Algoritmos Paralelos para Fecho Convexo

Emmanuel Kayembe Ilunga

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Orientador: **Prof. Dr. Siang Wun Song**

Durante a elaboração deste trabalho o autor recebeu auxílio financeiro da CAPES.

São Paulo, 30 de março de 2001

Algoritmos Paralelos
para
Fecho Convexo

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Emmanuel Kayembe Ilunga
e aprovada pela comissão julgadora.

São Paulo, 30 de março de 2001.

Banca examinadora:

- Prof. Dr. Siang Wun Song (orientador) - IME-USP
- Prof. Dr. Alfredo Goldman Vel Lejbman - IME-USP
- Prof. Dr. Jorge Stolfi- IC-UNICAMP

A minha família,
pelo apoio durante todos estes anos.

Agradecimentos

Ao Prof. Dr. Siang Wun Song, pela paciência e excelente orientação durante a realização deste trabalho.
Ao Prof. Dr. Alfredo Goldmann Vel Lejbman, pela amizade e por sua colaboração na realização deste trabalho.
Ao Prof. Dr. Maurice Tchuente, por ter me apresentado a área da computação paralela.
Ao meu pai Grégoire Ilunga Tshisaw, *in memoriam*, minha mãe Mutoba Katanga e irmãos, por tudo que fizeram por mim.
Ao Prof. Dr. Kabengele Munanga, pelo apoio moral.
Aos irmãos Lukota Kitoko, Richard Kouakou e Crispim C.M.Calonge, pelo acolhimento nessa terra brasileira e todo apoio.
Aos amigos Ricardo Couto Antunes da Rocha e Marco Aurélio Stefanes, pela ajuda e pela troca de idéias.
Aos professores e funcionários do IME-USP, pela formação e serviços prestados.
À CAPES, pelo apoio financeiro.
Finalmente, a todos que de alguma forma contribuiram para a realização deste trabalho.

Resumo

O principal objetivo de nossa dissertação é de estudar os algoritmos paralelos e de implementar alguns algoritmos probabilísticos para o problema do fecho convexo.

Nosso estudo começa com os algoritmos seqüenciais que podem ser usados na fase de cálculo local de cada processador. Em seguida apresentamos uma classificação dos modelos de computação paralela. Dois destes modelos são usados para estudar os algoritmos paralelos, um deles foi escolhido pelas características teóricas e outro pelas características práticas ligadas a realidade das máquinas atuais.

Por fim, descrevemos e implementamos dois algoritmos probabilísticos incluindo o algoritmo *Quickhull* paralelo na máquina paralela *Parsytec PowerXplorer*.

Abstract

The main goal of this dissertation is to study parallel algorithms for the convex hull problem and implement two parallel algorithms.

Our study starts with sequential algorithms that can be used in the local computing phase in each processor. We then present a classification of some parallel computing models. Two of such models are used to study the parallel algorithms, one of which chosen for its theoretical characteristics and the other for its practical characteristics related to actual parallel machines.

At last we describe the implementation of two probabilistic parallel algorithms including the parallel quickhull in a Parsytec PowerXplorer.

Índice

Agradecimentos **iv**

Resumo **v**

Abstract **vi**

1 Introdução **1**

2 Algoritmos Seqüenciais **6**

2.1 Introdução 6

2.2 Algoritmo Arestas-Extremas 7

2.3 Algoritmo Embrulho-para-Presente 8

2.3.1 Algoritmo 8

2.3.2 Análise do Algoritmo 9

2.4 Algoritmo Quickhull 10

2.4.1 Descrição do Algoritmo 10

2.4.2 Análise de Complexidade. 11

2.5 Algoritmo Mergehull 12

2.5.1 Descrição do Algoritmo 13

2.5.2 Análise do Algoritmo 15

2.6 Algoritmo de Graham 15

2.6.1 Pré-processamento 15

2.6.2 Construção do Fecho Convexo 16
2.6.3 Análise do Algoritmo 17

3 Modelos de Computação Paralela **18**
3.1 Introdução 18
3.2 Classificação dos Modelos de Computação Paralela 20
3.2.1 Apresentação dos Principais Modelos 20
3.2.2 Modelos 22
3.2.3 Classificação 27
3.3 Conclusão 30

4 Algoritmo Paralelo PRAM **31**
4.1 Estratégia de Divisão-e-Conquista 31
4.2 Um Algoritmo de Tempo Constante para Determinação da Tangente Comum Superior 33
4.3 Cálculo do Fecho Convexo 36

5 Algoritmos Paralelos CGM **38**
5.1 Algoritmo CGM Probabilístico 38
5.1.1 Caso $p \leq \sqrt{n}$ 39
5.1.2 Caso $1 \leq p \leq n^{1-\epsilon}$ 43
5.2 Algoritmo Paralelo Determinístico no Modelo CGM 44
5.2.1 Combinando os Fechos Convexos em Paralelo 46
5.2.2 Caracterização do Fecho Superior 48
5.2.3 Cálculo do g_i^* e x_i^* em Paralelo 49
5.2.4 Algoritmo MergeHulls 1 51
5.2.5 Algoritmo MergeHulls 2 52
5.3 Implementações Paralelas 54
5.3.1 Implementação do Algoritmo Paralelo Probabilístico 54
5.3.2 Implementação do Algoritmo Paralelo Determinístico 54

6 Implementações Paralelas **56**
6.1 Características da Máquina 56

6.2 Algoritmo Quickhull Paralelo 57
6.3 Resultados 58

7 Considerações Finais 61

A Implementação 63

Referências Bibliográficas 77

Capítulo 1

Introdução

A computação paralela é necessária 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, por exemplo, na previsão do tempo, na prospecção de petróleo, na dinâmica dos fluidos e na computação gráfica. Algoritmos de geometria computacional, em particular o de fecho convexo, encontram aplicações em diversas áreas como projeto de circuitos VLSI, robótica e computação gráfica, e nos casos práticos a quantidade de dados é grande.

As estratégias usadas em algoritmos seqüenciais para solucionar um problema nem sempre servem para a criação de algoritmos paralelos para o mesmo problema. Na elaboração dos algoritmos deve-se levar em conta também a natureza dos problemas. Alguns são trivialmente paralelizáveis, outros apresentam um paralelismo não tão transparente, precisando de estratégias mais elaboradas na confecção dos algoritmos. Existem alguns em que o *grau de paralelismo*[1] é muito baixo ou igual a um. O desenvolvimento dos algoritmos paralelos exige que consideremos a arquitetura paralela a ser usada, pois esta afeta diretamente o desempenho dos algoritmos.

A elaboração dos algoritmos paralelos exige a utilização de estruturas de dados que sejam ao mesmo tempo simples para serem calculadas eficientemente, e suficientes para responder as consultas.

Aggarwal et al. [2] mostram algoritmos paralelos para resolver alguns problemas de geometria computacional. O problema do fecho convexo (em inglês *convex hull*) 2-dimensional é o problema mais fundamental na área de geometria computacional e

[1] É o número máximo de processadores que podem estar computacionalmente ativos em um dado instante de tempo, durante a execução de um programa.

também o mais estudado [5]. Além disso, esse problema tem muitas aplicações, por exemplo, reconhecimento de padrões [6, 19], processamento de imagens [45], corte e alocação de estoque [25, 26, 46]. De fato, parece ser o primeiro problema de geometria computacional para o qual algoritmos paralelos foram elaborados.

O desenvolvimento dos algoritmos paralelos depende também do modelo de computação paralela. Por exemplo, um algoritmo de ordenação ou de inversão de matrizes não é tratado da mesma maneira sobre o modelo *PRAM* (*Parallel Random Access Machines*) e sobre o modelo *BSP* (*Bulk Synchronous Processors*). No primeiro caso, o número de processadores é ligado ao tamanho do problema e os processadores acessam a memória comum. No modelo *BSP*, o número de processadores é um dado do modelo e os processadores se comunicam através da troca de mensagens.

O modelo para a computação paralela deve atender alguns objetivos. Por um lado, tem que ser bastante detalhado para permitir uma predição exata do tempo de execução dos algoritmos. Por outro lado, tem ser bastante simples para fazer uma análise possível dos algoritmos. Um modelo paralelo deve refletir as restrições das máquinas paralelas atuais (existentes e futuras), mas sem perda de portabilidade dos algoritmos feitos nesse modelo. Outro objetivo que deve ser alcançado é a escalabilidade. Isto é, os algoritmos deverão proporcionar ganhos razoáveis para um grande intervalo de número de processadores.

Muitos modelos de computação paralela foram apresentados nos últimos anos [31]. O modelo PRAM [33] é o modelo mais antigo de computação paralela. Esse modelo não é realístico, pois não considera diversas características de máquinas reais. Mas oferece boas diretrizes em um primeiro aspecto da paralelização de um algoritmo. Ele é conveniente também para a análise de classe de complexidade *NC* [33, 40]. No uso prático, porém, tem igualmente as suposições fortes, por exemplo, ele recomenda somente uma unidade de tempo para comunicação entre os processadores. Algumas variações do modelo *PRAM* foram propostas as quais tentam aliviar essas limitações, mas ainda estão longe do funcionamento de máquinas paralelas reais.

Dados n pontos no plano, consideramos o problema da obtenção do fecho convexo.

Muitos algoritmos paralelos para este problema são descritos para diferentes arquiteturas tais como: CREW PRAM [11], CRCW PRAM [3], hipercubo [38] e MESH [37].

Um dos algoritmos paralelos estudados, nessa dissertação, foi projetado no modelo *PRAM* [33]. A sua complexidade de tempo é $O(\log n)$ usando um total de $O(n \log n)$

operações. Esse algoritmo usa a estratégia de divisão-e-conquista. O sucesso deste método depende da execução eficiente da primeira e da última fases. A primeira fase consiste em dividir o problema em vários subproblemas. A segunda fase é de resolver os subproblemas recursivamente. Na última fase são combinadas as soluções dos subproblemas para obter uma solução do problema original. Esta estratégia é muito utilizada no desenvolvimento de algoritmos seqüenciais.

Recentemente, alguns modelos mais realistícos como *BSP* [47] e *CGM* (*Coarse-Grained Multicomputers*) [15], que levam em conta a comunicação entre os processadores que podem ser um gargalo, têm sido utilizados no desenvolvimento de algoritmos paralelos. Estes modelos são bastante gerais e não consideram a arquitetura específica de interconexão entre os processadores.

O modelo *CGM* consiste de um conjunto de p processadores, cada um com memória local de tamanho $O(\frac{n}{p})$, interconectados através de uma rede de comunicação arbitrária, onde n é o tamanho de entrada do problema. Temos $p << n$ em geral. O desempenho dos algoritmos no modelo *CGM* é medido pelo número de rodadas de comunicação. Os algoritmos nesse modelo consistem da alternância entre uma rodada de computação local e uma rodada de comunicação.

No modelo *CGM*, vários resultados foram propostos para o problema do fecho convexo [13, 14, 15, 17, 18]. Nesses algoritmos, cada processador obtém na primeira etapa o fecho convexo de $\frac{n}{p}$ pontos inicialmente distribuídos em cada processador usando os algoritmos seqüenciais. As etapas seguintes consistem geralmente em fazer a combinação desses fechos convexos.

O algoritmo de Dehne, Fabri e Rau-Chaplin [15] foi o primeiro algoritmo paralelo *CGM* proposto para o fecho convexo de um conjunto de n pontos. Esse algoritmo é determinístico, aplicável para $n \geq p^2$ e requer $O(\log n)$ rodadas de comunicação. A complexidade de tempo desse algoritmo é $O(\frac{T_{sequencial}}{p} + T_s(n,p))$ onde n é o tamanho do problema, p o número de processadores e $T_s(n,p)$ refere-se a complexidade de tempo de ordenação de n pontos armazenados em p processadores. O algoritmo é ótimo se $\frac{T_{sequencial}}{p}$ domina $T_s(n,p)$ ou $T_s(n,p)$ é ótimo. A vantagem dele é de ser simples, determinístico e escalável sobre um grande intervalo dos valores do fator $\frac{n}{p}$. Mas a desvantagem é de possuir um número não constante de rodadas de comunicação.

No ano seguinte, Dehne, Fabri e Kenyon [14] apresentaram um algoritmo paralelo que requer, com alta probabilidade, um número constante de rodadas de comunicação. Esse algoritmo é ótimo e aplicável para $n \geq p^{3+\epsilon}$ onde ϵ é uma constante fixa tal que

$\epsilon > 0$. Ele tem complexidade $(k+2)(\frac{T_1(n)}{p} + O(\frac{n}{p}))$ para o tempo de computação local e $(k+1)(T_{pSum}(p) + T_{compr}(p,n))$ para o tempo de comunicação, onde $k = \lceil \frac{1}{2\epsilon} + \frac{1}{2} \rceil$ é uma constante. $T_1(n)$ denota a complexidade de tempo seqüencial e $T_{pSum}(p)$ é a complexidade paralela para calcular a soma parcial de p números, cada um armazenado em um processador. $T_{compr}(p,n)$ é o tempo para comprimir um subconjunto de dados de tamanho $n' \leq n$ em $p' \leq p$ processadores. A vantagem é que esse algoritmo resolve o problema do fecho convexo para os pontos em qualquer dimensão maior que dois, supondo a distribuição uniforme de pontos. Este algoritmo será apresentado de modo detalhado nesta dissertação.

No mesmo ano, Deng e Gu [17] publicaram dois algoritmos para fecho convexo bidimensional aplicável para $n \geq p^{3+\epsilon}$. Esse algoritmo determinístico requer $O(\log p)$ rodadas de comunicação, enquanto a versão probabilística requer com alta probabilidade $O(1)$ de rodadas de comunicação. A complexidade de tempo desses algoritmos é $O(\frac{n \log n}{p})$. A vantagem é de não ter restrição da distribuição uniforme de conjunto de pontos.

Num outro trabalho recente, Diallo, Ferreira, Rau-Chaplin e Ubéda [18] obtiveram um algoritmo determinístico, que requer somente um número constante de rodadas de comunicação no pior caso. Esse algoritmo, aplicável para $n \geq p^{1+\epsilon}$, é o terceiro estudado neste trabalho. O algoritmo proposto tem complexidade de tempo local $O(\frac{n \log n}{p} + T_s(n,p))$ onde n é o tamanho do problema, p o número de processadores e $T_s(n,p)$ refere a complexidade de tempo de ordenação de n pontos armazenados em p processadores. Este algoritmo é ótimo se $\frac{n \log n}{p}$ domina $T_s(n,p)$ [18].

Zhou, Deng e Dymond, em um trabalho ainda não publicado, propõem um algoritmo paralelo para o fecho convexo de um conjunto de n pontos no plano, de complexidade de tempo local $(O(\frac{n \log n}{p}))$ com o número de rodadas de comunicação ótimo.

O algoritmo seqüencial *Quickhull* foi paralelizado por nós e implementado utilizando a máquina paralela *Parsytec PowerXplorer*. Essa máquina tem 16 processadores, interligados por uma topologia de grade bidimensional. Cada nó é composto de um processador *PowerPC 601* para a computação, um processador *Transputer T805* para a comunicação e uma memória local de 32 *MBytes*. O sistema operacional usado é o *PARIX*. Sua complexidade de tempo da computação local é $O(\frac{n}{p} \log \frac{n}{p})$ com um número constante de rodadas de comunicação.

Na implementação do algoritmo, utilizamos os conjuntos de n pontos segundo

uma distribuição normal. Com esses tipos de dados, conseguimos eliminar quase 97 porcento dos pontos na primeira fase do cálculo local em cada processador. Implementamos também o algoritmo probabilístico de Dehne [14]. Os resultados mostram que as comunicações do algoritmo de Dehne são melhores. O código da implementação do algoritmo *Quickhull* paralelo pode ser encontrado no *Apêndice A*.

No capítulo 2, enunciaremos o problema do fecho convexo estudado neste trabalho. No mesmo capítulo, descreveremos os algoritmos seqüenciais que resolvem o problema com complexidade $O(n \log n)$. Esses algoritmos podem ser usados no modelo de computação *CGM* durante a rodada de computação local e são os melhores conhecidos até hoje.

O capítulo 3 apresenta a classificação dos modelos de computação paralela, destacando as características principais.

O capítulo 4 terá a descrição completa do algoritmo paralelo no modelo *PRAM*. Esse algoritmo é um dos primeiros para resolver o problema do fecho convexo em paralelo.

No capítulo 5, vamos descrever detalhadamente os algoritmos paralelos probabilístico de Dehne [14] e determinístico de [18], repectivamente. Esses algoritmos são escaláveis e facilmente implementáveis em máquinas paralelas existentes.

Nosso capítulo 6 trata da implementação na máquina paralela. Nesse mesmo capítulo apresentamos um algoritmo *Quickhull* paralelo, as características da máquina e o resultado. Implementamos também o algoritmo probabilístico de Dehne.

Por fim, apresentamos no capítulo 7 as considerações finais de nosso trabalho.

Capítulo 2

Algoritmos Seqüenciais

2.1 Introdução

Antes de apresentarmos os algoritmos seqüenciais, vamos definir o problema do fecho convexo.

Definição: *Dado um conjunto* $S = \{p_1, p_2, \cdots, p_n\}$ *de* n *pontos no plano ou num espaço de dimensão qualquer maior que dois, o* **fecho convexo** *de* S*, denotado* $CH(S)$*, é o menor polígono convexo que contém todos os pontos de* S*.*

Seja $\mathbb{E}^d$ um espaço euclidiano d-dimensional. Um conjunto S em $\mathbb{E}^d$ é convexo se $\forall x, y \in S$, o segmento $xy \subseteq S$.

Problema do fecho convexo [39]: *Dado um conjunto* $S = \{p_1, p_2, \cdots, p_n\}$ *de pontos no plano, encontrar* $CH(S)$*, o fecho convexo dos pontos de* S*.*

Existem vários algoritmos seqüenciais para encontrar o fecho convexo de um conjunto S de pontos no plano. Vamos examinar alguns deles, especialmente, aqueles de complexidade $O(n \log n)$ que servirão na computação local do modelo de granularidade grossa *CGM*.

Considere um ponto p no plano, de coordenadas (x, y). A abscissa de p será denotada por $x(p)$ e a ordenada denotada por $y(p)$.

2.2 Algoritmo Arestas-Extremas

O algoritmo [39] identifica arestas do fecho convexo ao invés de vértices extremos. Uma aresta xy é uma aresta êxtrema de um conjunto de pontos S se x e y forem vértices extremos e todos os pontos de S estão sobre ou de um mesmo lado da aresta.

Considere o segmento (a, b) e um ponto c (Figura 2.1). Se a tripla (a, b, c) forma um circuito anti-horário como mostra a figura 2.1, então c é dito à esquerda de (a, b). Analogamente, $c^{'}$ é dito à direita de (a, b) se $(a, b, c^{'})$ forma um circuito horário.

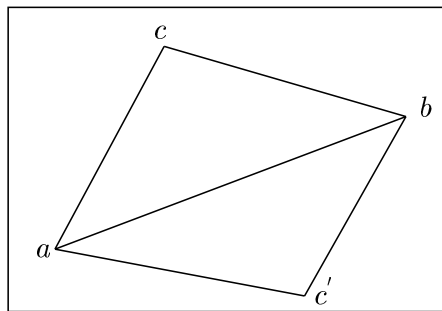

Figura 2.1: O ponto c está à esquerda do segmento (a, b) e o ponto $c^{'}$ está à direita do segmento (a, b).

Algoritmo *Arestas-Extremas*
Entrada: Um conjunto finito $S = \{p_1, ..., p_n\}$ de pontos no plano.
Saída: As arestas extremas de $CH(S)$
for cada i **do**
  for cada $j, j \neq i$ **do**
    for cada $k, k \neq i, k \neq j$ **do**
      if p_k não está à esquerda ou sobre (p_i, p_j)
        then p_i, p_j não é aresta de $CH(S)$.

O algoritmo acima tem complexidade de tempo $O(n^3)$, pois existem três laços encaixados e cada um desses laços fará $O(n)$ iterações.

2.3 Algoritmo Embrulho-para-Presente

O algoritmo *Embrulho-para-Presente* (*Gift-Wrapping*) também é conhecido como algoritmo de Jarvis [32, 39, 43] e pode ser visto como uma variação do algoritmo *Arestas-Extremas* [39]. Ele usa a aresta extrema mais recentemente encontrada para achar a próxima aresta. A idéia funciona, pois sabemos que no fecho convexo, uma aresta está conectada a outra, que por sua vez está ligada a uma outra aresta e assim por diante. Para achar a próxima aresta extrema, testamos $O(n)$ candidatas para cada aresta extrema encontrada. Logo a complexidade do algoritmo é $O(n^2)$, que reduz por um fator de n a complexidade do algoritmo *Arestas-Extremas*.

