List Ranking: Um Estudo Experimental

Guilherme Pereira Vanni

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

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

— São Paulo, outubro de 2004 —

***List Ranking*: Um Estudo Experimental**

Este exemplar corresponde à redação
final da dissertação devidamente
corrigida e defendida por
Guilherme Pereira Vanni
e aprovada pela comissão julgadora.

São Paulo, outubro de 2004.

Banca examinadora:

- Prof. Dr. Siang Wun Song (orientador) (IME-USP)
- Prof. Dr. Marco Dimas Gubitoso (IME-USP)
- Prof. Dr. Edson Norberto Cáceres (UFMS)

Aos meus pais Carlos (*in memoriam*) e Marise

Agradecimentos

Gostaria de agradecer aos meus pais por proporcionarem a base fundamental para eu chegar até aqui. Mãe, também por seu carinho e por suas orações. Aos meus irmãos Fernando, André e Míriam, que sempre depositaram em mim uma grande confiança. Ao meu saudoso pai, cuja memória, sempre presente, ajudou-me nos momentos mais difíceis, pois seu exemplo de vida, por si só, foi o meu grande incentivo.

Agradeço também a minha esposa Isabela, sempre amorosa e companheira, desde o dia da seleção para o ingresso no mestrado até a redação final deste trabalho. Sua paciência e dedicação tornou o meu trabalho menos árduo e mais prazeroso. Ao meu filho Guilherme, que nasceu nesta fase final, me trazendo ainda mais alegria, servindo para me fazer ter, a cada sorriso seu, mais vontade de continuar lutando.

Ao Siang Wun Song, primeiramente por ser meu orientador. Também pela sua sabedoria e competência. Sua dedicação e generosidade foram grandes incentivos para a concretização do meu trabalho. Também agradeço ao Professor Edson Cáceres por suas valiosas contribuições.

À Fundação Vunesp por todo seu apoio, me permitindo conciliar o mestrado com o emprego.

Aos colegas que fiz durante o curso. Ao Daniel, pelo seu companheirismo, amizade e sua contagiante perseverança. Ao Marcelo, pelos seus ensinamentos e por nossas constantes trocas de informações. Ao Ulisses, por suas dicas para a conclusão da fase final e aos demais, Mateus, Mimiça, Danielle, Ana Lúcia, entre outros que tiveram muita importância para a realização das disciplinas.

Aos funcionários da secretaria de pós-graduação, especialmente ao Pinho, pela competência e atenção com que trata os alunos.

Enfim, a Deus, que me deu saúde e discernimento para eu conseguir estar onde hoje eu estou.

Guilherme Pereira Vanni

Resumo

A solução de muitos algoritmos paralelos para grafos recai no problema do *List Ranking*. Este problema foi muito estudado para o modelo PRAM. Entretanto, o modelo PRAM não traz resultados satisfatórios na prática e para o modelo BSP/CGM são encontradas poucas implementações do *List Ranking*. Neste trabalho, analisamos os resultados experimentais encontrados no modelo BSP/CGM. Também foram implementados dois algoritmos usando o modelo BSP/CGM, onde um deles é probabilístico e requer, com alta probabilidade, $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ rodadas de comunicação. O outro é determinístico e requer apenas $O(\log p)$ rodadas de comunicação. Além disso, modificamos o algoritmo determinístico visando a obtenção de uma implementação mais eficiente. Nossa implementação alcançou um desempenho satisfatório, e as acelerações alcançadas foram melhores no algoritmo probabilístico.

Abstract

The solution of many parallel graph algorithms depend on List Ranking. This problem has been extensively studied under the PRAM model. The PRAM model, however, does not give satisfactory results in practice and for the BSP/CGM model there are very few implementations of List Ranking. In this work, we analyze the experimental results for the BSP/CGM. We also implemented two algorithms using the BSP/CGM model, one is probabilistic and requires $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ communication rounds with high probability. The other is a deterministic algorithm and requires only $O(\log p)$ communication rounds. Furthermore, we modified the deterministic algorithm to obtain a more efficient implementation. Our implementation obtained a satisfactory performance, and the probabilistic algorithm gave better speedups.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Uma lista linear ligada é uma seqüência de nós tal que cada nó aponta para outro nó, chamado seu sucessor, e não há ciclo em tal lista. Convenciona-se que o último nó aponta pra nil ou para si.

O problema de *List Ranking* consiste em determinar o *rank* para todos os nós, isto é, a distância para o último nó da lista. Sua importância se deve ao fato de que muitos algoritmos paralelos em teoria dos grafos o usam como uma subrotina, como podemos ver na Figura 1.1, que sintetiza uma família de algoritmos relacionados a partir de técnicas de algoritmos fundamentais [1].

Figura 1.1: Síntese de algoritmos paralelos para grafos

List Ranking foi um dos problemas bastante estudados no modelo PRAM (*Parallel Random Access Machine*), entretanto as acelerações teóricas não são as mesmas obtidas quando implementados em máquinas reais [2]. O modelo BSP (*Bulk-Synchronous Parallel*) foi sugerido por Valiant em [11] para ser um modelo adequado de paralelismo suficientemente próximo às máquinas paralelas existentes com memória distribuída.

Baseado neste modelo, foi proposto por Dehne *et al.* [12] o modelo CGM (*Coarse Grained Multicomputer*) que apresenta muita similaridade com o *BSP*, porém é mais simples, facilitando o projeto e análise de algoritmos. Ele compreende um conjunto de p processadores p_1,..., p_p com $O(n/p)$ memória local por processador e uma rede de comunicação arbitrária. Todos os algoritmos consistem de alternância entre rodadas de computação local e comunicação global. Cada rodada de comunicação consiste de roteamento de uma simples h-relação com $h =$

$O(n/p)$, isto é, cada processador envia e recebe no máximo $O(n/p)$ dados [2].

Encontrar um algoritmo eficiente no modelo CGM é equivalente à minimizar o número de rodadas de comunicação bem como o tempo de computação local total.

O modelo mencionado acima tem sido usado (implicita ou explicitamente) no desenvolvimento de algoritmos paralelos para vários problemas e tem mostrado resultados práticos muito bons [2].

Ao iniciar este trabalho os resultados das implementações para o problema do *List Ranking* não eram satisfatórios, como se pode observar em [5]. Durante a elaboração deste trabalho Chan e Dehne em [19] apresentaram a mais eficiente até então. Nesta dissertação foram estudados os algoritmos existentes no modelo CGM para este problema, implementamos dois deles e analisamos os resultados, comparando-os com os obtidos até então. Também fizemos modificações em um dos algoritmos para torná-lo mais eficiente.

Para isso foi utilizada a biblioteca MPI (*Message Passing Interface*) e o computador paralelo do tipo Beowolf presente no Departamento de Computação do IME/USP.

Além desta introdução, este trabalho está dividido em mais 5 capítulos. No capítulo 2, definimos e detalhamos os modelos BSP e CGM e a relação entre eles. No capítulo 3 apresentamos os algoritmos CGM para solução do PLR (*Problema de List Ranking*), comparando suas complexidades. Neste capítulo também detalhamos os algoritmos que implementamos. No capítulo 4 mostramos como foi feita a implementação dos algoritmos do capítulo anterior resaltando os aspectos mais importantes. No capítulo 5 mostramos os resultados experimentais obtidos com a implementação de algoritmos seguindo o modelo CGM encontrados na literatura. Mostramos também os resultados alcançados pela nossa implementação e a comparação com as anteriores. No capítulo 6, apresentamos a conclusão deste trabalho.

Capítulo 2

Modelos Realísticos BSP/CGM

O processamento paralelo sofre a falta de um modelo que seja amplamente aceito. Ao final dos anos 80, vários problemas haviam sido estudados e vários limites inferiores e superiores foram demonstrados para esses problemas em diferentes modelos de computação, tais como memória compartilhada, hipercubos e grades. Quando esses resultados teóricos eram implementados nas máquinas existentes, as acelerações obtidas eram, muitas vezes, desapontadoras. Esses resultados levaram à criação de modelos de computação paralela com granularidade grossa. Nesses modelos, o conceito de computação paralela é representado com uma série de *superpassos*, ao invés de passos com o envio individual de mensagens ou acessos individuais à memória compartilhada. O BSP e o CGM são exemplos desses modelos. Os algoritmos projetados para esses modelos, quando implementados nas máquinas existentes, têm obtido acelerações próximas das previstas em resultados teóricos. Isso faz com que sejam chamados de modelos realísticos [13].

2.1 O Modelo BSP (*Bulk-Synchronous Parallel*)

O modelo BSP foi proposto por Valiant [11] em 1990 e é um dos modelos realísticos mais importantes. Além disso, foi um dos primeiros a considerar os custos de comunicação e a abstrair as características de uma máquina paralela em um pequeno número de parâmetros. O modelo seqüencial de Von Neumann serve de ponte entre as necessidades de hardware e software da computação seqüencial, sendo essa a razão do seu sucesso. O principal objetivo do modelo BSP é ser essa ponte para a computação paralela e, conseqüentemente, servir como um padrão.

O modelo BSP consiste na combinação de três atributos, quais sejam, p pro-

cessadores com memória local, um roteador que entrega mensagens ponto-a-ponto entre pares de processadores e facilidade de sincronização de todos ou apenas um subconjunto de processadores.

Um algoritmo BSP consiste em uma seqüência de *superpassos* separados por *barreiras de sincronização*, como mostra a Figura 2.1.

Figura 2.1: Algoritmo BSP

Em um superpasso, a cada processador é atribuído um conjunto de operações independentes, consistindo de uma combinação de passos de computação, usando dados disponibilizados localmente no início do superpasso, e passos de comunicação, através de instruções de envio e recebimento de mensagens. Neste modelo, uma h-relação em um superpasso corresponde ao envio e/ou recebimento de, no máximo, h mensagens em cada processador. Os valores obtidos em resposta a uma mensagem enviada em um superpasso somente poderão ser usados no próximo superpasso.

