MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

Análise Comparativa de Algoritmos para Computação de Pontos de Intersecção entre Conjuntos de Segmentos de Reta em Máquinas Multi-Core

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA
(PIBIC/CNPq/INPE)

João Vitor Chagas (FATEC, Bolsista PIBIC/CNPq)
E-mail: joao.vitor.inpe@gmail.com
Dr. Gilberto Ribeiro de Queiroz (OBT/DPI/INPE, Orientador)
E-mail: gribeiro@dpi.inpe.br

COLABORADORES

Dr. Reinaldo Gen Ichiro Arakaki (FATEC/SJC)

Junho de 2016

Esta ficha será revisada pelo SID.

Dados Internacionais de Catalogação na Publicação

Cutter

Sobrenome, Prenome(s) Completo(s) do(s) Autor(es).
Título da publicação / Nome(s) Completo(s) do(s) Autor(es). - São José dos Campos: INPE, ano da publicação.

Grau (Mestrado ou Doutorado em Nome do Curso) - Instituto Nacional de Pesquisas Espaciais, São José dos Campos, ano de defesa.
Orientador: Nome completo do orientador(es).

1. Assunto. 2. Assunto. 3. Assunto. 4. Assunto. 5. Assunto.
I. Título

CDU

FOLHA DE APROVAÇÃO

CONFECCIONADA PELO SPG E INCLUÍDA PELO SID

RESUMO

A computação dos pontos de intersecção entre conjuntos de segmentos de reta é considerado um dos problemas mais relevantes para um Sistema de Informação Geográfica (SIG), sendo a base para a construção de diversas operações encontradas neste tipo de sistema. A computação de tais pontos envolve um grande consumo de processamento, principalmente, para grandes entradas de dados. Tanto na literatura de Geometria Computacional quanto na de Geoinformática, encontramos diversos algoritmos para solução deste problema. No entanto, esses algoritmos possuem diferentes compromissos de desempenho versus complexidade de implementação, propiciando um substancial desafio para desenvolvedores e projetistas de SIGs, no que diz respeito à escolha, refinamento e implementação desses algoritmos. Além disso, grande parte dos algoritmos foram desenvolvidos em uma época em que não existia as atuais arquiteturas de processadores multi-core e, consequentemente, foram projetados de forma sequencial ou de difícil paralelização. Neste trabalho, examinamos um conjunto de algoritmos de intersecção entre conjuntos de segmentos de reta – *força-bruta, x-ordering, fixed-grid e tiling-scheme*, e como adaptá-los para ambientes paralelos, utilizando o modelo de programação *multithread*. Nossas análises foram realizadas com base em testes empíricos realizados com a implementação em C++ de versões sequenciais dos algoritmos e posterior paralelização, utilizando dados geográficos reais acessados através da biblioteca TerraLib. Os resultados obtidos mostram que os algoritmos sequenciais são bem competitivos quando comparados com a solução trivial do problema. Além disso, mostram um ganho significativo em se paralelizar partes das instruções desses algoritmos.

Palavras-chave: SIG. Algoritmo Intersecção.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE SIGLAS E ABREVIATURAS

SIG Sistema de Informação Geográfica

SUMÁRIO

Pág.

1. INTRODUÇÃO

Os Sistemas de Informação Geográficas (SIGs) são sistemas utilizados para representação computacional do espaço geográfico, possuindo aplicações em várias áreas da sociedade (Camara et. al, 1996). Neste tipo de sistema, os dados geográficos, compostos pelos componentes espacial, temporal e alfanumérico, podem ser representados por matrizes ou vetores.

No caso da representação vetorial, são utilizadas formas geométricas como pontos, linhas e polígonos, para representação da geometria dos objetos espaciais. Um exemplo do uso deste tipo de representação ocorre em mapas municipais, onde cada um dos municípios é representado por um ou mais polígonos e atributos como população, renda per capita, número de escolas, entre outras.

Existem dois modelos clássicos de representação vetorial:

- **feições simples:** cada objeto geográfico é representado por pontos, linhas ou polígono que possuem seu próprio conjunto de coordenadas.
- **modelo topológico:** também conhecido por modelo arco-nó, os objetos geográficos podem compartilhar arestas e nós.

Neste trabalho estamos interessados na representação vetorial, mais especificamente de objetos geométricos com extensão, que é o caso de linhas e polígonos. Para objetos que utilizam esse tipo de representação, as operações mais empregadas para álgebra de mapas se baseiam na sobreposição de mapas (overlay). A Figura 1 mostra um exemplo de sobreposição de dois mapas: (1) aptidão agrícola do Estado de Minas Gerais e, (2) limites municipais de Minas Gerais. A sobreposição de mapas, neste caso, permite avaliar para cada município, as aptidões relacionadas.

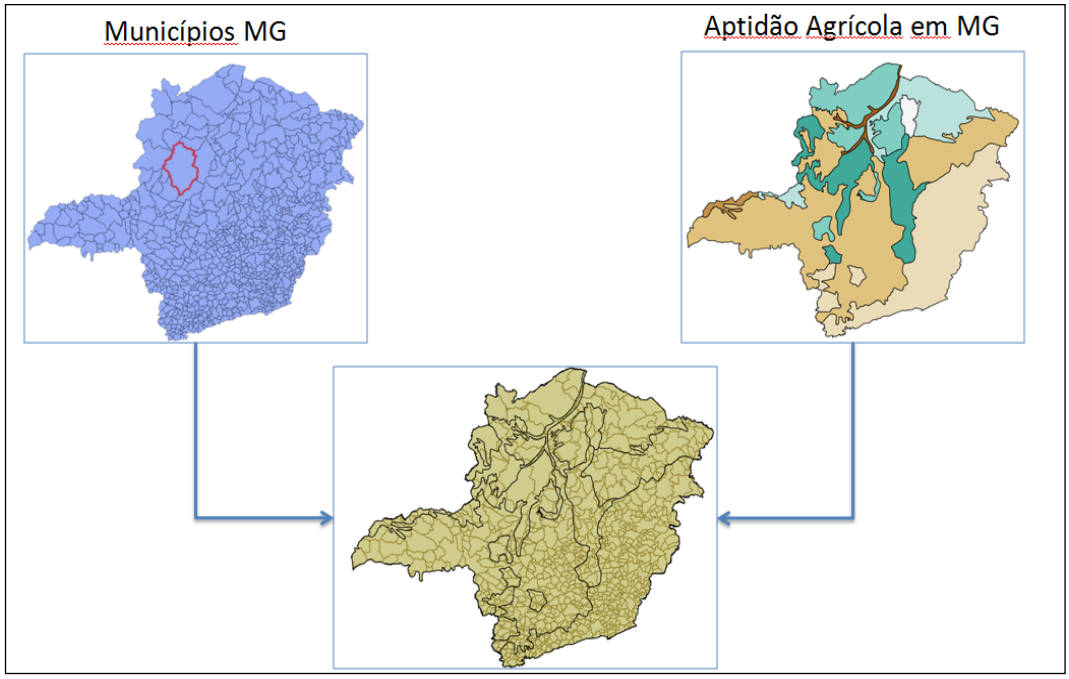

Figura1.1– Sobreposição de mapas (overlay)

Para operações como a sobreposição de mapas vetoriais, é necessário computar os pontos de intersecção entre os segmentos que representam as feições geográficas. A computação dos pontos de intersecção entre conjuntos de segmentos de reta é considerado um dos problemas mais relevantes em SIG, pois é uma das operações básicas para a construção não só do overlay mas também de diversas outras operações encontradas neste tipo de sistema.

A computação de pontos de intersecção envolve um grande consumo de processamento, principalmente, para grandes entradas de dados, isto é, para grande números de segmentos de reta. Tanto na literatura de Geometria Computacional (Preparata e Shamos, 1985) quanto na de Geoinformática (Longley et al., 2005), encontramos diversos algoritmos para solução deste problema(Bentley e Ottmann, 1979; Franklin et al., 1988; Pullar, 1990; Chazelle e Edelsbrunner, 1992; Chan, 1994; Andrews et al., 1994). No entanto, esses

algoritmos possuem diferentes compromissos de desempenho versus complexidade de implementação, propiciando um substancial desafio para desenvolvedores e projetistas de SIGs, no que diz respeito à escolha, refinamento e implementação desses algoritmos. Outro fator interessante é que grande parte desses algoritmos foram desenvolvidos em uma época em que não existia as atuais arquiteturas de processadores multi-core e, consequentemente, foram projetados de forma sequencial ou de difícil paralelização.

1.1. Objetivo

Neste trabalho, examinamos um conjunto de algoritmos de intersecção entre conjuntos de segmentos de reta – *força-bruta, x-ordering, fixed-grid e tiling-scheme*, e como adaptá-los para ambientes paralelos, utilizando um modelo de programação *multithread*. O principal objetivo deste trabalho será produzir um conjunto de operadores geométricos capazes de calcular de forma eficiente os pontos de intersecção entre conjuntos de segmentos de reta. Espera-se que estes operadores sejam capazes de responder à demanda cada vez maior de análise e processamento de grandes volumes de dados geográficos, um tema muito relevante dentro de projetos de pesquisa e desenvolvimento do INPE.

1.2. Resumo Metodológico

Para a realização do trabalho, foram estabelecidas as seguintes etapas:

- Realizar um estudo dirigido sobre os conceitos básicos de Geometria Computacional;
- Desenvolver em C++ a versão sequencial dos seguintes algoritmos: *força-bruta, x-ordering, fixed-grid e tiling-scheme*;
- Desenvolver em C++ versões utilizando programação *multithread* dos seguintes algoritmos: *força-bruta, x-ordering, fixed-grid e tiling-scheme*;

- Projetar e implementar uma bateria de testes empíricos utilizando dados geográficos reais acessados através da biblioteca TerraLib;
- Analisar os resultados obtidos com os testes.

2. REVISÃO DA LITERATURA

Dados dois segmentos de reta no plano bidimensional, determinar se eles possuem ou não uma intersecção, e computar este ponto, é uma das operações primitivas utilizadas pelos algoritmos que discutiremos neste capítulo. Por se tratar de uma operação que aparece nos laços internos dos demais algoritmos, com a possibilidade de ser chamada milhares de vezes, é preciso que sua implementação seja eficiente. Na literatura, encontramos diversos métodos para realizar esta computação (Prasad, 1991; Shaffer e Feustel, 1992; Antonio, 1992). Para entender melhor os métodos existentes, primeiramente, vamos fazer uma revisão sobre retas e segmentos de reta. Após esta discussão iremos apresentar os algoritmos de intersecção de conjuntos de segmentos de reta que foram estudados e que representam o foco central do trabalho.

2.1. Retas e Segmentos de Reta

Uma reta pode ser definida por uma equação linear da seguinte forma:

$$y = mx + b \qquad \text{(equação 1)}$$

onde: *m* e *b* são constantes.

A *equação 1* é denominada de equação reduzida da reta(ou *slope-intercept*). O termo “linear” deve-se ao fato de que o conjunto solução forma uma linha reta (ou *straightline*), onde a constante m define a inclinação desta reta (*slope*) e a constante b representa o valor onde a linha cruza o eixo-y.