2.3.1 Algoritmo

Seja S um conjunto de n pontos no plano. Suponhamos que não existem três pontos em S que sejam colineares. Seja a uma aresta de $CH(S)$ ligando o ponto p e o ponto q onde $x(p) < x(q)$. A idéia chave do algoritmo [32] é achar o ponto r tal que o ângulo polar θ_r entre rq e a reta passando pela aresta a seja mínimo. Essa observação é do algoritmo *Arestas-Extremas* que busca colocar todos pontos de mesmo lado, como mostra Figura 2.2. No início não temos nenhuma aresta de

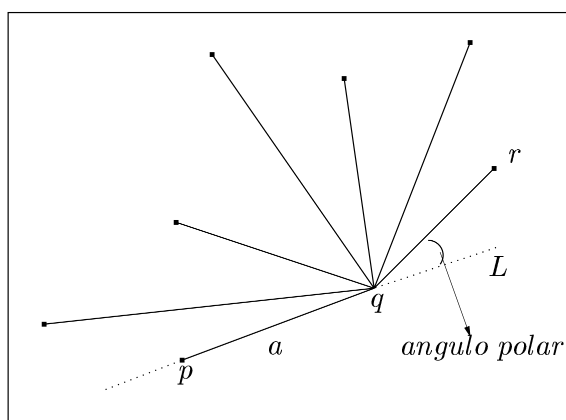

Figura 2.2: A aresta qr e a reta passando pela aresta a tem o menor ângulo polar.

$CH(S)$. Basta escolher o mais baixo ponto p_0 em S (isto é, com menor $y(p_0)$). A Figura 2.3 mostra como encontrar o ponto p_1 em S de menor ângulo polar (sentido anti-horário) em relação a p_0 e a reta horizontal.

Abaixo encontra-se o algoritmo *Embrulho-Para-Presente.*

p_2 p_3 p_4 p_1 L p_0

Figura 2.3: As operações feitas pelo algoritmo *Embrulho-para-Presente.*

Algoritmo *Embrulho-Para-Presente*

Entrada: Um conjunto finito $S = \{p_1, ..., p_n\}$ de pontos no plano.

Saída: As arestas extremas de $CH(S)$ tais que elas aparecem na fronteira de $CH(S)$ no sentido anti-horário.

1. Encontre o ponto em S de menor ordenada.

2. Seja i_0 o índice desse ponto. $i \leftarrow i_0$.

3. Seja L a reta horizontal passando por p_{i_0}.

4. repeat

- **a. for** cada $j, j \neq i$ **do**
  - Calcule o ângulo polar θ_j do ponto p_j em relação ao ponto p_i e a reta L.
- **b.** Seja k o índice do ponto p_k tal que $\theta_k = min_j \theta_j$
- **c. Output** $p_i p_k$ como uma aresta de $CH(S)$.
- **d.** $L \leftarrow$ reta passando por $p_i p_k$.
- **e.** $i \leftarrow k$.

until $i = i_0$.

2.3.2 Análise do Algoritmo

Seja h o número de arestas extremas de $CH(S)$. Então, no pior caso, o algoritmo tem complexidade de tempo $O(hn) = O(n^2)$. Se h é pequeno, por exemplo, $h = O(\log n)$, a complexidade fica $O(n \log n)$.

2.4 Algoritmo Quickhull

O problema de ordenação tem sido uma constante fonte de inspiração para o desenvolvimento de algoritmos do fecho convexo. O algoritmo que vamos ver nesta seção, com algumas variações, foi proposto independentemente por vários autores quase ao mesmo tempo [10, 20, 30]. Devido a semelhança com o algoritmo *Quicksort*, o algoritmo que vamos descrever foi batizado de *Quickhull* [24, 39, 43].

2.4.1 Descrição do Algoritmo

A idéia básica é tentar descartar uma grande parte dos pontos que estão no interior do fecho convexo. O trabalho então concentra nos pontos que estão próximos à fronteira. No início, o algoritmo busca os pontos extremos que estão na posição mais acima, mais abaixo, mais à esquerda e mais à direita. No caso de empate, devem ser escolhidos os pontos com menor abscissa ou ordenada. Note que os pontos no interior do quadrilátero são dispensáveis para a construção do fecho convexo, como mostra Figura 2.4. Para achar o fecho convexo, devemos concatenar os quatro subproblemas correspondentes às regiões triangulares que resultam após eliminar os pontos interiores ao quadrilátero.

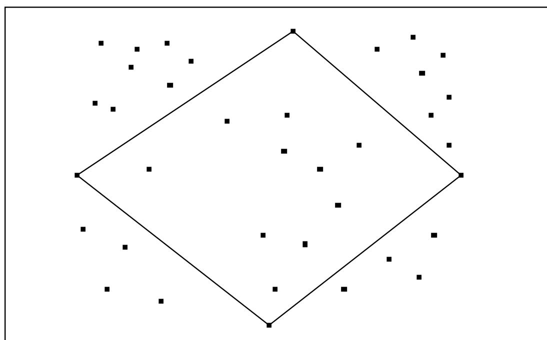

Figura 2.4: O algoritmo descarta os pontos que se encontram no quadrilátero formado pelos quatro pontos extremos.

Cada um dos quatro triângulos tem dois pontos que certamente estão no fecho convexo. O algoritmo resolve recursivamente cada um dos quatro subproblemas obtidos. Para isso, sejam a e b os pontos extremos do conjunto que formam a base do triângulo. O algoritmo encontra um ponto c mais distante do segmento $\overline{ab}$ (que es-

tará no fecho convexo), descarta os pontos que estão no interior do triângulo formado pelos pontos a, b e c e particiona o problema em dois subproblemas idênticos menores. Esses dois subproblemas são resolvidos recursivamente. A Figura 2.5 mostra os pontos a, b e c.

O algoritmo está descrito abaixo.

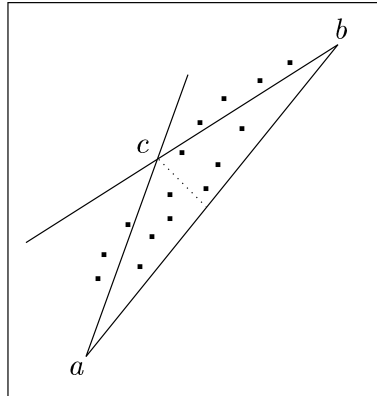

Figura 2.5: O *Quichull* descarta os pontos que estão no interior do triângulo formado pelos pontos a, b e c.

Algoritmo *Quickhull*

Entrada: Dois pontos distintos a e b e um conjunto S de pontos no plano tal que todo ponto de S está à esquerda da reta passando por a e b.

Saída: Fecho convexo $CH(S)$ de S.

1. if $S = \emptyset$ **then return** $\overline{ab}$.

2. else

- **a.** $c \leftarrow$ ponto com distância máxima ao segmento $\overline{ab}$.
- **b.** $A \leftarrow$ os pontos de S à esquerda do segmento $\overline{ac}$.
- **c.** $B \leftarrow$ os pontos de S à esquerda do segmento $\overline{cb}$.
- **d. return** Quickhull(a, c, A) concatenado com Quickhull(c, b, B).

2.4.2 Análise de Complexidade.

Na descrição do algoritmo acima, temos uma fase do pré-processamento que consiste em achar os quatro pontos extremos. Essa fase custa $O(n)$ operações elementa-

res, onde n é o número de pontos.

A análise de complexidade é semelhante à do Quicksort, depende do número de elementos nos conjuntos A e B. Seja $\alpha = |A|$ e $\beta = |B|$. Se $T(n)$ denota o tempo para achar o fecho convexo de n pontos. A complexidade é dada por:

$$T(n) \leq T(\alpha) + T(\beta) + dn,$$

onde d é uma constante. O termo dn corresponde ao número de operações gastas pelo algoritmo *Quickhull* para encontrar o ponto c e construir os conjuntos A e B.

Se $\alpha = \lfloor n/2 \rfloor$ e $\beta = \lceil n/2 \rceil$, a metade dos pontos estão em A e outra metade em B, teremos que:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + dn.$$

Isto é uma relação de recorrência cuja a solução é:

$$T(n) = O(n \log n).$$

No pior caso, a partição deixa $n-1$ pontos de um mesmo lado e $\alpha = 1$ e $\beta = n-1$. A equação de recorrência fica:

$$T(n) \leq T(n-1) + dn.$$

Ao resolver essa recorrência chegamos a:

$$T(n) \leq d + d2 + \cdots + d(n-2) + d(n-1) + dn = O(n^2).$$

2.5 Algoritmo Mergehull

O algoritmo *Mergehull* é chamado assim por causa da sua similaridade com o algoritmo *MergeSort*. Ele usa a técnica de divisão-e-conquista [12] para encontrar o fecho convexo de um conjunto de n pontos. A estratégia de divisão-e-conquista é composta de três passos principais:

- **Divisão:** O problema é dividido em um número de subproblemas de tamanho menor.
- **Conquista:** Os subprolemas são resolvidos recursivamente. Quando o subproblema tem o tamanho suficientemente pequeno, ele é resolvido de maneira direta.
- **Combinação:** As soluções dos subproblemas são combinadas para obter uma solução do problema original.

Em Geometria Computacional, muitos problemas são resolvidos, seqüencialmente, através do paradigma de divisão-e-conquista. Um destes problemas é a 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 resolução seqüencial deste problema, as fases de divisão e conquista são simples de se obter, a fase de combinação é a mais trabalhosa.

2.5.1 Descrição do Algoritmo

Antes de aplicarmos o algoritmo a um conjunto S de pontos dados, a primeira fase será de fazer um certo pré-processamento que consiste em ordenar os pontos dados em relação as suas abscissas.

A próxima fase do *Mergehull* consiste em dividir o conjunto de pontos em dois subconjuntos A e B de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, pela abscissa (em caso de empate considera-se a ordenada). Isso significa dividir o problema em dois subproblemas de tamanho menor. Na fase seguinte são resolvidos recursivamente os subproblemas independentemente. A última fase permite combinar as soluções dos subproblemas para obter a solução do problema inicial.

Para que o fecho convexo seja construído em tempo $O(n \log n)$, deve-se ter certeza que a última fase seja executada em tempo $O(n)$. A idéia é escolher o ponto mais à direita de A e mais à esquerda de B, chamaremos de pontos u_i e v_j. Para achar a aresta do fecho "de baixo" escorregue o segmento $u_i v_j$ para baixo, de um lado e depois do outro, até que as extremidades do segmento tangente procurado sejam encontradas, como mostra Figura 2.6. A aresta de cima é encontrada da maneira análoga. Essa idéia foi sugerida por Preparata e Hong [42].

Temos a seguir os algoritmos *Acha Tangente-Inferior e Mergehull.*

Algoritmo: *Acha Tangente-Inferior*
Entrada: Dois fechos convexos $CH(A)$ e $CH(B)$.
Saída: O segmento 'tangente' inferior a $CH(A)$ e $CH(B)$.
1. $v_i \leftarrow$ vértice mais à direita de $CH(A)$.
2. $u_j \leftarrow$ vértice mais à esquerda de $CH(B)$.
3. **while** $T = u_i v_j$ não é tangente inferior de $CH(A)$ e $CH(B)$ **do**
  a. **while** T não é tangente inferior de $CH(B)$ **do**
    $j \leftarrow j + 1$.

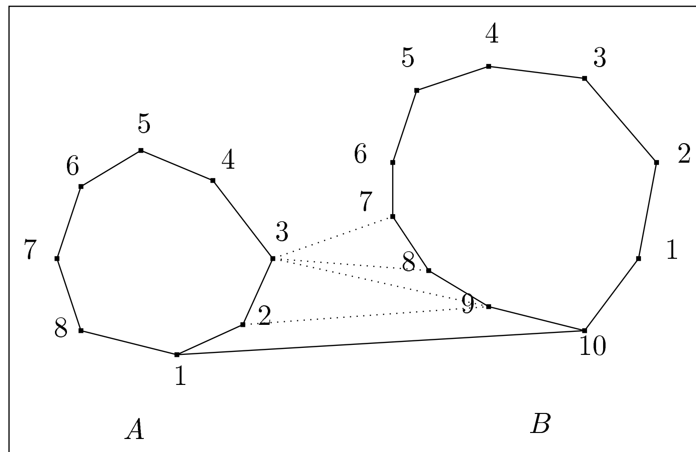

Figura 2.6: O algoritmo Acha Tangente-Inferior começa com $(u_i, v_j) = (3, 7)$ e termina com $(u_i, v_j) = (1, 10)$.

b. while T não é tangente inferior de $CH(A)$ **do**
$i \leftarrow i - 1$.
4. return T.

O algoritmo supõe que os pontos v_i de $CH(A)$ e os pontos u_j de $CH(B)$ estão em sentido anti-horário. A aritmética envolvendo os índices é módulo o tamanho do respectivo conjunto A ou B.

No algoritmo acima, dizer que "$T = u_i v_j$" não é tangente inferior de $CH(A)$ é equivalente a u_{i-1} está à direita do segmento $u_i v_j$. Da maneira análoga, "$T = u_i v_j$" não é tangente inferior de $CH(B)$ é equivalente a v_{j+1} está à direita do segmento $u_i v_j$.

Algoritmo: *Mergehull*
Entrada: Um conjunto finito $S = \{p_1, ..., p_n\}$ de pontos no plano.
(Suponha que estão ordenados pela abscissa.)
Saída: O fecho convexo de S.
1. if $|S| \leq k_0$, onde k_0 é uma constante pequena, construa o fecho convexo diretamente por algum método e **return**.
2. Particione o conjunto S em conjuntos A com $\lfloor n/2 \rfloor$ pontos e B com $\lceil n/2 \rceil$.
3. Recursivamente construa o fecho convexo de A e B.
4. Construa o fecho convexo de $A \cup B$.

2.5.2 Análise do Algoritmo

O pré-processamento do algoritmo *Mergehull* consiste em ordenar os pontos do conjunto S. Se usarmos o algoritmo *Heapsort* ou *Mergesort* [12], a complexidade de tempo dessa fase é $O(n \log n)$. Na fase de combinação, os laços são executados $O(n)$ vezes. Isto significa que as tangentes inferior e superior podem ser construídas em tempo $O(n)$. Então, a complexidade de tempo da fase de combinação é $O(n)$.

Se $T(n)$ é a complexidade de tempo do algoritmo *Mergehull*, ao resolver uma instância de tamanho n, teremos:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn = O(n \log n),$$

onde c é uma constante.

Portanto, a complexidade de tempo do algoritmo *Mergehull* é $O(n \log n)$.

2.6 Algoritmo de Graham

O construção do fecho convexo de um conjunto S de pontos pelo algoritmo de Graham [29] é feita em duas fases. Primeiramente, um pré-processamento é realizado para selecionar um ponto p_0 em S de ordenada mínima e ordenar os pontos restantes pelo ângulo ao redor de p_0. A segunda fase do algoritmo consiste em processar iterativamente os pontos, construido uma seqüência de fechos convexos que converge para $CH(S)$.

2.6.1 Pré-processamento

A fase de pré-processamento consiste em achar um ponto p_0 de S. O ponto p_0 vai ser o ponto em S com a menor ordenada. Em caso de empate, escolheremos entre os pontos com a menor ordenada, o ponto com a maior abscissa.

O passo seguinte de pré-processamento consiste em ordenar os pontos de $S \backslash \{p_0\}$ em relação ao ângulo polar que estes pontos formam ao redor do ponto p_0. Se dois pontos têm o mesmo ângulo polar (colineares), descarta o ponto mais próximo de p_0. Veja a figura 2.7.

2.6.2 Construção do Fecho Convexo

A segunda fase constrói o fecho convexo. Nesta fase, os pontos são examinados na ordem que estes foram colocados pelo pré-processamento. Em cada iteração, os pontos de fecho convexo já encontrados estão em uma pilha identificada pela variável *topo*. Para cada ponto p_i, se o ângulo entre $T_{topo-1}T_{topo}$ e $T_{topo}p_i$ for menor que 180^0, empilha p_i senão desempilha. Note que alguns pontos que eram pontos do fecho convexo anterior, podem não ser pontos do fecho convexo corrente. No final da última iteração do algoritmo, a pilha contém os pontos do fecho convexo.

A seguir o algoritmo de Graham.

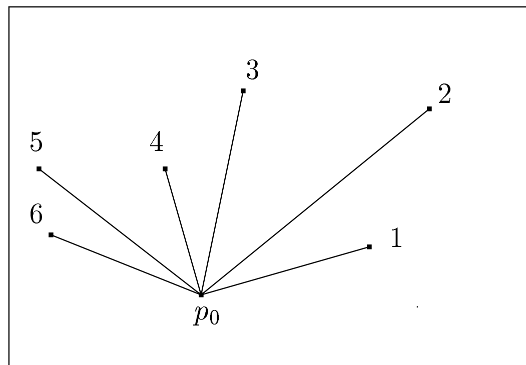

Figura 2.7: Os pontos ordenados conforme o ângulo polar que estes formam ao redor do ponto p_0.

Algoritmo: *Graham*
Entrada: Um conjunto finito S de pontos no plano $(n > 3)$.
Saída: O fecho convexo de S.

1. Encontre p_0 o ponto em S com ordenada mínima, em caso de empate tome o ponto mais à direita.
2. Ordene pontos pelo ângulo (elimine os empates escolhendo o ponto mais distante). Sejam $p_1, \cdots, p_{n-1}$ os pontos de S ordenados.
3. $T \leftarrow \emptyset$
4. Empilhe p_0, p_1, p_2 na pilha T.
5. $i \leftarrow 3$.
6. **while** $i < n$ **do**
  a. **if** p_i está à esquerda de (p_{topo-1}, p_{topo}) **then**

Empilhe p_i e $i \leftarrow i + 1$

b. else Desempilhe p_{topo}.

7. return T.

2.6.3 Análise do Algoritmo

A análise de complexidade de tempo depende das fases do algoritmo. O primeiro passo do algoritmo, na fase de pré-processamento, pode ser feito em tempo $O(n)$. O custo total da fase de pré-processamento é dominado pelo custo da ordenação dos pontos que é $O(n \log n)$ no segundo passo.

A segunda fase do algoritmo vai da linha 3 até 7. As linhas 3, 4 e 5 são executadas em tempo constante ($O(1)$). O número total de execuções do comando while da linha 6 será $O(n)$. Portanto, o número de operações Empilha e Desempilha na execução do algoritmo é $O(n)$.

Assim, a complexidade de tempo total do algoritmo é $O(n \log n)$.

Capítulo 3

Modelos de Computação Paralela

Neste capítulo apresentamos uma visão geral sobre alguns modelos de computação paralela. Os modelos *PRAM* e *CGM* vão ser usados no nosso trabalho para o desenvolvimento dos algoritmos paralelos que resolvem o problema do fecho convexo. Esse capítulo é baseado no capítulo 20 do livro de Capa Rumeur [1].

3.1 Introdução

Um dos principais objetivos deste capítulo é expor os diferentes modelos de computação paralela, do *PRAM* ao modelo *CGM*. A diversidade dos modelos de computação paralela apresentados na literatura é tal que uma apresentação exaustiva é hoje quase impossível. Vamos apresentar as grandes classes de modelos e focar nossa apresentação ao redor de uma classificação. A classificação que apresentaremos será eficaz pois continua compatível com os outros modelos da literatura.

O segundo objetivo desse capítulo é mostrar a influência do modelo de computação paralela no desenvolvimento dos algoritmos paralelos. Por exemplo, um algoritmo de ordenação ou de inversão de matrizes não é tratado da mesma maneira sobre o modelo *PRAM* e sobre o modelo *BSP*. No primeiro caso, o número de processadores é ligado ao tamanho do problema e os processadores acessam uma memória comum. No modelo *BSP*, o número de processadores é um dado do modelo e os processadores se comunicam por troca de mensagens.

O destaque vai também na adequação do modelo com a realidade das máquinas paralelas atuais. É conveniente, entretanto, de não se focalizar sobre esse parâmetro. Por um lado a tecnologia evolui tão rapidamente, que não se pode afirmar que as

máquinas atuais são representativas das máquinas da próxima geração. A definição de novos modelos pode ser vista de fato como uma maneira de determinar os funcionamentos dos algoritmos paralelos no futuro. Como no caso de modelo de von Neumann para a computação seqüencial, o modelo de computação paralela deve estabelecer uma ponte entre o *software* e o *hardware* [31, 34]. Por outro lado, existem outros parâmetros que vão determinar a importância de um modelo tais como:

- a portabilidade;
- a escalabilidade;
- a capacidade de desenvolver uma teoria da complexidade;
- a facilidade de utilização;
- a ampla aplicabilidade com respeito às máquinas existentes e futuras.

Cada um desses critérios tem importância variada conforme a utilização feita do modelo. A teoria de complexidade prefere certamente o modelo *PRAM* ao modelo mais próximo das máquinas, porém mais dificilmente tratável. Contrariamente, um programador de uma aplicação específica sobre uma categoria de máquinas específica usaria um modelo correspondente a todas as características da máquina tais como entrada/saída, cache, etc.