O modelo possui os seguintes parâmetros:

- n: tamanho do problema;
- p: número de processadores disponíveis, cada um com uma memória local de tamanho $O(n/p)$;
- L: tempo mínimo de um superpasso;
- g: descreve a taxa de eficiência de computação e comunicação, que corresponde à razão entre total de operações de computação local de todos os processadores em uma unidade de tempo, e o número total de mensagens enviadas/recebidas em uma unidade de tempo.

Os parâmetros L e g são utilizados para calcular o custo de comunicação de um algoritmo BSP. O parâmetro L é o custo de sincronização, de forma que cada operação de sincronização contribui com L unidades de tempo para o tempo total de execução. O parâmetro g está relacionado com capacidade de comunicação de uma rede de computadores. Se o número máximo de mensagens enviadas por algum processador durante uma troca simples é h, então seriam necessárias até gh unidades de tempo para a conclusão da troca [6]. Cada superpasso de um algoritmo BSP possui tempo total de execução $w_i + gh_i + L$, onde $w_i = max\{L; t_1; ...; t_p\}$, sendo que t_j é o tempo das operações de computação executadas pelo processador j

no superpasso i e $h_i = max\{L; c_1; ...; c_p\}$ onde c_j é o tempo das mensagens recebidas e/ou enviadas pelo processador j no superpasso i. Seja T o número de superpassos, teremos:

- Custo total de computações locais: $W = \sum_{i=0}^{T} w_i$
- Custo total de comunicação: $H = \sum_{i=0}^{T} h_i$

De tudo isso, o custo total de um algoritmo BSP é dado por $W+gH+LT$.

2.2 O Modelo CGM(*Coarse Grained Multicomputer*)

O modelo CGM foi proposto por Dehne [12] e consiste em um conjunto de p processadores, cada um com memória local de tamanho $O(n/p)$, onde n é o tamanho do problema. Um algoritmo CGM consiste em uma seqüência alternada de rodadas de computação e rodadas de comunicação, separadas por uma barreira de sincronização, como mostra a Figura 2.2. Normalmente, durante uma rodada de computação é utilizado o melhor algoritmo seqüencial para o processamento dos dados disponibilizados localmente. Um algoritmo CGM é um caso especial de algoritmo BSP, onde uma rodada de computação é equivalente a um superpasso de computação no BSP, e o custo total de computação é definido analogamente. Uma rodada de comunicação consiste em uma única h-relação, com $h \leq n/p$. Cada processador envia $O(n/p)$ dados e recebe $O(n/p)$ dados. Toda informação enviada de um processador para outro em uma rodada de comunicação é empacotada em uma grande mensagem, visando com isso reduzir a sobrecarga (*overhead*) para envio de mensagem. O custo das rodadas de comunicação é $H_{n,p} = O(n/p)$. Portanto, o custo total de comunicação de um algoritmo CGM com x rodadas de comunicação é $x \times H_{n,p}$.

Figura 2.2: Algoritmo CGM

A principal diferença entre os modelos BSP e CGM é que o CGM permite apenas um único tipo de operação de comunicação, a h-relação, e apenas conta o número de h-relações como sua principal medida de custo de comunicação. Uma rodada CGM de computação/comunicação corresponde a um superpasso do BSP com custo de comunicação $g(n/p)$ (mais o empacotamento) [2].

O custo de um algoritmo CGM é a soma dos tempos obtidos em termos do número total de rodadas de computação local e o número total de rodadas de comunicação.

Os algoritmos CGM, quando implementados em multiprocessadores atualmente disponíveis, comportam-se bem e exibem acelerações similares àquelas previstas em suas análises. Para esses algoritmos, o objetivo é minimizar o número de superpassos e a quantidade de computação local.

Capítulo 3

Algoritmos paralelos para *List Ranking* no modelo CGM

O primeiro algoritmo CGM para PLR foi proposto por Dehne e Song em [4]. Este é um algoritmo probabilístico que requer, com alta probabilidade, no máximo $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ [1] rodadas de comunicação (h-relações com $h = \tilde{O}(n/p)$) e $\tilde{O}(n/p)$ computação local, onde p é o numero de processadores e n é o tamanho da lista ligada.

Simultaneamente, foi proposta por Dehne e Song em [4] uma versão melhorada que requer, com alta probabilidade, somente $r \leq (4K + 6)\log(\frac{2}{3}p) + 8 = \tilde{O}(K \log p)$ rodadas de comunicação, onde $K = min\{i \geq 0 \mid \ln^{(i+1)} n \leq (\frac{2}{3}p)^{2i+1}\}$.

Observamos que $K < \ln^*(n)$ é um número extremamente pequeno. Para $n \leq 10^{10^{100}}$ e $p \geq 4$, o valor de K é no máximo 2. Isto é, para um dado número de processadores, p, o número requerido de rodadas de comunicação é, para todos os propósitos práticos, independente de n.

Depois disso, um novo algoritmo foi proposto por Dehne *et al.* em [2], [3]. Entretanto, este é um algoritmo determinístico. Ele requer $O(\log p)$ rodadas de comunicação e $O(n/p)$ computação local, sendo este o algoritmo mais eficiente encontrado na literatura.

Finalmente foi proposto por Sibeyn em [8] outro algoritmo determinístico que requer $O(p)$ rodadas de comunicação.

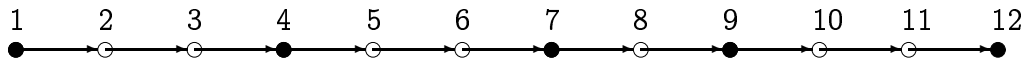
[1] $\tilde{O}(n)$ denota $O(n)$ "com alta probabilidade". Mais precisamente, $X = \tilde{O}(f(n))$, se e somente se $(\forall c > c_0 > 1)$ Prob $\{X \geq cf(n)\} \leq \frac{1}{n^{g(c)}}$ onde c_0 é uma constante fixa e $g(c)$ é um polimônio em c com $g(c) \to \infty$ para $c \to \infty$ [4].

Neste trabalho implementamos o algoritmo proposto por Dehne e Song [4], por ser probabilístico e por sua simplicidade. Também foi implementado o algoritmo determinístico de Dehne *et al.* [2], [3] o mais eficiente em teoria. A seguir, apresentamos os dois algoritmos.

3.1 Algoritmo Probabilístico

Considere uma lista ligada de n elementos, distribuídos igualmente em p processadores, de maneira arbitrária. Cada processador tem portanto n/p elementos.

Além disso, cada processador armazena uma parcela de uma amostra de n/p elementos, extraídos dos elementos originais de modo aleatório. Os elementos amostrados serão denominados *pivôs*. A Figura 3.1 mostra uma lista com alguns elementos escolhidos como pivôs.



- Elemento amostrado chamado pivô

Figura 3.1: Uma lista linear com pivôs escolhidos

Em Dehne [4] demonstra-se que a máxima distância m entre dois pivôs consecutivos não é maior que $3p\ln(n)$, com alta probabilidade. Em outras palavras, Probabilidade $\{m \geq c3p\ln(n)\} \leq \frac{1}{n^c}, c > 2$

Vamos inicialmente estabelecer algumas nomenclaturas. Considere uma lista linear S de n elementos. Seja um conjunto aleatório $S' \subset S$ de pivôs. Para cada $x \in S$ seja $nextPivot(x, S')$ o pivô mais próximo à direita de x na lista S. Sem perda de generalidade, vamos supor que o último elemento de S, λ, é sempre escolhido como pivô e seja $nextPivot(\lambda, S') = \lambda$. Note que para $x \neq \lambda$, $nextPivot(x, S') \neq x$.

Seja $distToPivot(x, S')$ a distância entre x e $nextPivot(x, S')$ na lista S. O problema de *"List Ranking Modificado"* para S com respeito a S' é o seguinte: Determinar para cada $x \in S$ seu próximo pivô, $nextPivot(x, S')$, bem como a distância $distToPivot(x, S')$.

Podemos agora descrever o algoritmo:

1. Selecione um conjunto aleatório de um total de n/p pivôs: Cada processador p_i considera cada um de seus n/p elementos, escolhendo-o como pivô com probabilidade $1/p$.

2. Todos os processadores resolvem coletivamente o subproblema de *List Ranking Modificado* que determina, para cada elemento x da lista, $nextPivot(x, S')$ e $distToPivot(x, S')$.

3. Usando troca completa, os valores $nextPivot$ e $distToPivot$ para todos os pivôs $x \in S'$ são enviados a todos os processadores.

4. Usando os dados recebidos no passo 3, cada processador p_i pode resolver o problema de *List Ranking* para os elementos armazenados em p_i seqüencialmente.

A prova de que o algoritmo requer, com alta probabilidade, $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ rodadas de comunicação é apresentada a seguir conforme [4].

Primeiramente vamos estudar uma amostra aleatória em uma lista ligada.

Considere uma lista ligada com um conjunto S de n elementos. Mostraremos que se nós selecionarmos n/p elementos aleatórios (pivôs) de S, então com alta probabilidade, estes pivôs irão dividir S em sublistas cujo tamanho máximo é limitado por $3p\ln(n)$.

Lema 1. $xk \geq n$ elementos escolhidos aleatoriamente de S (pivôs) particiona a lista S em sublistas S_i, tal que o tamanho da maior sublista é no máximo n/x com probabilidade de no mínimo $1 - 2x(1 - \frac{1}{2x})^{xk}$.

