Carla Rodrigues Figueiredo Lara

Um Estudo sobre a Construção Paralela de Árvores de Busca Binária Multidimensional

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtênção do título de Bacharel em Ciência da Computação.

Orientador
Prof. Jones Oliveira de Albuquerque

Lavras
Minas Gerais - Brasil
2001

Carla Rodrigues Figueiredo Lara

Um Estudo sobre a Construção Paralela de Árvores de Busca Binária Multidimensional

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtênção do título de Bacharel em Ciência da Computação.

Aprovada em *28 de junho de 2001*

Prof. Joaquim Quinteiro Uchôa

Prof. José Monserrat Neto

Prof. Jones Oliveira de Albuquerque
(Orientador)

Lavras
Minas Gerais - Brasil

Agradecimentos

Em primeiro lugar agradeço à Universidade Federal de Lavras que através da criação do Curso de Bacharelado em Ciência da Computação deu-me a oportunidade de estudar em curso de excelente qualidade. Neste agradecimento incluo todas as pessoas que trabalharam para que sua criação acontecesse. Um outro especial agradecimento quero destinar a todos os docentes do Departamento de Ciência da Computação da UFLA que, com raríssimas exceções, sempre procuram compreender e ensinar os seus alunos da melhor forma possível. E, por fim, agradecer aos colegas de curso, amigos com os quais vivenciei um momento muito importante e único de minha vida.

Resumo

Árvore de busca binária multidimensional (abreviada por árvore $k-d$) é uma estrutura de dados usada para a organização e manipulação de dados espaciais. Esta estrutura de dados é usada em muitas aplicações, sendo que as principais são: particionamento de grafos, aplicações hierárquicas tais como dinâmica molecular e simulações $n-body$ (agrupamento de objetos fisicamente próximos), banco de dados, computação geométrica, entre muitas outras. Este trabalho estuda formas eficientes de construir tal estrutura de dados. São apresentados vários métodos e em quais situações cada um deles melhor se aplica.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A árvore $k-d$, ou árvore de busca binária k-dimensional, é um tipo de estrutura de dados que permite um processamento eficiente de chaves multidimensionais e foi proposta por Bentley em 1975 [DJzC00], e desde então vem sendo testada e sofrendo vários tipos de modificações, na maioria das vezes com o objetivo de obter um melhor desempenho [Pro97]. A árvore $k-d$ é basicamente um tipo de Árvore Binária de Pesquisa (BST) que divide os pontos em parcelas menores mais viáveis (*buckets*). Minimizando, assim, o número dos pontos que necessitam ser procurados.

A árvore $k-d$ difere da BST pelo fato de que cada nível da árvore $k-d$ se ramifica baseada numa pesquisa de chave para o nível, chamado discriminador. Definimos o discriminador no nível i como sendo $i \ mod \ k$ para k dimensões. Por exemplo, podemos armazenar dados de coordenadas (x, y). Neste caso, k é 2 (há duas coordenadas), com a coordenada x definida arbitrariamente como chave 0, e a coordenada y como chave 1. Para cada nível, o discriminador alterna entre x e y. Portanto, um nodo N do nível 0 (raiz) teria em sua sub-árvore esquerda apenas nodos cujos valores de x são menores que N_x. Como, por exemplo, na Figura 1.1 onde o nó B é inserido à esquerda do nó A, pois o valor da coordenada x de B vale 15 e é menor que a coordenada de A que tem um valor 40. Um nodo M ao nível 1 teria em sua sub-árvore à esquerda nodos cujos valores de y são menores que M_y. Como o nó G, por exemplo. Não há restrição quanto aos valores dos pais de M_x e aos valores de x descendentes de M, desde que a decisão de ramificação realizada no nodo M seja baseada somente na coordenada y.

Um exemplo de uma árvore $k-d$ para duas dimensões ($k = 2$) pode ser encontrado na Figura 1.1.

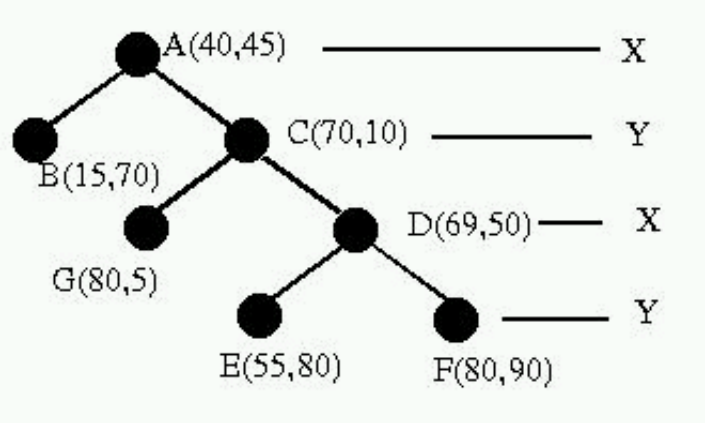

Figura 1.1: Exemplo de uma árvore $k-d$ (onde $k=2$)

De uma maneira mais formal podemos entender o funcionamento desta estrutura de dados da seguinte forma: considerando a existência de um conjunto de n pontos em um espaço k dimensional, tem-se que $d_1, d_2, ..., d_k$ denotam k dimensões. A árvore $k-d$ é construída da seguinte forma [AFAG00b]: o nó raiz da árvore corresponde ao conjunto de todos os pontos, é escolhida então uma dimensão d_l, e calcula-se a mediana de todos os pontos ao longo desta dimensão d_l. O próximo passo é dividir os pontos em dois conjuntos de tamanho aproximadamente igual - um conjunto conterá todos os pontos que possuem sua coordenada ao longo de d_l menor ou igual à mediana calculada e o segundo conjunto conterá todos os pontos restantes. As duas sub-partições são representadas pelos nós filhos da raiz. A árvore é construída recursivamente até que cada nó folha corresponda a um único ponto. Típicas estratégias usadas para escolher a dimensão a ser dividida são: escolher a dimensão $d_{(imodk)+1}$ para cada nó no nível i da árvore (a raiz é considerada nível 0), escolhendo a dimensão com maior comprimento, sendo que este valor é obtido pela diferença entre a maior e a menor coordenada dos pontos ao longo da dimensão.