O custo de tempo é um outro fator que pode influenciar uma possível classificação linear de modelos. Por exemplo, para resolver os problemas como a predição meteorológica ou o cálculo de trajetória de uma sonda sobre Marte, é importante se aproximar da máquina. Assim, adaptar-se-ia um produto de matrizes à estrutura da máquina para executá-lo o mais rapidamente possível. Entretanto, se a pergunta sobre a complexidade do produto de matrizes for levada em conta, uma tal aproximação não permite resposta satisfatória, pois uma máquina é muito pouco estável no tempo para que possa desenvolver uma teoria da complexidade sobre essas características. Esta é uma das razões pelas quais em teoria da complexidade o modelo *PRAM* é o melhor adaptado.

Muitos modelos de computação paralela foram apresentados nos últimos anos. A seção 3.2 descreve e classifica os modelos de computação paralela. Nessa seção, vamos falar do modelo *PRAM* [33] que vai ser usado para desenvolver um algoritmo paralelo para o problema do fecho convexo. Nessa mesma seção veremos um outro

modelo mais recente e realístico, o *CGM* [13, 14, 15, 16]. Esse modelo servirá para o desenvolvimento de outros algoritmos paralelos de nosso trabalho. A seção 3.3 conclui o capítulo com as perspectivas dos modelos de paralelismo.

3.2 Classificação dos Modelos de Computação Paralela

O objetivo dessa seção não é listar o conjunto de modelos de computação paralela apresentado na literatura nesses últimos anos. Procura-se concentrar sobre alguns principais modelos para compreender as suas características mais importantes.

3.2.1 Apresentação dos Principais Modelos

Por principais modelos, entendemos os modelos cujo impacto sobre a análise dos algoritmos é considerável em termos de número de publicações usando esses modelos como base da descrição e do estudo das suas complexidades. Todos esses modelos têm vantagens e desvantagens, as quais discutiremos. Muitos modelos são, aliás, as variações ao redor de modelos descritos abaixo. Na subseção 3.2.3, classificaremos os modelos apresentando suas principais propriedades.

Modelo PRAM

O modelo *PRAM* é historicamente o primeiro a aparecer. Ele corresponde a visão de uma máquina paralela como um conjunto de máquinas seqüenciais acessando a uma memória comum.

Da maneira formal, no modelo *PRAM* estão disponíveis um conjunto arbitrariamente grande de processadores, cada um possuindo um conjunto finito de registradores e tendo acesso a memória comum, de tamanho também arbitrário. O número de processadores e o tamanho da memória são os parâmetros do modelo e a complexidade dos problemas será expressa em função desses parâmetros. Em geral o número de processadores é considerado da mesma grandeza que o tamanho da memória. Uma máquina *PRAM* de n processadores pode potencialmente executar n instruções simultanemente. Essas instruções podem ser idênticas ou não em cada processador.

Todos os processadores acessam a memória comum em tempo constante, pois não há o custo de comunicação.

O modelo *PRAM* foi usado para o estudo da complexidade paralela dos problemas fundamentais. Esse modelo permite a definição formal de classes de complexidade como a classe *NC*, correspondendo ao conjunto de problemas do tamanho n solucionáveis em tempo polilogarítmico sobre um número polinomial de processadores. Uma das conjecturas mais famosas de algoritmos paralelos é de saber se $P = NC$, o que significa, se o conjunto dos problemas solucionáveis seqüenciamente em tempo polinomial podem ser resolvidos em tempo polilogarítmico sobre uma máquina *PRAM* com um número polinomial de processadores. Note que o contrário é trivial pela simples simulação de n processadores em um único processador: cada etapa paralela necessitando $O(n)$ etapas seqüenciais.

Modelo de topologias explícitas

O modelo *PRAM* deveria ser um padrão para a computação paralela, mas apesar da sua importância no desenvolvimento de algoritmos, ele não consegue atingir a exata noção do paralelismo. A principal desvantagem desse modelo é de não considerar a estrutura das máquinas disponíveis. Hoje, a maioria das máquinas tem memória distribuída, uma estrutura bem diferente do *PRAM*. Um outro problema sério do modelo *PRAM* é ligado à memória global e à hipótese de leitura/escrita em tempo constante, que não correspondem às máquinas existentes. O acesso à memória local é mais rápido que o acesso à memória do processador vizinho, que por sua vez é menos caro que o acesso à memória de um processador distante (não conectado diretamente).

Foram introduzidos novos modelos, com topologias explícitas tais como grade, hipercubo, etc. As arquiteturas *MIMD* (*Multiple Instructions Multiple Data*) e *SIMD* (*Single Instruction Multiple Data*) levaram a dois tipos de modelos: os modelos de granularidade fina modelando as máquinas *SIMD*, em particular os *Connection Machines CM1* e *CM2* [23], e os modelos de granularidade grossa modelando as máquinas tais como iPC'S de Intel, ou as máquinas a base de *Transputers* [4] na Europa.

Esses modelos permitiram o desenvolvimento de um número grande de algoritmos. Emergiram os paradigmas de programação paralela tais como pipeline, partição de dados, balanceamento de carga e agrupamento de mensagens. Também foram colocadas em evidência técnicas de programação complexas e problemas fundamentais

ainda não resolvidos.

Modelo sem topologias

Dois fatores influenciaram o comportamento dos algoritmos em relação aos modelos de topologia explícita. Por um lado, as comunicações das máquinas paralelas modernas tendem dissimular a distância entre os processadores que se comunicam. Isto é, dois processadores vizinhos ou distantes não modificam, na teoria, o custo da comunicação. Por outro lado, o paradigma de troca de mensagem compete com outros paradigmas, como a memória virtualmente compartilhada, na qual a topologia não é mais um critério a considerar na concepção de um algoritmo. Nenhuma topologia emergiu como candidata universal de rede de interconexão de processadores de uma máquina paralela.

As observações acima fazem aparecer os modelos que não consideram a topologia, ou seja, a topologia se dá de maneira implícita através de outros parâmetros. Um dos modelos mais conhecidos é o modelo *BSP* [47]. Nesse modelo, a máquina paralela é considerada como um conjunto de p processadores que se comunicam, dispondo de um processo de sincronização. Os algoritmos no modelo *BSP* consistem de uma seqüência de superpassos. Em cada superpasso temos duas fases: uma fase de computação local e uma fase de comunicação. A cada final de computação local, cada processador efetua as trocas de mensagens com os demais processadores. Recebendo informações necessárias para a computação no próximo superpasso. Esse modelo não considera a topologia de rede, só nos custos de comunicação e sincronização que a topologia pode ficar aparente.

Os modelos sem topologias tais como *BSP* e *CGM* (uma variante do *BSP*, a ser visto em seguinte) fizeram um grande sucesso na área de computação paralela. O fato de não considerar a topologia faz com que a análise dos algoritmos tenha um resultado considerável. A portabilidade dos algoritmos é um outro fator positivo. A riqueza desses modelos se dá também através das simulações formais de outros modelos, como o *PRAM*.

3.2.2 Modelos

Agora vamos focalizar nossa atenção sobre os modelos seguintes: *PRAM*, *BSP*, *CGM* e *G-RAM* de granularidade fina ou de granularidade grossa. Esses modelos

fixam as principais características dos modelos do paralelismo.

Modelo PRAM

No modelo de memória compartilhada existem dois modos básicos de operação. De um lado, o modo chamado *síncrono*, onde todos processadores processam sincronamente sob o controle de um mesmo relógio. O nome padrão para o modelo de memória compartilhada síncrono é modelo de máquina paralela de acesso aleatório ou *PRAM*. Por outro lado, há o modelo chamado *assíncrono*, onde cada processador opera sob relógios separados. Nesse modo de operação, os pontos de sincronização apropriados são de responsabilidade do programador, sempre que necessários. A figura 3.1 mostra uma visão geral do modelo de memória compartilhada.

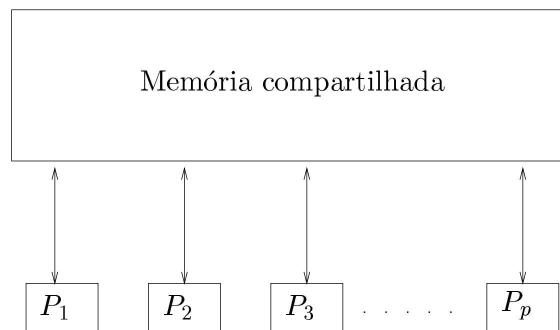

Figura 3.1: O modelo de memória compartilhada.

O modelo *PRAM* é o mais usado para computação paralela. Apesar de ser um modelo muito importante e usado para o desenvolvimento de algoritmos, não é um modelo realístico e não é adequado na prática. Esse modelo oferece uma excelente plataforma para estudar o paralelismo, uma vez que se abstrai de detalhes de implementação como acesso a dados. Ele usa o modo de comunicação síncrono, ou seja os processadores executam as instruções com um mesmo relógio. Cada processador é uma *RAM* seqüencial. Os processadores podem executar diferentes instruções em conjuntos de dados diferentes.

Como a memória é compartilhada, os processadores trabalham concorrentemente. Pode ocorrer de dois ou mais processadores tentarem acessar (para leitura ou escrita) simultaneamente uma mesma posição de memória. O acesso é feito das seguintes formas:

- **EREW** (*Exclusive Read, Exclusive Write*). Não é permitido o acesso concorrente para leitura ou para escrita em uma mesma posição de memória.
- **CREW** (*Concurrent Read, Exclusive Write*). O acesso é permitido para leitura simultânea de uma mesma posição de memória por vários processadores.
- **CRCW** (*Concurrent Read, Concurrent Write*). A leitura e a escrita concorrente são permitidas na mesma posição de memória. Para administrar os possíveis conflitos temos os critérios seguintes:
  - modelo **CRCW** com escrita **arbitrária**: um dos processadores deve ter sucesso na escrita, não importando qual deles.
  - modelo **CRCW** com escrita **comum**: é necessário que todos os processadores em conflito escrevam o mesmo valor.
  - modelo **CRCW** com escrita **prioritária**: o processador com índice menor tem prioridade para escrever.

Modelo G-RAM

O modelo *G-RAM* é um modelo para as máquinas paralelas à memória distribuída. A rede de interconexão da máquina paralela é representada por um grafo G, onde os vértices são os processadores e as arestas representam os canais de comunicação. Nesse modelo existem duas categorias principais. Por um lado, há o modelo de granularidade grossa, que tem dois parâmetros distintos: o número de processadores p e o tamanho do problema n. Por outro lado, o modelo *G-RAM* de granularidade fina supõe-se que a identidade desses dois parâmetros, pelo menos uma relação espécifica em termo de ordem de grandeza. Geralmente, supõe-se que a máquina tem $\theta(n)$ processadores. Como no modelo *PRAM*, o número de processadores é arbitrariamente grande no modelo *G-RAM* de granularidade fina.

Nos dois casos, o modelo é síncrono. No modelo de granularidade fina, cada etapa consiste de uma computação local e de trocas de dados com outros processadores. Como para o *PRAM*, este modelo pode distinguir diferentes hipóteses conforme a capacidade dos processadores de se comunicar, simultaneamente ou não, com vários vizinhos. No modelo *G-RAM* de granularidade grossa, cada processador efetua um grande número de cálculos locais antes de agrupar os dados a enviar durante a fase de comunicação.

Não há possibilidade de comunicações distantes e/ou globais nos dois tipos de granularidade. Assim, se um dado deve ir para um processador não vizinho, esse dado deve passar pelos processadores ou caminhos intermediários. Do mesmo modo, se um processador deve mandar uma mensagem a um conjunto de outros processadores, todos os processadores deverão participar. As comunicações necessitarão de várias etapas[1]. O número de etapas depende fortemente da topologia de rede e do grafo G.

Modelo BSP

O modelo *BSP*, proposto por L. G. Valiant [47], é um dos principais modelos realísticos, que busca abstrair a topologia de rede e apresentar uma descrição através de parâmetros numéricos. Ele é um modelo de propósito geral e um candidato a se tornar o padrão para a computação paralela, com resultados eficientes tanto para o projeto quanto para execução prática de algoritmos.

A máquina *BSP* consiste de um conjunto de p processadores com memória local. Os processadores se comunicam através de algum meio de interconexão, controlados por um *roteador* com facilidade de sincronização periódica global.

Um algoritmo no modelo BSP consiste de uma seqüência de superpassos (*supersteps*). Nestes, os processadores operam independentemente realizando cálculos locais e trocas de mensagens através de operações de envio e recebimento. Uma mensagem enviada é recebida no próximo superpasso. No final de um superpasso, uma barreira de sincronização é realizada. Os custos de comunicação dependem de dois parâmetros, a saber:

- L: a latência, que é o tempo máximo de um superpasso;
- g: que descreve a vazão (*throughput*) do roteador, medida como a razão entre o número de operações de computação local por segundo pelo número de palavras transmitidas pelo roteador por segundo (computação / comunicação).

Se um processador envia h dados e recebe h' durante a fase de comunicação, o custo dessa comunicação será de $L + \max(h, h')g$. Então, o custo total da fase de comunicação é $\max_{i=1,\cdots,p}\{L + \max(h_i, h'_i)g\}$, onde h_i e h'_i representam os números de dados enviados e recebidos, respectivamente. O modelo *BSP* permite que o valor de L seja

[1] Falaremos sobre isso na seção 3.2.3.

controlado pelo programa, mesmo em tempo de execução. As restrições de *software* e de *hardware* devem ser consideradas na escolha de L.

A simulação de algoritmos *PRAM* no modelo *BSP* foi também discutida por L. G. Valiant [47].

Modelo CGM

O modelo *CGM* para computadores paralelos com memória distribuída foi proposto por F. Dehne [13, 14, 15] com a inspiração no modelo *BSP* [47]. Ele privilegia a computação local, minimizando as operações globais. Nesse modelo, os processadores possuem uma memória local de tamanho $O(n/p)$, onde n é o tamanho da entrada e p é o número de processadores. Em geral, $p << n$. Veja a figura 3.2. A preocupação é aumentar a *granularidade* das máquinas paralelas. Isto significa aumentar a quan-

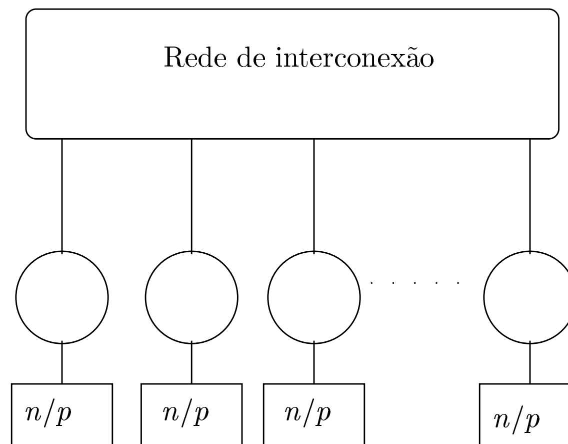

Figura 3.2: O modelo *CGM* de memória distribuída.

tidade de operações locais executadas por um processador sem se comunicar com outros. Cada processador é conectado por um roteador que pode trocar mensagens ponto a ponto.

Um algoritmo em *CGM* consiste da alternância entre fases de computação local e rodadas de comunicação separadas por uma barreira de sincronização. Seja R o número de rodadas, o esquema genérico é:

For $i = 1$ **to** R **do**
{

(fase de computação local)$_i$
(rodada de comunicação)$_i$
}

Uma fase de computação local mais uma rodada de comunicação é equivalente a um superpasso no modelo *BSP*. Nessa rodada, usualmente devemos utilizar o melhor algoritmo seqüencial em cada processador e processar os seus dados localmente.

Em uma rodada de comunicação, cada processador pode enviar e receber no máximo um total de $O(n/p)$ dados de outros processadores. Além disso, o modelo *CGM* exige que todos os dados enviados de um processador para outro em uma fase de comunicação sejam empacotados em uma mensagem.

O custo de comunicação no modelo *CGM* é medido pelo número de rodadas de comunicação. Vários algoritmos *CGM* conhecidos para problemas geométricos necessitam de um número de rodadas de comunicação constante ou $O(\log p)$ [18]. O modelo *CGM* é particularmente adequado para máquinas paralelas atuais, onde a velocidade da computação local é consideravelmente maior que a velocidade de comunicação entre processadores.

As implementações dos algoritmos neste modelo, nas máquinas atualmente disponíveis, se comportam bem e os *speedups* exibidos são similares àqueles previstos em suas análises. Assim, o objetivo do modelo *CGM* é construir algoritmos que minimizam o número de rodadas de computação local e de comunicação.

3.2.3 Classificação

A classificação dos modelos foi baseada em função de três critérios: a granularidade, a topologia e a maneira como o custo de comunicação é avaliado.

A granularidade é o primeiro fator de aproximação dos modelos. Ela pode ser fina ou grossa. No primeiro caso, o número de processadores é arbitrário e geralmente ligado ao tamanho do problema. No segundo caso, o tamanho do problema n e o número de processadores p são dois parâmetros distintos. Em geral, os algoritmos funcionam então em relação entre essas duas quantidades: $n \simeq p$ permite o acesso rápido na memória local mas multiplica as comunicações, ao contrário, $n >> p$ limita o número de comunicação, mas pode sobrecarregar a memória local de processador. A granularidade é uma medida do paralelismo disponível e possível de ser extraída

de um problema.

A topologia da máquina parece ser um parâmetro essencial na computação paralela. Como sugere o modelo *BSP*, a topologia pode não estar explicitamente representada, mas pode aparecer de maneira implícita através de outros parâmetros. O modelo da topologia explícita permite refinar o algoritmo mas pode se tornar muito dependente da arquitetura. Contrariamente, o modelo da topologia implícita é portável, mas pode talvez se revelar pouco confiável com os resultados experimentais sobre a máquina.

O custo de comunicação pode ser avaliado de diferentes formas. Vamos considerar as comunicações fixas e as comunicações paramétricas. No modelo das comunicações fixas, uma comunicação é uma operação atômica de custo unitário. No modelo da granularidade fina, por exemplo, poderia ser uma troca de um dado entre dois processadores vizinhos; ao contrário, na granularidade grossa, a noção de custos fixos é delicada. O modelo das comunicações paramétricas é um modelo no qual todas comunicações não são contabilizadas de maneira idêntica. Uma das medidas mais usadas é a quantidade de dados enviadas. Mas outras medidas podem ser consideradas tais como: o tempo de inicialização e a distância entre processadores.

Antes de classificar os diferentes modelos descritos acima, conforme os três critérios, lembramos que o objetivo não é de julgar. Não se tem, por enquanto, um modelo universal satisfazendo todos os critérios de um bom modelo. Do ponto de vista fundamental, um bom modelo é aquele em que podemos construir uma teoria rica, permitindo exprimir formalmente um grande número de resultados considerados naturalmente como de primeira importância. Como exemplo, a questão de $P = NC$ do modelo *PRAM* é crucial e natural, pois é legítimo perguntar até quando um problema solúvel seqüencialmente em tempo polinomial pode ser paralelizado. Na prática, um bom modelo poderia ser um modelo em que o comportamento de muitas máquinas reais poderia ser expresso. É conveniente buscar a qualificação dos modelos em função de sua utilidade, visto que é ilusório quantificar seu valor intrínseco sobre uma escala linear forçosamente arbitrária.

A figura 3.3 apresenta a classificação dos modelos em função dos critérios acima. No modelo *PRAM* a granularidade é fina. A topologia é implícita no sentido que ela não é considerada. Poderia ser estimado, ao contrário, que o *PRAM* tem intrinsecamente uma topologia de grafo completo, ou ainda que a memória compartilhada é uma forma de topologia explícita. Consideramos o modelo *PRAM* com a topologia

implícita, pois do ponto de vista da análise de algoritmos, a topologia não interfere.

Topologia	Granularidade grossa		Granularidade fina	
Explícita	Comunicação paramétrica	Comunicação fixa	Comunicação paramétrica	Comunicação fixa
	G-RAM “MIMD”	G-RAM Granularidade grossa		G-RAM “SIMD”
Implícita	Comunicação paramétrica	Comunicação fixa	Comunicação paramétrica	Comunicação fixa
	BSP	CGM		PRAM

Figura 3.3: A classificação dos modelos. As comunicações em relação ao custo são fixas ou paramétricas, as topologias são implícitas ou explícitas e a granularidade grossa ou fina.

O modelo *G-RAM SIMD* é um modelo de granularidade fina, comunicação fixa e topologia explícita. O termo *SIMD* faz referência a toda uma gama de máquinas massivamente paralelas, possuindo um único sequenciador distribuindo as instruções ao conjunto de processadores. Nesse modelo, a topologia é explícita, tais como grade e hipercubo. A natureza de granularidade fina implica quase necessariamente em considerar as comunicações atômicas de custo unitário.

