Paradigmas Divisão-e-Conquista e Varredura no Plano para Geometria Computacional Paralela

Henrique Mongelli

Resumo. Neste trabalho apresentamos duas estratégias muito utilizadas para o desenvolvimento de algoritmos para problemas de Geometria Computacional: divisão-e-conquista e varredura no plano. Vamos mostrar como estes dois paradigmas podem ser usados no desenvolvimento de algoritmos paralelos para Geometria Computacional. Através de dois problemas ilustramos a utilização destas estratégias no desenvolvimento de algoritmos paralelos. Os algoritmos são executados em tempo paralelo ótimo $O(\log n)$. Além disso apresentamos algumas referências que tratam de algoritmos paralelos para problemas de geometria computacional.

1 Introdução

Neste trabalho apresentamos duas estratégias muito utilizadas para o desenvolvimento de algoritmos para problemas de Geometria Computacional: divisão-e-conquista e varredura no plano. Vamos mostrar como estes dois paradigmas podem ser usados no desenvolvimento de algoritmos paralelos para Geometria Computacional.

Inicialmente vamos ver a necessidade dos computadores paralelos. Os computadores paralelos começaram a surgir com grande força na década de 80 pelas necessidades das aplicações, utilizando o computador. Além disso, o aumento da velocidade, diminuição do tamanho e o barateamento dos microprocessadores contribuiu para o desenvolvimento das máquinas paralelas.

Mas se conseguimos microprocessadores mais rápidos, mais baratos e menores, por que utilizarmos computadores paralelos? A resposta para isto está nas leis físicas que limitam a velocidade de comunicação entre os componentes dos computadores, fazendo com que todo o ganho obtido no processamento se perca nas comunicações entre eles. A aproximação dos componentes poderia parecer uma solução; porém, existe um limite a partir do qual estes acabam interferindo entre si.

A idéia de computação paralela não é recente. Quando o primeiro computador digital foi criado, já se sabia do ganho obtido com a divisão de tarefas. Na década de 70, os supercomputadores já adotavam o *pipeline* em sua arquitetura, alguns adotavam componentes especializados que liberavam o processador para outras tarefas. Estes exemplos não caracterizavam computadores paralelos, apesar de neles haver ganho de tempo.

O conceito de computador paralelo dado em JáJá [16] está descrita a seguir:

Um **computador paralelo** *é simplesmente uma coleção de processadores, tipicamente do mesmo tipo, interconectados de uma certa maneira a permitir a coordenação de suas atividades e a troca de dados.*

O uso de computadores paralelos faz-se necessário em diversos problemas onde o volume de dados e cálculos é grande e precisa-se de rapidez na obtenção das respostas. Tais problemas podem ser encontrados na previsão do tempo, na prospecção de petróleo, na dinâmica dos fluídos, na computação gráfica, entre outros.

Algoritmos de Geometria Computacional encontram aplicações em diversas áreas tais como projeto de circuitos VLSI, robótica e computação gráfica e envolvem grande quantidade de dados. Desde 1975 tem havido um grande desenvolvimento em algoritmos seqüenciais para Geometria Computacional. Porém até 1985 pouco havia se publicado sobre o desenvolvimento de algoritmos paralelos para tais problemas.

Com a computação paralela a criação de algoritmos modifica-se sensivelmente. As estratégias utilizadas em algoritmos seqüenciais para resolver um problema não servem totalmente para a criação de algoritmos paralelos para o mesmo problema. Os recursos disponíveis são maiores e deve-se aproveitá- los ao máximo. Na criação dos algoritmos leva-se em conta também a natureza dos problemas. Alguns são inerentemente paralelizáveis; outros apresentam um paralelismo não tão transparente, exigindo estratégias mais elaboradas na montagem dos algoritmos; e existem aqueles em que o grau de paralelismo é muito baixo ou inexistente. Além disso, no desenvolvimento dos algoritmos temos que considerar a arquitetura paralela a ser utilizada, pois isto afeta o desempenho dos algoritmos.

Muitos técnicas utilizadas em problemas de Geometria Computacional são inerentemente seqüenciais. Para desenvolver-se algoritmos paralelos precisamos explorar estruturas de dados que sejam simples o suficiente para serem calculadas, eficientemente, em paralelo enquanto sejam potentes o suficiente para responder a consultas, eficientemente. O artigo de Aggarwal *et al* [1] mostra algoritmos paralelos para resolver alguns problemas geométricos.

O conceito de "eficiente" aqui significa que o algoritmo deve rodar em tempo paralelo polilogarítmico – $O(\log^k n)$ – usando um número polinomial de processadores. Outro conceito é o de otimalidade. Seja T_s o tempo do melhor algoritmo seqüencial para um problema e T_p o tempo de um algoritmo paralelo usando p processadores para resolver o mesmo problema, este algoritmo paralelo é **ótimo** se $pT_p = O(T_s)$. O número total de operações realizadas pelo algoritmo, pT_p é denominado **custo** ou **trabalho** (alguns autores fazem distinção entre os dois conceitos [16]).

Como vimos, o desenvolvimento depende também do modelo de computação. Na seção 2.1 veremos a descrição da PRAM (*Parallel Random Access Machine*) que é um conjunto de processadores que rodam programas RAM e se comunicam acessando uma memória comum. Este modelo é o mais utilizado no desenvolvimento de algoritmos paralelos, entretanto não é um modelo realístico e não é utilizado na prática. Nos últimos anos alguns modelos mais realísticos como BSP(*Bulk Synchronous Parallel*) [24] e CGM(*Coarse Grained Multicomputer*) [10, 9, 8], que levam em conta a comunicação entre os processadores que podem ser um gargalo, tem sido utilizados no desenvolvimento de algoritmos paralelos. Estes modelos não levam em conta a arquitetura de interconexão entre os processadores. Alguns algoritmos são mais particulares e são desenvolvidos, especificamente, para uma dada arquitetura. Neste trabalho, os algoritmos são desenvolvidos utilizando-se o modelo PRAM. Na seção 5 apresentamos algumas referências para os problemas de Geometria Computacional em outros modelos.