Muitas aplicações requerem apenas a construção parcial de árvores $k-d$. Em particionamento paralelo de grafos, por exemplo, o objetivo é criar p partições e distribuir o grafo para p processadores, requerendo a construção de somente os primeiros $log\ p$ níveis da árvore. Em aplicações hierárquicas como simulação $n-body$, que será apresentada posteriormente, agrupamentos de objetos fisicamente próximos são essenciais e a árvore $k-d$ oferece tal esquema de agrupamento.

Em bancos de dados, registros podem ser tratados como sendo pontos em um espaço apropriado, mapeando cada chave para uma coordenada, resultando em um conjunto de pontos que podem ser organizados usando uma árvore $k-d$.

Este trabalho enfoca duas abordagens padrões utilizadas na construção das árvores $k-d$. A primeira usa o cálculo explícito da mediana, e a segunda usa ordenação como um passo de pré-processamento com o objetivo de eliminar o cálculo da mediana. Em adição, será apresentada uma nova abordagem proposta em [AFAG00b], onde é realizada uma ordenação parcial dos dados e refinamentos são realizados se necessário.

Esta monografia está organizada da seguinte forma: no Capítulo 2 são apresentados os principais objetivos do trabalho e as referências bibliográficas, encontradas sobre o assunto em questão. Também neste capítulo, encontra-se uma introdução aos principais conceitos de programação paralela. No Capítulo 3 são apresentados os diversos métodos para a construção da árvore $k-d$, tanto nos primeiros $log\ p$ níveis quanto para o restante dos níveis locais. Por fim, no Capítulo 4, encontram-se algumas aplicações e também as conclusões.

Capítulo 2

Objetivos do Trabalho

Os objetivos do trabalho são inúmeros, sendo que os mais importantes e que devem ser destacados são: estudar a estrutura de dados denominada árvore $k-d$, realizar um levantamento dos algoritmos paralelos para construção da mesma, estudar as características próprias de cada um deles, bem como apontar as principais diferenças entre estes algoritmos. E numa etapa posterior, encontrar aplicações nas quais esta estrutura de dados pode ser usada na tentativa de obter um bom desempenho.

2.1 Estado da Arte

Nesta seção serão apresentados os principais trabalhos publicados que são relacionados com o assunto desta monografia.

Em [Goi96] é abordada a paralelização de um grande número de aplicações hierárquicas, sendo que em uma de suas seções é apresentada a construção de árvores $k-d$ e assuntos relacionados. Neste documento são apresentados alguns algoritmos em altíssimo nível (escritos em linguagem natural). Já em [AFAG00b], o mesmo grupo do trabalho citado acima, apresenta um artigo sobre a construção de árvores $k-d$ com um menor grau de detalhamento, onde os algoritmos são apenas discutidos e nenhum deles é apresentado. Ambos apresentam seus resultados experimentais. Ainda deste mesmo grupo, existe um outro trabalho [AFAG00a], que apesar de possuir um outro título, tem grande parte de seu conteúdo exatamente igual ao de [AFAG00b].

Em [BM00], encontram-se várias soluções paralelas para problemas geométricos utilizando um modelo chamado Scan (diferente do RAM e do PRAM). Um dos problemas abordados é o das árvores $k-d$, mas de forma muito simplificada.

Em [Pro97], a árvore $k-d$ é apresentada juntamente com outras diversas estruturas de dados que podem ser utilizadas para construção de um sistema de banco de dados onde os registros são tratados como pontos num espaço multidimensional.

2.2 Algoritmos Paralelos

Muitos problemas que até pouco tempo pareciam ter solução impossível, principalmente pelos elevados cálculos envolvidos em sua resolução, estão hoje sendo resolvidos através do uso de processamento paralelo.

Processamento paralelo é o processamento de uma informação (ou dado) que enfatiza a manipulação concorrente de dados pertencentes a um ou mais processadores resolvendo um único problema. A comunicação dos processadores é realizada através de redes especiais de conexão, ou por meio de uma memória compartilhada [Qui94].

Projetar algoritmos paralelos, analisá-los, e provar que eles estão corretos é muito mais difícil que aplicar os mesmos passos para algoritmos seqüenciais, segundo [Man89].

As principais medidas de complexidade para algoritmos seqüenciais são tempo de execução e espaço utilizado. Estas medidas também são importantes em algoritmos paralelos, mas outros fatores também são importantes, sendo que um dos principais é o número de processadores.

Deve-se destacar que apesar da programação paralela ter trazido um grande avanço na solução de problemas complexos, existem problemas que são inerentemente seqüenciais. E que portanto não apresentam nenhum ganho de desempenho quando paralelizados, podendo em algumas vezes até ter um desempenho pior.

Resultados sobre o quanto um algoritmo paralelo é eficiente podem ser obtidos através de um valor conhecido como sppedup. Este valor é obtido através da divisão entre o tempo necessário para a execução do algoritmo seqüencial considerado mais eficiente pelo tempo para realizar a mesma computação em um dispositivo utilizando paralelismo.

2.2.1 Modelo de Computação Paralela

Para analisar algoritmos paralelos é necessário ter um modelo formal no qual possam ser baseados os custos das operações realizadas. O projeto de algoritmos seqüenciais é tipicamente realizado usando um modelo abstrato de computação,

chamado de random access machine (RAM), [Qui94]. Neste modelo, a máquina consiste de um único processador conectado a um sistema de memória.

Modelar a computação paralela costuma ser mais complicado que modelar computação seqüencial, porque na prática os computadores paralelos tendem a variar mais em sua forma de organização [Ski97, BM00]. Isto porque, de maneira geral todos os computadores seqüenciais são construídos baseados no mesmo modelo de arquitetura, enquanto que os computadores paralelos tendem a variar mais, principalmente quanto a forma que a memória está sendo usada pelos processadores. Como conseqüência existe um grande número de propostas de pesquisas em algoritmos paralelos foi no campo da modelagem, e muitos debates têm surgido sobre o melhor modelo. Não existe um consenso sobre o modelo certo, segundo este mesmo autor.

