Universidade Federal de São João del-Rei

Estratégias de paralelização em algoritmos para calcular o *Convex Hull*

Algoritmos e Estruturas de Dados III

Isabella Vieira Ferreira
Mônica Neli de Resende

1 Introdução

O termo Geometria Computacional se refere ao estudo de algoritmos e estruturas de dados para a solução computacional de problemas geométricos. Os problemas são tratados em termos de objetos elementares como pontos, retas, segmentos de reta, polígonos etc. Em geral, o objetivo é resolver tais problemas de forma eficiente, isto é, utilizando o menor número possível de operações simples.

A Geometria Computacional estuda tanto problemas geométricos clássicos, como também problemas motivados por diversas áreas da computação como computação gráfica, robótica, sistemas de informação geográfica, visão computacional, otimização combinatória, processamento de imagens, teoria dos grafos, desenho de circuitos integrados, aprendizagem de máquina, etc. Um dos problemas geométricos é o fecho convexo (do inglês *Convex Hull*). O fecho convexo de um conjunto de pontos é um polígono convexo que envolve todo o conjunto minimizando o número de vértices.

Constatado a importância da Geometria Computacional é comum que seja essencial que os algoritmos sejam eficientes e rápidos. Produzir soluções computacionais que reduzam o tempo de processamento são um dos grandes desafios da Ciência da Computação atualmente, cenário atuante da Computação Paralela.

A computação paralela é uma técnica que atua realizando os cálculos simultaneamente, operando sob o princípio de que grandes problemas geralmente podem ser divididos em problemas menores, que então são resolvidos concorrentemente (em paralelo). Os problemas que podem ser paralelizados são desde bits, instruções, dados ou tarefas. A técnica de paralelismo já é empregada por vários anos, principalmente na computação de alto desempenho, mas recentemente o interesse no tema cresceu devido às limitações físicas que dificultam o aumento da frequência de processamento. Com o aumento da preocupação do consumo de energia dos computadores, a computação paralela se tornou o paradigma dominante nas arquiteturas de computadores sob forma de processadores multinúcleo.

O objetivo desse trabalho é portanto implementar e paralelizar dois algoritmos para o cálculo do fecho convexo. Para isso analisaremos cinco algoritmos clássicos, identificaremos as oportunidades ou desvantagens de paralelização em cada um deles e entre as soluções mais promissoras projetaremos uma estratégia utilizando a biblioteca específica para paralelização de processos denominada *pthreads.h*.

A motivação do trabalho deve-se a importância da paralelização, que é uma técnica bastante utilizada para resolver problemas que seriam muito demorados para executar e a aplicabilidade dos problemas geométricos computacionais, tendo em vista o interesse em adquirir conhecimentos.

O desafio é analisar bem as oportunidades de paralelização para assim projetar estratégias de forma que os tempos de execução dos algoritmos paralelizados sejam evidentemente melhores em relação aos algoritmos comuns.

2 Entrada e saída de dados

A entrada de dados do programa é realizada através de um arquivo de texto que poderá conter um ou mais casos de teste. A primeira linha de cada caso possuirá o número de pontos a serem lidos e nas linhas seguintes as coordenadas x e y de cada ponto. Cada linha conterá apenas um ponto, sendo que as coordenadas deste deverão estar separadas por espaço. Caso haja mais de um caso de teste, este deverá estar na mesma forma descrita acima, iniciando-se na linha seguinte ao último ponto do caso anterior. O final do arquivo é marcado com zero seguido de uma quebra de linha, indicando que não há mais casos a serem processados.

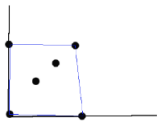
O programa criará um arquivo de texto denominado *"saida.out"* onde serão armazenados os fechos convexos. O arquivo conterá, para cada caso de teste lido do arquivo de entrada, o conjunto de pontos (ordenados no sentido anti-horário) correspondente aos vértices que compõem o polígono ou a palavra "impossivel" se não houver solução. Ressaltamos que caso haja mais de uma solução, apenas uma delas será apresentada. A Figura 1 apresenta um exemplo de arquivo de entrada e de saída.

```
6
3 3
2 2
0 0
4 4
4 0
0 4
3
1 1
2 2
3 3
0
```

(a) Exemplo de Entrada

```
Teste 1
0 0
4 0
4 4
0 4
Teste 2
impossivel
```

(b) Saída correspondente



(c) Representação gráfica da solução

Figura 1: Exemplo de entrada e saída

3 Estrutura de dados utilizada

Para a representação do conjunto de pontos foi utilizado um vetor alocado dinamicamente, onde cada célula armazena em seu respectivo campo a coordenada x e y, como apresentado na Figura 2.

