Flavio Signorelli Mendes

Soluções para o Problema do Planejamento de Trajetórias Utilizando Diagramas de Voronoi

Orientadora: Eliana Prado Lopes Aude

Rio de Janeiro

Março/2006

FICHA CATALOGRÁFICA

M538 Mendes, Flavio Signorelli

Soluções para o problema do planejamento de trajetórias utilizando diagramas de Voronoi / Flavio Signorelli Mendes. - Rio de Janeiro, 2006.

145f. : il.

Dissertação (Mestrado em Informática) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Núcleo de Computação Eletrônica, 2006.

Orientadora: Eliana Prado Lopes Aude

1. Diagramas de Voronoi - Teses. 2. Robôs Cooperativos - Teses 4. Computação Paralela - Teses. 4. Algoritmo Paralelo - Teses. 5. Simulador - Teses. I. Eliana Prado Lopes Aude (Orient.). II Universidade Federal do Rio de Janeiro. Instituto de Matemática. Núcleo de Computação Eletrônica. III. Título.

CDD

Soluções para o Problema do Planejamento de Trajetórias Utilizando Diagramas de Voronoi

Flavio Signorelli Mendes

Dissertação submetida ao Corpo Docente do Núcleo de Computação Eletrônica - Instituto de Matemática da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para a obtenção do grau de Mestre.

Aprovada por:

Profª. Eliana P. L. Aude, Ph.D.
(Orientadora)

Prof. Ernesto P. Lopes, Ph.D.

Prof. Fabrício Silva, Ph.D.

Prof. Jayme Szwarcfiter, Ph.D.

Rio de Janeiro, RJ – Brasil
Março – 2006

Aos meus avós Vincenzo, Dina, Brenno e Mizzi

Agradecimentos

A meu pai, minha mãe, meu irmão e minha tia Sonia pelo incentivo, por todo o esforço que fizeram e por sempre estarem ao meu lado.

Aos meus avós por todo o apoio que sempre me deram, especialmente ao vô Brenno e ao vô Vincenzo.

A minha orientadora Eliana Aude por sua dedicação, pelos conhecimentos transmitidos e pelo exemplo.

A minhas amigas Marcela e Patricia por me ajudar ao longo do mestrado.

A Henrique, Júlio, Mário, Rafael e todos os outros que convivi nesses anos de CONTROLAB.

Aos meus tios Attilio, Alfieri e Antônio Carlos pelo incentivo durante esse período.

Resumo

MENDES, F. S. **Soluções para o problema do planejamento de trajetórias utilizando diagramas de Voronoi.** 2006. 145 f. Tese (Mestrado em Informática) - Instituto de Matemática, Núcleo de Computação da Universidade Federal do Rio de Janeiro, Rio de Janeiro.

O conteúdo desta tese está relacionado ao problema do planejamento de trajetórias utilizando diagramas de Voronoi. É apresentada uma extensão ao algoritmo proposto em [1] que calcula o diagrama de Voronoi para ambientes tridimensionais. Uma solução paralela para esse cálculo também foi implementada e discutida. Além disso, uma revisão sobre técnicas de processamento paralelo é apresentada. Resultados experimentais mostram o *speed-up* da solução paralela proposta em relação a versão serial. Finalmente, um simulador foi desenvolvido para estudar o problema do planejamento de trajetórias baseado em diagramas de Voronoi. Nesse simulador um time de robôs trabalha em conjunto para encontrar, em tempo real, o caminho seguro a ser seguido dentro de um ambiente tridimensional com obstáculos modelados em forma de esferas. Experimentos simulados são apresentados e discutidos.

Abstract

MENDES, F. S. **Soluções para o problema do planejamento de trajetórias utilizando diagramas de Voronoi.** 2006. 145 f. Tese (Mestrado em Informática) - Instituto de Matemática, Núcleo de Computação da Universidade Federal do Rio de Janeiro, Rio de Janeiro.

This thesis is concerned with the path planning problem based on Voronoi diagrams. An extension to the algorithm proposed in [1] for calculating the Voronoi diagram for three dimensional environments is presented. A parallel solution for this calculation is also implemented and discussed. A review of the general parallel processing techniques is presented. Experimental results show the speed-up of the proposed parallel solution in relation to the non parallel one. Finally, a simulator was designed to study the path planning problem based on Voronoi diagrams where a team of robots work together to find, in real-time, the safest path to be followed within three dimensional environment full obstacles which are modeled as spheres. Simulated experiments are presented and discussed.

Lista de Figuras

1.1 Robô industrial KUKA em uma linha de montagem . . . . . p. 17
1.2 Robô humanóide ASIMO . . . . . p. 18
1.3 Robô autônomo CONTROLAB AGV . . . . . p. 20
1.4 Planta-baixa do laboratório CONTROLAB, sendo R o AGV . . . . . p. 21
1.5 CONTROLAB AGV utilizando campos potenciais . . . . . p. 23
1.6 CONTROLAB AGV desviando de um obstáculo . . . . . p. 24
1.7 Um caminho planejado utilizando Grafos de Visibilidade . . . . . p. 25
1.8 Exploração de um labirinto por um time de robôs . . . . . p. 28
2.1 Diagrama de Voronoi para um conjunto de pontos . . . . . p. 35
2.2 Construção da região de Voronoi para o ponto p_1 . . . . . p. 36
2.3 Lugares do plano com pontos eqüidistantes . . . . . p. 38
2.4 Aresta do diagrama passando por uma célula do grid . . . . . p. 40
2.5 Diagrama de Voronoi calculado com baixa precisão . . . . . p. 41
2.6 Algoritmo em funcionamento . . . . . p. 42
2.7 Algoritmo para o cálculo do Diagrama de Voronoi . . . . . p. 44
2.8 Raio de um robô com forma arbitrária . . . . . p. 45
2.9 Rótulos dos vértices das faces . . . . . p. 47
2.10 Exemplo de células em uma área de trabalho . . . . . p. 48
2.11 Aresta de Voronoi cortando células vizinhas . . . . . p. 49
2.12 Pseudo-código do algoritmo numérico para o cálculo diagrama de Voronoi em 3D . . . . . p. 51
2.13 Diagrama de Voronoi em 3D calculado com baixa precisão . . . . . p. 53
3.1 Exemplo de Hash com resolução de colisão . . . . . p. 59
3.2 Registros na lista encadeada . . . . . p. 64

3.3 Estrutura Hash+Bloom . . . . . p. 66
4.1 Exemplo de arquitetura SMP . . . . . p. 75
4.2 Arquiteturas MPP e NUMA . . . . . p. 77
4.3 Funções básicas da biblioteca MPI . . . . . p. 80
4.4 Operações de movimento de dados da biblioteca MPI . . . . . p. 84
4.5 Exemplo de Decomposição Regular de Domínio . . . . . p. 92
4.6 Matriz de dados de um processo . . . . . p. 93
4.7 Granularidade . . . . . p. 94
4.8 Exemplo de gráfico de Speed-Up vs. Número de Processadores . . . . . p. 95
5.1 Particionamento da área de trabalho . . . . . p. 99
5.2 Pseudo-código do algoritmo numérico paralelo para o cálculo diagrama de Voronoi em 3D . . . . . p. 101
5.3 Pseudo-código da função para o tratamento de mensagens recebidas . . . p. 104
5.4 Configuração com três obstáculos calculada por dois processadores . . . . p. 106
5.5 Configuração com três obstáculos calculada por seis processadores . . . . p. 107
5.6 Configuração com oito obstáculos calculada por quatro processadores . . p. 108
6.1 Estrutura do Simulador . . . . . p. 111
6.2 Pseudo-código do módulo Observador . . . . . p. 112
6.3 Exemplo de cena 3D . . . . . p. 114
6.4 Parâmetro fov de gluPerspective() . . . . . p. 116
6.5 w e h para o cálculo do parâmetro aspect . . . . . p. 116
6.6 Planos de corte do tronco da pirâmide . . . . . p. 117
6.7 Primitivas gráficas em OpenGL . . . . . p. 118
6.8 Exemplo de código para o desenho de um polígono . . . . . p. 119
6.9 Pseudo-código do algoritmo de Dijkstra . . . . . p. 125
6.10 Encontrando a seqüência de vértices de s para t . . . . . p. 126
6.11 Pseudo-código do módulo Robô . . . . . p. 129
6.12 Pseudo-código da função Robo_em_Vértice() . . . . . p. 130
6.13 Pseudo-código da função descobre_próximo_estado() . . . . . p. 131

6.14 Logo após o passo inicial da simulação . . . . . p. 133
6.15 Time de robôs explorando a área de trabalho (parte 1) . . . . . p. 134
6.16 Time de robôs explorando a área de trabalho (parte 2) . . . . . p. 135
6.17 Time de robôs explorando a área de trabalho (parte 3) . . . . . p. 136
6.18 Time de robôs explorando a área de trabalho (parte 4) . . . . . p. 137
6.19 Time de robôs explorando a área de trabalho (parte 5) . . . . . p. 138
6.20 Área de trabalho totalmente explorada pelo time . . . . . p. 139
6.21 Estatísticas geradas pelo simulador para o experimento apresentado . . . p. 140

Lista de Tabelas

3.1 Valores dos vetores val e col_ind no formato CRS. . . . . p. 57

3.2 Valores para row_ptr no formato CRS . . . . . p. 58

3.3 Resultados para precisão 4096 (em %) . . . . . p. 68

3.4 Resultados para precisão 8192 (em %) . . . . . p. 69

3.5 Resultados para precisão 16384 (em %) . . . . . p. 69

4.1 Classificação de arquiteturas paralelas . . . . . p. 72

5.1 Resultados . . . . . p. 106

5.2 Resultados . . . . . p. 106

Sumário

1 Introdução p. 16

1.1 Robôs Autônomos . . . . . p. 18

1.2 Planejamento de Trajetórias . . . . . p. 19

1.2.1 Campos Potenciais . . . . . p. 22

1.2.2 Grafos de Visibilidade . . . . . p. 24

1.2.3 Diagramas de Voronoi . . . . . p. 25

1.3 Robôs Cooperativos . . . . . p. 27

1.4 Trabalhos Relacionados . . . . . p. 29

1.5 Descrição do Trabalho . . . . . p. 31

2 Diagramas de Voronoi p. 33

2.1 Diagramas de Voronoi em 2D . . . . . p. 33

2.2 Algoritmo para Pontos . . . . . p. 34

2.3 Algoritmo Incremental para Pontos . . . . . p. 36

2.4 Algoritmo Algébrico . . . . . p. 37

2.5 Algoritmo Numérico em 2D . . . . . p. 39

2.5.1 Pseudo-código do Algoritmo . . . . . p. 42

2.6 Diagramas de Voronoi em n Dimensões . . . . . p. 45

2.7 Algoritmo Numérico em 3D . . . . . p. 46

2.7.1 Pseudo-código do Algoritmo . . . . . p. 50

2.7.2 Comentários sobre o Algoritmo . . . . . p. 52

3 Estruturas de Dados p. 55

3.1 Matrizes Esparsas . . . . . p. 55

3.1.1 Compressed Row Storage . . . . . p. 56
3.2 Tabelas Hash . . . . . p. 58
3.3 Filtro de Bloom . . . . . p. 60
3.3.1 Probabilidade de Falsos Positivos . . . . . p. 61
3.4 Estrutura Hash+Bloom . . . . . p. 62
3.4.1 Motivação . . . . . p. 62
3.4.2 Implementação da Estrutura Hash+Bloom . . . . . p. 64
3.4.3 Resultados . . . . . p. 67

4 Computação Paralela p. 70
4.1 Crescimento da Computação Paralela . . . . . p. 70
4.2 Arquiteturas de Computadores Paralelos . . . . . p. 72
4.2.1 Paralelismo de Dados . . . . . p. 73
4.2.2 SMP . . . . . p. 74
4.2.3 NUMA . . . . . p. 76
4.2.4 MPP . . . . . p. 77
4.3 MPI . . . . . p. 78
4.3.1 Modelo de Programação . . . . . p. 79
4.3.2 Funções Básicas . . . . . p. 80
4.3.3 Comunicação Coletiva . . . . . p. 82
4.3.3.1 Barreira . . . . . p. 83
4.3.3.2 Movimento de Dados . . . . . p. 83
4.3.4 Comunicação Assíncrona . . . . . p. 85
4.4 Medidas de Desempenho . . . . . p. 86
4.4.1 Lei de Amdahl . . . . . p. 89
4.4.2 “Overhead” dos Algoritmos . . . . . p. 90
4.4.3 Desbalanceamento de Carga . . . . . p. 91
4.4.4 Comunicação e Sincronização . . . . . p. 91
4.5 Decomposição Regular de Domínio . . . . . p. 91

4.5.1 Atualizações nos Dados . . . . . p. 93
4.5.2 Desempenho . . . . . p. 95

5 Algoritmo Paralelo p. 97
5.1 Algoritmo Numérico 3D Paralelo . . . . . p. 97
5.2 Decomposição Regular de Domínio . . . . . p. 98
5.3 Pseudo-código do Algoritmo . . . . . p. 99
5.3.1 Inicialização . . . . . p. 100
5.3.2 Condição de Parada . . . . . p. 101
5.3.3 Troca de Mensagens . . . . . p. 103
5.3.4 Laço Principal . . . . . p. 104
5.4 Resultados . . . . . p. 105

6 Simulador p. 109
6.1 Características do Simulador . . . . . p. 109
6.2 Estrutura do Simulador . . . . . p. 110
6.3 Módulo Observador . . . . . p. 111
6.4 OpenGL . . . . . p. 113
6.4.1 Introdução . . . . . p. 113
6.4.2 A máquina de estados da OpenGL . . . . . p. 114
6.4.3 GLUT . . . . . p. 115
6.4.4 Projeção Perspectiva . . . . . p. 115
6.4.5 Primitivas Gráficas . . . . . p. 117
6.4.6 Transformações . . . . . p. 119
6.5 Módulo Robô . . . . . p. 120
6.5.1 Exploração do Ambiente . . . . . p. 120
6.5.2 Algoritmo de Dijkstra . . . . . p. 122
6.5.2.1 Grafos . . . . . p. 122
6.5.2.2 Algoritmo . . . . . p. 122

6.5.2.3 Descrição do Algoritmo . . . . . p. 123
6.5.2.4 Pseudo-código do Algoritmo . . . . . p. 124
6.5.2.5 Implementação . . . . . p. 125
6.5.3 Máquina de Estados do Módulo Robô . . . . . p. 127
6.5.4 Heurísticas . . . . . p. 127
6.5.5 Pseudo-código do Algoritmo de Exploração . . . . . p. 128
6.6 Resultados . . . . . p. 131

7 Conclusão p. 141
7.1 Trabalhos Futuros . . . . . p. 142

Referências p. 143

1 Introdução

A palavra *robô* é utilizada para se referir a uma larga gama de máquinas, tendo todas como característica comum a capacidade de movimento e a utilidade de realizar tarefas físicas. Os robôs podem ter várias formas distintas, desde humanóides, os quais replicam a forma e os movimentos humanos, até industriais, com sua aparência ditada pela tarefa a ser realizada. Normalmente, eles podem ser classificados em robôs móveis e robôs manipuladores. Um exemplo de robô móvel é um veículo autônomo. Um exemplo de manipulador é o braço mecânico do ônibus espacial.

Robôs podem ser controlados diretamente por humanos, tais como robôs para desarmamento de bombas por controle remoto ou o já citado braço do ônibus espacial, ou podem agir de acordo com sua própria capacidade de decisão, disponibilizada por técnicas de inteligência artificial.

Os robôs estão cada vez mais complexos e seu uso na indústria cada vez mais difundido. O principal uso dos robôs até hoje tem sido na automatização em indústrias de produção em massa, onde a mesma tarefa tem que ser repetida várias vezes exatamente da mesma forma. A produção de carros é o exemplo principal do emprego de robôs grandes e complexos em várias etapas da linha de montagem, tais como pintura, solda e montagem de partes. Esses tipos de tarefas repetitivas podem ser definidas facilmente e pouca realimentação é necessária para controlar a sua qualidade. Na figura 1.1 pode-se ver o

Figura 1.1: Robô industrial KUKA em uma linha de montagem

robô industrial KUKA, da empresa alemã KUKA Roboter, montando parte do chassis de um carro em uma linha de montagem.

Outra utilidade dos robôs é seu emprego em ambientes desagradáveis ou perigosos para humanos trabalharem. Como exemplos pode-se citar a limpeza de lixo tóxico, trabalhos no espaço, desarmamento de bombas, exploração no fundo do mar e mineração.

Robôs domésticos para realização de tarefas simples tais como aspirar pó, limpar chão e cortar grama já se encontram disponíveis no mercado. Até o fim de 2005, mais de 1.500.000 unidades dos robôs aspiradores de pó Roomba da empresa iRobot já tinham sido vendidas[1] . Outros exemplos de robôs domésticos são o Aibo da Sony e o Scooba também da iRobot.

Robôs humanóides estão sendo desenvolvidos com o objetivo de introduzir de forma

[1]http://www.irobot.com/sp.cfm?pageid=95

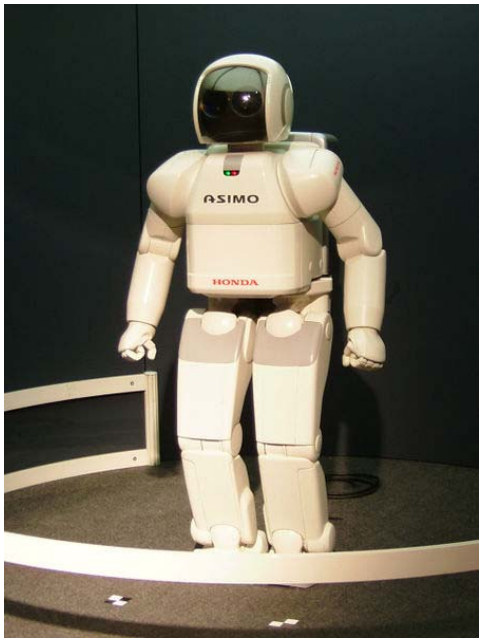

Figura 1.2: Robô humanóide ASIMO

mais agradável os robôs e suas funcionalidades na sociedade. Esse tipo de robô se enquadra na categoria de robôs sociais, com o objetivo de interagir e proporcionar companhia a seres humanos. Um exemplo de robô humanóide é o ASIMO da Honda[2], mostrado na figura 1.2.

1.1 Robôs Autônomos

A robótica é a ciência ou estudo da tecnologia associada com o desenvolvimento, fabricação, teoria e aplicação de robôs. Atualmente, um dos tópicos de pesquisa mais estudados nessa ciência é o desenvolvimento de robôs autônomos.

Robôs autônomos são caracterizados como sistemas inteligentes que integram percepção, raciocínio e ação, para realizar tarefas sob circunstâncias que são insuficientemente

[2]http://asimo.honda.com

conhecidas a priori e que modificam-se dinamicamente no decorrer de sua execução.

Embora as aparências e funcionalidades dos robôs sejam muito variadas, todos eles compartilham as características de uma estrutura mecânica e móvel sob alguma forma de controle. Dentre as principais partes de um robô pode-se citar seu corpo, seu motor, seus sensores, atuadores e alguma forma de armazenamento e ou obtenção de energia do ambiente para que todos os componentes funcionem.

Um robô autônomo pode ser colocado no seu ambiente de trabalho e funcionar sem intervenção externa, detectando automaticamente suas tarefas, ou pode funcionar recebendo ordens de usuários. Neste caso, basta que a ordem seja dada, não é necessário que o usuário explique como ela deverá ser realizada, tal como a definição da tarefa em um robô industrial. Mas, em ambos os casos, é necessário, dentre muitas outras coisas, que o robô saiba planejar sua trajetória de um ponto a outro no espaço.

O robô autônomo CONTROLAB AGV [2], visto na figura 1.3, é um exemplo desse tipo de pesquisa. O usuário envia, via Internet, ordens a um servidor dizendo para onde o AGV[3] deve ir. Esse servidor transmite por rádio as ordens ao robô. A partir daí, sem nenhuma intervenção do usuário o AGV planeja sua trajetória e parte para cumprir todas as ordens dadas, desviando de possíveis obstáculos presentes no seu caminho. O problema do planejamento de trajetórias enfrentado pelo AGV e por outros robôs será mais detalhado na próxima seção.

1.2 Planejamento de Trajetórias

O problema do planejamento de trajetórias consiste em encontrar um caminho adequado de um ponto de partida conhecido para um destino específico. Este caminho deverá

[3]Autonomous Guided Vehicle

Figura 1.3: Robô autônomo CONTROLAB AGV

ser traçado na área de trabalho do robô, e pode possuir ou não obstáculos.

Supondo o ambiente de trabalho em que o robô se encontra tridimensional, que o robô se movimente somente em um plano e que os obstáculos possam ser decompostos em poliedros convexos, com todas as faces paralelas ou ortogonais ao plano de movimentação do robô, pode-se reduzir o problema a um caso de planejamento de trajetória bidimensional.

Para planejar seu movimento, o robô precisa ter algum conhecimento sobre o ambiente no qual se encontra. Essas informações podem ser previamente fornecidas ao robô ou devem ser adquiridas com o auxílio de seus sistemas sensoriais.

Por exemplo, um robô em um ambiente de escritório pode ter informações sobre a localização de portas e das paredes já predefinidas em uma planta-baixa e utilizar seus sensores para detectar os obstáculos não previamente mapeados. Na figura 1.4 pode ser vista a planta-baixa do laboratório CONTROLAB, tal como a utilizada pelo AGV. Finalmente, de posse dessas informações sobre o ambiente, o robô tem que planejar uma

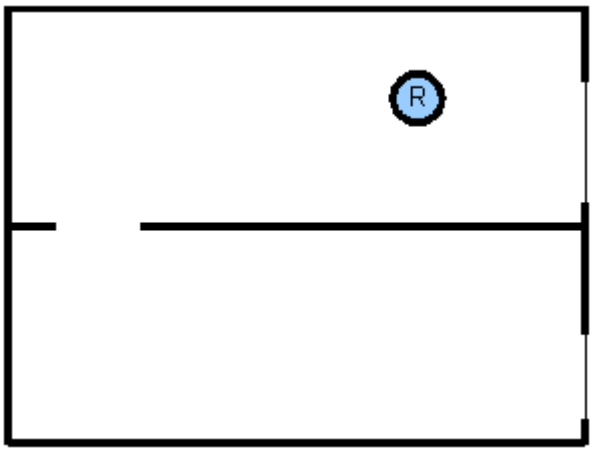

Figura 1.4: Planta-baixa do laboratório CONTROLAB, sendo R o AGV

trajetória da posição de origem para a de destino sem colidir com nenhum dos obstáculos no seu caminho.