Prova. Suponha que os elementos de S são ordenados por seus ranks. Esta lista ordenada pode ser vista como $2x$ segmentos de tamanho $\frac{n}{2x}$. Se todo segmento contém no mínimo um pivô, então $max_{1 \leq j \leq xk}|S_j| \leq \frac{n}{x}$. Considere um segmento. Como os pivôs são escolhidos aleatoriamente, a probabilidade de que um pivô específico não esteja no segmento é de $(1 - \frac{1}{2x})$. Como xk pivôs são selecionados independentemente, a probabilidade de que nenhum pivô esteja no segmento é $(1 - \frac{1}{2x})^{xk}$. Portanto, mesmo assumindo exclusão mútua, a probabilidade de que exista um segmento que não contenha nenhum pivô é no máximo $2x(1 - \frac{1}{2x})^{xk}$. Assim, todo segmento contém no mínimo um pivô com probabilidade de no mínimo $1 - 2x(1 - \frac{1}{2x})^{xk}$.

Corolário 1. $xk \leq n$ pivôs escolhidos aleatoriamente particionam a lista S em $xk + 1$ sublistas S_i, tal que existe uma sublista S_i de tamanho maior que $c\frac{n}{x}$ com probabilidade de no máximo $\frac{2x}{c}(1 - \frac{c}{2x})^{xk} \leq \frac{2x}{c}e^{-\frac{1}{2}ck}$.

Lema 2. Considere $xk \leq n$ pivôs escolhidos aleatoriamente que particionam S em $xk + 1$ sublistas S_i, e seja $m = max_{0 \leq i \leq xk}|S_i|$. 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 1 implica que $Prob\{m > c\frac{n}{x}\} \leq \frac{2x}{c}e^{-\frac{1}{2}ck}$.

Nós observamos que, para $c > 2$,

$$\ln(x) + 2\ln(n) \leq k \Rightarrow$$

$$\frac{2}{c}\ln(\frac{2x}{c}) + 2\ln(n) \leq k \Rightarrow$$

$$\ln(\frac{2x}{c}) + c\ln(n) \leq \frac{ck}{2} \Rightarrow$$

$$\frac{2x}{c}n^c \leq e^{\frac{ck}{2}} \Rightarrow$$

$$Prob\{m > c\frac{n}{x}\} \leq n^{-c}.$$

Teorema 1. n/p pivôs escolhidos aleatoriamente particionam S em $\frac{n}{p} + 1$ sublistas S_j com $m = max_{0 \leq j \leq p}|S_j|$ tal que $Prob\{m \geq c3p\ln(n)\} \leq \frac{1}{n^c}$, $c > 2$.

Prova. Seja $x = \frac{n}{3p\ln(n)}$, $k = \ln(x) + 2\ln(n) = 3\ln(n) - \ln(3p\ln(n))$. Então $xk = \frac{n}{p}\frac{3\ln(n) - \ln(3p\ln(n))}{3\ln(n)} \leq \frac{n}{p}$, e Teorema 1 segue do Lema 2.

Agora, para provarmos o Passo 1 do algoritmo, recaímos no seguinte:

Lema 3. Considere uma variável aleatória X com distribuição binomial. Seja n o número de possibilidades, onde cada uma é sucesso com probabilidade q. A expectativa de X é $E(X) = nq$ e $Prob\{X > cnq\} \leq e^{-\frac{1}{2}(c-1)^2nq}$, para qualquer $c > 1$.

Para a implementação do Passo 2, simplesmente simula-se a técnica do *pointer jumping*. (Para todo o x em paralelo: WHILE $next(x) \neq nextPivot(x, S')$ DO $next(x) := next(next(x))$.) Do Teorema 1 segue que, com alta probabilidade, $m(S, S') \leq 3p\ln(n)$. Portanto, o Passo 2 requer, com alta probabilidade, no máximo $\log(3p\ln(n)) = \log(3p) + \log\ln(n)$ rodadas de comunicação. O passo 3 requer 1 rodada de comunicação e o Passo 4 é imediato. Em resumo, obtém-se:

Teorema 2. O Algoritmo probabilístico resolve o problema do *List Ranking*, usando, com alta probabilidade, no máximo $1 + \log(3p) + \log\ln(n)$ rodadas de comunicação e $\tilde{O}(n/p)$ computação local.

Observa-se que, se $\frac{n}{p} \leq e^{(3p)^\alpha}$ para algum $\alpha > 1$ então,

$$\ln(n) \leq \ln(p) + (3p)^\alpha \Rightarrow$$

$$\log\ln(n) \leq \log(\ln(p) + (3p)^\alpha) \leq \log(2(3p)^\alpha) \Rightarrow$$

$$\log\ln(n) \leq 1 + \alpha\log(3p) \Rightarrow$$

$$\log(3p) + \log\ln(n) \leq 1 + (\alpha + 1)\log(3p).$$

Isto implica que

Corolário 2. Se $\frac{n}{p} \leq e^{(3p)^\alpha}$, para alguma constante $\alpha > 1$, então o número de rodadas de comunicação requeridas pelo Algoritmo Probabilístico é limitado por $2 + (\alpha + 1)\log(3p) = \tilde{O}(\log(p))$.

3.2 Algoritmo Determinístico

Antes do algoritmo propriamente dito é necessário fazer algumas definições.

Seja L uma lista ligada com n elementos representada por $s[1 \cdots n]$. Para cada $i \in \{1 \cdots n\}$, $s[i]$ é um ponteiro para o sucessor de i na lista L. O último elemento da lista L, λ é aquele onde $s[\lambda] = \lambda$. Denominamos i e $s[i]$ por vizinhos. A distância entre i e j, $d_L(i,j)$, é o número de nós em L entre i e j mais um. O problema de *List Ranking* consiste em calcular para cada $i \in L$ a distância entre i e λ, referido como $rank_L(i) = d_L(i,\lambda)$.

Cada processador recebe inicialmente n/p elementos da lista com seus respectivos ponteiros.

Um *r-ruling set L'* de L é definido como um subconjunto de elementos selecionados de L com as seguintes características: (1) Dois vizinhos nunca são selecionados. (2) A distância de qualquer elemento não selecionado ao próximo elemento selecionado é no máximo r.

Algoritmo *List Ranking* determinístico

Entrada: Uma lista ligada L de comprimento n onde cada processador armazena n/p elementos $i \in L$ e seus respectivos ponteiros $s[i]$. *Saída*: Para cada elemento i seu *rank* $rank_L(i)$ em L.

1. Calcular $O(p^2)$-*ruling set* R com $|R| = O(n/p)$ como descrito no algoritmo abaixo.

2. Fazer um broadcast de R para todos os processadores.

3. Calcular seqüencialmente em cada processador o *List Ranking* de R, isto é calcular para cada $j \in R$ seu $rank_L(j)$ em L.

4. Obter para cada elemento da lista $i \in L - R$ sua distância $d_L(i, s_R[i])$ para o próximo elemeno $s_R[i]$ em R através do pointer jumping.

5. Calcular em cada processador os *ranks* dos seus elementos $i \in L - R$ com: $rank_L(i) = d_L(i, s_R[i]) + rank_L(s_R[i])$.

A parte difícil do algoritmo é a computação de um $O(p^2)$-*ruling set* R de tamanho $O(n/p)$ que mostraremos no algoritmo a seguir. Dado um $O(p^2)$-*ruling set* a prova de que o algoritmo funciona é imediata. Observamos também que os passos de 2 a 5 podem ser implementdados em $O(\log p)$ rodadas de comunicação com $O(n/p)$ computação local por rodada.

Definimos um *intervalo-s* de comprimento k como uma seqüência $I = (i_1, ..., 1_k)$ de elementos da lista com $s[i_j] = i_{j+1}$, $1 \leq j \leq k - 1$.

Para calcular o $O(p^2)$-*ruling set* cada elemento de L deve ser rotulado com o índice do processador. Assim, $L(i)$ é o número do processador onde i se encontra. Temos, portanto, no máximo p rótulos.

Algoritmo para calcular um $O(p^2)$-*ruling set*

Entrada: Uma lista ligada L de comprimento n onde cada processador armazena n/p elementos $i \in L$ e seus respectivos ponteiros $s[i]$. *Saída*: Um conjunto de nós selecionados de L representando $O(p^2)$-*ruling set* de tamanho $O(n/p)$.

1. Marcar todos os elementos da lista como nós não selecionados.

2. Para todo $i \in L$ fazer em paralelo
   Se $L(i) < L(s[i]) > L(s[s[i]])$ então
   marcar s[i] como selecionado

3. Seqüencialmente, em cada processador, processar as sublistas de elementos subsequentes que estão armazenados no mesmo processador. Para cada sublista, marcar todo segundo elemento. Se a sublista tem somente dois elementos, e ambos os vizinhos não possuem um rótulo menor, marcar os elementos da sublista como não selecionados.

4. Para $k = 1 \cdots \log p$ fazer

4.1. Para todo elemento $i \in L$ fazer em paralelo
Se $s[i]$ é não selecionado então
$s[i] \leftarrow s[s[i]]$.

4.2. Para todo elemento $i \in L$ fazer em paralelo
Se $(i, s[i], s[s[i]]$ são selecionados) e not $(L(i) < L(s[i]) > L(s[s[i]]))$ e $(L(i) \neq L(s[i]))$ e $(L(s[i]) \neq L(s[s[i]]))$ então
marcar s[i] como não selecionado.

4.3. Seqüencialmente, em cada processador, processar as sublistas de elementos selecionados subseqüentes da lista que estão armazenados no mesmo processador. Para cada sublista marcar todo segundo elemento como não selecionado. Se a sublista tem apenas dois elementos e ambos os vizinhos não possuem rótulos menores, então marcar ambos elementos da lista como não selecionados.

5. Marcar o último elemento como selecionado.

O Teorema 3 garante a complexidade do algoritmo no modelo CGM para computar o *List Ranking*. Sua demostração está descrita a seguir conforme [2].