Os modelos *BSP* e *CGM* são de granularidade grossa e de topologia implícita. Os custos de comunicação fazem a diferença entre eles. O *BSP* modela seus custos considerando os parâmetros L e g. De fato, a topologia da máquina poderia ser ocultada atrás dos parâmetros L e g do modelo *BSP*. O modelo *CGM* é uma variante de modelo *BSP*. O objetivo dos algoritmos *CGM* é de minimizar as fases de comunicação.

Os modelos *G-RAM MIMD* e granularidade grossa são ambos de topologia explícita e de granularidade grossa. A diferença entre eles é como no caso dos modelos *BSP* e *CGM*, baseada no custo de comunicação. No modelo *G-RAM MIMD*, as comunicações são modeladas finamente. Por exemplo, no modelo de troca de mensagem, o custo de comunicação de vizinho a vizinho é modelado por uma função linear do tamanho L das mensagens: $\beta + L\tau$, onde β representa um tempo de inicialização e $\frac{1}{\tau}$ a largura da banda de comunicação. No modelo por comutação de circuitos, esse

mesmo tempo é representado por $\beta + d(x, y)\delta + L\tau$, onde $d(x, y)$ denota a distância entre a origem x e o destino y, e δ é função de tempo de comutação dos caminhos intermediários. As comunicações no modelo *granularidade fina* são atômicas com custo unitário.

Observamos que duas casas são vazias que correspondem aos modelos de granularidade fina, às comunicações paramétricas e à topologia explícita ou implícita. Isso vem simplesmente do fato de que um modelo de granularidade fina supõe que muitas vezes as comunicações são atômicas e de custos unitários. As fases de comunicação são calculadas onde cada fase tem um custo constante. A quantidade de dados transferidos é constante a cada fase.

3.3 Conclusão

A definição de um modelo resulta da escolha a priori de aspectos a serem considerados. Assim, a concepção de um modelo com determinadas funcionalidades é sempre feita em detrimento de outras características. O modelo *PRAM* é simples e permite extrair o paralelismo. O modelo *G-RAM* de topologia explícita garante a boa previsibilidade de custos medidos. O modelo *BSP* permite o desenvolvimento dos algoritmos genéricos. É recomendável conhecer todas as grandes classes de modelos e estar apto a analisar e comparar os algoritmos, não somente do ponto de vista intra-modelos, mas também inter-modelos.

Capítulo 4

Algoritmo Paralelo PRAM

Neste capítulo, descrevemos um algoritmo paralelo [33], no modelo *PRAM*, que resolve o problema de fecho convexo. A estratégia, usada geralmente na computação paralela, é de divisão-e-conquista [8, 9, 12, 36]. Essa estratégia nos leva a uma forma natural de explorar o paralelismo: os subproblemas gerados podem ser executados independentemente e, portanto, em paralelo. O algoritmo paralelo usando esta estratégia, a ser apresentado, leva tempo ótimo $O(\log n)$ em uma *PRAM CREW*. A maior dificuldade em garantir esta complexidade de tempo está na fase em que os resultados dos subproblemas são combinados. O algoritmo paralelo de busca e de ordenação [33] é utilizado no desenvolvimento desse algoritmo.

4.1 Estratégia de Divisão-e-Conquista

Considere um conjunto $S = \{p_1, p_2, \cdots, p_n\}$ de pontos no plano, onde n é uma potência de 2. O algoritmo começa a ordenar os pontos de S pela sua abscissa, através de um algoritmo paralelo que leva tempo $O(\log n)$ usando $O(n)$ processadores [33, 44].

Com isto temos os pontos p e q de menor e maior abscissa respectivamente. Sabemos que esses pontos pertencem ao fecho convexo $CH(S)$. Eles particionam $CH(S)$ nos **fechos superior** e **inferior**. O fecho superior, denotado $UH(S)$, consiste dos pontos de p até q, no sentido horário. Os pontos do q até p formam o fecho inferior e denotado $LH(S)$. A Figura 4.1 mostra os fechos superior e inferior. A idéia é determinar separadamente o fecho convexo superior $UH(S)$ e inferior $LH(S)$ para então achar o fecho convexo.

Para obter o fecho convexo superior $UH(S)$, precisa-se combinar $UH(S_1)$ e $UH(S_2)$

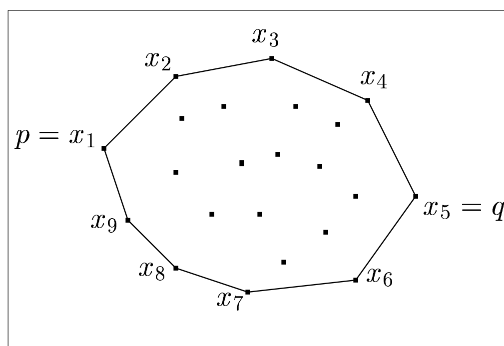

Figura 4.1: Um fecho convexo para um conjunto de pontos. O fecho superior é $(x_1, x_2, x_3, x_4, x_5)$ e o fecho inferior é $(x_5, x_6, x_7, x_8, x_9, x_1)$.

através de sua tangente comum superior, onde onde $S_1 = \{p_1, p_2, \cdots, p_{n/2}\}$ e $S_2 = \{p_{(n/2)+1}, p_{(n/2)+2}, \cdots, p_n\}$. O algoritmo paralelo *PRAM* acha a tangente comum superior dos $UH(S_1)$ e $UH(S_2)$ em tempo constante, usando um número linear de operações. A tangente comum inferior é encontrada de maneira análoga.

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 $UH(S_1)$ e $UH(S_2)$.

Algoritmo: *FECHO CONVEXO SUPERIOR*
Entrada: Um conjunto $S = \{p_1, p_2, \cdots, p_n\}$ de pontos ordenados pela abscissa no plano.
Saída: O fecho convexo superior $UH(S)$.

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

2. Sejam $S_1 = \{p_1, p_2, \cdots, p_{n/2}\}$ e $S_2 = \{p_{(n/2)+1}, p_{(n/2)+2}, \cdots, p_n\}$. Calcule recursivamente $UH(S_1)$ e $UH(S_2)$.

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

4.2 Um Algoritmo de Tempo Constante para Determinação da Tangente Comum Superior

Como queremos um algoritmo de complexidade de tempo $O(\log n)$, a tangente comum superior deve ser determinada em tempo constante. Abaixo descrevemos como obtê-la com este tempo desejado.

Seja $S = \{p_1, p_2, \cdots, p_n\}$ um conjunto de pontos do plano ordenados pela abscissa e sejam $S_1 = \{p_1, p_2, \cdots, p_{n/2}\}$ e $S_2 = \{p_{(n/2)+1}, p_{(n/2)+2}, \cdots, p_n\}$. A tangente comum superior entre $CH(S_1)$ e $CH(S_2)$ é a tangente comum tal que $CH(S_1)$ e $CH(S_2)$ estão abaixo dela. A tangente comum inferior é definida de maneira analoga.

Sejam $UH(S_1) = (r_1, \cdots, r_s)$ e $UH(S_2) = (q_1, \cdots, q_t)$ os fechos superiores do S_1 e S_2, respectivamente, dados no sentido horário, onde $r_i, q_j \in S$ para $1 \leq i \leq s$ e $1 \leq j \leq t$. Nosso problema é determinar os pontos $u = r_i$ e $v = q_j$ de $UH(S_1)$ e $UH(S_2)$, 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 seqüencia $(r_1, \cdots, r_i, q_j, \cdots, q_t)$. 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 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 r_i de $UH(S_1)$ e $UH(S_2)$ pode ser determinada; isto é, queremos determinar um ponto $q_{j(i)}$ de $UH(S_2)$ tal que $UH(S_2)$ está abaixo da linha determinada por r_i e $q_{j(i)}$. O lema seguinte mostra como podemos obter esta tangente usando um algoritmo de busca paralela.

Lema 4.2.1 *Seja r_i um ponto qualquer de $UH(S_1)$. Então, dado um ponto qualquer q_l de $UH(S_2)$, pode-se determinar, em tempo seqüencial $O(1)$ se $q_{j(i)}$ está à direita, é igual a, ou está à esquerda de q_l, onde $q_{j(i)}$ é o ponto de $UH(S_2)$ tal que $r_i q_{j(i)}$ é a tangente a $UH(S_2)$.*

Prova. Seja L uma reta determinada por r_i e q_l, e L' (respectivamente L'') a parte de L estritamente à esquerda (respectivamente à direita) de q_l, como indicado na figura 4.2. Então, se L' está acima do segmento $q_{l-1}q_l$ e L'' está abaixo do segmento

$q_l q_{l+1}$, então $q_{j(i)}$ está à direita de q_l. Se $q_l = q_{j(i)}$ então q_{l-1} e q_{l+1} estão abaixo de L senão $q_{j(i)}$ está à esquerda de q_l. Uma vez que r_i e q_l são dados, podemos determinar cada uma das três condições em tempo seqüencial $O(1)$. ∎

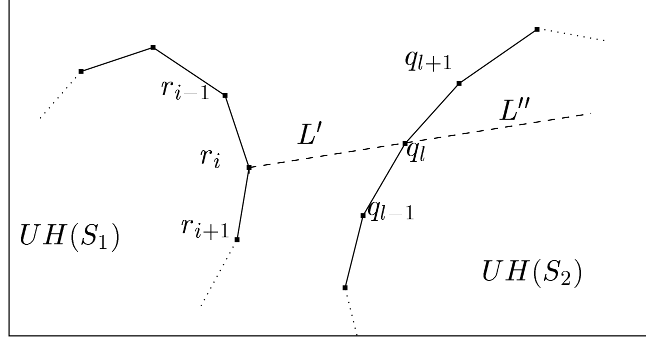

Figura 4.2: Determinação da tangente a $UH(S_2)$ passando por r_i. Nesse caso, L' está acima do segmento $q_{l-1}q_l$, e L'' está abaixo do segmento $q_l q_{l+1}$. Assim, a tangente de r_i a $UH(S_2)$ cruza o ponto de $UH(S_2)$ que está à direita de q_l.

Corolário 4.2.2 *Dados dois fechos superiores $UH(S_1)$ e $UH(S_2)$, e um ponto r_i do $UH(S_1)$, a tangente $r_i q_{j(i)}$ a $UH(S_2)$ pode ser determinada em tempo $O(\frac{\log t}{\log k})$ usando k processadores, onde t é o número de pontos de $UH(S_2)$ e $1 < k \leq t$.*

Prova. Lembrando que os pontos de $UH(S_2)$ são ordenados no sentido horário, podemos usar o algoritmo paralelo de busca [33] para encontrar o ponto $q_{j(i)}$. Escolhemos k pontos de $UH(S_2)$ que o dividem em $k+1$ partes, cada um com aproximadamente o mesmo número de pontos. Para cada um dos k pontos $q_{t_1}, \cdots, q_{t_k}$, determinamos em paralelo e em tempo $O(1)$, a posição relativa entre o segmento $r_i q_h$ e o sucessor de q_h em $UH(S_2)$, com $t_1 \leq h \leq t_k$. Repetimos o processo até que $q_{j(i)}$ seja encontrado. O tempo é dado por $T(n) = T(n/k) + c$, o que dá $T(n) = O(\frac{\log t}{\log k})$. ∎

Lema 4.2.3 *Seja (u, v) a tangente comum superior aos fechos convexos superiores $UH(S_1)$ e $UH(S_2)$. Suponhamos que, para qualquer ponto r_i de $UH(S_1)$, seja dado $q_{j(i)}$ de $UH(S_2)$ tal que o segmento de reta $r_i q_{j(i)}$ seja a tangente a $UH(S_2)$. Então, em tempo seqüencial $O(1)$, podemos determinar se u está à esquerda, é igual a, ou está à direita de r_i.*

Note que, se $r_i q_{j(i)}$ é tangente a $UH(S_1)$, então u é igual a r_i, pois $r_i q_{j(i)}$ já é

tangente a $UH(S_2)$. Senão, u está à esquerda de r_i se e somente se r_{i-1} está acima de $r_i q_{j(i)}$. Veja Figura 4.3.

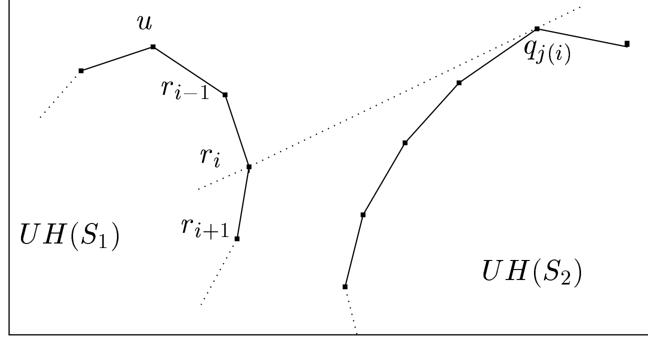

Figura 4.3: Determinação da posição do ponto u onde a tangente comum superior cruza $UH(S_1)$, dada a tangente $r_i q_{j(i)}$ a $UH(S_2)$. Na figura, r_{i-1} está acima da tangente, e então u deve estar à esquerda de r_i.

Com esses dois lemas, podemos determinar, para qualquer ponto r_i de $UH(S_1)$, se o ponto u aparece à esquerda de, à direita de, ou é igual a u_i em tempo $O(\frac{\log t}{\log k})$ usando k processadores. Se escolhemos 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.

Essas observações nos levam a um algoritmo paralelo de busca para isolar u da seguinte maneira. Escolhemos $\sqrt{s}$ pontos de $UH(S_1)$, que dividem $UH(S_1)$ em partes aproximadamente do mesmo tamanho. Podemos então isolar u em uma destas partes em tempo $O(1)$ usando $\sqrt{s}\sqrt{t}$ processadores. Da maneira análoga podemos isolar v em uma parte de $UH(S_2)$ contendo aproximadamente $\sqrt{t}$ pontos 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 verificar em tempo seqüencial $O(1)$, se ele forma uma tangente comum superior entre $UH(S_1)$ e $UH(S_2)$. Portanto, 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 a seguir encontra a tangente comum superior.

Algoritmo: *Tangente Comum Superior*

Entrada: Os fechos superiores $UH(S_1) = (r_1, r_2, \cdots, r_s)$ e
$UH(S_2) = (q_1, q_2, \cdots, q_t)$ ordenados no sentido horário, onde
$S_1 = \{p_1, p_2, \cdots, p_{n/2}\}$ e $S_2 = \{p_{n/2+1}, p_{n/2+2}, \cdots, p_n\}$ são tais que
$x(p_1) < x(p_2) < \cdots < x(p_n)$. Suponha que $\sqrt{s}$ e $\sqrt{t}$ são inteiros.

Saída: Os pontos u e v tais que a reta determinada por u e v é
a tangente comum superior entre $UH(S_1)$ e $UH(S_2)$.

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

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 $A = (r_{l\sqrt{s}+1}, \cdots, r_{(l+1)\sqrt{s}-1})$ contendo u.

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

Teorema 4.2.4 *O algoritmo Tangente Comum Superior calcula corretamente a tangente comum superior entre UH(S_1) e UH(S_2). Ele executa em tempo $O(1)$ utilizando um número linear de operações.*

Prova. Suponha que temos $s + t$ processadores disponíveis. O passo 1 pode ser executado em $O(1)$ utilizando $\sqrt{t}$ processadores para cada i (Lema 4.2.1 e seu Corolário 4.2.2). Como $\sqrt{st} \leq s + t$, todos pontos $q_{j(i\sqrt{s})}$ podem ser encontrados em tempo $O(1)$ usando $s + t$ processadores. O passo 2 pode ser realizado em tempo $O(1)$ com $O(\sqrt{s})$ processadores. O passo 3 é análogo. Portanto, o algoritmo pode ser executado em tempo $O(1)$ usando $s + t$ processadores. O número total de operações é $O(s + t) = O(n)$. ∎

4.3 Cálculo do Fecho Convexo

O fecho convexo é obtido usando a estratégia de divisão-e-conquista com o cálculo das tangentes comum superior e inferior.

Teorema 4.3.1 *O fecho convexo de um conjunto de n pontos no plano pode ser calculado em tempo $O(\log n)$, usando $O(n \log n)$ operações. Assim, o algoritmo para encontrar o fecho convexo é ótimo*[1].

[1]Um algoritmo paralelo é ótimo se $pT_p(n) = O(T^*(n))$, onde $T^*(n)$ é o tempo do melhor algoritmo

Prova. Seja $S = \{p_1, p_2, \cdots, p_n\}$ um conjunto de n pontos no plano. O algoritmo ordena os pontos de S de acordo com suas abscissas. Esse procedimento pode ser feito em tempo $O(\log n)$ [33, 44]. Então, para os pontos vale a relação: $x(p_1) < x(p_2) < \cdots < x(p_n)$.
Para determinar o fecho convexo $CH(S)$ usaremos a estratégia da divisão e conquista da seguinte maneira. Se $n \leq 4$, calcula diretamente $CH(S)$. Senão, o algoritmo acha recursivamente $CH(S_1)$ e $CH(S_2)$. A última fase é a combinação de $CH(S_1)$ e $CH(S_2)$ com as tangentes superior e inferior comum e obtenção de $CH(S)$. ∎

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

O modelo *PRAM* usa o modo de acesso a memória *CREW* pois apenas a leitura concorrente é necessária pelos algoritmos de ordenação e pelo algoritmo de busca paralela. A escrita concorrente é não necessária.

seqüencial para o problema, $T_p(n)$ o tempo do algoritmo paralelo usando p processadores para resolver o mesmo problema e n o tamanho da entrada do problema.

Capítulo 5

Algoritmos Paralelos CGM

Neste capítulo apresentaremos dois algoritmos, no modelo *CGM*, para resolver o problema do fecho convexo. Um deles é probabilístico e outro é determinístico.

Os algoritmos que veremos nesse capítulo são escaláveis, de complexidade ótima na computação local, portáveis e independentes de arquitetura. Eles são desenvolvidos no modelo de computação paralela *CGM*, com as características seguintes:

- Há p processadores $P_1, \cdots, P_p$ interligados por alguma rede de interconexão;
- Cada processador P_i tem memória local $O(n/p)$;
- Para alguma constante arbitrária pequena e fixa $\alpha > 0$, $n/p \geq p^{\alpha}$. Isso é equivalente a $1 \leq p \leq n^{1-\epsilon}$ onde $\epsilon = \frac{\alpha}{1+\alpha}$;
- O tamanho total de dados n é um número grande mas p não é necessariamente grande.

Como vimos no capítulo 3, o modelo *CGM* é realístico em termo de máquinas existentes. Por isso, as características enumeradas acima são em geral verificadas.

Na seção 5.1 descrevemos o algoritmo paralelo probabilístico e na seção 5.2 apresentamos o algoritmo paralelo determinístico.

5.1 Algoritmo CGM Probabilístico

O algoritmo probabilístico que vamos ver nessa seção se encontra em [14]. Com o modelo de distribuição de dados uniforme, a complexidade dos algoritmos abaixo

é, com alta probabilidade[1], $(k+2)(\frac{T_1(n)}{p} + O(\frac{n}{p}))$ para tempo de computação local e $(k+1)(T_{pSum}(p) + T_{compr}(p,n))$ para tempo de comunicação, onde $k = \lceil \frac{1}{2\alpha} + \frac{1}{2} \rceil$ é uma constante e $\alpha > 0$ uma constante fixa. $T_1(n)$ denota a complexidade de tempo seqüencial, e $T_{pSum}(p)$ é a complexidade paralela para calcular a soma parcial de p números cada um armazenado em um processador. $T_{compr}(p,n)$ é o tempo para comprimir um subconjunto de dados de tamanho $n' \leq n$ em $p' \leq p$ processadores.

Estes algoritmos abaixo requerem, com alta probabilidade, somente um número pequeno e fixo de rodadas de comunicação, independentemente do tamanho do problema. Cada rodada de comunicação corresponde a uma soma parcial ou operação de comprimir. Os restantes dos cálculos são locais.

Em [14] dois casos são considerados. O primeiro caso é para $p \leq \sqrt{n}$ e o segundo caso que generaliza o primeiro é dado para $1 \leq p \leq n^{1-\epsilon}$ onde $\epsilon = \frac{\alpha}{1+\alpha}$.

Recordamos o problema: dado um conjunto aleatório S de n pontos em $\mathbb{R}^2$, encontrar seu fecho convexo $CH(S)$. Prova-se que este fecho $CH(S)$ verifica as propriedades seguintes:
P1. $CH(A_1 \cup \cdots \cup A_i) = CH(CH(A_1) \cup \cdots \cup CH(A_i))$ para cada $A_1, \cdots, A_i \subseteq S$, e
P2. Existe uma função $h(n)$ tal que:

(a) para cada subconjunto aleatório $A \subseteq S$, $\mathrm{E}(|CH(A)|) \leq h(|A|)$.
Observação: Para alguma váriavel aleatória X, E(X) e Pr{X=y} denotam o valor esperado de X e a probabilidade que X toma um certo valor y, respectivamente.

