Algoritmos Paralelos para Classificação de Dados

Karina Kohl Silveira, Tiarajú Asmuz Diverio

Instituto de Informática , UFRGS
Caixa Postal 15064, 91505-970, Porto Alegre – RS
E-mail: kohl@inf.ufrgs.br

Introdução

Há muito tempo, fala-se que a utilidade dos computadores paralelos depende do desenvolvimento de algoritmos paralelos, que operem eficientemente em tais computadores e do desenvolvimento de linguagens de programação paralela, nas quais esses algoritmos possam ser expressos [KRO85]. É notório que os computadores evoluíram, mas essas duas questões básicas (algoritmos e linguagens) continuam na busca de soluções eficientes. Constata-se a veracidade do ditado dos teóricos: "A tecnologia muda mas a Ciência permanece".
Esse trabalho apresenta os primeiros resultados obtidos em uma pesquisa sobre paralelização de algoritmos de classificação em um ambiente computacional de cluster de PCs. Os algoritmos estão sendo estudados quanto ao paralelismo intrínseco ao problema e a complexidade de tempo e de espaço. Os objetivos principais incluem o desenvolvimento de versões desses algoritmos, utilizando paralelismo baseado em troca de mensagens e, o estudo de aspectos de desempenho (*speedup* e eficiência). Outro ponto importante é a análise do impacto da complexidade de comunicação (incluindo sincronização, troca de mensagens e transferência de dados da memória). Ao final são apresentados alguns resultados obtidos a partir das primeiras implementações e testes realizados.

Exploração do Paralelismo

Esse trabalho leva em consideração dois aspectos fundamentais para a exploração do paralelismo inerente aos problemas: a metodologia e a abordagem usada no desenvolvimento de um algoritmo paralelo. *Foster* [FOS95] propôs uma metodologia composta de quatro estágios para se projetar um algoritmo paralelo: *particionamento* (decomposição do problema e do domínio), *comunicação (*gerência do fluxo de informação e coordenação entre as tarefas), *aglomeração (*redistribuição das tarefas em tarefas maiores, se necessário, visando o aumento de desempenho) e *mapeamento* (alocação das tarefas aos processadores de forma eficiente).
O particionamento e a comunicação são estágios que caminham juntos, pois ao projetar a decomposição do problema, precisa-se considerar a comunicação entre as partes. Nessa fase é definida a abordagem que será utilizada na implementação do algoritmo em função da arquitetura disponível e do tipo de paralelismo inerente ao problema [BUY99]. As abordagens mais comumente usadas são: Pipeline, Mestre-Escravo, Divisão-e-Conquista e SPMD.
O ambiente de execução do problema também deve ser levado em consideração. Atualmente, há uma grande expansão dos ambientes de clusters de computadores, compostos por vários nodos, monoprocessados ou multiprocessados, ligados por redes de interconexão de alta velocidade, onde a comunicação se dá por trocas de mensagem. Essas trocas de mensagens, são baseadas na comunicação entre os processos envolvidos no problema, através de envio e recepção de mensagens de forma síncrona ou assíncrona. É num ambiente de *cluster* de computadores que o presente trabalho está sendo desenvolvido.

O Problema de Classificação

Segundo *Kumar* [KUM 94], o problema de classificação de dados é definido como a tarefa de ordenar uma coleção de elementos desordenados em uma ordem monotônica crescente ou decrescente. Especificamente, tem-se $S=(a_1, a_2,...,a_n)$ uma seqüência de *n* elementos em uma ordem arbitrária, a classificação transforma S em uma seqüência monotônica crescente $S'=(a'_1, a'_2,..., a'_n)$ onde $a'_i \leq a'_j$ para $1 \leq i \leq j \leq n$, e S' é uma permutação de S.

Para a resolução desse problema foram escolhidos três algoritmos de classificação de dados, utilizando-se três abordagens diferentes em suas implementações, são eles: *Insertion Sort* (pipeline), *Merge Sort* (divisão e conquista) e o *Rank Sort* (mestre–escravo).