Na seção 3 apresentamos o conceito de Divisão-e-Conquista. Utilizamos o paradigma para desenvolver um algoritmo paralelo, que leva tempo $O(\log n)$ usando $O(n \log n)$ operações, para resolver o problema de determinação do fecho convexo de um conjunto S de n pontos no plano.

Na seção 4 discutimos como funciona o método de varredura no plano. Consideramos o problema: dado um conjunto de segmentos horizontais e verticais, determinar, para cada segmento vertical, o número de segmentos horizontais que o intersectam. A idéia da varredura no plano parece ser inerentemente seqüencial, uma vez que a linha de varredura percorre todos os pontos chamados pontos de evento, um a um, atualizando a estrutura que mantém informações sobre a intersecção da linha de varredura com os objetos geométricos dados. A estrutura utilizada no exemplo e que pode ser estendida para outras aplicações é a árvore de varredura no plano. Esta estrutura juntamente com outras estruturas auxiliares podem ser construídas em tempo paralelo $O(\log n)$ usando $O(n \log n)$ operações; e é usada para processar todas as linhas de varredura concorrentemente.

2 Modelos Paralelos

Nesta seção introduziremos os modelos PRAM e CGM.

2.1 PRAM – *Parallel Random Access Machine*

A PRAM é o modelo mais usado para computação paralela. Ela é uma coleção de processadores síncronos que trabalham em paralelo e se comunicam através de uma memória compartilhada. Cada processador é uma RAM seqüencial e os processadores podem executar diferentes instruções em diferentes conjuntos de dados (MIMD). Como temos uma memória compartilhada e os processadores trabalham concorrentemente, pode ocorrer de dois ou mais processadores tentarem acessar (ler ou escrever) uma mesma posição de memória simultaneamente. Conforme o acesso concorrente seja permitido ou não, temos três possibilidades de modelo:

- **CRCW** (*Concurrent Read, Concurrent Write*)
  É permitida simultaneidade na leitura e escrita na mesma posição de memória por diferentes processadores. Para administrar os possíveis conflitos temos os critérios:

  modelo **CRCW** com escrita **prioritária**
  o processador com índice menor tem prioridade para escrever.

  modelo **CRCW** com escrita **comum**
  é necessário que todos os processadores em conflito escrevam o mesmo valor.

  modelo **CRCW** com escrita **arbitrária**
  um dos processadores deve ter sucesso na escrita, não importando qual deles.

- **CREW** (*Concurrent Read, Exclusive Write*)
  É permitida leitura simultânea de uma mesma posição de memória por mais de um processador. Não é permitida a escrita simultânea em uma mesma posição por mais de um processador.

- **EREW** (*Exclusive Read, Exclusive Write*)
  Não é permitido leitura ou escrita simultâneas em uma mesma posição de memória.

2.2 CGM – *Coarse Grained Multicomputer*

O modelo CGM foi proposto por Dehne [10, 9, 8]. Ele é muito parecido com o modelo BSP(*Bulk-Synchronous Parallel*). Entretanto ele usa apenas dois parâmetros, n, o tamanho da entrada; e p, o número de processadores, cada um com memória local de tamanho $O(n/p)$. O termo *coarse grained*[1] vem do fato de o tamanho da memória local ser muito maior que $O(1)$. Cada processador é conectado por um roteador que pode enviar mensagens ponto-a-ponto. Um algoritmo CGM consiste de uma seqüência alternada de fases de computação e fases de comunicação separadas por uma barreira de sincronização. Uma fase de computação é equivalente a um superpasso no modelo BSP. Na fase de computação, usualmente utilizamos o melhor algoritmo seqüencial em cada processador para processar seus dados localmente.

Em cada fase de comunicação cada processador troca no máximo um total de $O(n/p)$ dados com outros processadores. No modelo CGM o custo de comunicação é modelado pelo número de fases de comunicação. Muitos algoritmos geométricos e para grafos necessitam de um número de rodadas de comunicação constante ou de ordem $O(\log p)$ [10, 9, 8, 11, 6]. Ao contrário do modelo PRAM que freqüentemente é projetado para $p = O(n^k)$, com $k \in \mathcal{N}$ e cada processador recebe um número pequeno de dados de entrada, neste modelo consideramos os casos mais realísticos onde $n \gg p$. O modelo CGM é particularmente adequado nas máquinas paralelas atuais onde a velocidade global de computação é consideravelmente maior que a velocidade global de comunicação.

3 Divisão-e-Conquista

A estratégia de **divisão-e-conquista** é composta de três passos básicos:

Dividir. O problema é dividido em um número de subproblemas.

Conquistar. Os subproblemas são resolvidos recursivamente. Se o tamanho do subproblema é suficientemente pequeno ele é resolvido de maneira direta.

Combinar. As soluções dos subproblemas são combinadas para obter a solução do problema original.

[1] Granularidade grossa.

O sucesso desta estratégia depende se a primeira e a última etapa podem ser executadas eficientemente. Esta estratégia é muito usada no desenvolvimento de algoritmos seqüenciais. Ela também nos leva a uma forma natural de explorar o paralelismo: os subproblemas gerados podem ser executados independentemente e portanto em paralelo.

Em Geometria Computacional muitos problemas são resolvidos, seqüencialmente, através do paradigma de divisão-e-conquista. Um destes problemas é a de determinação do fecho convexo de um conjunto de pontos no plano, que pode ser resolvido através desta estratégia em tempo ótimo $O(n \log n)$. Na solução seqüencial deste problema as fases de dividir e conquistar são simples de se obter, a fase de combinar é a mais trabalhosa.