Dado os pontos $P_1(x_1, y_1)$ e $P_2(x_2, y_2)$, a inclinação da reta que passa por esses ponto sé dada por:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$ (equação 2)

para $x_1 \neq x_2$[1].

Outra maneira usual de representarmos uma reta é através da equação geral(*general form*), definida como:

$$ax + by + c = 0$$ (equação 3)

onde: *a, b* e *c* são constantes.

Utilizando a *equação 3*, uma reta contendo os pontos $P_1(x_1, y_1)$ e $P_2(x_2, y_2)$, terá sua equação escrita da seguinte forma:

$$(y_1 - y_2)x + (x_2 - x_1)y + (x_1y_2 - x_2y_1) = 0$$ (equação 4)

Outra forma de representarmos os pontos de uma reta é através da seguinte equação paramétrica (ou *parametric-form*):

$$P = P_1 + \alpha(P_2 - P_1)$$ (equação 5)

Na *equação 5* acima, se ***α*** variar no intervalo [0, 1], P corresponderá aos pontos que formam o segmento de reta $\overline{P_1P_2}$, ou seja, valores de *x* e *y* compreendidos entre os ponto $P_1(x_1, y_1)$ e $P_2(x_2, y_2)$. Esta equação pode ser usada na forma abaixo:

$$x = x_1 + \alpha(x_2 - x_1)$$
$$y = y_1 + \alpha(y_2 - y_1)$$ (equação 6)

Por último, temos a forma de representação vetorial utilizando determinantes (2D *vector determinant-form*):

---

[1] Esta distinção se deve ao fato de que nesta equação retas verticais possuem um resultado indeterminado.

$$\det(\overrightarrow{P_1P}, \overrightarrow{P_1P_2}) = 0 \qquad \text{(equação 7)}$$

Desenvolvendo o determinante mostrado na *equação 7*, obtemos a *equação 8*:

$$\begin{vmatrix} x - x_1 & y - y_1 \\ x_2 - x_1 & y_2 - y_1 \end{vmatrix} = 0$$

$$(x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1) = 0 \qquad \text{(equação 8)}$$

2.2. Intersecção entre Dois Segmentos de Reta

A coleção de livros Graphics Gems apresenta várias maneiras de como computar os pontos de intersecção entre dois segmentos de reta (Prasad, 1991; Shaffer e Feustel, 1992; Antonio, 1992). Neste trabalho, optamos por implementar o algoritmo proposto por Antonio (1992), que utiliza a representação paramétrica da reta (*equação 5*).

Seu método, baseia-se na ideia de que dado os segmentos de reta *S* e *T*, com $P_1(x_1, y_1)$ e $P_2(x_2, y_2) \in S$ e $P_3(x_3, y_3)$ e $P_4(x_4, y_4) \in T$, podemos construir as equações paramétricas dos dois segmentos da seguinte forma:

$$\begin{cases} P_S = P_1 + \alpha(P_2 - P_1) \\ P_T = P_3 + \beta(P_4 - P_3) \end{cases} \qquad \text{(equação 9)}$$

Onde *α* e *β* devem estar no intervalo [0, 1]. Logo, a intersecção dos dois segmentos ocorrerá quando $P_S = P_T$, ou seja, podemos desenvolver as equações acima, obtendo a *equação 10:*

$$(P_1 - P_3) + \alpha(P_2 - P_1) + \beta(P_3 - P_4) = 0 \qquad \text{(equação 10)}$$

A partir da *equação 10*, obtemos equações para computação de *α* e *β*:

$$\mathrm{A} = \mathrm{P}_2 - \mathrm{P}_1 \qquad \text{(equação 11)}$$

$$\mathrm{B} = \mathrm{P}_3 - \mathrm{P}_4 \qquad \text{(equação 12)}$$

$$C = \mathrm{P}_1 - \mathrm{P}_3 \qquad \text{(equação 13)}$$

$$\alpha = \frac{\mathrm{B}_y \times C_x - \mathrm{B}_x \times C_y}{\mathrm{A}_y \times \mathrm{B}_x - \mathrm{A}_x \times \mathrm{B}_y} \qquad \text{(equação 14)}$$

$$\beta = \frac{\mathrm{A}_x \times C_y - \mathrm{A}_y \times C_x}{\mathrm{A}_y \times \mathrm{B}_x - \mathrm{A}_x \times \mathrm{B}_y} \qquad \text{(equação 15)}$$

Pode-se reparar nas *equações 14* e *15* que os denominadores dos dois coeficiente, *α* e *β*, são iguais e, por isso, só precisam ser computados uma única vez. Para verificar se os segmentos possuem ou não intersecção, a partir dessas duas equações, é necessário apenas verificar os sinais dos numeradores e denominadores das equações , como mostrado no algoritmo da Figura 2.1.

```
01  se (denominador > 0)
02      então
03          se (numerador < 0) ou (numerador > denominador)
04              então
05                  retorna segmentos não se interceptam
06          fim se
07      senão
08          se (numerador > 0) ou numerador < denominador
09              então
10                  retorna segmentos não se interseptam
11          fim se
12  fim se
```

Figura 2.1 – Verificando se dois segmentos possuem ou não intersecção.

No algoritmo da Figura 2.1, o teste da linha 03 verifica se o numerador é menor do que zero. Se esse for o caso, significa que a computação do coeficiente (*α* ou *β*) irá gerar um número negativo, ou seja, fora do intervalo [0, 1], o que significa que não haverá intersecção entre os segmentos. A outra expressão do teste na linha 03, somente será avaliada caso o numerador seja um número positivo. Neste caso, se o numerador for maior que o denominador,

o resultado da computação do coeficiente (*α* ou *β*) irá gerar um número maior do que um, ou seja, fora do intervalo [0, 1], não havendo assim intersecção entre os segmentos.

Caso o denominador do coeficiente sendo computado seja negativo, a cláusula “senão” na linha 07 irá realizar testes semelhantes para determinar se o valor do coeficiente encontra-se no intervalo [0, 1].

Na Figura 2.2 apresentamos a parte central do código em Linguagem C++ desenvolvido no escopo do projeto.

```
01 segment_relation_type
02 compute_intesection(const     line_segment&     s1,     const
03 line_segment& s2,
04                     point& first, point& second)
05 {
06   double ax = s1.p2.x - s1.p1.x;
07   double ay = s1.p2.y - s1.p1.y;
08
09   double bx = s2.p1.x - s2.p2.x;
10   double by = s2.p1.y - s2.p2.y;
11
12   double den = ay * bx - ax * by;
13
14   if(den == 0.0) // are they collinear?
15   {
16     if(do_collinear_segments_intersects(s1, s2) == false)
17       return DISJOINT;
18
19 // and we know they intersects: let's order the segments
20 // and find out intersection(s)
21     const point* pts[4];
22     pts[0] = &s1.p1;
23     pts[1] = &s1.p2;
24     pts[2] = &s2.p1;
25     pts[3] = &s2.p2;
26
27     sort(pts, pts + 4, point_cmp);
28
29 // at least they will share one point
30     first = *pts[1];
31
32 // and if segments touch in a single point they are equal
33     if((pts[1]->x == pts[2]->x) && (pts[1]->y == pts[2]->y))
34       return TOUCH;
35
36 // otherwise, the middle points are the intersections
37     second = *pts[2];
38
```

```
39      return OVERLAP;
40    }
41
42  // they are not collinear, let's see if they intersects
43    double cx = s1.p1.x - s2.p1.x;
44    double cy = s1.p1.y - s2.p1.y;
45
46  // is alpha in the range [0..1]
47    double num_alpha = by * cx - bx * cy;
48
49    if(den > 0.0)
50    {
51  // is alpha before the range [0..1] or after it?
52      if((num_alpha < 0.0) || (num_alpha > den))
53        return DISJOINT;
54    }
55    else // den < 0
56    {
57  // is alpha before the range [0..1] or after it?
58      if((num_alpha > 0.0) || (num_alpha < den))
59        return DISJOINT;
60    }
61
62  // is beta in the range [0..1]
63    double num_beta = ax * cy - ay * cx;
64
65    if(den > 0.0)
66    {
67  // is beta before the range [0..1] or after it?
68      if((num_beta < 0.0) || (num_beta > den))
69        return DISJOINT;
70    }
71    else // den < 0
72    {
73  // is beta before the range [0..1] or after it?
74      if((num_beta > 0.0) || (num_beta < den))
75        return DISJOINT;
76    }
77
78  // compute intersection point
79    double alpha = num_alpha / den;
80
81    first.x = s1.p1.x + alpha * (s1.p2.x - s1.p1.x);
82    first.y = s1.p1.y + alpha * (s1.p2.y - s1.p1.y);
83
84    return CROSS;
85  }
```

Figura 2.2 – Função para computação do ponto de intersecção entre dois segmentos de reta.

Embora seja muito relevante o tratamento dos erros de arredondamento na computação dos pontos intersecção entre segmentos de reta, neste trabalho não nos preocuparemos com questões de robustez desta operação. Esta

questão será investigada em outro projeto de pesquisa. Portanto, aqui nos concentraremos nas questões de desempenho.

2.3. Algoritmos de Intersecção de Conjuntos de Segmentos

Dado um conjunto de n segmentos de reta no plano, se cada um dos n segmentos interceptar todos os demais, teremos pontos de intersecção. Isto significa, que no pior caso, qualquer algoritmo desenvolvido deverá realizar $O(n^2)$ operações.

Considerando o pior cenário, com $O(n^2)$ interseções, um algoritmo trivial ou de força bruta, que avalie a combinação de todos os pares de segmentos, irá realizar este trabalho em tempo ótimo, realizando $O(n^2)$ operações de intersecção entre pares de segmentos.

No entanto, vários dos conjuntos de dados que utilizamos na prática, principalmente em SIGs, possuem um número muito menor de interseções. Neste caso, um algoritmo força bruta, de complexidade $O(n^2)$, irá realizar uma grande quantidade desnecessária de operações de intersecção entre pares de segmentos que não irão se interceptar.

Shamos e Hoey (1976) foram os pioneiros no estudo desse tipo de problema geométrico: “dado um conjunto de N segmentos no plano, determinar todas as interseções entre pares de segmentos desse conjunto”. Tal problema possui ampla aplicação, sendo que nos SIGs diversas operações dependem da solução deste problema: (a) computar a união, intersecção ou diferença entre polígonos; (b) avaliar se dois polígonos ou duas linhas poligonais se interceptam; (c) testar se uma linha poligonal é simples ou não, isto é, se ela não possui auto-interseções; (d) realizar a decomposição de polígonos em partes mais simples (triângulos ou trapézios).

Os algoritmos apresentados por Shamos e Hoey (1976) apenas detectam a existência ou não de alguma intersecção entre algum par de segmentos do conjunto. Posteriormente, Bentley e Ottman (1979) refinaram

esse algoritmo para reportar todos os pontos de intersecção, introduzindo uma técnica conhecida por *Plano de Varredura* (*Plane Sweep*) ou Linha de Varredura(*LineSweep*), que por sua vez deu origem a inúmeros algoritmos criados pelos pesquisadores de Geometria Computacional (Preparata e Shamos, 1985; Chazelle e Edelsbrunner, 1992; Chan, 1994; Andrews et al., 1994).