Modelos paralelos podem ser divididos em duas grandes classes: modelo multiprocessador e modelo *work-depth.*

O modelo multiprocessador é uma generalização do modelo seqüencial RAM, com a diferença de considerar a existência de mais de um processador. O modelo multiprocessador pode ser classificado em três tipos básicos: máquinas de memória local, máquinas de memória modular, e máquinas de acesso randômico paralelo (PRAMs). Uma máquina de memória local consiste de um conjunto de n processadores, cada qual com sua memória local. Estes processadores são conectados em uma rede para comunicação. Uma máquina de memória modular consiste de m módulos de memória e n processadores, todos conectados em uma rede comum. Uma PRAM consiste de um conjunto de n processadores, todos conectados em uma memória compartilhada. A diferenças entre os três está na forma como a memória pode ser acessada.

O modelo PRAM é um dos mais usados, ele facilita o projeto de algoritmos eficientes. Se um algoritmo projetado para um modelo PRAM (ou qualquer outro modelo) puder ser traduzido para um algoritmo que roda eficientemente em um computador real, então o modelo pode ser considerado um modelo de sucesso [BM00].

Capítulo 3

Construção da Árvore

Considera-se a tarefa de construir uma árvore $k-d$ balanceada de N pontos em um espaço k dimensional até um número arbitrário de níveis usando p processadores. Por motivo de simplicidade assume-se que N e p são potências de dois. Também se assume que $N \geq p^2$ e a construção da árvore vai até pelo menos $log\ p$ níveis. Este número $log\ p$ níveis é escolhido por causa da forma de ativação dos processadores num ambiente paralelo, e pelo mesmo motivo prefere-se trabalhar com potências de dois. Os primeiros $log\ p$ níveis da árvore são construídos em paralelo e os níveis restantes são construídos localmente.

A mediana dos N pontos ao longo da dimensão d_1 (uma dimensão qualquer) é encontrada e os pontos são separados em duas partições, conforme já descrito anteriormente. As partições são redistribuídas, tal que a primeira metade de processadores contenha uma partição e o restante dos processadores contenha a outra partição. Isto é repetido recursivamente até que os primeiros $log\ p$ níveis tenham sido construídos. Neste ponto, cada processador contém $n = \frac{N}{p}$ pontos. A árvore local para estes n pontos é construída até o número de níveis desejados. O ideal é que o número de níveis para a árvore local seja $log\ m, (m \leq n)$. Provavelmente, com o objetivo de que a árvore fique balanceada.

Qualquer algoritmo que encontre a mediana pode ser usado para a construção de árvores $k-d$. Por outro lado, considerando que a tarefa envolve repetidos cálculos de medianas, algum pré-processamento pode ser usado para ajudar a aumentar a velocidade de cálculo da mediana. Serão apresentados dois algoritmos que usam tal pré-processamento. O método padrão para construção de uma árvore $k-d$ completa usa pré-ordenação dos pontos ao longo de cada dimensão para eliminar completamente uma busca explícita da mediana. A construção paralela da árvore

pode ser decomposta em duas partes: construção dos primeiros $log\ p$ níveis da árvore em paralelo, seguida pela construção da árvore localmente. A estratégia para construir as duas partes não precisa ser a mesma. Neste trabalho são analisadas três estratégias para cada uma das partes.

3.1 Construção da Árvore Local

3.1.1 Método Sort-Based

Para evitar o *overhead* associado com a busca explícita da mediana para todos os nós internos da árvore $k-d$, utiliza-se uma abordagem que envolve ordenação dos pontos de uma dimensão exatamente uma vez. Neste método são mantidos k vetores $A_1, A_2, ..., A_k$. Inicialmente, A_l contém todos os pontos ordenados de acordo com a dimensão d_l. Qualquer nó da árvore $k-d$, correspondente a uma partição que ainda será divida, tem dois ponteiros i e j ($i < j$) associados a ele, tal que o sub-vetor A_l contém os pontos da partição ordenada ao longo de d_l. Se a partição é dividida baseada em d_1, a divisão do vetor A_1 pode ser realizada em tempo constante. Considerando a divisão A_l para qualquer outra dimensão d_l. Se o sub-vetor A_l simplesmente for percorrido (*scan*) até o fim e realizadas as trocas dos ponteiros quando necessários a ordenação pode ser destruída. No método *median-based*, este problema não ocorre, porque nele a ordem não precisa ser mantida. Então, é necessário revisar os pontos duas vezes: uma vez para contar o número de pontos nas duas sub-partições e a segunda vez para de fato mover os dados. A contagem é necessária porque o número de pontos menores ou iguais à mediana não necessita ser exatamente igual à metade dos pontos.

Este método requer um passo de pré-processamento para ordenar n pontos ao longo de cada uma das k dimensões, isto é realizado com complexidade de tempo $O(kn\ log\ n)$. Construir cada nível da árvore $k-d$ envolve percorrer cada um dos k vetores com complexidade de tempo $O(kn)$. Depois do pré-processamento, $log\ m$ níveis da árvore podem ser construídos em $O(kn\ log\ m)$. Em alguns casos, tal como quando o método sort-based é usado para construir os primeiros $log\ p$ níveis da árvore, os dados podem já estar ordenados.

É possível reduzir a complexiadde de tempo $O(kn)$, por nível, para $O(n)$ usando o seguinte esquema: existem n registros de tamanho k, cada um correspondente a um ponto. Considera-se a existência de um vetor L de tamanho n, onde um elemento de L é um ponteiro para um registro e k índices indicando as posições nos vetores $A_1, A_2, ..., A_k$ que correspondem a este registro. Um elemento do vetor A_l não é mais um ponteiro para um registro, mas armazena o índice de L, que contêm

um ponteiro para o registro. Sendo assim, é preciso mudar a referência de três ponteiros para selecionar o registro atual, apontado por um elemento de qualquer dos vetores A_l. Inicialmente, $L[i]$ contém um ponteiro para o registro i e todos os vetores podem ser montados em complexidade de tempo $O(kn)$. Os vetores $A_1, A_2, ..., A_k$ são ordenados em $O(kn\ log\ kn)$ exatamente como antes.

