Avaliando o Hyperquicksort Utilizando uma NOW

JONES ALBUQUERQUE[1]
CLAUDIONOR J. N. COELHO JR.[2]

[1]Departamento de Ciências Exatas - UFLA
Caixa Postal 37. 37200-000. Lavras, MG.
joa@dcc.ufmg.br

[2]Departamento de Ciência da Computação - UFMG
Caixa Postal 702. 30161-970. Belo Horizonte, MG.
coelho@dcc.ufmg.br

Resumo. Neste trabalho, apresentamos uma implementação para o algoritmo Hyperquicksort utilizando uma NOW, Network Of Workstations, e a máquina abstrata de PVM, Parallel Virtual Machine. Também apresentamos um estudo das complexidades de tempo e espaço para a implementação em questão e avaliamos a eficácia deste algoritmo quando implementado da maneira apresentada neste trabalho.

Palavras Chaves: programação paralela, parallel virtual machine, hyperquicksort, network of workstations

1 Introdução

Quando os problemas excedem os limites físicos e algorítmicos da computação seqüencial, o uso de sistemas de computação paralela e distribuída se faz necessário. Esses sistemas são compostos por vários processadores que operam concorrentemente, cooperando na execução de uma determinada tarefa.

Nas chamadas arquiteturas paralelas o objetivo principal é o aumento da capacidade de processamento, utilizando o potencial oferecido por um grande número de processadores. A comunicação dos processadores é realizada através de redes especiais de conexão ou por meio de uma memória compartilhada, implicando em estruturas fisicamente concentradas.

Por outro lado, nas arquiteturas distribuídas o atrativo principal é a flexibilidade, obtida pela integração de computadores de diversos tipos em um mesmo sistema, sem restrições quanto à distribuição física dos componentes. Contudo, para a utilização de arquiteturas distribuídas se faz necessária a definição de uma máquina virtual para o gerenciamento dos processos concorrentes. Algumas definições de máquinas virtuais alcançaram grande repercurssão como PVM [4], cuja biblioteca de classes foi utilizada na nossa implementação.

Nossa intenção com este trabalho é avaliar a flexibilidade de uso e a real eficiência de uma máquina paralela virtual formada por uma rede de estações de trabalho [1]. O algoritmo hyperquicksort foi escolhido para a avaliação por ser de fácil entendimento e por sua versão seqüencial (quicksort) possuir eficiência incontestável.

Assim, este artigo está organizado da seguinte maneira: a Seção 2 apresenta o algoritmo hyperquicksort; a Seção 3 apresenta a implementação, execução passo-a-passo com um exemplo e discussões sobre a complexidade do algoritmo. A Seção 4 apresenta a análise de SpeedUp com os resultados obtidos. Por fim, as conclusões são apresentadas na Seção 5.

2 O Algoritmo Hyperquicksort

O algoritmo hyperquicksort é uma versão paralela do quicksort. A popularidade do algoritmo quicksort vem de seu comportamento assintoticamente ótimo para o caso médio de $\theta(nlogn)$ [5]. Quicksort é um algoritmo recursivo que repetidamente divide uma lista não ordenada em duas sub-listas menores relativas a um suposto valor médio (*splitter*). Uma das sub-listas (*lowList*) contém valores menores ou iguais ao valor médio suposto; a outra sub-lista (*highList*) contém valores maiores que o valor médio suposto. Assim, a posição de ordenação correta do valor médio já está encontrada, uma vez que a esquerda dele podemos ter a sub-lista com valores menores ou iguais a ele e a sua direita a sub-lista com valores maiores que ele. Recursivamente, novos valores médios são encontrados e novas sub-listas formadas, até que só reste um elemento em cada sub-lista. Então, todas as posições corretamente ordenadas dos elementos estarão determinadas e a lista original ordenada.

Observemos que quando as duas sub-listas são for-

madas, geram dois novos problemas independentes que podem ser resolvidos simultaneamente. Este é o princípio do hyperquicksort [6, 2]: cada processador resolve um sub-problema usando o algoritmo seqüencial quicksort e utiliza um mecanismo de comunicação paralela eficiente para gerar a solução final (*result*) partindo das soluções parciais dos processadores: um hipercubo. Hipercubo é uma organização de processadores na qual os nodos são rotulados por $0, 1, ..., 2^d - 1$ e dois nodos são adjacentes (*partner*) se seus rótulos diferem exatamente em um bit, e.g., um hipercubo de dimensão quatro (4) está ilustrado na Figura 1.