(b) $h(n) \leq n^\delta$ para $0 < \delta < min\{\epsilon, 1/8\}$.

As provas destas propriedades estão em [14] e são omitidas neste trabalho.

5.1.1 Caso $p \leq \sqrt{n}$

O algoritmo abaixo resolve o problema para o caso $p \leq \sqrt{n}$.

Algoritmo: *Algoritmo 1.*
Entrada: Um conjunto finito $S = \{p_1, ..., p_n\}$ de pontos. Cada processador P_i armazena um subconjunto aleatório S_i de n/p pontos de S. S_i são

[1] $X = O(f(n))$ com "alta probabilidade", se e somente se $\forall c > c_0 > 1$, $Prob\{X \geq cf(n)\} \leq \frac{1}{n^{g(c)}}$, onde c_0 é uma constante fixa e $g(n)$ é um polinômio em c com $g(c) \to \infty$ para $c \to \infty$.

disjuntos.

Saída: O fecho convexo de S.

1. Cada processador P_i calcula sequencialmente $CH(S_i)$.
Seja $S' = CH(S_1) \cup \cdots \cup CH(S_p)$, $n' = |S'|$.

2. If $n' \leq n/p$ **Then** S' é comprimido dentro do processador P_1, o qual calcula sequencialmente $CH(S) = CH(S')$. Fim.

3. O conjunto S' é comprimido em $p' \leq \frac{2n'p}{n}$ processadores da seguinte forma: a seqüência $CH(S_1), \cdots, CH(S_p)$ é dividida em p' subseqüências maximais de $CH(S_j)$ consecutivas, tal que o tamanho total de cada subseqüência é no máximo n/p. Seja S'_i o conjunto de pontos na subseqüência i tal que $1 \leq i \leq p'$. O conjunto S'_i é armazenado no processador P_i.

4. Cada processador P_i $(1 \leq i \leq p')$ calcula sequencialmente $CH(S'_i)$.
Seja $S'' = CH(S'_1) \cup \cdots \cup CH(S'_{p'})$, $n'' = |S''|$.

5. If $n'' \leq n/p$ **Then** S'' é comprimido dentro do processador P_1 o qual calcula sequencialmente $CH(S) = CH(S'')$.
Else continua com qualquer algoritmo determinístico.

A corretude desse algoritmo vem da propriedade que o fecho convexo da união dos conjuntos é igual o fecho convexo da união dos fechos convexos (Propriedade P1). Notemos que para o caso da linha 5, qualquer algoritmo paralelo determinístico pode ser aplicado. Um caso trivial seria de usar o algoritmo seqüencial no processador P_1 e com a memória acessando as memórias de outros processadores, através de passagem de mensagens.

Os lemas e teorema seguintes são de [14].

Lema 5.1.1 *Com alta probabilidade, vale $n' \leq n/p$ ou $n'' \leq n/p$.*

Prova. Consideramos alguns δ tal que $0 < \delta < 1/4$. Então, segue da propriedade P2 que $h(n) \leq n^{\delta}$. Três casos são considerados para provar o lema.

Caso 1: $p \leq n^{\frac{1-\delta}{3}}$

A partir da desigualdade de Chebyshev [22] e da suposição acima, como $\delta \leq \frac{1}{4}$, temos que:

$$\begin{aligned}\Pr\{\exists i \ 1 \leq i \leq p : |CH(S_i)| > n^{\frac{2}{3}(\frac{1}{2}-\delta)}h(n)\} &\leq p \ \Pr\{|CH(S_i)| > n^{\frac{2}{3}(\frac{1}{2}-\delta)}h(n)\} \\ &\leq \frac{p}{n^{\frac{4}{3}(\frac{1}{2}-\delta)}} \\ &\leq \frac{n^{\frac{1-\delta}{3}}}{n^{\frac{4}{3}(\frac{1}{2}-\delta)}}\end{aligned}$$

$$\leq \frac{1}{n^{\frac{1}{3}-\delta}} \to 0 \text{ para } n \to \infty.$$

Logo, depois do passo 1 do *Algoritmo 1* e com probabilidade de, no mínimo, $1 - \frac{1}{n^{\frac{1}{3}-\delta}} \to 1$ (para $n \to \infty$), é obtido um conjunto S' de tamanho:

$$\begin{aligned} n' &\leq pn^{\frac{2}{3}(\frac{1}{2}-\delta)}h(n) \\ &\leq n^{\frac{1-\delta}{3}} n^{\frac{2}{3}(\frac{1}{2}-\delta)} n^{\delta} \\ &= n^{\frac{2}{3}}. \end{aligned}$$

Como $p \leq n^{\frac{1-\delta}{3}}$ segue que $\frac{n}{p} \geq n^{\frac{2}{3}}$. Portanto, $n' \leq \frac{n}{p}$ com alta probabilidade , e segue o lema 5.1.1 para o caso 1.

Caso 2: $n^{\frac{1-\delta}{3}} \leq p \leq \frac{1}{\sqrt{2}} n^{\frac{1-\delta}{2}}$

Como $n^{\frac{1-\delta}{3}} \leq p$, temos que $p \to \infty$ para $n \to \infty$. Neste caso, segue da lei dos grandes números [22] que $\Pr\{n' \leq 2ph(n)\} \to 1$ para $n \to \infty$. Observamos que $2ph(n) \leq \frac{n}{p}$ se $p \leq \frac{\sqrt{n}}{\sqrt{2h(n)}}$. A última expresão é válida se $p \leq \frac{\sqrt{n}}{\sqrt{2n^{\delta}}}$, o qual é verdadeiro se $p \leq \frac{1}{\sqrt{2}} n^{\frac{1-\delta}{2}}$. Conseqüentemente, $n' \leq \frac{n}{p}$ com alta probabilidade, e segue o lema 5.1.1 para o caso 2.

Caso 3: $\frac{1}{\sqrt{2}} n^{\frac{1-\delta}{2}} \leq p \leq \sqrt{n}$

Como $\frac{1}{\sqrt{2}} n^{\frac{1-\delta}{2}} \leq p$, vale $p \to \infty$ para $n \to \infty$. De novo neste caso, segue da lei dos grandes números que, com alta probabilidade:

$$\begin{aligned} \tfrac{1}{2}ph(n) &\leq n' \\ &\leq 2ph(n) \\ &\leq 2\sqrt{n}n^{\delta}. \end{aligned}$$

Como $p' \leq \frac{2n'}{\frac{n}{p}}$ e $p \leq \sqrt{n}$, temos com alta probabilidade:

$$\begin{aligned} p' &\leq \frac{4ph(n)}{\frac{n}{p}} \\ &\leq 4h(n) \\ &\leq 4n^{\delta}. \end{aligned}$$

Vamos agora estudar os passos 3 e 4 do *Algoritmo 1*. Para cada $S'_i, 1 \leq i \leq p'$, existem conjuntos $S_{t_i}, S_{t_i+1}, \cdots, S_{u_i}$ tais que $S'_i = CH(S_{t_i}) \cup CH(S_{t_i+1}) \cup \cdots \cup CH(S_{u_i})$. Conseqüentemente, pela propriedade P1:

$$\begin{aligned} CH(S'_i) &= CH(CH(S_{t_i}) \cup CH(S_{t_i+1}) \cup \cdots \cup CH(S_{u_i})) \\ &= CH(S_{t_i} \cup S_{t_i+1} \cup \cdots \cup S_{u_i}). \end{aligned}$$

Defina $orig(S'_i) = S_{t_i} \cup S_{t_i+1} \cup \cdots \cup S_{u_i}$, então $CH(S'_i) = CH(orig(S'_i))$. ∎

Lema 5.1.2 *Para cada $1 \leq i \leq p'$, $orig(S'_i)$ é um subconjunto aleatório de S e, portanto, $E(|CH(S'_i)|) = E(|CH(orig(S'_i))|) \leq h(n)$.*

Prova. Vamos apresentar o esboço da prova.
$orig(S'_i) = S_{t_i} \cup S_{t_i+1} \cup \cdots \cup S_{u_i}$. Claramente, S_{t_i} é um conjunto aleatório de $\frac{n}{p}$ pontos. Para cada $l \in [t_i, u_i]$, $S_{t_i} \cup S_{t_i+1} \cup \cdots \cup S_l \cup \cdots \cup S_{u_i}$ é na bijeção com $S_l \cup S_{t_i} \cup S_{t_i+1} \cup \cdots \cup S_{l-1} \cup S_{l+1} \cup \cdots \cup S_{u_i}$, nessa ordem. Conseqüentemente, segue que S_l tem a mesma distribuição como S_{t_i}. Portanto, $orig(S'_i)$ é a união de subconjuntos aleatórios de S, e o lema 5.1.2 é provado. ∎

Conseqüentemente, segue da desigualdade de Chebyshev que:

$$\begin{aligned} \Pr\{\exists i\ 1 \leq i \leq p' : |CH(S'_i)| > n^{\frac{1}{3}-2\delta}h(n)\} &\leq p' \Pr\{|CH(S_i)| > n^{\frac{1}{3}-2\delta}h(n)\} \\ &\leq \frac{p'}{n^{\frac{2}{3}-2\delta}} \\ &\leq \frac{4n^{\delta}}{n^{\frac{2}{3}-4\delta}} \to 0 \text{ pois } \delta < \tfrac{1}{8}. \end{aligned}$$

Portanto, depois o passo 4 do *Algoritmo 1* obtemos, com alta probabilidade, um conjunto S'' de tamanho:

$$\begin{aligned} n'' &\leq p' n^{\frac{1}{3}-2\delta} h(n) \\ &\leq 4n^{\delta} n^{\frac{1}{3}-2\delta} n^{\delta} \\ &\leq n^{\frac{1}{2}}. \end{aligned}$$

Como $p \leq \sqrt{n}$ então $\frac{n}{p} \geq \sqrt{n}$. Conseqüentemente, $n'' \leq \frac{n}{p}$ com alta probabilidade, e segue o lema 5.1.1 para o caso 3. Isto conclui a prova do lema 5.1.1.

Teorema 5.1.3 *Dado um multicomputador com p processadores, $p \leq \sqrt{n}$, então , com alta probabilidade, o Algoritmo 1 calcula CH(S), $|S| = n$, com tempo de comunicação no máximo $2(T_{pSum}(p) + T_{compr}(p, n))$ e tempo de cálculo local no máximo $3\frac{T_1(n)}{p} + O(\frac{n}{p})$.*

5.1.2 Caso $1 \leq p \leq n^{1-\epsilon}$

Podemos agora generalizar o algoritmo *Algoritmo 1* para resolver o problema do fecho convexo. Seja k uma constante inteira positiva fixa tal que $k \geq \frac{1}{2(\epsilon-\delta)} - \frac{1}{6} = \frac{1+\alpha}{2\alpha-2\delta(1+\alpha)} - \frac{1}{6}$.

Algoritmo: *Algoritmo 2.*

Entrada: Um conjunto finito $S = \{p_1, ..., p_n\}$ do pontos. Cada processador P_i armazena um subconjunto aleatório S_i de n/p pontos de S.

Saída: O fecho convexo de S.

Seja $n^{(0)} = n, p^{(0)} = p$ e $S^{(0)} = S$.

1. For $j = 0 \cdots k$ **do**

a. Cada processador $P_i, 1 \leq i \leq p^{(j)}$, calcula sequencialmente $CH(S_i^{(j)})$.
Seja $S^{(j+1)} = CH(S_1^{(j)}) \cup \cdots \cup CH(S_{p^{(j)}}^{(j)})$, $n^{(j+1)} = |S^{(j+1)}|$.

b. If $n^{(j+1)} \leq n/p$ **Then** $S^{(j+1)}$ é comprimido dentro do processador P_1 o qual calcula sequencialmente $CH(S) = CH(S^{(j+1)})$. Fim.

c. O conjunto $S^{(j+1)}$ é comprimido em $p^{(j+1)} \leq \frac{2n^{(j+1)}p}{n}$ processadores da seguinte forma: a seqüência $CH(S_1^{(j)}), \cdots, CH(S_{p^{(j)}}^{(j)})$ é dividida em $p^{(j+1)}$ subseqüências maximais de $CH(S_i^{(j)})$ consecutivas, tal que o tamanho total de cada subseqüência é no máximo n/p.
Seja $S_i^{(j+1)}$ o conjunto de pontos na subseqüência i, tal que $1 \leq i \leq p^{(j+1)}$. O conjunto $S_i^{(j+1)}$ é armazenado no processador P_i.

2. If $n^{(k+1)} \leq n/p$ **Then** no passo anterior 1.c., $S^{(k+1)}$ foi comprimido no processador P_1 o qual pode agora calcular sequencialmente $CH(S) = CH(S^{(k+1)})$.

Else continua com qualquer algoritmo determinístico.

Lema 5.1.4 *Com alta probabilidade, vale $n^{(j)} \leq n/p$ para algum $j \leq k+1$.*

Prova. Consideramos dois casos.
Se $p \leq \sqrt{n}$ então aplicamos a análise do *Algoritmo 1* dada na seção anterior.
Suponhamos agora que $\sqrt{n} \leq p \leq n^{1-\epsilon}$.
A ideia básica é de usar o lema 5.1.2 em cada iteração e a análise de caso 3 do lema 5.1.1. Obtemos, com alta probabilidade:

$$\begin{aligned} p^{(j+1)} &\leq \frac{4p^{(j)}h(n)}{\frac{n}{p}} \\ &\leq p^{(j)} \frac{4n^{\delta}}{n^{\epsilon}} \ \forall j \geq 0. \end{aligned}$$

Conseqüentemente, com alta probabilidade, $p^{(j)} \leq p\left(\frac{4n^\delta}{n^\epsilon}\right)^j \forall j \geq 0$. Observamos que $p\left(\frac{4n^\delta}{n^\epsilon}\right)^j \leq n^{\frac{\epsilon-\delta}{3}}$ se $i \geq \frac{1}{2(\epsilon-\delta)} - \frac{1}{6}$. A menos que o laço do *Algoritmo 2* seja parado para $j < k$, temos:

$$\begin{aligned}\Pr\{\exists i\ 1 \leq i \leq p : |CH(S_i^{(k)})| > n^{\frac{2}{3}(\epsilon-\delta)}h(n)\} &\leq \frac{p^{(k)}}{n^{\frac{4}{3}(\epsilon-\delta)}} \\ &\leq \frac{n^{\frac{1}{3}(\epsilon-\delta)}}{n^{\frac{4}{3}(\epsilon-\delta)}} \\ &\leq \frac{1}{n^{\epsilon-\delta}} \to 0 \text{ para } n \to \infty, \text{ pois } \delta < \epsilon.\end{aligned}$$

Portanto, com alta probabilidade:

$$\begin{aligned}n^{(k+1)} &\leq p^{(k)} n^\delta n^{\frac{2}{3}(\epsilon-\delta)} \\ &\leq n^{\frac{1}{3}(\epsilon-\delta)} n^\delta n^{\frac{2}{3}(\epsilon-\delta)} \\ &\leq n^\epsilon \\ &\leq \frac{n}{p}.\end{aligned}$$

Isso conclui a prova do lema 5.1.4. ∎

Teorema 5.1.5 *Dado um multicomputador com p processadores, então, com alta probabilidade, o Algoritmo 2 calcula CH(S),* $|S| = n$*, com tempo de comunicação no máximo* $(k+1)(T_{pSum}(p) + T_{compr}(p,n))$ *e tempo de cálculo local no máximo* $(k+2)(\frac{T_1(n)}{p} + O(\frac{n}{p}))$*, onde* $T_1(n)$ *é o tempo seqüencial e* $k \geq \frac{1}{2(\epsilon-\delta)} - \frac{1}{6} = \frac{1+\alpha}{2\alpha - 2\delta(1+\alpha)} - \frac{1}{6}$ *uma constante inteira positiva determinada.*

Aplicamos *Algoritmo 2* e teorema 5.1.5 com $k = \lceil \frac{1}{2\alpha} + \frac{1}{2} \rceil$. Temos então:

Teorema 5.1.6 *Dado um multicomputador com p processadores, então, com alta probabilidade, o Algoritmo 2 calcula CH(S),* $|S| = n$*, com tempo de comunicação no máximo* $(k+1)(T_{pSum}(p) + T_{compr}(p,n))$ *e tempo de cálculo local no máximo* $(k+2)(\frac{T_1(n)}{p} + O(\frac{n}{p}))$*, onde* $T_1(n)$ *é o tempo seqüencial e* $k = \lceil \frac{1}{2\alpha} + \frac{1}{2} \rceil$ *uma constante inteira positiva determinada.*

5.2 Algoritmo Paralelo Determinístico no Modelo CGM

O algoritmo que vamos ver agora é determinístico e escalável [18]. No pior caso, a complexidade de tempo é $O(\frac{n \log n}{p} + T_s(n,p))$ e o número de rodadas de comunicação

é $O(1)$, onde n é o tamanho do problema, p o número de processadores e $T_s(n,p)$ o tempo de ordenação de n números usando p processadores. O algoritmo é ótimo. A idéia chave, como no caso do algoritmo probabilístico, é de particionar os dados em dados locais, que são executados seqüencialmente em cada processador.

A estrutura do algoritmo é a seguinte. Os dados de problema estão distribuídos em memórias locais, onde ficam até o final da solução. Dados um conjunto S de n pontos e p processadores, o algoritmo mostra como encontrar o fecho superior $UH(S)$. O fecho inferior $LH(S)$ é calculado de maneira análoga. Sem perda da generalidade, suponha que os pontos estão no primeiro quadrante, isto é, suas coordenadas cartesianas são não negativas.

Algoritmo: *CGM UPPER HULL(S)*
Entrada: Cada processador armazena um conjunto de $\frac{n}{p}$ pontos escolhidos arbitrariamente de S.
Saída: A representação distribuída do fecho superior de S. Todos os pontos do fecho superior estão identificados e numerados da esquerda para a direita.

1. Ordena todos os pontos de S pela abscissa. Seja S_i o conjunto de $\frac{n}{p}$ pontos ordenados e armazenados no processador i.
2. Independentemente e em paralelo, cada processador i calcula o fecho superior de S_i. Seja X_i o resultado no processador i.
3. Cada processador calcula para X_i, $1 \leq i \leq p$, as linhas de tangente comum superior entre ele e todos X_j, $i < j \leq p$, e identifica o fecho superior de S usando as retas de tangente superior. (Este passo será detalhado adiante.)

O passo 1 pode ser executado usando a operação de ordenação global descrito em [18]. O passo 2 é totalmente seqüencial e pode ser completado em tempo $O(\frac{n \log n}{p})$, usando os algoritmos seqüenciais conhecidos [43]. O desafio principal está no passo 3. Neste passo, um algoritmo de *merge* combina os p fechos superiores disjuntos em um único fecho superior. Apresentaremos dois algoritmos de *merge* diferentes: *MergeHulls 1 e MergeHulls 2* ambos. O primeiro, descrito na subseção 5.2.4, é o *merge* simples ou direto requerendo um número constante de rodadas de comunicação global e $\frac{n}{p} \geq p^2$ de memória local por processador. O segundo *merge*, descrito na subseção 5.2.5, é mais complexo e usa o primeiro como um subprocedimento. Mas tem o grau alto de escalabilidade e pode ser implementado com somente $\frac{n}{p} \geq p^\epsilon$ de memória local, onde ϵ é uma constante fixa tal que $0 < \epsilon \leq 1$ e também requer um número constante de rodadas de comunicação. Os dois algoritmos usam a idéia de

conjuntos divisores selecionados, que foi introduzida em [38].

Antes de apresentarmos os algoritmos vamos relacionar alguns resultados que motivam e justificam a idéia.

5.2.1 Combinando os Fechos Convexos em Paralelo

Vamos ver como podemos combinar, em um único fecho superior, os p fechos superiores disjuntos armazenados em p processadores. Isso corresponde ao desafio no passo 3 do algoritmo *CGM UPPER HULL(S)*. Para isso, precisamos de algumas definições.