Na versão do algoritmo paralelo utilizando esta estratégia leva tempo ótimo $O(\log n)$ em uma PRAM EREW. Neste caso, a maior dificuldade em garantir esta complexidade de tempo está, também, na fase combinar. Antes de apresentarmos o algoritmo vamos ver algumas definições.

3.1 Fecho Convexo

Definição 1 (Fecho Convexo) *Dado um conjunto $S = \{p_0, p_1, \ldots, p_{n-1}\}$ de n pontos no plano, o* ***fecho convexo*** *de S, denominado conv(S), é o menor polígono convexo que contém todos os n pontos de S.*

Nosso problema é:

Problema 1 *Dado um conjunto $S = \{p_0, p_1, \ldots, p_{n-1}\}$ de pontos no plano determinar conv(S), o fecho convexo dos pontos de S.*

Dado um conjunto $S = \{p_0, p_1, \ldots, p_{n-1}\}$ de pontos no plano, ordenamos este conjunto através de um algoritmo paralelo que leva tempo $O(\log n)$ usando $O(n)$ processadores [16, 22]. Com isto temos os pontos p e q de menor e maior abscissa. Claramente, estes dois pontos pertencerão ao fecho convexo e dividirão o fecho convexo em duas partes: uma superior (Sconv(S)) e uma inferior (Iconv(S)). A idéia é determinar separadamente o fecho superior e o inferior para então determinar o fecho convexo.

Descreveremos a seguir como determinar o fecho superior. O fecho inferior é obtido de maneira análoga.

O algoritmo numa descrição de alto nível está dado a seguir.

A complexidade deste algoritmo é dada por:

$$T(n) \leq T(n/2) + cT_{\tan},$$

onde $T_{\tan}$ é o tempo para determinar em paralelo a tangente comum superior de $\mathrm{Conv}(S_1)$ e $\mathrm{Conv}(S_2)$.

ALGORITMO FECHO CONVEXO SUPERIOR

ENTRADA: um conjunto $S = \{p_0, p_1, \ldots, p_{n-1}\}$ de pontos ordenados pela $x-$coordenada no plano.
SAÍDA: o fecho convexo conv(S).

1. Se $n \leq 4$ então determine o fecho convexo destes pontos e termine.

2. Sejam $S_1 = \{p_0, p_1, \ldots, p_{(n-1)/2}\}$ e $S_2 = \{p_{(n-1)/2+1}, \ldots, p_{n-1}\}$. Calcule recursivamente Sconv(S_1) e Sconv(S_2).

3. Encontre a tangente superior comum entre Sconv(S_1) e Sconv(S_2) e obtenha Sconv(S).

3.1.1 Um algoritmo de tempo constante para determinação da tangente comum superior

Para obtermos uma complexidade de tempo $O(\log n)$ temos que determinar a tangente comum superior em tempo constante. A seguir descrevemos como obter esta tangente com este tempo.

Seja $S = (p_0, p_1, \ldots, p_{n-1})$ uma seqüência de pontos do plano ordenados pela $x-$coordenada e sejam $S_1 = \{p_0, p_1, \ldots, p_{(n-1)/2}\}$ e $S_2 = \{p_{(n-1)/2+1}, \ldots, p_{n-1}\}$. A tangente comum superior entre Conv(S_1) e Conv(S_2) é a tangente comum tal que Conv(S_1) e Conv(S_2) estão abaixo dela. A tangente comum inferior é definida de maneira análoga.

Sejam Sconv$(S_1) = (l_1, l_2, \ldots, l_s)$ e Sconv$(S_2) = (r_1, r_2, \ldots, r_t)$ os fechos convexos superiores dos conjuntos S_1 e S_2, respectivamente, dados no sentido anti-horário, com l_i,$r_j \in S$, para $1 \leq i \leq s$ e $1 \leq j \leq t$; l_s é o extremo direito de Sconv(S_1) e r_t é o extremo direito de Sconv(S_2). Nosso problema é determinar pontos $u = l_i$ e $v = r_j$ de Sconv(S_1) e Sconv(S_2), respectivamente, com $1 \leq i \leq s$ e $1 \leq j \leq t$, tais que uv é a tangente superior comum. O fecho resultante é dado pela seqüência $(r_t, \ldots, r_j, l_i, \ldots, l_1)$. Assim, se tivermos determinado u e v, o fecho pode ser determinado em tempo paralelo $O(1)$ usando n processadores.

Nosso foco agora é justamente determinar estes pontos de tangência. A idéia é muito parecida com aquela do algoritmo seqüencial de divisão-e-conquista para se determinar as tangentes comuns. Antes de apresentarmos o algoritmo vamos relacionar alguns resultados que motivam e justificam a idéia.

Primeiramente, vamos ver como a tangente entre um ponto l_i de Sconv(S_1) e Sconv(S_2) pode ser determinada; isto é, queremos determinar $r_{j(i)}$ tal que Sconv(S_2) está abaixo da linha determinada por l_i e $r_{j(i)}$. O lema a seguir mostra como podemos obter esta tangente usando um algoritmo de busca paralela.

Lema 1 *Seja l_i um ponto qualquer de Sconv(S_1). Então, dado um ponto qualquer r_k de Sconv(S_2) pode-se determinar, em tempo seqüencial $O(1)$, se $r_{j(i)}$ está à direita, é igual a, ou está à esquerda de r_k, onde $r_{j(i)}$ é o ponto de Sconv(S_2) tal que $l_i r_{j(i)}$ é a tangente a Sconv(S_2).*

Prova. Seja L uma reta determinada por l_i e r_k, e seja L' (L'') a parte de L estritamente à esquerda (estritamente à direita) de r_k (figura 3.1.1). Então, se L' está acima do segmento

$r_k r_{k+1}$ e L'' está abaixo do segmento $r_{k-1}r_k$, então $r_{j(i)}$ está à direita de r_k. Se $r_k = r_{j(i)}$, então r_{k-1} e r_{k+1} estão abaixo de L. Caso contrário, $r_{j(i)}$ está à esquerda de q_k.