Supondo que a árvore seja construída até i níveis e o vetor L seja particionado em 2^i sub-vetores, o vetor A_l deve ser organizado em 2^i sub-partições. Rotulam-se as partições de L usando $1...2^i$ da esquerda para a direita. Para cada elemento de L, usando o índice para A_l, o correspondente elemento de A_l é rotulado com o número da partição. Trocando os elementos de A_l tal que todos os elementos com o rótulo menor apareçam antes dos elementos de maior rótulo, a ordenação é preservada. Tudo isto pode ser feito em complexidade de tempo $O(n)$. Como antes, A_l pode ser usado para escolher a mediana de cada sub-partição.

Embora este método reduza a complexidade do tempo de execução, por nível, de $O(kn)$ para $O(n)$, não é esperado melhor desempenho para pequenos valores de k, tal como $k = 2$ e $k = 3$, que cobre muitas das aplicações práticas como particionamento de grafos e métodos hierárquicos.

3.1.2 Método Median-Based

Nesta abordagem, a partição é representada por um conjunto não ordenado de pontos e a mediana é explicitamente calculada para dividir a partição. Os autores em [AFAG00b] afirmam terem testado vários algoritmos para encontrar a mediana e encontraram que o algoritmo de Floyd e Rivest é o que possui melhor desempenho. O algoritmo trabalha escolhendo um elemento do conjunto de maneira aleatória, particionando o conjunto baseado neste elemento, descartando a partição que não possui o elemento desejado. O pior caso de complexidade de tempo de execução é $O(n^2)$, mas a complexidade de tempo esperada é somente $O(n)$. O número esperado de interações é $O(log\ n)$. Uma abordagem diferente pode ser usada para reduzir o número esperado de interações para $O(log\ log\ n)$. Os mesmos autores ainda informam que as versões paralelas destes dois métodos possuem tempos de execução muito parecidos, mas seqüencialmente o algoritmo de Floyd é melhor.

Pode ser observado que o mesmo processo de calcular a mediana de uma partição a divide em duas sub-partições. Na construção do nível i da árvore local, são calculadas 2^i medianas, cada partição contendo $\frac{n}{2^i}$ pontos. Como o número total de pontos em todas as partições de qualquer nível é n, construir cada nível possui complexidade de tempo $O(n)$. Desta forma, um número m de níveis pode ser construído em complexidade de tempo $O(n\ log\ m)$.

3.1.3 Método Bucket-Based

A complexidade seqüencial do método *median-based* é proporcional a $O(n \log m)$. Até mesmo a constante associada à ordenação é pequena se comparada com o custo da busca pela mediana, a ordenação é proporcional a $O(kn \log n)$. A melhoria na constante pode, no entanto, não compensar a complexidade mais alta do método *sorted-based.* Para resolver este problema, pode-se induzir uma ordenação parcial dos dados, que pode sofrer um refinamento mais adiante somente se necessário.

Depois de ordenado, dividem-se os pontos em b unidades menores, chamadas de *buckets*. Depois de definidos os *buckets*, o passo seguinte é encontrar o *bucket* que deve conter cada um dos n pontos. Realiza-se isto utilizando uma busca binária em complexidade de tempo $O(\log b)$ por ponto. Os n pontos são agora distribuídos entre os b *buckets* e o número esperado de pontos em um *bucket* é $O(n/b)$ com alta probabilidade (assumindo que b é $O(\frac{n}{\log n})$. O mesmo procedimento é repetido para induzir uma ordenação parcial ao longo de todas as dimensões. O tempo total para a realização da ordenação parcial ao longo de todas as dimensões é $O(kn \log b)$. Este é o processamento prévio necessário no método *bucket-based.* Este método pode ser visto como uma abordagem híbrida que combina ordenação e busca da mediana. Se $b = 1$, a abordagem híbrida torna-se equivalente ao método que encontra a mediana e se $b = n$, torna-se equivalente ao método que ordenada os dados completamente.

Em qualquer fase do algoritmo, tem-se a partição e k vetores, que armazenam os pontos da partição parcialmente ordenada em *buckets* usando suas coordenadas ao longo de cada dimensão. Sem perda da generalização, assume-se que a partição deve ser dividida ao longo de d_1. O *bucket* que contém a mediana é facilmente identificado em tempo logarítmico, de acordo com o número de *buckets.* Encontrar a mediana significa encontrar o elemento no *bucket* que contém a mediana. Isto é calculado usando o algoritmo de seleção em um tempo proporcional ao número de pontos no *bucket*. Para dividir a partição, é necessário calcular os vetores parcialmente ordenados correspondentes às sub-partições. Isto é realizado ao longo de d_1 através da divisão do *bucket*, que contém a mediana, em dois *buckets*. Para criar os vetores ao longo de qualquer outra dimensão d_l, cada *bucket* no vetor parcialmente ordenado só precisa ser dividido em dois *buckets*. Todos os *buckets* com pontos que possuem uma coordenada ao longo de d_1, menor que a mediana, são agrupados em uma sub-partição e o restante dos *buckets* são agrupados em uma segunda partição.

Quando uma partição é dividida em duas sub-partições, o número de pontos na partição é divido ao meio. O número de *buckets* que sobram é aproximadamente

o mesmo (exceto pelo fato de um *bucket* poder ser dividido em dois) ao longo da dimensão que está sendo usada para a divisão da partição. Ao longo de todas as outras dimensões, o número de *buckets* é aumentado por um fator dois. Em um estágio quando o nível i da árvore estiver construído, existirão 2^i partições e $\Theta(b2^{\frac{i(k-1)}{k}})$ *buckets*. Assim, construir o nível i da árvore requer resolver 2^i problemas de encontrar a mediana. Este tempo é dominado pelo tempo $O(kn)$ necessário para dividir os *buckets* ao longo das $k-1$ dimensões. Desde que a constante associada com a busca da mediana seja alta, este método tem a vantagem de realizar a busca pela mediana em dados de menor tamanho. A complexidade de execução para a construção de $log\ m$ níveis é $O(kn\ log\ m)$.