Definição 5.2.1 *Seja $\overline{ab}$ o segmento de reta ligando os pontos a e b. Vamos denotar por (ab) a reta passando por a e b. Dizemos que o ponto c está dominado pelo segmento de reta $\overline{ab}$ se e somente se a abscissa de c está estritamente entre as abscissas de a e b, e c está localizado abaixo do segmento $\overline{ab}$.*

Definição 5.2.2 *Seja $\{S_i\}$, $1 \leq i \leq p$, uma partição de S tal que $\forall x \in S_j, y \in S_i, j > i$, a abscissa de x é maior que de y.*

Definição 5.2.3 *Seja $X_i = \{x_1, x_2, \cdots, x_m\}$ um fecho superior. Então, $pred_{X_i}(x_i)$ denota x_{i-1} e $suc_{X_i}(x_i)$ denota x_{i+1}. Veja Figura 5.1.*

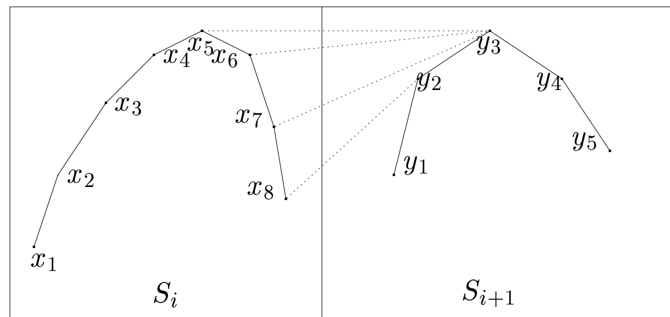

Figura 5.1: Seja $S' = S_i \cup S_{i+1}$ então $suc(x_j) = x_{j+1}, x_3 = Next_{S'}(x_2), x_4 = Next_{S'}(x_3), x_5 = Next_{S'}(x_4), Next_{S'}(x_5) = Next_{S'}(x_6) = Next_{S'}(x_7) = y_3, Next_{S'}(x_8) = y_2$, e $lm(S_i) = x_5$.

Dados dois fechos superiores $X_i = UH(S_i)$ e $X_j = UH(S_j)$, onde todos os pontos em S_j estão à direita de todos os pontos em S_i. A operação do merge consiste em

achar para um ponto $p \in X_i$, o ponto $q \in X_i \cup X_j$ que segue p no $UH(X_i \cup X_j)$. Veja Figura 5.1.

Definição 5.2.4 *Seja $Q \subseteq S$. Então, $Next_S : Q \longrightarrow S$ é a função tal que $Next_S(p) = q$ se e somente se q está a direita de p e $\overline{pq}$ está acima de $\overline{pq'}$ para todo $q' \in S$, q' a direita de p.*

Definição 5.2.5 *Seja $Y \subseteq S_i$. Então, $lm(Y)$ é uma função tal que $lm(Y) = y^*$ se e somente se y^* é o ponto mais à esquerda em Y tal que $Next_{Y \cup S_j, j>i}(y^*) \notin S_i$.*

Seja X um fecho superior de um conjunto de n pontos e c um ponto localizado à esquerda desse conjunto. Apresentamos um algoritmo seqüencial chamado QueryFindNext que busca $q = Next_X(c)$. A busca binária processa em tempo $O(\log |X|)$ [43]. Veja Figura 5.2.

Algoritmo: *QueryFindNext(X,c,q)*
Entrada: Um fecho superior $X = \{x_1, \cdots, x_m\}$ ordenado pela abscissa e o ponto c à esquerda de x_1.
Saída: Um ponto $q \in X, q = Next_X(c)$.
1. Se $X = \{x\}$ então $q \longleftarrow x$ e pára.
2. Se $\overline{x_{\lceil m/2 \rceil} suc(x_{\lceil m/2 \rceil})}$ está localizado abaixo da reta $(cx_{\lceil m/2 \rceil})$
então *QueryFindNext*$(\{x_1, \cdots, x_{\lceil m/2 \rceil}\}, c, q)$,
senão *QueryFindNext*$(\{x_{\lceil m/2 \rceil}, \cdots, x_m\}, c, q)$

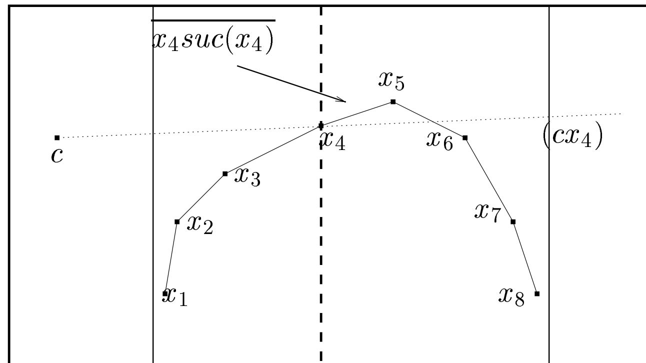

Figura 5.2: Um passo da busca binária de *QueryFindNext.* O segmento de reta $\overline{x_4 suc(x_4)}$ está acima da reta (cx_4) e o algoritmo busca sobre $\{x_4, \cdots, x_8\}$.

5.2.2 Caracterização do Fecho Superior

Uma caracterização do fecho superior de um conjunto S de pontos é baseada na observação seguinte [43]: *O segmento de reta $\overline{ab}$ é uma aresta do fecho superior de um conjunto S do pontos localizado no primeiro quadrante se e somente se todos os $n-2$ pontos restantes caem abaixo da reta (ab).* Vamos trabalhar com uma nova caracterização do fecho superior de S baseada na mesma observação, mas definida em termos da partição de S dada nas definições 5.2.2 e 5.2.5.

Consideramos os conjuntos S, S_i e X_i como definidos anterioramente.

Definição 5.2.6 *Seja $S' = \{c \in \cup X_i \mid c$ não está dominado pelo segmento de reta $\overline{x_i^* Next_{\cup X_j, j>i}(x_i^*)}$, $1 \leq i < p\}$, onde $x_i^* = lm(X_i)$.*

O teorema seguinte caracteriza o fecho superior $UH(S)$.

Teorema 5.2.1 *$S' = UH(S)$.*

Prova. Consideremos dois casos.

- $UH(S) \subseteq S'$.
  Suponha $y \in UH(S), y \notin S'$. Então, existe i tal que $\overline{x_i^* Next_{X_j, j>i}(x_i^*)}$ domina y. Portanto, $y \notin UH(S)$, o que é uma contradição.

- $S' \subseteq UH(S)$.
  Suponha $y \in S'$, $y \notin UH(S)$. Então existem p, q com $p \in$ UH(S), $q \in UH(S)$, e $q = Next_S(p)$, tais que $\overline{pq}$ domina y. Portanto, ambos p e q não podem pertencer a S_i, uma vez que $y \in X_i$ significa que y não está dominado por qualquer segmento de reta com pontos extremos em S_i. Assim, $p \in X_j$ e $q \in X_k$ com $j \neq k$. Consequentemente, como $q = Next_S(p)$, $p \in$ UH(S), e $q \in UH(S)$, então existe i tal que $p = x_i^*$, onde $x_i^* = lm(X_i)$. Portanto, y está dominado pelo segmento $\overline{x_i^* Next_{X_j, j>i}(x_i^*)}$, o que é uma contradição.

∎

As definições e o lema seguintes são necessários na descrição dos dois algoritmos paralelos do merge nas subseções 5.2.4 e 5.2.5. Eles ajudam não somente na descrição do algoritmo mas também na análise das complexidade de tempo e de espaço.

Definição 5.2.7 *Seja $G_i \subseteq X_i$ e $g_i^* = lm(G_i)$. Seja $R_i^- \subseteq X_i$ composto dos pontos entre $pred_{G_i}(g_i^*)$ e g_i^*, e $R_i^+ \subseteq X_i$ composto dos pontos entre g_i^* e $suc_{G_i}(g_i^*)$. Veja Figura 5.3.*

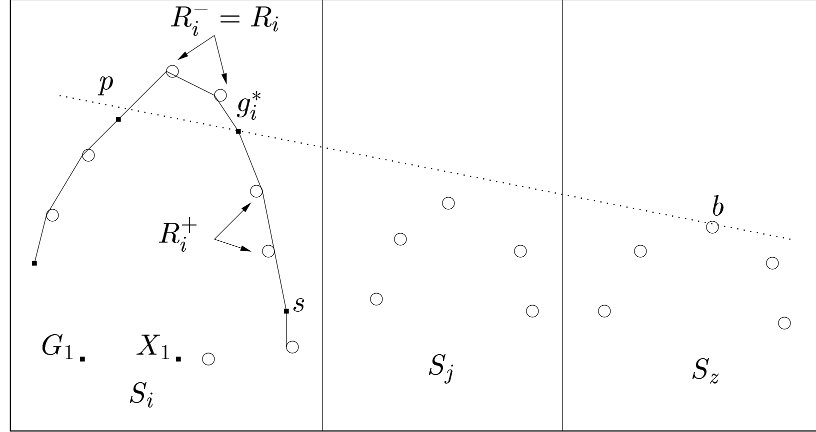

Figura 5.3: Os pontos pretos são elementos de G_i que é um subconjunto de X_i composto de pontos circulares e pretos. Seja $p = pred_{G_i}(g_i^*)$ e $s = suc_{G_i}(g_i^*)$. Temos $R_i = R_i^-$ porque R_i^+ não tem nenhum ponto acima da reta$(g_i^* b)$.

Lema 5.2.2 *Os pontos de R_i^+ ou R_i^-, ou ambos, estão abaixo da reta $(g_i^* Next_{X_{j,j>i}}(g_i^*))$.*

A prova desse lema é direta, senão $g_i^* \notin X_i$.

Definição 5.2.8 *R_i denota o conjunto R_i^+ ou R_i^- que tem pelo menos um ponto acima da reta $(g_i^* Next_{X_{j,j>i}}(g_i^*))$.*

Os conjuntos R_i^+, R_i^- e R_i são mostrados na figura 5.3.

Observamos que o tamanho de cada um dos conjuntos R_i é limitado pelo número de pontos cercados entre dois pontos consecutivos em G_i.

5.2.3 Cálculo do g_i^* e x_i^* em Paralelo

Mostraremos como calcular $g_i^* = lm(G_i)$, onde $G_i \subseteq X_i$. O algoritmo, descrito abaixo, funciona da seguinte forma. Os elementos de G_i são enviados, no passo 2, aos processadores com numeração maior de maneira que eles, recebendo $G = \cup G_j, j < i$,

podem calcular seqüencialmente $Next_{\Delta_i}(g)$ para cada $g \in G$. No passo 3, estes pontos são calculados usando o algoritmo *QueryFindNext*, que são mandados de volta no passo 4. Então, os processadores podem calcular independentemente os g_i^*. Quando os processadores estão divididos em grupos, faça q_z^i denotar o z-ésimo processador do grupo i.

Algoritmo: *FindLMSubset*($\Delta_i, k, w, G_i, g_i^*$)
Entrada: Fechos superiores Δ_i, $1 \leq i \leq p^k$ representando cada um em p^w
processadores numerados consecutivamente $q_z^i, 1 \leq z \leq p^w$ e
o conjunto $G_i \subseteq \Delta_i$.
Saída: O ponto $g_i^* = lm(G_i)$. g_i^* vai estar em cada $q_z^i, 1 \leq z \leq p^w$

1. Junte G_i no processador q_1^i, para todo i.
2. Cada processador q_1^i manda seu G_i a todos processadores q_z^j, $j > i, 1 \leq z \leq p^w$.
Cada processador q_z^i, para todo i, z, recebe $\mathcal{G}^i = \cup G_j, \forall j < i$.
3. Cada processador $q_z^i, \forall i, z$, calcula seqüencialmente $Next_{\Delta_i}(g)$ para cada $g \in \mathcal{G}^i$, usando o algoritmo *QueryFindNext*.
4. Cada processador $q_z^i, \forall i, z$, manda de volta a todos processadores $q_1^j, j < i$,
o $Next_{\Delta_i}(g)$ calculado, $\forall g \in G_j$.
Cada processador $q_1^i, \forall i$, recebe para cada $g \in G_i$, o $Next_{\Delta_j}(g)$ calculado,
$\forall j > i$.
5. Cada processador $q_1^i, \forall i$, calcula para cada $g \in G_i$ o segmento de reta com a maior inclinação entre $\overline{gsuc_{G_i}(g)}$ e $\overline{gNext_{\Delta_j}(g)}$, $j > i$, achando $Next_{G_i \cup \Delta_j, j>i}(g)$. Então, calcula $g_i^* = lm(G_i)$.
6. Cada processador $q_1^i, \forall i$, faz *broadcasts* de g_i^* a $q_z^i, 1 \leq z \leq p^w$.

Lema 5.2.3 *O algoritmo FindLMSubset calcula* $g^* = lm(G_i)$ *em um número constante de rodadas de comunicação. Requer um espaço de memória local* $\frac{n}{p} \geq p^k|G_i|$.

Prova. A corretude desse algoritmo depende das definições 5.2.4, 5.2.5 e do algoritmo de busca QueryFindNext. Os espaços de memórias requeridos são: $|G_i|$ no passo 1, $p^k|G_i|$ no passo 2, p^k no passo 4 equivalendo a um espaço total de $O(p^k|G_i|)$. A complexidade das operações seqüencias é $O(p^k|G_i|\log n)$ e precisa somente de quatro rodadas de comunicação. ∎

5.2.4 Algoritmo MergeHulls 1

Esse algoritmo combina p fechos superiores, armazenados numa *CGM* de p processadores, em um único fecho superior usando um número constante de rodadas de comunicação global. Ele requer $\frac{n}{p} \geq p^2$ e isso limita a escalabilidade.

Para achar a tangente comum superior entre os fechos superiores X_i e X_j (à sua direita), o algoritmo calcula a função $Next$, não para todo X_i mas apenas para o subconjunto de p pontos igualmente espaçados de X_i. Chamamos esse subconjunto de pontos igualmente espaçados um *divisor* de X_i. Essa estratégia baseada no divisor reduz bastante o total de dados que deve ser comunicado entre processadores sem aumentar muito o número de rodadas de comunicação global.

Algoritmo: *MergeHulls 1*$(X_i(1 \leq i \leq p), S, n, p, UH)$
Entrada: O conjunto de p fechos superiores X_i consistindo, no máximo, de n pontos de S, onde X_i é armazenado no processador $q_i, 1 \leq i \leq p$.
Saída: A representação do fecho superior de S. Todos os pontos sobre o fecho superior estão identificados e numerados da esquerda à direita.

1. Cada processador q_i identifica sequencialmente um conjunto divisor G_i composto de p pontos espaçados igualmente de X_i.

2. Os processadores acham em paralelo $g_i^* = lm(G_i)$. Isso é feito via FindLMSubset.

3. Cada processador q_i calcula seus próprios R_i^- e R_i^+ de acordo com a definição 5.2.7, e o conjunto R_i de pontos que estão acima da reta $(g_i^* Next_{G_i \cup X_j, j>i}(g_i^*))$, de acordo com o Lema 5.2.2.

4. Os processadores acham em paralelo $x_i^* = lm(R_i \cup g_i^*)$, usando o algoritmo FindLMSubset. Note que por definição $Next_S(x_i^*) \notin X_i$.

5. Cada processador q_i faz *broadcasts* de seu ponto $Next_S(x_i^*)$ a $q_j, j > i$, e calcula seu próprio S_i' de acordo a definição 5.2.6.

Lema 5.2.4 *O algoritmo MergeHulls 1 calcula UH(S) em tempo* $O(\frac{n \log n}{p} + T_s(n,p))$, *requer um espaço de memória local igual a* $\frac{n}{p} \geq p^2$ *e um número constante de rodadas de comunicação.*

Prova. Pelo teorema 5.2.1, os conjuntos calculados S_i' no passo 5 são uma representação do fecho superior $UH(S)$. O algoritmo MergeHulls 1 chama o procedimento FindLMSubset duas vezes. No passo 2, os parâmetros são $\Delta_i = X_i$, $k = 1$, $w = 0$ e

$G_i = G_i$, implicando que $|G_i| = p$. No passp 4, têm novamente os mesmos parâmetros, mas para $G_i = R_i \cup g_i^*$. Portanto, $|G_i| = |R_i \cup g_i^*| = \frac{|X_i|}{p} \leq \frac{n}{p^2}$. Com isso, o espaço de memória local requerido é $\frac{n}{p} \geq p^2$, pelo lema 5.2.3 O mesmo lema garante que a complexidade das operações seqüenciais no algoritmo MergeHulls 1 é $O(p^2 \log n) = O(\frac{n}{p} \log n)$ e que um número constante de rodadas de comunicação é usado. ∎

5.2.5 Algoritmo MergeHulls 2

Ao contrário do *MergeHulls 1*, esse algoritmo requer somente $\frac{n}{p} \geq p^\epsilon$ de espaço de memória por processador, $0 < \epsilon \leq 1$. Para simplificar usamos $\epsilon = 1$. O algoritmo funciona assim:

1. Na primeira fase, o algoritmo *MergeHulls 1* é utilizado para encontrar o fecho superior de grupo de $\sqrt{p}$ processadores, com $|G_i|$ igual a $\sqrt{p}$. A complexidade dessa fase é $O(p)$.

2. A segunda fase combina esses $\sqrt{p}$ fechos superiores, cada um de tamanho máximo $\frac{n\sqrt{p}}{p}$, no lugar dos p fechos iniciais. Então, com $|G_i| = \sqrt{p}$, o passo 2 requer somente espaço de tamanho p. Porém, precisamos ser mais cuidadosos, porque o tamanho de cada conjunto R_i é, no pior caso, $\frac{n\sqrt{p}}{p|G_i|}$, implicando que o passo 4 vai precisar de espaço até $\frac{n\sqrt{p}}{p}$, o que é muito. Por isso, uma nova redução desses tamanhos é necessária. Sabemos que os conjuntos R_i também são de fechos superiores e podemos aplicar recursivamente a todos os R_i, o mesmo método usado no algoritmo *MergeHulls 1* para achar $x_i^* = lm(R_i \cup g_i^*)$.

Algoritmo: *BuildHulls*($\Phi_i, \Psi_j, p, k, w, \epsilon$)

Entrada: Fechos superiores $\Phi_i, 1 \leq i \leq \frac{p}{p^w}$, representado cada em p^w processadores númerados consecutivamente $q_z^i, 1 \leq z \leq p^w$. O parâmetro ϵ reflete o tamanho da memória local de cada um dos p processadores.

Saída: Fechos convexos $\Psi_j = \text{UH}(\cup_{i=(j-1)p^k+1}^{jp^k} \Phi_i)$, para $1 \leq j \leq \frac{p}{p^{w+k}}$.

1. $G_i \longleftarrow \Phi_i$.

2. While $|G_i|p^k > p^\epsilon$ **do**

(a)Cada grupo de processadores $q_z^i, 1 \leq z \leq p^w$, identifica sequencialmente um conjunto divisor G_i' composto de $p^{\epsilon/2}$ pontos espaçados igualmente de G_i.

(b)FindLMSubset($\Phi_i, k, w, G_i', g_i^*$).

(c)Se $\overline{g_i^* suc_{\Phi_i}(g_i^*)}$ está acima do $\overline{g_i^* Next_{\Phi_i, j>i}(g_i^*)}$ então R_i é composto

de todos pontos de Φ_i entre g_i^* e $suc_{G_i}(g_i^*)$; senão R_i é composto de todos pontos de Φ_i entre g_i^* e $pred_{G_i}(g_i^*)$.

(d)Seja $G_i \longleftarrow R_i \cup \{g_i^*\}$.

3. FindLMSubset(Φ_i, k, w, G_i, x_i^*).

4. Cada processador q_1^i faz *broadcasts* de seu x_i^* a todos $q_j^h, h = (i \bmod (p^k+1))p^k + 1, \cdots, (i \bmod (p^k+1))p^k + p^k$ e $1 \leq j \leq p^w$.

5. Cada processador calcula seu próprio S_i' de acordo a definição 5.2.6.

Lema 5.2.5 *O algoritmo BuildHulls calcula $\Psi_j = UH(\cup_{i=(j-1)p^k+1}^{jp^k}\Phi_i)$, para $1 \leq j \leq \frac{p}{p^{w+k}}$ em tempo $O(\frac{n \log n}{p} + T_s(n,p))$. Seja $\alpha = max\{(k+\epsilon/2), \epsilon\}$. Então o algoritmo requer um espaço de memória local igual a $\frac{n}{p} \geq p^\alpha$ e um número constante de rodadas de comunicação.*

Prova. O algoritmo BuildHulls exige a memória como seguir. No passo 2(b), $\frac{n}{p} \geq p^{k+\epsilon/2}$. No passo 3, $\frac{n}{p} \geq p^\epsilon$ e no passo 4, $\frac{n}{p} \geq p^k$. A complexidade de tempo da computação é $O(\frac{n}{p}\log n + p^k|G_i|) = O(\frac{n}{p}\log n)$. Com respeito ao número de rodadas de comunicação, note que no passo 2 existem no máximo $p^{\alpha+w}$ pontos em Φ_i. Cada passagem no passo 2(a) reduz esse tamanho pelo fator de $p^{\epsilon/2}$. Com isso, existem $\frac{2(\alpha+w)}{\epsilon}$ fases, onde uma fase é chamada no procedimento FindLMSubset. ∎

Algoritmo: *MergeHulls 2($X_i(1 \leq i \leq p), S, n, p, UH(S), \epsilon$)*

Entrada: O conjunto de p fechos superiores X_i consistindo, no máximo, de n pontos de S, onde X_i está armazenado no processador $p_i, 1 \leq i \leq p$. O parâmetro ϵ reflete o tamanho da memória local.