Como l_i e r_k são dados, podemos determinar cada uma das três condições em tempo seqüencial $O(1)$. Portanto, segue o lema. □

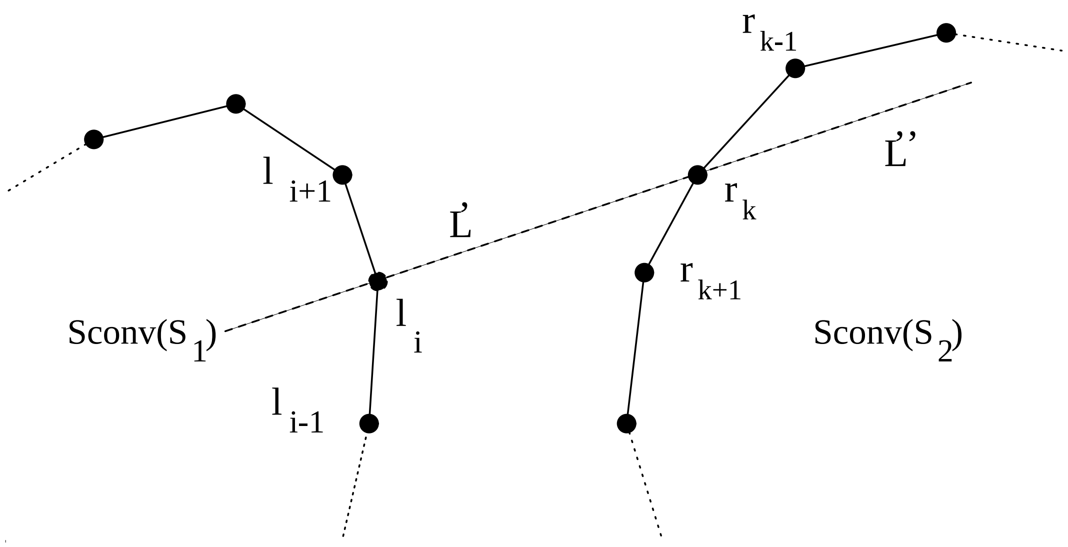

Figura 1: Determinação da tangente a $\mathrm{Sconv}(S_2)$ passando por l_i.

Corolário 1 *Dados os fechos convexos superiores $Sconv(S_1)$ e $Sconv(S_2)$, um ponto l_i de $Sconv(S_1)$, a tangente ao fecho $Sconv(S_2)$ $l_i q_j$ pode ser determinada em tempo $O\left(\frac{\log t}{\log k}\right)$ usando k processadores, onde t é o número de pontos de $Sconv(S_2)$ e $1 < k \leq t$.*

Prova. O fecho convexo $\mathrm{Sconv}(S_2)$ é dado com seus pontos ordenados em sentido anti-horário. Podemos usar um algoritmo de busca em paralelo [16] para determinar q_j. Escolhemos k pontos de $\mathrm{Sconv}(S_2)$ que o dividem em $k+1$ partes, com aproximadamente o mesmo número de pontos. Para cada um dos k pontos $r_{q_1}, \ldots, r_{q_k}$, determinamos, em paralelo, em tempo $O(1)$, a posição relativa entre o vetor $l_i r_m$ e o sucessor de r_m em $\mathrm{Sconv}(S_2)$, com $q_1 \leq m \leq q_k$. Repetimos o processo até que q_j seja identificado. O tempo é dado por $T(n) = T(n/k) + c$, o que nos dá $T(n) = O\left(\frac{\log t}{\log k}\right)$. □

Seja uv a tangente comum superior aos fechos convexos superiores $\mathrm{Sconv}(S_1)$ e $\mathrm{Sconv}(S_2)$. Suponhamos que para qualquer ponto l_i de $\mathrm{Sconv}(S_1)$ seja dado o ponto r_j de $\mathrm{Sconv}(S_2)$ tal que $l_i r_j$ seja tangente a $\mathrm{Sconv}(S_2)$, podemos determinar a posição relativa entre u e l_i em tempo seqüencial $O(1)$. Com isto podemos determinar, para qualquer ponto l_i de $\mathrm{Sconv}(S_1)$ sua posição relativa ao ponto u em tempo $O\left(\frac{\log t}{\log k}\right)$ usando k processadores. Se fizermos k aproximadamente igual a $\sqrt{t}$, temos um algoritmo paralelo de tempo $O(1)$ que determina , para qualquer ponto r_i, sua posição relativa em relação a u.

Estas idéias nos levam a um algoritmo paralelo de busca para isolar u que funciona como descrito a seguir. Escolhemos $\sqrt{s}$ pontos de Sconv(S_1), que dividem Sconv(S_1) em partes aproximadamente do mesmo tamanho. Podemos, então isolar u em uma destas partes em tempo $O(1)$ usando $\sqrt{s}\sqrt{t}$ processadores. De maneira análoga podemos isolar v em uma parte de Sconv(S_2) contendo aproximadamente $\sqrt{t}$ pontos em em tempo $O(1)$ usando $\sqrt{s}\sqrt{t}$ processadores. Assim, o tempo desta busca em paralelo é $O(1)$, com um total de $O(\sqrt{s}\sqrt{t}) = O(n)$ operações.

Temos agora no máximo $\sqrt{s}$ candidatos para o ponto u e $\sqrt{t}$ candidatos para o ponto v. Para cada par de candidatos, podemos chegar em tempo seqüencial $O(1)$, se ele forma uma tangente comum superior entre Sconv(S_1) e Sconv(S_2). Assim, podemos testar todos os pares em paralelo e, então, identificar a única tangente comum superior em tempo $O(1)$ usando $O(n)$ operações.

O algoritmo para determinação da tangente comum superior é dado a seguir.