Essa trajetória, não importando o método utilizado no seu cálculo, pode ser modificada devido a várias restrições. Como exemplo de restrições tem-se a área livre em relação ao tamanho do robô, o caminho mais curto entre origem e destino, conservação de energia, dentre outros. Existem restrições inerentes a um determinado tipo de ambiente – em um escritório, é melhor o robô passar pelo corredor e achar um caminho maior do que passar pela sala do diretor para encontrar o caminho mínimo.

A trajetória planejada pelo robô pode ser calculada utilizando diferentes métodos. Nas seções seguintes serão vistos três métodos de planejamento de trajetória. Dois deles são do tipo *roadmap*, pois geram caminhos parecidos com um mapa rodoviário. Nesse tipo de planejamento, o robô utiliza uma técnica chamada *retraction* que transforma uma área ou um volume em um conjunto de curvas unidimensionais contínuas. No resultado final esse conjunto de curvas é o caminho do robô.

O primeiro método apresentado, chamado de *Campos Potenciais*, não se trata de

um método *roadmap*. Nele o robô é uma partícula sob o efeito de um campo potencial artificial. Esse método será visto na seção 1.2.1.

Os dois métodos *roadmap* vistos são os *Grafos de Visibilidade*, seção 1.2.2, e os *Diagramas de Voronoi*, na seção 1.2.3. Este último é o método utilizado no presente trabalho.

1.2.1 Campos Potenciais

O método de Campos Potenciais[3, 4] é muito utilizado para resolver o problema do planejamento de trajetórias. Nesse método, o robô é considerado como uma partícula sob a influência de um campo potencial artificial U. As variações locais do campo U devem refletir a estrutura do ambiente ao seu redor.

A função de potencial é tipicamente, mas não necessariamente, definida sobre o espaço livre da área de trabalho como a soma de um potencial que atrai o robô para o seu destino e o repele dos obstáculos.

O planejamento de trajetória é realizado de forma iterativa. Sendo r a partícula que representa o robô, a cada iteração do algoritmo, a força artificial $\vec{F}(r)$ é dada por:

$$\vec{F}(r) = -\vec{\nabla}U(r)$$

Essa força $\vec{F}$ é tida como a direção mais promissora para o robô seguir. O planejamento de trajetória é realizado seguindo essa força. Mas, como o método de campos potenciais é essencialmente um método de otimização de máxima descida, ele pode ficar preso em um ponto de mínimo local da função de potencial, ponto este diferente do destino. Uma das técnicas para se evitar mínimos locais é a utilização de funções com poucos ou nenhum ponto de mínimo local[5].

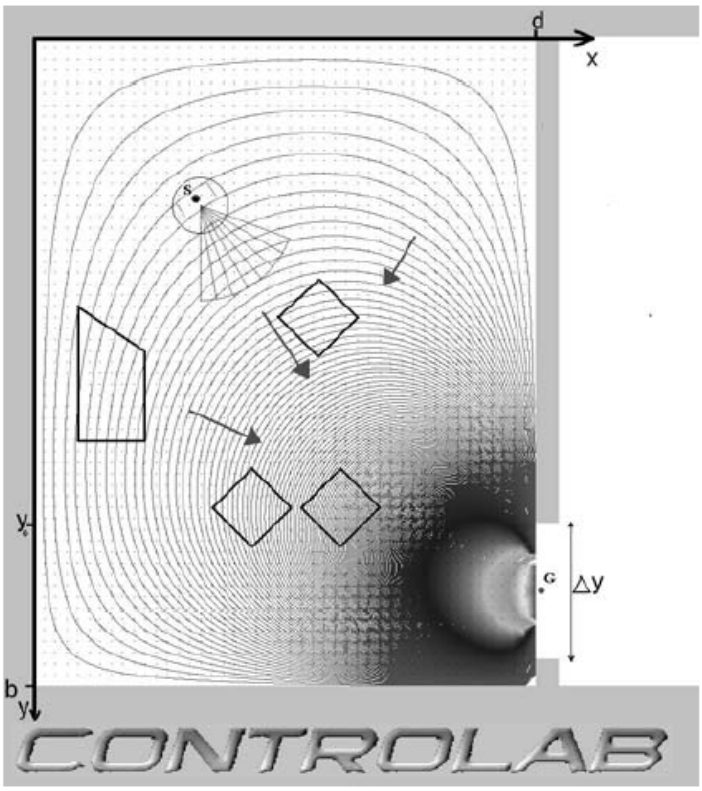

Figura 1.5: CONTROLAB AGV utilizando campos potenciais

Para resolver esse problema dos mínimos locais, pode-se usar *funções harmônicas*. A característica mais interessante dessa classe de funções é a ausência de pontos de mínimo local no seu domínio.

Na figura 1.5, pode-se ver o CONTROLAB AGV sob o efeito de um campo potencial devido a repulsão das paredes e a atração do ponto de destino, no caso, a porta. O robô, representado na figura pelo círculo, se encontra no ponto origem (S) e deseja ir para o ponto destino (G). O robô sempre usa a direção perpendicular as linhas de mesmo potencial (isolinhas) do campo para se movimentar.

O objeto triangular na frente do AGV é a sua *bengala eletrônica*[6], um conjunto de sensores infravermelhos utilizado para detecção de objetos não existentes na planta baixa.

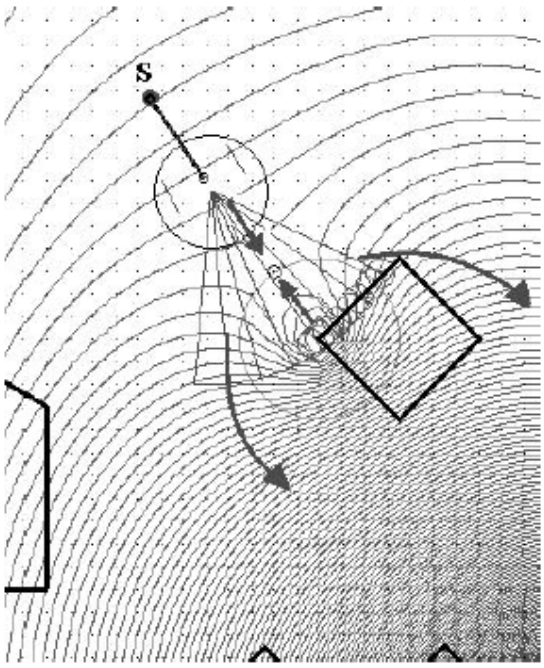

Figura 1.6: CONTROLAB AGV desviando de um obstáculo

A cada objeto detectado pela bengala, o planejador de trajetória do CONTROLAB AGV insere no campo uma partícula que repele o robô. Os quatro quadriláteros na figura representam obstáculos ainda não detectados pelo robô.

Na figura 1.6, é mostrado o AGV desviando de um obstáculo. Os pontos de contato da bengala com o obstáculo são os círculos pequenos na borda do quadrado. A adaptação do campo potencial original pode ser vista pela deformação nas linhas eqüipotenciais devido às cargas repelentes adicionadas.

1.2.2 Grafos de Visibilidade

O primeiro método de planejamento de trajetórias mostrado são os *Grafos de Visibilidade*[7, 5]. Nesse método, o robô se encontra em uma posição da área de trabalho (*origem*) e deseja locomover-se para um outro ponto (*destino*). A área de trabalho do

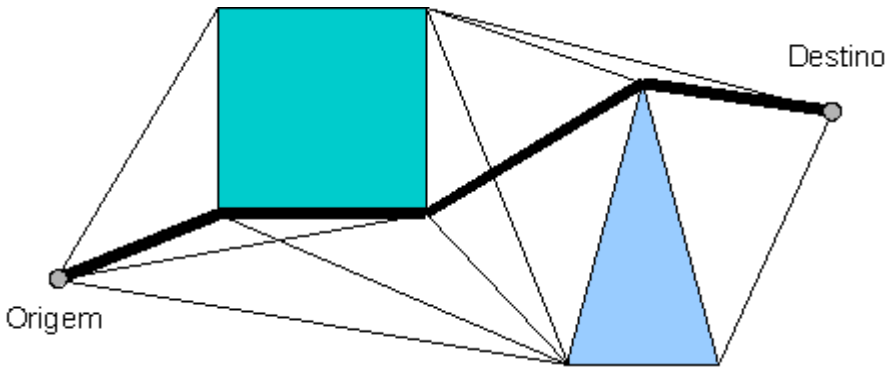

Figura 1.7: Um caminho planejado utilizando Grafos de Visibilidade

robô contém obstáculos poligonais.

Chamando de G um grafo de visibilidade, os vértices de G são dados pelos pontos de *origem* e *destino* do robô e todos os vértices dos obstáculos. Dois vértices de G estão conectados por uma aresta se, e somente se, o segmento de reta que os une não corta nenhum dos obstáculos ou o segmento é, exatamente, uma aresta de um obstáculo.

Estando montado o grafo G, o problema agora é encontrar um caminho entre os vértices de origem e destino. Para isso existem vários métodos, tal como o algoritmo de Dijkstra, visto no capítulo 3.

Na figura 1.7 pode ser visto um exemplo de grafo de visibilidade. Os vértices do grafo são os pontos de origem e destino, além do vértices dos polígonos. Um caminho, dentro dos muitos disponibilizados pelas arestas do grafo, para o robô se locomover entre a origem e o destino é mostrado pela linha mais grossa.

1.2.3 Diagramas de Voronoi

Como já foi dito, o problema do planejamento de trajetória tem que ser resolvido sempre que um robô queira se movimentar em algum espaço físico. No exemplo do CON-

TROLAB AGV, o robô tinha que se movimentar em um espaço relativamente seguro: um escritório. Para achar o menor caminho, dado um conjunto de restrições, o robô tende a passar perto dos obstáculos. Uma ligeira colisão do robô com a borda de uma cadeira, cuja posição não foi precisada pelos seus sensores, pode não apresentar danos para nenhuma das partes envolvidas.

Agora, tomando como exemplo um cenário de guerra, onde um robô submarino é utilizado para detecção de minas submersas colocadas em uma determinada área pelo inimigo. Uma simples colisão entre o robô e uma das minas pode significar o término, com falha, da missão.

Para ambientes perigosos como o do campo de minas submarinas, precisa-se de um tipo de planejamento de trajetórias diferente, que leve em conta obstáculos perigosos. Esse tipo é chamado de planejamento de trajetórias seguras. Usando um planejamento desse tipo, o robô se mantém o mais afastado possível dos obstáculos presentes no seu ambiente de trabalho, tentando assim minimizar a possibilidade de danos físicos.

O método de planejamento de trajetórias seguras apresentado nesse trabalho utiliza Diagramas de Voronoi. O diagrama de Voronoi de um ambiente com obstáculos é o conjunto de pontos eqüidistantes desses obstáculos. Uma descrição mais detalhada será dada no capítulo 2.

Os diagramas de Voronoi são largamente utilizados em vários campos da ciência e ocorrem espontaneamente em muitas escalas na natureza. Em antropologia e em geografia, são utilizados para descrever a ocupação do meio-ambiente pelo ser humano; na biologia, os padrões de domínio entre animais e competição de espécies de plantas; na química, modelam a aglomeração de átomos em estruturas cristalinas; na astronomia, a influência da gravidade em estrelas e seus aglomerados; em marketing, na estratégia de localização de

lojas de grandes cadeias; em robótica, como aplicado no trabalho atual, em planejamento de trajetórias; e, em ciência da computação, a solução para problemas de triangulação e de vizinho mais próximo (*nearest-neighbour problems*). Esse último tipo de problema consiste em, dados um ponto p qualquer e um conjunto de pontos vizinhos a ele, encontrar o ponto vizinho mais próximo de p.

1.3 Robôs Cooperativos

Outro tópico de pesquisa de grande importância em robótica atualmente é o estudo de técnicas de cooperação entre robôs. O objetivo é utilizar dois ou mais robôs cooperando para obter maior eficiência na realização de uma dada tarefa. Esse conjunto de robôs operando cooperativamente é chamado de *time*. De uma forma geral, aplicações nessa área começam com a definição da tarefa a ser executada pelo time de robôs. A partir daí, os robôs do time se coordenam para que a tarefa seja executada da melhor forma possível.

Um exemplo de aplicação de um time de robôs cooperativos é a exploração de um labirinto [8, 9] desenvolvida no CONTROLAB. Nessa aplicação, os robôs, representados por quadrados cinzas com borda preta na figura 1.8, têm que explorar um labirinto. Enquanto estão explorando, eles devem encontrar várias tarefas, que devem ser executadas em uma ordem pré-determinada. Os robôs, mesmo que já tenham encontrado a saída do labirinto, somente podem deixá-lo após a execução de todas as tarefas. As tarefas são representadas na figura pelos quadrados brancos hachurados e as já executadas pelos quadrados pretos.

Conforme os robôs vão explorando o labirinto, eles trocam informações sobre os caminhos já percorridos. Assim, evita-se que um robô percorra um caminho já explorado por outro. Com isso, diminui-se o trabalho redundante e a eficiência da exploração realizada

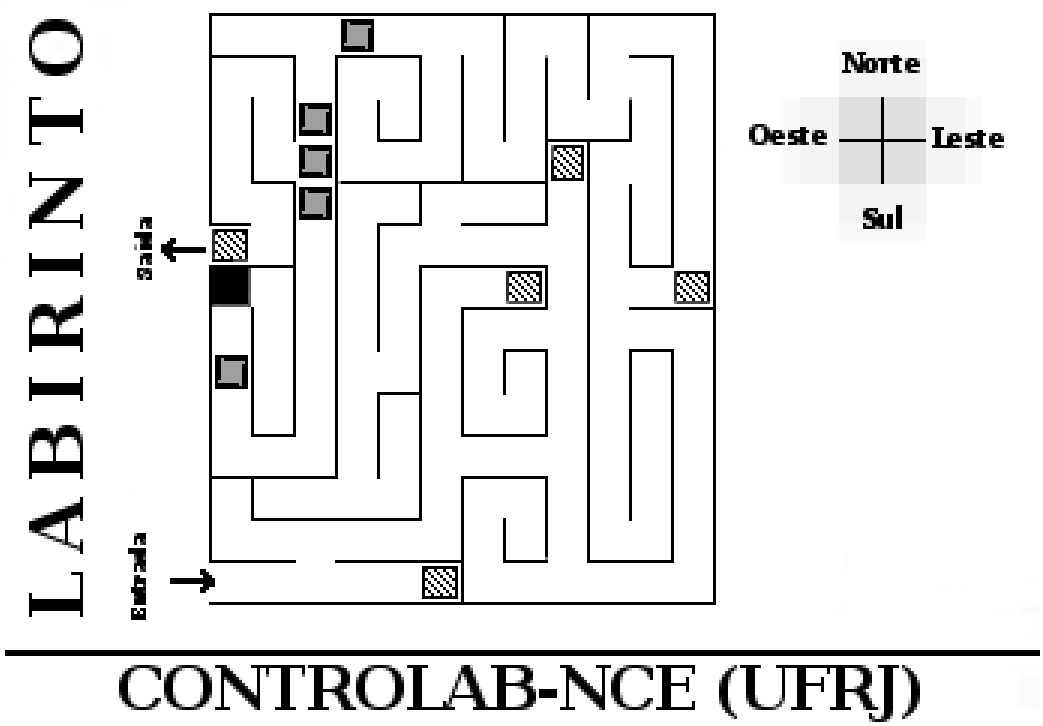

Figura 1.8: Exploração de um labirinto por um time de robôs

pelo time é aumentada.

As informações sobre a posição de cada tarefa e sobre a posição de saída do labirinto também são distribuídas entre eles. Quando todas as posições das tarefas forem descobertas, cada robô do time percorre o caminho mínimo para o local de execução da tarefa que lhe foi designada. Após terminada a execução da sua tarefa, cada robô se dirige, também pelo caminho mínimo, para a saída do labirinto. Quando todos os robôs do time saem do labirinto, a aplicação termina.

Geralmente, os robôs que compõem um time cooperativo são muito simples. Eles possuem algum equipamento de comunicação, necessário para a cooperação; um número limitado de sensores, o que torna sua percepção do ambiente de certa forma limitada; e

uma CPU[4], geralmente com restrições de poder computacional devido a necessidade de conservação de energia.

Para obter uma maior autonomia do robô, as CPUs geralmente empregadas nesse tipo de aplicação operam com clock mais baixo do que CPUs de estações de trabalho. Isso se deve as restrições térmicas e de consumo de energia da bateria.

Além disso, esses robôs possuem pouca memória, pois a necessidade constante de ciclos de recarga das RAM[5] dinâmicas também aumenta o consumo da bateria. Com uma CPU com pequeno poder de processamento e pouca memória disponível, faz-se necessária a divisão da carga total de trabalho entre cada um dos robôs membros do time para a execução da tarefa.

O problema do planejamento de trajetórias, discutido na seção anterior, dependendo da estratégia utilizada, coloca uma carga muito grande na já sobrecarregada CPU do robô do time cooperativo. Daí a necessidade da distribuição da carga de trabalho advinda da solução desse tipo de problema. Torna-se então necessária a utilização de técnicas de computação paralela para distribuir essa carga pelo time, diminuindo assim a quantidade de trabalho individual. Com isso, a CPU de cada robô tem mais tempo livre para realizar outras funções tais como processamento de dados coletados pelos sensores, entre outros.

1.4 Trabalhos Relacionados

Inúmeros trabalhos foram desenvolvidos com o objetivo de calcular o diagrama de Voronoi em n-dimensões para obstáculos pontuais, utilizando métricas de distância variadas, dentre elas a métrica euclidiana e a L_1 (distância Manhattan)[6]. Vários desses métodos

[4]CPU: Unidade Central de Processamento
[5]RAM: Memória de Acesso Aleatório
[6]Distância Manhattan (métrica L_1): $dist(p,q) = |p_{x_1} - q_{x_1}| + |p_{y_1} - q_{y_1}|$

podem ser vistos no artigo de F. Aurenhammer em [10]. Também podem ser encontrados nesse artigo aplicações para o diagrama de Voronoi desde a ciência da computação até suas aplicações em ciências naturais, tais como cristalografia e metalurgia.

Entre os algoritmos conhecidos para o cálculo de diagramas de Voronoi para pontos em duas, três ou mais dimensões estão algoritmo de M. Shamos [11], que utiliza o método de divisão e conquista, e o algoritmo de varredura proposto por S. Fortune [12].

H. Choset and J. Burdick mostraram em [13, 14] um esquema para planejamento de trajetórias baseado em informações coletadas por sensores. Esse método é baseado em estruturas do tipo *retract* chamadas de *Generalized Voronoi Graph* e, sua extensão, o *Hierarchical Generalized Voronoi Graph*. Nesse método, é necessário que o robô seja pontual ou esférico.

Um método para o cálculo do diagrama utilizando o hardware gráfico de estações de trabalho da empresa SGI foi proposto por K. Hoff em [15]. Nesse método, implementado em OpenGL, o diagrama é calculado utilizando o hardware de rasterização de polígonos para acelerar o processo.

Aplicações para o diagrama de Voronoi na área de redes podem ser vistas no trabalhos realizados por A. Jardosh [16] e no de B. Harrington [17]. No primeiro, o diagrama foi utilizado na construção de um modelo de mobilidade em redes móveis *ad hoc*. No segundo, foi utilizado em redes de sensores sem fio com o objetivo de reduzir o custo da troca de mensagens entre sensores da rede.

No caso de algoritmos para o cálculo paralelo do diagrama de Voronoi encontram-se, mais recentemente, os algoritmos apresentados por P.G. Tzionas [18] e por N. Sudha em [19]. Todos os dois foram desenvolvidos com o objetivo de serem implementados em

arquiteturas VLSI[7] celulares dedicadas. Essas arquiteturas são baseadas em paralelismo de granularidade fina e com forte acoplamento entre as células, que consistem de hardware muito simples. Por isso, a aplicação desses algoritmos em um ambiente distribuído, tal qual um time de robôs, não é indicada.

1.5 Descrição do Trabalho

O trabalho apresentado nesta tese está relacionado ao problema do planejamento de trajetórias utilizando diagramas de Voronoi. É apresentada uma extensão ao algoritmo proposto em[1] que calcula o diagrama de Voronoi para ambientes tridimensionais. Uma solução paralela para esse cálculo também foi implementada e discutida. Além disso, os resultados experimentais obtidos em uma rede de estações de trabalho mostram o *speed-up* da solução paralela proposta em relação a versão serial.

Finalmente, um simulador foi desenvolvido para estudar o problema do planejamento de trajetórias baseado em diagramas de Voronoi. Nesse simulador um time de robôs trabalha em conjunto para encontrar, em tempo real, o caminho seguro a ser seguido. A área de trabalho do time é um ambiente tridimensional com obstáculos modelados em forma de esferas. Experimentos simulados foram realizados e serão apresentados e discutidos.

A seguir, um resumo do conteúdo de cada um dos capítulos que compõem o presente trabalho é apresentado:

Capítulo 2 Neste capítulo é dada a definição formal do diagrama de Voronoi no plano. Em seguida, o diagrama é definido para o caso n-dimensional. Também são apresentados algoritmos para o seu cálculo em duas e três dimensões.

[7]VLSI: Very Large Scale Integration

Capítulo 3 Onde serão apresentados as estruturas de dados utilizadas na implementação do trabalho.

Capítulo 4 Onde será feita a revisão de técnicas de processamento paralelo. As arquiteturas paralelas mais utilizadas e seus modelos de programação também serão discutidos. Além disso, a biblioteca MPI, que foi utilizada no trabalho, será apresentada. Por último, medidas de desempenho utilizadas para analisar programas paralelos serão discutidas.