Saída: A representação do fecho superior de S. Todos os pontos sobre o fecho superior são identificados e numerados da esquerda à direita.

1. Seja $k = \frac{\epsilon}{2}, w = 0$, e $X_i^0 = X_i$.

2. **Do**

(a) BuildHulls($X_i^w, X_j^{w+k}, p, k, w, \epsilon$).

(b) $w \longleftarrow w + k$.

Until $w \geq 1$.

Teorema 5.2.6 *O algoritmo MergeHulls 2 calcula UH(S) em tempo $O(\frac{n \log n}{p} + T_s(n,p))$, requer um espaço de memória local $\frac{n}{p} \geq p^\epsilon$ e um número constante de rodadas de comunicação.*

Prova. O lema 5.2.5 implica que $\frac{n}{p} \geq p^{k+\epsilon/2}$, somente $\frac{n}{p} \geq p^{\epsilon}$. Com respeito ao número de rodadas de comunicação, o procedimento BuildHulls é chamada $\frac{2}{\epsilon}$ vezes. Pelo lema 5.2.5, existe $\frac{2(\epsilon+w)}{\epsilon}$ fases, em cada chamada. Conseqüentemente, como $w = (t-1)\frac{\epsilon}{2}$ na t-ésima execução do passo 2(a), o número total de rodadas de comunicação é $\sum_{t=1}^{2/\epsilon} t = O((\frac{2}{\epsilon})^2)$. Finalmente, isso implica que a complexidade de tempo da computação local é $O(\frac{1}{\epsilon^2}\frac{n}{p}\log n)$. ∎

Corolário 5.2.7 *O fecho convexo de n pontos no plano pode ser calculado no $CGM(n,p)$ em tempo $O(\frac{T_{sequencial}}{p} + T_s(n,p))$, onde $T_s(n,p)$ se refere ao tempo de ordenação global de n dados sobre p processadores. Além disso, envolve só um número constante de rodadas de comunicação e $\frac{n}{p} \geq p^{\epsilon}$, para $\epsilon > 0$ constante e arbitrariamente pequeno.*

5.3 Implementações Paralelas

5.3.1 Implementação do Algoritmo Paralelo Probabilístico

O algoritmo *Algoritmo 2* foi implementado [14] sobre a máquina CM5, com 32 processadores, cada um com 40 MB de memória. Foi usado como algoritmo seqüencial, o algoritmo *Quickhull*. As implementações foram feitas para achar o fecho convexo nas dimensões 3 e 4.

O número de rodadas de comunicação observado, para todos os pontos, foi sempre idêntico para os 20 testes.

Quando $\frac{n}{p}$ está entre 256 e 2K, ou 1K e 8K, foi observado para o fecho convexo 3D ou 4D, respectivamente, que o *Algoritmo 2* requer entre 2 e 3 rodadas de comunicação.

Se $\frac{n}{p} \geq 4K$ (em dimensão 3) ou $\frac{n}{p} \geq 16K$ (em dimensão 4), respectivamente, o algoritmo termina sempre depois de uma rodada de comunicação.

5.3.2 Implementação do Algoritmo Paralelo Determinístico

As implementações do algoritmo paralelo determinístico foram feitas [18] na máquina T3D-CRAY, com 128 processadores, usando as rotinas PVM (Parallel Virtual Machine) para as comunicações.

A primeira versão foi implementada com $\frac{n}{p} \geq p^2$. No caso de conjuntos de pontos

aleatórios, foi constatado que a complexidade total aumenta proporcionalmente a n, desde que p é fixo, enquanto a complexidade de comunicação é quase constante.

No pior caso (conjunto linear de pontos), o tempo de computação local cresce proporcionalmente a $n \log n$. O tempo de comunicação total aumenta linearmente com n. O valor de p foi igual a 64.

Com o número de pontos fixos, os conjuntos de pontos aleatórios apresentam um bom comportamento quando p é pequeno. Quando $p \geq 64$, foi observado uma inversão completa da inclinação da curva de tempo total de execução, isto significa que o algoritmo precisa de um grande número de processadores para resolver um problema de tamanho pequeno.

O *Speedup*[2] observado, no pior caso, é melhor desde que a quantidade de computação local é suficientemente grande comparada com a comunicação.

O algoritmo foi implementado também no caso onde o número de dados por processador é fixo. No caso de conjunto aleatório, a curva de tempo total é quase paralela àquela de tempo de comunicação. Isto mostra o impacto de comunicação sobre a escalabilídade do algoritmo.

[2] O Speedup de um algoritmo paralelo é definido pela seguinte fórmula: $S_p(n) = T^*(n)/T_p(n)$, onde $T^*(n)$ é o tempo do melhor algoritmo seqüencial do problema, $T_p(n)$ o tempo de um algoritmo paralelo para o problema usando p processadores e n o tamanho da entrada do problema.

Capítulo 6

Implementações Paralelas

Na seção 2.4 apresentamos o algoritmo seqüencial *Quickhull*. Nesse capítulo paralelizamos este algoritmo e mostrarmos a nossa implementação, na máquina paralela *Parsytec PowerXplorer*, do algoritmo *Quickhull* paralelizado para achar o fecho convexo. Esse algoritmo é descrito no modelo *CGM*. A topologia considerada é uma estrela onde os processadores filhos são ligados ao processador raiz. Os pontos usados obedecem a distribuição normal ou de *Gauss*.

Implementamos também o algoritmo paralelo (*Algoritmo 1*) de Dehne [14]. Fazemos ainda uma comparação entre os resultados dos dois algoritmos implementados.

Na seção 6.1 descrevemos as características da máquina *Parsytec PowerXplorer* do *LCPD* do *IME/USP* (Laboratório de Computação Paralela e Distribuída do *IME/USP*). A seção 6.2 trata da descrição do algoritmo *Quickhull Paralelo*. Na seção 6.3 apresentamos os resultados. Há um *Apêndice A* nesse trabalho que contém o código da implementação do algoritmo *Quickhull* paralelizado.

6.1 Características da Máquina

A implementação do algoritmo foi feita na máquina paralela *Parsytec PowerXplorer*. Essa máquina tem 16 processadores, interconectados por uma topologia de grade bidimensional. Cada nó é formado por um *PowerPC 601* para a computação e de um *Transputer T805* para a comunicação, que compartilham uma memória local de 32 *MBytes*.

Os processadores são agrupados em 4 partições de 4 nós cada, chamadas A, B, C

e D. Uma vez alocada, uma partição (ou mais) é dada para se rodar uma aplicação e permanece exclusiva para esta aplicação. As outras partições não usadas ficam livres para outras aplicações.

A máquina *Parsytec PowerXplorer* usa o sistema operacional *PARIX* que é uma extensão paralela para o sistema operacional *UNIX*. O sistema fornece as ferramentas necessárias para o desenvolvimento de aplicações paralelas. Essas ferramentas rodam em um computador hospedeiro (*host*) e o código produzido roda na *Parsytec PowerXplorer*. O *PARIX* oferece uma extensão para a linguagem C para o desenvolvimento de aplicações paralelas.

O *PARIX* permite que o mesmo código seja executado por todos os processadores, de forma que cada nó executa uma cópia do programa até o fim. O comportamento de cada processador depende do conteúdo do código, da posição de processador e dos dados locais. Como não existe memória global, as informações são trocadas via troca de mensagens.

No ambiente *PARIX*, as comunicações são feitas através de *links* virtuais. As conexões são ponto-a-ponto entre processadores arbitrários da grade. Um conjunto bem definido de *links* virtuais pode ser combinado para construir a topologia virtual como anel, árvore, pipelines, hipercubo, etc.

6.2 Algoritmo Quickhull Paralelo

Paralelizar o algoritmo sequêncial *quickhull* que resolve o problema do fecho convexo já estava no plano original da dissertação. Essa paralelização é feita no modelo de computação paralela *CGM*. Apesar de parecer trivial a sua paralelização, não se encontra nenhum resultado de paralelização deste algoritmo na literatura.

A idéia do algoritmo é de achar primeramente os quatro pontos extremos em cada processador. Os pontos que estão no interior de quadrilatero são eliminados. Esses pontos extremos são trocados em cada processador para encontrar os verdadeiros quatro pontos extremos. A fase seguinte consiste a colocar os pontos que sobram em um único processador e achar o fecho convexo. Esse algoritmo precisa de três rodadas de comunicação.

Algoritmo: *Quickhull Paralelo*
Entrada: Um conjunto finito $S = \{p_1, ..., p_n\}$ de pontos. Cada processador P_i

armazena um subconjunto aleatório S_i de n/p pontos de S. S_i são disjuntos.

Saída: O fecho convexo de S.

1. Cada processador acha os quatro pontos extremos.

2. Um processador determinado (P_1) recebe dos outros processadores seus quatro pontos extremos, acha os novos quatro pontos extremos e manda de volta aos demais processadores.

3. Os processadores eliminam os pontos que estão no interior de quadrilátero.

4. Processador P_1 recebe dos outros processadores os pontos que sobram e calcula o fecho convexo.

O algoritmo acima tem complexidade de tempo de computação local $O(\frac{n}{p} \log n)$ e um número constante de rodadas de comunicação.

6.3 Resultados

Utilizamos os dados de entrada segundo uma distribuição normal ou de Gauss. Nessa distribuição, são eliminados quase 97% de pontos depois de achar os quatro pontos extremos.

As tabelas listadas a seguir apresentam os tempos (T_1, T_4, T_8) obtidos na nossa implementação com a máquina usando um, quatro e oito processadores, do algoritmo *Quickhull* paralelo e do algoritmo de Dehne.

n	T_1	T_4		T_8	
		Tempo	Speedup	Tempo	Speedup
8000	93825	46223	2,029	34442	2,724
40000	472804	135078	3,500	80802	5,851
80000	922457	271032	3,403	198921	4,637
120000	1382823	376272	3,675	210574	6,566
160000	1844777	509180	3,623	366931	5,027

Tabela 6.1: Tempos em μs e *Speedup* para achar o fecho convexo de n pontos segundo a distribuição normal com o algoritmo *Quickhull* paralelo.

n	T_1	T_4		T_8	
		Tempo	Speedup	Tempo	Speedup
8000	93825	26036	3,603	16390	5,724
40000	472804	119773	3,947	63596	7,434
80000	922457	232947	3,959	119194	7,739
120000	1382823	348541	3,967	175525	7,878
160000	1844777	467269	3,947	234199	7,876

Tabela 6.2: Tempos em μs e *Speedup* para achar o fecho convexo de n pontos segundo a distribuição normal com o algoritmo de Dehne.

n	T_4		T_8	
	computação	comunicação	computação	comunicação
8000	34647	11576	20941	13501
40000	123547	11531	67033	13769
80000	248588	22444	153033	45888
120000	356089	20183	186683	23891
160000	482293	26887	276435	90496

Tabela 6.3: Tempos em μs da computação e comunicação para achar o fecho convexo de n pontos segundo a distribuição normal com o algoritmo *Quickhull* paralelo.

Os resultados de nossa implementação mostram que as comunicações do algoritmo de Dehne são melhores. Com alta probabilidade, o algoritmo de Dehne necessita de apenas uma rodada de comunicação, em que apenas são transmitidos os pontos dos fechos convexos armazenados em cada processador. Esse número de pontos transmitidos é menor que $\frac{n}{p}$. No algoritmo de *Quickhull* paralelo, são transmitidos os quatro pontos extremos e os pontos que não são eliminados em cada processador.

n	T_4		T_8	
	computação	comunicação	computação	comunicação
8000	24676	1360	13335	3055
40000	118499	1274	60569	3027
80000	231583	1364	115985	3209
120000	347060	1481	172217	3308
160000	465340	1929	228187	6012

Tabela 6.4: Tempos em μs da computação e comunicação para achar o fecho convexo de n pontos segundo a distribuição normal com o algoritmo de Dehne.

Capítulo 7

Considerações Finais

O problema do fecho convexo é um dos problemas da geometria computacional mais estudados em computação paralela. Esse problema tem muitas aplicações em diversas áreas que motivam o estudo dos algoritmos para a sua construção. Nesta dissertação consideramos o problema do fecho convexo de um conjunto de pontos no plano.

No estudo desses algoritmos procuramos compreender os algoritmos seqüenciais de complexidade $O(n \log n)$. Especialmente vimos a estratégia de divisão-e-conquista que serve no projeto de desenvolvimento dos algoritmos paralelos para esse problema. Outros algoritmos seqüenciais podem ser usados na fase de computação local em cada processador.

O principal objetivo desta dissertação foi estudar os algoritmos paralelos para o fecho convexo. Para isso, definimos o modelo de computação paralela que será usado. Apresentamos vários modelos de computação paralela, fazendo uma classificação dos modelos apresentados. Nesse trabalho, usamos os modelos *PRAM* e *CGM* para desenvolver os algoritmos.

No modelo *PRAM* procuramos descrever detalhadamente o algoritmo apresentado em [33], que consideramos ser mais importante dos algoritmos paralelos pioneiros. Sua complexidade de tempo é $O(\log n)$ usando $O(n \log n)$ operações. O algoritmo paralelo descrito mostra como o método de divisão-e-conquista pode ser usado para o projeto de desenvolvimento de um algoritmo paralelo, para o problema do fecho convexo. Em Aggarwal *et al* [2] encontramos um outro algoritmo paralelo para o mesmo problema usando o paradigma de divisão-e-conquista.

Vimos também de modo sucinto alguns algoritmos paralelos mais recentes no mo-

delo *CGM*. O algoritmo de Dehne *et al* [15] visa a eliminar os pontos que estão no interior de cada fecho convexo calculado em cada processador. O algoritmo proposto é escalável, probabilístico e independente da arquitetura. Ele tem, com alta probabilidade, um número constante de rodadas de comunicação e complexidade de tempo $(k+2)(\frac{T_1(n)}{p}+O(\frac{n}{p}))$ para a computação local e $(k+1)(T_{pSum}(p)+T_{compr}(p,n))$ para a comunicação, onde $k=\lceil \frac{1}{2\epsilon}+\frac{1}{2} \rceil$ é uma constante e $\epsilon > 0$ uma constante fixa. $T_1(n)$ denota a complexidade de tempo seqüencial e $T_{pSum}(p)$ é a complexidade paralela para calcular a soma parcial de p números cada um armazenado em um processador. $T_{compr}(p,n)$ é o tempo para comprimir um subconjunto de dados de tamanho $n' \leq n$ em $p' \leq p$ processadores. Em [18], Diallo *et al* descrevam um algoritmo paralelo determinístico, também escalável e independente da topologia de interconexão para o problema do fecho convexo no plano. O algoritmo requer somente um número constante de rodadas de comunicação e tem a complexidade de tempo de computação local $O(\frac{n \log n}{p}+T_s(n,p))$ onde n é o tamanho do problema, p o número de processadores e $T_s(n,p)$ refere a complexidade de tempo de ordenação de n pontos armazenados em p processadores. Zhou, Deng e Dymond, em um trabalho ainda não publicado, propõem um algoritmo paralelo para o fecho convexo de um conjunto de n pontos no plano, de complexidade de tempo local $(O(\frac{n \log n}{p}))$ com o número de rodadas de comunicação ótimo.

Vários artigos relatam algoritmos para o problema do fecho convexo [7, 21, 27, 35, 38, 41]. O artigo de Goodrich [28] mostra técnicas probabilísticas para o problema no modelo *BSP*. Atallah *et al* [8] trata do método de divisão-e-conquista em cascata como um método para o desenvolvimento de algoritmos paralelos.

Neste trabalho foi proposta a paralelizão do algoritmo seqüencial *Quickhull*. A idéia é eliminar os pontos que estão no interior dos quatro pontos extremos. Sua complexidade de tempo de computação local é $(\frac{n}{p}\log n)$ com um número constante de rodadas de comunicação.

Dentre os algoritmos estudados, implementamos na máquina paralela *Parsytec PowerXplorer* o algoritmo de Dehne e o algoritmo proposto. O desempenho obtido mostra que o algoritmo de Dehne é melhor, pois ele consegue eliminar os pontos que estão no interior de cada fecho convexo calculado em cada processador.