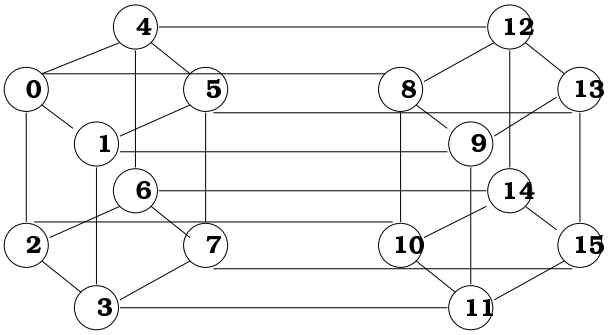

Figura 1: Hipercubo de dimensão (4) quatro, [6]

Na primeira fase do hyperquicksort, cada processador usa o quicksort para ordenar sua lista local. Durante cada passo da segunda fase do algoritmo, um hipercubo é dividido em dois sub-cubos. Cada processador envia valores ao seu adjacente no outro sub-cubo, então cada processador recebe valores e os acrescenta a sua lista remanescente. O resultado desta operação *split-and-merge* é dividir o hypercubo de valores ordenados em dois hypercubos. Como cada processador tem uma lista ordenada de valores, o maior valor no hypercubo inferior é menor que o menor valor no hypercubo superior. Depois de *d* passos de *split-and-merge*, o hypercubo original de 2^d processadores foi dividido em 2^d hypercubos de um único processador. Para melhor entendimento do algoritmo ver uma execução passo-a-passo na Seção 3.3. Um pseudocódigo do algoritmo pode ser encontrado em [6].

3 A Implementação

Esta seção apresenta uma implementação do algoritmo *Hyperquicksort* descrito anteriormente. Para a implementação foi utilizada a biblioteca de classes do PVM - *Parallel Virtual Machine* [4, 3].

3.1 Alguns Comentários sobre a Implementação

A implementação é para uma máquina paralela do tipo MIMD - multiple instructions multiple data - do tipo composta por estações de trabalho - NOW.

O algoritmo foi implementado no modelo mais simples de programação paralela, o modelo mestre-escravo. Neste modelo, o mestre dispara os processos escravos, que são quem de fato realizam a computação.

Nesta implementação, os programas escravos executam o algoritmo hyperquicksort com os seus dados e o programa mestre é responsável por:

- interface com usuário para determinação da dimensão do hipercubo a ser utilizada;
- geração da massa de dados;
- distribuição entre os processos escravos da massa de dados;
- recebimento das partes ordenadas por cada processo escravo.

Observamos também que por simplicidade não fizemos alocação dinâmica dos vetores de dados (*result, lowList, highList*) dos processos escravos. Isso não traz maiores complicações pois como executaremos com vetores de 16K elementos do tipo *int* (32 bits = 4 Bytes) com no máximo 8 processos (3-cube), onde cada processo possui 3 destes vetores, temos uma área máxima alocada de 16Kx4x8x3 = 1,5MB. Isto quando comparado aos espaços de memória (+/- 16MB) disponíveis nas estações de trabalho não representa nenhum agravante. Entretanto, para efeito de complexidade de tempo, observamos que os tamanhos utilizados das estruturas são armazenados. Desta forma, só percorremos o vetor até o tamanho utilizado.

3.2 Executando um Exemplo

A execução dos programas mestre e escravo são mostradas a seguir. As listagens abaixo apresentam todo o processo de obtenção do passo-a-passo, como em [6]:

```
turquesa:~->cd pvm3/bin/SUN4/
turquesa:~/pvm3/bin/SUN4->pvm
pvm> quit

pvmd still running.
turquesa:~/pvm3/bin/SUN4->mestre
Qual a dimensao do hypercubo (0-5)?2
spawn com 4 procs...

Data = [ 97, 48, 16, 8, 66, 96, 17, 49,
         58, 76, 54, 39, 82, 47, 65, 51,
```

```
        11, 50, 53, 95, 36, 67, 86, 44,
        35, 16, 81, 1, 44, 23, 15, 5,]

recebi de 0 [ 1 5 8 11 15 16 16 17]
recebi de 1 [ 23 35 36 39 44 44 47 48]
recebi de 2 [ 49 50 51 53 54 58 65 66 67]
recebi de 3 [ 76 81 82 86 95 96 97]

turquesa:~/pvm3/bin/SUN4->cd /tmp
turquesa:/tmp->id
uid=5019(joa) gid=50(pos) groups=50(pos)
turquesa:/tmp->more pvml.5019
```