ALGORITMO TANGENTE COMUM SUPERIOR

ENTRADA: Os fechos convexos superiores Sconv$(S_1) = (l_1, l_2, \ldots, l_s)$ e Sconv$(S_2) = (r_1, r_2, \ldots, r_t)$ no sentido anti-horário, com l_s extremidade direita de Sconv(S_1) e r_t extremidade direita de Sconv(S_2); onde $S_1 = \{p_0, p_1, \ldots, p_{(n-1)/2}\}$ e $S_2 = \{p_{(n-1)/2+1}, \ldots, p_{n-1}\}$ e tais que $x(p_1) < x(p_2) < \ldots < x(p_{n-1})$. Vamos assumir que $\sqrt{s}$ e $\sqrt{t}$ são inteiros.

SAÍDA: Pontos u e v tais que a reta que os une é a tangente comum superior entre Sconv(S_1) e Sconv(S_2).

1. Para cada i tal que $1 \leq i \leq \sqrt{s}$, determinar $q_{j(i\sqrt{s})}$ tal que $r_{i\sqrt{s}}q_{j(i\sqrt{s})}$ seja tangente a Sconv(S_2).

2. Para cada i tal que $1 \leq i \leq \sqrt{s}$, determinar se u está à esquerda, é igual a, ou está à direita de $r_{i\sqrt{s}}$. Se $u = r_{i\sqrt{s}}$, para algum i, então fim. Senão, determinar o bloco $A = (r_{l\sqrt{s}+1}, \ldots, r_{(l+1)\sqrt{s}-1})$ contendo u.

3. Para cada r_i no bloco A, determinar $q_{j(i)}$ tal que $r_i q_{j(i)}$ seja a tangente a Sconv(S_2), faça $u \leftarrow r_i$ e $v \leftarrow q_{j(i)}$ se $r_i q_{j(i)}$ é também tangente a Sconv(S_1).

A corretude do algoritmo segue das discussões que fizemos antes de apresentá-lo. Vamos assumir que $s + t$ processadores estejam disponíveis. O passo 1 pode ser executado em tempo $O(1)$ usando $\sqrt{t}$ processadores para cada i. como $\sqrt{st} \leq s + t$, todos os $q_{j(i\sqrt{s})}$ podem ser obtidos em tempo $O(1)$ usando $s + t$ processadores. O passo 2 pode ser feito em tempo $O(1)$ com $O(\sqrt{s})$ processadores. Analogamente para o passo 3. Assim o algoritmo pode ser executado em tempo $O(1)$ usando $s + t$ processadores. Assim, o número total de operações é $O(s + t) = O(n)$.

3.1.2 Combinando os resultados

Com o algoritmo de tempo constante para determinação da tangente comum superior obtido temos que o fecho convexo pode ser determinado em tempo $O(\log n)$, usando um total de $O(n \log n)$ operações. Assim, o algoritmo para determinação do fecho convexo é ótimo.

Dado um conjunto $S = \{p_0, p_1, \ldots, p_{n-1}\}$ de n pontos do plano. O algoritmo inicialmente pré-processa S, ordenando seus pontos pela $x-$coordenada. Esta etapa leva tempo ótimo $O(\log n)$ [16, 22]. Assumimos que os pontos agora são dados pela seqüência $S = (p_0, p_1, \ldots, p_{n-1})$ com $x(p_0) < x(p_1) < \ldots < x(p_{n-1})$.

Usamos a técnica de divisão-e-conquista para determinar conv(S) como se segue. Se $n \leq 4$ então resolvemos o problema de maneira direta. Caso contrário, chamamos o algoritmo recursivamente para calcular conv(S_1) e conv(S_2), onde $S_1 = (p_0, p_1, \ldots, p_{(n-1)/2})$ e $S_2 = (p_{(n-1)/2+1}, \ldots, p_{n-1})$. Combinamos conv$(S_1)$ e conv(S_2) determinando as tangentes comuns superior e inferior de conv(S_1) e conv(S_2), obtendo conv(S).

O algoritmo executa $O(\log n)$ iterações, cada uma delas leva tempo $O(1)$ usando um número linear de operações. Assim o algoritmo todo leva tempo $O(\log n)$ e $O(n \log n)$ operações.

O modelo PRAM utilizado é o CREW pois a leitura concorrente é necessária pelos algoritmo de ordenação e pelo algoritmo de busca paralela. Não é necessária escrita concorrente.

3.2 Notas

Aggarwal *et al* [1] apresentam um algoritmo paralelo para determinação do fecho convexo de um conjunto S de n pontos do plano, usando a estratégia de divisão-e-conquista, que é executado também em tempo ótimo $O(\log n)$. Resolve-se o problema determinando-se os fechos convexos superior e inferior. Vamos relatar brevemente como o algoritmo determina o fecho superior. O fecho inferior é determinado de maneira simétrica. Neste algoritmo o conjunto S também é pré-processado ordenando-se seus pontos pela $x-$coordenada. Os pontos de S são divididos em $\sqrt{n}$ conjuntos de $\sqrt{n}$ pontos. Recursivamente, o fecho convexo de cada conjunto é determinado. O algoritmo, então, combina estes fechos em tempo $O(\log n)$. Novamente, esta fase de combinar não é óbvia. No artigo é mostrado como concatenar os $\sqrt{n}$ fechos em $O(\log n)$ passos usando n processadores. Assim, o algoritmo também leva tempo $O(\log n)$ usando $O(n \log n)$ operações para determinar o fecho convexo de um conjunto S de n pontos no plano.

4 Varredura no Plano

A **varredura no plano** é uma técnica importante em Geometria Computacional. Dado um conjunto de objetos no plano, imaginamos uma reta vertical que varre o plano da esquerda para a direita. Para cada posição da linha de varredura podemos deduzir informações relevantes à solução procurada examinando a intersecção da linha de varredura com os objetos. Esta linha de varredura pára em pontos especiais, chamados **pontos críticos** ou **pontos de evento**. Temos duas estruturas básicas:

1. A **lista de pontos de evento** (*event point schedule*), que é uma seqüência de abscissas, ordenadas da esquerda para a direita, que define as posições de parada da linha de varredura.