Capítulo 5 Neste capítulo é apresentada a solução paralela para o cálculo do diagrama de Voronoi. Os resultados obtidos serão comparados em relação a solução serial.

Capítulo 6 Apresenta o simulador para estudo do problema do planejamento de trajetórias baseado em diagramas de Voronoi por um time de robôs cooperativos. Também serão discutidos alguns experimentos simulados.

Capítulo 7 Onde são apresentadas as principais conclusões e são propostos trabalhos futuros.

2 Diagramas de Voronoi

Neste capítulo serão estudados diversas modalidades para determinação dos Diagramas de Voronoi. Estes diagramas são utilizados, em robótica, no planejamento de trajetórias do tipo *roadmap*. Nas seções 2.1 e 2.6 a idéia básica para o cálculo do diagrama de Voronoi para o caso bidimensional e n-dimensional é apresentada.

Quatro algoritmos para o cálculo do diagrama em duas dimensões são discutidos. O algoritmo para pontos, mostrado na seção 2.2, considera os obstáculos como objetos pontuais. O algoritmo incremental para objetos pontuais, na seção 2.3, constroi o diagrama adicionando os pontos um a um e modificando o diagrama do passo anterior. Para caso onde os obstáculos são objetos poligonais, são mostrados os algoritmos algébrico e numérico, nas seções 2.4 e 2.5 respectivamente.

Finalmente, é apresentado o algoritmo numérico para o caso onde a área de trabalho é tridimensional e os obstáculos são aproximados por volumes esféricos.

2.1 Diagramas de Voronoi em 2D

Um *diagrama de Voronoi* [20] em duas dimensões particiona o plano em regiões centradas em objetos de tal maneira que os pontos de uma região estejam mais próximos do objeto central desta região do que de qualquer outro objeto. Essas regiões são chamadas de *regiões de Voronoi*. A fronteira entre duas dessas regiões define uma *aresta de Voro-*

noi. Os pontos dessa fronteira que são eqüidistantes a três ou mais objetos no plano, são chamados de *vértices de Voronoi*. Já os pontos restantes, eqüidistantes a exatamente dois objetos, terminam de compor a aresta. Um exemplo de diagrama em que todos os objetos são pontos pode ser visto na figura 2.1.

Para definir matematicamente uma região de Voronoi, será necessária a definição de distância euclideana. A distância euclideana entre dois pontos p e q no plano será denotada por $dist(p, q)$ e é definida da seguinte forma:

$$dist(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Pode-se definir também a distância de um ponto p a um objeto O da seguinte forma:

$$dist_{obj}(O, p) = \inf_{x \in O} (dist(p, x))$$

Uma região de Voronoi para um determinado objeto O_i, denotada por $V(O_i)$, é definida por:

$$V(O_i) = \{x | dist_{obj}(O_i, x) \leq dist_{obj}(O_j, x), \forall j \neq i\}$$

2.2 Algoritmo para Pontos

Como citado em [21], o algoritmo para o cálculo de diagramas de Voronoi para obstáculos pontuais parte de uma idéia bem simples: cada região de Voronoi é a interseção de semi-planos gerados por linhas que separam o ponto dos outros.

Para dois pontos p_i e p_j, a mediatriz do segmento $\overline{p_i p_j}$ divide o plano em dois semi-

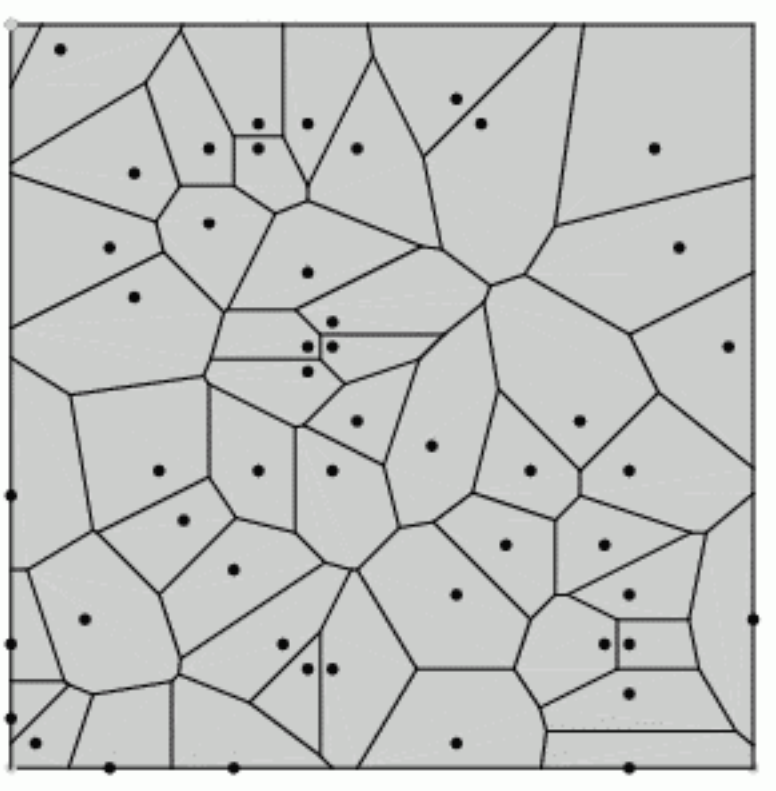

Figura 2.1: Diagrama de Voronoi para um conjunto de pontos

planos. Considerando p_i como ponto de interesse, define-se $h(p_i, p_j)$ como o semi-plano aberto que contém este ponto. O algoritmo acha $V(p_i)$ para todos os n pontos no plano. $V(p_i)$ é calculado da seguinte forma:

$$V(p_i) = \bigcap_{1 \leq j \leq n, j \neq i} h(p_i, p_j)$$

Na figura 2.2 podemos ver a construção da região de Voronoi do ponto p_1. Em 2.2a, é computado $h(p_1, p_2)$. Em 2.2b, é feita a interseção entre $h(p_1, p_2)$ e $h(p_1, p_3)$. Já em 2.2c, é realizada a interseção do resultado de 2.2b com $h(p_1, p_4)$. Finalmente, podemos ver em 2.2d, a região $V(p_1)$. Agora é só repetir o processo para os pontos p_2, p_3 e p_4 para ter calculado o diagrama de Voronoi.

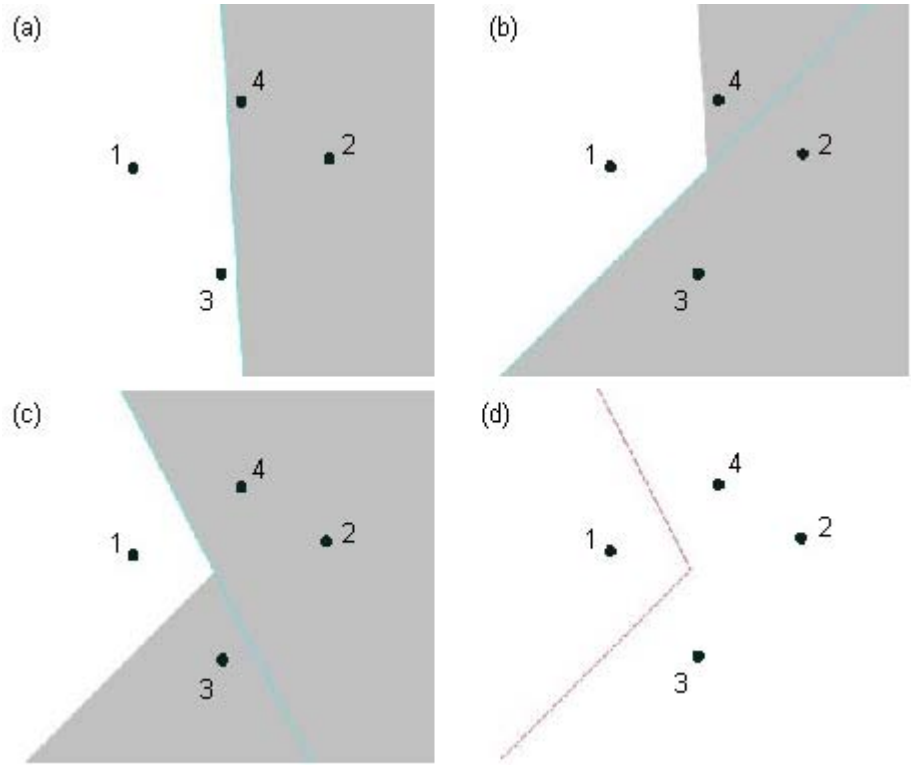

Figura 2.2: Construção da região de Voronoi para o ponto p_1

2.3 Algoritmo Incremental para Pontos

Um outro algoritmo para cálculo de diagramas de Voronoi para objetos pontuais é o chamado *algoritmo incremental*[22]. Ele começa com um diagrama de Voronoi calculado para dois dos pontos desejados (a mediatriz entre eles). A cada passo, um novo ponto é adicionado, modificando assim o diagrama anterior. Para $p = 1, 2, \ldots, n$, diz-se que V_p denota o diagrama de Voronoi para os primeiros p pontos $s_1, s_2, \ldots, s_p$. A parte mais importante do algoritmo incremental é transformar V_{p-1} em V_p para um p qualquer.

A idéia básica do algoritmo incremental é, supondo já montado o diagrama V_{p-1}, adicionar um novo ponto s_p e obter o novo diagrama V_p. O primeiro passo é encontrar um ponto, chamado aqui de s_i, no qual a sua região de Voronoi contém o novo ponto s_p.

Depois, desenha-se a mediatriz entre s_p e s_i, denotada por $M(s_p, s_i)$. Essa mediatriz corta a fronteira da região de s_i em dois pontos, aqui chamados de x_1 e x_2. O segmento de reta x_1x_2 divide a região de Voronoi de s_i em duas partes, uma delas agora pertencendo a região de s_p.

Começando com a aresta x_1x_2, expande-se a fronteira da região de s_p pelo procedimento descrito a seguir. $M(s_p, s_i)$ cruza a fronteira da região de Voronoi de s_i em x_2, entrando em outra região adjacente, pertencente a um ponto s_j. Então, agora desenha-se $M(s_p, s_j)$ e encontra-se o ponto x_3 onde a mediatriz corta a fronteira de s_j. Repete-se esse procedimento para todas as regiões vizinhas até encontrar novamente o ponto x_1. Então, a seqüência formada por $(x_1x_2, x_2x_3, \ldots, x_{m-1}x_m, x_mx_1)$ é a fronteira da região de Voronoi para o novo ponto s_p.

Finalmente, adicionando a seqüência de segmentos gerada acima em V_{p-1} e excluindo os pedaços de aresta que invadem a nova região de s_p, obtém-se V_p. Repetindo para os n pontos desejados, determina-se completamente o diagrama de Voronoi.

2.4 Algoritmo Algébrico

Na prática, nem sempre podemos considerar os obstáculos como objetos pontuais. Então se faz necessário um algoritmo que consiga calcular o diagrama de Voronoi para objetos não pontuais. O primeiro algoritmo para tratar problemas com objetos convexos apresentado nesse trabalho será o chamado algoritmo algébrico.

O algoritmo algébrico para cálculo dos diagramas de Voronoi no plano consiste na seleção das retas e curvas que compõem o diagrama. Como entrada do algoritmo é dado um mapa com a posição de cada um dos obstáculos. Cada obstáculo no espaço, por hipótese, ou é ou pode ser decomposto em poliedros convexos. As projeções desses poliedros

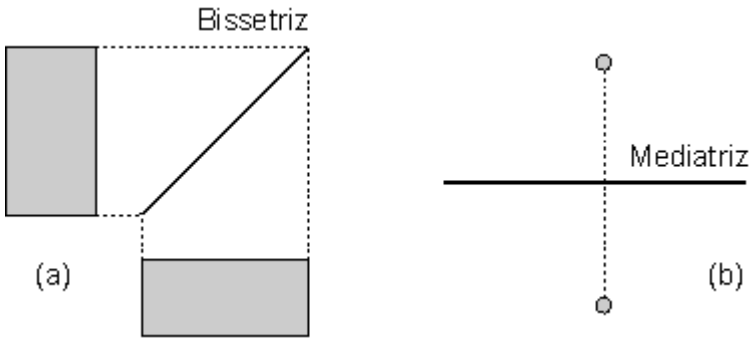

Figura 2.3: Lugares do plano com pontos eqüidistantes

no plano de movimentação do robô são polígonos convexos. As retas e as curvas citadas acima são geradas em função dos vértices e arestas desses polígonos. Quando acontece uma combinação entre duas arestas, o lugar do plano cujos pontos são eqüidistantes a elas é a reta bissetriz do ângulo formado pelas duas, como pode ser visto na figura 2.3a. Agora, quando são dois vértices, o lugar do plano cujos pontos são eqüidistantes a eles é a reta mediatriz, como na figura 2.3b. Já para o caso entre uma aresta e um vértice, temos uma parábola. Essa parte do algoritmo é chamada de *fase de geração*.

Calculadas todas as retas e parábolas possíveis, o algoritmo entra na chamada *fase de seleção*. Nessa fase, são escolhidas as retas e as parábolas pertencentes ao diagrama. Após terminada essa fase, o algoritmo retorna o conjunto de equações que compõem o diagrama.

A maior vantagem desse método é sua precisão. Como temos as equações de todas as retas e parábolas que compõem o diagrama, é possível calculá-lo exatamente. A desvantagem desse método aparece quando o ambiente de trabalho do robô possui muitos obstáculos, pois tanto a fase de geração quanto a fase de seleção exigem muitos recursos computacionais.

2.5 Algoritmo Numérico em 2D

Outra forma de calcular diagramas de Voronoi em 2D para objetos não pontuais foi apresentada em [1] e é aqui chamado de algoritmo numérico. Nesse algoritmo, o plano é dividido na forma de um grid retangular. O tamanho de cada célula do grid pode ser tão pequeno quanto se queira.

Para determinar se uma célula do grid pertence ou não ao diagrama de Voronoi, verifica-se qual objeto está mais próximo de cada um dos seus vértices. Cada vértice ganha um rótulo com o identificador do objeto mais próximo a ele.

Se todos os vértices da célula estão mais próximos de um mesmo objeto, ou seja, com rótulos iguais, ela não pertence ao diagrama. Se os vértices da célula estiverem mais próximos de dois objetos diferentes, sabe-se que uma aresta de Voronoi cruza essa célula. Se estiverem mais próximos de três ou mais objetos, a célula contém um vértice de Voronoi.

Na figura 2.4, pode ser visto o caso onde uma aresta do diagrama corta uma célula do grid. Três dos quatro vértices da célula estão mais próximos do obstáculo 1 e, o quarto, mais próximo do obstáculo 2. Logo, essa célula pertence ao diagrama.

Sabe-se que a aresta do diagrama corta uma determinada aresta da face quando esta possui dois rótulos de vértices diferentes. A existência de dois rótulos diferentes numa aresta implica que existe um ponto eqüidistante aos dois obstáculos indicados pelos rótulos nessa aresta. E, se existe um ponto eqüidistante a dois obstáculos, esse ponto pertence ao diagrama.

Ao invés de calcular os objetos mais próximos de todos os vértices de todas as células do grid, o algoritmo utiliza duas idéias para diminuir o tempo de cálculo do diagrama.

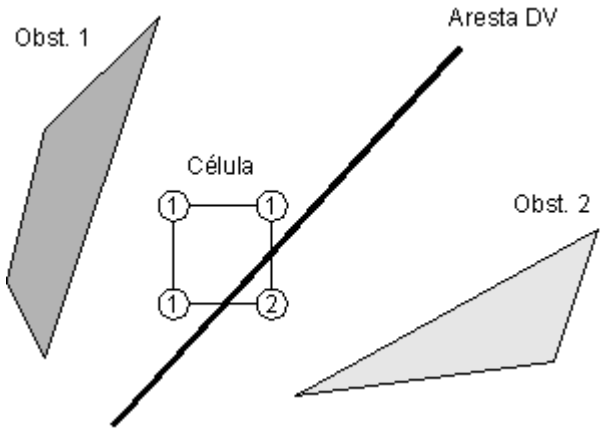

Figura 2.4: Aresta do diagrama passando por uma célula do grid

A primeira idéia se vale de uma propriedade dos diagramas de Voronoi: eles são sempre conexos. Então basta o algoritmo encontrar uma célula do diagrama e depois se manter em cima delas, como se seguisse um caminho. Ou seja, quando uma célula que pertence ao diagrama é encontrada, verificam-se todas as células vizinhas buscando por células do diagrama. Então, o procedimento é repetido para as novas células encontradas.

A segunda idéia advém de um problema criado pela idéia anterior: como achar eficientemente uma célula pertencente ao diagrama. A solução para esse problema é simples: o algoritmo trata os limites do plano de trabalho como objetos. Sendo assim, as células dos cantos do grid são pertencentes ao diagrama, pois estão eqüidistantes aos limites do plano. Basta pegar uma dessas células e aplicar o procedimento da primeira idéia para obter o diagrama de forma muito rápida e eficiente.

O diagrama pode ser calculado de forma bastante rápida e o algoritmo pode ser paralelizado sem muitos "overheads". Supondo um grid representado por uma matriz quadrada com n de lado, o espaço de memória necessário para a execução do algoritmo é $O(n^2)$, devido à necessidade de armazenar o grid. O tempo de cálculo do diagrama aumenta com a diminuição do tamanho das células, com o aumento do número de obstáculos e com o

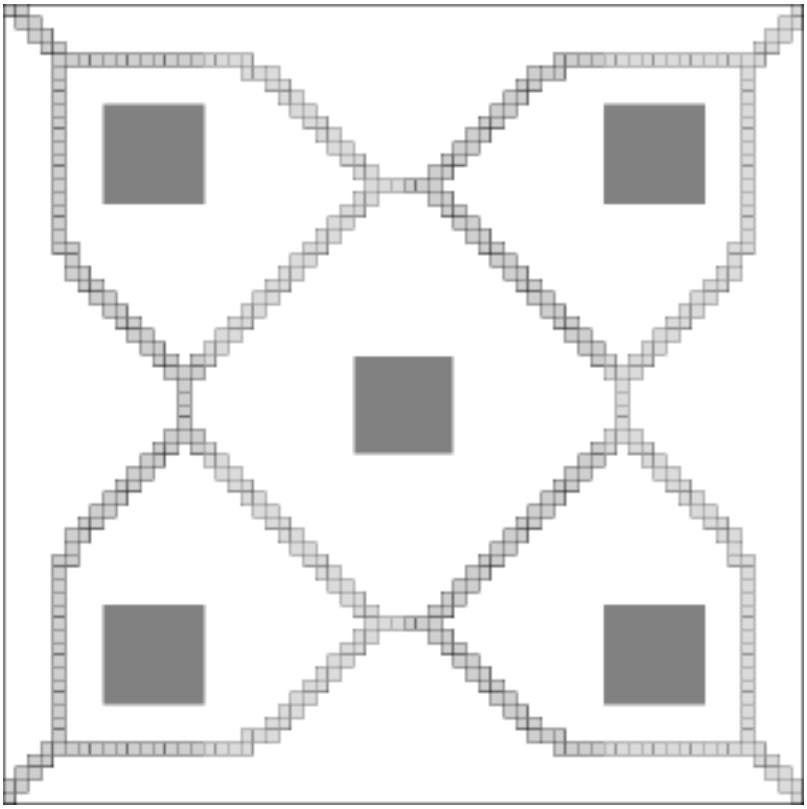

Figura 2.5: Diagrama de Voronoi calculado com baixa precisão

aumento do número de células percorridas.

O tamanho das células do grid deve ser escolhido baseado na aplicação e nos requisitos temporais. Como o algoritmo não calcula exatamente o diagrama, mas sim uma aproximação dele, quanto maior a célula, maior o grau de incerteza da exata localização do diagrama. Na figura 2.5, é possível ver um diagrama de Voronoi calculado pelo algoritmo numérico com células de tamanho relativamente grande, ou seja, com baixa precisão.

O número de objetos impacta no tempo de execução do algoritmo pois é preciso calcular qual o objeto está mais próximo de cada vértice de cada célula. O comprimento do diagrama também aumenta o tempo de cálculo pois quanto maior o comprimento, mais células devem ser processadas.

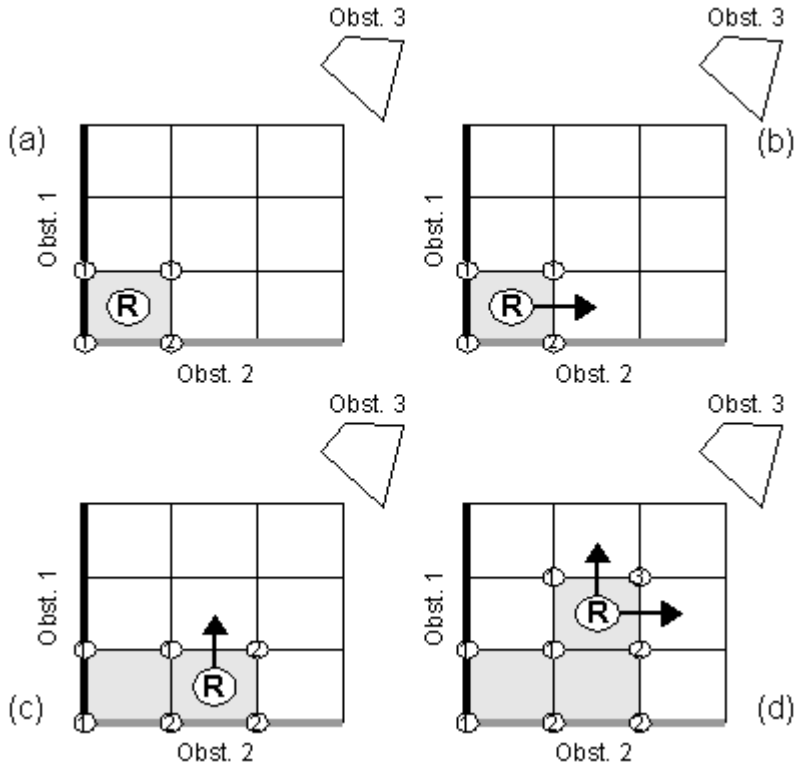

Figura 2.6: Algoritmo em funcionamento

2.5.1 Pseudo-código do Algoritmo

Na figura 2.6 é exemplificado o funcionamento do algoritmo. O círculo com a letra R mostra a célula atual do algoritmo, simbolizando o robô. Em 2.6a, a célula do canto inferior esquerdo do grid é escolhida como célula inicial do algoritmo.