Estratégias similares à que foi apresentada para a ordenação podem ser usadas para reduzir a complexidade de tempo para $O(n\ log\ m)$. Contudo, estas estratégias não costumam melhorar o desempenho para pequenos valores de k, tal como $k = 2$ e $k = 3$, devido ao alto *overhead.*

3.1.4 Resultados Experimentais da Literatura

Em [AFAG00b] são descritos resultados obtidos após vários testes realizados, utilizando os três métodos anteriormente expostos. Os tempos gastos por cada um deles são apresentado na Tabela **??**, podendo ser decomposto em diferentes partes: o tempo para o pré-processamento X (custo de ordenação), o custo para encontrar a mediana em todos os níveis (c), e o tempo para movimentação dos dados com a finalidade de decompor os vetores em sub-vetores baseando-se na mediana (s). Os métodos *bucket-based* e *sort-based* necessitam da propriedade de *ordem estável*. Isto significa que a ordem relativa dos dados tem que ser preservada durante a movimentação dos dados, resultando em um maior valor de (s). Os dados precisam ser movidos para $k-1$ listas no *sort-based* e na estratégia *bucket-based,* enquanto que no *median-based* apenas para uma única lista. Existe um *overhead r* associado a manutenção e processamento das sub-listas, que dobram a cada nível. Este *overhead* é menor para ordenação, ligeiramente maior para o *median-based,* e mais alto para o *bucket-based,* onde o custo para os *buckets* cresce exponencialmente com o aumento do número de níveis.

Um importante aspecto prático do cálculo da mediana é que o passo de movimentação de dados e o cálculo da mediana podem ser combinados, resultando em uma constante global menor. Os experimentos práticos se limitaram a duas dimensões porque os resultados e conclusões obtidos podem ser estendidos para dimensões mais altas [AFAG00b].

Uma comparação entre os métodos *sort-based* e *bucket-based* mostra que a

Tabela 3.1:
resultados1

	Construção da àrvore para $log\ m$ níveis		
Método	Pré-Processamento (X)	Cálculo Mediana (c)	Movimentação de dados (s)
Median-Based	-	$O(n\ log\ m)$	$O(n\ log\ m)$
Sort-Based	$O(kn\ log\ m)$	$O(m)$	$O(kn\ log\ m)$
Bucket-Based	$O(kn\ log\ b)$	$O\left(\frac{n}{b}\ \frac{m^{\frac{1}{k}}-1}{2^{\frac{1}{k}}-1}\right)$	$O(kn\ log\ m)$

segunda estratégia possui um valor de (X) mais baixo, similares valores de (s), e valores de (c) e *r* muito maiores. Foram experimentados diferentes tamanhos de *buckets*, e foi observado que existe uma troca entre os valores de (c) e *r*. O valor de *r* é diretamente proporcional ao número de *buckets*, enquanto que (c) é inversamente proporcional a este. O efeito do *overhead r* é por lista e aumenta exponencialmente com o aumento do número de níveis. Conclui-se que os valores de *r* e (c) são suficientemente grandes fazendo com que o *sort-based* seja melhor que *bucket-sort*, exceto quando o número de níveis é muito pequeno (menor que quatro). Porém, para estes casos, a estratégia *median-based* é melhor. Na verdade, foi descoberto que a estratégia *median-based* é muito melhor que o *bucket-based* para pouco níveis, até mesmo ignorando o tempo requerido para construção dos *buckets*.

Uma comparação dos métodos *sort-based* e *median-based* mostra que o valor de (X) para este último é zero, o de (s) é menor, o valor de *r* é mais alto, e o de (c) muito maior, como mostrado na Tabela **??** e verificado nos resultados experimentais da literatura. Espera-se que o *median-based* seja melhor para poucos níveis e o *sort-based* para um número alto de níveis, a expectativa é que o custo de pré-processamento seja amortizada mediante os vários níveis.

Foram realizadas comparações com conjuntos de diferentes tamanhos: 8K, 32K, e 128K. Os resultados mostraram que o *median-based* é melhor que o *sort-based*, exceto quando o número de níveis é próximo de $log\ N$. Para muitos níveis, o aumento devido ao alto valor de (c) torna-se significante para o método *median-based*. Estes mesmos resultados também mostraram que quando os dados já estão ordenados, o *sorted-based* tem um menor tempo de execução que o *median-based.*

A estratégia *median-based* é melhor, a menos que o número de níveis seja próximo de *log N* ou se os dados já estiverem disponíveis de forma ordenada, quando então o *sorted-based* é o melhor.

3.2 Construção Paralela da Árvore para os $log\ p$ Níveis

3.2.1 Método Sort-Based

No algoritmo paralelo, cada processador recebe N/p elementos. Os elementos são ordenados usando um algoritmo paralelo de ordenação para criar os vetores ordenados $A_1, A_2, ..., A_k$ distribuindo-os uniformemente entre os processadores. Foi usada uma variação do *parallel sample sort* que se mostrou muito eficiente na prática. O tempo total para a ordenação é

$$O\left(\frac{N}{p}logN\ +\ p\ log^2 p\ +t_s p\ +t_w\left(\frac{N}{p}+p^2\right)\ +t_h\ p\ logp\right)$$

para um hipercubo, segundo [AFAG00b].

Seguindo um modelo onde $t_s+t_h d+t_w m$ significa o tempo de uma mensagem de tamanho m atravessar d pontos de comunicação. Sendo que t_s é o custo de inicialização, t_h é o tempo gasto por link, e t_w é o inverso da banda passante. Segundo [AFAG00b], para grandes valores de N ($N\ \geq\ O(t_s p^2 + t_w p^3)$), o tempo de execução é $O(\frac{N}{p}logN\ +t_w\frac{N}{P})$, em um hipercubo.

Uma vez que os dados são pré-processados por ordenação, o trabalho de calcular a mediana é completamente eliminado. Assume-se que a partição deve ser feita ao longo da dimensão d_1. O processador que contém a mediana de d_1 transmite (*broadcasts*) este valor a todos os processadores. A partição é dividida em duas sub-partições, sendo que uma sub-partição vai para a metade dos processadores e a outra para o restante (outra metade). Para qualquer outra dimensão d_l, cada processador percorre a parte do vetor ordenado por d_l e o divide em dois sub-vetores dependendo da coordenada ao longo de d_l. Todos os pontos menores que a mediana da coordenada d_1 são movidos para primeira metade dos processadores. Pontos maior que a mediana são movidos para a segunda metade de processadores. Dado que os vetores já estão ordenados, dividir as partições de todos os nós da árvore requer apenas a movimentação dos elementos dos vetores para os apropriados processadores.