2. O **estado da linha de varredura**(*sweep-line status*), que é uma descrição das informações relevantes da intersecção da linha de varredura com os objetos que estão sendo varridos.

Muitos problemas, cujos algoritmos seqüenciais usam o paradigma de varredura no plano, têm seus algoritmos paralelos desenvolvidos usando paradigmas completamente diferentes. Goodrich, Ghouse e Bright [15] citam algumas referências que exemplificam esta mudança e algumas que mantém o paradigma no algoritmo paralelo. Eles desenvolvem algoritmos para alguns problemas que utilizam a varredura não com uma linha vertical varrendo os objetos, mas outros tipos de varredura.

Aqui, veremos a varredura no plano através de um exemplo onde os objetos geométricos são segmentos horizontais e verticais. A técnica básica pode ser estendida para manipular outros tipos de objetos mais gerais.

4.1 Um exemplo

Dado um conjunto de segmentos horizontais e verticais, consideremos o problema de determinar, para cada segmento vertical, o número de segmentos horizontais que o intersectam.

Neste caso, os pontos críticos correspondem às extremidades dos segmentos. Inicialmente, as abscissas das extremidades dos segmentos horizontais são ordenados. A varredurra inicia-se na extremidade de menor x-coordenada. Os estados da linha de varredura são armazenados em uma árvore binária balanceada que admite as operações de busca, inserção e remoção em tempo $O(\log n)$. Cada folha da árvore corresponde à y-coordenada da extremidade esquerda de um segmento horizontal encontrada, mas cuja extremidade direita ainda não foi varrida. Cada nó interno contém o número de descendentes da sub-árvore da qual é raiz.

A linha de varredura ao atingir um ponto de evento verifica se o ponto é a extremidade esquerda de um segmento horizontal, em caso afirmativo o ponto é inserido na árvore. Se o ponto for uma extremidade direita de um segmento horizontal, então o valor de y é removido da árvore. Se um segmento vertical é encontrado, procuramos na árvore o número de valores de y entre os dois valores de y do segmento. Então, podemos determinar o número de segmentos horizontais intersectando cada segmento vertical desta maneira. Como cada extremidade é processada em tempo seqüencial $O(\log n)$, onde n é o número de segmentos, o algoritmo leva tempo total $O(n \log n)$.

Não é claro como pode ser feito o algoritmo paralelo para este problema, usando a técnica de varredura. Na seção a seguir vamos descrever uma estrutura que permite processar todas as linhas de varredura nos pontos de evento, concorrentemente.

4.2 A árvore de varredura no plano

Seja $S = \{s_1, s_2, \ldots, s_n\}$ um conjunto de segmentos horizontais que não se intersectam. Vamos supor que dois segmentos não tenham extremidades com as mesmas x-coordenadas. A **árvore de varredura no plano** é definida a seguir.

Sejam $x_1 < x_2 < \ldots < x_{2n}$ as x-coordenadas das extremidades dos n segmentos. Estes $2n$ pontos formam no eixo x $2n+1$ intervalos. Seja T a árvore balanceada construída nestes $2n+1$ intervalos, com cada folha u associada ao intervalo $[a_u, b_u]$ correspondente. Um nó interno v corresponde ao intervalo que é a união dos intervalos associados com seus descendentes. Temos, ainda, que armazenar informação adicional em cada nó v.

Seja v nó qualquer de T cujo intervalo seja dado por $[a_v, b_v]$. Seja Π_v a faixa vertical $[a_v, b_v] \times (-\infty, +\infty)$. Um segmento s_i **cobre** v se ele se estende ao longo de Π_v mas não ao longo de Π_z, onde $z = \text{pai}(v)$. Definimos $H(v)$ e $W(v)$:

$$H(v) = \{s_i / \ s_i \text{ cobre } v\},$$

$$W(v) = \{s_i / \ s_i \text{ tem no mínimo uma extremidade em } \Pi_v\}.$$

Notemos que, se s_i tem uma extremidade na fronteira de Π_v e está todo fora de Π_v, então s_i não será incluído em $W(v)$. Assumimos que os segmentos em cada $H(v)$ e $W(v)$ estão ordenados pelos valores das suas y-coordenadas. Esta árvore é chamada de árvore de varredura no plano.

Lema 2 *Seja T uma árvore de varredura no plano de um conjunto de n segmentos horizontais que não se intersectam, e sejam $H(v)$ e $W(v)$ as listas armazenadas em cada nó v. Então, cada uma das somas $\sum_v |H(v)|$ e $\sum_v |W(v)|$ tem tamanho $O(n \log n)$.*

Prova. Cada segmento pode cobrir no máximo dois nós em cada nível, uma vez que um segmento que cobre um nó v não pode cobrir seu pai. Donde, a soma dos tamanhos das listas em cada nível é $O(n)$, e portanto $\sum_v |H(v)| = O(n \log n)$. Como cada segmento tem duas extremidades, nenhum segmento pode aparecer mais que duas vezes em todos os $W(v)$s em cada nível. Donde, $\sum_v |W(v)| = \sum_v |H(v)| = O(n \log n)$. □

Agora veremos como construir eficientemente $H(v)$ e $W(v)$ para cada nó v em T. Inicialmente vamos ver como $W(v)$ é construída. Se v é uma folha correspondente ao intervalo $[x_i, x_{i+1}]$, então $W(v)$ consiste de no máximo dois segmentos: um com abscissa x_i e outro com abscissa x_{i+1}. Assim, a determinação de $W(v)$ para cada v que é folha leva tempo constante usando um número linear de operações.

