Avaliando o Bitonic Merge Sort utilizando PVM

RODRIGO MARINHO PASSOS[1]
HESLI DE ARAUJO CARVALHO[1]
JONES OLIVEIRA DE ALBUQUERQUE[1]

[1]UFLA – Universidade Federal de Lavras
DCC – Departamento de Ciência da Computação
Cx. Postal 37 – CEP 37.200-000 Lavras (MG)
{passos,araujo,joa }@comp.ufla.br

Resumo: O Bitonic Merge é um método de ordenação. Neste artigo implementou-se e avaliou-se o desempenho de uma implementação em paralelo do algoritmo de ordenação Bitonic Merge paralelo. Assim, implementou-se o algoritmo no modelo hipercubo para uma *NOW (*NOW – Network of Workstations), com os objetivos de testar e avaliar o software *PVM* (PVM - Parallel Virtual Machine) para fins didáticos e avaliar o desempenho deste método de ordenação.

Palavras Chave: ordenação, Bitonic Merge, programação paralela, PVM, hipercubo.

1 Introdução

O objetivo da ordenação é facilitar a localização dos membros de um conjunto de dados. Assim sendo, é uma atividade fundamental e universalmente utilizada para a elaboração de algoritmos mais complexos [quinn, 1994]. Exemplos de casos em que os objetos são ordenados podem ser encontrados em listas telefônicas, impostos de renda, índices, bibliotecas, dicionários, etc.

A ordenação é uma das atividades mais comuns realizadas nos computadores seriais. Muitos algoritmos incorporam a ordenação para que a informação possa ser acessada eficientemente. A técnica de ordenação ilustra claramente como um dado problema pode ser resolvido por meio de diferentes algoritmos. Diversas literaturas facilmente encontradas nas livrarias apresentam as vantagens e desvantagens, e quais devem ser consideradas em cada aplicação.

A ordenação tem uma importância adicional para os desenvolvedores de algoritmos paralelos. Ela é freqüentemente utilizada para realizar permutações de dados em computadores de memória distribuída. Estas operações de movimentos de dados podem ser usadas para resolver problemas de teoria dos grafos, computação geométrica e processamento de imagem em tempo próximo ao ótimo [quinn, 1994].

Muitos algoritmos de ordenação paralela foram propostos para *Processors Arrays*, multiprocessadores e multicomputadores [patterson, 1996]. O algoritmo de ordenação apresentado neste artigo é o Bitonic Merge, que possui como principais características a ordenação interna, isto é, o algoritmo ordena tabelas de dados que caibam inteiramente na memória primária, e a comparação de pares de elementos.

2 Bitonic Merge Sort

Este algoritmo é a base para algoritmos de tempo de ordenação polilogarítmico de complexidade $O(\log^2 n)$ [batcher, 1968]. Sua operação fundamental é o *compare-exchange*, onde dois números são comparados e se necessário eles são trocados para obter a ordem correta.

O algoritmo ordena uma seqüência bitônica, fato que deu origem ao nome Bitonic Merge. Deve-se pensar que nem sempre os dados a serem ordenados estão nesta forma. Assim o primeiro passo é transformar uma seqüência qualquer em uma seqüência bitônica, e o próximo passo é ordenar esta seqüência.

Definição: Uma seqüência bitônica é uma seqüência de valores $a_0, \ldots, a_{n-1}$, com a propriedade de que (1) existe um índice i, onde $0 \leq i \leq n-1$, tal que a_0 até a_i é monotonicamente crescente e a_i até a_{n-1} é monotonicamente

decrescente, ou (2) existe um deslocamento cíclico de índices até que a primeira condição seja satisfeita [quinn, 1994].

Informalmente uma seqüência bitônica, é uma seqüência que contêm no máximo um "pico" e um "vale". De acordo com a Figura 1, as três primeiras seqüências são bitônicas e a última não.