No entanto, esses algoritmos de Geometria Computacional são de difícil implementação, necessitando de estruturas de dados mais complexas e interpretações que demandam maior conhecimento matemático. Por conta disso, neste trabalho nos baseamos em algoritmos com técnicas consagradas por pesquisadores de SIGs, denominados por David Pullar de algoritmos pragmáticos (Pullar, 1990).

2.3.1. Força-Bruta

O ***algoritmo força-bruta*** (ou ingênuo) avalia a possibilidade de intersecção entre todos os pares de segmentos. Isso significa que ele realiza $O(n^2)$ operações, como pode ser observado pelo algoritmo mostrado na Figura 2.3.

```
01  algoritmo LazyIntersection(S)
02      n ← tamanho(S)
03      para i ← 1 até n
04          para j ← i + 1 até n - 1
05              se S[i] ∩ S[j]
06                  então reportar(S[i] ∩ S[j])
07              fim se
08          fim para
09      fim para
10  fim algoritmo
```

Figura 2.3 - Algoritmo de força-bruta para computação dos pontos de intersecção entre pares de segmentos de um conjunto

O algoritmo mostrado na Figura 2.3, recebe como entrada um conjunto de segmentos de reta *S*, de tamanho *n*. Os dois laços “for” garantem que o teste de intersecção da linha 05 é realizado para todos os pares de segmentos do conjunto *S*. Pode-se demonstrar que este algoritmo é $\Theta(n^2)$. Logo, quando submetido a uma grande massa de dados, sua utilização se torna impraticável.

2.3.2. x-Ordering

O ***algoritmo x-Ordering*** (Pullar, 1990) utiliza parte da técnica de varredura. Inicialmente, os segmentos do conjunto de entrada (S) são ordenados pelo menor valor de suas abscissas (x), de forma a propiciar que os segmentos possam ser percorridos da esquerda para a direita. Cada segmento acessado nessa ordem só poderá interceptar segmentos que possuam o menor valor de abscissa no seu intervalo, ou seja, para cada segmento acessado em ordem, temos a definição de uma faixa vertical na qual podemos descartar os segmentos que estejam fora dela. A Figura 2.4 mostra a ideia central desse algoritmo.

```
01  algoritmo xOrderingIntersection(S)
02    ordena_menor_abscissa(S)
03    n ← tamanho(S)
04    para i ← 1 até n - 1
05      scorrente ← S[i]
06      para j ← i + 1 até n
07        spróximo ← S[j]
08        se maior_abscissa(scorrente) < menor_abscissa(spróximo)
09          então continue
10        fim se
11        se scorrente ∩ spróximo
12          então reportar(scorrente ∩ spróximo)
13        fim se
14      fim para
15    fim para
16  fim algoritmo
```

Figura 2.4 - Algoritmo x-Ordering para computação dos pontos de intersecção entre pares de segmentos de um conjunto.

O algoritmo apresentado na Figura 2.4 possui complexidade de pior caso $O(n^2)$. No entanto, conforme mostrado por Pullar (1990), o tempo esperado na prática depende do número de segmentos em cada faixa vertical definida por cada segmento do conjunto. Se, na média cada faixa vertical contiver m segmentos, podemos escrever a complexidade deste algoritmo como: $O(n \times m^2)$.

2.3.3. Fixed-Grid

O ***algoritmo Fixed-Grid*** (Franklin et al., 1988) associa os segmentos de reta do conjunto de entrada à células de uma grade, de forma que apenas os

segmentos associados a uma mesma célula podem apresentar intersecção. A eficiência desse método depende de como é estabelecida a resolução das células dessa grade. Se a resolução da grade for muito grosseira, isto é, com células muito grandes, diversos segmentos serão associados à mesma célula e logo muita computação desnecessária poderá ser realizada. Por outro lado, se a resolução das células for muito fina, isto é, com células muito pequenas, poderemos ter os mesmo segmentos associados a muitas células, e, neste caso, os mesmos pares de segmentos serão avaliados diversas vezes. Os trabalhos de Franklin et al. (1988) e de Pullar (1990) sugerem a utilização da média do tamanho dos segmentos ao longo dos eixos *x* e *y*. Nossos testes, discutidos nos Capítulos 3 e 4, também confirmaram esta suposição, com resultados bem competitivos. A Figura 2.5 mostra esse algoritmo.

```
01  algoritmo FixedGridIntersection(S)
02      G ← computa_grade(S)
03      n ← tamanho(S)
04      para i ← 1 até n
05          associa_segmento_células(S[i], G)
06      fim para
07      para cada célula c de G que contiver segmentos
08          Stemp ← segmentos(c)
09          LazyIntersection(Stemp)
10      fim para
11  fim algoritmo
```

Figura 2.5 - Algoritmo Fixed-Grid para computação dos pontos de intersecção entre pares de segmentos de um conjunto

No algoritmo da Figura 2.5, temos alguns passos importantes, que são mostrados, respectivamente nos algoritmos das Figuras 2.6 e 2.7.

```
01 estrutura Retângulo
02 {
03     xmin numérico,
04     xmax numérico,
05     ymin numérico,
06     ymax numérico,
07 }
08 estrutura Grade
09 {
10     extensão Retângulo,
11     resolução_x numérico,
12     resolução_y numérico,
13     num_linhas numérico,
14     num_colunas numérico,
15     células[1:num_linhas][1:num_colunas]
16 }
17 algoritmo computa_grade(S)
18     extensão ← computa_extensão(S)
19          resolução_x ← tamanho_médio_segmentos_em_x(S)
20     resolução_y ← tamanho_médio_segmentos_em_y(S)
21     G ← cria_grade(extensão, resolução_x, resolução_y)
22     retorna G
23 fim algoritmo
```

Figura 2.6 – Computando a grade a ser usada no algoritmo fixed-grid Intersection.

```
01 algoritmo associa_segmento_células(s, G)
02     xmin ← menor_abscissa(s)
03     xmax ← maior_abscissa(s)
04     ymin ← menor_ordenada(s)
05     ymax ← maior_ordenada(s)
06     imin ← floor( (xmin – G.extensão.xmin) / G.resolução_x )
07     imax ← ceil( (xmax – G.extensão.xmin) / G.resolução_x )
08     jmin ← trunca( (ymin – G.extensão.ymin) / G.resolução_y)
09     jmax ← trunca( (ymax – G.extensão.ymin) / G.resolução_y)
10     para i ← imin até imax
11         para j ← jmin até jmax
12             G[i][j] ← s
13         fim para
14     fim para
15 fim algoritmo
```

Figura 2.7 – Associando um segmento às células da grade que ele intercepta.

A análise de tempo do algoritmo de intersecção por grades fixas depende do número de células definidas pela grade e pelo número de segmentos em cada célula. No pior caso, para uma grade com *c* células onde

todos os segmentos encontram-se associados a todas as células, teremos um algoritmo O(cn^2), ou seja, para grades com grande número de células este algoritmo teria o pior desempenho de todos. No entanto, com a definição de uma boa resolução para a grade o tempo esperado é bem menor: O(cm^2), onde *c* é o número de células com segmentos associados e *m* é o número médio de segmentos associados em cada célula.

2.3.4. Tiling-Scheme

O ***algoritmo tiling-scheme*** (Pullar, 1990) é muito semelhante ao do fixed-grid. Os segmentos de reta do conjunto de entrada são associados a faixas horizontais (*tiles horizontais*), de maneira que para cada faixa horizontal, é utilizado o *algoritmo x-Ordering* para computar os pontos de intersecção entre os segmentos dessa faixa. Esse algoritmo contorna parte dos problemas que podem comprometer a eficiência do *algoritmo x-Ordering* aplicado diretamente a todo o conjunto.

```
01  algoritmo TilingIntersection(S)
02      T ← computa_tiling(S)
03      n ← tamanho(S)
04      para i ← 1 até n
05          associa_segmento_tilings(S[i], T)
06      fim para
07      para cada t de T
08          Stemp ← segmentos(t)
09          xOrderingIntersection(Stemp)
10      fim para
11  fim algoritmo
```

Figura 2.8 - Algoritmo tiling-schema para computação dos pontos de intersecção entre pares de segmentos de um conjunto.

As Figuras 2.9 e 2.10 contém mais detalhes do algoritmo apresentado na Figura 2.8.

```
01  estrutura Tiling
02  {
03      ymin numérico,
04      ymax numérico,
05      resolução_y numérico,
06      num_linhas numérico,
07      tiles[1:num_linhas]
```

```
08 }
09 algoritmo computa_tiling(S)
10     ymin ← obtém_ymin(S)
11     ymax ← obtém_ymax(S)
12     resolução_y ← tamanho_médio_segmentos_em_y(S)
13     T ← cria_tiling(ymin,ymax, resolução_y)
14     retorna T
15 fim algoritmo
```

Figura 2.9 – Computando os tiles usados no algoritmo Tilins-Grid

```
01 algoritmo associa_segmento_células(s, G)
02     xmin ← menor_abscissa(s)
03     xmax ← maior_abscissa(s)
04     ymin ← menor_ordenada(s)
05     ymax ← maior_ordenada(s)
06     imin ← floor( (xmin – G.extensão.xmin) / G.resolução_x )
07     imax ← ceil( (xmax – G.extensão.xmin) / G.resolução_x )
08     jmin ← trunca( (ymin – G.extensão.ymin) / G.resolução_y)
09     jmax ← trunca( (ymax – G.extensão.ymin) / G.resolução_y)
10     para i ← imin até imax
11         para j ← jmin até jmax
12             G[i][j] ← s
13         fim para
14     fim para
15 fim algoritmo
```

Figura 2.10 – Associando um segmento aos tiles que ele intercepta.

A análise de tempo do algoritmo de intersecção por tiling depende do número de tiles definidos e pelo número de segmentos em cada *tile*. No pior caso, para *t tiles* onde todos os segmentos encontram-se associados a todos os tiles, teremos um algoritmo $O(tn^2)$, ou seja, este algoritmo possui um pior caso com desempenho abaixo do algoritmo de força-bruta. No entanto, com a definição de uma boa resolução para os tiles, o tempo esperado é bem menor: $O(tm^2)$, onde *t* é o número de *tiles* com segmentos associados e *m* é o número médio de segmentos associados a cada *tile*.

3. METODOLOGIA DE DESENVOLVIMENTO

O principal objetivo deste trabalho é produzir um conjunto de operadores geométricos capazes de calcular de forma eficiente os pontos de intersecção entre conjuntos de segmentos de reta. Para isso, iremos adaptar os algoritmos apresentados no Capítulo 2 para explorar o paralelismo do hardware de máquinas *multi-core* com o uso de *threads*.

Os seguintes algoritmos foram implementados de forma sequencial e então paralelizados, inteira ou parcialmente: *força-bruta*, *x-ordering, fixed-grid e tiling-scheme*.