Primeiro provaremos que o conjunto de elementos selecionados no final do algoritmo que calcula $O(p^2)$-*ruling set* tem tamanho máximo $O(n/p)$.

Lema 4. Após a k-ésima iteração no passo 4, não há mais que dois elementos selecionados em qualquer intervalo-s de comprimento 2^k na lista original L.

Prova. Por indução em k. O Lema é trivial para $k = 1$. Seja S um intervalo-s de comprimento 2^k para a lista original L e sejam S_1 e S_2 a primeira e a segunda metade de S, respectivamente. Assuma que depois da iteração $k - 1$ no passo 4, S_1 e S_2 contém no máximo dois elementos selecionados cada. Denote por $e_1, ..., e_4$ os quatro elementos (no máximo) selecionados ordenados em relação à L. Observe que a distância entre estes elementos, em relação à L, é no máximo 2^k. Considere agora a iteração k no passo 4. Após o passo 4.1, quaisquer dois elementos selecionados têm a distância máxima de 2^k e os elementos não-selecionado entre eles (com relação à L) estão diretamente conectados por uma ligação que é representada pelo vetor s atual. Portanto, $s[e_1] = e_2$, $s[e_2] = e_3$, $s[e_3] = e_4$. Note que há, no máximo, quatro casos possíveis (não simétricos) para aplicar os passos 4.2 e 4.3 para $e_1, ..., e_4$. Se $e_1, ..., e_4$ são dois a dois distintos, então somente o passo 4.2 é aplicado. Se $e_1, ..., e_4$ são todos iguais, então o passo 4.3 é aplicado. Se $e_1, ..., e_3$ são dois a dois distintos e $e_3 = e_4$, então o passo 4.2 é aplicado para $e_1, ..., e_3$ e o passo 4.3 para e_3, e_4. O outro caso possível é $e_1 \neq e_2 = e_3 \neq e_4$. Então o passo 4.2 é aplicado para e_1 e e_4, e o passo 4.3 é aplicado para e_2 e e_3. Em qualquer

caso, depois dos passos 4.2 e 4.3, no máximo dois elementos em $e_1, ..., e_4$ ainda estarão selecionados.

Agora provaremos que elementos subseqüentes selecionados no final do Algoritmo de cálculo do $O(p^2)$-*ruling set* têm a distância máxima de $O(p^2)$ na lista original L. Primeiro precisamos do seguinte.

Lema 5. Depois de cada execução do passo 4.3, a distância (com relação ao vetor s atual) entre dois elementos subseqüentes selecionados é no máximo $O(p)$.

Prova. Considere dois elementos subseqüentes selecionados e_1 e e_2. Há três casos possíveis: (1) e_1, e_2 e todos os elementos entre eles, em relação à s, têm o mesmo rótulo. (2) Para e_1, e_2 e todos os elementos entre eles, em relação à s, qualquer par de elementos consecutivos possuem rótulos diferentes. (3) O caso misturado onde alguns pares de elementos subsequentes têm o mesmo rótulo. Note que no caso (3) é impossível para três ou mais elementos consecutivos tenham o mesmo rótulo, porque um deles será um elemento selecionado (passo 4.3). No caso (1) a distância entre e_1 e e_2 é, no máximo 2, por causa do passo 4.3. No caso (2), devido ao esquema para rotular os elementos, há no máximo p rótulos diferentes. Portanto, por [21], a distância é, no máximo, $O(p)$. O caso (3) é equivalente ao caso (2) exceto por um fator de dois.

Lema 6. Após a k-ésima execução do passo 4.3, dois vizinhos-s com respeito ao vetor s atual têm distância $O(2^k)$ com respeito à lista original L.

Prova. Conseqüência do fato de que somente k operações *pointer jumping* foram efetuadas no passo 4.1.

Lema 7. A distância entre quaisquer dois elementos subseqüentes selecionados é, no máximo, $O(p^2)$ com respeito à lista original L.

Prova. Segue dos Lemas 5 e 6.

Lema 8. Em um modelo CGM com p processadores e $O(n/p)$ memória local por processador, $n/p \geq p^{\epsilon}(\epsilon > 0)$, o algoritmo determinístico calcula um $O(p^2)$-*ruling set* de tamanho $O(n/p)$ em $O(\log p)$ rodadas de comunicação com $O(n/p)$ computação local por rodada.

Prova. Corretude do algoritmo que calcula $O(p^2)$-*ruling set* segue dos Lemas 4 a 7. Para $n/p \geq p^{\epsilon}$, o algoritmo de ordenação em [22] requer $O(1)$ rodadas com $O(n/p)$ computação local por rodada. A comunicação utilizada pelo algoritmo consiste de duas ordenações globais para os passos 2 e 3, e 3 $\log p$ ordenações globais para o passo 4. Toda computação local em cada passo pode ser executada

em tempo linear.

Em resumo, obtém-se

Teorema 3. O problema do *List Ranking* para uma lista ligada com n elementos pode ser resolvido no modelo CGM com p processadores e $O(n/p)$ memória local por processador, $n/p \geq p^{\epsilon}(\epsilon > 0)$, em $O(\log p)$ rodadas de comunicação e $O(n/p)$ computação local por rodada.

Capítulo 4

Implementações

Implementamos os algoritmos descritos no capítulo 3 utilizando a linguagem C e a biblioteca *MPI (Message Passsing Interface)*.

Os programas desenvolvidos no presente trabalho seguem o modelo SPMD (*Single Program Multiple Data*), no qual cada processo executa o mesmo programa com dados distintos. Todos os nossos programas recebem como entrada um arquivo contendo uma lista ligada a qual desejamos determinar o *List Ranking*. O arquivo de entrada também fornece o início da lista. Inicialmente um dos processos lê toda a lista e distribui igualmente entre os processadores. O tempo de leitura e distribuição de dados de entrada não é incluido na contagem do tempo dos programas. Para podermos analisar o desempenho dos programas paralelos, foi implementada uma versão seqüencial do *List Ranking*. A contagem de tempo dos programas paralelos é feita pela função *MPI_Wtime* do MPI. Para que houvesse coerência entre as medidas de tempo seqüencial e paralelo, utilizamos a biblioteca MPI também no programa seqüencial, de forma que pudéssemos utilizar a função MPI_Wtime para computar o tempo do programa. Desta forma o programa seqüencial utiliza a biblioteca MPI, mas não utiliza as primitivas de comunicação, e logicamente é executado por apenas um processo, sendo que a única função da biblioteca MPI nesse programa é computar o tempo. A mesma entrada é usada no programa seqüencial, e o tempo de leitura da lista ligada também não é considerado. As listas ligadas que serviram de entrada para os programas foram geradas de forma aleatória.

Utilizamos um sistema de processamento paralelo do tipo *Beowulf* do Instituto de Matemática e Estatística da USP. O objetivo do *Beowulf* do IME é fornecer uma plataforma de computação de alto desempenho para o desenvolvimento de ferramentas computacionais utilizadas em aplicações de bioinformática. Por isso,

o sistema ganhou o nome de *Biowulf/IME*.

O Biowulf/IME é formado por 16 AMD PCs, conectados por uma chave (*switch*) Fast Ethernet 100Mbit/seg. O *Node 1* é o servidor e seu nome é tiramisu.ime.usp.br para o acesso externo. O sistema está conectado à rede do IME via Ethernet 100Mbit/seg. Cada AMD PC é basicamente configurado como segue [20].

- Processador: 1.2 GHz AMD Thunderbird Athlon, 256KB L2 cache.
- Memória: 768MB PC 133 SDRAM.
- Disco: 30.73GB ATA100 7200 RPM HD Deskstar 756GXP DLT-307030 IBM.
- Sistema Operacional: Debian Linux 2.2.19.

Utilizamos o compilador GNU gcc 2.95.2-13 e o software MPI-LAM.

4.1 Implementação do Algoritmo Probabilístico

Vamos, nessa seção, detalhar os aspectos considerados mais importantes em cada passo na implementação do algoritmo probabilístico de Dehne e Song [4].

No primeiro passo, selecionamos como pivô os primeiros $\frac{n}{p^2}$ elementos de cada processador. Isto só foi possível porque consideramos que a lista ligada é gerada aleatoriamente. Com isso, a execução do algoritmo fica mais rápida, uma vez que não é necessário fazer o sorteio dos pivôs. O primeiro e o último elemento da lista também são tratados como pivôs.

O passo 2, o qual calcula o *list ranking modificando*, é resolvido utilizando-se a técnica do *pointer jumping*. Um vetor auxiliar é usado para guardar o sucessor do sucessor de cada elemento. A atualização destes dados é feita utilizando-se a função *MPI_Alltoallv*, que realiza a comunicação de todos os processos entre si.

No passo 3, o algoritmo sugere troca completa dos valores de *NextPivot* e *DistToPivot* para todos os pivôs. Isto poderia ser implementado com a função *MPI_Allgather* para os dois conjuntos de valores. Entretando, visando minimizar o tempo de comunicação e conseqüentemente de execução do programa, enviamos os valores de *NextPivot* e *DistToPivot* somente para um processador. Depois, calculamos o *list ranking* para os pivôs somente neste processador. Como o primeiro elemento da lista foi marcado com pivô, não precisamos descobrir o início da lista

dos pivôs. Após computar enviamos o *list ranking* dos pivôs para os demais processadores. Finalmente, no passo 4, cada processador resolve o PRL para seus elementos seqüencialmente.

4.2 Implementações do Algoritmo Determinístico

A diferença fundamental entre o algoritmo determinístico e probabilístico está na determinação dos elementos escolhidos como pivô.

Foram implementadas duas versões do algoritmo determinístico. A primeira delas segue exatamente os passos do algoritmo proposto por Dehne *et al.* [2], [3]. Na outra versão realizamos alterações no cálculo do *ruling set* que detalhamos a seguir.