Em 2.6b, o algoritmo decide para onde ir baseado no rótulo de cada um dos vértices da célula. Se os vértices de uma face possuem rótulos diferentes, o diagrama cruza essa face e a célula vizinha a atual também pertence ao diagrama.

Em 2.6c, o algoritmo decide que a próxima célula a ser visitada é a de cima da atual. Novamente, a escolha se dá pela diferença dos obstáculos mais próximos a cada um dos vértices da aresta. A célula a esquerda da atual não é selecionada pois já pertence ao

diagrama.

Finalmente, em 2.6d, pode-se ver que os vértices da célula estão mais próximos de três obstáculos diferentes. Nas células que possuem três ou quatro obstáculos diferentes mais próximos de seus vértices há, dentro de sua área, um vértice de Voronoi. Logo, nessa célula há uma bifurcação do caminho, como pode ser visto pela escolha das células superior e a direita da atual.

O algoritmo implementado que realiza os passos descritos pela figura 2.6 utiliza o procedimento mostrado na figura 2.7 para encontrar o diagrama de Voronoi.

O procedimento começa escolhendo uma célula do canto do grid e a insere na fila. Então, enquanto a fila não está vazia, o algoritmo remove células da fila. Para cada célula removida, é realizado um processo para determinar quais células vizinhas serão postas na fila. Esse processo é baseado no exame do par de rótulos presentes em cada face.

Todas as células selecionadas nesse processo são marcadas como pertencentes ao diagrama. Depois disso, as células selecionadas são inseridas na fila. O procedimento é repetido até não restarem mais células na fila. Nesse momento, o diagrama de Voronoi está completamente determinado.

Esse procedimento implementado pelo algoritmo descrito seria suficiente para encontrar o diagrama de Voronoi em problemas de planejamento de trajetória com os robôs reduzidos a objetos pontuais. Para que esse algoritmo funcione para robôs com formas arbitrárias, é necessário introduzir uma pequena modificação. Considere que a distância do centro do robô até o seu ponto mais externo é r, como visto na figura 2.8.

Quando o algoritmo encontra uma célula que deveria pertencer ao diagrama de Voronoi, ele deve realizar um teste de proximidade. Esse teste verifica se a distância de todos os vértices da célula aos seus obstáculos mais próximos são maiores que r. Se essa

```
calc_diagrama()
{
    criar uma fila para armazenar temporariamente
       as células de Voronoi;
    marcar todas as células dentro da área de trabalho
       como disponíveis e as células externas e a
       volta da área de trabalho como bloqueadas;
    escolher uma das células do canto do grid como
          o ponto de início do Diagrama de Voronoi (DV);
    marcar essa célula como pertencente ao DV;
    inserir essa célula na fila;
    while (fila não está vazia)
    {
       remover uma célula da fila;
       for (cada aresta cujos vértices estão mais
          próximos de obstáculos diferentes)
          if (não é a aresta de entrada e a célula
             vizinha a aresta não pertence ao DV)
             inserir essa célula vizinha na fila,
                preencher quais obstáculos estão mais
                próximos de cada vértice e marcar essa
                célula como pertencente ao DV;
    }
}
```

Figura 2.7: Algoritmo para o cálculo do Diagrama de Voronoi

condição for verdadeira, a célula é marcada como pertencente ao diagrama e inserida na fila. Senão, a célula é marcada como *falsa* mas, mesmo assim, é inserida na fila. Isso se deve ao fato de que suas células vizinhas podem pertencer ao diagrama e, sendo assim, estabelecerem caminhos viáveis.

Para lidar com obstáculos representados como polígonos não convexos, o algoritmo requer que eles sejam decompostos em polígonos convexos. Quando isso acontece, o algoritmo encontra entre as arestas do diagrama de Voronoi as que dividem os polígonos gerados por essa decomposição. Na realidade, essas arestas não existem. Para o correto

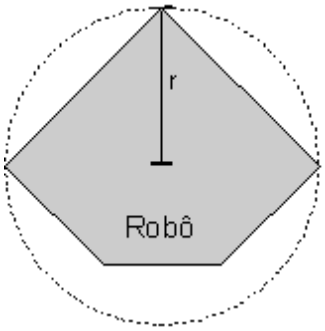

Figura 2.8: Raio de um robô com forma arbitrária

funcionamento do algoritmo, quando há esse tipo de obstáculo, é necessária uma pequena modificação: todas as células do grid que estão sobre um obstáculo qualquer são marcadas como *bloqueadas*. Quando o algoritmo encontra uma célula bloqueada, ele a descarta não inserindo na fila, evitando assim essas arestas inexistentes.

Para o problema de planejamento de trajetórias, é importante determinar o erro do algoritmo. No caso do algoritmo numérico, o erro ao definir o diagrama de Voronoi é menor que o tamanho da diagonal de uma célula do grid. Se, por exemplo, a área de trabalho do robô for um quadrado de $10m$ x $10m$ e a estrutura do grid for definida como uma matriz 1000 x 1000 células, o erro máximo do algoritmo será de $1.41cm$, pois a aresta da célula mede $1cm$.

2.6 Diagramas de Voronoi em n Dimensões

Já foi dito na seção 2.1 que um diagrama de Voronoi em duas dimensões particiona o plano em regiões centradas em objetos de tal maneira que todos os pontos dessa região estejam mais próximos dele do que de qualquer outro objeto. Vendo de outra forma, cada ponto de uma aresta do diagrama está eqüidistante a dois objetos e cada vértice de Voronoi eqüidistante a três ou mais objetos.

Para o caso n-dimensional [10], pode-se generalizar esse conceito. Um ponto de uma aresta está eqüidistante a exatamente n objetos. Já os vértices de Voronoi, eqüidistantes a $n+1$ ou mais objetos.

Então, uma região de Voronoi para um determinado objeto p_i, denotada por $V(p_i)$, é definida em um espaço n-dimensional por:

$$V(p_i) = \{x | dist(p_i, x) \leq dist(p_j, x), \forall j \neq i\}$$

Idêntica a definição no caso bidimensional, somente sendo necessário modificar a função da distância para:

$$dist(p, q) = \sqrt{(p_{x_1} - q_{x_1})^2 + (p_{x_2} - q_{x_2})^2 + \ldots + (p_{x_n} - q_{x_n})^2}$$

Neste trabalho, são estudados algoritmos para o cálculo do diagrama de Voronoi em 3 dimensões.

2.7 Algoritmo Numérico em 3D

Para o cálculo do diagrama de Voronoi em três dimensões foi desenvolvido um algoritmo similar ao exposto na seção 2.5. Nesse algoritmo a área de trabalho é um cubo. Este cubo é dividido em células cúbicas. O algoritmo retorna o conjunto mínimo de células que contém completamente o diagrama de Voronoi.

O diagrama é calculado em relação a objetos, que nesse caso são os obstáculos a serem evitados pelo robô, presentes na área de trabalho. Esses obstáculos devem ser envolvidos pela esfera com menor diâmetro possível. As faces do cubo que definem a área de trabalho também são consideradas obstáculos.

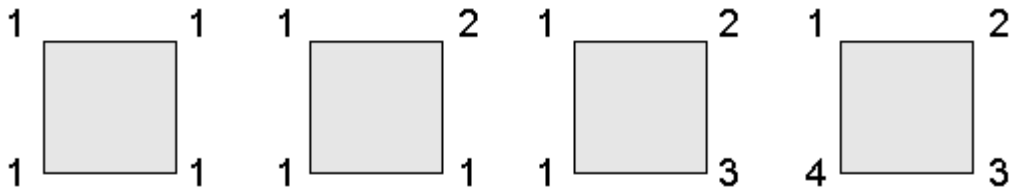

Figura 2.9: Rótulos dos vértices das faces

No cálculo, é preciso determinar quando uma célula pertence ou não ao diagrama. Para isso faz-se uma análise de cada uma das suas faces. Primeiro, determina-se qual obstáculo está mais próximo de cada um dos vértices da face. Cada vértice é rotulado com o identificador do obstáculo mais próximo dele. Como cada face só tem quatro vértices, no máximo pode-se ter quatro identificadores diferentes em cada face.

Na figura 2.9, pode-se ver quatro faces. A primeira, da esquerda para a direita, possui todos os vértices rotulados com o identificador 1. Então esta face está mais próxima de um só obstáculo. Na segunda, três vértices estão mais próximos do obstáculo 1 e um vértice do 2. Então ela está mais próxima de dois obstáculos. A terceira e a quarta estão mais próximas de, respectivamente, três e quatro obstáculos.

Sabendo quantos obstáculos estão mais próximos de cada face, pode-se determinar se o diagrama de Voronoi passa pela face ou não. Já foi visto na seção 2.6 que um ponto que compõe uma aresta de um diagrama de Voronoi em N dimensões é eqüidistante a N obstáculos e um vértice a $N + 1$ ou mais. Então, no caso tridimensional, um ponto de uma aresta está eqüidistante a três obstáculos e um vértice a quatro ou mais. Logo, se uma face está mais próxima de três ou mais obstáculos diferentes, ela é cruzada por uma aresta do diagrama.

Na figura 2.10, pode-se ver duas células da área de trabalho e três obstáculos. Cada um dos três obstáculos tem nele seu próprio identificador. Junto a cada vértice das duas

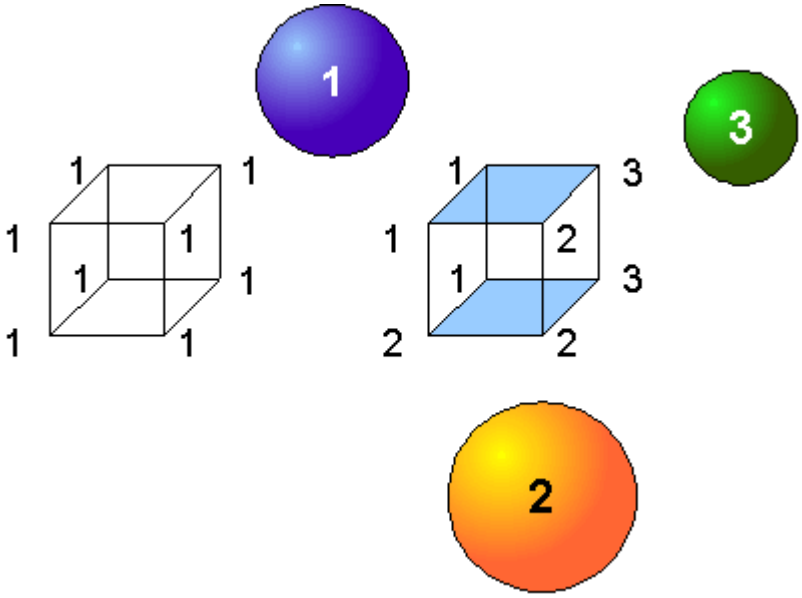

Figura 2.10: Exemplo de células em uma área de trabalho

células, podem ser vistos seus rótulos, que indicam qual obstáculo está mais próximo dele. Na célula da esquerda, todos os vértices estão mais próximos do obstáculo 1. Como todas as faces dessa célula estão mais próximas de um mesmo obstáculo, ela não é cruzada por nenhuma aresta do diagrama. Logo esta célula não faz parte do diagrama.

A célula da direita da figura 2.10 tem duas faces, a de cima e a de baixo, com cor diferente das demais. Elas têm rótulos de três obstáculos diferentes em seus vértices. Então, uma aresta atravessa a célula verticalmente, ou seja, essa célula pertence ao diagrama.

É interessante notar que as células que possuem duas faces cruzadas pelo diagrama de Voronoi são células que fazem parte de uma aresta. Se uma célula possui três ou mais faces cruzadas por arestas de Voronoi, ela contém um vértice do diagrama.

Conhecendo o método para descobrir se uma célula pertence ou não ao diagrama, precisa-se de uma forma eficiente de calculá-lo. Para isso, utiliza-se a propriedade do diagrama de Voronoi ser sempre conexo. Com isso, achando uma célula do diagrama,

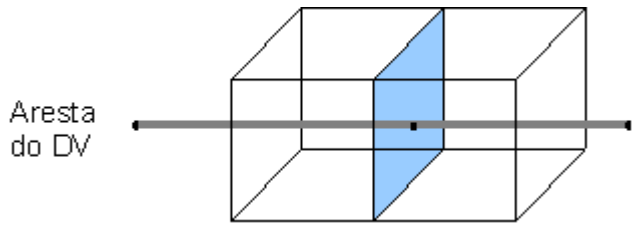

Figura 2.11: Aresta de Voronoi cortando células vizinhas

pode-se achar todas as outras seguindo o caminho indicado por cada face atravessada por uma aresta. Se uma face da célula atual é atravessada por uma aresta, a célula vizinha com essa face em comum também faz parte do diagrama. Na figura 2.11 pode-se ver uma aresta de Voronoi atravessando duas células vizinhas. A face de cor mais escura comum entre as duas células onde passa o diagrama indica o caminho a ser seguido.

O problema agora se torna encontrar, de forma eficiente, uma célula de Voronoi para que o algoritmo possa se aproveitar da propriedade do diagrama ser conexo. Sabendo que a área de trabalho do algoritmo possui a forma de um cubo e, como já dito, sendo consideradas suas faces obstáculos, as células de todos os oito cantos do cubo farão parte do diagrama. Assim, o algoritmo pode escolher uma dessas células para utilizar como ponto de partida.

Tendo a célula inicial, o algoritmo começa um processo chamado *expansão* da célula. Esse processo retorna todas as células vizinhas a atual que são atravessadas pelo diagrama. Após expandida a célula inicial, as células retornadas são, por sua vez, expandidas. E assim por diante até que o diagrama esteja totalmente determinado.

Durante o cálculo do diagrama, o algoritmo separa as células em dois conjuntos. No primeiro, chamado `Não_Expandido`, estão as células que ainda não foram expandidas. No segundo, chamado `Expandido`, encontram-se as células que já foram expandidas. Quando o algoritmo terminar, o diagrama de Voronoi será dado pelas células presentes no conjunto

`Expandido`.

Cada um desses conjuntos é uma estrutura de dados que permite operações de inserção (`Insere()`), busca (`Busca()`) e remoção (`Remove()`). A operação de inserção adiciona uma célula ao conjunto; a de busca retorna um valor *booleano* que indica ou não a presença da célula no conjunto; e já a de remoção retira uma célula qualquer do conjunto e a retorna para o programador.

Na seção seguinte será mostrado e discutido o pseudo-código do algoritmo.

2.7.1 Pseudo-código do Algoritmo

Como dito na seção anterior, o algoritmo pega uma célula de um dos cantos da área de trabalho como ponto de partida. Ela é inserida no conjunto `Não_Expandido`, como pode ser visto na linha 04 da figura 2.12.

O laço principal do algoritmo realiza iterações até que o diagrama de Voronoi esteja completamente determinado, ou seja, o conjunto `Não_Expandido` estar vazio. Nesse ponto, todas as células que compõem o diagrama estarão no conjunto `Expandido`.

Dentro do laço, o algoritmo vai realizando operações de remoção do conjunto `Não_Expandido`, como visto na linha 07. Em cada uma dessas operações, uma célula qualquer é retirada do conjunto e retornada para variável `C`.

Após a célula ser transferida do conjunto `Não_Expandido` para a variável `C`, começa seu processo de expansão, mostrado entre as linhas 08 e 16. Cada uma das faces da célula `C` é testada para verificar se o diagrama a atravessa.

No caso do diagrama atravessar uma face `F` de `C`, será chamada de `K` a célula vizinha a `C` por `F`. Então, sabe-se que `K` pertence ao diagrama de Voronoi. Mas, antes dela ser

```
Voronoi3D(Célula célula_inicial, limite área de trabalho, obstáculos)
{
01:    Conjunto Não_Expandido, Expandido;
02:    Célula   C;
03:
04:    Insere(Não_Expandido, célula_inicial);
05:    while(NOT Vazio(Não_Expandido)) {
06:        // Retira uma celula qualquer de Não_Expandido
07:        C = Remove(Não_Expandido);
08:        for each(face F na célula C) { // Expande a célula C
09:            if(Voronoi atravessa face F) {
10:                Seja K a célula adjacente a C pela face F;
11:                // Verifica se a célula já foi ou vai ser expandida
12:                if(NOT Busca(Não_Expandido, K) AND NOT
13:                    Busca(Expandido, K))
14:                    Insere(Não_Expandido, K);
15:            }
16:        }
17:        Insere(Expandido, C);  // Terminou a expansão de C.
18:    }
19:    return(Expandido);
}
```

Figura 2.12: Pseudo-código do algoritmo numérico para o cálculo diagrama de Voronoi em 3D

inserida no conjunto `Não_Expandido` para sua eventual expansão, é preciso testar se ela já foi ou está para ser expandida, ou seja, se ela está contida em `Não_Expandido` ou `Expandido`. Se `K` não estiver contida em nenhum dos dois conjuntos, ela pode ser inserida em `Não_Expandido`. Estando ela já presente em um dos dois conjuntos, não há necessidade de adicioná-la novamente.

Se o teste das linhas 12 e 13 não fosse realizado antes da inserção da linha 14, o algoritmo jamais iria sair do laço principal, pois adicionaria as mesmas células já expandidas várias vezes, nunca tornando falsa a expressão lógica da linha 05.

Após completada a expansão da célula `C`, ela é inserida no conjunto `Expandido`, como visto na linha 17, e se tornando parte, definitivamente, do diagrama.

Quando o conjunto `Não_Expandido` estiver vazio, o laço principal termina e o diagrama resultante do cálculo se encontra no conjunto `Expandido`.

2.7.2 Comentários sobre o Algoritmo

Como já visto anteriormente, uma célula que é atravessada por uma aresta tem duas faces cruzadas por uma aresta do diagrama de Voronoi. Uma célula que tem três ou mais faces cruzadas por arestas contém um vértice do diagrama.

Quando o algoritmo processa uma célula `C` que faz parte de uma aresta do diagrama, o `if` da linha 09 é verdadeiro somente duas vezes. Nesse caso, o `if` da linha 12 será executado duas vezes e quatro operações de busca serão realizadas nos conjuntos. Mas, das duas vezes que o `if` da linha 12 será executado, no máximo em uma delas ocorrerá a inserção da célula vizinha no conjunto `Não_Expandido`. Isso se deve ao fato de que pelo menos uma das duas células vizinhas a `C` já foi expandida, pois seria impossível `C` aparecer no conjunto `Não_Expandido` se uma de suas células vizinhas não tivesse sido expandida.

Após a expansão de `C`, ela é inserida no conjunto `Expandido`. Então, totalizam-se, por iteração, quatro operações de busca, no máximo duas de inserção e uma de remoção nos dois conjuntos utilizados pelo algoritmo.

Já no caso da célula C conter um vértice, todas as suas seis faces podem ser cruzadas pelo diagrama. Então, o `if` da linha 09 pode ser verdadeiro até seis vezes, implicando em até 12 buscas nos conjuntos. Mas, pelo mesmo raciocínio aplicado quando `C` era célula de aresta, pode-se ver que no máximo cinco inserções serão realizadas durante a expansão de `C`. Adicionando estas à remoção e à inserção no conjunto `Expandido`, têm-se até 12

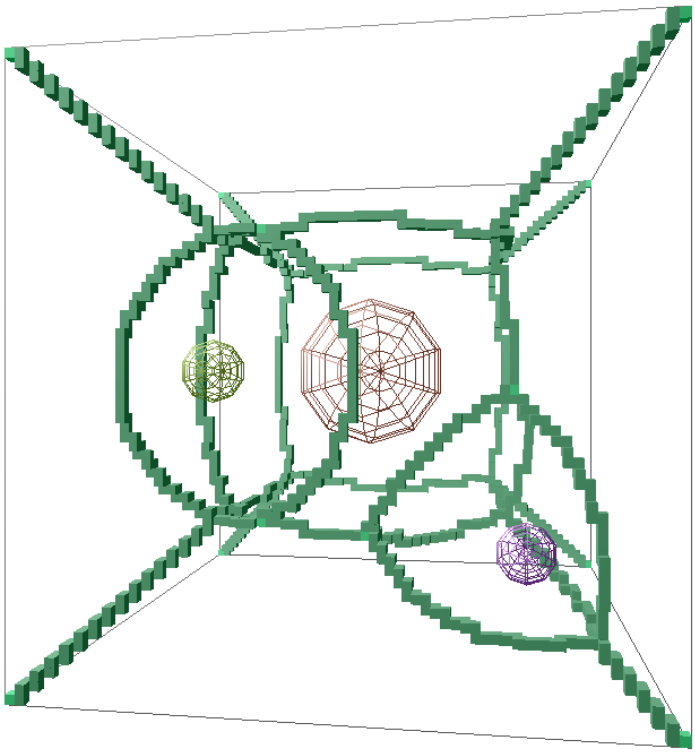

Figura 2.13: Diagrama de Voronoi em 3D calculado com baixa precisão

buscas, seis inserções e uma remoção.

Baseado nestes fatos, estes conjuntos devem ser implementados por uma estrutura de dados que seja rápida e eficiente nas buscas e inserções, principalmente nas buscas, devido a sua freqüência. A operação de remoção tem a flexibilidade de poder retornar qualquer célula do conjunto `Não_Expandido`, e não uma célula específica.

Outra consideração importante é o tamanho das células do grid. Ele deve ser escolhido baseado na aplicação, nos requisitos temporais e na precisão desejada. Como o algoritmo não calcula exatamente o diagrama, quanto maior a célula, maior o grau de incerteza da exata localização do diagrama.

Na figura 2.13, é possível ver um diagrama de Voronoi calculado pelo algoritmo nu-

mérico em 3D com células com baixa precisão. Os obstáculos são as três esferas presentes na área de trabalho.