3.1. Ambiente de Desenvolvimento

O ambiente de trabalho é composto dos seguintes sistemas e hardware:

- **Microcomputador:** Intel Core i7, 16 GiB RAM, HD SATA 7.200 rpm e 2 TiB.
- **Sistema Operacional**: Linux Ubuntu 14.04 LTS.
- **Linguagem de Programação:** Linguagem C++ através do compilador GNU g++ versão 4.8.4.
- **IDE:** Qt Creator.
- **Projeto de Build:** para criação dos projetos de build do código desenvolvido, foi utilizada a ferramenta CMake. Utilizamos a versão 2.8.12 e a aplicação gráfica CMake GUI.
- **Sistema de Controle de Versão de Código Fonte:** para o controle de versionamento do código fonte do projeto foi utilizado o Git, através de repositório criados no GitHub. O repositório principal encontra-se no seguinte endereço: https://github.com/gqueiroz/gde. O repositório de

trabalho (fork) encontra-se no seguinte endereço: https://github.com/JoaoVitor123/gde.

- **Ferramenta SIG:** para acesso e manipulação de dados geográficos reais foi empregada a biblioteca TerraLib, na versão 5.1.2.

3.2. Algoritmos Desenvolvidos

Os algoritmos apresentados no Capítulo 2 tomam como entrada um único conjunto de segmentos de reta e avalia todas as interseções entre os segmentos deste conjunto. Existe também uma variação do problema de intersecção de conjuntos de segmentos conhecida por Intersecção *Vermelho-Azul* (ou *red-blue intersection*), na qual somente as intersecções entre segmentos pertencentes a dois conjuntos distintos são reportados. Neste caso, temos um conjunto de segmentos dito vermelhos e o outro conjunto de segmentos azuis, e reportamos interseções apenas entre pares vermelho-azul.

Essas duas variações de problemas são muito importantes em SIG, de forma que optamos por implementar variantes dos algoritmos para os dois casos. No entanto, neste capítulo iremos nos concentrar no desenvolvimento realizado dos algoritmos para o problemas dos conjuntos vermelho-azul.

3.2.1. Força-Bruta Sequencial

A Figura 3.1 apresenta a implementação do ***algoritmo Força-Bruta sequencial*** para computar os pontos de intersecção entre dois conjuntos de segmentos, um chamado de vermelho e outro de azul. O método possui uma implementação trivial, com base no algoritmo apresentado na Seção 2.3.1.

```
01 vector<point>
02 lazy_intersection_rb(const vector<line_segment>& red_segments,
03                       const vector<line_segment>&
04 blue_segments)
05 {
06   vector<point> result;
07
08   point ip1;
09   point ip2;
10
11   const size_t rsize = red_segments.size();
12   const size_t bsize = blue_segments.size();
13
14   for(size_t i = 0; i != rsize; ++i)
15   {
16     const line_segment& red = red_segments[i];
17
18     for(size_t j = 0; j < bsize; ++j)
19     {
20        const line_segment& blue = blue_segments[j];
21
22        if(!do_bounding_box_intersects(red, blue))
23          continue;
24
25        segment_relation_type spatial_relation =
26                            compute_intesection_v3(red, blue,
27                                                   ip1, ip2);
28
29        if(spatial_relation == DISJOINT)
30          continue;
31
32        result.push_back(ip1);
33
34        if(spatial_relation == OVERLAP)
35          result.push_back(ip2);
36      }
37   }
38   return result;
39 }
```

Figura 3.1 – Implementação do algoritmo força-bruta para computação dos pontos de interseção entre dois conjuntos de segmentos.

3.2.2. Força-Bruta Multithread

As Figura 3.2 e 3.3 mostram a versão *multithread* do ***algoritmo Força-Bruta***. Nesta versão utilizamos um *pool* de threads para computar os pontos de intersecção de diferentes segmentos de forma concorrente, assim diminuindo o tempo de computação do algoritmo.

```
01 void
02 lazy_intersection_rb_thread(const vector<line_segment>&
03                              red_segments,
04                              const
05 vector<line_segment>&blue_segments,
06                              size_t nthreads,
07                              vector<vector<point> >&
08 intersetion_pts)
09 {
10   intersetion_pts.resize(nthreads);
11
12   vector<thread> threads;
13
14   for(size_t i = 0; i != nthreads; ++i)
15   {
16     intersection_computer ic = { i, nthreads,
17 &(intersetion_pts[i]),
18                                  &red_segments,
19 &blue_segments};
20
21     threads.push_back(thread(ic));
22   }
23
24   for(size_t i = 0; i != nthreads; ++i)
25     threads[i].join();
26 }
```

Figura 3.2 – Implementação do algoritmo força-bruta *multithread* para computação dos pontos de interseção entre dois conjuntos de segmentos.

```
01 struct intersection_computer
02 {
03   size_t thread_pos;
04   size_t num_threads;
05   vector<point>* ipts;
06   const vector<line_segment>* red_segments;
07   const vector<line_segment>* blue_segments;
08
09   void operator()()
10   {
11     point ip1;
12     point ip2;
13
14     size_t nred_segments = red_segments->size();
15     size_t nblue_segments = blue_segments->size();
16
17     for(size_t i = thread_pos; i < nred_segments;
18                                i += num_threads)
19     {
20       const line_segment& red = (*red_segments)[i];
21
22       for(size_t j = 0; j < nblue_segments; ++j)
23       {
24         const line_segment& blue = (*blue_segments)[j];
25
```

```
26          if(!do_bounding_box_intersects(red, blue))
27            continue;
28
29          segment_relation_type spatial_relation =
30                            compute_intesection_v3(red, blue,
31                                                   ip1, ip2);
32
33          if(spatial_relation == DISJOINT)
34            continue;
35
36          ipts->push_back(ip1);
37
38          if(spatial_relation == OVERLAP)
39            ipts->push_back(ip2);
40        }
41      }
42    }
43  };
```

Figura 3.3 – Thread de computação dos pontos de interseção do algoritmo de força-bruta.

3.2.3. x-Ordering Sequencial

As Figuras 3.4, 3.5 e 3.6 apresentam a implementação do ***algoritmo x-Ordering sequencial*** para computar os pontos de intersecção entre dois conjuntos de segmentos. Esta implementação possui um *functor* (estrutura que sobrecarrega o *operator*()) auxiliar que dado um segmento faz com que a primeira coordena seja a de menor abscissa (Figura 3.5). O *outrp functor* mostrado na Figura 3.6 realiza a comparação entre dois segmentos para dizer qual deles possui um dos pares de coordenada com menor abscissa.

```
01  vector<point>
02  x_order_intersection_rb(const vector<line_segment>&
03  red_segments,
04                          const vector<line_segment>&
05  blue_segmentes)
06  {
07    vector<point> ipts;
08    const size_t nred_segments = red_segments.size();
09    const size_t nblue_segments = blue_segments.size();
10
11    if((nred_segments == 0) || (nblue_segments == 0))
12      return ipts;
13
14    size_t nsegments = nred_segments + nblue_segments;
15
16    vector< pair< line_segment, color_type> >
17                  ordered_segments(nsegments);
18
```

```
19    auto it = transform(red_segments.begin(), red_segments.end(),
20                        ordered_segments.begin(),
21                        red_blue_sort_segment_xy(RED));
22
23    transform(blue_segments.begin(), blue_segments.end(), it,
24              red_blue_sort_segment_xy(BLUE));
25
26    sort(ordered_segments.begin(), ordered_segments.end(),
27         red_blue_segment_xy_cmp());
28
29    point ip1, ip2;
30
31    const size_t nbands = nsegments - 1;
32
33    for(size_t i = 0; i < nbands; ++i)
34    {
35      const auto& current_seg = ordered_segments[i];
36
37      for(j = i + 1; j < nsegments; ++j)
38      {
39        const auto& next_seg = ordered_segments[j];
40
41        if(current_seg.first.p2.x < next_seg.first.p1.x)
42          break;
43
44        if(current_seg.second == next_seg.second)
45          continue;
46
47
48  if(!do_y_interval_intersects(current_seg.first,next_seg.first))
49          continue;
50
51        segment_relation_type result =
52                  compute_intesection_v3(current_seg.first,
53                                         next_seg.first, ip1,
54  ip2);
55
56        if(result == DISJOINT)
57          continue;
58
59        ipts.push_back(ip1);
60
61        if(result == OVERLAP)
62          ipts.push_back(ip2);
63      }
64    }
65
66    return ipts;
67  }
```

Figura 3.4 – Implementação do algoritmo x-Ordering para computação de interseção entre dois conjuntos de segmentos.

```
01 struct red_blue_sort_segment_xy : unary_function<
02 line_segment,
03                                    <line_segment, color_type>>
04 {
05   color_type color;
06
07   red_blue_sort_segment_xy(color_type c)
08     : color(c)
09   {
10   }
11
12   pair<line_segment, color_type> operator()(const
13 line_segment& s)
14   {
15     if(s.p1.x < s.p2.x)
16       return make_pair(s, color);
17
18     if(s.p1.x > s.p2.x)
19       return make_pair(line_segment(s.p2, s.p1), color);
20
21     if(s.p1.y < s.p2.y)
22       return make_pair(s, color);
23
24     if(s.p1.y > s.p2.y)
25       return make_pair(line_segment(s.p2, s.p1), color);
26
27     return make_pair(s, color);
28   }
29 };
```

Figura 3.5 – Ordenação das coordenadas de um segmento.

```
01 struct red_blue_segment_xy_cmp : pair< line_segment,
02 color_type> >
03 {
04   bool operator()(const pair<line_segment, color_type>& lhs,
05                   const pair<line_segment, color_type>&
06 rhs)const
07
08   {
09     if(lhs.first.p1.x < rhs.first.p1.x)
10       return true;
11
12     if(lhs.first.p1.x > rhs.first.p1.x)
13       return false;
14
15     if(lhs.first.p1.y < rhs.first.p1.y)
16       return true;
17
18     return false;
19   }
20 };
```

Figura 3.6 – Compara dois segmentos para determinar qual deles possui a coordenada de menor abscissa

3.2.4. x-Ordering Multithread

As Figura 3.7 e 3.8 mostram a versão *multithread* do ***algoritmo x-Ordering***. Esta versão utiliza um *pool* de threads para computar os pontos de intersecção de diferentes segmentos de forma concorrente. A Figura 3.8 apresenta a estrutura onde a computação dos pontos de intersecção é realizada.

```
01 void
02 x_order_intersection_rb_thread(const
03
04 vector<line_segment>&red_segments,
05                                const vector<line_segment>
06                                &blue_segments, size_t nthreads,
07
08 vector<vector<point>>&intersection_pts)
09 {
10   intersection_pts.resize(nthreads);
11
12
13   vector<thread> threads;
14
15   for(size_t i = 0; i != nthreads; ++i)
16   {
17     intersection_computer ic = {i, nthreads, nbands,
18 nsegments,
19                                 &(intersection_pts[i]),
20                                 &ordered_segments};
21
22     threads.push_back(thread(ic));
23   }
24
25   for(size_t i = 0; i != nthreads; ++i)
26     threads[i].join();
27 }
```

Figura 3.7 – Compara dois segmentos para determinar qual deles possui a coordenada de menor abscissa.

```
01 struct intersection_computer
02
03 {
04   size_t thread_pos;
05   size_t num_threads;
06   size_t nbands;
07   size_t nsegments;
08   vector<point>* ipts;
09   vector<pair< line_segment, color_type> >* ordered_segments;
10
11
12   void operator()()
13   {
14     point ip1;
15     point ip2;
16     for(size_t i = thread_pos; i < nbands; i += num_threads)
17     {
18       const auto& current_seg = (*ordered_segments)[i];
19
20
21       for(size_t j = i + 1; j < nsegments; ++j)
22       {
```

```
23          const auto& next_seg = (*ordered_segments)[j];
24
25          if(current_seg.first.p2.x < next_seg.first.p1.x)
26            break;
27
28
29          if(current_seg.second == next_seg.second)
30            continue;
31
32
33
34          if(!do_y_interval_intersects(current_seg.first,
35                                       next_seg.first))
36            continue;
37
38
39          segment_relation_type result =
40
41  compute_intesection_v3(current_seg.first,
42                             next_seg.first, ip1, ip2);
43
44          if(result == DISJOINT)
45            continue;
46
47          ipts->push_back(ip1);
48
49          if(result == OVERLAP)
50            ipts->push_back(ip2);
51        }
52      }
53    }
54  };
```

Figura 3.8 – Thread de computação dos pontos de interseção do algoritmo de x-Ordering.