Para o cálculo do $O(p^2)$-*ruling set* o passo 3 do algoritmo original processa as sublistas de elementos subseqüentes que estão armazenados no mesmo processador. Para cada sublista, é marcado todo segundo elemento. Se a sublista tem somente dois elementos e ambos vizinhos não possuem um rótulo menor, são marcados os elementos da sublista como não selecionados. Alteramos este passo não marcando todo segundo elemento e sim todo *p-ésimo* elemento. Também não fazemos tratamento algum se a sublista tem somente dois elementos.

O passo 4.3 do algoritmo é: Seqüencialmente, em cada processador, processar as sublistas de elementos selecionados subseqüentes da lista que estão armazenados no mesmo processador. Para cada sublista marcar todo segundo elemento como não selecionado. Se a sublista tem apenas dois elementos e ambos os vizinhos não possuem rótulos menores, então marcar ambos elementos da lista como não selecionado. Este passo não é executado na versão modificada.

Os demais passos dos algoritmos determinístico e determinístico modificado são iguais. Os resultados destas alterações são apresentados no capítulo 5.

Capítulo 5

Resultados Experimentais

Poucos artigos lidam com a parte de implementação do PLR. Reid-Miller em [17] apresenta uma implementação para o Cray C-90 de diferentes algoritmos PRAM que dão bons resultados, entretanto estas implementações são específicas para esta máquina [5]. Dehne e Song em [4] realizam algumas simulações, mas eles somente mostram os resultados em relação ao número de rodadas de comunicação.

Santana *et al.* em [6] mostra a escalabilidade do algoritmo probabilístico de Dehne e Song. Entretanto, Santana em [7] não obtém uma implementação com acelerações favoráveis.

Sibeyn em [14] e Sibeyn *et al.* em [15] mostram vários algoritmos para PLR com técnicas derivadas do PRAM e outras novas. Eles ajustam seus algoritmos de acordo com a rede de interconexão do Intel Paragon. Os resultados são bons e promissores desde que mais de dez processadores sejam usados. Nestes trabalhos as implementações são específicas para a rede de interconexão da máquina alvo e não parecem ser portáveis [5].

Gustedt *et al.* apresenta em [5] dois algoritmos, um determinístico e outro probabilístico.

O algoritmo determinístico está baseado em duas idéias dadas por algoritmos PRAM. A básica e primeira técnica, chamada de *pointer-jumping*, foi mencionada por Wyllie em [16]. A segunda técnica PRAM usada é um *k-ruling-set* apresentada por Cole e Vishkin em [18].

No probabilístico, a técnica empregada é conhecida por *conjuntos independentes*, como descrita em [10].

As Figuras 5.1 e 5.2 referem-se às implementações de Gustedt [5]. A Figura 5.1 mostra o tempo de execução por elemento em função do tamanho da lista para o algoritmo determinístico, enquanto a Figura 5.2 para o algoritmo probabilístico.

Figura 5.1: Tempo de execução por elemento para o algoritmo determinístico

Figura 5.2: Tempo de execução por elemento para o algoritmo probabilístico

O algoritmo determinístico é sempre mais lento do que o seqüencial. Para o algoritmo probabilístico, a partir de 9 processadores, o algoritmo paralelo se torna mais rápido que o seqüencial. Em ambos os casos o tempo de execução paralelo decresce com o número de processadores empregados. As acelerações são de qualquer modo pequenas, uma vez que para 12 processadores, por exemplo, a aceleração obtida é igual a 1.3 [5].

Sibeyn em [8] apresenta um trabalho que possui resultados experimentais para o PLR. Neste trabalho, um novo algoritmo *one-by-one cleaning*, que realiza $O(p)$ rodadas de comunicação, é implementado.

Para conseguir bons resultados, a idéia é reduzir o número de *start-ups*. Para isso, reduz-se o número de roteamento durante o envio das mensagens.

Esse experimento é realizado em um Intel Paragon. O Paragon é um computador paralelo com uma rede de interconexão tipo *grid*. Para $n \geq 250000$, onde n é o número de elementos da lista, o algoritmo é mais rápido do que o seqüencial [8].

O algoritmo tem um melhor desempenho para valores mais modestos de p (número de processadores). Por exemplo:

Para $p = 36$ e $k = 2^{14}$, obtém-se uma aceleração de 5.3 [8].

Por útimo, encontramos o trabalho de Chan e Dehne em [19]. Neste trabalho, implementou-se o algoritmo determinístico de [2], [3] que apresentamos nesta dissertação no capítulo 3. Seus estudos experimentais foram realizados em duas plataformas paralelas: THOG e ULTRA. O cluster THOG consiste de $p = 64$ nós, cada um com dois processadores Xeon. Os nós são interconectados via chave (switch) usando Ethernet Gigabit. A plataforma ULTRA é uma rede de estações de trabalho consistindo de $p = 10$ Sun Sparc Ultra 10. Os nós são interconectados via Fast Ethernet usando chave (switch) de 100Mb.

Figura 5.3: Curvas dos tempos observados para o algoritmo determinístico na plataforma THOG

Os desempenhos em cada plataforma podem ser observados na Figura 5.3 e na Figura 5.4. Infelizmente, na Figura 5.3 não foi apresentado o tempo do algoritmo seqüencial, inviabilizando o cálculo da aceleração. De qualquer forma, observamos que o tempo de execução em ambas as figuras diminui com o aumento do número de processadores.

Figura 5.4: Curvas dos tempos observados para o algoritmo determinístico na plataforma ULTRA

5.1 Resultados Obtidos Neste Trabalho

Nesta seção, apresentamos os tempos, em segundos, obtidos com as nossas implementações do algoritmo probabilístico de Dehne e Song [4], do algoritmo determinístico de [2], [3] e por último os resultados da sua versão modificada. Cada gráfico mostra os tempos de um determinado programa para uma quantidade de elementos de entrada, assim podemos observar como o programa se comporta em cada situação. Cada um dos tempos apresentados representa a média entre 3 tempos mensurados. Foram utilizadas como entrada listas ligadas de tamanho $n = 1\text{M}$, $n = 8\text{M}$, $n = 16\text{M}$ e $n = 32\text{M}$. O número de processadores p utilizados para os algoritmos paralelos foram 2, 4 e 8.

5.1.1 Resultados do Programa Probabilístico

Nesta subseção, apresentamos os tempos, em segundos, obtidos com a nossa implementação do algortimo probabilístico de Dehne e Song [4].

Na Figura 5.5 observamos que o algoritmo seqüencial é mais rápido que o algoritmo paralelo para qualquer número de processadores. Além disso o tempo de execução para $p = 8$ é pior que para $p = 4$. Isso se deve ao fato do tamanho da entrada, $n = 1\text{M}$, não ser suficientemente grande para compensar o tempo de comunicação.

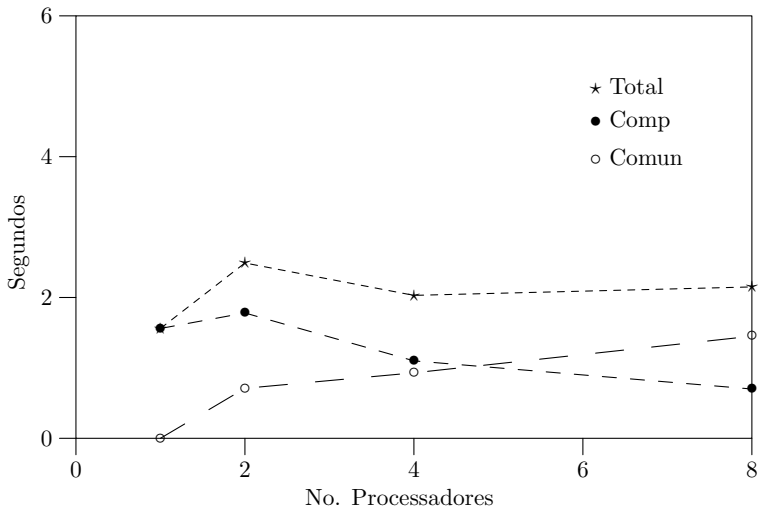


Figura 5.5: Curvas dos tempos observados para o algoritmo probabilístico com entrada $n = 1\text{M}$.

Na Figura 5.6, $n = 8\text{M}$, o tempo de processamento paralelo diminui com o aumento do número de processadores. Para $p = 4$ e $p = 8$ o algoritmo paralelo é mais rápido que o seqüencial.

Na Figura 5.7, $n = 16\text{M}$, o tempo de processamento paralelo também diminui com o aumento do número de processadores. As respectivas acelerações podem ser observadas na Figura 5.8.

Por último, temos a entrada com $n = 32\text{M}$, na Figura 5.9 notamos que o tempo de processamento seqüencial e de processamento paralelo para $n = 2$ é praticamente o mesmo. Para maiores valores de p o tempo paralelo diminui. A Figura 5.10 representa as acelerações obtidas. Para $p = 8$ a aceleração é, aproximadamente, 2,21.

5.1.2 Resultados do Programa Determinístico

Nesta subseção, apresentamos os tempos, em segundos, obtidos com a nossa implementação do algoritmo determinístico de Dehne *el al.* [2], [3].

Para valores de $n = 1\text{M}$ e $n = 8\text{M}$ o algoritmo seqüencial é mais rápido que o algoritmo paralelo.

Na Figura 5.13, $n = 16\text{M}$, observamos que o algoritmo paralelo é mais rápido que o seqüencial somente para $p = 8$.