Considerando a tarefa de construir os primeiros $log\ p$ níveis da árvore tem-se que: no nível i da árvore, tem-se 2^i partições contendo $\frac{N}{2^i}$ pontos cada. Uma partição é representada por k vetores distribuídos uniformemente em $\frac{p}{2^i}$ processadores. Dividir os vetores localmente e preparar os dados para comunicação requer um tempo de $O(\frac{(k-1)kN}{p})$. Isto porque os $k-1$ vetores podem conter diferentes registros e o tamanho de cada registro é $O(k)$.

Segundo [AFAG00b], estando os dados já ordenados, o tempo necessário para construir os $log\ p$ níveis da árvore em um hipercubo é

$$\sum_{i=0}^{log\ p-1} O\left(k(k-1)\frac{N}{p}\ + t_s\frac{p}{2^i}\ + t_w\ k\ (k-1)\frac{N}{p}\ + t_h\frac{p}{2^i}\ log\ \frac{p}{2^i}\right)$$

$$= O\left(k(k-1)\frac{N}{p}\ log\ p\ + t_s p\ + t_w\ k\ (k-1)\ \frac{N}{p}\ log\ p\ + t_h\ p\ log\ p\right).$$

Para grandes valores de N, a complexidade de tempo total requerida pelo algoritmo é $O\left(k\ (k-1)\ t_w\frac{N}{p}\ log\ N\right)$ para o hipercubo, segundo o mesmo autor. Em um método seqüencial *sort-based* é possível reduzir o custo computacional de $O(kn)$ para $O(n)$. Porém, o método trabalha com mudança de referência de ponteiros, com o objetivo de obter os pontos nos outros processadores. A comunicação resultante deste método é impraticável até mesmo para grandes valores de k.

3.2.2 Método Median-Based

Comparado-se algoritmos paralelos determinísticos e randômicos para seleção, verifica-se que o melhor resultado é obtido utilizando dados randômicos. Possuir dados não randômicos não é problema, porque é possível torná-los aleatórios. Uma maneira seria alocar cada ponto para um processador e usar uma primitiva de transporte para mover os pontos para os processadores apropriados. O custo para fazer isto é insignificante se comparado com o custo de construir a árvore $k-d$.

Considere $N^{(j)} = \sum_{i=0}^{p-1}\ N_i^{(j)}$ o número de elementos em um processador P_i no início da iteração j do algoritmo para encontrar a mediana, e que $k^{(j)}$ é o grau de interesse em um elemento. Todos os processadores usam o mesmo gerador de números aleatórios e com a mesma *semente* para produzir números aleatórios idênticos. Considerando o comportamento do algoritmo na iteração j, primeiro uma operação paralela *prefix* é realizada em $N_i^{(j)}s$. Um número aleatório entre 1 e $N^{(j)}$ é usado para escolher a mediana estimada. Depois da operação paralela de *prefix*, cada processador pode determinar se ele possui a mediana estimada, e caso tenha, deve comunicar-se com os outros processadores. Cada processador percorre o conjunto de pontos e o divide em dois sub-conjuntos baseados na mediana estimada. Uma operação *combine* e uma comparação com $k^{(j)}$ determina o qual destes dois sub-conjuntos pode ser descartado e o valor de $k^{(j+1)}$ é usado para a próxima iteração.

Para dados aleatórios, os pontos que sobram à esquerda após cada iteração são mapeados igualmente entre todos os processadores com alta probabilidade, a menos que o número de pontos remanescentes seja muito pequeno. Assim, o tempo total esperado para a computação é de $O(\frac{N}{p})$. Outra opção é assegurar que o balanceamento de carga seja feito após cada iteração. Contudo, balanceamento de carga sempre resulta em um aumento do tempo de execução.

O número de iterações requerido para N pontos é $O(log\ N)$ com alta probabilidade. Então, o tempo esperado de execução da busca paralela pela mediana, é

$$O\left(\frac{N}{p} + (t_s + t_w)\ log\ p\ log\ N\right)$$

para o hipercubo, segundo [AFAG00b].

3.2.3 Método Bucket-Based

Para usar este método, primeiro é preciso criar os *buckets* usando p processadores. Os *buckets* necessários ao longo de d_1 podem ser calculados da seguinte maneira: seleciona-se um total de n^ε pontos ($0 < \varepsilon < 1$) depois eles são ordenados por um algoritmo de ordenação padrão tal como *merge sort*. O tempo para esta ordenação é

$$O\left(\frac{N^\varepsilon\ log\ N^\varepsilon}{p} + \frac{N^\varepsilon}{p}\ log^2\ p + \left(t_s + t_w\frac{N^\varepsilon}{p}\right) log^2\ p\right)$$

para o hipercubo, segundo [AFAG00b].

Usando estes dados ordenados, divide-se os pontos em p intervalos chamados *buckets*. Usando uma operação global de concatenação, os p intervalos são armazenados em cada processador. Cada processador percorre seus $\frac{N}{p}$ pontos e para cada ponto determina o *bucket* ao qual ele pertence em um tempo $O\left(\frac{N}{p}\ log\ p\right)$. Os pontos são então divididos em p listas, uma para cada *bucket*. Deseja-se mover todas as listas pertencentes ao *bucket* i para o processador p_i. Os pontos são enviados para os processadores adequados usando primitivas de transporte. O número esperado de pontos por *bucket* é $O(\frac{N}{p})$, com alta probabilidade.

Considerando mais uma vez a tarefa de construir os primeiros $log\ p$ níveis da árvore, e supondo que a primeira divisão será ao longo da dimensão d_1, é fácil encontrar o *bucket* que contém a mediana, gastando apenas uma operação parale-la *prefix*. A mediana é encontrada através do elemento com o valor adequado no *bucket* usando o algoritmo seqüencial de seleção. Esta mediana é então transmitida a todos os outros processadores os quais dividem seus *buckets* ao longo de

todas as outras dimensões baseando-se nesta mediana. Desde que o tamanho do *bucket* seja menor que o número de elementos em um processador, o tempo para localização da mediana em um *bucket* é dominado pelo tempo gasto para dividir os *buckets* ao longo de cada dimensão. Os primeiros $log\ p$ níveis da árvore podem ser construídos em

$$\sum_{i=0}^{log\ p-1} O\left(k(k-1)\frac{N}{p} + t_s\frac{p}{2^i} + t_w k(k-1)\frac{N}{p} + t_h\frac{p}{2^i}\ log\ \frac{p}{2^i}\right)$$

$$= O\left(k(k-1)\frac{N}{p}\ log\ p + t_s p + t_w k(k-1)\frac{N}{p} log\ p + t_h\ p\ log\ p\right)$$

para um hipercubo, segundo [AFAG00b].