O número de obstáculos impacta no tempo de execução do algoritmo pois é preciso calcular qual o objeto está mais próximo de cada vértice de cada célula. O comprimento do diagrama também aumenta o tempo de cálculo pois quanto maior o comprimento, mais células devem ser processadas.

3 Estruturas de Dados

No algoritmo numérico 2D a estrutura de dados utilizada no cálculo do diagrama de Voronoi era uma matriz. No caso do algoritmo numérico 3D a utilização de matrizes iria gerar um consumo muito grande de memória. Como somente uma pequena parte da matriz é utilizada para representar o diagrama, pode-se utilizar técnicas de matrizes esparsas para resolver esse problema. Esse tipo de matriz, junto com um dos seus principais métodos de representação, o *Compressed Row Storage*, é mostrado na seção 3.1.

Devido ao grande número de buscas, inserções e remoções de células nos conjuntos `Expandido` e `Não_Expandido`, foi utilizada na implementação, tanto do algoritmo numérico 3D serial quanto do paralelo, uma estrutura aqui chamada de Hash+Bloom. Essa estrutura apresentou desempenho superior a *Compressed Row Storage* para o problema do cálculo do diagrama de Voronoi. Para entendê-la melhor, são revisados conceitos de Tabelas Hash, na seção 3.2, e de Filtros de Bloom, na seção 3.3, antes de estuda-la na seção 3.4.

3.1 Matrizes Esparsas

O cálculo do diagrama de Voronoi em 3D, tanto serial quanto paralelo, requer uma matriz cúbica com aresta N. Mas, para casos típicos de N, tais como $N = 512$, $N = 1024$ ou $N = 2048$ e ocupando apenas 1 byte para o dado de um único elemento da matriz,

a quantidade de memória necessária para a solução seria muito grande. No caso de $N = 1024$, seria necessário 1 gigabyte de memória RAM somente para a matriz.

Como o resultado do cálculo de um diagrama de Voronoi normalmente ocupa menos de 1% dos elementos da matriz, a maior parte dela fica ocupada com valores sem importância. Tendo isso em vista, pode-se representá-la utilizando técnicas aplicadas às matrizes esparsas. Uma matriz é chamada esparsa quando poucos dos seus elementos contém dados relevantes à computação.

Existem várias técnicas para a representação de matrizes esparsas. A mais simples consiste de uma lista encadeada onde cada elemento representa uma posição da matriz. São inseridos na lista somente os elementos de importância para a computação. A estrutura de cada nó que compõe a lista é formada pelas coordenadas do elemento na matriz, seu valor e um ponteiro para o próximo nó.

Uma forma muito comum de se representar matrizes esparsas é a *Compressed Row Storage*, também chamada de CRS, e a sua forma análoga a *Compressed Column Storage*, ou CCS.

3.1.1 Compressed Row Storage

Os formatos Compressed Row Storage e Compressed Column Storage[23] são bastante genéricos pois não fazem nenhuma suposição quanto à estrutura esparsa da matriz (se é diagonal, etc.).

O formato CRS coloca os valores diferentes de zero das linhas matriz em posições contíguas de memória.

Seja uma matriz esparsa A mostrada abaixo, criam-se três vetores: um para os valores (`val`), um para o índice das colunas (`col_ind`) e um para os ponteiros das linhas

(`row_ptr`).

$$A = \begin{pmatrix} 7 & 0 & 0 & 6 \\ 2 & 4 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{pmatrix}$$

O vetor `val` guarda os valores diferentes de zero presentes na matriz A como se eles fossem lidos linha a linha. No exemplo da matriz A, o vetor `val` é preenchido com os valores vistos na tabela 3.1. Nessa tabela, também podem ser vistos os valores do vetor `col_ind`. Este vetor armazena os índices das colunas onde estão os elementos não zero da matriz. Então, se `val(k)` $= a_{i,j}$ tem-se que `col_ind(k)` $= j$.

Já o vetor `row_ptr` armazena o índice das posições do vetor `val` onde uma linha começa. Ou seja, se `val(k)` $= a_{i,j}$ então `row_ptr(i)` $\leq k <$ `row_ptr(i+1)`. Para o exemplo da matriz A, os valores do vetor podem ser vistos na tabela 3.2.

Para encontrar o valor do elemento da posição $(1, 3)$ da matriz A, começa-se olhando na posição 1 de `row_ptr`, onde o valor encontrado é 2. Então, sabe-se que o elemento desejado está entre a posição 2 e 4 de `val` (`row_ptr(1)` $= 2$, `row_ptr(2)` $= 5$, lembrando-se que `row_ptr(i)` $= 2 \leq k <$ `row_ptr(i+1)` $= 5$). Agora, basta percorrer as posições 2, 3, 4 do vetor `col_ind` até encontrar 3, o valor desejado. No caso do exemplo, o índice encontrado é 4. Finalmente, encontra-se $A_{1,3} = 9$ na posição 4 do vetor `val`.

`val`	7	6	2	4	9	5
`col_ind`	0	3	0	1	3	3

Tabela 3.1: Valores dos vetores val e col_ind no formato CRS.

O formato CCS tem o mesmo princípio de funcionamento do CRS, diferindo apenas no funcionamento dos vetores `row_ptr` e `col_ind`. São criados vetores com funções análogas,

row_ptr	0	2	5

Tabela 3.2: Valores para row_ptr no formato CRS

o vetor equivalente ao `row_ptr` agora armazena o ponteiro para o início das colunas em `val`. E o análogo ao vetor `col_ind` ao invés de armazenar o índice da coluna da matriz, armazena o índice da linha.

3.2 Tabelas Hash

Uma das operações mais comuns realizadas em tarefas computacionais é a operação de *busca*. Nela, uma informação em particular é procurada entre uma grande quantidade de dados. Os dados estão estruturados na forma de *registros*. Cada registro contém uma *chave* e a informação propriamente dita. O objetivo de uma busca é, dada uma chave, encontrar todos os registros cujas chaves sejam iguais a chave dada.

Um registro de um cadastro de contas bancárias pode ter como chave, por exemplo, o CPF ou o número da conta, além de informações não discriminatórias, tais como nome do titular, data de nascimento, etc.

A estrutura chamada Hash é uma forma de armazenar os registros em um tipo de tabela diminuindo o espaço de busca. Quando se deseja acessar um registro específico dessa tabela, aplica-se uma função na chave desejada. O resultado dessa função é um endereço da tabela.

O objetivo da tabela Hash é conseguir buscar uma chave em tempo $O(1)$, ou seja, a transformação daria exatamente o endereço desejado. Na prática porem, esse tempo pode levar até $O(n)$ no pior caso. Apesar disso, a tabela Hash se mostra extremamente prática e eficiente.

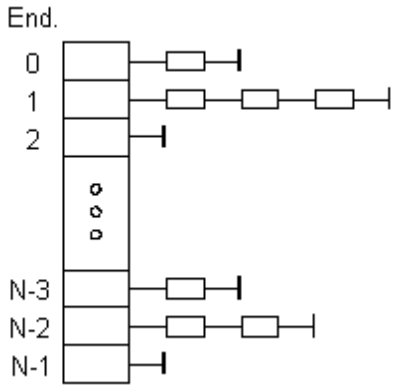

Figura 3.1: Exemplo de Hash com resolução de colisão

A primeira parte da estrutura Hash é uma função, chamada *função de Hash*, que realiza a transformação da chave em um endereço da tabela. Como nenhuma função de Hash é injetora, duas ou mais chaves diferentes podem ser colocadas em um mesmo endereço da tabela, ocorrendo assim uma colisão. A segunda parte da estrutura é a *resolução de colisão*, que lida com os casos em que chaves diferentes são mapeadas para um mesmo endereço da tabela.

Um exemplo de função de Hash para chaves compostas por números inteiros é a função módulo (%). Para uma tabela com N posições, determina-se um endereço e de uma chave c realizando-se a seguinte operação $e = c\%N$. Essa função é muito utilizada na prática pois retorna valores sempre dentro do espaço de endereçamento da tabela.

Uma possível forma de resolver o problema da resolução de colisão é colocar em cada posição da tabela uma lista encadeada. Quando uma chave é mapeada para uma posição da tabela, ela começa a encabeçar a lista. Cada chave seguinte mapeada para o mesmo endereço é posta no final da lista.

Um exemplo de Hash com resolução de colisão onde uma tabela com N endereços, cada um com sua lista encadeada, é representada de forma gráfica na figura 3.1. Os

endereços 2 e o $N - 1$ estão associados a listas vazias, pois estes endereços não foram gerados pela função para nenhuma chave. Os endereços onde aconteceram colisões são o 1 e $N - 2$, neste caso, o mesmo endereço foi gerado por diversas chaves.

No presente trabalho utilizou-se a mesma função de Hash exemplificada acima. Houve somente uma modificação no método de resolução de colisões. Os elementos são inseridos no início da lista, e não no final. No caso do armazenamento em listas simplesmente encadeadas, as inserções são efetuadas de forma imediata nos seus extremos. Contudo, a inserção no fim da lista requer um ponteiro adicional para o último elemento. Desta forma, a resolução de possíveis colisões foi implementada armazenando-se as novas chaves no início das listas para evitar a manutenção desse ponteiro.

3.3 Filtro de Bloom

O filtro de Bloom [24, 25] é uma estrutura de dados probabilística que é usada para testar se um elemento está contido ou não em um conjunto. É possível que ocorra um resultado *falso positivo*, ou seja, que o filtro diga que o elemento está contido quando na verdade não está. No entanto, resultados *falso negativos* são impossíveis.

Um filtro de Bloom é formado por um vetor de m bits, todos inicialmente zerados, e por k funções de Hash associadas. Quando um elemento é adicionado ao conjunto, sua chave é submetidas as k funções, que retornam k posições do vetor onde seus valores devem ser postos em 1.

Para recuperar um elemento, ou seja, testar se ele se encontra em um conjunto, obtém-se k posições do vetor com as funções de Hash. Se qualquer bit dessas k posições estiver com valor 0, o elemento não pertence ao conjunto. Se ele pertencesse, todos os k bits teriam sido colocados em 1 quando o elemento foi inserido. Se todos os bits estão em

1, ou o elemento está no conjunto, ou os bits mudaram para 1 quando outros elementos foram inseridos.

Infelizmente, remover um elemento desse filtro de Bloom simples é impossível. O elemento mapeia para k bits e, mesmo que colocando todos eles em 0 realmente o removeria do filtro, isso também remove outros elementos que mapeiam para qualquer um desses bits. O resultado disso seria o acontecimento de resultados falso negativos, o que não é permitido.

3.3.1 Probabilidade de Falsos Positivos

Assuma que uma função de Hash selecione cada posição do vetor com igual probabilidade. A probabilidade que um certo bit não foi posto com valor 1 por uma dessas funções, quando o elemento foi inserido, será dada por $1 - \frac{1}{m}$. A probabilidade de que não foi colocado em 1 por nenhuma das k funções será:

$$\left(1 - \frac{1}{m}\right)^k$$

Se foram inseridos n elementos, a probabilidade de que um bit ainda esteja em 0 será dada por:

$$\left(1 - \frac{1}{m}\right)^{nk}$$

Então, a probabilidade que seja 1 é:

$$1 - \left(1 - \frac{1}{m}\right)^{nk}$$

Agora, testa-se a presença de um elemento que não pertence ao conjunto. Cada uma

das k posições do vetor computadas pelas funções Hash está em 1 com a probabilidade acima. A probabilidade de todas as posições estarem em 1, o que causaria um resultado falso positivo, é:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k$$

Pode-se observar que a probabilidade de falsos positivos cái com aumento do valor de m (o número de bits no vetor) e aumenta com o aumento de n (o número de elementos inseridos). Para um dado valor de m e n, o valor de k que minimiza essa probabilidade é:

$$k \approx \frac{m}{n} \ln 2 \approx 0.7 \frac{m}{n}$$

Resultando assim em uma probabilidade de:

$$\left(\frac{1}{2^{\ln 2}}\right)^{m/n} \approx 0.62^{m/n}$$

3.4 Estrutura Hash+Bloom

3.4.1 Motivação

O algoritmo numérico para o cálculo do diagrama de Voronoi em 3D visto na seção 2.7 requer que sua área de trabalho seja representada na forma de uma matriz cúbica com aresta N. Assumindo uma implementação onde cada elemento necessite de um byte para indicar se pertence ou não ao diagrama, seriam alocados, para uma matriz com $N = 1024$, 1Gb de memória. Torna-se necessário então, uma estrutura capaz de representar essa matriz na memória sem ocupar tanto espaço.

Se for utilizado o valor 1 para indicar que um elemento ou célula da matriz pertence

ao diagrama e zero para indicar o contrário, tem-se uma matriz com muito mais valores zero do que 1. Como visto na seção 3.1, isto é uma característica de problemas de matrizes esparsas. Aplicando-se técnicas para representação desse tipo de matriz na estrutura do problema, pode-se diminuir a quantidade de memória ocupada.

A estrutura a ser utilizada, além de economizar memória, tem que apresentar bom desempenho nas operações freqüentemente utilizadas pelo algoritmo. Os três tipos de operação realizadas no caso do algoritmo numérico 3D são as de busca, inserção e remoção.

Na busca, um elemento da matriz é especificado pelas suas coordenadas x, y e z. O valor de retorno dessa operação é um booleano representando a existência ou não de um elemento com aqueles valores de x, y, z na estrutura. Na operação de inserção, um elemento, também especificado pelas coordenadas x, y, z, é inserido na estrutura. Já na remoção, o algoritmo requer que um elemento qualquer seja retirado da estrutura e devolvido para posterior utilização.

Agora se torna necessário analisar o algoritmo para saber quantas operações de cada tipo são realizadas por iteração. Como o algoritmo trata de uma célula por iteração, pode-se simplificar essa análise classificando as células que compõem o diagrama em dois tipos: células que contém vértices e células que são parte de arestas. Vale ressaltar que esse último tipo ocorre com muito mais freqüência que o primeiro.

Em uma célula que é parte de uma aresta de Voronoi, o algoritmo realiza 4 operações de busca, 2 de inserção e uma de remoção. Já nas células que contém vértices, são realizadas de 6 a 12 operações de busca, de 3 a 6 inserções e somente uma remoção. Com esses dados, percebe-se a necessidade de otimizar as operações de busca e inserção.

Normalmente, o pior caso de uma operação de busca é a procura por um elemento que não está presente na estrutura. Devido as características do problema do algoritmo

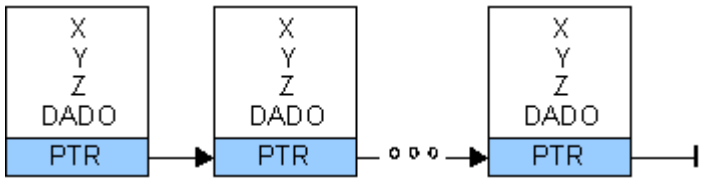

Figura 3.2: Registros na lista encadeada

numérico 3D, as operações de busca retornam com muito mais freqüência o valor *falso*, que indica a inexistência do elemento na estrutura. Então, seria interessante uma estrutura que permitisse que essas operações de busca de elementos inexistentes fosse melhorada.

Então, baseando-se nas características do problema, foi utilizada uma estrutura chamada aqui de *Hash+Bloom*. Esta estrutura, descrita em detalhes nas seções seguintes, une as tabelas Hash com filtro de Bloom, e apresentou-se como uma alternativa interessante na implementação do algoritmo numérico 3D.

3.4.2 Implementação da Estrutura Hash+Bloom

A estrutura Hash+Bloom consiste de uma tabela de Hash onde cada posição da tabela possui, além de um ponteiro para uma lista encadeada com os dados daquela posição, um ou mais filtros de Bloom. Agora será visto cada um desses elementos que compõem a estrutura e como eles estão interligados.

Cada registro, como visto na figura 3.2, contém os valores das coordenadas x, y e z da posição da matriz e uma informação relativa àquela posição, além de um ponteiro com a função de encadear o próximo elemento da lista.

A tabela Hash é construída de forma similar a apresentada na seção 3.2 da página 58. Ela possui M posições, ou endereços, onde os registros podem ser incluídos. A chave utilizada pela função de Hash é a coordenada x do elemento da matriz.

A função de Hash aplicada na chave para gerar o endereço onde o elemento será inserido na tabela é a função módulo. Pode-se utilizar qualquer valor para M mas, com o objetivo de obter um melhor desempenho na implementação, restringiu-se seu valor a potências de 2. Com isso, pode-se substituir a função módulo pelo operador binário 'E' (&). Nos processadores modernos, instruções para o cálculo do módulo levam dezenas de ciclos para completar, enquanto as operações binárias necessitam de apenas um ciclo.

Para calcular o endereço e de uma chave c utilizando o 'E' binário (&) ao invés do módulo, realiza-se a seguinte operação:

$$e = c\&(M - 1)$$

Supondo, como exemplo, que $M = 32$, $c = 47$ e que cada variável ocupa somente 1 byte na memória (8 bits), tem-se que $M - 1$ em binário é "00011111" e c em binário é "00101111". O valor de e será então 15 em decimal ou "00001111" em binário.

Calculado então o endereço, o registro pode ser inserido na sua posição da tabela Hash. Como um mesmo endereço pode ser gerado para várias chaves, podem ocorrer colisões de dois ou mais elementos. Então, ao invés de serem armazenados dados nas posições, são armazenados ponteiros para listas encadeadas. Ou seja, cada posição da tabela tem sua própria lista encadeada, semelhante a mostrada na figura 3.1 da página 59.

Então, para inserir um elemento na estrutura Hash+Bloom basta calcular seu endereço utilizando o método descrito acima e inserí-lo no início da lista encadeada da sua posição na tabela. Como pode ser visto, uma operação muito rápida. Já a operação de remoção requer um pouco mais de esforço. Nela precisa-se percorrer as posições da tabela Hash em busca do primeiro ponteiro não nulo. Quando é encontrado, basta retirar o primeiro elemento daquela lista encadeada.

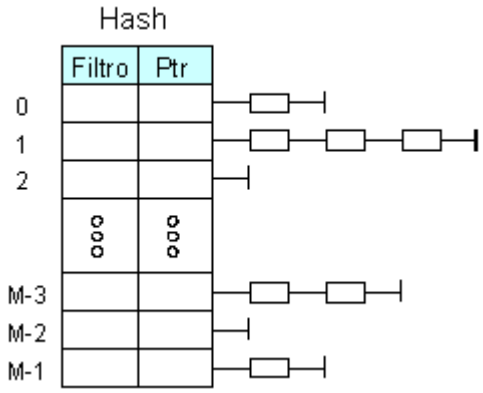

Figura 3.3: Estrutura Hash+Bloom

Com relação a operação de busca, já foi visto na seção anterior que são mais freqüentes os casos onde é retornado o valor *falso*, ou seja, o elemento desejado não é encontrado. Para melhorar o desempenho da operação de busca, foram utilizados filtros de Bloom, já vistos na seção 3.3.

Na figura 3.3, pode-se ver a tabela Hash modificada com o filtro de Bloom. Cada posição da tabela possui seu próprio filtro de Bloom. Quando ocorre uma operação de inserção, o elemento a ser inserido passa pela função do filtro de Bloom, gerando a posição dos bits a serem colocados em 1 no filtro. Depois, o elemento é colocado normalmente na lista encadeada.

O filtro de Bloom não consegue afirmar se um elemento está na lista, mas consegue dizer com certeza que um dado elemento não está presente. Então, para fazer a busca de um elemento, primeiro calcula-se seu endereço da tabela Hash. Depois, o elemento é passado pela função do filtro de Bloom onde obtém-se a posição dos bits a serem examinados. Se algum dos bits estiver em zero, o elemento desejado não se encontra na lista e a operação de busca pode retornar um valor *falso* sem mesmo percorrer a lista encadeada. Se todos os bits estiverem em 1, deve-se percorrer a lista em busca do elemento, lembrando ainda

que o filtro pode ter gerado um valor falso positivo.

O filtro, da forma como foi implementado, é um vetor de 32 bits para cada posição da tabela Hash. A função do filtro recebe como parâmetro a coordenada y do elemento da matriz e gera como resultado o endereço de um bit do filtro. O endereço e é calculado pela função utilizando $e = y\&31$.

Uma outra variação do filtro foi implementada utilizando dois vetores de 32 bits para cada posição da tabela, com uma função para cada um dos dois vetores. Uma das funções recebe como parâmetro a coordenada y do elemento da matriz, já a outra recebe a coordenada z.

Na seção seguinte, serão comparadas três propostas de estruturas aplicadas no problema do algoritmo numérico 3D, sendo elas a Compressed Row Storage (CRS), a tabela Hash e a estrutura Hash+Bloom.

3.4.3 Resultados

Os experimentos realizados consistem do cálculo de um diagrama de Voronoi pelo algoritmo númerico 3D. Para uma mesma configuração de obstáculos, o diagrama é calculado com três precisões diferentes, ou seja, três tamanhos diferentes de matrizes. Como a matriz da área de trabalho do diagrama é cúbica, uma precisão n significa dizer que a matriz possui $n \times n \times n$ células.

O desempenho de três estruturas de dados (CRS, Tabela Hash e Hash+Bloom) serão comparadas exercendo a função dos conjuntos `Expandido` e `Não_Expandido` do algoritmo numérico 3D. A estrutura CRS requer um vetor com um ponteiro para cada linha da matriz. O comprimento desse vetor será chamado de H. Então, para a estrutura CRS, tem-se $H = n$.

Para a utilização da Tabela Hash, foi utilizada uma função de módulo para fazer o mapeamento da coordenada x da célula para um endereço na tabela. O número de posições da tabela também será chamado de H. Então, no caso onde $H = n$, a tabela Hash se comporta como o método CRS. Nas tabelas com os resultados, a linha correspondente ao desempenho da tabela Hash tem o título Hx. Na linha Hx de todas as tabelas, ocorre, na última coluna, o caso onde $H = n$, ou seja, é apresentado o resultado do método CRS.

A estrutura Hash+Bloom utiliza como função de Hash o módulo da coordenada x da célula pelo tamanho H da tabela. Ela é testada utilizando duas variantes do filtro de Bloom. Uma, chamada HxBy, aplica a função de Hash na coordenada x e o filtro de Bloom na coordenada y da célula. A outra, HxByBz, aplica, também, o filtro de Bloom na coordenada z da célula.