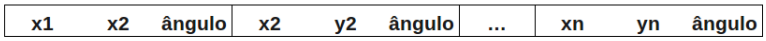


Figura 2: Ilustração da estrutura de dados

A Figura 3 apresenta a estutura utilizada para a passagem de parâmetros para as *threads*.

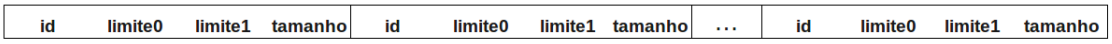


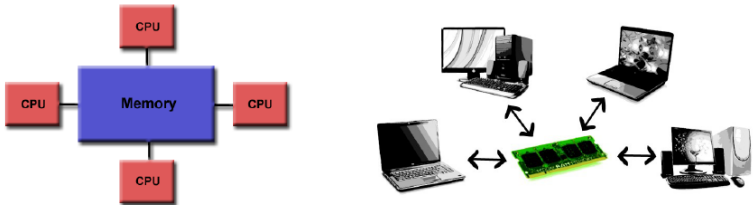
Figura 3: Ilustração da estrutura de dados para o vetor de argumentos das *threads*

4 Computação Paralela

A computação paralela é uma estratégia onde os cálculos são divididos e computados simultaneamente. Neste trabalho utilizaremos a *Pthreads*, uma API (Application Programming Interface) criada no padrão POSIX e por isto leva o seu nome POSIX *Threads*. O padrão POSIX – *Portable Operation System Interface* foi definido para padronizar as interfaces do UNIX e permitir a portabilidade das aplicações para as várias versões do sistema criado pelo Instituto de Engenheiros Elétricos e Eletrônicos – IEEE. Esse padrão define o conjunto de funções que as bibliotecas devem oferecer e permite que as aplicações possam ser executadas em qualquer sistema operacional que siga suas recomendações.

Algoritmos paralelos são muito mais complicados de serem implementados do que algoritmos sequenciais, pois como existem processos sendo executados em paralelo, novos erros podem ocorrer, como a condição de corrida dos processos. A comunicação e a sincronização dos diferentes processos sendo executados é a maior barreira para se atingir um ótimo desempenho em programas paralelizados.

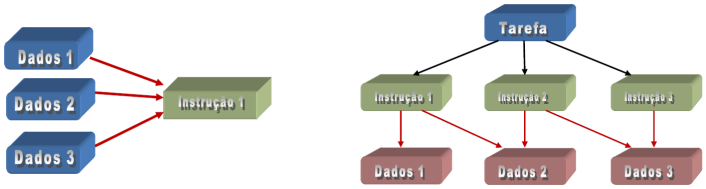
A paralelização de algoritmos é dividida basicamente em dois métodos distintos. Primeiramente, temos o compartilhamento de dados em uma mesma área de memória e em segundo lugar, temos que cada processo possui as suas próprias variáveis em memórias distintas. A Figura 4 ilustra o compartilhamento de dados em memória compartilhada e distribuída.



(a) Memória Compartilhada (b) Memória Distribuída

Figura 4: Compartilhamento de dados

Além disso, há outras duas divisões que podem ser aplicadas em ambos os métodos citados acima: a paralelização de dados e paralelização de controle. O paralelismo de dados é o uso de múltiplas unidades para aplicar a mesma operação (simultaneamente) a elementos do conjunto de dados. O paralelismo de controle trata-se da computação dividida em estágios ou segmentos.



(a) Paralelismo de Dados (b) Paralelismo de Controle

Figura 5: Estratégia de Paralelismo

Considerando o nosso problema de calcular o fecho convexo temos que o modelo utilizado será o de paralelização de dados. A arquitetura utilizada será a SIMD (*Single Instruction Multiple Data*), ou seja, vários processadores sincronizados operando segundo o modelo PRAM padrão (*Parallel Randon Acess Machine*). Tal modelo permite a leitura concorrente, ou seja, vários processos podem ler da memória global durante mesma instrução.

Utilizamos a linguagem de programação C, juntamente com uma biblioteca específica para a paralelização de processos denominada *pthreads.h*, a qual nos fornece funções para a criação dos processos, chamados *threads*, bem como funções para o controle e sincronização das mesmas, evitando alguns erros como condições de corridas e *deadlock's*.