A execução passo-a-passo foi obtida mediante a listagem dos comandos de impressão dos processos escravos. Para propiciar um melhor entendimento, as mensagens foram agrupadas de forma que os passos do algoritmo fossem respeitados. Pois, como sabemos, os processos executam de acordo com o escalonamento PVM; o que pode levar, num determinado instante, a processos escravos executarem iterações diferentes do mesmo código.

Na listagem abaixo, *DIM 2: [1] é partner de [3] L2 H6* significa que na dimensão 2, o processador 1 tem como parceiro o processador 3 e suas *lowList* e *highList* possuem 2 e 6 elementos, respectivamente. Para obter quais elementos basta observar as linhas *DIM 2:[1] COMECA: [39 47 51 54 58 65 76 82]* e *DIM 2: [1] recebe splitter=48!* e verificar que valores menores ou igual ao *splitter* estão na *lowList* e valores maiores, na *highList*.

```
turquesa (150.164.10.12:3173) SUN4 3.3.11
ready Thu Jun 19 11:14:44 1997
DIM 2: [0] COMECA: [ 8 16 17 48 49 66 96 97]
DIM 2: [1] COMECA: [ 39 47 51 54 58 65 76 82]
DIM 2: [2] COMECA: [ 11 36 44 50 53 67 86 95]
DIM 2: [3] COMECA: [ 1 5 15 16 23 35 44 81]

DIM 2: [0] 'e root e passa data[3]=48 p/ P0 a P3

DIM 2: [0] recebe splitter=48!
DIM 2: [1] recebe splitter=48!
DIM 2: [2] recebe splitter=48!
DIM 2: [3] recebe splitter=48!

DIM 2: [0] e' partner de [2] com L4 H4!
DIM 2: [1] e' partner de [3] com L2 H6!
DIM 2: [2] e' partner de [0] com L3 H5!
DIM 2: [3] e' partner de [1] com L7 H1!

DIM 2: [0] TERMINA: [ 8 11 16 17 36 44 48]
DIM 2: [1] TERMINA: [ 1 5 15 16 23 35 39 44 47]
DIM 2: [2] TERMINA: [ 49 50 53 66 67 86 95 96 97]
DIM 2: [3] TERMINA: [ 51 54 58 65 76 81 82]

DIM 1: [0] COMECA: [ 8 11 16 17 36 44 48]
DIM 1: [1] COMECA: [ 1 5 15 16 23 35 39 44 47]
DIM 1: [2] COMECA: [ 49 50 53 66 67 86 95 96 97]
DIM 1: [3] COMECA: [ 51 54 58 65 76 81 82]

DIM 1: [0] 'e root e passa data[3]=17 p/ P0 a P1
DIM 1: [2] 'e root e passa data[4]=67 p/ P2 a P3

DIM 1: [0] recebe splitter=17!
DIM 1: [1] recebe splitter=17!
DIM 1: [2] recebe splitter=67!
DIM 1: [3] recebe splitter=67!

DIM 1: [0] e' partner de [1] com L4 H3!
DIM 1: [1] e' partner de [0] com L4 H5!
DIM 1: [2] e' partner de [3] com L5 H4!
DIM 1: [3] e' partner de [2] com L4 H3!

DIM 1: [0] TERMINA :[ 1 5 8 11 15 16 16 17]
DIM 1: [1] TERMINA :[ 23 35 36 39 44 44 47 48]
DIM 1: [2] TERMINA :[ 49 50 51 53 54 58 65 66 67]
DIM 1: [3] TERMINA :[ 76 81 82 86 95 96 97]
```

3.3 Análise de Complexidade do Algoritmo

Apresentamos algumas avaliações com respeito à nossa implementação. Apesar de não possuírmos dados sobre os tempos de comunicação entre os processos, apresentamos as complexidades estimadas para o programa.