3.2.5. Fixe-Grid Sequencial

A Figura 3.9 apresenta a implementação ***do algoritmo fixe-grid sequencial*** para computar os pontos de intersecção entre dois conjuntos de segmentos. Esta versão possui uma grande quantidade de operações devido à separação dos conjuntos de segmentos vermelho e azul em grades. Para este algoritmo foi necessário adicionar alguns dados de entrada como dimensão em *x* (*dx*), dimensão em *y* (*dy*), valor máximo e mínimo de *x* e *y* (*xmin, xmax, ymin, ymax*), e com esses dados dimensionar o tamanho da grade.

```
01  vector <point>
02  fixed_grid_intersection_rb(vector<line_segment>&
03  red_segments,
04                             const vector<line_segment>
05  &blue_segments,
06                             double dx, double dy, double xmin,
07                             double xmax, double ymin, double
08  ymax)
09  {
10    vector< point> ipts;
11
12    size_t nrows = ceil(((ymax - ymin) / dy));
13
14    const size_t nred_segments = red_segments.size();
15    const size_t nblue_segments = blue_segments.size();
16
17    multimap<size_t, size_t>blue_grid;
18
19    for(size_t i = 0; i != nblue_segments; ++i)
20    {
21      const line_segment& blue = blue_segments[i];
22
23      size_t first_col = (blue.p1.x - xmin) / dx;
24      size_t first_row = (blue.p1.y - ymin) / dy;
25
26      size_t second_col = (blue.p2.x - xmin) / dx;
27      size_t second_row = (blue.p2.y - ymin) / dy;
28
29      pair<size_t, size_t> min_max_col =
30                             minmax(first_col, second_col);
31      pair<size_t, size_t> min_max_row =
32                             minmax(first_row, second_row);
33
34      for(size_t col = min_max_col.first;
35          col <= min_max_col.second; ++col)
36      {
37        size_t offset = col * nrows;
38
39        for(size_t row = min_max_row.first;
40            row <= min_max_row.second; ++row)
41        {
42          size_t k = row + offset;
43
44          blue_grid.insert(make_pair(k, i));
45        }
46      }
47    }
48    point ip1;
49    point ip2;
50
51    for(size_t i = 0; i < nred_segments; ++i)
52    {
```

```
53      const auto& red = red_segments[i];
54      size_t first_col = (red.p1.x - xmin) / dx;
55      size_t first_row = (red.p1.y - ymin) / dy;
56      size_t second_col = (red.p2.x - xmin) / dx;
57      size_t second_row = (red.p2.y - ymin) / dy;
58
59      pair<size_t, size_t> min_max_col =
60           minmax(first_col, second_col);
61      pair<size_t, size_t> min_max_row =
62           minmax(first_row, second_row);
63
64      for(size_t col = min_max_col.first;
65          col <= min_max_col.second; ++col)
66      {
67        size_t offset = col * nrows;
68
69        for(size_t row = min_max_row.first;
70            row <= min_max_row.second; ++row)
71        {
72          size_t k = row + offset;
73
74          auto range = blue_grid.equal_range(k);
75
76          while(range.first != range.second)
77          {
78            size_t blue_idx = range.first->second;
79
80            const auto& blue = blue_segments[blue_idx];
81
82            if(do_bounding_box_intersects(red, blue))
83            {
84              segment_relation_type spatial_relation =
85                        compute_intesection_v3(red, blue,
86                                               ip1, ip2);
87
88              if(spatial_relation != DISJOINT)
90              {
91                if(is_in_cell(xmin, ymin, dx, dy,
92                              col, row, ip1.x, ip1.y))
93                  ipts.push_back(ip1);
94
95                if(spatial_relation == OVERLAP)
96                {
97                  if(is_in_cell(xmin, ymin, dx, dy,
98                                col, row, ip2.x, ip2.y))
99                    ipts.push_back(ip2);
100               }
101             }
102           }
103           ++(range.first);
104         }
105       }
106     }
107   }
108
109   return ipts;
```

```
110 }
```

Figura 3.9- Implementação do algoritmo fixed-grid para computação dos pontos de interseção entre dois conjuntos de segmentos

3.2.6. Fixed-Grid Multi-thread

As Figura 3.10 e 3.11 mostram a versão *multithread* do ***algoritmo fixed-grid***. Nesta versão utilizamos um *pool* de threads para computar os pontos de intersecção de diferentes segmentos de forma concorrente ,esta versão possui grande parte de sua implementação paralelizada, desta forma procuramos maximizar seu desempenho sendo que grande parte de suas operações são feitas de forma concorrente.

```
01 void
02 fixed_grid_intersection_rb_thread
03                              (const
04                               vector<line_segment>&red_segments,
05                               const vector<line_segment>
06                               &blue_segments, Size_t nthreads,
07                               double dx, double dy,
08                               double xmin, double xmax,
09                               double ymin, double ymax,
10                               Vector <vector< point> >&
11 intersetion_pts)
12 {
13   intersetion_pts.resize(nthreads);
14
15   vector<thread> threads;
16
17   for(size_t i = 0; i != nthreads; ++i)
18   {
19     intersection_computer ic = {i, nthreads,dx ,dy ,xmin ,ymin
20 ,
21
22 nrows,&(intersetion_pts[i]),&blue_grid
23                                 ,&red_segments ,&blue_segments};
24     threads.push_back(thread(ic));
25   }
26
27   for(size_t i = 0; i != nthreads; ++i)
28     threads[i].join();
29 }
```

Figura 3.10 - Implementação do algoritmo fixed-grid multithread para computação dos pontos de interseção entre dois conjuntos de segmentos

```
01 struct intersection_computer
02 {
03   size_t thread_pos;
04   size_t num_threads;
05   double dx;
06   double dy;
07   double xmin;
08   double ymin;
09   size_t nrows;
10   vector<point>* ipts;
11   multimap<size_t, size_t>* blue_grid;
12   const vector< line_segment>* red_segments;
13   const vector< line_segment>* blue_segments;
14
15   void operator()()
16   {
17       point ip1;
18       point ip2;
19       size_t nred_segments = red_segments->size();
20
21       for(size_t i = thread_pos; i <nred_segments;
22                                  i += num_threads)
23       {
24         const auto& red = (*red_segments)[i];
25
26         size_t first_col = (red.p1.x - xmin) / dx;
27         size_t first_row = (red.p1.y - ymin) / dy;
28
29         size_t second_col = (red.p2.x - xmin) / dx;
30         size_t second_row = (red.p2.y - ymin) / dy;
31
32         pair<size_t, size_t>
33                        min_max_col = minmax
34                        (first_col, second_col);
35         pair< size_t, size_t>
36                        min_max_row = minmax
37                        (first_row, second_row);
38
39
40         for(size_t col = min_max_col.first;
41             col <= min_max_col.second; ++col)
42         {
43           size_t offset = col * nrows;
44
45
46           for(size_t row = min_max_row.first;
47               row <= min_max_row.second; ++row)
48           {
49             size_t k = row + offset;
50
51             auto range = blue_grid->equal_range(k);
52
53
54             while(range.first != range.second)
55             {
56               size_t blue_idx = range.first->second;
```

```
57
58                const auto& blue = (*blue_segments)[blue_idx];
59
60
61                if(do_bounding_box_intersects(red, blue))
62                {
63                  segment_relation_type spatial_relation =
64                          compute_intesection_v3(red, blue,
65                                                 ip1, ip2);
66
67                  if(spatial_relation != DISJOINT)
68                  {
69                    if(is_in_cell(xmin, ymin, dx, dy,
70                                  col, row, ip1.x, ip1.y))
71                      ipts->push_back(ip1);
72
73                    if(spatial_relation == OVERLAP)
74                    {
75                      if(is_in_cell(xmin, ymin, dx, dy,
76                                    col, row, ip2.x, ip2.y))
77                        ipts->push_back(ip2);
78                    }
79                  }
80                }
81                ++(range.first);
82              }
83            }
84          }
85        }
86      }
87    };
```

Figura 3.11 - Thread de computação dos pontos de interseção do algoritmo de fixed-grid.

3.2.7. Tiling-Scheme Sequencial

A Figura 3.12 apresenta a implementação do ***algoritmo tiling-scheme sequencial*** para computar os pontos de intersecção entre dois conjuntos de segmentos. Está versão além dos conjuntos de segmentos, possui como entrada: dimensão *y* (*dy*), mínimo *y* (*ymin*) e maximo *y* (*ymax*), e a partir destes dados definir sua faixa horizontal.

```
01 vector <point>
02 tiling_intersection_rb (const vector<line_segment>&
03 red_segments,
04                         const vector<line_segment>&
05 blue_segments,
06                         double dy, double ymin, double ymax)
07 {
08   vector<point> ipts;
09   size_t nrows = ceil(((ymax - ymin) / dy));
10
11   vector< vector< line_segment> > red_tile_idx(nrows + 1);
12   vector< vector< line_segment> > blue_tile_idx(nrows + 1);
13
14   for(const auto& red : red_segments)
15   {
16     size_t first_row = (red.p1.y - ymin) / dy;
17     size_t second_row = (red.p2.y - ymin) / dy;
18
19     pair<size_t, size_t> min_max_row = minmax(first_row,
20                                               second_row);
21     for(size_t row = min_max_row.first;
22         row <= min_max_row.second; ++row)
23     {
24       red_tile_idx[row].push_back(red);
25     }
26   }
27
28   for(const auto& blue : blue_segments)
29   {
30     size_t first_row = (blue.p1.y - ymin) / dy;
31     size_t second_row = (blue.p2.y - ymin) / dy;
32
33     pair< size_t, size_t> min_max_row = minmax(first_row,
34                                                second_row);
35
36     for(size_t row = min_max_row.first;
37          row <= min_max_row.second; ++row)
38     {
39       blue_tile_idx[row].push_back(blue);
40     }
41   }
42
43
44   for(size_t i = 0; i <= nrows; ++i)
```

```
45    {
46      const vector< line_segment>& r_segs = red_tile_idx[i];
47      const vector< line_segment>& b_segs = blue_tile_idx[i];
48
49      vector<point> ips = x_order_intersection_rb(r_segs,
50  b_segs);
51
52      copy_if(ips.begin(), ips.end(),back_inserter(ipts),
53              [&i, &dy, &ymin] (const point& ip)
54
55              { return is_in_tile(ymin, dy, i, ip.y); } );
56    }
57
58    return ipts;
59  }
```

Figura 3.12 - Implementação do algoritmo tiling-scheme para computação dos pontos de interseção entre dois conjuntos de segmentos.