Vamos considerar que v seja um nó interno com u e w, filhos esquerdo e direito, respectivamente. Todo segmento que esteja em $W(v)$ deve estar em $W(u)$ ou $W(w)$. A recíproca também é válida, isto é, todo segmento que esteja em $W(u)$ ou em $W(w)$ deve estar em $W(v)$. Assim, $W(v)$ é uma lista ordenada que é obtida intercalando-se $W(u)$ e $W(w)$ em relação aos valores das y-coordenadas dos segmentos. A primeira idéia para se obter $W(v)$, a partir do que foi colocado no parágrafo acima, é de determinar a intercalação um nível de cada vez, mas isto nos leva a um algoritmo que roda em tempo maior que $O(\log n)$. Entretanto, podemos usar o algoritmo *pipelined merge-sort* [16] que calcula as listas ordenadas geradas em todos os nós de uma árvore binária qualquer. Isto nos leva a um algoritmo que determina $W(v)$ para cada nó v da árvore binária em tempo $O(\log n)$ usando um total de $O(n \log n)$ operações.

Para determinar $H(v)$ usaremos os valores de $W(v)$ já calculados. Seja v um nó qualquer, e seja z o pai de v. Todo segmento em $H(v)$ deve aparecer em $W(z)$. Nosso objetivo é selecionar os subconjuntos de segmentos em $W(z)$ que formam $H(v)$.

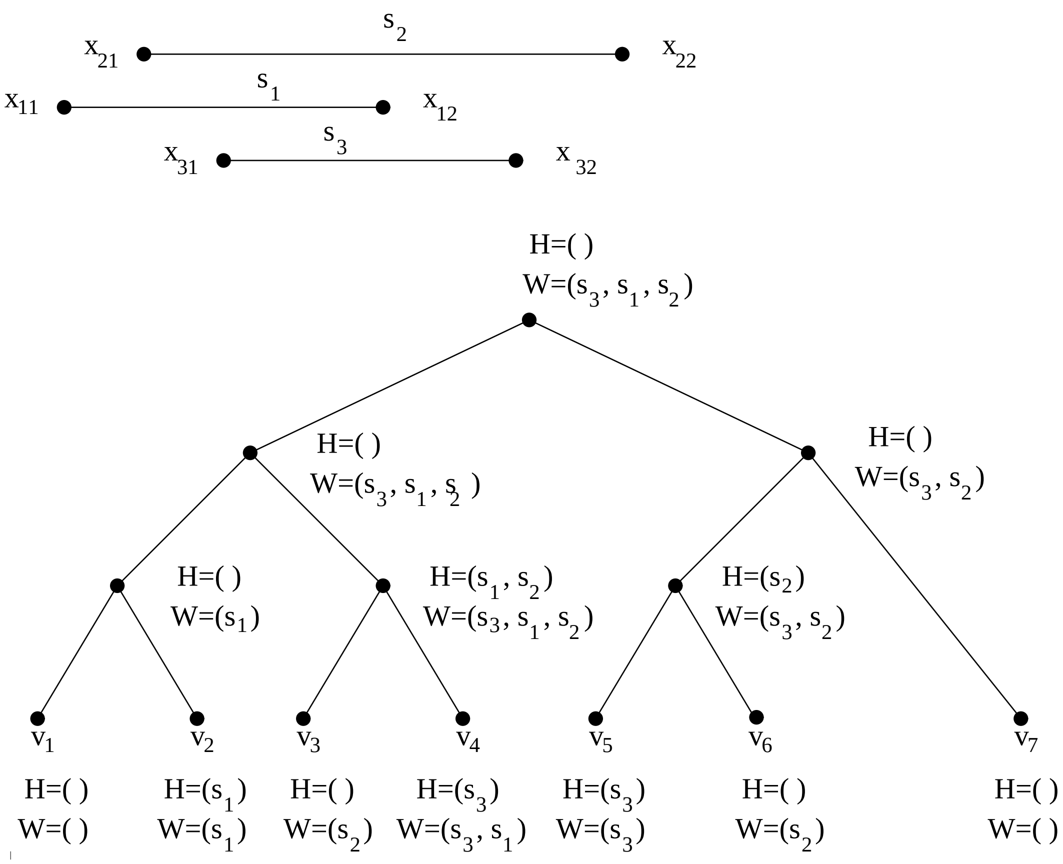

Figura 2: Exemplo de uma árvore de varredura para os três segmentos: s_1, s_2 e s_3

Podemos testar se um segmento $W(z)$ pertence a $H(v)$ em tempo seqüencial $O(1)$ comparando as x-coordenadas de suas extremidades com $[a_v, b_v]$ e $[a_w, b_w]$, onde w é irmão de v. Marcamos cada tal segmento como pertencente a $H(v)$.

Temos, então, que podemos marcar todos os segmentos de $W(z)$ pertencentes a $H(v)$ em tempo $O(1)$ usando um número linear de operações. Como $\sum_v |W(z)| = O(n \log n)$, o número de operações total é $O(n \log n)$. Observemos que $H(r) = \emptyset$, onde r é a raiz da árvore. Cada nó interno contém uma cópia marcada para seu filho esquerdo e uma cópia marcada para seu filho direito. Além disso, os elementos marcados em cada $W(z)$ aparecem ordenados pelo valor de suas y-coordenadas. Usando algoritmos de soma-prefixa [16, 22] podemos compactar cada cópia em paralelo em tempo $O(\log n)$ usando um total de $O(n \log n)$ operações, uma vez que $\sum_{v \in T} |W(v)| = O(n \log n)$.

Utilizamos o modelo PRAM CREW pois o algoritmo *pipelined merge-sort* o utiliza. Além disso, a marcação dos elementos de $W(z)$ que pertencem a $H(v)$ em tempo $O(1)$ necessita de acesso concorrente ao intervalo $[a_v, b_v]$.

