um para o seu conteúdo, aqui representado por *Elem*, e o apontador para o seu sucessor, que dividimos em dois outros campos, *NextProc* e *NextElem*, para simplificar a rodada de comunicação e garantir o acesso direto aos dados em cada processador e viabilizando o custo de $O(n/p)$ operações de computação local.

IsPivot é preenchido na *etapa (1)*, quando se realiza o sorteio independente para a escolha de pivôs. Feito este sorteio, precisamos atribuir nomes diferentes[1] aos pivôs através de *NewNumber*, a fim de definir L' e permitir que a *etapa (4)* custe $O(n/p)$ operações locais. Note que não é necessário armazenar *IsPivot* e *NewNumber* simultaneamente, já que *NewNumber* só se aplica aos nós em que *IsPivot* é *TRUE*. Apenas por clareza vamos manter os dois, já que diferem pela natureza da informação armazenada.

Assim que passar a apontar para um pivô, o nó deve armazenar sua identificação em *NextPivot*, para facilitar as atualizações finais em *DistToPivot* na *etapa (4)*. A necessidade deste campo e de *NewNumber* ficarão mais evidentes adiante.

Para melhor ilustrar os procedimentos de implementação, acompanharemos a execução de cada etapa com um exemplo.

Suponha $n = 12$, $p = 4$ e a lista linear da figura 3.1 distribuída arbitrariamente entre os processadores, onde os pivôs já foram sorteados. Para o sorteio basta que cada processador percorra seus nós, escolhendo cada um deles como pivô com probabilidade $1/p$, independentemente. A situação pode ser vista a seguir.

Processador 1

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
1	1				2	3
5	0				3	2
10	0				4	2

Processador 2

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
8	0				2	2
9	1				1	3
2	0				4	3

Processador 3

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
4	1				1	2
6	0				4	1
12	1				3	3

[1] Precisamos apenas distinguir os pivôs uns dos outros para compor L', não há necessidade de ordem entre eles, como veremos no exemplo.

Processador 4

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
7	1				2	1
11	0				3	3
3	0				3	1

Antes de se iniciar a *etapa (2)*, precisamos preencher *NewNumber*. Usando *all-to-all broadcast*, todos os processadores informam aos demais o número de pivôs sorteados. Cada processador monta uma tabela com estas quantidades e descobre em qual número, digamos *InitNumber*, deve iniciar a numeração de seus próprios pivôs. Observe a figura a seguir. Esta operação custa uma rodada de comunicação para o *all-to-all broadcast* e mais uma rodada de computação local de $O(n/p)$ operações para a numeração dos pivôs a partir de *InitNumber*.

Processador	Qtde. de Pivôs	InitNumber
1	1	1
2	1	2
3	2	3
4	1	5

(Uma variante seria fazer estas operações em apenas um dos processadores, enviando todos os dados apenas para ele, mas isto custaria mais uma rodada de comunicação para informar $InitNumber$ a cada um dos processadores, enquanto os demais ficariam aguardando e ociosos.)

A figura ilustra a situação após a escolha e renumeração dos pivôs, completando a *etapa (1)* do algoritmo. Observe.

Processador 1

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
1	1	1			2	3
5	0				3	2
10	0				4	2

Processador 2

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
8	0				2	2
9	1	2			1	3
2	0				4	3

Processador 3

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
4	1	3			1	2
6	0				4	1
12	1	4			3	3

Processador 4

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
7	1	5			2	1
11	0				3	3
3	0				3	1

Para cada iteração do *LRM(etapa (2))*, os processadores devem obter o sucessor de cada um dos nós nele contidos e substituir este sucessor pelo sucessor do sucessor (obviamente atualizando as distâncias), de acordo com a técnica de *pointer jumping*, vista no capítulo 2.

Neste ponto, precisamos obter os sucessores dos nós.

Lembramos que o modelo *CGM* exige que todos os dados enviados de um dado processador para outro em uma rodada devem estar *empacotados* em uma única mensagem. Então, fazemos o seguinte: cada processador P_j percorre seus nós e monta uma mensagem de comunicação apenas com as requisições que serão enviadas para cada processador P_i. P_i deve responder apenas com os dados sobre os sucessores de nós de P_j que estão em P_i. Então P_i preenche a mensagem de resposta e devolve para P_j, que pode continuar com o seu processamento local.

Note que só enviamos e recebemos de cada processador as informações relevantes a ele. Os dados das mensagens trocadas devem ser os seguintes:

IsPivot	NewNumber	DistToPivot	NextProc	NextElem

Se as informações sobre os sucessores estiverem em P_i, evidentemente não há comunicação.