3.2.8. Tiling-Scheme Multithread

As Figura 3.13 , 3.14 e 3.15 mostram a versão *multithread* do ***algoritmo tiling-scheme***. Nesta versão utilizamos um *pool* de threads para computar os pontos de intersecção de diferentes segmentos de forma concorrente, além de duas estruturas para computação de diferentes threads entre os algoritmos implementados. A primeira estruturas computa os pontos de intersecção, e a segunda separa os conjuntos de seguimentos vermelho e azul em suas respectivas faixas horizontais.

```
01  void
02  tiling_intersection_rb_thread (const vector< line_segment>&
03                                 red_segments, const vector<
04  line_segment>&
05                                 blue_segments, size_t
06  nthreads,double dy,
07                                 double ymin, double ymax,
08                                 vector<vector<point> >&
09  intersetion_pts)
10  {
11    size_t nrows = ceil(((ymax - ymin) / dy));
12
13    vector<vector<line_segment>> red_tile_idx(nrows + 1);
14    vector<vector<line_segment>> blue_tile_idx(nrows + 1);
15
16    intersetion_pts.resize(nthreads);
17
18    vector<thread> threads;
19    tiling _computer rt = {dy, ymin, &red_segments,
```

```
20  &red_tile_idx};
21    threads.push_back(thread(rt));
22
23    tiling _computer bt = {dy, ymin, &blue_segments,
24  &blue_tile_idx};
25    threads.push_back(thread(bt));
26
27    for(size_t i = 0; i != 2; ++i)
28      threads[i].join();
29
30    threads.clear();
31
32    for(size_t i = 0; i != nthreads; ++i)
33    {
34      intersection_computer ic = {i,nthreads,
35                                  &(intersetion_pts[i]),nrows,
36                                  dy, ymin,
37                                  &red_tile_idx,
38                                  &blue_tile_idx};
39                                  threads.push_back(thread(ic));
40    }
41
42    for(size_t i = 0; i != nthreads; ++i)
43      threads[i].join();
44  }
```

Figura 3.13 - Implementação do algoritmo tiling-scheme *multithread* para computação dos pontos de interseção entre dois conjuntos de segmentos.

```
01  struct intersection_computer
02  {
03    size_t thread_pos;
04    size_t nthread;
05    vector< point>* ipts;
06    const size_t nrows;
07    double dy;
08    double ymin;
09    const vector<vector<line_segment>>* red_tile_idx;
10    const vector<vector<line_segment>>* blue_tile_idx;
11
12    void operator()()
13    {
14
15      for(size_t i = thread_pos; i <= nrows; i += nthread)
16      {
17        double dy = this->dy;
18
19        double ymin = this->ymin;
20        const vector<line_segment>& r_segs = (*red_tile_idx)[i];
21        const vector<line_segment>& b_segs =
22  (*blue_tile_idx)[i];
23
24        vector<point> ips = x_order(r_segs , b_segs);
25
26        copy_if(ips.begin(), ips.end(),back_inserter(*ipts),
```

```
27                              [&i, &dy, &ymin] (const point& ip)
28                  { return is_in_tile(ymin, dy, i, ip.y); } );
29      }
30    }
31  };
```

Figura 3.14 - Thread de computação dos pontos de interseção do algoritmo de tiling-scheme.

```
01  struct tiling_computer
02    const double dy;
03    const double ymin;
04    const vector<line_segment>* segments;
05    vector<vector<line_segment>>* tile_idx;
06
07    void operator()()
08    {
09        for(const auto& seg : (*segments))
10        {
11          size_t first_row = (seg.p1.y - ymin) / dy;
12          size_t second_row = (seg.p2.y - ymin) / dy;
13
14          pair<size_t, size_t> min_max_row =
15                                  minmax(first_row, second_row);
16
17          for(size_t row = min_max_row.first;
18              row <= min_max_row.second; ++row)
19          {
20            (*tile_idx)[row].push_back(seg);
21          }
22        }
23    };
```

Figura 3.15 - separa os segmentos em suas respectivas faixas horizontais.

3.3. Dados de Testes

Para avaliar o desempenho dos algoritmos desenvolvidos, preparamos uma bateria de testes tomando como entrada conjuntos de dados reais. O primeiro conjunto de dados a ser utilizado nos testes contém a hidrografia de parte da região nordeste, conforme mostrado na Figura 3.16. Esse conjunto é formado por 2.544.805 segmentos.

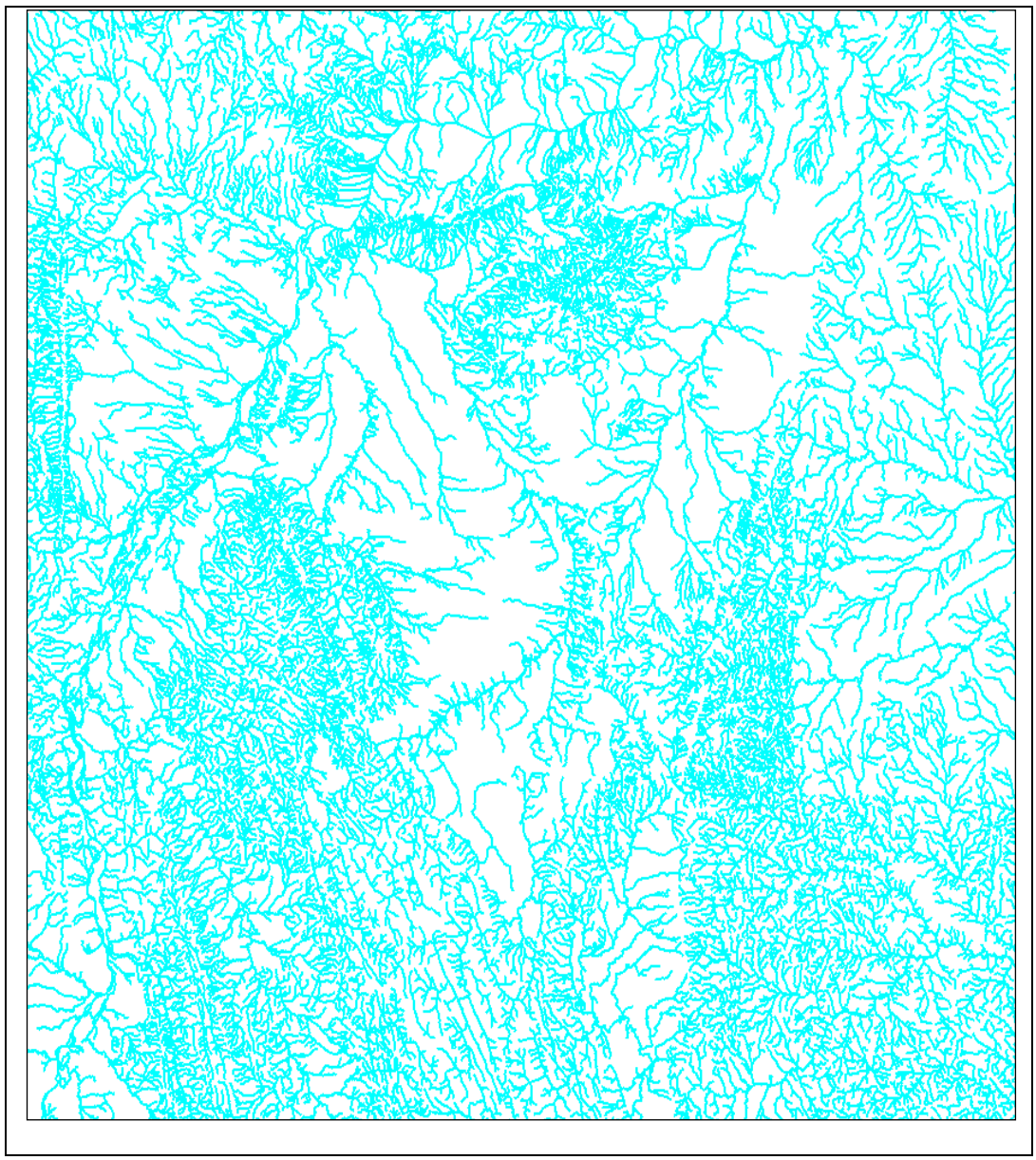

Figura 3.16 – Hidrografia de parte da região nordeste.

O segundo conjunto de dados utilizado, mostrado na Figura 3.17, contém 656.048 segmentos, representando trechos da malha viária da região nordeste.

Figura 3.17 – Malha Viária de parte da região nordeste.

O terceiro conjunto de dados utilizado, mostrado na Figura 3.18, contém 425.678 segmentos, representando a fronteira dos municípios do Estado de Goiás.