5 Notas finais

O objetivo deste trabalho foi mostrar como dois métodos importantes usados no desenvolvimento de algoritmos seqüenciais para problemas de Geometria Computacional: divisão-e-conquista e varredura no plano, são implementados em paralelo. Utilizamos o modelo PRAM que é o mais simples para se desenvolver algoritmos paralelos, apesar de não ser realístico. A seguir apresentamos algumas referências a outros artigos, envolvendo Geometria Computacional e outros conceitos desenvolvidos no texto.

Mostramos como o método de divisão-e-conquista pode ser usado para obtenção de um algoritmo paralelo para o problema do fecho convexo de um conjunto de pontos do plano. Em Aggarwal *et al* [1] encontramos um outro algoritmo paralelo para o mesmo problema usando o paradigma de divisão-e-conquista. Um estudo mais completo do paradigma de divisão-e-conquista em paralelo pode ser visto em Axford [5].

Em Dehne *et al* [9] são apresentados algoritmos paralelos escaláveis e independentes da arquitetura para alguns problemas de Geometria Computacional, inclusive para o fecho convexo de pontos em $\Re^k$. Dehne *et al* [8] apresentam um algoritmo aleatório para determinação do fecho convexo tridimensional no modelo CGM. O próprio modelo CGM foi inicialmente introduzido em um artigo de Dehne *et al* [10] e aplicado no desenvolvimento de algoritmos paralelos para problemas de Geometria Computacional. Resultados mais recentes no modelo CGM estão em Dehne e Song [11] e Cáceres *et al* [6].

No modelo BSP [24], uma referência relacionada a Geometria Computacional é o relatório técnico de Siniolakis [23].

Como foi visto, muitos problemas geométricos cujos algoritmos seqüenciais utilizam a técnica de varredura não têm um correspondente algoritmo paralelo utilizando a mesma técnica. O exemplo apresentado mostra esta correspondência. Em Goodrich *et al* [15] são apresentados alguns algoritmos que utilizam a técnica de varredura e contém algumas referências a problemas cujos problemas usam o método de varredura no algoritmo seqüencial e um paradigma completamente diferente no algoritmo paralelo.

Outras referências sobre algoritmos paralelos para o problema do fecho convexo são [3, 12, 13, 17, 19, 20]. Goodrich [14] apresenta técnicas probabilísticas para fecho convexo no modelo BSP. O artigo em [4] trata do método de divisão-e-conquista em cascata como um método para desenvolvimento de algoritmos paralelos. Em [18] temos um artigo sobre algoritmos de divisão-e-conquista no modelo hipercubo.

No desenvolvimento deste trabalho foram feitas buscas nas bases de dados bibliográficos a seguir, e que podem ser úteis para localização de trabalhos correlatos ou não.

- http://theory.lcs.mit.edu/ dmjones/hbp/bibsearch.html
- http://www.lpac.ac.uk/SEL-HPC/Articles/HpcArchive.html
- http://www.cs.ruu.nl/usr/cgi-bin/bib-search/cgi-biblook/Geometry+Literature/rene/geombib/geom

- http://www.mpi-sb.mpg.de/guide/staff/hustadt/world/mpiref.html

Referências

[1] A. Aggarwal, B. Chazelle, L. J. Guibas, C. Ó'Dúnlaing, e C. K. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

[2] A. V. Aho, J. E. Hopcroft, e J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1975.

[3] N. M. Amato, M. T. Goodrich, e E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proc. 35th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 683–694, 1994.

[4] M. J. Atallah, R. Cole, e M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Jounal on Computing*, 18(3):499–532, 1989.

[5] Tom H. Axford. *The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design*, chapter 2. Blackwell, 1992.

[6] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, e S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proc. ICALP'97*, 1997.

[7] T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[8] F. Dehne, X. Deng, P. Dymond, A. Fabri, e A. A. Kokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 27–33, 1995.

[9] F. Dehne, A. Fabri, e C. Kenyon. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing*, pages 586–593, 1994.

[10] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proc. ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[11] F. Dehne e S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. In *Proc. Second Asian Computing Science Conference (ASIAN'96)*, pages 1–10, 1996.

[12] D. J. Evans e S. Mai. Two parallel algorithms for the convex hull problem in a two dimensional space. *Parallel Computing*, 2:313–326, 1985.

[13] M. T. Goodrich. Finding the convex hull of a sorted point set in parallel. *Information Processing Letters*, 26(4):173–179, 1987.

[14] M. T. Goodrich. Randomized fully-scalable BSP techniques for multi-searching and convex hull construction (preliminary version). In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 767–776, 1997.

[15] M. T. Goodrich, M. R. Ghouse, e J. Bright. Sweep methods for parallel computational geometry. *Algorithmica*, 15(2):126–153, 1996.

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

[17] S. N. Maheshwari e P. C. P. Bhatt. Parallel algorithms for the convex hull problems in two dimensions. In *Conf. Analy. Prob. Classes Program. Parallel Comput.*, volume 111 of *Lecture Notes Comput. Sci.*, pages 358–372. Springer-Verlag, 1981.

[18] E. W. Mayr e R. Werchner. Divide-and-conquer algorithms on the hypercube. *Theoretical Computer Science*, 162(2):283–296, 1996.

[19] R. Miller e Q. F. Stout. Efficient parallel convex hull algorithms. *IEEE Trans. Comput.*, C-37(12):1605–1618, 1988.

[20] W. Preilowski, E. Dahlhaus, e G. Wechsung. New parallel algorithms for convex hull and triangulation in 3-dimensional space. In *Mathematical Foundations of Computer Science 1992, 17th International Symposium*, pages 442–450, 1992.

[21] F. C. Preparata e M. I. Shamos. *Computational Geomety: An introduction.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.

[22] J. H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, 1993.

[23] C. Siniolakis. Bulk-synchronous parallel algorithms in computational geometry. Technical Report PRG-TR-10-96, Oxford University, Computing Laboratory, Maio 1996.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.