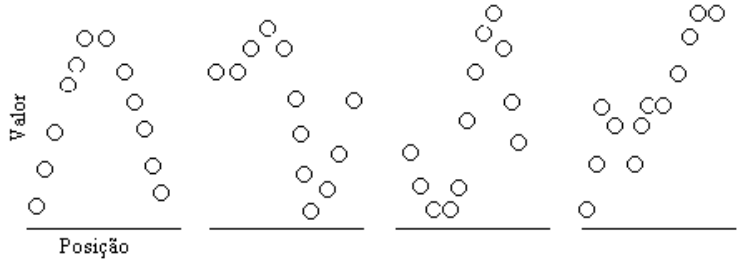

Figura 1 : Seqüência Bitônica [quinn, 1994]**. Na figura, os 3 primeiros gráficos representam uma seqüência bitônica e o último não.**

Uma seqüência bitônica pode ser dividida em duas seqüências bitônicas através de uma simples operação de *compare-exchange*.

Lema: Se *n* é par, então *n/2* comparadores são suficientes para transformar uma seqüência bitônica de *n* valores, $a_o,a_1,a_2,...,a_{n-2}, a_{n-1}$ em duas seqüências bitônicas de *n/2* valores,

$\min (a_0,a_{n/2}), \min (a_1,a_{n/2+1}),...,\min (a_{n/2-1},a_{n-1})$

e

$\max (a_0,a_{n/2}), \max (a_1,a_{n/2+1}),...,\max (a_{n/2-1},a_{n-1})$

tal que nenhum valor da primeira seqüência é maior do que qualquer valor da segunda seqüência [quinn, 1994].

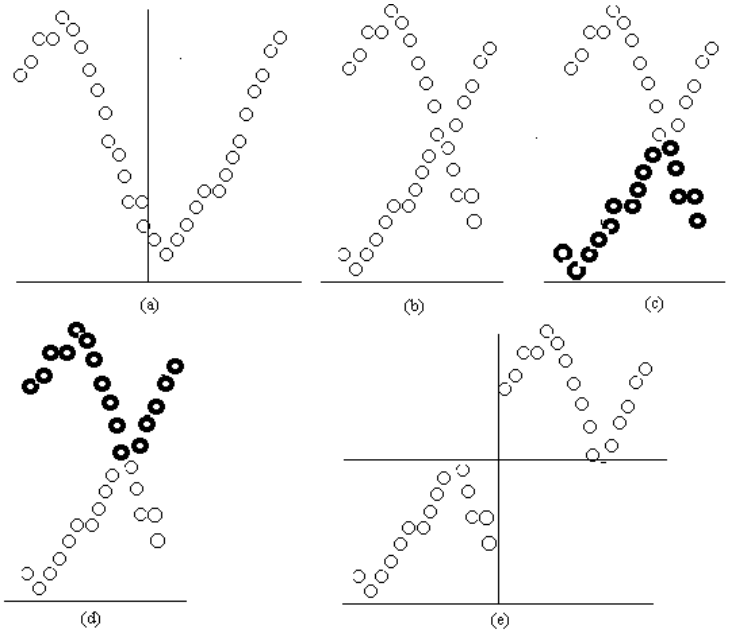

Figura 2: Divisão de uma Seqüência Bitônica [quinn, 1994]

A Figura 2(a) indica a seqüência bitônica inicial. A Figura 2(b) indica a primeira metade da seqüência inicial sobreposta à segunda metade da seqüência inicial. A Figura 2(c) enfatiza a metade da seqüência bitônica com os valores mínimos e a Figura 2(d) a metade com os valores máximos. A Figura 2(e) mostra a seqüência bitônica transformada em 2 seqüências bitônicas distintas.

Dada uma seqüência bitônica, um simples passo de *compare-exchange* divide a seqüência em duas seqüências bitônicas com a metade de seu tamanho. Aplicando este passo recursivamente se produz uma seqüência ordenada. Em outras palavras, dada uma seqüência bitônica de tamanho $n = 2^k$, onde $k > 0$, então k passos de *compare-exchange* são suficientes para produzir uma seqüência ordenada. A Figura 3 mostra um exemplo em que sete passos de *compare-exchange* transformam uma seqüência bitônica de tamanho 128 em uma seqüência ordenada de tamanho 128.