Os testes foram realizados no sistema operacional Microsoft Windows XP e os programas compilados utilizando o Visual Studio.NET 2005. A máquina de teste foi um AMD Athlon 64 3200+ funcionando a 2GHz, com 128kB de cache L1, 512kB de cache L2 e com 512MB de memória RAM DDR400. Os tempos apresentados nas tabelas foram normalizados em relação ao tempo de execução do algoritmo utilizando o método CRS.

Método	$H = 128$	$H = 2048$	$H = 4096$
Hx	97.94	98.24	100.00
HxBy	**96.99**	**96.40**	99.26
HxByBz	97.72	99.63	98.16

Tabela 3.3: Resultados para precisão 4096 (em %)

Na tabela 3.3, podem ser vistos os tempos normalizados (em %) obtidos no cálculo do diagrama com precisão de 4096. Os dois melhores tempos são mostrados em negrito. O mesmo acontece para as tabelas 3.4 e 3.5, onde são apresentados, respectivamente, os resultados para a precisão de 8192 e de 16384.

Método	$H = 128$	$H = 4096$	$H = 8192$
Hx	98.83	98.54	100.00
HxBy	**98.03**	98.89	99.40
HxByBz	**97.69**	99.06	98.85

Tabela 3.4: Resultados para precisão 8192 (em %)

Método	$H = 128$	$H = 8192$	$H = 16384$
Hx	98.90	99.47	100.00
HxBy	**98.12**	99.06	99.53
HxByBz	**98.59**	99.37	99.22

Tabela 3.5: Resultados para precisão 16384 (em %)

O ganho apresentado pela estrutura Hash+Bloom foi pequeno, variando entre 1.4 e 3.6%. Mas, devido a pequena complexidade de código adicionada em relação ao método CRS, o método é uma alternativa viável, principalmente em sistemas de tempo-real, onde poucos milisegundos podem fazer a diferença entre o escalonamento ou não de uma tarefa.

4 Computação Paralela

Neste capítulo serão estudados tópicos de *Computação Paralela*, dentre eles as principais arquiteturas de computadores utilizadas e seus modelos de programação. O aumento da importância da computação paralela nos dias de hoje é mostrada na seção 4.1.

Na seção 4.2 as principais arquiteturas paralelas são classificadas e estudadas, tanto do ponto de vista de hardware como de software. Depois, na seção 4.3, é estudada uma das principais bibliotecas utilizadas na criação de softwares paralelos, a biblioteca *MPI*.

Conceitos para a analise de desempenho de softwares paralelos, junto com fatores limitantes para a sua escalabilidade e para seu tempo de execução, são apresentados na seção 4.4.

Finalmente, a *Decomposição Regular de Domínio*, que é uma das principais técnicas utilizadas em aplicações paralelas para o balanceamento de carga entre processadores, é mostrada na seção 4.5.

4.1 Crescimento da Computação Paralela

Computadores cada vez mais rápidos tem estimulado uma nova forma de fazer ciência, possibilitando a comprovação da teoria por meio das simulações computacionais. Cientistas de computação tem simulado em supercomputadores fenômenos tão complexos que

são dificilmente preditos teoricamente e muito perigosos ou caros para serem realizados em laboratório. Repetidas experiências bem sucedidas nessa área da ciência da computação tem causado um grande aumento na demanda de recursos computacionais nos últimos quinze anos.

Aplicações científicas não se caracterizam somente pela grande demanda de poder de processamento. Elas também requerem normalmente uma grande quantidade de memória e, às vezes, um alto desempenho de entrada/saída. Por exemplo, uma simulação do comportamento do clima da Terra ao longo de dez anos usando um modelo de última geração executa 10^{16} operações de ponto flutuante – dez dias rodando em uma máquina com uma velocidade de 10^{10} operações de ponto flutuante por segundo (10 gigaflops). E, além disso, essa mesma simulação pode gerar facilmente mais de 100 gigabytes de dados.

A velocidade dos processadores é limitada por vários fatores, tais como a velocidade da luz e a eficiência da dissipação térmica. Esses fatores motivaram uma grande evolução no mundo dos computadores paralelos: eles passaram de experimentos de laboratórios para ferramentas no dia-a-dia dos cientistas de computação.

Mais recentemente, com o aumento do desempenho dos PCs[1] e com grande queda nos seus preços, além do preço acessível nas redes de alta performance para conectá-los, está sendo possível criar clusters com alto poder computacional. Esses clusters compostos de PCs estão chegando a ponto de competir com supercomputadores de várias empresas tradicionais na área de computação de alto desempenho.

Modelo computacional é uma visão conceitual dos tipos de operações disponíveis para um programa. Não inclui nenhuma sintaxe específica de alguma linguagem de programação e é praticamente independente da camada de hardware logo abaixo. Nas seguintes sub-seções será feita uma pequena revisão de arquiteturas de computadores paralelos e

[1] PC: Personal Computer

de alguns dos muitos modelos computacionais disponíveis na atualidade. Cada um desses modelos pode ser implementado em todos os computadores paralelos modernos, bastando uma pequena ajuda do sistema operacional.

4.2 Arquiteturas de Computadores Paralelos

As arquiteturas utilizadas em computadores paralelos podem ser categorizadas sob dois aspectos: memória e espaço de endereçamento.

No caso da memória a classificação depende dela ser fisicamente distribuída ou centralizada. Quando a memória é distribuída, cada elemento processador tem sua própria memória. Já no caso de ser centralizada, os vários elementos compartilham uma mesma memória.

Quanto ao espaço de endereçamento, ele pode ser compartilhado ou individual. Quando o espaço é individual, cada elemento processador tem, por exemplo, seu próprio endereço zero. Quando é compartilhado, todos os elementos enxergam somente um único endereço zero.

Memória	Endereçamento Compartilhado	Endereçamento Individual
Centralizada	SMP	–
Distribuída	NUMA	MPP

Tabela 4.1: Classificação de arquiteturas paralelas

A tabela 4.1 mostra as relações entre esses atributos para os três principais modelos de máquinas paralelas da atualidade: SMP (*Symmetric Multiprocessor*), NUMA (*Non-Uniform Memory Access*) e MPP (*Massively Parallel Processors*). Um quarto modelo de arquitetura paralela não citado acima é o processamento vetorial ou paralelismo de

dados. As arquiteturas vetoriais foram de grande importância do final dos anos 70 até o fim da década de 90. Na lista *TOP500 Supercomputer Sites*[2], onde são listados os 500 maiores supercomputadores do mundo, em Junho de 1993 os computadores vetorias ocupavam 66,8% das posições do ranking. Na última lista, de Novembro de 2005, ocupam apenas 2,8%, com destaque para o Earth-Simulator da NEC em sétimo lugar. Nas seções seguintes, essas quatro arquiteturas e seus modelos de programação serão explicados em detalhes.

4.2.1 Paralelismo de Dados

Embora o paralelismo aconteça em vários lugares (periféricos, etc.) e em diferentes níveis (registradores, caches, etc.) em um computador moderno, um dos primeiros lugares que esteve disponível foi nos processadores vetoriais. As chamadas máquinas vetoriais foram as que começaram a atual era da supercomputação. A idéia dessas máquinas é executar uma mesma instrução paralelamente em cada elemento de um vetor de dados similares. Nas primeiras máquinas desse tipo, os vetores eram armazenados na memória, depois começaram a ser armazenados em registradores especiais dentro da CPU. Exemplos de máquinas desse tipo são as CDC Star-100, CRAY 1 e CRAY X-MP.

Atualmente, esse modelo de arquitetura é chamado SIMD[3] e perdura como unidades funcionais nos processadores superescalares atuais. Exemplos desse tipo de unidades funcionais são as que executam o conjunto de instruções MMX da Intel ou o conjunto 3DNow! da AMD.

Existe uma outra vertente desse modelo que é implementada como estilo de programação, e não como uma forma de arquitetura de computadores, que é chamada de

[2]http://www.top500.org
[3]SIMD: Single Instruction, Multiple Data

paralelismo de dados. O particionamento dos dados nessa variante do modelo é realizado pelo compilador, e não pelo programador. Exemplos são o High-Performance Fortran e o OpenMP, onde o programador dá dicas para o compilador achar o paralelismo nos dados.

Não importando a variante, o modelo continua praticamente o mesmo: o paralelismo advém inteiramente dos dados; o programa em si se parece muito com um programa seqüencial.

4.2.2 SMP

O modelo de paralelismo de dados é determinado implicitamente pela independência dos dados. Os modelos onde o paralelismo não é determinado por essa independência, mas sim especificado explicitamente pelo programador, é chamado de paralelismo de *controle*.

Um modelo simples de paralelismo de controle é o modelo de *memória compartilhada*, implementado pela arquitetura *SMP*, do inglês *symmetric multiprocessor*. Esse nome, *symmetric*, é devido a utilização de processadores de um mesmo tipo na construção da máquina e da ausência de hierarquia entre eles.

Nesse modelo, cada processador tem total acesso a um único espaço de endereçamento. O acesso às posições de memória manipuladas por vários processos tem que ser coordenado pelo uso de semáforos ou outros tipos de mecanismos de sincronização colocados pelo programador. Pode-se ver na figura 4.1 um esquemático desse modelo. As várias CPUs do sistema têm acesso a toda memória disponível por um único barramento. Exemplos de máquinas que usam esse modelo são as da família SGI Power Challenge.

Torna-se muito difícil e caro construir máquinas de memória compartilhada que possuam mais de umas poucas dezenas de processadores. Até para máquinas mono-processadas, a tecnologia das memórias atuais já não consegue suprir a demanda de dados

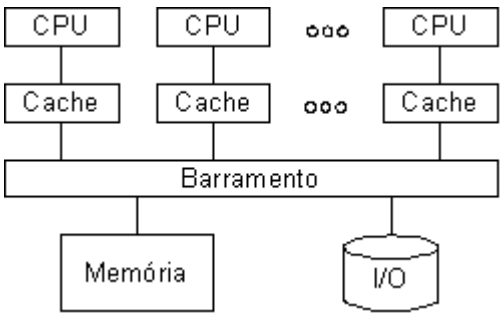

Figura 4.1: Exemplo de arquitetura SMP

requeridas por CPUs cada vez mas rápidas. Cada vez mais se observam processadores mais rápidos com caches maiores para esconder a grande latência das memórias RAM atuais. Com a colocação de mais de um processador no já sobrecarregado barramento de dados, além de aumentar a pressão sobre o sistema de memória, cria-se um outro problema: a coerência de caches.

O problema da coerência de caches se deve à existência de várias cópias de um mesmo dado distribuídas em caches de processadores diferentes. Quando uma dessas CPUs modifica esse dado, todas as outras que possuem cópia dele precisam ser avisadas, o que acarreta mais tráfego no barramento e mais alguns acessos à memória.

Uma forma de resolver esses problemas são as arquiteturas NUMA. Nesse tipo de arquitetura, alguns acessos à memória levam mais tempo que outros. No modelo SMP padrão, todos os acessos que passam pelos caches locais das CPUs e vão na memória compartilhada levam um mesmo tempo. A arquitetura NUMA será vista em mais detalhes na seção seguinte.

4.2.3 NUMA

A idéia da arquitetura NUMA, do inglês *Non-Uniform Memory Access*, é resolver o problema da escalabilidade e da contenção no barramento de memória. Nesse tipo de arquitetura, cada processador possui uma memória local e cada um deles pode endereçar a memória local de todos os outros. Ou seja, essa arquitetura possui um espaço de endereçamento compartilhado entre todos os seus processadores.

A arquitetura NUMA utiliza um hardware similar a MPP, a diferença entre os dois é o espaço de endereçamento. Na NUMA, é compartilhado, na MPP é individual para cada elemento processador. A memória é dividida em cada um dos processadores, como pode ser visto na figura 4.2.

Em uma máquina NUMA, apesar de melhorarem a contenção de memória e a escalabilidade em relação a arquitetura SMP, ainda existe o problema da coerência de cache. Esse problema é resolvido por um método de diretório. Nesse método, cada processador sabe onde a cópia mais recente de um dado se encontra.

O modelo recebe o nome de *Non-Uniform Memory Access* pois acessos à memória possuem tempos variados, diferentemente da SMP. O acesso à memória local por um processador é feito pelo barramento local. Acessos à memória remota, ou seja, de um processador vizinho, levam mais tempo por que precisam passar pela rede de interconexão. Através dessas diferenças nos tempos de acessos, essa arquitetura ganhou o seu nome. Exemplos de famílias de máquinas utilizando essa arquitetura são as Cray T3D, SGI Origin e as Hewlett-Packard Exemplar.

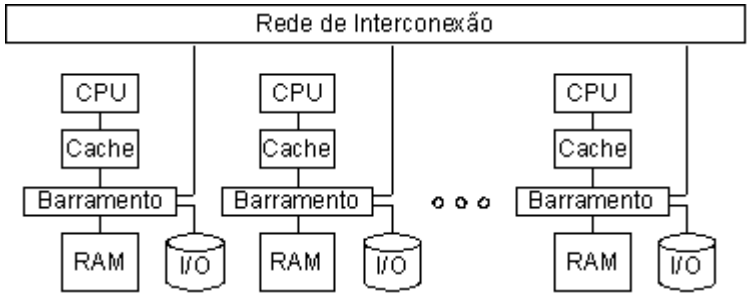

Figura 4.2: Arquiteturas MPP e NUMA

4.2.4 MPP

O modelo de *troca de mensagens*, implementado pela arquitetura MPP (*Massively Parallel Processors*), consiste de um conjunto de processos que possuem somente memória local, mas que são capazes de se comunicarem com outros processos enviando e recebendo mensagens. O nome MPP vem da alta escalabilidade presente nesse tipo de máquinas. Existem máquinas desse tipo com mais de 10000 processadores.

Uma das características mais marcantes desse modelo é que a troca de dados entre dois processos requer que ambos realizem operações, ou seja, um dos processos realiza a operação de envio enquanto o outro realiza a de recebimento. Esse modelo está sendo utilizado no presente trabalho.

A arquitetura desse modelo é mostrada na figura 4.2. Vários elementos processadores são ligadas a uma rede de interconexão. Cada elemento processador possui sua própria memória e sub-sistema de I/O. Um processo em execução não consegue acessar a memória de um outro como na arquitetura NUMA; é necessário que os processos cooperem trocando mensagens pela rede de interconexão.

Não é mostrado na figura uma topologia de rede específica pois não faz parte do modelo. Além disso, na prática, várias famílias de máquinas utilizam topologias diferentes:

o IBM Blue Gene/L utiliza uma topologia 3D torus, o NEC Earth Simulator utiliza crossbar switches, Intel IPSC/860 usa a topologia hipercubo, Intel Paragon e ASCI TFLOPS usam malha, IBM SP-1 utiliza variações de multilevel switches, o MULTIPLUS[26], multiprocessador desenvolvido no NCE/UFRJ, utiliza uma rede multi-estágio formada por crossbar 2x2. No presente trabalho o tipo de rede utilizada é a Ethernet[27], que utiliza a topologia de barramento. Dada essa grande variedade de topologias em uso, torna-se praticamente irrelevante para o programador qual está sendo utilizada.

4.3 MPI

No método de troca de mensagens para aplicações de computação paralela, um conjunto de processos executa programas escritos em uma linguagem seqüencial padrão, tal como C ou FORTRAN. Nesses programas são inseridas chamadas a bibliotecas para envio e recebimento de mensagens.

A biblioteca MPI[4][28, 29] é o resultado de um fórum que contou com a participação da comunidade científica e de grandes empresas de computação paralela, tais como IBM, Cray, Intel, Convex e Thinking Machines. O padrão foi baseado nas experiências com outras bibliotecas, sendo algumas delas as bibliotecas PVM[30] e PARMACS[31]. O principal objetivo da especificação da biblioteca MPI era permitir que os usuários não tivessem que abrir mão do desempenho para obter portabilidade. Ou seja, um usuário pode escrever programas que podem utilizar um hardware especializado de um determinado fabricante e mesmo assim não perder a portabilidade, pois em cada sistema a implementação da biblioteca utiliza da melhor forma possível o hardware disponível.

A MPI é uma biblioteca complexa. No seu todo, ela contém 129 funções, muitas delas

[4]MPI: Message-Passing Interface

com numerosos parâmetros ou variantes. Atualmente, o padrão especifica a biblioteca MPI para as linguagens C, C++ e FORTRAN. Existem tanto implementações de domínio público tais como LAM-MPI[32, 33] e MPICH[34] quanto proprietárias, tais como as das empresas Sun e IBM.

4.3.1 Modelo de Programação

No modelo de programação da MPI, a computação compreende um ou mais processos que se comunicam chamando rotinas da biblioteca para enviarem e receberem mensagens entre si. Na maioria das implementações da MPI, um conjunto fixo de processos é criado na inicialização da aplicação paralela e um processo é executado por cada processador. No entanto, estes processos podem executar diferentes programas. Desta maneira, o modelo de programação da MPI muitas vezes passa a ser chamado de *Multiple Program Multiple Data* (MPMD) para distingui-lo do modelo *Single Program Multiple Data* (SPMD), no qual cada processador executa um mesmo programa.

Existem vários mecanismos utilizados para trocar dados entre os processos. Processos podem utilizar operações de comunicação ponto-a-ponto para enviar uma mensagem para um destinatário específico; essas operações podem ser usadas para implementar uma forma de comunicação local e não estruturada. Um grupo de processos pode chamar as funções de comunicação coletiva para realizar operações de comunicação global, tais como *broadcast* e dispersão de dados.

Outro mecanismo importante disponível na biblioteca MPI é a capacidade de testar, sem receber, a ocorrência da chegada de uma mensagem. Esse mecanismo é importante pois cria suporte para comunicação assíncrona entre processos. Se para cada mensagem que chega o processo fosse obrigado a recebê-la na hora, ele estaria sempre implicitamente sincronizado com o processo que lhe enviou a mensagem.

```
MPI_Init()        :  Inicializa a computação MPI

MPI_Finalize()    :  Termina a computação

MPI_Comm_size()   :  Determina o número de processos

MPI_Comm_rank()   :  Determina o identificador do processo

MPI_Recv()        :  Recebe uma mensagem

MPI_Send()        :  Envia uma mensagem
```

Figura 4.3: Funções básicas da biblioteca MPI

4.3.2 Funções Básicas

Mesmo com a biblioteca MPI sendo um sistema complexo e com muitas funções, uma grande variedade de problemas pode ser resolvido com somente seis dessas funções, as quais podem ser vistas na figura 4.3. Neste trabalho as cinco primeiras foram utilizadas. A sexta, `MPI_Send()` foi substituída pela sua variante `MPI_Isend()`, que será vista na seção 4.3.4.

As duas primeiras funções servem, respectivamente, para inicializar e terminar a biblioteca. A função `MPI_Init()` precisa ser chamada antes de qualquer outra função MPI e deve ser chamada somente uma única vez. Nenhuma função MPI pode ser chamada após `MPI_Finalize`. A terceira, `MPI_Comm_size()` determina o número de processos na aplicação paralela. Sua utilização é importante pois a aplicação pode, em suas várias execuções, rodar em um número diferente de processadores. Já a quarta retorna um número inteiro chamado *rank*. Todos os processos são identificados com um rank único e contíguo começando em zero. Esse valor é usado, por exemplo, como endereço nas comunicações ponto-a-ponto ou para identificar o processo como mestre no modelo Mestre/Escravo.

Todas as funções menos as duas primeiras recebem um parâmetro chamado *comunicador*. Um comunicador identifica o grupo de processos e o contexto na qual uma operação está sendo realizada. Comunicadores servem como um mecanismo para identificar subconjuntos de processos dentro de uma aplicação, garantindo assim a criação de programas modulares e a utilização correta das mensagens recebidas. O principal comunicador, que especifica todos os processos da aplicação, é o `MPI_COMM_WORLD`.

As funções `MPI_Send()` e `MPI_Recv()` são usadas, respectivamente, para enviar e receber mensagens. Uma chamada a essas funções tem a seguinte forma:

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

`count` determina o número de elementos do tipo `datatype` a serem lidos, no caso de envio, ou escritos, no caso de recebimento, a partir do endereço `buf`. Os argumentos `source` e `dest` contém, respectivamente, o rank do remetente e o rank do destinatário da mensagem. O argumento `tag` é um número inteiro que indica qual é o tipo de mensagem atual. Esse tipo é um valor criado e utilizado pelo programador para saber como seu programa deve processar a mensagem. Já `comm` é o comunicador onde se encontra o processo destinatário.

O programador tem a liberdade, no caso da função `MPI_Recv()`, de receber uma mensagem com qualquer `tag` ou com qualquer remetente ao invés de um específico. Basta ele colocar como parâmetro as constantes `MPI_ANY_TAG` ou `MPI_ANY_SOURCE`. Se depois for do seu interesse descobrir qualquer um desses valores, basta olhar o parâmetro `status`.

Os modelos de programação que utilizam troca de mensagens são, geralmente, não determinísticos. A biblioteca MPI garante que duas mensagens enviadas de um processo

A, para um outro processo B chegarão na mesma ordem de envio. Porém a ordem de chegada das mensagens enviadas por dois processos, A e B, para um terceiro, C, não é determinada. É de responsabilidade do programador garantir o comportamento determinístico do sistema.

4.3.3 Comunicação Coletiva

Algoritmos paralelos normalmente têm a necessidade de se comunicar de forma coordenada entre seus processos. Por exemplo, todos os processos podem ter que cooperar para transpor uma matriz distribuída ou somar um conjunto de números distribuídos um a um nos processos. Isso poderia ser realizado pelo programador utilizando `MPI_Send()` e `MPI_Recv()`. Mas, por conveniência, e para permitir implementações otimizadas para um tipo especial de hardware, a MPI provê um conjunto de funções de comunicação coletiva que realiza as operações mais freqüentes desse tipo. Dentre essas funções, podemos citar as seguintes:

Barreira Sincroniza todos os processos

Broadcast Envia dados de um processo para todos

Gather Coleta dados de todos os processos e envia para um deles

Scatter Dispersa os dados de um processo para todos

Redução Soma, multiplica, etc., dados distribuídos entre todos os processos e envia o resultado para um deles

Todas essas funções são executadas coletivamente, significando que cada processo de um grupo de processos chama a função com os mesmos parâmetros.