Insertion Sort - A caracterização do *insertion sort* seqüencial é dada pelo princípio no qual os *n* dados a serem ordenados são divididos em dois segmentos: um ordenado e outro a ser ordenado. Os elementos do segmento não ordenado vão sendo transferidos um a um para o segmento ordenado, sendo inseridos nas suas posições corretas[AZE96]. A solução paralela é descrita em *Wilkinson* [WIL99] e baseia-se em uma abordagem *pipeline*. Nessa solução a comunicação entre os processos [FOS95] ocorre da seguinte forma: o primeiro processo (P_0), aceita uma série de números, um por vez, armazena o maior número assim que é recebido e passa para os próximos processos todos os menores números que àquele que está armazenado. Se um número recebido é maior que o número corrente armazenado, esse é passado adiante e trocado pelo novo maior número.

Merge Sort - A idéia do *merge sort* seqüencial consiste em dividir o vetor a ser classificado em dois ou mais, ordená-los separadamente e, depois, intercalá-los dois a dois, formando cada par intercalado, novos segmentos ordenados, os quais serão intercalados entre si, reiterando-se o processo até que resulte apenas um único segmento ordenado. A implementação paralela segue a mesma idéia do algoritmo seqüencial, usando o paradigma Dividir e Conquistar. O particionamento [FOS95]do vetor a ser ordenado é feito em forma de árvore, ou seja, o primeiro processo divide o vetor para dois processos, cada um desses dois, divide o seu subvetor em mais dois, assim sucessivamente até que todos os processos disponíveis tenham uma parte do vetor e possam começar a intercalação aos pares. Na fase de intercalação dos pares, a comunicação [FOS95] entre os processos se dá de maneira semelhante ao particionamento, porém os subvetores vão sendo intercalados até se obter um único vetor ordenado.

Rank Sort - No *rank sort* seqüencial, é contado o número de chaves que são menores que uma determinada chave. Essa contagem representa a posição da chave no vetor, ou seja, o *rank* do número na lista. A versão paralela baseia-se no paradigma Mestre-Escravo, ou seja, a comunicação ocorre apenas entre o mestre e o escravo, não há comunicação entre os escravos. Assumindo-se que se quer ordenar *n* números e tem-se *n* processos, cada processo pode ser alocado para encontrar a posição de um número do vetor, encontrando a posição em *n* passos.

É importante observar que, o paradigma SPMD esteve presente, pois, na implementação dos algoritmos, foi escolhida a biblioteca MPI, que utiliza uma abordagem SPMD.

Resultados

Os resultados apresentados a seguir foram obtidos utilizando-se quatro nodos duais (Pentium Pró 200 Mhz, 128 RAM, com Sistema Operacional Linux) de um cluster interconectado com rede *Myrinet.* A primeira abordagem foi a de distribuir um elemento para cada processador, portanto, um vetor de tamanho *n* necessita *n* processadores para ser classificado. Sabe-se que essa abordagem não é eficiente, pois na prática, necessita-se ordenar um grande número de elementos. Porém, os resultados obtidos são apresentados para que se possa confirmar a ineficiência dessa forma de particionamento.

	P=1	P=2	P=4	P=8
Insert	0,000210	0,000289	0,000460	0,000830
Rank	0,000049	0,001182	0,003116	0,007070
Merge	0,000008	0,002668	0,006392	0,018460

Tabela 1 – Tempos de vetor tamanho *n* para *n* processadores

A unidade de tempo apresentada nos resultados da tab. 1 é o segundo, e os valores são médias obtidas a partir de cinqüenta execuções dos algoritmos. Pode-se observar que o *merge sort* é o melhor algoritmo no caso de existir apenas um processador, porém esse é o único caso em que ele apresenta um bom resultado em comparação aos demais. O *rank sort* é o segundo melhor algoritmo para qualquer caso e o *insertion sort* mostrou os melhores valores para dois, quatro e oito processadores. Observa-se que, enquanto o tempo de processamento do *insertion sort* permanece abaixo de 0,001s para qualquer número de processadores utilizados, os outros dois algoritmos ultrapassam esse tempo, quando suas execuções são realizadas com dois processadores (p =2) e, aumentam seus tempos de processamento em ordem polinomial.