5 *Convex Hull*

O *Convex Hull*, também chamado de casco convexo, é o conjunto mínimo de vértices que formam um polígono convexo contendo um conjunto de pontos no plano euclidiano. Vale ressaltar, que no plano euclidiano, um objeto é convexo se, para cada par de pontos no interior do objeto, cada ponto do segmento que os une também está dentro do objeto. A Figura 6 apresenta exemplo de um conjunto convexo e um conjunto não convexo.

Figura 6: Exemplo de conjunto convexo e não convexo

Apresentaremos nesta seção cinco algoritmos clássicos para calcular o *Convex Hull*. Avaliaremos as oportunidades de paralelização de cada um e ressaltamos que iremos trabalhar apenas com a geometria em duas dimensões.

5.1 Análise dos algoritmos para calcular o *Convex Hull*

Para determinar o *Convex Hull* de forma eficiente e rápida foram analisados os algoritmos: Gift Wrapping, QuickHull, Divisão e Conquisa, Graham Scan e Monotone Chain.

Estudamos e analisamos os cinco algoritmos e percebemos que as oportunidades de paralelização são mais promissoras no Graham Scan e no Monotone Chain (a explicação detalhada está na subseção 5.1.2). A subseção 5.1.1 explicará detalhadamente sobre os algoritmos Gift Wrapping, QuickHull e Divisão e Conquista, além de apresentar também as desvantagens em cada um deles, o que inviabilizou a paralelização.

5.1.1 Algoritmos com ausência da oportunidade de paralelização

Gift Wrapping

O algoritmo Gift Wrapping foi desenvolvido por Jarvis em 1973 e é também conhecido como algoritmo de Jarvis. A idéia do método guloso é adicionar um elemento de cada vez na solução e, jamais, remover algum elemento dela. É análogo ao algoritmo de ordenação *"Selection Sort"*, a cada passo, escolhe o menor dos valores e acrescenta à coleção ordenada.

O algoritmo baseia-se na observação que se pq é uma aresta do fecho convexo, então a próxima aresta do fecho é qr, onde r é um ponto tal que qr define o menor ângulo com pq. Caso houver um empate escolhe-se o ponto com maior coordenada x.

Inicia-se de um vértice p_0 de menor coordenada e o ponto p_1 é encontrado no conjunto com o menor ângulo polar no sentido anti-horário com relação ao ponto p_0 e a reta horizontal. Em seguida, basta utilizar a aresta encontrada para encontrarmos a próxima aresta que forma o menor ângulo com a anterior.

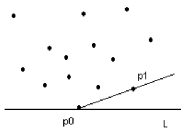


Figura 7: Seleção dos pontos do fecho através da escolha do ponto que forma o menor ângulo com a aresta anterior

Esse passo é repetido até retornarmos ao ponto inicial. Caso tenhamos mais de um ponto com o mesmo valor de ângulo, devemos escolher, dentre esses, o mais distante.

O que inviabiliza a paralelização desse algoritmo é o fato dele precisar de resultados feitos em cálculos anteriores, o que faria com que uma *thread* ficasse esperando a outra terminar, onde no pior caso o algoritmo será praticamente sequencial. Uma estratégia seria dividir o vetor com relação ao número de *threads*, onde cada uma elegeria o ponto de menor coordenada de sua partição. Porém, essa não é a parte mais custosa do algoritmo, logo a estratégia não apresentaria grandes resultados.

Algoritmo Quickhull

O algoritmo foi proposto independentemente por várias pessoas quase ao mesmo tempo e devido a semelhança com o algoritmo Quicksort foi denominado Quickhull por Preparata e Shamos.

A idéia do algoritmo se baseia no fato de que é mais fácil descartar muitos pontos que estão no interior do fecho convexo e concentrar o trabalho nos pontos que estão próximos da fronteira. Para isso, o algoritmo encontra os 4 pontos extremos(máximos e mínimos de x e y) que garantidamente fazem parte do fecho convexo e descarta os pontos no interior do quadrilátero, como é mostrado na Figura 8.

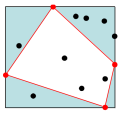
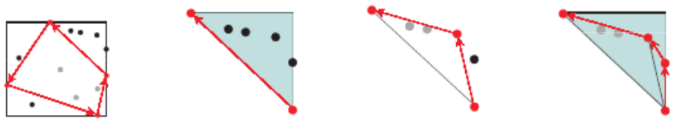


Figura 8: Quadrilátero formado pela escolha dos extremos

Em seguida o problema é dividido em 4 grupos, cada um associado a uma aresta, onde são resolvidos recursivamente. Para cada aresta do quadrilátero, o algoritmo prossegue elegendo um ponto c_i do grupo que é um vértice do fecho convexo. O método mais usado consiste em escolher o ponto mais distante da aresta em questão.