Nas seções seguintes serão vistas com mais detalhes as operações de barreira e de movimento de dados (Broadcast, Gather, etc.). As operações de redução não serão vistas pois fogem ao escopo deste trabalho.

4.3.3.1 Barreira

A função `MPI_Barrier()` implementa a sincronização por barreira no modelo de programação da MPI. Ela é utilizada para realizar a sincronização de um grupo de processos. Esse grupo de processos é especificado pelo seu único parâmetro, que é o comunicador. Nenhum processo retorna da chamada dessa função até que todos os processos tenham realizado a chamada. Uma barreira é uma forma simples de separar duas fases de uma computação e também serve para garantir que mensagens geradas em uma fase não interfiram na outra.

4.3.3.2 Movimento de Dados

`MPI_Bcast()`, `MPI_Gather()` e `MPI_Scatter()` são funções coletivas para movimento de dados. Em todas elas os processos interagem com um processo especial chamado de *raiz* para realizar, respectivamente, um broadcast, uma coleta ou uma dispersão dos dados.

As operações dessas funções são ilustradas na figura 4.4 para um grupo de quatro processos. Em cada conjunto de 16 caixas, cada linha representa a localização dos dados em diferentes processos. Então, no broadcast, o dado D_0 está somente no processo 0; depois da chamada, ele é replicado em todos os processos.

`MPI_Bcast()` implementa a operação de broadcast, onde um único processo, o processo raiz, envia os mesmos dados para todos os outros processos. No exemplo acima, o processo 0 é o processo raiz.

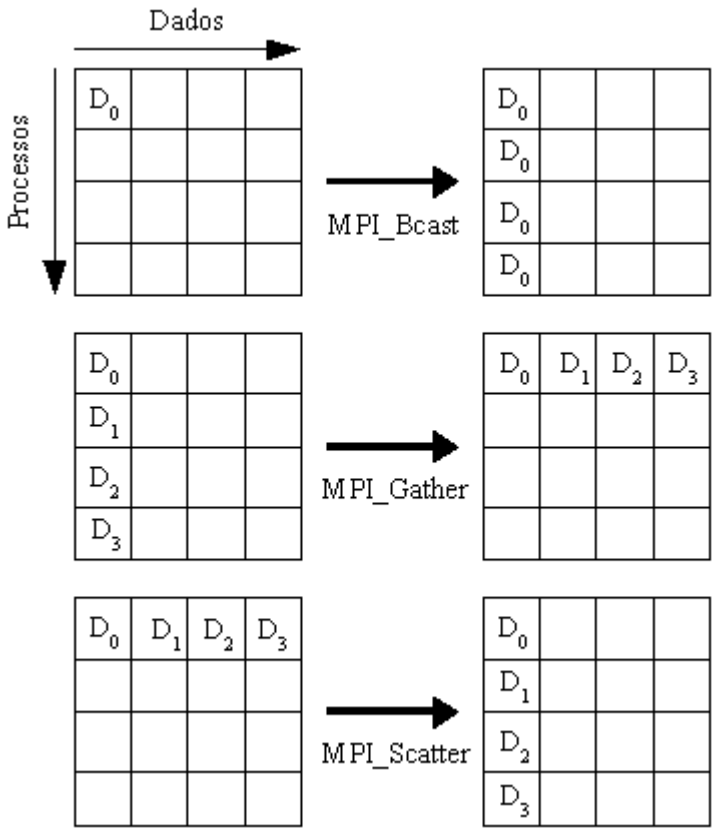

Figura 4.4: Operações de movimento de dados da biblioteca MPI

A função `MPI_Gather()` realiza a coleta de dados. Todos os processos, incluindo o raiz, enviam seus dados contidos em um buffer de entrada para o processo raiz. Esse processo coloca os dados recebidos em um buffer de saída, que ocupa um área contígua de memória, onde os dados do processo i precedem os dados do processo $i + 1$. Então, o buffer de saída no processo raiz tem que ser P vezes maior que o buffer de entrada, onde P é o número de processos participando da operação. O buffer de saída na chamada a `MPI_Gather()` nos processos diferentes do raiz é ignorado.

Já a função `MPI_Scatter()` realiza a dispersão dos dados; exatamente o oposto da `MPI_Gather()`. Um processo raiz envia os dados para todos os processos, enviando sua i-ésima parte dos dados para o processo i; cada processo recebe os dados no seu buffer

de saída. Note a sutil diferença entre essa função e `MPI_Bcast()`: enquanto nesta todos os processos recebem o mesmo valor do raiz, na `MPI_Scatter()` cada processo recebe um valor diferente.

4.3.4 Comunicação Assíncrona

Na comunicação síncrona, os processos produtores e consumidores de dados atuam de forma coordenada, cooperando nas trocas de mensagens. O produtor realiza uma operação de envio dos dados e espera o consumidor efetuar uma operação para recebê-los. Já na comunicação assíncrona, os consumidores não sabem quando os produtores enviarão os dados, então, precisam verificar se uma mensagem chegou ou não. Se os dados não estão disponíveis, o processo consumidor pode continuar realizando algum trabalho útil ao invés de ficar bloqueado em uma `MPI_Recv()` esperando os dados chegarem.

Para escolher entre realizar uma comunicação síncrona ou assíncrona o programador precisa saber qual tipo de função de envio deverá ser usado. A biblioteca MPI fornece dois tipos de funções para essa finalidade: o *bloqueante* e o *não bloqueante*.

Funções de envio bloqueantes somente retornam quando a comunicação foi realmente efetuada, ou seja, quando ocorreu o envio e o destinatário efetivamente fez uma chamada a `MPI_Recv()` ou a uma de suas variantes. Como ocorre essa sincronização nos processos entre as operações de envio e recebimento, esse tipo de comunicação ganha o nome de síncrona. A principal função de envio da MPI que realiza esse tipo de comunicação é a `MPI_Send()`.

As funções de envio não bloqueantes retornam praticamente após serem chamadas, elas somente pedem à biblioteca para enviar os dados. Nenhuma delas espera o processo destinatário efetuar um `MPI_Recv()`. Somente algumas dessas funções esperam o envio,

mas não o recebimento, para retornar. Como não ocorre sincronização entre o envio e recebimento, essas funções realizam, no modelo de programação da MPI, a comunicação assíncrona. A principal função dessa classe se chama `MPI_Isend()`.

O principal problema da comunicação assíncrona é saber se um processo tem ou não mensagens para receber. Para isso, a biblioteca MPI disponibiliza duas funções: `MPI_Probe()` e `MPI_Iprobe()`.

A função `MPI_Iprobe()` verifica a existência ou não de mensagens prontas para serem recebidas sem realmente recebê-las. Essa funcionalidade permite ao programador escrever programas que conseguem entremear a computação local com o processamento das mensagens. Uma chamada a essa função tem geralmente a seguinte forma:

```
MPI_Iprobe(source, tag, comm, flag, status)
```

e retorna um valor booleano `flag` indicando se uma mensagem com o `source`, `tag`, e `comm` especificados está disponível ou não. Se uma mensagem com essas características estiver disponível, ela poderá ser recebida utilizando `MPI_Recv()`. A chamada a função `MPI_Recv()` deve conter como argumentos os mesmos `source`, `tag`, e `comm` da chamada a `MPI_Iprobe()`, senão corre-se o risco de receber uma mensagem diferente da desejada.

A `MPI_Probe()` é a versão bloqueante de `MPI_Iprobe()`. Ela não retorna até que uma mensagem com `source`, `tag`, e `comm` especificados esteja disponível.

4.4 Medidas de Desempenho

No modelo de computação serial há um processador que executa uma série de instruções para produzir um resultado, ou seja, existe exatamente um fluxo lógico de controle no programa. Isto é verdade até em programas, tais como sistemas operacionais, que

dão a impressão de estarem executando várias tarefas ao mesmo tempo. Mas, em um determinado instante de tempo, só há uma operação sendo executada pelo processador.

A computação paralela é um modelo que se propõe a produzir os mesmos resultados da computação serial utilizando múltiplos processadores. O problema a ser resolvido é dividido entre um certo número de processadores. Então, se um programa estiver rodando em P processadores, o ideal seria que ele rodasse P vezes mais rápido que em um só processador. Na prática isto é extremamente difícil de se obter devido a "overheads" tais como comunicação e partes do código intrinsecamente seriais.

Um problema, para funcionar bem no modelo de computação paralela, precisa ser dividido de uma forma sensata e eficiente. Uma divisão bem feita é imprescindível para a obtenção de um bom desempenho em uma máquina paralela. Geralmente, é necessário garantir que cada processador execute uma carga igual de trabalho, ou seja, que a carga de trabalho esteja *balanceada*. Outra característica desejável do programa é que ele passe a maior parte do tempo realizando cálculos do que se comunicando com outros processos [35, 36].

Torna-se necessário então a definição de medidas de performance de aplicações paralelas para que seja possível realizar comparações entre diferentes implementações delas, verificar seu desempenho em relação a versões seriais, avaliar o balanceamento de carga e o tempo de comunicação. As medidas mais comuns de desempenho utilizadas são o *tempo de execução*, o *speed-up* e a *eficiência*.

O tempo de execução de um algoritmo que resolve um problema de tamanho n em P processadores é definido por $T(n, P)$. Seu valor é obtido por meio de instrumentos de medição de tempo disponíveis no sistema operacional utilizado na plataforma de desenvolvimento. Exemplos: $T(100, 1)$ é o tempo de execução de um problema de tamanho 100

em 1 processador, $T(1000, 8)$ é o tempo de execução de um problema de tamanho 1000 em 8 processadores.

O speed-up paralelo obtido por um algoritmo paralelo rodando em P processadores é a relação entre o tempo de execução de um problema de tamanho n rodando em um processador e o tempo de execução do mesmo problema com P processadores, ou seja:

$$S(n, P) = \frac{T(n, 1)}{T(n, P)}$$

A eficiência paralela é o speed-up dividido por P, ou seja:

$$E(n, P) = \frac{S(n, P)}{P} = \frac{T(n, 1)}{PT(n, P)}$$

Essas medidas especificadas acima não dão nenhuma informação sobre a qualidade do algoritmo paralelo. Elas simplesmente medem a qualidade da paralelização do algoritmo. Então, devem ser acompanhadas por dados que indiquem a *eficiência numérica* do algoritmo paralelo. Essa medida é a seguinte: $T_{best}(n)/T(n, 1)$, onde $T_{best}(n)$ é o tempo necessário para o melhor algoritmo seqüencial executar um problema de tamanho n.

A combinação das definições de speed-up e eficiência paralelos levam a noção de *speed-up total* e *eficiência total*, que são dados por:

$$\overline{S}(n, P) = \frac{T_{best}(n)}{T(n, P)}$$

$$\overline{E}(n, P) = \frac{\overline{S}(n, P)}{P} = \frac{T_{best}(n)}{PT(n, P)}$$

No entanto, considerações práticas limitam a utilidade dessas últimas definições. Pri-

meiramente, às vezes é muito difícil determinar qual é o *melhor* algoritmo seqüencial. Essa escolha pode depender do hardware utilizado, do tamanho n do problema, detalhes de implementação, dentre outros fatores. Além disso, a noção de melhor algoritmo pode mudar com o tempo, conforme melhores algoritmos vão aparecendo. Às vezes, também não está disponível uma boa implementação do algoritmo. Na prática, pode-se utilizar o tempo de execução de um bom algoritmo serial para substituir $T_{best}(n)$.

Como uma máquina de P processadores não pode executar P vezes mais rápido que uma máquina com um processador, obviamente temos:

$$S(n, P) \leq P$$

$$E(n, P) \leq 100\%$$

Os principais fatores que não permitem um speed-up linear, ou seja, que mantém $S(n, P) \leq P$ são a Lei de Amdahl, "overhead" dos algoritmos, desbalanceamento de carga e operações de comunicação e sincronização. Todos eles serão explicados em mais detalhes nas seções seguintes.

4.4.1 Lei de Amdahl

O speed-up que pode ser obtido em um computador paralelo pode ser significativamente limitado pela existência de uma pequena fração intrinsecamente serial do código que não pode ser paralelizada. Sendo α a fração das operações seqüenciais no código, onde $0 \leq \alpha \leq 1$, o speed-up máximo obtido por um computador paralelo com P processadores é dado pela lei de Amdahl, que é mostrada abaixo:

$$S(n, P) \leq \frac{1}{\alpha + \frac{(1-\alpha)}{P}} \leq \frac{1}{\alpha}$$

Então, pela lei de Amdahl, o speed-up máximo é dado por:

$$S_{max}(n, P) = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}$$

Por exemplo, quando 10% do código tem que ser executado seqüencialmente ($\alpha = 0.1$) e está à disposição um número muito grande de processadores ($P \rightarrow \infty$), o speed-up máximo será:

$$\lim_{P \to \infty} S(n, P) = \lim_{P \to \infty} \frac{1}{\alpha + \frac{(1-\alpha)}{P}} = 10$$

Por exemplo, se em um código com $\alpha = 0.1$ forem utilizados apenas 8 processadores ($P = 8$), temos como speed-up máximo:

$$S_{max}(n, 8) = \frac{1}{0.1 + \frac{(1-0.1)}{8}} = 4.706$$

É necessário lembrar que, quando se aumenta o tamanho n do problema, temos uma diminuição do valor de α, pois o programa passa mais tempo na parte paralela do seu código.

4.4.2 "Overhead" dos Algoritmos

O melhor algoritmo seqüencial para um determinado problema pode ser difícil ou impossível de paralelizar. Nesse caso, o algoritmo paralelo pode executar um número maior de instruções para resolver o mesmo problema, gerando assim esse tipo de "overhead".

Além disso, um programador, desejando diminuir o "overhead" de comunicação, pode optar por repetir cálculos em processadores diferentes. Com isso, ele evita que um processador envie o resultado dos cálculos para os outros, diminuindo assim a comunicação,

mas aumentando o número de instruções extras de cada processador.

4.4.3 Desbalanceamento de Carga

O tempo de execução de um algoritmo paralelo é determinado pelo tempo de execução do processador que tem a maior quantidade de trabalho a fazer, ou seja, o que demora mais a gerar os resultados.

Quando a carga de trabalho não está distribuída igualmente, ocorre o desbalanceamento de carga e, como resultado, processadores da máquina ficam ociosos, ou seja, alguns processadores tem que esperar os outros terminarem sua parte do problema.

4.4.4 Comunicação e Sincronização

Um processador executando um problema paralelo precisa realizar alguma comunicação para distribuir dados ou realizar alguma operação de sincronização. Se esse mesmo problema fosse resolvido com um só processador em um modelo de programação serial, essas operações de sincronização e comunicação não seriam necessárias. Qualquer tempo gasto nos programas paralelos tanto em comunicação quanto em sincronização é puramente "overhead".

4.5 Decomposição Regular de Domínio

Muitos problemas manipulam um conjunto muito grande de dados, arrumados em forma de grid. Para obter sua solução, aplicam-se operações nos dados que compõem o grid. Quando esses dados podem ser particionados em subgrids e distribuídos por um conjunto de processos, essas operações podem ser realizadas em paralelo. Essa técnica é chamada de *decomposição de domínio*[35]. Isso permite que o problema seja resolvido em

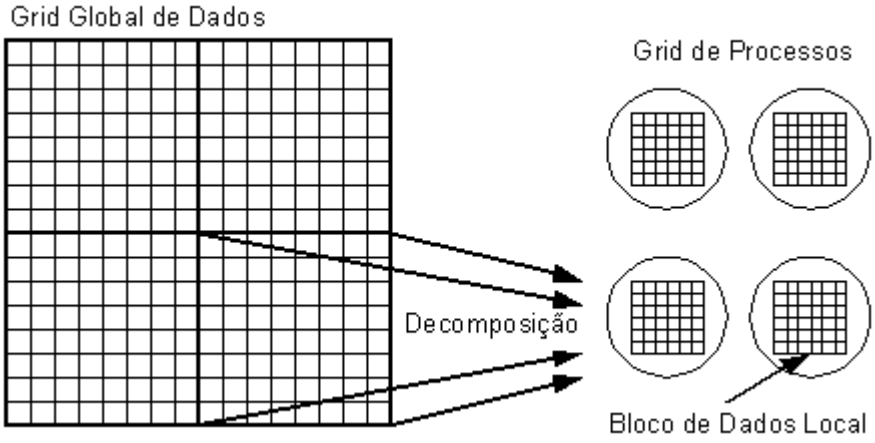

Figura 4.5: Exemplo de Decomposição Regular de Domínio

menor tempo ou que problemas muito maiores possam ser resolvidos.

Nessa técnica de particionamento de problemas, o objetivo é decompor os dados associados ao problema que se deseja solucionar. Se possível tenta-se dividir os dados em pedaços pequenos, aproximadamente do mesmo tamanho. Se todos os pedaços ficarem do mesmo tamanho, esse tipo de particionamento passa a ser chamado de *decomposição regular de domínio*.

Mais formalmente, o método da *decomposição regular de domínio* consiste na divisão de um grande grid de dados, o *grid global*, em subgrids regulares e da distribuição, via operações de comunicação, desses subgrids para processos separados. O grid global de dados é decomposto em seções chamadas de *blocos de dados locais*, sendo cada uma delas colocada sob controle de um determinado processo, como mostrado na figura 4.5. Deve-se lembrar que o grid de processos precisa estar organizado logicamente de forma a refletir a forma como o grid global foi dividido para, entre outros motivos, facilitar a comunicação entre os processos.

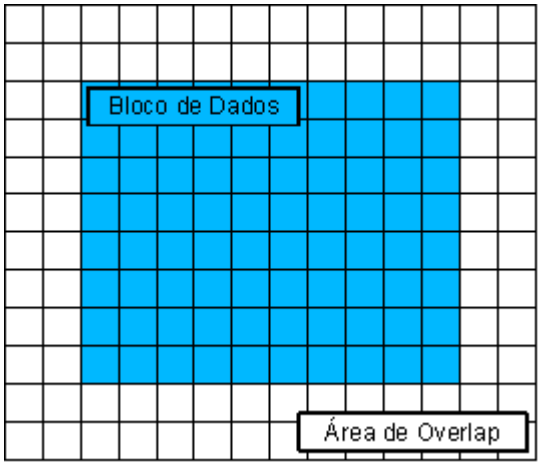

Figura 4.6: Matriz de dados de um processo

4.5.1 Atualizações nos Dados

Normalmente, em cada iteração do algoritmo o processo precisa realizar operações nos dados e atualizar o conteúdo de seu bloco de dados local. Para computar o próximo valor de um elemento do grid o processo realiza cálculos baseados no valor atual da célula e possivelmente nos valores das células vizinhas.

Quando é necessário recorrer as células vizinhas, nem sempre elas se encontram no bloco de dados local. Então o processo realiza uma comunicação com o processo vizinho no grid de processos e coloca o valor das células requisitadas numa região chamada de *área de overlap*. Essa área de overlap junto com o bloco local de dados forma a *matriz de dados*, como pode ser visto na figura 4.6.

A largura de uma área de overlap não precisa ser a mesma em todas as direções. Ela depende simplesmente da localidade das referências utilizadas no cálculo para atualização. Se for necessário para a atualização apenas o valor da própria célula não é necessária a presença de uma área de overlap. Já a área de overlap de um cálculo que precise de todas as células adjacentes a uma determinada célula, precisará ter a largura de um elemento

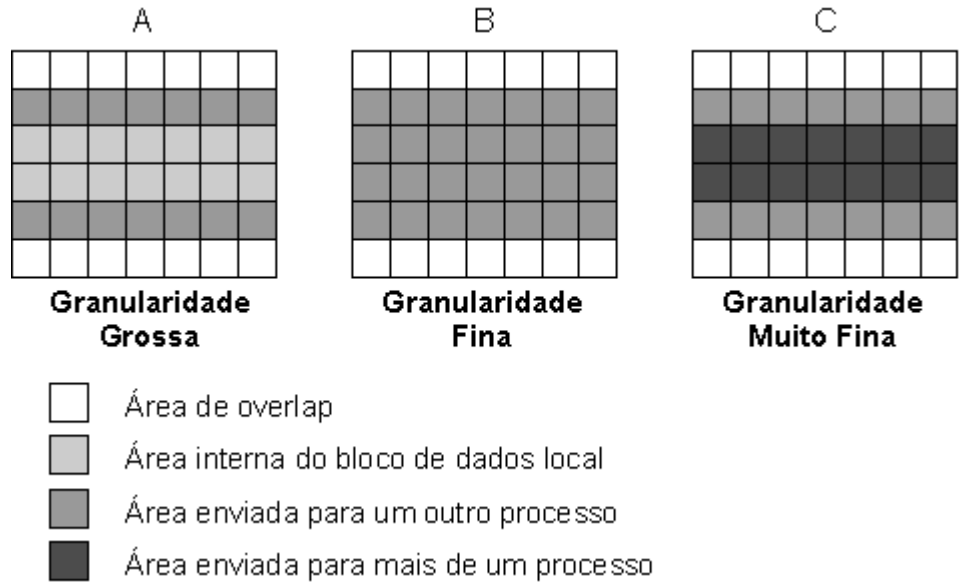

Figura 4.7: Granularidade

em todas as direções.

Se a largura de uma região de overlap for suficientemente grande, então todos os elementos do bloco de dados local precisam ser trocados com os processos vizinhos. Uma decomposição onde isso ocorre possui *granularidade fina*. Em uma decomposição na qual alguns elementos dos blocos de dados nunca contribuem na atualização é dito que ela possui *granularidade grossa*.

Um grid que possui decomposição de granularidade fina gasta mais tempo mantendo atualizadas as regiões de overlap devido a maior quantidade de dados a comunicar com o processo vizinho. Assim, a eficiência da paralelização é menor do que nos casos com granularidade grossa.