Como já foi dito, classificar um elemento por processador é ineficiente, porém é útil em uma primeira abordagem (acadêmica), para uma visualização do particionamento e da comunicação dos dados, além de possibilitar a observação dos efeitos da comunicação e da sincronização entre os processadores. O primeiro algoritmo implementado de maneira que classifique mais de um elemento por processador foi o *rank sort*. O objetivo foi o de classificar um vetor de tamanho n com p processadores, onde cada processador recebe n/p elementos. A idéia do algoritmo é a mesma, porém, o processador encontra o *rank* de cada um dos n/p elementos atribuídos a ele. A tab.2(a) mostra os tempos médios de execução (em segundos) obtidos das execuções do programa, para quatro tamanhos de entradas diferentes, utilizando-se 1, 2, 4 e 8 processadores. A tab.2(b) mostra os respectivos *speedups,* e a tab.2(c) a eficiência em termos de percentagem.

	n=1024	n=2048	n=4096	n=8192
p=1	0,1554	0,4559	1,46579	5,54984
p=2	0,137	0,2669	0,7305	2,59979
p=4	0,1268	0,2569	0,65919	1,28901
p=8	0,1299	0,5828	1,26128	2,11915

(a) Tempos

n=1024	n=2048	n=4096	n=8192
1	1	1	1
1,13445	1,7082	2,0065	2,1347
1,2253	1,7746	2,2236	4,3055
1,19693	0,7823	1,1621	2,6189

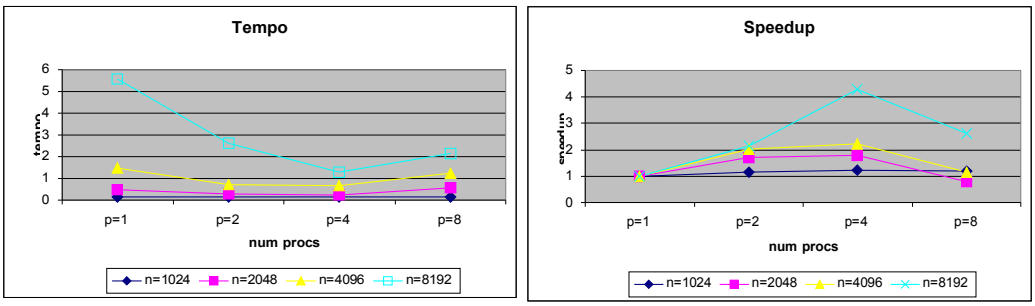
(b)Speedup

n=1024	n=2048	n=4096	n=8192
100%	100%	100%	100%
44%	29%	25%	23%
20%	14%	11%	6%
10%	16%	11%	5%

(c)Eficiência

Tabela 2 – Tempos de execução e speedup do *rank sort*

A fig. 1 mostra o gráfico dos tempos e o gráfico do *speedup* obtidos na execução do programa para as entradas versus os diferentes números de processadores. O *speedup* apresentado é relativo a execução do programa paralelo em um único processador. Observa-se que a execução do programa, usando-se quatro processadores, leva a tempos menores de execução, aumentando com isso o *speedup* e, trazendo ganhos de desempenho. Para qualquer tamanho de entrada, o uso de oito processadores provocou aumento do tempo de execução, causando uma queda no *speedup.* Também, pode-se notar que, os *speedup*s aumentam com o crescimento do tamanho das entradas, tem-se assim que, no caso apresentado, o ganho é proporcional a relação entre o tamanho da entrada e o número de processadores. Para um dado número de processadores, o ganho aumenta com o número de elementos a serem classificados. Sabe-se que o *speedup* máximo em aplicações paralelas é p para p processadores. No caso da execução com $p = 4$, observou-se que: para a entrada de tamanho n =8192, obteve-se um *speedup* superlinear. (Questão a ser analisada na continuidade do trabalho); para as demais entradas, obteve-se tempos menores, porém os *speedups* não ficaram próximos do valor máximo (quatro). Conclui-se, a partir dos dados, que o uso de quatro processadores, produz, para estas estradas, *bons* tempos de classificação.