Esta etapa deve ser repetida até que cada nó chegue ao pivô que o suceda na lista, *NextPivot*, obtendo a respectiva distância até ele, *DistToPivot*. A sua duração está relacionada à máxima distância entre dois pivôs consecutivos, limitada a $3p\ln(n)$, com alta probabilidade, conforme vimos na análise do algoritmo. Após o término desta etapa, cada processador terá obtido os valores *NextPivot* e *DistToPivot* para todos os seus nós que pertençam também a L'.

A figura seguinte mostra o que se tem após a *etapa (2)*.

Processador 1

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
1	1	1	3	3	3	1
5	0		5	2	4	1
10	0		4	2	3	3

Processador 2

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
8	0		2	1	2	2
9	1	2	4	3	3	3
2	0		3	2	3	1

Processador 3

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
4	1	3	5	3	4	1
6	0		5	1	4	1
12	1	4	4	0	3	3

Processador 4

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
7	1	5	2	2	2	2
11	0		4	1	3	3
3	0		3	1	3	1

Na *etapa (3)* propagam-se os dados referentes aos elementos de L' a todos os processadores. Cada processador constrói uma tabela como a abaixo:

NewNumber	NextPivot	DistToPivot
1	3	3
2	4	3
3	5	3
4	4	0
5	2	2

Na *etapa (4)* cada processador faz LR apenas para os nós de L', utilizando um algoritmo seqüencial trivial e armazenando os resultados em *DistToEnd*, conforme a tabela abaixo. O campo *DistToEnd* armazena a distância calculada de cada pivô até o último nó da lista.

NewNumber	NextPivot	DistToPivot	DistToEnd
1	3	3	11
2	4	3	3
3	5	3	8
4	4	0	0
5	2	2	5

Aqui, torna-se clara a necessidade da renumeração dos pivôs. Sem ela, seria necessário ordenar os nós de L' com o custo de $\Omega((n/p)\log(n/p))$ operações locais. Da mesma forma que na renumeração dos pivôs, estas operações poderiam ser feitas em apenas um dos processadores e propagadas aos demais com o custo extra de uma rodada de comunicação.

Para resolver o LR cada processador percorre sua parcela dos nós da lista, somando à distância *DistToPivot* de cada nó o valor *DistToEnd* calculado, referente ao pivô *NextPivot*.

Observe o resultado final do algoritmo:

Processador 1

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
1	1	1	4	11	3	3
5	0		4	7	3	3
10	0		4	2	3	3

Processador 2

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
8	0		4	4	3	3
9	1	2	4	3	3	3
2	0		4	10	3	3

Processador 3

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
4	1	3	4	8	3	3
6	0		4	6	3	3
12	1	4	4	0	3	3

Processador 4

Elem	IsPivot	NewNumber	NextPivot	DistToPivot	NextProc	NextElem
7	1	5	4	5	3	3
11	0		4	1	3	3
3	0		4	9	3	3

Note que esta implementação acrescenta duas rodadas à execução do algoritmo. Uma para numerar os pivôs logo após a *etapa (1)* e outra para descobrir o término do *LRM*.

Referências Bibliográficas

[1] M. Atallah and S. Hambrusch, *Solving tree problems on a mesh-connected processor array*, Information and Control, 69 (1986), pp. 168–187.

[2] S. Baase, *Introduction to parallel connectivity, list ranking and euler tour techniques*, in Synthesis of Parallel Algorithms, J. H. Reif, ed., Morgan Kaufmann Publisher, 1993, pp. 61–114.

[3] G. Blelloch, C. Leiserson, B. Maggs, and C. Plaxton, *A comparison of sorting algorithms for the connection machine cm-2*, in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1991, pp. 3–16.

[4] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. R. A. Roncato, N. Santoro, and S. Song, *Efficient parallel graph algorithms for coarse grained multicomputers and bsp*, in ICALP'97 - 24th International Colloquium on Automata, Languages, and Programming, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, eds., Lecture Notes in Computer Science, Springer-Verlag, July 1997, pp. 390–400.

[5] R. Cole and U. Vishkin, *Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms*, ACM, (1986), pp. 206–219.

[6] ——, *Faster optimal parallel prefix sum and list ranking*, Information and Control, (1989), pp. 334–352.

[7] T. Cormem, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[8] F. Dehne, X. Deng, and A. F. and, *A randomized parallel 3d convex hull algorithm for coarse grained multicomputers*, tech. rep., Carleton University, 1986.