3.3.1 Tempo

A complexidade apresentada em [6], em número de comparações, é $O(n(logn + d))$. Como seguimos o algoritmo apresentado também temos esta complexidade, pois o *qsort()* seqüencial e as **d** (dimensão) vezes que o *merge* executa entre as listas *lowList* e *highList* acarretam esta ordem de complexidade de tempo em número de comparações.

Quanto à complexidade de tempo, medir as latências entre as mensagens enviadas e o tempo necessário para um processador mandar um valor para outro não foi possível devido à rede não estar exclusivamente dedicada para este experimento.

Entretanto, podemos calcular o número de mensagens trocadas (enviadas ou recebidas) durante todo o programa. O mestre troca $2 * 2^d$ mensagens, enviando e recebendo para/de cada processador do hipercubo de dimensão d. Cada escravo não *root* troca $3 * d + 2$ (recebendo e enviando para o mestre; e a cada iteração: recebendo o *splitter*, enviando e recebendo para/de *partner*). Se for um escravo *root*, ainda tem mais uma de broadcast do *splitter*.

Portanto, na nossa implementação, o número de mensagens trocadas em todo o programa é $2 * 2^d + 2^d * 3 * d$, desconsiderando as enviadas pelos *root*. Isto leva a 88 mensagens no caso de dimensão 3. Se considerarmos o tempo de latência apresentado em [6] (500 microsegundos), teremos um atraso total na execução do programa de $500 * 10^{-6} * 88 = 44$ milisegundos. Como a nossa máquina de testes é uma NOW, este tempo deve ser consideravelmente aumentado.

3.3.2 Espaço

Por simplicidade não fizemos alocação dinâmica dos vetores de dados (*result, lowList, highList*) dos processos

escravos. Ou seja, cada processo escravo armazena 3 vezes o tamanho da massa de dados. Portanto, o espaço de memória utilizado por cada processador é $O(3n)$, onde n é o número de elementos do vetor de entrada.

O mestre também mantém um vetor para a massa de dados. Na nossa implementação possui 2, mas só por questão de manter a entrada intacta ao final da execução.

Concluindo, a área total armazenada durante a execução do programa é $O(3n * 2^d + 2)$, considerando d a dimensão do hipercubo. Isto nos leva a $O(n)$ se n é muito maior que 2^d; o que geralmente acontece (tamanho da entrada muito maior que o número de processadores).

4 Análise de SpeedUp

Nesta seção é mostrado como o programa se comporta quando executado com uma *massa de dados de 4208, 8417, 16384 elementos gerados aleatoriamente*.

A análise do SpeedUp é realizada sobre a relação entre o tempo de execução do programa seqüencial e o do programa paralelo, isto com paralelismo em 2, 4 e 8 máquinas.

4.1 Procedimento

Uma rotina de medição de tempo foi acrescentada ao código do *mestre.c* para computar o tempo entre mandar os dados para cada escravos e receber deles suas partes ordenadas.

A dimensão do hipercubo irá determinar o número de máquinas/processos necessário para a computação. Hipercubos de dimensão 1, 2 e 3 precisarão de 2, 4 e 8 máquinas, respectivamente. No nosso caso, cada máquina possui um único processador.

Observamos que para a obtenção do tempo seqüencial utilizamos o programa num hipercubo de dimensão 0 (zero), pois o algoritmo seqüencial *qsort()* é muito eficiente e devido aos tempos de comunicação e ao tratamento dado pelo PVM a massas de dados grandes, não seria possível observar o speedUp em relação à implementação seqüencial pura. Deste modo, os resultados apresentados mostram o comportamento do algoritmo *Hyperquicksort* quanto à variação do número de processadores e não quanto ao programa seqüencial puro.

Desta forma, se computarmos o tempo para hipercubo de dimensão 0 (zero) e comparamos com os tempos para as dimensões 1, 2 e 3, teremos uma forma de avaliar o SpeedUp. Os tempos foram obtidos entre 10 tomadas, desprezando-se os valores fora da média acumulada e são apresentados nas Tabelas 1, 2 e 3.