(a) Tempo de Execução (b) Speedup

Figura 1 – Gráficos de Tempo e Speedup

Conclusões

Inicialmente, os algoritmos foram implementados tendo-se como objetivo ordenar n elementos com n processos, ou seja, um elemento por processo, sendo esses elementos números inteiros. Era esperado que o desempenho não seria melhor que os algoritmos seqüenciais, pois com o processamento paralelo há o acréscimo do tempo de comunicação e os algoritmos seqüenciais possuem uma complexidade ótima. Trabalhando-se com um elemento por processador tem-se pouca computação e muita comunicação. Além disso, na prática, um elemento por processador

se torna inviável no caso de grandes quantidades de elementos a serem ordenados. Apresentou, também, uma versão do *rank sort* estendido para classificar mais de um elemento por processador. Obteve-se aumento no desempenho, como se pode observar na tab. 2 e no gráficos apresentados na fig. 2. Observou-se ainda, que o aumento do número de processadores nem sempre significou o aumento do desempenho.

Portanto, a classificação de um elemento por processador se mostrou uma forma ineficiente de particionar o problema pois gera comunicação excessiva. Na segunda versão implementada do *rank sort,* onde cada processador classifica mais de um elemento, observou-se que se ganha em desempenho até um determinado número de processadores, que para as entradas apresentadas foi $p = 4$. Além do *rank sort*, os outros algoritmos apresentados nesse trabalho estão sendo implementados para que classifiquem vários elementos por processador.

Com esse trabalho, buscou-se aproximar a Teoria da Computação Concorrente com a prática do desenvolvimento de programas paralelos, avançando na direção de criar uma cultura de Processamento de Alto Desempenho.

Trabalhos futuros

Os resultados do primeiro experimento (usar n processadores para classificar n elementos) deverão ser estudados mais profundamente no intuito de inferir custos de paralelização e comunicação dos algoritmos de classificação. Além disso, o mesmo experimento feito para o *rank sort* (classificação de n elementos em p processadores, onde $n>p$) será realizado para os outros algoritmos descritos no texto. Esses algoritmos estão sendo estudados quanto a melhor forma de particionar os dados, visando diminuir a comunicação. A investigação mais aprofundada sobre o *speedup* superlinear obtido no *rank sort,* também será alvo de estudo mais detalhado. Por fim, serão estudados e desenvolvidos programas para outros algoritmos de classificação.

Agradecimentos

Ao CNPq pelo suporte econômico, através das bolsas concedidas, para o desenvolvimento dessa pesquisa.

Referências

[AZE 96] AZEREDO, Paulo A. **Métodos de Classificação de Dados**. Rio de Janeiro: Editora Campus, 1996.

[BUY 99] BUYYA, Rajkumar. **High Performance Cluster Computing**. Upper Saddle River: Prentice Hall,1999.

[FOS 95] FOSTER, Ian. **Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering**. New York: Addison-Wesley, 1995.

[KRO 85] KRONSJO, Lydia. **Computational Complexity of Sequential and Parallel Algorithms.** London: John Wiley & Sons, 1985.

[KUM 94] KUMAR, Vipin; GRAMA, Ananth; GUPTA, Anshul; KARYPIS, Georg. **Introduction to Parallel Computing, Design and Analysis of Algorithms.** California: Benjamin/Cumming Publishing Company, 1994.

[WIL 99] WILKINSON, Barry; ALLEN, Michael. **Parallel Programming Techniques and Applications Using Networkedd Workstations and Parallel Computers.** Upper Saddle River: Prentice Hall, 1999.