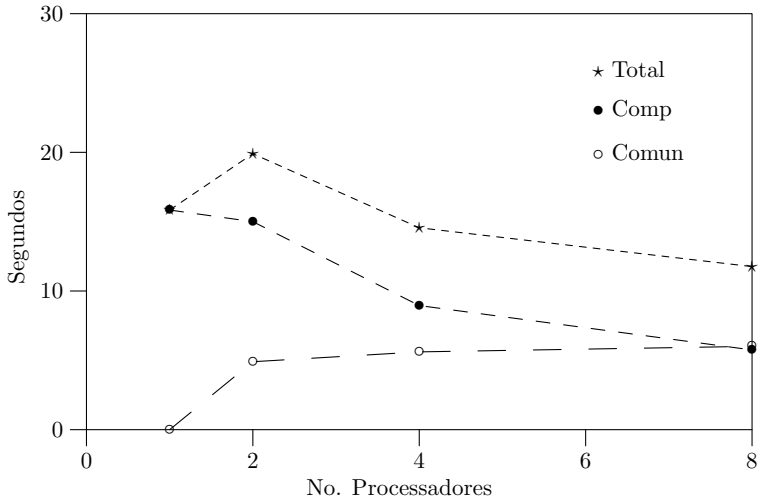


Figura 5.6: Curvas dos tempos observados para o algoritmo probabilístico com entrada $n = 8$M.

Por último, na Figura 5.14, $n = 32$M, o algoritmo paralelo é mais rápido que o seqüencial para $p = 4$ e $p = 8$. As acelerações, que estão representadas na Figura 5.15, são menores que no algoritmo probabilístico para o mesmos valores de n e p.

5.1.3 Resultados do Programa Determinístico Modificado

Nesta subseção, apresentamos os tempos, em segundos, obtidos com a implementação da versão que modificamos do algoritmo determinístico de Dehne *el al.* [2], [3].

Para $n = 1$M o algoritmo seqüencial é mais rápido que o algoritmo determinístico modificado. Entretanto para $n = 8$M, diferentemente, do que ocorre entre o algoritmo derteminístico de [2], [3] e o algoritmo seqüencial, o algoritmo determinístico modificado, com $p = 8$, é um pouco mais rápido que o seqüencial, como podemos observar na Figura 5.17.

Com $n = 16$M, observamos na Figura 5.18 que o algoritmo paralelo modificado é mais rápido que o seqüencial para $p = 8$ e um pouco mais lento para $p = 4$.

Finalmente, para $n = 32$M (Figura 5.19), o algoritmo paralelo modificado é mais rápido que o seqüencial para $p = 4$ e $p = 8$. As acelerações podem ser observadas na Figura 5.20.

Em todos os casos o tempo de execução do algoritmo determinístico modificado é menor que o tempo de execução do algoritmo determinístico.

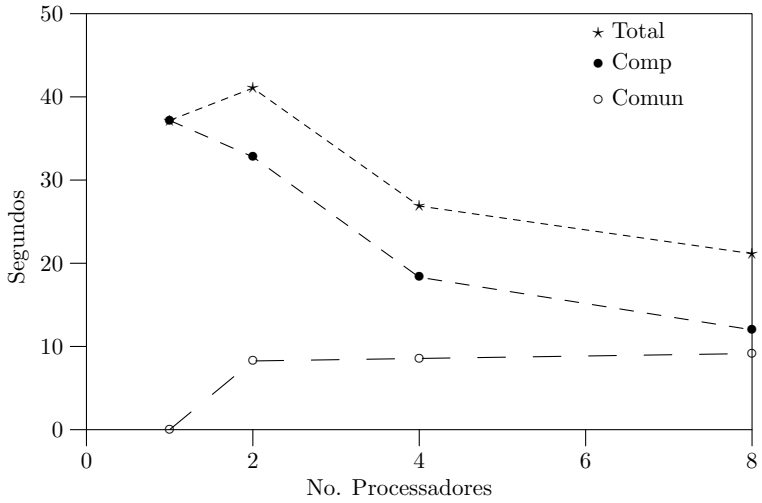


Figura 5.7: Curvas dos tempos observados para o algoritmo probabilístico com entrada $n = 16$M.

A Figura 5.21 apresenta a comparação entre as acelarações obtidas em cada um dos algoritmos implementados para uma entrada de $n = 32$M.

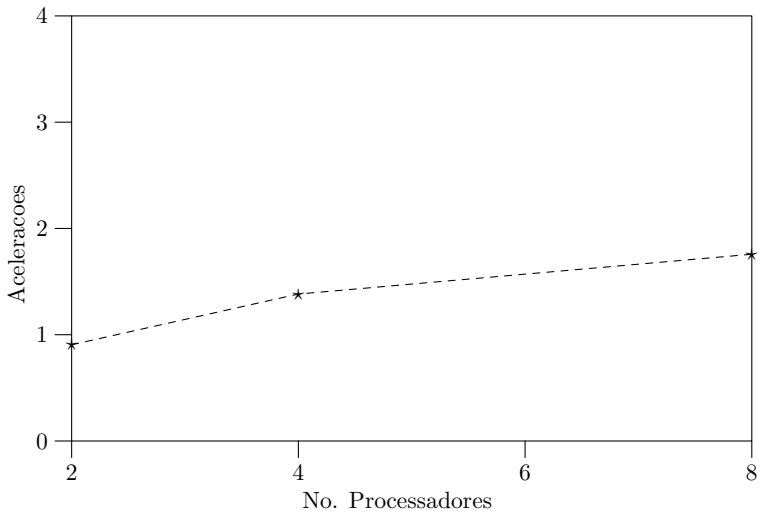


Figura 5.8: Acelerações (*speedups*) obtidas para o algoritmo probabilístico com entrada $n = 16\text{M}$.

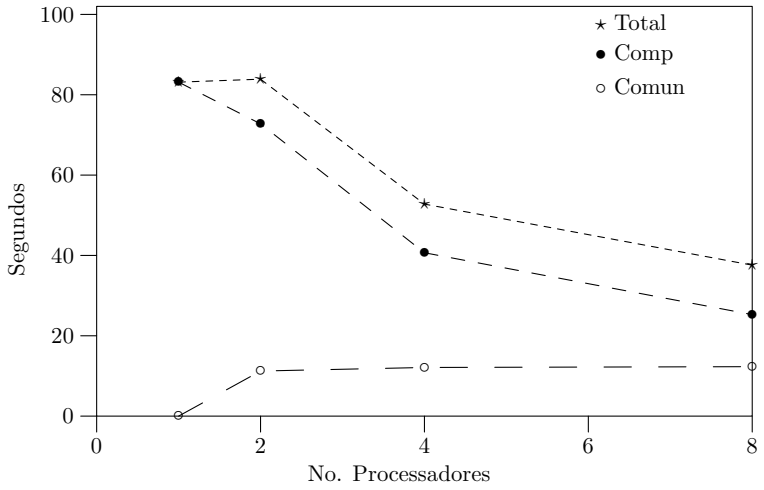


Figura 5.9: Curvas dos tempos observados para o algoritmo probabilístico com entrada $n = 32\text{M}$.

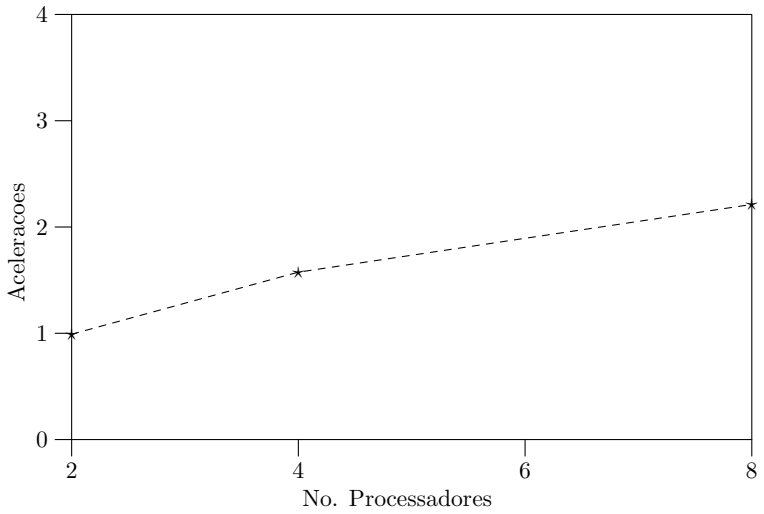


Figura 5.10: Acelerações (*speedups*) obtidas para o algoritmo probabilístico com entrada $n = 32\text{M}$.

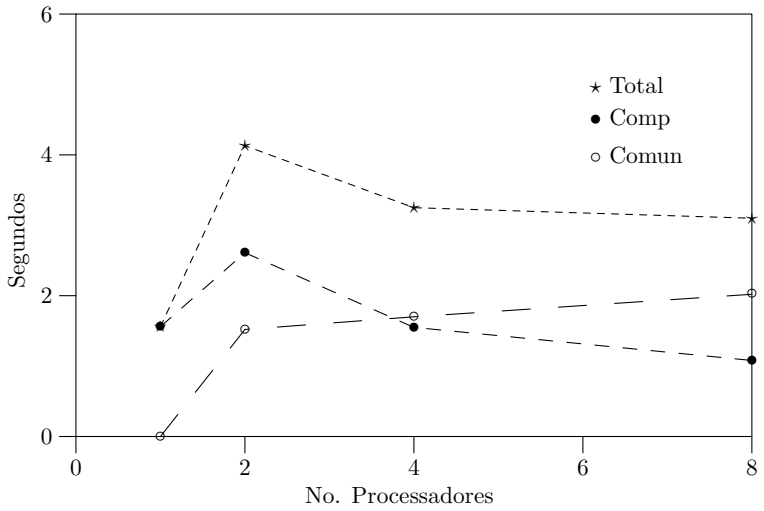


Figura 5.11: Curvas dos tempos observados para o algoritmo determinístico com entrada $n = 1\text{M}$.