Apêndice A

Implementação

Na proposta de nossa dissertação, sugerimos a implementação de um algoritmo paralelo probabilístico. Por isso, implementamos o algoritmo *Quickhull* paralelo. Neste apêndice listamos o código fonte como descrito no capítulo 6 do trabalho.

```
/* ***************************************************************************
 Programa  : emma.c
 Objetivo  : Fecho Convexo
 Autor     : Emmanuel Kayembe Ilunga
*****************************************************************************/
/*              Esse programa usa a topologia STAR                          */
/* o processador raiz recebe dos processadores filhos os vetores deles,     */
/*   depois de eliminar os pontos que estao no interior de quadrilatero     */
?****************************************************************************/
#include <sys/root.h>
#include <virt_top.h>
#include <sys/link.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <sys/comm.h>
#include <sys/rrouter.h>

#define TAM 4
```

```
#define PMAX 1000
#define DESVIO 20
#define X 0
#define Y 1
#define DIM 2 /*dimensao do espaco*/
#define EXIT_FAILURE 1
#define FREE(p) if (p) {free ((char *)p); p = NULL;}
#define NEW(p,type) \
        if ((p=(type *) malloc (sizeof(type))) == NULL) {\
        printf("NEW : out of Memory!\n");\
        exit(EXIT_FAILURE);\
        }

typedef enum{
  FALSO , TRUO
    }boleno;
typedef double tPointd[DIM]; /*tipo ponto real */
typedef tPointd tConjuntd[PMAX]; /*conjunto de pontos reais*/
typedef struct tPilhacelu tsPilha;
typedef tsPilha *tPilha;
struct tPilhacelu {
  tPointd p;
  tPilha next;
};
/* As variaveis globais */
tConjuntd A;/* Vetor inicial */
tPointd *A1, *A2, *A3, *A4;/* Vetores que recebem os pontos de cada regiao */
tPilha AA1, AA2, AA3, AA4;/* Pilhas que recebem os pontos do fecho */
tPointd *vetor_xy;
tPointd v_xy[TAM], v1_xy[TAM];
tPointd vetor_recvxy[TAM];
int ID;
tPointd *vetor1, *vetor2, *vetor3;
/* As funcoes definidas */
void prt(tPointd *v, int taille){
  int j;
```

```
  for (j = 0; j < taille; j++){
    printf(" vetor[%d][%d] = %g, vetor[%d][%d] =%g, procs= %d\n",j,X,v[j][X],j,Y,v
  }
}
/* Generates random numbers according to a gaussian */
/*                distribution                       */
double gausse( double mean, double deviation )
{
  int i;
  double rnd = 0.0;
  for( i = 0  ; i < 12 ; i++ )
    rnd += (double) (rand() % 10000);
  return(mean + (rnd - 6.0) * deviation);
}
/* Acha a area de um triangulo abc */
double Area2(tPointd a, tPointd b, tPointd c)
{
  return (a[X]*b[Y]-a[Y]*b[X]+a[Y]*c[X]-a[X]*c[Y]+b[X]*c[Y]-c[X]*b[Y]);
}
/* Verifica se o ponto c esta' na esquerda do segmento ab */
boleno left(tPointd a, tPointd b, tPointd c)
{
  return (Area2(a,b,c) > 0.00001);
}
/* Desempilha um ponto de uma pilha */
tPilha desempilha(tPilha s)
{
  tPilha top;
  top = s->next;
  FREE(s);
  return top;
}
/* Empilha um ponto na pilha */
tPilha empilha(tPointd p,tPilha top)
{
  tPilha s;
```

```
  NEW(s,tsPilha);
  s->p[X] = p[X];
  s->p[Y] = p[Y];
  s->next = top;
  return s;
}
/* Imprime a pilha */
void PrintPilha(tPilha t)
{
  while (t!= NULL){
    printf(" Ponto[%d]= %g, Ponto[%d]= %g\n",X,t->p[X],Y,t->p[Y]);
    t = t->next;
  }
}
/* Acha o numero de pontos do fecho convexo */
int Numero(tPilha t)
{
  int k;
  k = 0;
  while (t!= NULL){
    k++;
    t = t->next;
  }
  return k;
}
/* Acha o tamanho de um conjunto */
int tamanho(tPointd a2, tPointd b2, int taille2, tConjuntd SS)
{
  int i,k;
  k = 0;
  for( i = 0; i < taille2; i++){
    if (left(a2,b2,SS[i])) k++;
  }
  return k;
}
/* Preenche o conjunto de pontos */
```

```
tPointd *conjunto(tPointd a3, tPointd b3, int taille3, tConjuntd Ss, int contador)
{
  int i,k;
  tPointd *B;
  k = 0;
  B = (tPointd *) malloc((contador) * sizeof(tPointd));
  for( i = 0; i < taille3; i++){
    if (left(a3,b3,Ss[i])) {
      B[k][X] = Ss[i][X];
      B[k][Y] = Ss[i][Y];
      k++;
    }
  }
  return B;
}
/* Acha o ponto com distante maxima de um segmento */
double *maxdist(tPointd a1, tPointd b1, tConjuntd B1, int total)
{
  double area;
  int n, indice;
  double *p;
  area = Area2(a1,b1,B1[0]);
  indice = 0;
  p = (double *) malloc((DIM) * sizeof(double));
  for (n = 1; n < total; n++){
    if (area < Area2(a1,b1,B1[n])) {
      area = Area2(a1,b1,B1[n]);
      indice = n;
    }
  }
  p[X] = B1[indice][X];
  p[Y] = B1[indice][Y];
  return p;
}
/* Funcao quickhull */
tPilha quickhull(tPointd a, tPointd b, tConjuntd S, int taille)
```

```
{
  tPointd *S1;
  tPointd *S2;
  double *c;
  tPilha BB1;
  tPilha BB2;
  tPilha AB;
  int conta1, conta2;
  BB1 = NULL;
  BB2 = NULL;
  AB = NULL;
  if (taille == 0) {
    AB = empilha(a,AB);
    return (AB);
  }
  else {
    c = maxdist(a,b,S,taille);
    /* Divide o problema em dois */
    conta1 = tamanho(a,c,taille,S);
    conta2 = tamanho(c,b,taille,S);
    S1 = (tPointd *) malloc((conta1) * sizeof(tPointd));
    S2 = (tPointd *) malloc((conta2) * sizeof(tPointd));
    S1 = conjunto(a,c,taille,S,conta1);
    S2 = conjunto(c,b,taille,S,conta2);
    /* As chamadas recursivas */
    BB1 = quickhull(a,c,S1,conta1);
    BB2 = quickhull(c,b,S2,conta2);
    while (BB1 != NULL){
      BB2 = empilha(BB1->p,BB2);
      BB1 = desempilha(BB1);
    }
    FREE(S1);
    FREE(S2);
    return (BB2);
  }
}
```

```
/************************ Programa principal *******************/
/***************************************************************/
int main(int argc, char **argv){
  int ind1;
  int iter, tamanho1;
  int total_procs;
  int ID_topologia;  /* identificador da topologia */
  int i, j, k, l, m;
  double valor, valor1;
  StarData_t *starData;
  int contador1, contador2, contador3, contador4, soma;
  int *tama;
  unsigned int tempo_i, tempo_f, tempo_icom, tempo_fcom;
  /* determina o ID do processador e o total de processadores */
  ID = GET_ROOT()->ProcRoot->MyProcID;
  total_procs = GET_ROOT()->ProcRoot->nProcs;
  tama = (int *) malloc((total_procs) * sizeof(int));
  /* cria topologia virtual do tipo anel */
  ID_topologia = MakeStar(42, total_procs,
                        MINSLICE, MAXSLICE,
                        MINSLICE, MAXSLICE,
                        MINSLICE, MAXSLICE );

  if( ID_topologia < 0 ){
    printf("Erro (%d) : processador  %d\n",ID_topologia,ID);
    return( 1 );
  }
  /* Inicializar as pilhas */
  AA1 = NULL;
  AA2 = NULL;
  AA3 = NULL;
  AA4 = NULL;
  tamanho1 = 4*total_procs;
  vetor_xy = (tPointd *) malloc((tamanho1) * sizeof(tPointd));
  /* Preenche o vetor v_xy em cada processador */
  srand(ID*total_procs);
```

```
for( i = 0; i < PMAX; i++ ){
   valor = gausse(100,DESVIO);
   A[i][X] = valor;
   valor1 = gausse(100,DESVIO);
   A[i][Y] = valor1;
}
/* Buscar os pontos extremos em cada processador */
tempo_i = TimeNow();
j = 0;
for( i = 1; i < PMAX; i++ ){
  if (A[j][X] >= A[i][X]) j = i;
}
v_xy[0][X] = A[j][X];
v_xy[0][Y] = A[j][Y];
k = 0;
for( i = 1; i < PMAX; i++ ){
  if (A[k][X] <= A[i][X]) k = i;
}
v_xy[1][X] = A[k][X];
v_xy[1][Y] = A[k][Y];
l = 0;
for( i = 1; i < PMAX; i++ ){
  if (A[l][Y] >= A[i][Y]) l = i;
}
v_xy[2][X] = A[l][X];
v_xy[2][Y] = A[l][Y];
m = 0;
for( i = 1; i < PMAX; i++ ){
  if (A[m][Y] <= A[i][Y]) m = i;
}
v_xy[3][X] = A[m][X];
v_xy[3][Y] = A[m][Y];
tempo_f = TimeNow() - tempo_i;
/*  Os processadores folhas mandam os quatro pontos */
/*  deles para o processador raiz */
tempo_icom = TimeNow();
```

```
starData = GetStar_Data(ID_topologia);
if (starData != NULL){
  switch (starData->status) {
  case STAR_ROOT:
    k = 0;
    iter = 4*starData->id + 3;
    for(ind1 = 4*starData->id; ind1 <= iter; ind1++){
      vetor_xy[ind1][X] = v_xy[k][X];
      vetor_xy[ind1][Y] = v_xy[k][Y];
      k++;
    }
    for(i = 1; i < starData->size; i++){
      Recv(ID_topologia,i,&vetor_recvxy,sizeof(vetor_recvxy));
      iter = 4*i + 3;
      j = 0;
      for(ind1 = 4*i; ind1 <= iter; ind1++){
        vetor_xy[ind1][X] = vetor_recvxy[j][X];
        vetor_xy[ind1][Y] = vetor_recvxy[j][Y];
        j++;
      }
    }
    break;
  case STAR_LEAVE:
    Send(ID_topologia,0,&v_xy,sizeof(v_xy));
    break;
  }
}
tempo_fcom = TimeNow() - tempo_icom;
/* A raiz acha os novos pontos extremos */
tempo_i = TimeNow();
if (starData->status == STAR_ROOT){
  /* Buscar os pontos extremos no processador raiz */
  j = 0;
  for( i = 1; i < tamanho1; i++ ){
    if (vetor_xy[j][X] >= vetor_xy[i][X]) j = i;
  }
```

```
  v1_xy[0][X] = vetor_xy[j][X];
  v1_xy[0][Y] = vetor_xy[j][Y];
  k = 0;
  for( i = 1; i < tamanho1; i++ ){
    if (vetor_xy[k][X] <= vetor_xy[i][X]) k = i;
  }
  v1_xy[1][X] = vetor_xy[k][X];
  v1_xy[1][Y] = vetor_xy[k][Y];
  l = 0;
  for( i = 1; i < tamanho1; i++ ){
    if (vetor_xy[l][Y] >= vetor_xy[i][Y]) l = i;
  }
  v1_xy[2][X] = vetor_xy[l][X];
  v1_xy[2][Y] = vetor_xy[l][Y];
  m = 0;
  for( i = 1; i < tamanho1; i++ ){
    if (vetor_xy[m][Y] <= vetor_xy[i][Y]) m = i;
  }
  v1_xy[3][X] = vetor_xy[m][X];
  v1_xy[3][Y] = vetor_xy[m][Y];
}
tempo_f += TimeNow() - tempo_i;
/* A raiz manda os verdadeiros  pontos extremos para os filhos */
tempo_icom = TimeNow();
if (starData != NULL){
  switch (starData->status) {
  case STAR_ROOT:
    for(i = 1; i < starData->size; i++){
      Send(ID_topologia,i,&v1_xy,sizeof(v_xy));
    }
    break;
  case STAR_LEAVE:
    Recv(ID_topologia,0,&v1_xy,sizeof(v1_xy));
    break;
  }
}
```

```
tempo_fcom += TimeNow() - tempo_icom;
/* Cada processador acha o conjunto de pontos que nao estao no interior */
/* de quadrilatero */
tempo_i = TimeNow();
contador1 = tamanho(v1_xy[0],v1_xy[3],PMAX,A);
contador2 = tamanho(v1_xy[3],v1_xy[1],PMAX,A);
contador3 = tamanho(v1_xy[1],v1_xy[2],PMAX,A);
contador4 = tamanho(v1_xy[2],v1_xy[0],PMAX,A);
soma = contador1 + contador2 + contador3 + contador4;
vetor1 = (tPointd *) malloc((soma) * sizeof(tPointd));
/* Preenche o vetor a mandar */
 k = 0;
for( i = 0; i < PMAX; i++){
  if (left(v1_xy[0],v1_xy[3],A[i])) {
    vetor1[k][X] = A[i][X];
    vetor1[k][Y] = A[i][Y];
    k++;
  }
}
for( i = 0; i < PMAX; i++){
  if (left(v1_xy[3],v1_xy[1],A[i])) {
    vetor1[k][X] = A[i][X];
    vetor1[k][Y] = A[i][Y];
    k++;
  }
}
for( i = 0; i < PMAX; i++){
  if (left(v1_xy[1],v1_xy[2],A[i])) {
    vetor1[k][X] = A[i][X];
    vetor1[k][Y] = A[i][Y];
    k++;
  }
}
for( i = 0; i < PMAX; i++){
  if (left(v1_xy[2],v1_xy[0],A[i])) {
    vetor1[k][X] = A[i][X];
```

```
    vetor1[k][Y] = A[i][Y];
    k++;
  }
}
m = k;
tempo_f += TimeNow() - tempo_i;
/* Aloca dinamicamente o vetor vetor2 que recebera os pontos para encontra */
/*                          o fecho convexo                                */
tempo_icom = TimeNow();
if (starData != NULL){
  switch (starData->status) {
  case STAR_ROOT:
    tama[starData->id] = m;/* A raiz preenche no vetor tama seu valor */
    for(i = 1; i < starData->size; i++){
      Recv(ID_topologia,i,&k,sizeof(int));
      tama[i] = k;
    }
    break;
  case STAR_LEAVE:
    Send(ID_topologia,0,&m,sizeof(int));
    break;
  }
}
tempo_fcom += TimeNow() - tempo_icom;
tempo_i = TimeNow();
if (starData->status == STAR_ROOT){
  soma = 0;
  for (ind1 = 0; ind1 < total_procs; ind1++) {
    printf("tama[%d] = %d\n",ind1,tama[ind1]);
    soma +=tama[ind1];
  }
  vetor2 = (tPointd *) malloc((soma) * sizeof(tPointd));
}
tempo_f += TimeNow() - tempo_i;
/* O processador raiz recebe os pontos de outros processadores */
tempo_icom = TimeNow();
```

```
if (starData != NULL){
  switch (starData->status) {
  case STAR_ROOT:
    for(ind1 = 0; ind1 < m; ind1++){
      vetor2[ind1][X] = vetor1[ind1][X];
      vetor2[ind1][Y] = vetor1[ind1][Y];
    }
    iter = tama[0];
    for(i = 1; i < starData->size; i++){
      vetor3 = (tPointd *) malloc((tama[i]) * sizeof(tPointd));
      Recv(ID_topologia,i,vetor3,tama[i]*sizeof(tPointd));
      j = 0;
      iter += tama[i];
      k = iter - tama[i];
      for(ind1 = k; ind1 < iter; ind1++){
        vetor2[ind1][X] = vetor3[j][X];
        vetor2[ind1][Y] = vetor3[j][Y];
        j++;
      }
    }
    break;
  case STAR_LEAVE:
    Send(ID_topologia,0,vetor1,m*sizeof(tPointd));
    break;
  }
}
tempo_fcom += TimeNow() - tempo_icom;
/* O processador raiz calcula o fecho convexo */
tempo_i = TimeNow();
if (starData->status == STAR_ROOT){
  /* Determina o tamanho de cada regiao */
  contador1 = tamanho(v1_xy[0],v1_xy[3],soma,vetor2);
  A1 = (tPointd *) malloc((contador1) * sizeof(tPointd));
  contador2 = tamanho(v1_xy[3],v1_xy[1],soma,vetor2);
  A2 = (tPointd *) malloc((contador2) * sizeof(tPointd));
  contador3 = tamanho(v1_xy[1],v1_xy[2],soma,vetor2);
```

```
    A3 = (tPointd *) malloc((contador3) * sizeof(tPointd));
    contador4 = tamanho(v1_xy[2],v1_xy[0],soma,vetor2);
    A4 = (tPointd *) malloc((contador4) * sizeof(tPointd));
    /* Constroi as quatro regioes */
    A1 = conjunto(v1_xy[0],v1_xy[3],soma,vetor2,contador1);
    A2 = conjunto(v1_xy[3],v1_xy[1],soma,vetor2,contador2);
    A3 = conjunto(v1_xy[1],v1_xy[2],soma,vetor2,contador3);
    A4 = conjunto(v1_xy[2],v1_xy[0],soma,vetor2,contador4);
    /* Aplica o algoritmo recursivo para cada uma das regioes */
    AA1 = quickhull(v1_xy[0],v1_xy[3],A1,contador1);
    AA2 = quickhull(v1_xy[3],v1_xy[1],A2,contador2);
    AA3 = quickhull(v1_xy[1],v1_xy[2],A3,contador3);
    AA4 = quickhull(v1_xy[2],v1_xy[0],A4,contador4);
    /* Junta a resposta das 4 regioes  */
    while (AA1 != NULL){
      AA4 = empilha(AA1->p,AA4);
      AA1 = desempilha(AA1);
    }
    while (AA2 != NULL){
      AA4 = empilha(AA2->p,AA4);
      AA2 = desempilha(AA2);
    }
    while (AA3 != NULL){
      AA4 = empilha(AA3->p,AA4);
      AA3 = desempilha(AA3);
    }
    tempo_f += TimeNow() - tempo_i;
    /* Imprime o resultado */
    PrintPilha(AA4);
    m = Numero(AA4);
    printf("O tempo do calculo e’ %u e o tempo de comunicacao e’
  %u\n",tempo_f,tempo_fcom);
  }
  return (0);
}
```

Referências Bibliográficas

[1] *Conception et mise en oeuvre d'application parallèles irregulières de grande taille*, Groupes thématiques de PRS, Capa - Rumeur- Exec, CNRS, 1997.

[2] A. Aggarwal, B. Chazelle, L. J. Guibas, C. O'Dunlaing and C. K. Yap, Parallel computational geometry, *Algorithmica* (1988), no. 3, 293–327.

[3] S. G. Akl, A constant-time parallel algorithm for computing convex hulls, *BIT* **22** (1982), 130–134.

[4] S. G. Akl, The Design and Analysis of Parallel Algorithms, ch. 12 – Traversing Combinatorial Spaces, pp. 310–340, Prentice-Hall International, Engelwood Cliffs, NJ, 1989, pp. 310–340.

[5] S. G. Akl and K. Lyons, *Parallel computational geometry*, Prentice Hall, 1993.

[6] S. G. Akl and G. T. Toussaint, Efficient convex hull algorithms for pattern recognition applications, *Proc. 4th International Joint Conf. on Pattern Recognition, Kyoto, Japan* (1979), 483–487.

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

[8] M. J. Atallah, R. Cole and M. T. Goodrich, Cascading divide-and-conquer: A technique for designing parallel algorithms, *SIAM Journal on Computing* **18** (1989), no. 3, 499–532.

[9] T. H. Axford, The divide-and-conquer paradigm as a basis for parallel language design, *Advances in Parallel Algorithms* (L. Kronsjö and D. Shumsheruddin, eds.), Blackwell, London, UK, 1992, pp. 25–65.

[10] A. Bykat, Convex hull of a finite set of points in two dimensions, *Information Processing Letters* **7** (1978), no. 6, 296–298.

[11] A. L. Chow, A parallel algorithm for determining convex hull of sets of points in two dimension, *Proc. of the Nineteenth Annual Allerton Conf. on Communication, Control and Computing* (1981), 214–222.

[12] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to algorithms*, The MIT Press, MacGraw-Hill Book Company, 1990.

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

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

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

[16] F. Dehne and S. W. Song, Randomized parallel list ranking for distributed memory multiprocessors, *ACSC: Asian Computing Science Conference, LNCS*, 1996.

[17] X. Deng and N. Gu, Good algorithm design style for multiprocessors, *In Proc. 6th IEEE Symposium on Parallel and Distributed Processing, Dallas, USA* (October 1994), 538–543.

[18] A. Diallo, A. Ferreira, A. Rau-Chaplin and S. Ubéda, Scalable 2D convex hull and triangulation algorithms for coarse grained multicomputers, *Proc. 7th IEEE Symposium on Parallel and Distributed Processing (SPDD'95)* (1995).

[19] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*, Wiley-Interscience, New York, 1973.

[20] W. F. Eddy, A new convex hull algorithm for planar sets, *ACM Transactions on Mathematical Software* **3** (1977), no. 4, 398–403.

[21] D. J. Evans and Shao-wen Mai, Two parallel algorithms for the convex hull problem in a two dimensional, *Parallel Computing* **2** (1985), no. 4, 313–326.

[22] W. Feller, *An introduction to probability theory and its applications*, John Wiley & Sons, 1950.

[23] A. Ferreira, Parallel and communication algorithms on hypercube multiprocessors, *Handbook of Parallel and Distributed Computing* (Albert Y. Zomaya, ed.), McGraw-Hill, New York, 1995.

[24] L. H. Figueiredo and P. C. P. Carvalho, *Introdução a geometria computacional*, 18 Colóquio Brasileiro de Matemática, IMPA, 1991.

[25] H. Freeman, Computer processing of line-drawing images, *Comput. Surveys* **6** (1974), 57–97.

[26] H. Freeman and R. Shapira, Determining the minimum-area encasing rectangle for an arbitrary closed curve, *Communications of the ACM* **18** (1975), no. 7, 409–413.

[27] M. T. Goodrich, Finding the convex hull of a sorted point set in parallel, *Information Processing Letters* **26** (1987), no. 4, 173–179.

[28] ______, Randomized fully-scalable BSP techniques for multi-searching and convex hull construction (preliminary version), *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana), 5–7 January 1997, pp. 767–776.

[29] R. L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Information Processing Letters* **1** (1972), 132–133.

[30] P. J. Green and B. W. Silverman, Constructing the convex hull of a set of points in the plane, *The Computer Journal* **22** (1979), no. 3, 262–266.

[31] S. E. Hambrusch, Models for parallel computation, *ICPP: 25th International Conference on Parallel Processing*, 1996.

[32] R. A. Jarvis, On the identification of the convex hull of a finite set in the plane, *Information Processing Letters* **2** (1973), 18–21.

[33] J. JáJá, *An introduction to parallel algorithms*, Addison-Wesley Publishing Company, 1992.

[34] B. M. Maggs, L. R. Matheson and R. E. Tarjan, Models of parallel computation: a survey and synthesis, *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, vol. 2, 1995, pp. 61–70.

[35] S. N. Maheshwari and P. C. P. Bhatt, Parallel algorithms for the convex hull problems in two dimensions, *Conf. Analy. Prob. Classes Program. Parallel Comput.*, Lecture Notes in Computer Science, vol. 111, Springer-Verlag, 1981, pp. 358–372.

[36] E. W. Mayr and R. Werchner, Divide-and-conquer algorithms on the hypercube, *Theoretical Computer Science* **162** (1996), no. 2, 283–296.

[37] R. Miller and Q. Stout, Computational geometry on a mesh-connected computer (preliminary version), *Proc. of the 1984 International Conference on Parallel Processing* (1984), 236–271.

[38] ______, Efficent parallel convex hull algorithms, *IEEE Transactions on Computers* **37(12)** (1988), 1605–1618.

[39] J. O'Rourke, *Computational geometry in c*, Cambridge University Press, Cambridge, 1993.

[40] C. H. Papadimitriou, *Computational complexity*, Addison-Wesley, New York, 1994.

[41] W. Preilowski, E. Dahlhaus and G. Wechsung, New parallel algorithms for convex hull and triangulation in 3–dimensional space, *Proceedings of MFCS '92. Mathematical Foundations of Computer Science (MFCS '92)* (Berlin, Germany) (Ivan M. Havel and Vaclav Koubek, eds.), LNCS, vol. 629, Springer, August 1992, pp. 442–450.

[42] F. P. Preparata and S. J. Hong, Convex hulls of finite setes os points in two and three dimensions, *Commun. ACM* (1977), no. 20, 87–93.

[43] F. P. Preparata and M. I. Shamos, *Computational geometry: An introduction, texts and monographs in computer science*, Springer-Verlag, New York, 1985.

[44] J. H. Reif, *Synthesis of parallel algorithms*, Morgan Kaufmann, San Mateo, 1993.

[45] A. Rosenfeld, *Picture processing by computers*, Academic Press, New York, 1969.

[46] J. Sklansky, Measuring concavity on a rectangular mosaic, *IEEE Trans. Comp.* **C-21** (1972), 1355–1364.

[47] L. Valiant, A bridging model for parallel computation, *Communications of the ACM* (1990), no. 33, 103–111.