Dessa forma, são formados mais dois subproblemas devido aos segmentos formados pelos extremos do segmento original até o ponto c_i. Tais subproblemas são solucionados da mesma forma até que não haja nenhum ponto à esquerda do segmento. Por fim as soluções dos subproblemas são combinadas resultando na solução para o fecho convexo.

A desvantagem desse algoritmo está na recursão, onde a cada chamada recursiva ocorre uma subdivisão dos "problemas". Apesar do paradigma de "Divisão e conquista" ser uma oportunidade de paralelização, o número de *threads* poderia ser insuficiente com relação ao número de subdivisões e quando isso acontecer uma parte do problema terá que ficar esperando a outra acabar de ser processada. Nesse caso teríamos dificuldade em controlar o que cada *thread* faria, de forma que não acontece um desbalanceamento ou *deadlock*.



(a) Quadrilátero (b) Subproblema 1 (c) Elegendo c (d) Novo subproblema

Figura 9: Divisão do problema em subproblemas

(a) (b) (c)

Figura 10: Divisão em novos subproblemas e combinação de soluções

Algoritmo Dividir para Conquistar

A técnica de divisão e conquista é muito utilizada no projeto de algoritmos. Um exemplo é o algoritmo de ordenação MergeSort, sendo o algoritmo Dividir para Conquistar semelhante a ele.

A idéia para encontrar o fecho convexo é dividir o problema em 2 subproblemas de tamanho aproximadamente igual, encontrar recursivamente a solução dos subproblemas e combinar as soluções dos subproblemas para obter a solução final.

Para isso, os casos básicos onde o conjunto tem 3 pontos ou menos são resolvido trivialmente. Após a divisão os pontos são ordenados considerando a abcissa e divididos em dois subconjuntos A e B, cada um com aproximadamente a metade dos pontos através da mediana de x. Em seguida, deve-se encontrar recursivamente os fechos convexos de A e B. Por fim após a identificação das tangentes inferiores e superiores, os pontos entre elas são removidos de forma a produzir a solução final.

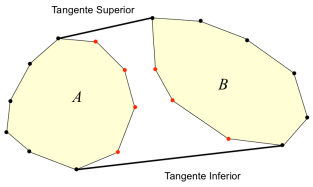


Figura 11: Divisão e Conquista

A remoção dos pontos entre as tangentes é feita após calculá-las, podendo inclusive ser feita durante o seu cálculo.

Não optamos por paralelizar este algoritmo devido à dificuldade de implementação sequencial, sendo que as oportunidades de paralelização teriam que ser analisadas após sua implentação. Assim como o algoritmo QuickHull o que também inviabilizaria o paralelismo é a técnica de dividir para conquistar, onde teríamos dificuldade em controlar o que cada *thread* faria, de forma que não acontecesse um desbalanceamento ou *deadlock*.

5.1.2 Algoritmos com oportunidade de paralelização

Algoritmo de Graham Scan

Desenvolvido em 1972 por R. L. Graham o algoritmo foi o primeiro em envoltória convexa a rodar em O(n log n).

A fase de pré-processamento consiste em escolher o pivô como sendo o ponto com a menor coordenada y pertencente à envoltória convexa ou o ponto com menor coordenada x se houver um empate. Em seguida, ordenar todos os pontos restantes em relação ao ângulo polar, ou seja, o ângulo que o segmento p_0p_i faz com o eixo x.

A ordenação pode ser feita utilizando qualquer algoritmo de ordenação. A etapa de pré-processamento reduz o problema de encontrar o invólucro convexo de um conjunto de pontos ao problema de encontrar o invólucro convexo de um polígono estrelado.