3.2.4 Resultados Experimentais da Literatura

Os resultados apresentados nesta seção se baseiam nos experimentos realizados em [AFAG00b], que apresenta uma comparação entre os três algoritmos usando uma implementação feita num computador CM-5 (*Connection Machine*) utilizando o modelo de comunicação hipercubo. O tempo de execução é decomposto em muitas partes, conforme mostrado na Tabela **??**: o tempo necessário para o pré-processamento (X), o custo de encontrar a mediana para todos os níveis (c), o tempo para um rearranjo local dos dados baseando-se na mediana (s) e a comunicação devido à movimentação dos dados (T).

Tabela 3.2:
resultados2

Método	**Pré-Processamento (X)**	**Cálculo Mediana (c)**	**Processamento Local (s)**	**Comunicação (T)**
Median	-	$O(\frac{N}{p}logp + (t_s + t_w)log^2 plogN)$	$O(k\frac{N}{p}logp)$	$O(t_s p + t_w(k\frac{N}{p}logp))$
Sort	$O(\frac{N}{p}logN + p^2logp + t_s p + t_w(\frac{N}{p} + p^2))$	$O(logp)$	$O(k(k-1)\frac{N}{p}logp)$	$O(t_s p + t_w(k(k-1)\frac{N}{p}logp))$
Bucket	$O(\frac{N}{p}(k + logp) + t_s p + t_w(k\frac{N}{p}))$	$O(\frac{N}{p})$	$O(k(k-1)\frac{N}{p}logp)$	$O(t_s p + t_w(k(k-1)\frac{N}{p}logp))$

O custo de movimentação dos $k-1$ vetores deve ser maior para os métodos *bucket-based* e *sort-based*, isto porque eles exigem a preservação da ordem dos

dados em oposição ao método *median-based* que trabalha somente com um vetor e não precisa usar primitivas que mantenham a ordem dos dados. O custo do *overhead r* é associado com a manutenção e processamento de sub-listas de todos os níveis, crescendo exponencialmente com o aumento do número de níveis.

Uma comparação dos métodos *sort-based* e *median-based* mostra que o primeiro tem um maior valor de (X), (s) e (T), e um menor valor de *r*. Para pequenos valores de $\frac{N}{p}$, o *median-based* não apresenta bons resultados na paralelização porque o custo de comunicação (c) predomina. O custo de calcular a mediana é insignificantemente baixo porque é necessário este cálculo em ambas as estratégias. O valor de (T) é menor para o *median-based* se comparado com o *sort-based* desde que ele use uma primitiva para movimentação dos dados que não mantenha a ordem. Espera-se que o método *median-based* tenha melhor desempenho, exceto para grandes valores de p.

Uma comparação do *bucket-based* e do *median-based*, baseada na complexidades de tempo, mostra que o primeiro possui maiores valores de (X), (s), *r*, e (T). Para pequenos valores de $\frac{N}{p}$ espera-se que o método *median-based* tenha pior desempenho que o *bucket-sort*, que tem pequeno *overhead* de comunicação para cálculo da mediana a menos que p seja muito pequeno. Um pequeno valor de p mantém o valor de (c) baixo para o método *median-based*, mas o valor de (X) pode ser grande para o *bucket-based*. Para grandes valores de $\frac{N}{p}$, (c) não domina no custo global e é significantemente mais baixa que (T), necessário a ambas as estratégias. O método *bucket-based* necessita que a ordem dos dados seja mantida, e conseqüentemente, tem um maior valor de (T) que o método *median-based*. Por causa disto é esperado que o *median-based* seja melhor, exceto para grandes valores de p.

O método *bucket-based* tem baixos valores de (X) e (c) quando comparado como o método *sort-based*. Em conseqüência disto, ele trabalha melhor quando a diferença no tempo de processamento é maior que o tempo total gasto para o cálculo da mediana. O custo do *sort-based* decresce a cada nível (porque o tamanho do *bucket* diminui com o aumento do número de níveis).

Em [AFAG00b] são apresentados testes experimentais utilizando os três métodos expostos, utilizando diferentes valores de $\frac{N}{p}$. Os resultados destes testes mostram que o método *bucket-based* é sempre melhor que o método *sort-based*. O método *median-based* é a melhor abordagem para grandes valores de $\frac{N}{p}$ (maior que 8K por processador), enquanto que o método *bucket-based* é melhor para pequenos valores de $\frac{N}{p}$ (menores que 4K). As melhorias de cada um destes métodos sobre o outro é substancial para estes valores. Para grandes valores de k, espera-

se que o tempo gasto pelo *bucket-based* cresça mais rapidamente que a estratégia *median-based* principalmente devido ao *overhead* no gerenciamento das listas e dos *buckets*.

3.3 Construção Global da Árvore

Conforme já mencionado, a construção paralela da árvore $k-d$ pode ser decomposta em duas partes: construção paralela da árvore até $log\ p$ níveis seguida por uma construção local do restante. Potencialmente, diferentes estratégias podem ser usadas em cada parte. Dada a existência dos três métodos estudados, isto resulta em nove possibilidades de combinação. Mas, baseando-se nos resultados obtidos nas seções anteriores, apenas cinco opções são viáveis:

1. Abordagem *sort-based* para os primeiros $log\ p$ níveis, seguida da mesma abordagem localmente. Usar a abordagem *sort-based* para os $log\ p$ níveis, tem o benefício adicional de que os dados locais estarão ordenados à esquerda. Assim, não é necessário pré-processamento para a construção da árvore local. Usar outra abordagem localmente não deverá terá melhor desempenho devido justamente a esta razão;

2. Abordagem *median-based* para os primeiros $log\ p$ níveis, seguida por um *sort-based* localmente;

3. Abordagem *median-based* para os primeiros $log\ p$ níveis, seguida por um *median-based* localmente;

4. Abordagem *bucket-based* para os primeiros $log\ p$ níveis, seguida por um *median-based* localmente;

5. Abordagem *median-based* para os primeiros $log\ p$ níveis, seguida por ordenação de cada *bucket*, e depois a aplicação do método *sorted-based.*

Comparações foram realizadas com estas cinco abordagens para um número diferente de níveis ($log\ p$ a $log\ N$), para um número diferente de processadores (8, 32, 128), e diferentes valores de $\frac{N}{p}$ (4K, 16K, e 128K). Os resultados mostraram que para grandes valores de $\frac{N}{p}$, a estratégia 3 é melhor, a menos que o número de níveis seja próximo de $log\ N$, para o qual a estratégia 2 é melhor. Para pequenos valores de $\frac{N}{p}$, a estratégia 4 é preferível. Se o número de níveis for próximo de $log\ N$, as estratégias 1 e 5 são as melhores.