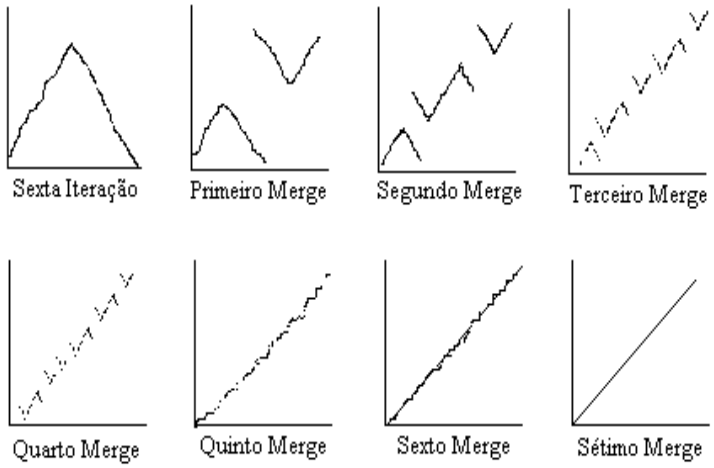

Figura 3: Merge [quinn, 1994]

Teorema: Uma lista de $n = 2^k$ elementos desordenados pode ser ordenada utilizando-se uma rede de $2^{k-2}(k+1)$ comparadores em tempo $O(\log^2 n)$. [batcher, 1968]

Prova: Bitonic Merge recebe uma seqüência bitônica e a transforma em uma lista ordenada que pode ser entendida como metade de uma seqüência bitônica de duas vezes o tamanho. Se uma seqüência bitônica de tamanho 2^m é ordenada em ordem ascendente, enquanto uma seqüência adjacente de tamanho 2^m é ordenada em ordem descendente, então depois de m passos de *compare-exchange* a seqüência combinada de tamanho 2^{m+1} é uma seqüência bitônica. Uma lista de elementos para serem ordenados pode ser vista como um conjunto de n seqüências desordenadas de tamanho 1 ou como $n/2$ seqüências bitônicas de tamanho 2. Portanto, pode-se ordenar qualquer seqüência de elementos fundindo (merging) sucessivamente maiores e maiores seqüências bitônicas. Dado $n = 2^K$ elementos desordenados, uma rede com $k(k+1)/2$ níveis é suficiente. Cada nível contém $n/2 = 2^{k-1}$ comparadores. Então o número total de comparadores é $2^{k-2}k(k+1)$. A execução paralela de cada nível requer tempo constante. Note que $k(k+1)/2 = \log n(\log n+1)/2$. Então o algoritmo possui complexidade $O(\log^2 n)$.

A Figura 4 ilustra como uma série de *bitonic merges* ordena uma lista. Cada gráfico representa uma lista em algum estágio da ordenação. Elementos desordenados formam uma "nuvem" (gráfico inicial). Os elementos ordenados formam uma linha diagonal (gráfico final). Os gráficos intermediários mostram a forma da lista depois de cada uma das $\log n$ iterações. Neste caso $n = 128$ e $\log n = 7$. A Figura 3 ilustra o que acontece na sétima e final iteração. O conjunto inteiro de n elementos foi transformado em uma simples seqüência bitônica, e $\log n$ *bitonic merges* de menores e menores seqüências bitônicas são suficientes para completar a ordenação.

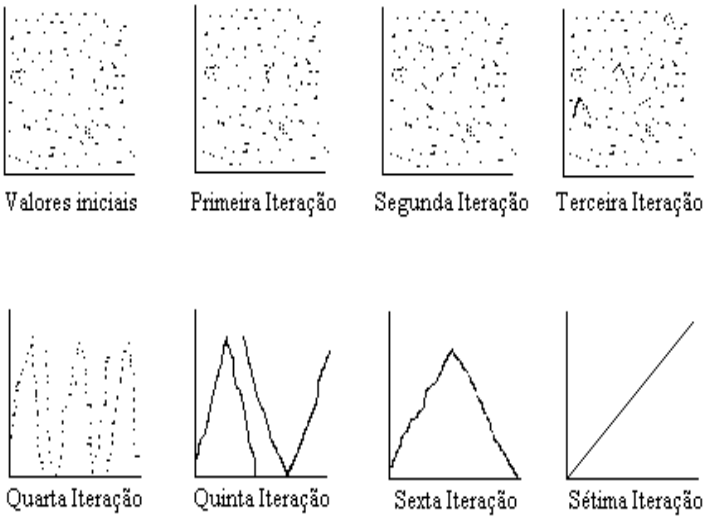

Figura 4: Ordenação [quinn, 1994]

O algoritmo do método de ordenação Bitonic Merge paralelo, possui 3 implementações propostas em [quinn, 1994]; Bitonic Merge na Rede *Shuffle-Exchang;* Bitonic Merge na Rede *Mesh Two-Dimensional;* e Bitonic Merge na Rede *Hypercube*. Nas seções seguintes, concentraram esforços na implementação do Bitonic Merge na Rede Hipercubo.