O algoritmo consiste em formar o menor caminho fechado no polígono, respeitando os ângulos e o pivô, através de uma pilha para manter os pontos do fecho convexo. Primeiramente o pivô e o primeiro ponto são empilhados e para cada ponto seguinte é verificado se ocorre uma curva à direita ou à esquerda. Para isso realiza-se o seguinte cálculo do produto vetorial entre o i-ésimo item do vetor e os dois vetores a partir do topo da pilha:

$$resultado = (x_2 - x_1) * (y_i - y_1) - (y_2 - y_1) * (x_i - x_1);$$

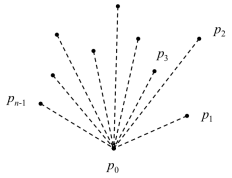
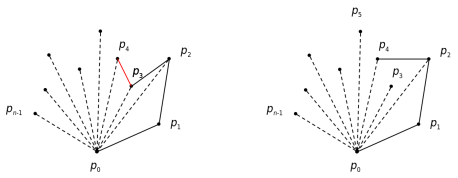


Figura 12: Polígono estrelado

O ponto faz parte do fecho convexo se o resultado for maior que zero, ou seja, se a nova aresta juntamente com a anterior fizerem uma curva à esquerda. Dessa forma, tal i-ésimo ponto é empilhado e o passo é realizado para o próximo ponto do vetor. Do contrário, o ponto do topo da pilha é desempilhado e novamente são feitos o cálculo e a verificação até que ocorra uma curva à esquerda. Ao final do vetor teremos na pilha os pontos do fecho convexo no sentido anti-horário.

Na Figura 13 (a) é mostrado a aresta p_2p_3 e p_3p_4 fazem uma curva voltada para a direita, portanto o ponto p_3 é desempilhado e analisa-se a curva formada pelas arestas p_1p_2 e p_2p_4 como mostrado na Figura 13 (b).



(a) Curva à direita

(b) Curva à esquerda

Figura 13: Determinação do Fecho Convexo através da orientação das arestas

Solução Apresentada

Primeiramente, é necessário ordenar os pontos pelo ângulo polar. Optamos por utilizar o algoritmo de ordenação *Quicksort* de forma não recursiva. Dessa forma, ele é mais eficiente por utilizar uma pequena pilha como memória auxiliar. De forma a acelerar os cálculos, não é necessário calcular os ângulos que estes pontos formam com o eixo x; ao invés disto é suficiente calcular a tangente deste ângulo, que pode ser feito através da função *atan2* em C. Optamos por escolher o pivô que contém a menor coordenada x e caso tenha empate, escolhemos o ponto que possui menor coordenada y. Fizemos essa escolha para que a ordem dos pontos no resultado final não seja diferente do resultado no algoritmo Monotone Chain.

```
Graham Scan

Entrada: um vetor V com os n pontos no plano.

Ordenar os pontos em ordem crescente pelo coordenada ângulo polar.

Empilha  (V[1]);
Empilha  (V[2]);

i= 3;

Enquanto i >= n
     resultado = Produto Vetorial;

     Se (resultado > 0)
          Empilha V[i];
          i++;
     Fim Se
     Senão Se (Tamanho da Pilha >2)
          Desempilha V[i];
    Fim Senão Se
Fim Enquanto

     Se (Tamanho da Pilha > 3)
          i = Tamanho da Pilha – 1;
          Enquanto cont != 0
                cont = Tamanho da Pilha;
                Desempilha V[i];
                i--;
                cont--;
          Fim Enquanto
     Fim Se
```

Figura 14: Pseudocodigo do algoritmo Graham Scan

Algoritmo Monotone Chain

Publicado em 1979 por Andrew A.M, o algoritmo pode ser visto como uma variante de *Graham Scan* que classifica os pontos lexicograficamente pelas suas coordenadas. O algoritmo constrói o casco convexo de um conjunto de pontos em duas dimensões em O(n log n) de tempo, onde n é o número de pontos de entrada.

Primeiramente é feita a escolha dos pontos lexicograficamente (primeiro pela coordenada x e em seguida pela coordenada y). Logo após é feita a construção dos cascos superior e inferior dos pontos. Vale ressaltar, que um casco superior é a parte do casco convexo visível a partir de cima do polígono e o caso inferior é a parte restante do casco convexo. O algoritmo roda a partir de seu ponto mais à direita até o ponto mais à esquerda no sentido anti-horário.

A Figura 15 ilustra o casco convexo superior e inferior.