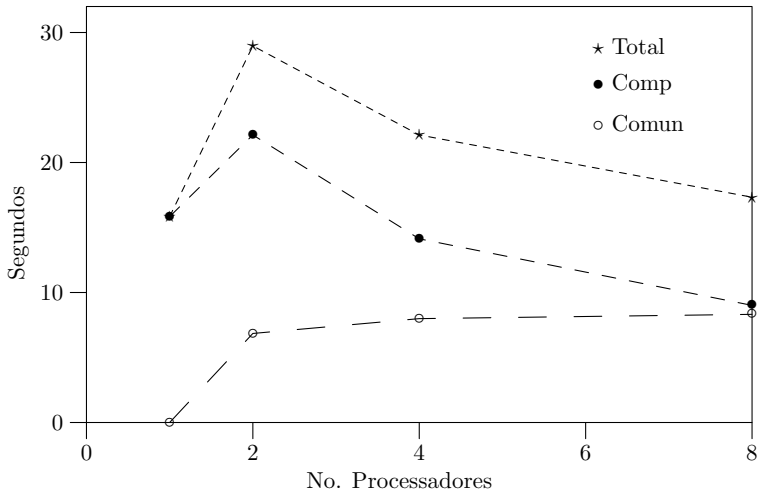


Figura 5.12: Curvas dos tempos observados para o algoritmo determinístico com entrada $n = 8$M.

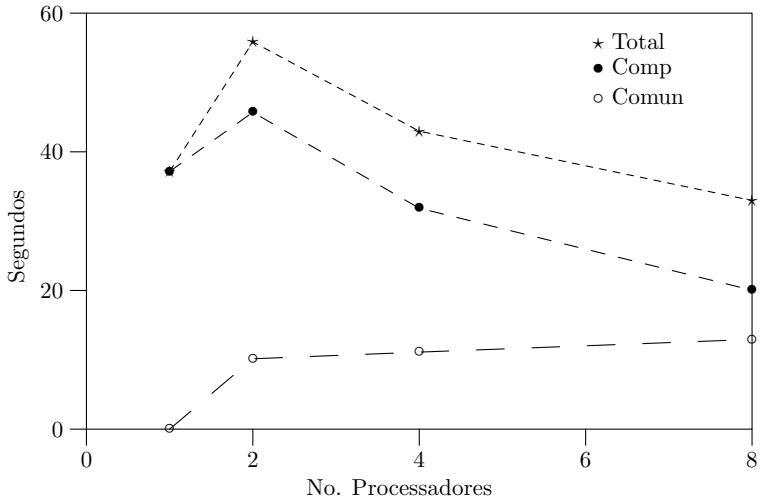


Figura 5.13: Curvas dos tempos observados para o algoritmo determinístico com entrada $n = 16$M.

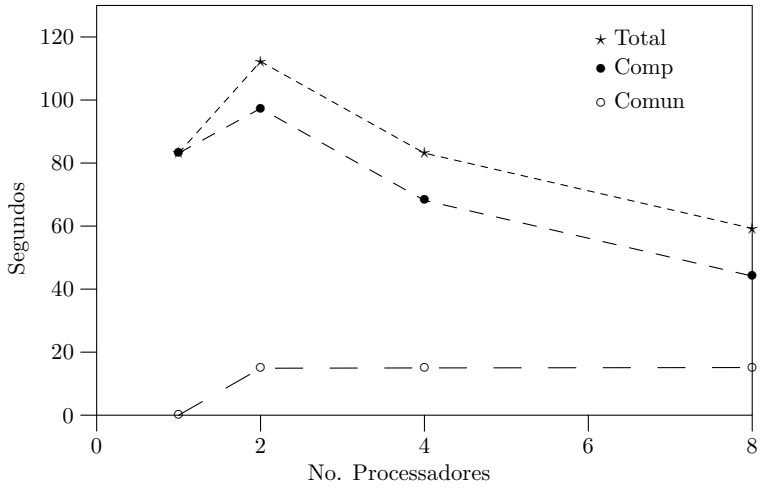


Figura 5.14: Curvas dos tempos observados para o algoritmo determinístico com entrada $n = 32$M.

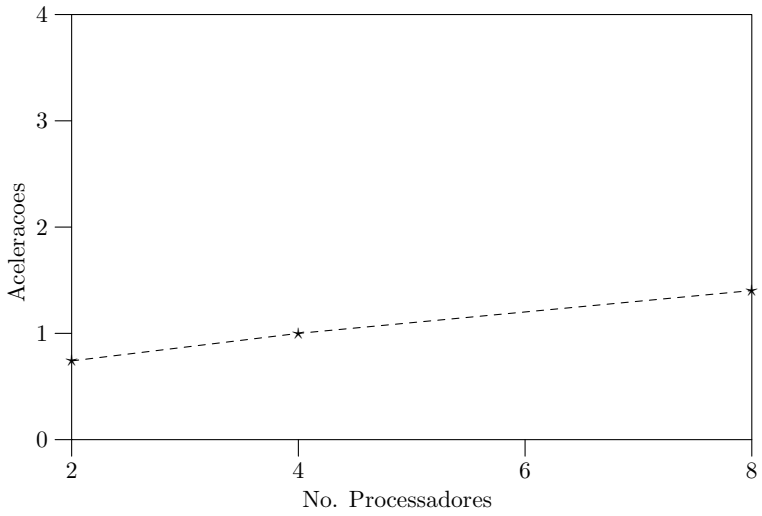


Figura 5.15: Acelerações (*speedups*) obtidas para o algoritmo determinístico com entrada $n = 32$M.

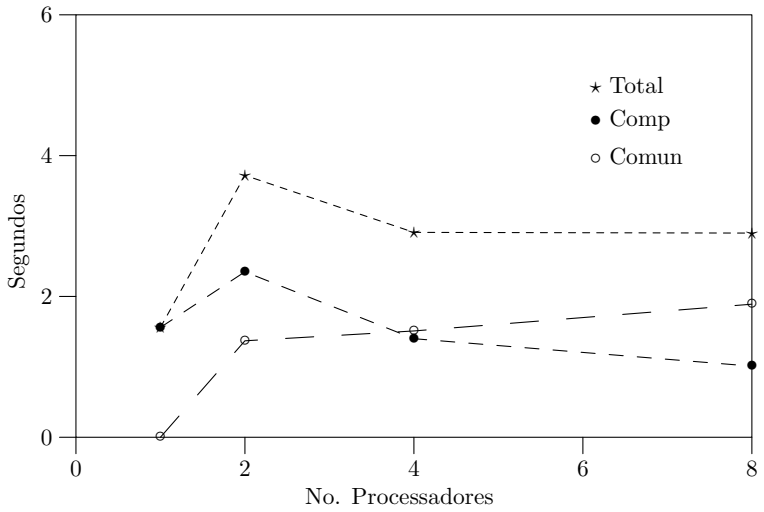


Figura 5.16: Curvas dos tempos observados para o algoritmo determinístico modificado com entrada $n = 1\text{M}$.

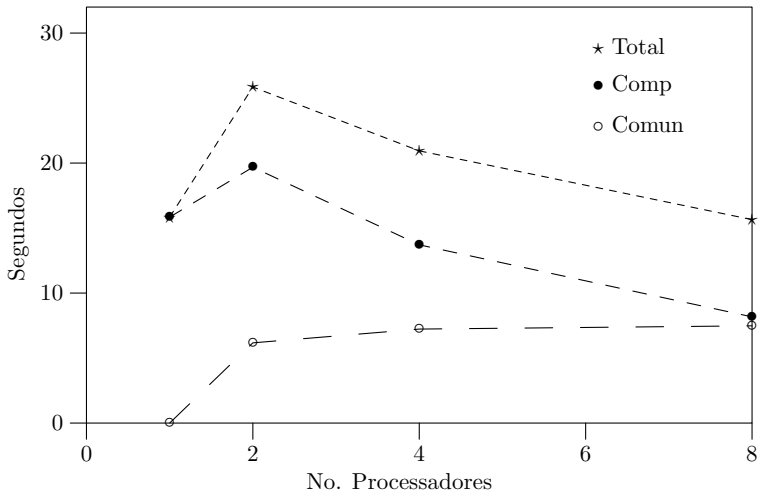


Figura 5.17: Curvas dos tempos observados para o algoritmo determinístico modificado com entrada $n = 8\text{M}$.

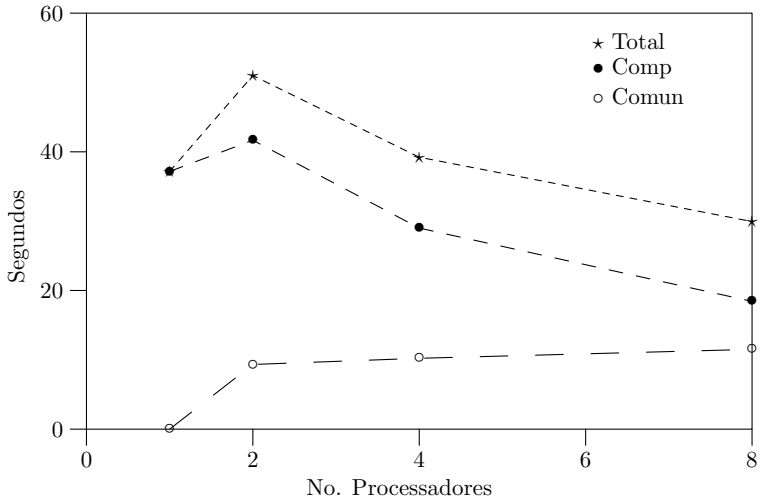


Figura 5.18: Curvas dos tempos observados para o algoritmo determinístico modificado com entrada $n = 16$M.