[9] F. Dehne, A. Fabri, and A. Rau-Chaplin, *Scalable parallel computational geometry for coarse grained multicomputers*, in Proc. ACM Symposium on Computational Geometry, 1993, pp. 298–307.

[10] F. Dehne, C. Kenyon, and A. Fabri, *Scalable and architecture independent parallel geometric algorithms with high probability optimal time*, in Proc. IEEE Symposium on Parallel and Distributed Processing, 1994, pp. 586–593.

[11] F. Dehne, F. Santana, and S. Song, *Validação da escalabilidade de um algoritmo paralelo para list ranking*, in Proceedings of XIX CNMAC, S. B. de Matemática Aplicada e Computacional, ed., 1996, pp. 132–133. Em português.

[12] F. Dehne and S. Song, *Randomized parallel list ranking for distributed memory multiprocessors*, tech. rep., Carleton University, 1996.

[13] X. Deng, *Good programming style on multiprocessors*, in Proc. IEEE Symposium on Parallel and Distributed Processing, 1994.

[14] X. Deng and P. Dymond, *Efficient routing and message bounds for optimal parallel algorithms*, in Proc. Int. Parallel Proc. Symposium, 1995.

[15] M. Flynn, *Some computers organizations and their effectiveness*, IEEE Transactions on Computers, (1972).

[16] A. Gerbessiotis and L. Valiant, *Direct bulk-syncronous parallel algorithms*, in Third Scandinavian Workshop on Algorithm Theory, O. Nurmi and E. Ukkonen, eds., Algorithm Theory, July 1992, pp. 1–18.

[17] M. Ghouse and M. Goodrich, *Experimental evidences for the power of random sampling in practical parallel algorithms*, IEEE International Parallel Processing Symposium, (1993).

[18] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, 1992.

[19] D. Knuth, *The Art of Computer Programming*, vol. 1, Addison-Wesley Publishing Company, 1973.

[20] ——, *Big omicron and big omega and big theta*, SIGACT News, (1976), pp. 18–24.

[21] H. Li and K. Sevcik, *Parallel sorting by overpatitioning*, in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 46–56.

[22] R. Menezes, *Um estudo sobre modelos de computação paralela*, Master's thesis, Universidade Estadual de Campinas, June 1995. Em português.

[23] G. Miller, F. Modugno, and M. Reid-Miller, *List ranking and parallel tree contraction*, in Synthesis of Parallel Algorithms, J. H. Reif, ed., Morgan Kaufmann Publisher, 1993, pp. 115–194.

[24] B. Monien, R. Diekmann, R. Feldmann, R. Klasing, R. Luling, K. Menzel, T. Romke, and U. Schroeder, *Efficient use of parallel and distributed systems: From theory to practice*, in Computer Science Today, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer, 1995, pp. 62–77.

[25] K. Mulmuley, *Computational geometry: An introduction through randomized algorithms*, Prentice Hall, (1993).

[26] D. Nussbaum and A. Agarwal, *of parallel machines*, Communications os The ACM, 34 (1991), pp. 57–61.

[27] F. Preparata and M. Shamos, *Computacional geometry - an introduction*, in Texts and Monographs in Computer Science, Springer-Verlay, 1985.

[28] F. Santana, E. Saukas, and S. Song, *Design of communication-efficient parallel algorithms for distributed memory parallel computers*, tech. rep., Universidade de São Paulo, 1997.

[29] L. Snyder, *Experimental validation of models of parallel computation*, in Computer Science Today, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer, 1995, pp. 78–100.

[30] T. Standish, *Data Structure Techniques*, Addison-Wesley Publishing Company, 1980.

[31] A. Tanenbaum, *Modern Operating Systems*, Prentice-Hall Inc., 1992.

[32] L. Valiant, *A bridging model for parallel computation*, Communications os The ACM, 33 (1990), pp. 103–111.

[33] ——, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., MIT Press/Elsevier, 1990, pp. 934–972.

[34] ——, *A combining mechanism for parallel computers*, in Parallel Architectures and Their Efficient Use, F. M. auf der Heide, B. Monien, and A. Rosenberg, eds., Fisrt Heinz Nixdorf Symposium, Springer-Verlag, November 1992.

[35] ——, *Why bsp computers?*, in Seventh International Parallel Processing Symposium, IEEE Computer Society, April 1993, pp. 1–5.

[36] J. Wiedermann, *Quo vadetis, parallel machine models*, in Computer Science Today, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer, 1995, pp. 101–114.