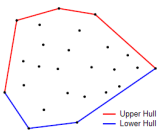


Figura 15: Ilustração do casco convexo superior e inferior

Solução Apresentada

De forma análoga ao algoritmo Graham Scan, utilizamos o algoritmo de ordenação *Quicksort* de forma não recursiva para fazer a ordenação pela coordenada x, caso haja empate utilizamos a menor coordenada y. Após a ordenação, constrói-se a parte inferior e superior do *Convex Hull* através do produto vetorial, de tal forma que se o resultado for maior ou igual a zero (curva no sentido horário ou colinear), os pontos são descartados pois não fazem parte do envoltório convexo. Após a construção das duas partes do envoltório convexo, elas são unidas em um conjunto final. Ressaltamos que 3 é o número mínimo de pontos para o *Hull*. A Figura 16 apresenta o pseudocódigo do algoritmo.

```
Monotone Chain

Entrada: um vetor V com os n pontos no plano.

Ordenar os pontos em ordem crescente pela coordenada x (em caso de
empate, classificar pela menor coordenada y)

Alocar vetor CascoConvexo de tamanho 2*n

k = 0;

// Cálculo do casco inferior
Para i = 1, 2, ... n
      Enquanto k >=2 && Produto Vetorial <=0
            k--;
      Fim Enquanto
      CascoConvexo[k++] = V[i];
Fim Para

// Cálculo do casco superior
t = k + 1;
Para i = n-1, n-2..., 1
      Enquanto k >=t && Produto Vetorial <=0
            k--;
      Fim Enquanto
      CascoConvexo[k++] = V[i];
Fim Para

Se (k>3)
      retorna CascoConvexo;
Fim Se
```

Figura 16: Pseudocódigo do Algoritmo Monotone Chain

6 Estratégia de Paralelização

Nos algoritmos Graham Scan e Monotone Chain a oportunidade de paralelização deve-se ao fato da necessidade dos pontos serem ordenados. A escolha do algoritmo de ordenação QuickSort é uma das grandes oportunidades de paralelização, pois se encaixa bem ao modelo de paralelismo de dados, ou seja, o encaixe entre a divisão em subproblemas e a divisão dos dados em *threads* (cada uma operando em uma parte dos dados). A determinação do *Convex Hull* nos dois algoritmos é realizada a partir dos resultados anteriores, logo não faz sentido paralelizar essa parte do código, sendo a ordenação o procedimento mais custoso em termos computacionais.

A escolha do algoritmo de ordenação Quicksort se deve ao fato de ser um algoritmo rápido e eficiente que em termos de complexidade é O(n log n) no melhor caso e caso médio e O(n^2) no pior caso (quando os valores do vetor está em ordem decrescente e é para ordenar em ordem crescente, por exemplo). Para tal, foi utilizado a versão não recursiva do QuickSort, pois eliminando a recursão o desempenho do algoritmo é melhorado e dessa forma os resultados da paralelização também.

A paralelização é realizada através da chamada do algoritmo de ordenação para cada partição do vetor e para cada *thread* definimos os limites da partição que a corresponde através dos índices do *loop*.

limite inferior = i * Pontos.Tam/NUMTHREADS;
limite superior = (i+1)*Pontos.Tam/NUMTHREADS - 1;

Cada *thread* executa o algoritmo segundo os limites definidos acima, ordenando parcialmente o vetor. Para terminar a ordenação chamamos o algoritmo Quicksort na forma sequencial. Utilizamos essa estratégia pois o custo de ordenar o vetor quase ordenado é bem menor do que utilizar o procedimento de "Merge", que realizaria a junção das partições de duas a duas. Tal procedimento teria o seu custo aumentado a medida que as partições são unidas e ordenadas, onde para cada "merge" percorremos todas as posições da maior partição. Dessa forma, concluímos que a utilização do Quicksort na forma sequencial para terminar a ordenação do vetor é mais eficiente.

Entre as desvantagens do paralelismo de dados podemos ressaltar o desbalanceamento de carga, devido a alguma parte dos subvetores possuir uma sequência pior para a ordenação do que a outra, o que no pior caso é quando o vetor está invertido. Além desse tipo de desbalanceamento, temos quando uma partição possui mais posições do que a outra. Porém, considerando o objetivo destes trabalho que é utilizar apenas os núcleos físicos do computador tal desbalanceamento será pequeno, considerando as arquiteturas atuais.

7 Análise de Complexidade

Nesta seção faremos uma análise da complexidade de tempo de cada função, em seguida calcularemos a complexidade total a partir da análise da função principal para as duas estratégias. Ressaltamos que n é o número de pontos lidos do arquivo de entrada, m é o número de pontos do *Convex Hull* e que as complexidades serão calculadas para o pior caso. Os cálculos e atribuições serão considerados com complexidade constante.

1. **void Leitura (FILE *entrada, vetor *Pontos, int *continua)**

   Essa função é responsável por realizar a leitura do arquivo. Nela temos a alocação dinâmica de um vetor, cujo tamanho é n. A medida que o arquivo é lido são feitas inserções dos valores no vetor. Dessa forma, a complexidade de tempo para essa função é O(n).