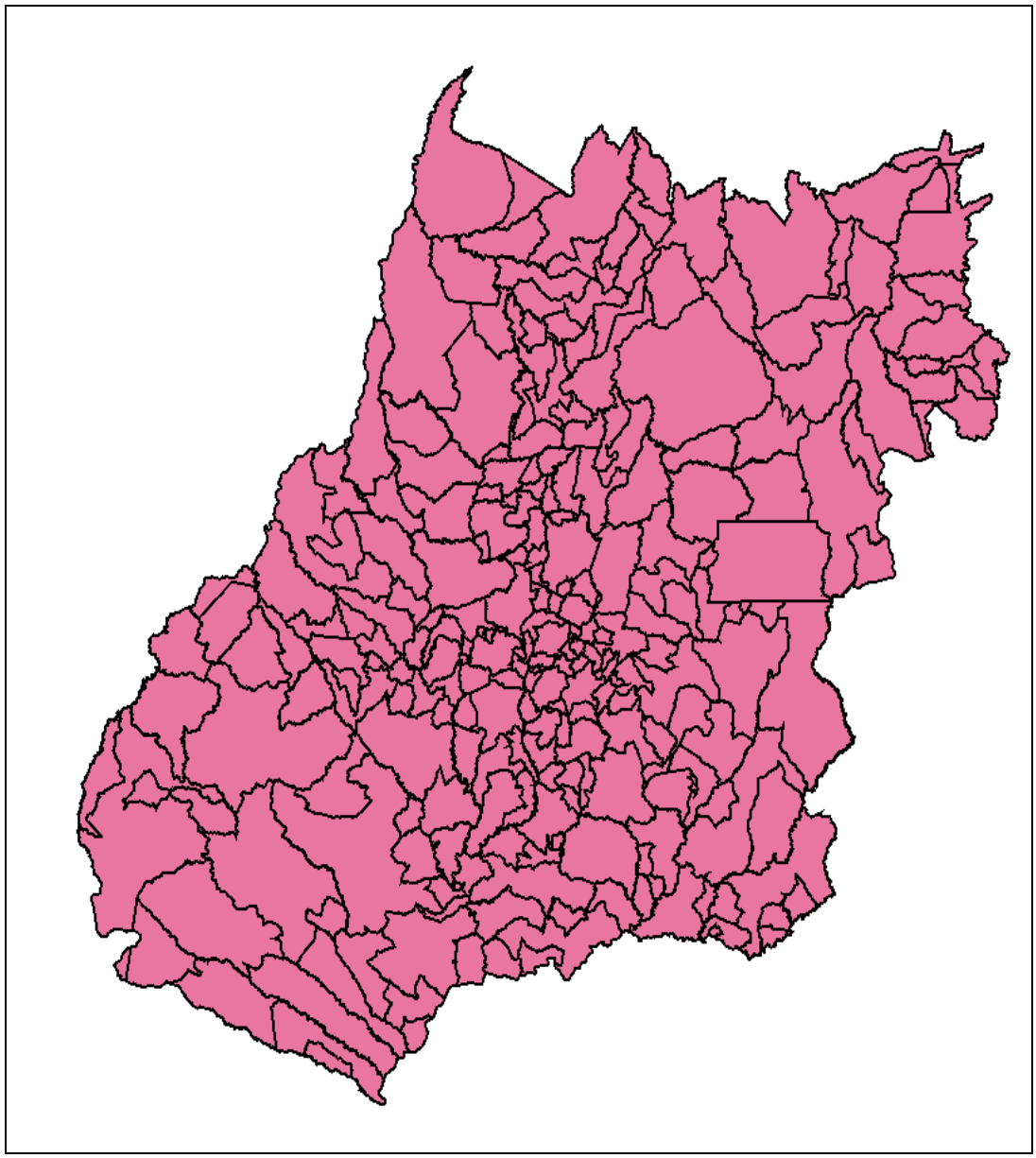

Figura 3.18 – Municípios do Estado de Goiás.

O quarto conjunto de dados utilizado, mostrado na Figura 3.19, contém 1.594.880 segmentos, representando o mapa geológico do Estado de Goiás.

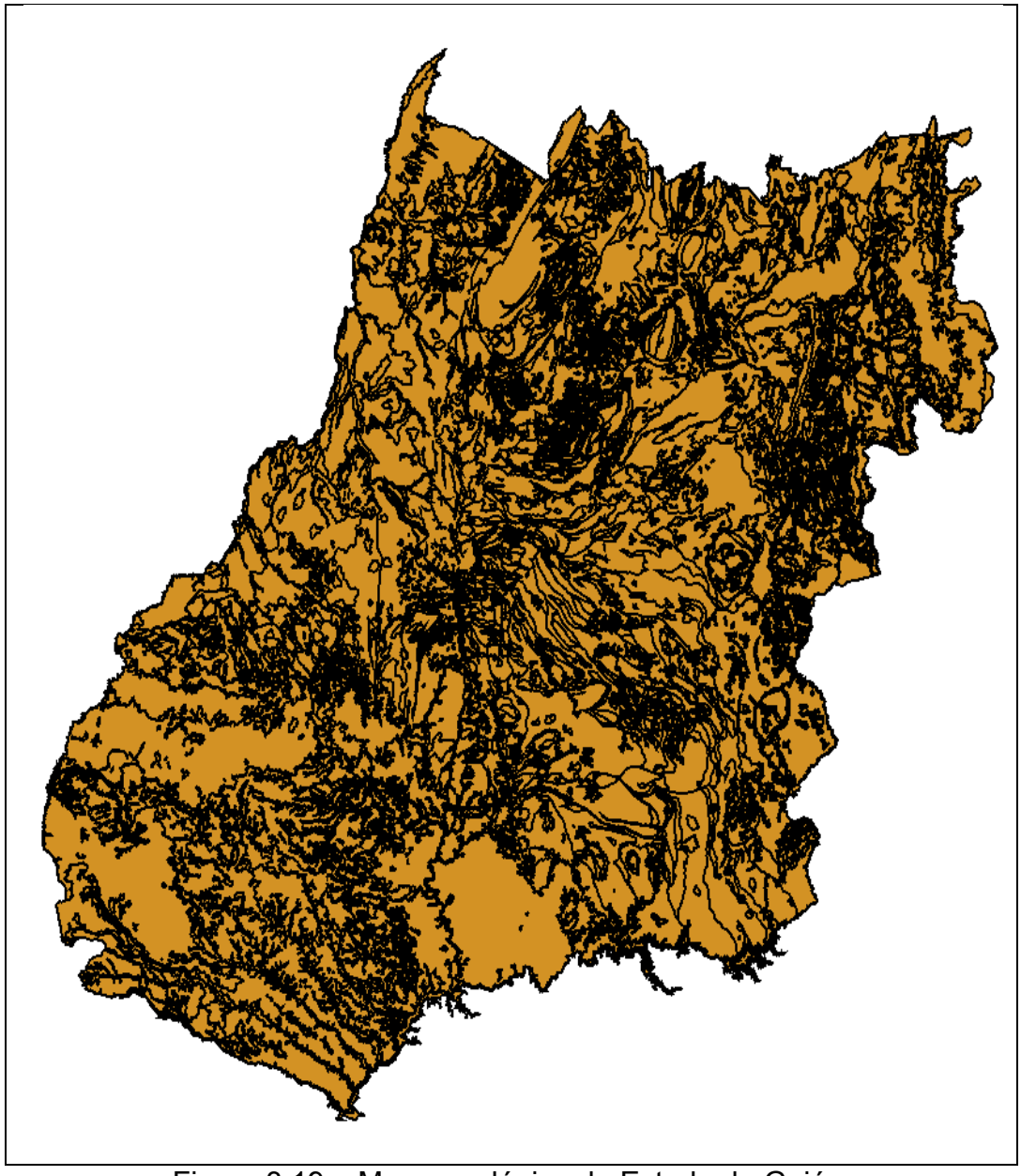

Figura 3.19 – Mapa geológico do Estado de Goiás.

4. RESULTADOS

Para verificar a diferença prática dos algoritmos implementados na Seção 3.2, realizamos uma bateria de testes com os dados geográficos descritos na seção 3.3. Para isso, utilizando a API da biblioteca TerraLib para acessar e manipular esses dados, transformando-os em segmentos de entrada para os algoritmos.

Para a avaliação dos tempos dos algoritmos, criamos dois casos de teste, T_1 e T_2. O caso de teste T_1 foi gerado com os segmentos de entrada correspondendo aos mapas de hidrografia (2.544.805 segmentos) e malha viária (656.048), que geraram 54.982 pontos de interseção. O caso T_2 foi gerado com os segmentos de entrada correspondendo aos mapas de municípios (425.678 segmentos) e geologia (1.594.880 segmentos), que geraram 23.476 pontos de intersecção.

A Tabela 4.1 apresenta os tempos, em segundos, para a execução dos algoritmos

Algoritmo	**T_1**	**T_2**
força-bruta sequencial	6152,477	1901,856
força-bruta multithread	1395,988	433,317
x-ordering sequencial	4,827	1,493
x-ordering multithread	1,446	0,568
fixed-grid sequencial	0,550	0,605
fixed-grid multithread	0,376	0,534
tiling-scheme sequencial	0,272	0,162
tiling-scheme multithread	0,114	0,085

Tabela 4.1 – desempenho dos algoritmos sequenciais.

Entre os algoritmos sequencias, o *tiling-scheme* foi o mais eficiente, confirmando os resultados obtidos por Pullar (1990). Isso se deve à estratégia de divisão dos segmentos em faixas horizontais combinadas com o algoritmo *x-ordering*.

Os resultados também indicam que o desempenho dos algoritmos com o uso de múltiplas threads (8-threads) melhorou consideravelmente o tempo de computação dos algoritmos. No caso do algoritmo *x-Ordering*, a versão com 8-threads foi três vezes mais rápida para computação dos pontos de interseção do que a versão sequencial.

O uso de threads também não alterou a ordem de eficiência dos algoritmos:

força-bruta > *x-ordering* > *fixed-grid* > *tiling-scheme*.

5. CONCLUSÕES E TRABALHOS FUTUROS

De acordo com os experimentos realizados, os algoritmos paralelizados através de threads se mostraram mais eficientes que as versões sequenciais. No entanto, é necessário elaborar uma bateria de testes que possibilite estudar o número de threads adequado aos algoritmos, além de formas de paralelizar completamente os algoritmos. Neste trabalho, paralelizamos apenas partes de cada algoritmo, de forma a ser necessário mais estudos nessa direção.

O algoritmo *tiling-scheme* apresentou o menor tempo de computação tanto de forma sequencial quanto na versão *multithread*, confirmando os resultados obtidos por Pullar (1990).

Como trabalho futuro podemos utilizar os algoritmos desenvolvidos neste trabalho para criação de operações de *overlay* de mapas.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDREWS, D. S.; SNOEYINK, J.; BORITZ, J. CHAN, T.; DENHAM, G.; HARRISON, J.; ZHU, C. **Further comparison of algorithms for geometric intersection problems**. In: International Symposium on Spatial Data Handling, 6., 1994, Edinburgh. Proceedings... Taylor and Francis, 1994, p. 709-724.

ANTONIO, F. Faster Line Segment Intersection. In: KIRK, D. (Ed.). **Graphics Gems III**. San Diego, CA, EUA: Academic Press, 1992. p. 199-202.

BENTLEY, J. L.; OTTMANN, T. A. Algorithms for reporting and counting geometric intersections**. IEEE Transactions on Computers**, v. C-28, n. 9, p. 643-647, set. 1979.

CAMARA, G.; CASANOVA, M. A.; HEMERLY, A. S.; MAGALHÃES, G. C.; MEDEIROS, C. M. B. **Anatomia de sistemas de informação geográfica**. Unicamp: Campinas, 1996.

CHAN, T. M. A Simple trapezoid sweep algorithm for reporting red/blue segment intersections**. 6th Can. Conf**. Comp. Geom. Saskatoon, Saskatchewan, 1994.

CHAZELLE, B.; EDELSBRUNNER, H. An optimal algorithm for intersecting line segments in the plane. **JACM**, v. 39, n. 1, p. 1-54, 1992.

FRANKLIN, W. R.; CHANDRASEKHAR, N.; KANKANHALLI, M.; SESHAN, M.; AKMAN, V. **Efficiency of uniform grids for intersection detection on serial and parallel machines**. In: Computer Graphics International, 1988, Geneva, Switzerland. Proceedings... Geneva, maio 1988, p. 51-62.

LONGLEY, P. A.; GOODCHILD, M. F.; MAGUIRE, D. J.; RHIND, D. W. **Geographic Information Systems and Science**. 2aEdição. John Wiley & Sons, 2005. 517 p.

PRASAD, M. Intersection of Line Segments. In: ARVO, J. (Ed.). **Graphics Gems II**. San Diego, CA, EUA: Academic Press, 1991. p. 7-9.