3 Sistema PVM

O software PVM (Parallel Virtual Machine) fornece uma estrutura uniforme na qual programas paralelos podem ser desenvolvidos de uma maneira eficiente e direta usando o hardware existente. O PVM permite que uma coleção de sistemas de computadores heterogêneos possa ser vista como uma máquina virtual paralela simples. O PVM, transparentemente, trata todas as rotinas de mensagem, conversão de dados e programação de tarefas em uma rede de arquiteturas de computador incompatíveis.

O modelo de computação PVM é simples, muito geral e acomoda uma grande variedade de estruturas de programas aplicáveis. A interface de programação é deliberadamente direta, permitindo que estruturas de programas simples possam ser implementadas de maneira intuitiva. O usuário escreve sua aplicação como uma coleção de tarefas. As tarefas acessam os recursos do PVM através de uma biblioteca de rotinas de interface padronizada. Essas rotinas permitem a inicialização e finalização de tarefas na rede bem como a comunicação e sincronização entre as tarefas. As primitivas de envio de mensagem no PVM são orientadas para operação heterogênea, envolvendo estruturas para *buffering* e transmissão. Estruturas para comunicação incluem estruturas de envio e recepção de dados bem como primitivas de alto nível tais como *broadcast*, sincronização de barreira, e soma global [pvm, 1997].

Tarefas PVM podem possuir controle arbitrário e estruturas dependentes. Em outras palavras, em qualquer ponto da execução de uma aplicação, qualquer tarefa existente pode iniciar ou parar outras tarefas ou adicionar ou deletar computadores da máquina virtual. Qualquer processo pode comunicar e/ou sincronizar com qualquer outro. Qualquer controle específico ou estrutura dependente pode ser implementado no sistema PVM através do uso apropriado de estruturas PVM e de uma linguagem de controle de fluxos.

Devido a estas vantagens e disponibilidade do PVM na máquina utilizada para testes, implementou-se o Bitonic Merge utilizando-se a máquina paralela virtual criada pelo PVM.

4 Implementação: Bitonic Merge na Rede Hipercúbica

O algoritmo proposto para o método de ordenação Bitonic Merge em paralelo, sempre compara elementos nos quais os índices diferem em exatamente um bit. Uma vez que os processadores em um modelo hipercúbico são conectados como se seus índices fossem diferentes de apenas 1 bit, torna-se fácil implementar o Bitonic Merge neste modelo. Os processadores substituem os comparadores. Ao invés de comparar pares de elementos em comparadores, os próprios processadores comparam os dados nos processadores adjacentes.

Assumindo-se n elementos a serem ordenados, onde $n=2^m$ para algum inteiro positivo m, obtemos o seguinte algoritmo:

```
1 BITONIC MERGE SORT:
2 Global   d      {Distância entre elementos que estão sendo     comparados}
3 Local    a      {Um dos elementos a ser ordenado}
4          t      {Elemento obtido do processador adjacente}
5 begin
6    for i ← 0 to m –1 do
7       for j ← i downto 0 do
8          d ← 2^j
9          for all P_k where 0 ≤ k ≤ 2^m-1 do
10            if k mod 2d < d then
11               t ← [k + d]a
12               if k mod 2^(i+2) < 2^(i + 1) then
13                  [k + d]a ← max(t,a)
14                   a ← min(t,a)
15               else
16                  [k + d]a ← min(t,a)
17                   a ← max (t,a)
18               endif
19            endif
20         endfor
21      endfor
22   endfor
23 end
```

Como mencionado anteriormente, o algoritmo do Bitonic Merge garante que em *log n* passos, sendo n o número de elementos a serem ordenados, a ordenação será realizada. No algoritmo acima descrito, representa-se por *m* a condição de parada, ou seja, o número de iterações (linha 6). Assim realiza-se mais um laço (linha 7), para que as operações que são feitas em todos os processadores, sejam feitas para todas as distâncias possíveis no hipercubo, a partir da distância 1 (linha 8).