   Complexidade de tempo: O(n)

2. **void particionar(int esq, int dir,int *i, int *j, vetor *Pontos)**

   A função particionar possui dois *loops* cada um percorrendo metade do vetor e a partir de comparações realiza a divisão do vetor logo complexidade O(n).

   Complexidade de tempo: O(n)

3. **void QuickSort_NaoRec (vetor *Pontos, int n)**

   A função é responsável por ordenar o vetor quanto ao ângulo polar. Para isso, temos um *loop* percorrendo todo o vetor e a partir das operações de pilha (empilhar e desempilhar de custo constante O(1)) e a chamada à função Particionar realiza a ordenação. Dadas as complexidades das operações de pilha e da função particionar, temos que a complexidade é O(n^2), para o pior caso.

   Complexidade de tempo: O(n^2).

4. **int Graham (pilha *Pilha, vetor *Pontos, vetor *CascoConvexo)**

   A função utiliza dentro de um *loop* percorrendo o vetor apenas operações de pilha, cálculos e verificações, cujo custo é constante, portanto sua complexidade é O(n);

   Complexidade de tempo: O(n).

5. **int MonotoneChain (vetor Pontos, vetor *CascoConvexo)**

   A Função possui dois *loops* percorrendo o vetor de pontos, porém contém apenas cálculos, atribuições e comparações, por isso a complexidade da função é O(n).

   Complexidade de tempo: O(n).

6. **int main(int argc, char *argv[])**

   Além das funções descritas acima o algoritmo Graham Scan tem as chamadas às funções: menor_coordenada (determinação do ponto com menor coordenada x, denominado p_0), angulo_polar (calcula o ângulo polar entre p_0 e cada ponto), Criar (alocação da pilha), Destruir_pilha, Destruir (desalocação do vetor) e imprimir (imprime o resultado no arquivo de saída). As complexidades são respectivamente O(n), O(n), O(m), onde m é o número de pontos do *Convex Hull*, por fim O(1) e O(m).

   - Graham Scan:
     Complexidade: K_1O(n) + O(n^2) + O(n)(Graham Scan) + K_2O(m) + K_3O(1) = O(n^2), onde K_1, K_2 e K_3 são os números de chamadas das funções de mesma complexidade.
     Complexidade de tempo: O(n^2).

   O algoritmo Monotone Chain tem as chamadas às funções: Leitura, QuickSort_NaoRec, Criar, Destruir_pilha, Destruir e imprimir.

   - Monotone Chain:
     Complexidade: O(n^2) + O(n)(Monotone Chain) + K_2O(m) + K_3O(1) = O(n^2), onde K_1, K_2 e K_3 são os números de chamadas das funções de mesma complexidade.
     Complexidade de tempo: O(n^2).

8 Testes e resultados

Nesta seção avaliamos o desempenho dos algoritmos Monotone Chain e Graham Scan em termos do tempo de execução (em microssegundos) para diferentes valores de entrada. Os experimentos foram executados em um computador pessoal com sistema operacional Ubuntu 12.04, processador Intel Core 2 Duo 3.6Ghz e 4GB de memória principal.

Ressaltamos que o tempo de execução reflete diretamente no custo de processamento e execução das operações de ordenação dos pontos e do cálculo do *Convex Hull*. Durante os experimentos também variamos os valores de entrada e o numero de *threads*(de 1 a 4, ou seja, utilizando apenas núcleos físicos). Cada experimento foi executado 50 vezes e foram tomados os valores médios para análise.

8.1 Testes na versão sequencial

O algoritmo Graham Scan teve um pior desempenho(curva superior) pois necessita de fazer mais cálculos (verificar o ponto de menor coordenada e cálculo do ângulo polar) que o Monotone Chain(curva inferior), resultando assim em um tempo de execução maior.

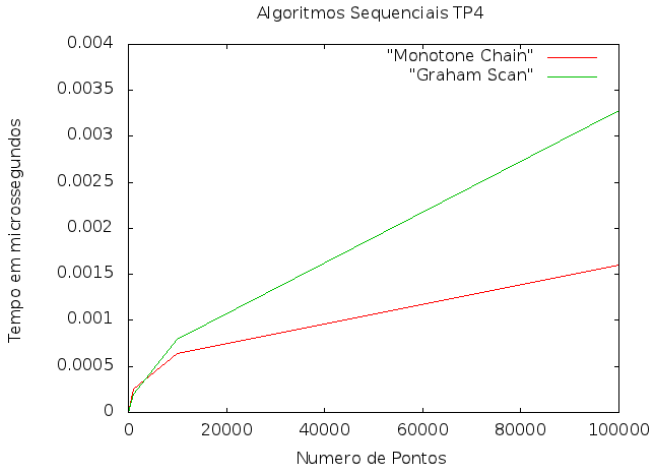


Figura 17: Testes na versão sequencial

8.2 Testes na versão paralelizada

A Figura 18, apresenta o resultado dos testes realizados.