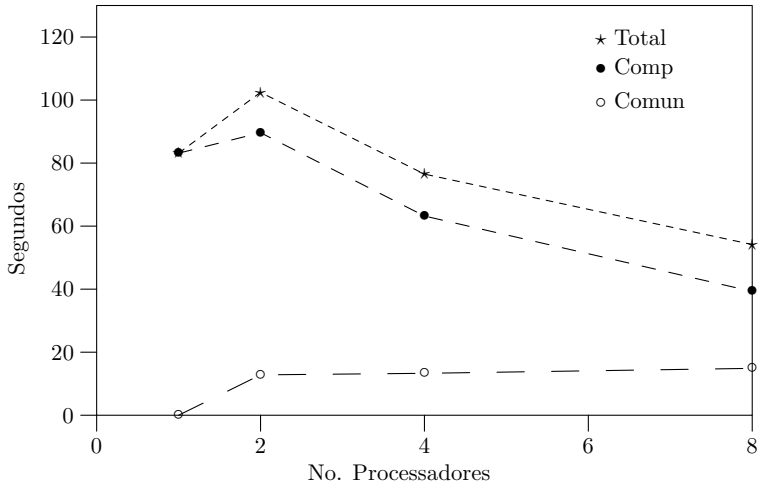


Figura 5.19: Curvas dos tempos observados para o algoritmo determinístico modificado com entrada $n = 32$M.

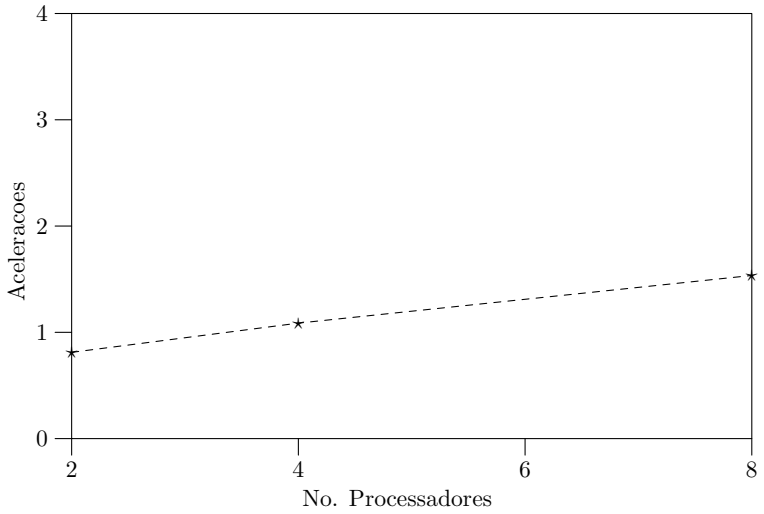


Figura 5.20: Acelerações (*speedups*) obtidas para o algoritmo determinístico modificado com entrada $n = 32\text{M}$.

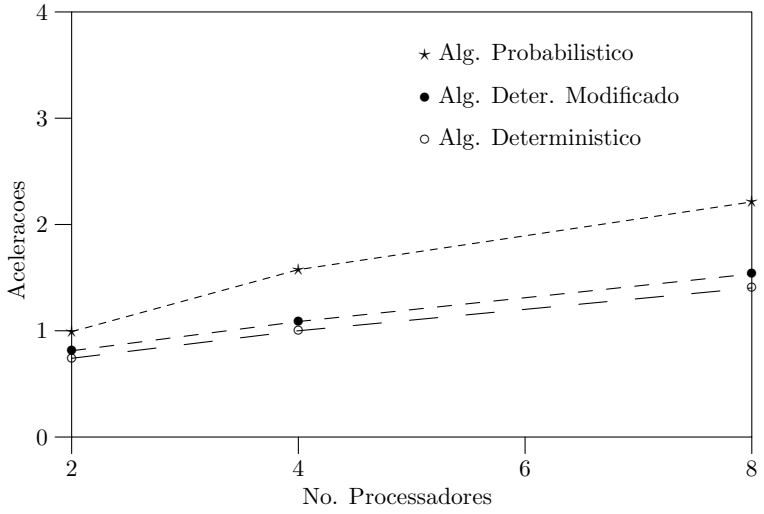


Figura 5.21: Acelerações (*speedups*) obtidas para cada algoritmo com entrada $n = 32\text{M}$.

Capítulo 6

Conclusão

List Ranking é um importante problema básico utilizado por uma grande classe de algoritmos [1].

Este problema foi bastante estudado para o modelo PRAM. Embora o modelo PRAM seja muito conhecido, ele não é adequado para as máquinas paralelas reais, isto porque, o modelo não considera o alto custo de comunicação entre os processadores. Portanto, muitos algoritmos teoricamente eficientes para o modelo PRAM não produzem o desempenho esperado quando implementados em máquinas paralelas reais.

Para resolver este problema foram propostos os modelos realísticos como o BSP e o CGM com a intenção de modelar um algoritmo que tivesse resultados teóricos próximos aos resultados práticos. O modelo CGM é mais simples (simplifica os custos de comunicação), facilitando, assim, o desenvolvimento de algoritmos.

Para o problema do *List Ranking*, diferentemente do algoritmo seqüencial, que pode ser resolvido de forma eficiente, o algoritmo paralelo na maioria dos casos, não leva a uma implementação eficiente [5].

Implementamos o algoritmo probabilístico de Dehne e Song [4] que requer com alta probabilidade, $\log(3p) + \log\ln(n) = \tilde{O}(\log p + \log\log n)$ rodadas de comunicação e $\tilde{O}(n/p)$ computação local. Também foi implementado o algoritmo determinístico de Dehne *et al.* que necessita de apenas $O(\log p)$ rodadas de comunicação e $O(n/p)$ computação local. Além disso, alteramos este último algoritmo e fizemos também sua implementação. Todos os algoritmos foram implementados na linguagem C utilizando a biblioteca MPI e executados em um *Beowulf* com 16 processadores AMD.

Os dados experimentais obtidos mostram um desempenho satisfatório dos programas. Estes dados foram obtidos utilizando-se listas ligadas, geradas aleatoriamente, com tamanhos $n = 1\text{M}$, $n = 8\text{M}$, $n = 16\text{M}$ e $n = 32\text{M}$ e os programas paralelos executados em 2, 4 e 8 processadores.

Em todos os programas paralelos o tempo de execução diminui com o aumento do número de processadores, exceto para $n = 1\text{M}$, onde o tempo de comunicação não é compensado pela quantidade de dados. Com o aumento do tamanho da entrada, os algoritmos paralelos se tornam mais eficientes. Para, por exemplo, $p = 8$ e $n = 16$ ou $n = 32$ todos os programas paralelos são mais rápidos que o programa seqüencial.

Para todos os valores de n e p, nossos experimentos mostram que, o programa determinístico modificado foi um pouco mais rápido que o programa determinístico, ou seja, com uma pequena alteração no algoritmo determinístico conseguimos alguma melhora nos tempos de execução.

O algoritmo probabilístico, embora tenha, na teoria, um maior número de rodadas de comunicação, sua implementação obteve melhor desempenho que o algoritmo determinístico para todas as entradas testadas.

Referências Bibliográficas

[1] S. Baase, Introduction to parallel connectivity, list ranking, and Euler tour techniques. J. H. Reif (ed.) *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publisher, 1993.

[2] F. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, Vol. 33, pp. 183-200, 2002.

[3] E. N. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *24th International Colloquium on Automata, Languages, and Programming (ICALP 97). Lecture Notes in Computer Science*, Vol. 125, pp. 390-400, 1997.

[4] F. Dehne, and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, Vol. 25, n° 1, pp. 1-16, 1997.

[5] I. G. Lassous, and J. Gustedt. Portable list ranking: an experimental study. *Workshop on Algorithm Engineering*, pp. 111-122, 2001.

[6] F. Dehne, F. S. Santana, and S. W. Song. Validação da escalabilidade de um algoritmo paralelo para list ranking. *Anais do XIX Congresso Nacional de Matemática Aplicada*, pp. 111-122, 1996.

[7] F. S. Santana. Algoritmos probabilísticos de list ranking para máquinas paralelas com memória distribuída. *Tese de Mestrado*, USP, 1997.

[8] J. F. Sibeyn. One-by-one cleaning for pratical parallel list ranking. *Algorithmica*, Vol. 32, pp. 345-363, 2002.

[9] J. R. Anderson and G. L. Miller. A simple randomized parallel algorithm for list ranking. *Information Processing Letters*, Vol. 33, No. 5, pp. 269-273, 1990.

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

[11] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, Vol.33, no. 8, pp. 103-111, 1990.

[12] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. *Proc. ACM 9th Annual Computation Geometry*, pp. 298-307, 1993.

[13] F. Dehne. Coarse grained parallel algorithms. *Algorithmica*, 24(3/4): 173-176, 2000.

[14] J. F. Sibeyn. Better trade-offs for parallel list ranking. *Proc. of 9th ACM Symposium on Parallel Algorithms and Architectures*, 221-230, 1997.

[15] J. F. Sibeyn, F. Guillaume, and T. Sidel. Pratical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56:156-180, 1999.

[16] J. C. Wyllie. The Complexity of Parallel Computations. *PhD thesis*, Computer Science Departament, Cornell University, 1979.

[17] M. Reid-Miller. List ranking and list scan on the Cray C-90. *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 104-113, 1994.

[18] R. Cole and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3): 128-142, 1989.

[19] Albert Chan and Frank Dehne. CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters. *Proc. 10th EuroPVM/MPI 2003*, Venice, Italy Sep 29 - Oct 2, 2003.

[20] M. X. T. Delgado. http://www.vision.ime.usp.br/ cage/Beowulf/,2001.

[21] R. Cole and U. Vishkin. Approximate parallel schedulling, part i: The basic technique with applications to optimal parallel list ranking in logaritmic time. *SIAM J. of Computing*, 17(1): 128-142, 1988.

[22] M. T. Goodrich. Communication Efficient Parallel Sorting. *ACM Symposium on Theory of Computing*, 1996.