Assim as *(m-1)* iterações realizadas transformam qualquer seqüência de elementos em uma seqüência bitônica, através de operações *compare-exchange*. Somente na última iteração, onde *j* recebe o maior valor de *i*, ou seja*, (m - 1)*, o vetor será ordenado, pois todas as distâncias no hipercubo são consideradas, necessitando de mais *log n* passos. Pela definição do algoritmo, a seqüência bitônica formada pelas *(m - 1)* iterações iniciais, será dividida em seqüências bitônicas cada vez menores em *log n* passos.

A condição da linha 10 define quais processadores poderão realizar uma operação de *compare-exchange,* onde os elementos são ordenados na ordem certa, justificando-se a condição da linha 12.

Na implementação, as linhas 5 a 8 são tarefas do mestre e as linhas 9 a 18 tarefas de todos os escravos, ou seja, de todos os processadores.

5 Análise de Complexidade

Pela análise das linhas 10 a 18 pode-se perceber claramente que as tarefas realizadas pelos escravos (processadores) possuem complexidade de *O(1).* Da mesma maneira, pode-se analisar a complexidade do mestre.

Como pela definição deve-se realizar sempre *log n* iterações para transformar qualquer seqüência em uma seqüência bitônica, e na última iteração de *i* (linha 6), mais *log n* operações são realizadas para ordenar o vetor dividindo-o em seqüências bitônicas cada vez menores, obtém-se uma complexidade $O(\log^2 n)$.

Em termos mais formais, como o número de passos do algoritmo é em função de *m*, obtém-se a complexidade $O(m^2)$ e, como $n = 2^m$, verifica-se que a complexidade do algoritmo Bitonic Merge é $O(\log^2 n)$.

No entanto, deve-se notar que como cada processador realiza as mesmas operações de complexidade *O(1)* (linhas 10 a 18), e pode-se ter *n* processadores, tem-se então um custo *O(n)*, para as tarefas dos escravos (processadores). Considerando todo o algoritmo (mestre e escravo), o custo total é obtido pela função $O(n\log^2 n)$.

6 Análise de Dados

Para testar-se a complexidade teórica obtida na análise de complexidade do algoritmo Bitonic Merge, $O(n\log^2 n)$, alguns testes foram realizados após implementar o algoritmo, utilizando a máquina paralela virtual criada pelo PVM. Deve-se ressaltar que todos os testes foram feitos em uma única máquina, e assim sendo, não foi possível obter a *parallelizability* [quinn, 1994] deste algoritmo. A Figura 5 ilustra os testes realizados e os resultados obtidos.

Os testes foram realizados dobrando-se o número de processadores a cada amostragem, e mediu-se o tempo necessário para se ativar os processadores (*spawn*), juntamente com o tempo necessário para os processadores executarem a ordenação. Obteve-se assim, a curva de complexidade prática. A curva de complexidade teórica foi obtida pela função $O(n\log^2 n)$.

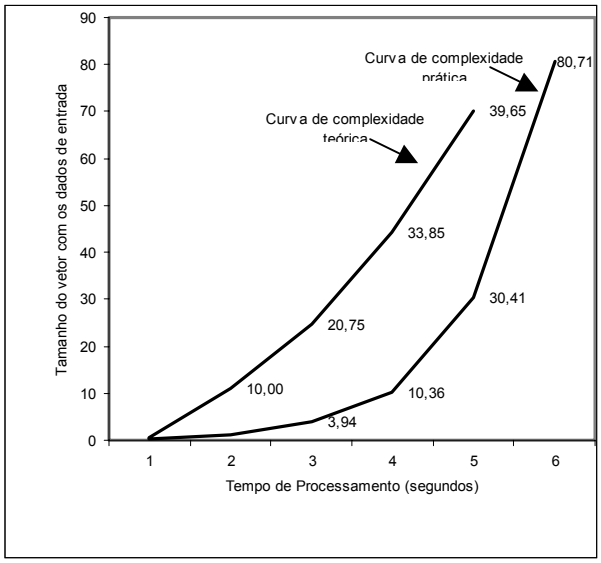

Figura 5: Curvas de Complexidade

Nota-se que a curva de complexidade obtida na prática, parece crescer exponencialmente, contrariando a complexidade teórica $O(n\log^2 n)$. Deve-se considerar, no entanto, que uma máquina SIMD foi utilizada para a realização dos testes e o algoritmo foi proposto para um *processor array* no modelo hipercubo. Outro fator que pode ter contribuído para este fato, é o número considerável de comparações (alto custo computacional) efetuadas no algoritmo, e implementada na máquina PVM através da troca de mensagens. Como não se conhece a fundo o processo de troca de mensagens utilizado pelo PVM, não é possível afirmar, também, o quanto de tempo é acrescido durante esse processo. Mas pode-se afirmar que o PVM parece funcionar adequadamente para fins didáticos.