Entrada (N)	Tempo Gasto (microssegundos)			
	Número de Threads			
	1	2	3	4
10	0.004	0.0037	0.0029	0.0030
1000	0.135	0.128	0.109	0.115
1000000	10.653	9.225	8.431	9.225
100000000	101.024	95.331	92.055	103.077

(a) Graham Scan

Entrada (N)	Tempo Gasto (microssegundos)			
	Número de Threads			
	1	2	3	4
10	0.005	0.0032	0.0030	0.0029
1000	0.129	0.130	0.100	0.112
1000000	10.221	9.116	8.435	9.221
100000000	103.017	95.243	92.043	103.056

(b) Monotone Chain

Figura 18: Tabelas de Resultados dos testes de tempo

A partir dos dados observamos que em termos de tempo gasto dos algoritmos são desprezíveis e a diferença entre eles é insignificante, não justificando uma comparação justa entre o tempo de execução, que pode ter sido afetado pelo escalonamento do sistema operacional. Foi observado também que a partir de 4 *threads* o desempenho do algoritmo paralelizado não cresce mais. Podemos também observar que houve um speed-up, ou seja, aumento de desempenho quando passamos de 1 Thread para 2 Threads, na ordem de 25% a menos no tempo de execução, e isso acontece pelo fato do computador utilizado ser Core 2 Duo e ter conseguido utilizar ambos os núcleos da máquina. O speed-up não foi da ordem de 100%, pois como em programas totalmente paralelizáveis isso deve-se ao fato das frações seriais dos algoritmos reduzindo o ganho de desempenho do programa.

9 Conclusão

As características presentes nos dois algoritmos implementados e paralelizados foram a possibilidade de dividir os dados para a ordenação, permitindo que as tarfeas pudessem ser executadas em cada parte, o que permitiu que os resultados fossem positivos. Nos demais algoritmos analisados a ausência de independência entre as tarefas assim como a presença de recursão, tornando difícil o controle, não garantiriam bons resultados ou impossibilitou a paralelização.

Com o desenvolvimento deste trabalho tivemos a oportunidade de obter mais conhecimentos sobre a programação paralela, além do aprendizado sobre a identificação de vantagens, desvantagens e possibilidades de implementação de estratégias de paralelização através da análise de suas características a procura de oportunidades. Nos proporcionou também a aquisição de experiência na análise experimental de algoritmos assim como em desenvolvimento e projeto de soluções para problemas computacionais.

Conseguimos atingir o objetivo deste trabalho que era identificar oportunidades de paralelização, implementar duas estratégias de paralelização e a realizar uma análise experimental comparativa para a obtenção dos resultados.

10 Referências Bibliográficas

[1] Kernighan, Brian W., Ritchie, Dennis M. C - A linguagem de programação padrão ANSI C, 1989.

[2] Ziviani, Nívio - Computação Paralela.

[3] O'Rourke, Joseph - Computational Geometry in C, Second Edition.

[4] Fernandes, Cristina G., Pina, José Coelho - Geometria Computacional, 24 de outubro de 2011.

[5] Bajuelos, Antônio Leslie - Invólucros Convexos no Plano, Universidade de Aveiro - Departamento de Matemática.

[6] Figueiredo, Celina M. H. de, Fonseca, Guilherme D. da - Apostila Introdutória de Algoritmos - 2003.

[7] Pio, José Luiz de Souza - Geometria Computacional, Universidade Federal do Amazonas - Instituto de Computação.

[8] Fernandes, Cristina G. - Geometria Computacional, Departamento de Ciência da Computação do IME-USP - 2009.

[9] Toffolo, Túlio, Carvalho, Marco Antônio - Geometria Computacional - Departamento de Computação - UFOP.

[10] Mourão, Fernando - Introdução à Pthreads - Departamento de Ciência da Computação - UFMG - 2008.

[11] Oliveira, Rafael Sachetto - Computação Paralela - Departamento de Ciência da Computação - UFSJ - 2012.