PREPARATA, F. P.; SHAMOS, M. I. **Computational Geometry an Introduction.** 1a Edição. New York: Springer-Verlag, 1985. 390 p.

PULLAR, D. **Comparative study of algorithms for reporting geometrical intersections**. In: International Symposium on Spatial Data Handling, 4., 1990, Zurich. Proceedings... 1990, p. 66-76.

SHAFFER, C, A; FEUSTEL, C. D. Exact Computation of 2-D Intersections. In: KIRK, D. (Ed.). **Graphics Gems III**. San Diego, CA, EUA: Academic Press, 1992. p. 188-192.

SHAMOS, M. I.; HOEY, DAN. Geometric intersection problems. 17th Annual Symposium on **Foundations of Computer Science**, Houston, TX, USA, 1976, pp. 208–215

APÊNDICE A - OUTROS EXEMPLOS DE MÉTODOS PARA COMPUTAÇÃO DE PONTOS DE INTERSECÇÃO ENTRE DOIS SEGMENTOS DE RETAS

As Figuras apresentadas nesta seção são referentes a algoritmos para computação do ponto de intersecção entre dois segmentos de reta, porem não utilizados por se mostrarem menos competitivos que o método apresentado por Antonio (1992).

A Figura A.1 Demonstra o método desenvolvido por Método de Prasad (1991). Código para computação dos pontos de interseção entre dois segmentos de reta.

```
01 comp_intersection(line_segment s1,line_segment s2,
02                                point first, point second)
03 {
04   double a1 = s1.p2.y - s1.p1.y;
05   double b1 = s1.p1.x - s1.p2.x;
06   double c1 = (s1.p2.x * s1.p1.y) - (s1.p1.x * s1.p2.y);
07   double r3 = a1 * s2.p1.x + b1 * s2.p1.y + c1;
08   double r4 = a1 * s2.p2.x + b1 * s2.p2.y + c1;
09
10   if((r3 != 0.0) && (r4 != 0.0) && same_signs(r3, r4))
11     return DISJOINT;
12   double a2 = s2.p2.y - s2.p1.y;
13   double b2 = s2.p1.x - s2.p2.x;
14   double c2 = (s2.p2.x * s2.p1.y) - (s2.p1.x * s2.p2.y);
15   double r1 = a2 * s1.p1.x + b2 * s1.p1.y + c2;
16   double r2 = a2 * s1.p2.x + b2 * s1.p2.y + c2;
17   if((r1 != 0.0) && (r2 != 0.0) && same_signs(r1, r2))
18     return DISJOINT;
19   double denom = a1 * b2 - a2 * c1;
20   if(denom == 0.0)  // are they collinear?
21   {
22     if(do_collinear_segments_intersects(s1, s2) == false)
23       return DISJOINT;
24   const point* pts[4];
25   pts[0] = &s1.p1;
26   pts[1] = &s1.p2;
27   pts[2] = &s2.p1;
28   pts[3] = &s2.p2;
29
30   sort(pts, pts + 4, point_cmp);
31   first = *pts[1];
32   if((pts[1]->x == pts[2]->x) && (pts[1]->y == pts[2]->y))
33     return TOUCH;
34   second = *pts[2];
35   return OVERLAP;
36   }
37   Ddouble offset = denom < 0.0 ? - denom / 2.0 : denom / 2.0;
```

```
38    Double num_alpha = b1 * c2 - b2 * c1;
39    first.x = (num_alpha < 0.0 ? num_alpha- offset
40                 : num_alpha + offset) / denom;
41    Double num_beta = a2 * c1 - a1 * c2;
42    first.y = (num_beta < 0.0 ? num_beta - offset
43                  : num_beta + offset) / denom;
44    Return CROSS;
45  }
```

Figura A.1 - Trecho de código para computação dos pontos de interseção entre dois segmentos de reta, de acordo com o método apresentado por Prasad (1991).

A Figura A.2 Demonstra o método desenvolvido por Método de Shaffer e Feustel (1992). Código para computação dos pontos de interseção entre dois segmentos de reta.

```
01  compute_intesection_v2(const line_segment& s1,
02                                          const line_segment& s2,
03                                          point& first, point&
04  second)
05  {
06    doublea = (s2.p1.x - s1.p1.x) * (s1.p2.y - s1.p1.y) –
07                    (s2.p1.y - s1.p1.y) * (s1.p2.x - s1.p1.x);
08    doubleb = (s2.p2.x - s1.p1.x) * (s1.p2.y - s1.p1.y) –
09                    (s2.p2.y - s1.p1.y) * (s1.p2.x - s1.p1.x);
10
11    if((a != 0.0) && (b != 0.0) &&same_signs(a, b))
12      return DISJOINT;
13
14    doublec = (s1.p1.x - s2.p1.x) * (s2.p2.y - s2.p1.y) –
15                    (s1.p1.y - s2.p1.y) * (s2.p2.x - s2.p1.x);
16    doubled = (s1.p2.x - s2.p1.x) * (s2.p2.y - s2.p1.y) –
17                    (s1.p2.y - s2.p1.y) * (s2.p2.x - s2.p1.x);
18
19    if((c != 0.0) && (d != 0.0) &&same_signs(c, d))
20      return DISJOINT;
21
22    doubledet = a - b;
23
24    if(det == 0.0)
25    {
26      If(do_collinear_segments_intersects(s1, s2) == false)
27        return DISJOINT;
28
29      intersection(s)
30      constpoint* pts[4];
31      pts[0] = &s1.p1;
32      pts[1] = &s1.p2;
33      pts[2] = &s2.p1;
```

```
34      pts[3] = &s2.p2;
35
36      sort(pts, pts + 4, point_cmp);
37      first = *pts[1];
38
39      if((pts[1]->x == pts[2]->x) && (pts[1]->y == pts[2]->y))
40        return TOUCH;
41
42      second = *pts[2];
43
44      return OVERLAP;
45      }
46   Ddouble tdet = -c;
47
48    if(det < 0.0)
49    {
50      det = -det;
51      tdet = -tdet;
52    }
53    Ddouble alpha = tdet / det;
54    first.x = s1.p1.x + alpha * (s1.p2.x - s1.p1.x);
55    first.y = s1.p1.y + alpha * (s1.p2.y - s1.p1.y);
56
57    return CROSS;
58  }
```

Figura A.2 - Pseudocódigo da computação dos pontos de interseção entre dois segmentos de reta, de acordo com o método apresentado por Clifford and Feustel (1992).

APÊNDICE B - EXEMPLOS DE ALGORITMOS DE INTERSECÇÃO ENTRE UM ÚNICO CONJUNTO DE SEGMENTOS

As Figuras apresentadas nesta seção são referentes a algoritmos para computação dos pontos de intersecção entre um único conjunto de segmentos de reta.

A Figura B.1 demonstra o ***Algoritmo de força-bruta*** desenvolvido para computar os pontos de intersecção entre um único conjunto de segmentos.

```
01 vector<point>
02 lazy_intersection(const vector<line_segment>& segments)
03 {
04   vector<point> result;
05   point ip1;
06   point ip2;
07   const size_t number_of_segments = segments.size();
08
09   for(std::size_t i = 0; i < number_of_segments; ++i)
10   {
11     const line_segment& red = segments[i];
12
13     for(size_t j = i; j < number_of_segments; ++j)
14     {
15       if(i == j)
16         continue;
17
18       const line_segment& blue = segments[j];
19
20       if(!do_bounding_box_intersects(red, blue))
21         continue;
22
23       segment_relation_type spatial_relation =
24 compute_intesection_v3(red, blue, ip1, ip2);
25
26       if(spatial_relation == DISJOINT)
27         continue;
28
29       result.push_back(ip1);
30
31       if(spatial_relation == OVERLAP)
32         result.push_back(ip2);
33     }
34   }
35   return result;
36 }
```

Figura B.1 - Algoritmo de força-bruta para computação de Intersecção entre um único conjunto de segmentos.

A Figura B.2 demonstra o ***algoritmo x-Ordering*** desenvolvido para computar os pontos de intersecção entre um único conjunto de segmentos.

```
01 vector<point>
02 x_order_intersection(vector< line_segment>& segments)
03 {
04   vector< point> ipts;
05   const std::size_t nsegments = segments.size();
06   if(nsegments <= 1)
07     return ipts;
08   vector<line_segment> ordered_segments(nsegments);
09   transform(segments.begin(), segments.end(),
10                    ordered_segments.begin(),   sort_segment_xy());
11   sort(ordered_segments.begin(), ordered_segments.end(),
12           line_segment_xy_cmp());
13   point ip1, ip2;
14
15   const size_t nbands = nsegments - 1;
16   for(size_t i = 0; i < nbands; ++i)
17   {
18     const line_segment& current_seg = ordered_segments[i];
19     for(size_t j = i + 1; j < nsegments; ++j)
20     {
21       const line_segment& next_seg = ordered_segments[j];
22       if(current_seg.p2.x < next_seg.p1.x)
23         break;
24       if(!do_y_interval_intersects(current_seg, next_seg))
25         continue;
26
27       segment_relation_type result =
28 compute_intesection_v3(current_seg,
29
30 next_seg, ip1, ip2);
31
32       if(result == DISJOINT)
33         continue;
34       ipts.push_back(ip1);
35       if(result == OVERLAP)
36         ipts.push_back(ip2);
37     }
38   }
39   return ipts;
40 }
```

Figura B.2 - Algoritmo x-Ordering para computação de Intersecção entre um único conjunto de segmentos.

A Figura B.3 demonstra o ***algoritmo tiling-scheme*** desenvolvido para computar os pontos de intersecção entre um único conjunto de segmentos.

```
01 vector<point>
02 tiling_intersection(vector<line_segment>& segments,
03                                const double& max_length, const
04 double& max_range,
05                                const double& min_range)
06 {
07   int range = (return_positive_value(max_range) / max_length);
08   vector<point> ipts;
09   vector<line_segment> segments_range[4];
10   double t_max;
11   double block;
12   t_max = max_range + return_positive_value(min_range);
13   double block_size = t_max/range;
14   block = min_range + block_size;
15   for(int i = 0;i < segments.size(); ++i)
16   {
17      int cont = 0;
18     for(int y = block; y <= t_max; y += block_size)
19     {
20        if(segments[i].p1.y < y || segments[i].p2.y < y)
21       {
22         segments_range[cont].push_back(segments[i]);
23         if(segments[i].p1.y > y + block_size || segments[i].p2.y >
24 y +
25
26 block_size && y < t_max)
27            continue;
28         if((segments[i].p1.y > y || segments[i].p2.y > y) &&
29            y < t_max)
30           segments_range[cont+1].push_back(segments[i]);
31         break;
32       }
33       cont++;
34     }
35   }
36   x-ordering
37   int size = 0;
38   for(auto& elemento: segments_range)
39   {
40     ipts = x_order_intersection(elemento, block,(block-
41 block_size));
42     block += block_size;
43    size += ipts.size();
44   }
45   return ipts;
46 }
```

Figura B.3 - Algoritmo tiling-schame para computação de Intersecção entre um único conjunto de segmentos.