D	# P	T	SO	SM
0	1	0.38	1	1
1	2	0.25	2	1.52
2	4	0.21	4	1.81
3	8	0.39	8	0.97

Tabela 1: Entrada com 4096 elementos: D - dimensão do hypercubo, # P - número de processadores, T - tempo em segundos, SO- valores dos speedups ótimos e SM- speedups medidos (tempoSeqüencial/tempoParalelo)

D	# P	T	SO	SM
0	1	0.40	1	1
1	2	0.26	2	1.54
2	4	0.26	4	1.54
3	8	0.42	8	0.95

Tabela 2: Entrada com 8192 elementos: D - dimensão do hypercubo, # P - número de processadores, T - tempo em segundos, SO- valores dos speedups ótimos e SM- speedups medidos (tempoSeqüencial/tempoParalelo)

D	# P	T	SO	SM
0	1	0.69	1	1
1	2	0.45	2	1.53
2	4	0.35	4	1.97
3	8	0.62	8	1.11

Tabela 3: Entrada com 16384 elementos: D - dimensão do hypercubo, # P - número de processadores, T - tempo em segundos, SO- valores dos speedups ótimos e SM- speedups medidos (tempoSeqüencial/tempoParalelo)

5 Análise de Resultados

Observamos que apesar de considerarmos o seqüencial como o programa paralelo de dimensão 0, o SpeedUp alcançado é mínimo ou até piora quando se aumenta o número de processadores.

Para até 4 processadores nota-se um sensível SpeedUp, mas a partir de 8 processadores os tempos aumentam resultando em SpeedUps cada vez menores, o que pode ser observado na Figura 2. Isto pode ser analisado da seguinte forma: com 8 processadores o custo com comunicação é consideravelmente aumentado devido à quantidade de mensagens e dados trocados entre os processadores, pois PVM manda os dados juntamente com as mensagens de controle.

A distância entre o ótimo ideal e os valores medidos pode ser explicado por estarmos executando sob uma

rede não dedicada (choques de pacotes, latência alta de comunicação, etc) e pelas restrições impostas pelo PVM (pacotes grandes são divididos, dados trafegando pela rede em pacotes, etc).

6 Conclusões

Observamos que a nossa implementação para o problema de ordenação paralela, utilizando o algoritmo *Hyperquicksort* e a máquina abstrata do tipo *NOW - Network Of Workstations* fornecida por PVM, não é escalável. Obviamente, vários fatores contribuem para isso: rede não exclusivamente dedicada para o experimento, choques entre pacotes, latência de comunicação, característica do algoritmo (divisão da massa de dado pode levar a processadores ociosos) e envio da massa de dados através de pacotes realizada por PVM.

Portanto, é desaconselhável um mecanismo de ordenação nas condições que realizamos neste trabalho, pois todos os tempos medidos são superados quando utilizamos o *qsort()* seqüencial. Uma possível solução seria o uso de um mecanismo de memória compartilhada, o que evitaria o tráfego da massa de dados através da rede.

Observamos, ainda, que uma abstração como a fornecida pela máquina virtual PVM simplifica bastante a programação de algoritmos paralelos e distribuídos, apesar dos *overheads* acrescentados.

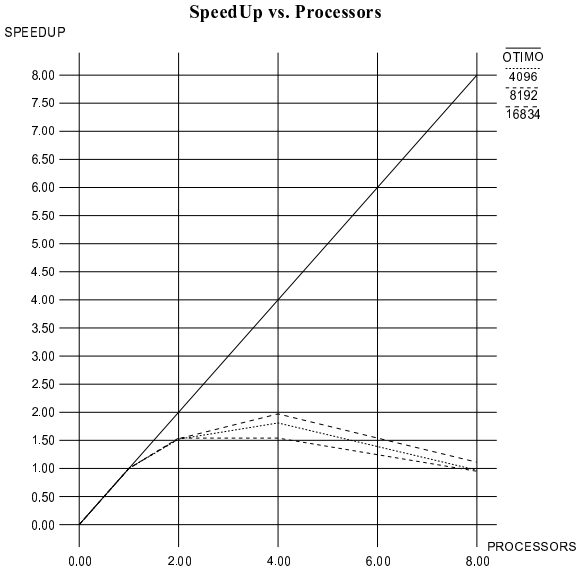

Figura 2: SpeedUp X Processsors. Resultados obtidos comparando-se hipercubos de dimensões 1, 2 e 3 com hipercubo de dimensão 0 (seqüencial não puro)

Referências

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for now (network of workstations). *IEEE Micro*, 1995.

[2] B. Bollobás. *Graph Theory*. Springer - Verlag, 1979.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. Pvm site. http://www.epm.ornl.gov/pvm/.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[5] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[6] M. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.