3.4 Aplicações

As árvores $k-d$ possuem muitas aplicações nas mais diversas áreas, por este motivo este trabalho não se aprofundará muito nas aplicações, apenas fará uma breve descrição de algumas encontradas na literatura [Ski97].

Particionamento de grafos. O problema é criar p partições para distribuir o grafo para p processadores, requerendo a construção de somente os primeiros $log\ p$ níveis.

Aplicações hierárquicas como $n-body$ simulações. Por exemplo: O problema é calcular o estado (posição e velocidade) de N corpos dado um tempo $t > 0$, dado um estado inicial $t = 0$.

Outra aplicação hierárquica é a dinâmica das moléculas. Dinâmica das moléculas simula o movimento de átomos e moléculas, com o objetivo de integrar equações Newtonianas para um sistema de N partículas.

Pesquisa de pontos vizinhos. Dado um ponto q, quais são os pontos mais próximos? A árvore $k-d$ é um dos métodos mais rápidos para resolver este problema, pois necessita de $O(log\ n)$, sendo que normalmente são requeridos n^2.

Pesquisa em extensão ortogonal. Existem aplicações onde se deseja encontrar um conjunto de pontos que cabem em um dado retângulo do plano.

Localização de pontos em um região. A decomposição do plano é feita em regiões poligonais.

Existem muitas outras aplicações para a árvore $k-d$, estas são apenas algumas e são demonstradas apenas com o objetivo de dar uma noção do tipo de problema que pode se beneficiar de um método eficiente para a construção de árvores $k-d$. A aplicação mais citada na bibliografia é sem dúvida a pesquisa de pontos vizinhos, foram encontrados diversos artigos falando especificamente desta aplicação.

Capítulo 4

Conclusões

Neste trabalho foram consideradas várias estratégias para a construção paralela de árvores de busca binária multidimensional. Tradicionalmente, a estratégia usada para a construção de árvores $k-d$, seja de maneira seqüencial ou paralela tem sido a $sort-based$. Segundo [AFAG00b], para conjuntos de pontos com duas dimensões a estratégia *median-based* é mais rápida para grandes valores de $\frac{N}{p}$, a menos que o número de níveis seja próximo de $log\ N$. Neste caso, a melhor opção é usar o *median-based* para os primeiros $log\ p$ níveis, seguida do método *sort-based* para a construção local. Para pequenos valores de $\frac{N}{p}$, usar o método *bucket-based* e, em seguida, o *median-based* localmente é a melhor escolha para aplicações onde a árvore só precisa ser parcialmente construída. Quando ocorre o oposto, ou seja, quando a árvore precisar ser construída quase totalmente, a melhor alternativa é usar o método *bucket-based* e em seguida o *sort-based* para a construção local.

É interessante observar que uma abordagem completamente *sort-based* não apresenta o melhor desempenho nem mesmo quando a árvore é construída totalmente, isto quer dizer que o custo de ordenação é tão alto que, mesmo tendo que construa-se muitos níveis, ele não compensa.

Sobre o método *median-based,* [AFAG00b] comenta que ele exibe uma boa escalabilidade, conforme ele pôde observar em seus experimentos e também na análise da complexidade.

A estratégia *bucket-based* deve ser usada em aplicações onde o número de pontos por processador é pequeno, como por exemplo, particionamento de grafos.

Para dados em k dimensões, o custo de pré-processamento por nível nos métodos *sort-based* e *bucket-based* aumenta proporcionalmente a k. No *median-based*

este custo computacional não existe. Outra observação que pode ser feita é que enquanto o tempo de movimentação dos dados no *median-based* aumenta proporcionalmente a k, nos outros métodos ele aumenta em $k(k-1)$. Baseado nisto, o mesmo autor afirma que o *median-based* deve ser o melhor método para $k \geq 3$.

Por fim, baseando-se em tudo o que foi exposto pode-se concluir que a estrutura de dados estudada possui vários métodos de construção, bem como diversas formas de combinar estes métodos. Assim, não existe uma única e definitiva melhor forma de construção, a opção mais viável varia de uma aplicação para outra. E como existem aplicações muito variadas é possível que todos os métodos estudados continuem sendo utilizados e pesquisados, principalmente porque está é uma área de estudos relativamente recente.

Referências Bibliográficas

[AFAG00a] Ibraheem Al-Faraih, Srinivas Aluru, and Sanjay Goil. Parallel construction of k-d trees and related problems. ResearchIndex - The NECI Scientific Literature Digital Library, 2000.

[AFAG00b] Ibraheem Al-Faraih, Srinivas Aluru, and Sanjay Goil. Parallel construction of multidimensional binary search trees. *IEEE Transaction on Parallel and Distributed Systems*, 11(2):136–148, February 2000.

[BM00] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. ResearchIndex - The NECI Scientific Literature Digital Library, 2000.

[DJzC00] Luc Devroye, Jean Jabbour, and Carlos zamora Cura. Squarish k-d trees. ResearchIndex - The NECI Scientific Literature Digital Library, 2000.

[Goi96] Sanjay Goil. Phantom: Parallelization of hierarchical applications using treecodes on multiprocessors. Technical report, New York, 1996. Final Report.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[Pro97] Octavian Procopiuc. Data structures for spatial database systems. ResearchIndex - The NECI Scientific Literature Digital Library, 1997.

[Qui94] M. J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.

[Ski97] Steven S. Skiena. *The Algorithm Design Manual*. Telos, New York, 1997.