No entanto, os testes realizados comprovaram que, o custo total deste algoritmo em paralelo é maior do que o custo total do melhor seqüencial de ordenação, fato pelo qual este algoritmo de ordenação não é utilizado, como base para algoritmos de programação paralela.

7 Conclusão

Desenvolver algoritmos paralelos é uma tarefa fácil quando um algoritmo PRAM de custo ótimo existe e as iterações entre os processadores estejam de acordo com a arquitetura utilizada. No algoritmo Bitonic Merge Sort, aqui estudado e implementado, verificou-se um custo de $O(n\log^2 n)$, o qual é maior que o custo do melhor algoritmo

seqüencial de ordenação. Além disso, é necessário um número de processadores igual à quantidade de elementos de entrada, o que parece ser impraticável, devido ao custo elevado de uma arquitetura própria para esse algoritmo.

Assim, conclui-se que o algoritmo de ordenação paralela Bitonic Merge não é aplicável para ordenação em paralelo, já que seu custo é alto em relação a um algoritmo seqüencial, pelo fato de que o algoritmo precise inicialmente montar uma seqüência bitônica, para depois ordená-la. No entanto, deve-se considerar que o algoritmo foi desenvolvido para o modelo hipercubo do *processor array*, e os testes foram realizados em uma máquina seqüencial SIMD, com o auxílio de uma máquina paralela criada pelos recursos do PVM. Desta maneira, propõe-se para trabalhos futuros, a implementação dos outros algoritmos descritos para este método de ordenação [quinn, 1994] a fim de comparar as complexidades de cada modelo na prática.

Portanto, ao desenvolver um algoritmo paralelo de ordenação de custo ótimo, deve-se escolher o melhor algoritmo de ordenação seqüencial existente. Fato que leva o quicksort a ser a base para os algoritmos paralelos de ordenação.

Quanto ao sistema PVM, observou-se que a linguagem do sistema PVM é bastante similar à linguagem de descrição de algoritmos do modelo teórico PRAM, o que faz com que o sistema PVM seja bastante didático para a construção de algoritmos paralelos.

O PVM é um software gratuito, disponível na distribuição RedHat ou pode ser obtido por ftp (ftp://netlib2.cs.utk.edu), WWW (http://www.netlib. org/pvm3/index.html), xnetlib (netlib@ornl.gov) ou e-mail (netlib@ornl.gov). Além disso, este software possui uma vasta documentação disponível em postscript, incluindo guia do usuário, manual de referência e o livro de referência deste artigo [pvm, 1997]. Esta documentação está disponível para ftp no endereço já mencionado.

Para o uso da comunidade acadêmica brasileira, o PVM já se encontra instalado na rede do CENAPAD-MG (http://www.cenapad.ufmg.br). Dessa forma pode-se testar e avaliar a performance de algoritmos PRAM na rede do CENAPAD.

Portanto, devido às vantagens e limitações práticas do sistema PVM, conclui-se que o PVM é um sistema apropriado para implementação de algoritmos PRAM para fins didáticos.

8 Referências

[batcher, 1968] Batcher, K. E. Sorting Networks and their applications. In proceedings of the AFIPS Spring Joint Computer Conference, vol. 32, AFIPS Press, Reston, Va, pp. 307 – 314, 1968.

[quinn, 1994] Quinn, Michael J. - Parallel Computing: Theory and Practice. McGRAW-HILL, 1994.

[patterson, 1996] David A. Patterson and John L. Hennessy – Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 2nd Edition. 1996.

[pvm, 1997] Geist. Al. et al. - Parallel Virtual Machine: A user's guide and tutorial for networked parallel computing. The MIT Pres (Massachusetts Institute of Technology), 1997.