A figura 4.7 mostra o efeito da diminuição da granularidade em um bloco de dados, cada um correspondendo a três diferentes cálculos de atualização. Todos os três cálculos somente utilizam elementos vizinhos na vertical, então só são necessárias áreas de overlap

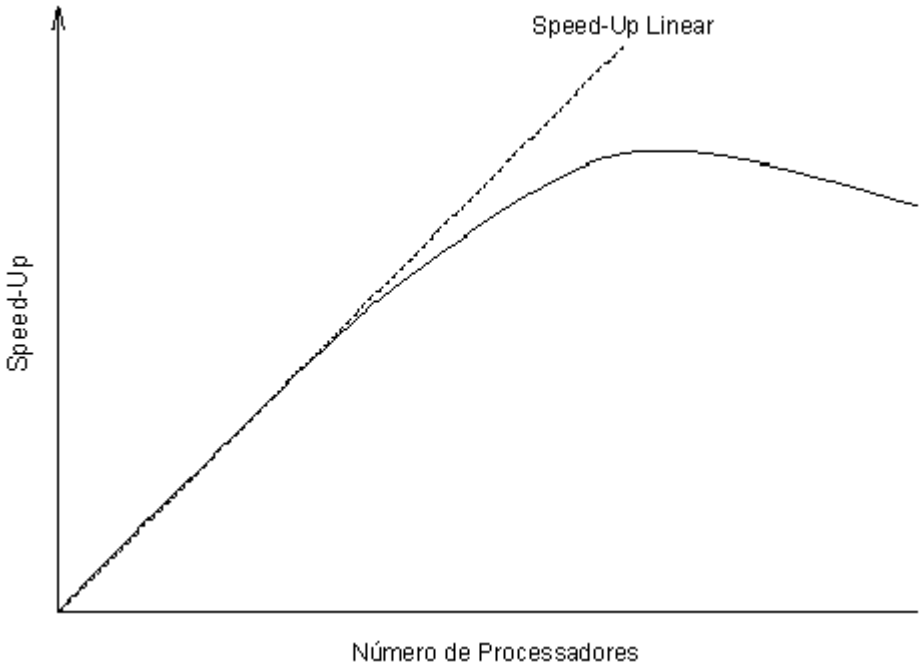

Figura 4.8: Exemplo de gráfico de Speed-Up vs. Número de Processadores

verticais.

Se o número de processos que está disponível para uma aplicação aumenta, a granularidade da decomposição torna-se mais fina. Chegará um certo ponto onde não é mais interessante adicionar mais processos, pois o "overhead" imposto pela granularidade iria ser maior que os benefícios gerados pelo ganho de poder de processamento.

4.5.2 Desempenho

Conforme são adicionados mais processadores para resolver um problema de tamanho fixo, o tamanho do bloco de dados local de cada um diminui. Enquanto que inicialmente a adição de mais processadores aumenta o desempenho, após um certo ponto, o speed-up irá começar a diminuir. Chegando a um ponto onde a adição de mais processadores irá causar uma diminuição do desempenho.

Essa diminuição de desempenho acontece pois os processadores não estão realizando trabalho suficiente para justificar o custo do aumento da comunicação. Isto pode ser visto no gráfico da figura 4.8 quando a linha que representa o speed-up começa a cair. A forma precisa do gráfico mostrado na figura depende do hardware em que o problema está rodando.

Deve ser lembrado que o gráfico é para um problema de tamanho fixo. Normalmente, quando se quer aumentar o número de processadores é porque se deseja aumentar o tamanho do problema a ser resolvido.

5 Algoritmo Paralelo

Neste capítulo é apresentado o desenvolvimento do cálculo paralelo do diagrama de Voronoi. Na seção 5.1 são apresentadas as condições de contorno do algoritmo. Depois, na seção 5.2, é explicado como o método da decomposição regular de domínio foi aplicado na paralelização do algoritmo.

O pseudo-código e os detalhes de implementação são apresentados na seção 5.3. Por último, na seção 5.4, são mostrados exemplos de diagramas calculados com diferentes configurações de obstáculos, seus resultados e a analise do desempenho do algoritmo.

5.1 Algoritmo Numérico 3D Paralelo

O algoritmo numérico 3D para o cálculo do diagrama de Voronoi trabalha em um grid cúbico com n de aresta, ou seja, que possui n^3 células. Na atual implementação não é necessário mantê-las todas na memória devido a estrutura Hash+Bloom, mostrada no capítulo 3.

Como entrada, o algoritmo recebe um conjunto de obstáculos. Esses obstáculos devem ser aproximados pela esfera de menor raio possível que contenha totalmente o objeto. Essa limitação é devida a função utilizada para calcular a distância entre um ponto do grid e os obstáculos. Uma função mais genérica poderia ser implementada para, por exemplo, calcular a distância entre um ponto do grid e o fecho convexo de um obstáculo.

Os conjuntos `Expandido` e `Não_Expandido` possuem a mesma função que exerciam no algoritmo serial, ou seja, armazenar as células que já sofreram ou não o processo de expansão. Na versão paralela do algoritmo, esses conjuntos são representados pela estrutura Hash+Bloom. As funções `Insere()`, `Remove()` e `Busca()` operam diretamente em uma estrutura desse tipo.

A paralelização do algoritmo foi realizada utilizando o método da decomposição regular de domínio. Como já visto no capítulo 4, o método da decomposição regular de domínio particiona igualmente o grid entre os processos da aplicação paralela. A troca de mensagens entre estes processos foi realizada utilizando a biblioteca MPI.

5.2 Decomposição Regular de Domínio

Tendo a área de trabalho e seus obstáculos definidos, o algoritmo realiza o particionamento da área de trabalho. Esse particionamento é realizado utilizando o método da decomposição regular de domínio. Partições de igual tamanho ao longo do eixo Z do grid são distribuídas para cada um dos elementos processadores como visto na figura 5.1. Cada um dos processadores recebe um tamanho igual do grid.

No exemplo da figura 5.1, estão disponíveis três elementos processadores. Então, a área de trabalho é particionada em três partições. A primeira partição, chamada P1, fica sob a responsabilidade da CPU 1, P2 com a CPU 2 e assim por diante.

Os processadores somente se comunicam com outros cujas partições façam fronteira com a sua. A comunicação ocorre quando um processador, durante a execução do algoritmo, descobre uma célula de Voronoi na borda da sua partição. Então, uma mensagem é enviada para o processador vizinho avisando sobre a existência dessa célula. Quando o processador vizinho recebe essa mensagem, ele insere essa célula no seu conjunto de

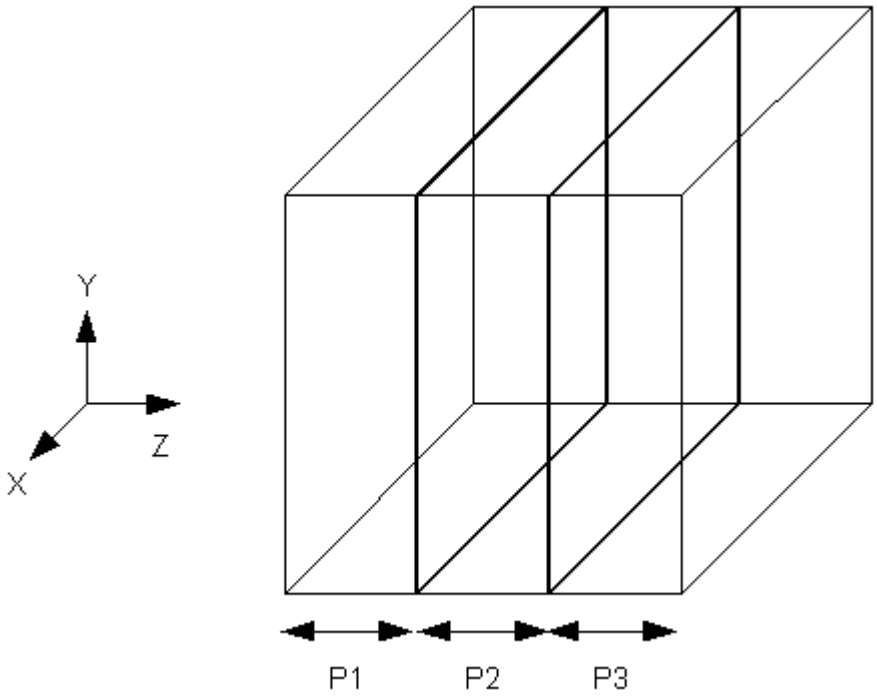

Figura 5.1: Particionamento da área de trabalho

células não expandidas e continua seu processamento normalmente.

Esse tipo de comunicação é necessário pois o particionamento da área de trabalho torna o diagrama não conexo dentro das partições individuais. Sem essa comunicação, partes do diagrama não seriam calculadas pelo algoritmo numérico quando ocorresse o particionamento.

5.3 Pseudo-código do Algoritmo

Realizado o particionamento e tendo disponível a configuração dos obstáculos na área de trabalho, o algoritmo começa sua execução. Os dois conjuntos chamados `Expandido` e `Não_Expandido` são implementados pela estrutura Hash+Bloom.

Em uma aplicação paralela escrita em MPI, o usuário especifica na linha de comando

quantos processos estarão disponíveis para execução. Esse número é disponibilizado internamente para o programa pela função `MPI_Comm_size()`. A chamada a essa função retorna um valor *size* indicando esse número.

Cada um dos *size* processos da aplicação paralela possui um identificador único, variando entre 0 e $size - 1$, chamado *rank*. O processo obtém seu *rank* realizando uma chamada a função `MPI_Comm_rank()`. Em um programa paralelo, o *rank* é utilizado, dentre muitas outras coisas, para identificar o destinatário de uma mensagem.

O pseudo-código do algoritmo númerico 3D paralelo é mostrado na figura 5.2. Para a obtenção do algoritmo paralelo, foi necessário modificar o algoritmo serial em várias partes, sendo elas a sua inicialização, sua condição de parada e a troca de mensagens devido a decomposição regular de domínio. Todas essas modificações serão apresentadas nas seções seguintes.

5.3.1 Inicialização

O algoritmo numérico 3D seqüencial é inicializado com uma ou mais células das bordas do grid inseridas no seu conjunto de células não expandidas. Mas, no algoritmo paralelo, nem todos os processos possuem nas suas partições células dos cantos do grid.

Com isso, somente os processos com $rank = 0$ e $rank = size - 1$ possuem células nos seus conjuntos de células não expandidas no momento da inicialização, enquanto os outros começam com o conjunto vazio. No exemplo da figura 5.1, onde $size = 3$, o processo com $rank = 0$ seria P1 e com $rank = size - 1$, P3. Os outros processos, nesse exemplo P2, ficam aguardando até receber uma mensagem com as coordenadas de uma célula da fronteira para iniciar o algoritmo.

O passo da inicialização discutido aqui é mostrado no pseudo-código da figura 5.2

```
Voronoi3D_Paralelo()
{
01:    Conjunto Não_Expandido, Expandido;
02:
03:    inicialização();
04:    while(NOT fim) {
05:        if(mensagem recebida)
06:               receber_mensagens();
07:        C = Remove(Não_Expandido);
08:        for each(face F na célula C) { // Expande a célula C
09:            if(Voronoi atravessa face F) {
10:                Seja K a célula adjacente a C pela face F;
11:                if(K não pertence a partição local)
12:                    enviar_mensagem(K, P); // P = proc. vizinho
13:                else if(NOT Busca(Não_Expandido, K) AND NOT
14:                    Busca(Expandido, K))
15:                    Insere(Não_Expandido, K);
16:            }
17:        }
18:        Insere(Expandido, C);  // Terminou a expansão de C.
19:        fim = terminar();  // Calcula a condição de parada
20:    }
21:    return(Expandido);
}
```

Figura 5.2: Pseudo-código do algoritmo numérico paralelo para o cálculo diagrama de Voronoi em 3D

na linha 03 como a chamada a função `inicialização()`. Ela simplesmente insere as células do canto do grid no conjunto `Não_Expandido` dos processos com $rank = 0$ e $rank = size - 1$ e, nos outros, deixa o conjunto vazio.

5.3.2 Condição de Parada

A condição de parada do laço principal no algoritmo numérico 3D seqüencial era o esvaziamento total do conjunto `Não_Expandido`. Mas, como visto na seção anterior, no

momento da inicialização somente dois processos possuem células nesse conjunto. Os outros encontram-se com ele vazio.

Tomando novamente como exemplo o particionamento da figura 5.1 na página 99, o cálculo paralelo começa com os processadores 1 e 3 com trabalho e o processador 2 ocioso, ou seja, com seu conjunto de células não expandidas vazio. Se a condição de término do algoritmo serial não fosse modificada, o processador 2 daria como terminado o cálculo imediatamente. As mensagens que chegassem enviadas pelos processadores 1 e 2 seriam ignoradas e, por conseqüência, o diagrama não seria calculado corretamente.

Esse fato torna necessária uma modificação na condição de parada original. Agora é necessário para o término do algoritmo que todos os processos estejam com seu conjunto `Não_Expandido` vazio. Isto garante que o diagrama de Voronoi foi totalmente calculado, não restando nenhuma célula para processar.

Ao fim de cada iteração do laço principal do algoritmo paralelo, cada processador comunica para os outros via a função `MPI_Allgather()` o estado do seu conjunto `Não_Expandido`, ou seja, se ainda contém elementos ou se está vazio. Ao chamar a função de comunicação coletiva `MPI_Allgather()`, cada processo envia um dado seu e recebe, em troca, os dados de todos os outros processos. Neste caso, o dado enviado é o estado do conjunto `Não_Expandido`. Baseado no retorno dessa função, o algoritmo decide terminar ou não.

O comportamento visto no parágrafo acima é representado na figura 5.2, linha 19, pela chamada a função `terminar()`. Ela retorna o valor da variável booleana `fim`, que controla o laço `while` da linha 04. Na função `terminar()`, uma chamada a `MPI_Allgather()` obtém o estado do conjunto `Não_Expandido` em todos os processadores e decide quanto ao término do algoritmo.

5.3.3 Troca de Mensagens

Devido ao particionamento do grid, o diagrama perde sua propriedade de ser conexo. Uma solução para esse problema é a comunicação para avisar o processo vizinho sobre células na fronteira das partições. Somente processos com partições adjacentes se comunicam. Essa comunicação é realizada através de chamadas a funções da biblioteca MPI, sendo elas as funções `MPI_Send()`, `MPI_Recv()` e `MPI_Iprobe()`.

A função `MPI_Send()` realiza o envio de dados de um processo para outro especificado pelo seu *rank*. No pseudo-código da figura 5.2, o papel dessa função é realizado na linha 12 pela função `enviar_mensagem(K, P)`, que envia a célula de fronteira `K` para o processo vizinho com rank `P`.

Na linha 11, o algoritmo verifica se a célula encontra-se ou não na sua partição. Essa verificação é realizada inspecionando o valor da coordenada Z da célula no grid. Se o valor de Z pertencer a partição do processo atual, o algoritmo testa se a célula já pertence a um dos conjuntos e, caso não pertença, a insere no conjunto `Não_Expandido`. Se a coordenada Z estiver fora da sua partição, a célula é enviada para o processo vizinho.

O tratamento das mensagens recebidas por um processo é realizado nas linhas 05 e 06. O `if` da linha 05 testa pela chegada de novas mensagens utilizando a função `MPI_Iprobe()`. Caso tenham chegado novas mensagens, a chamada a esta função retorna um valor verdadeiro, caso contrário, retorna um valor falso.

No caso do valor retornado ser verdadeiro, o processo realiza uma chamada a função `receber_mensagens()` na linha 06. Caso contrário, continua normalmente o seu processamento. O pseudo-código dessa função pode ser visto na figura 5.3.

O laço `while` é executado enquanto houverem mensagens a ser recebidas. Somente

```
receber_mensagens()
{
    Boolean flag;
    Célula  K;

    while(true) {  // O laço é executado enquanto houver mensagens
        K = MPI_Recv();
        if(NOT Busca(Não_Expandido, K) AND NOT Busca(Expandido, K))
            Insere(Não_Expandido, K);

        flag = MPI_Iprobe();
        if(NOT flag)
            break;
    }
}
```

Figura 5.3: Pseudo-código da função para o tratamento de mensagens recebidas

uma mensagem é recebida a cada iteração do loop. Cada uma delas contém uma célula da fronteira de uma das partições e é recebida, no processo destino, pela função `MPI_Recv()`. A célula recebida é testada, como no algoritmo serial, para saber se já está contida em um dos conjuntos. Após o teste, a função `MPI_Iprobe()` é chamada para verificar se existem mais mensagens esperando. Tendo recebido todas as mensagens, o algoritmo volta a executar o laço principal.

5.3.4 Laço Principal

O laço principal do algoritmo, excluindo-se a parte de troca de mensagens e sua condição de parada, é praticamente igual à versão serial. Na sua primeira parte, vista nas linhas 05 e 06 da figura 5.2, ele recebe as mensagens como explicado na seção anterior.

Depois, na linha 07, uma célula `C` é retirada do conjunto `Não_Expandido` e sofre o processo de expansão realizado pelo laço `for` das linhas 08 a 17. Dentro do `for`, são

encontradas células vizinhas a atual que são atravessadas pelo diagrama de Voronoi. Cada uma dessas células, no laço chamadas de `K`, são testadas para saber se pertencem a partição local ou a partição de um dos processadores vizinhos.

No caso de pertencerem a partição local, o algoritmo testa se `K` está ou não contida em um dos dois conjuntos e, se não estiver, a insere em `Não_Expandido`. Se `K` estiver na partição de um processador vizinho, uma mensagem é enviada para ele indicando o fato.

Quando a expansão de `C` termina, ela é inserida no conjunto `Expandido`. Após a inserção, é calculada a condição de parada do algoritmo pela função `terminar()`. Quando todos os processadores tiverem seus conjuntos `Não_Expandido` vazios, o algoritmo termina e o diagrama de Voronoi está completamente determinado.

5.4 Resultados

A seguir, serão mostrados exemplos do diagrama calculado pelo algoritmo 3D paralelo. Na figura 5.4, pode-se ver uma área de trabalho com três esferas como obstáculos. Foram utilizados dois elementos processadores no cálculo. As partições podem ser vistas pela coloração diferente das células dos lados esquerdo e direito da área de trabalho.

Na figura 5.5 pode-se ver o mesmo diagrama calculado por seis processadores. As partições de cada um deles também é notada pela diferença na cor das células.

Outro exemplo de diagrama pode ser visto na figura 5.6. Nela oito esferas se encontram presentes na área de trabalho. Esse diagrama foi calculado utilizando quatro processadores.

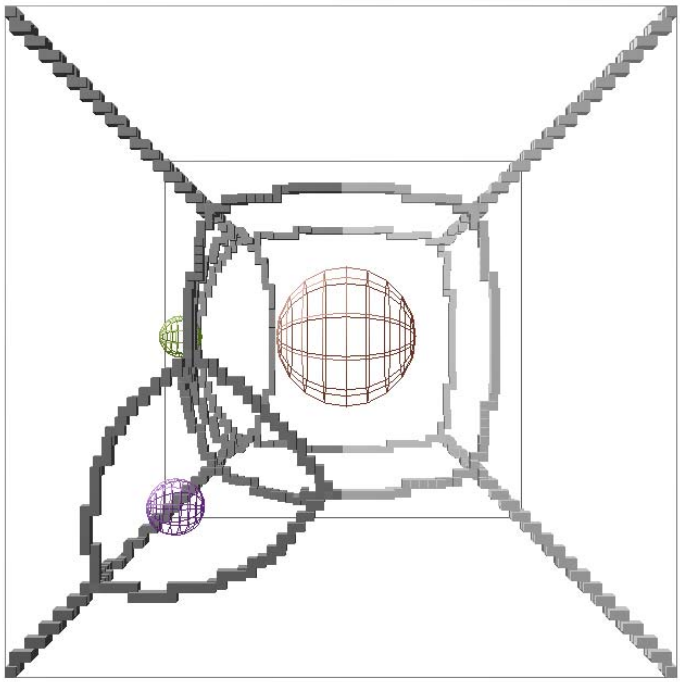

Figura 5.4: Configuração com três obstáculos calculada por dois processadores

	$n = 1024$	$n = 2048$	$n = 4096$
$P = 1$	1.000	1.000	1.000
$P = 2$	1.626	2.185	2.936
$P = 4$	0.845	2.075	5.441

Tabela 5.1: Resultados

	$n = 1024$	$n = 2048$	$n = 4096$
$P = 1$	1.000	1.000	1.000
$P = 2$	3.107	2.773	2.400
$P = 4$	4.767	8.791	6.700

Tabela 5.2: Resultados

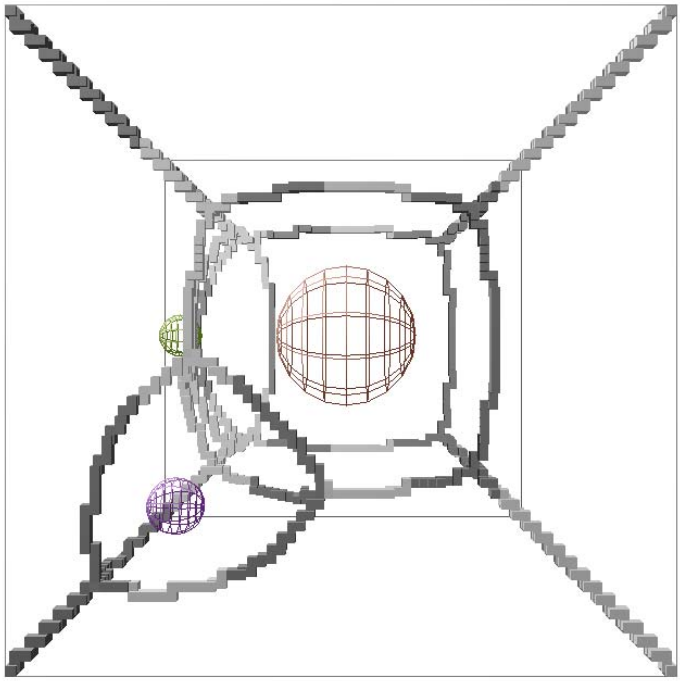

Figura 5.5: Configuração com três obstáculos calculada por seis processadores

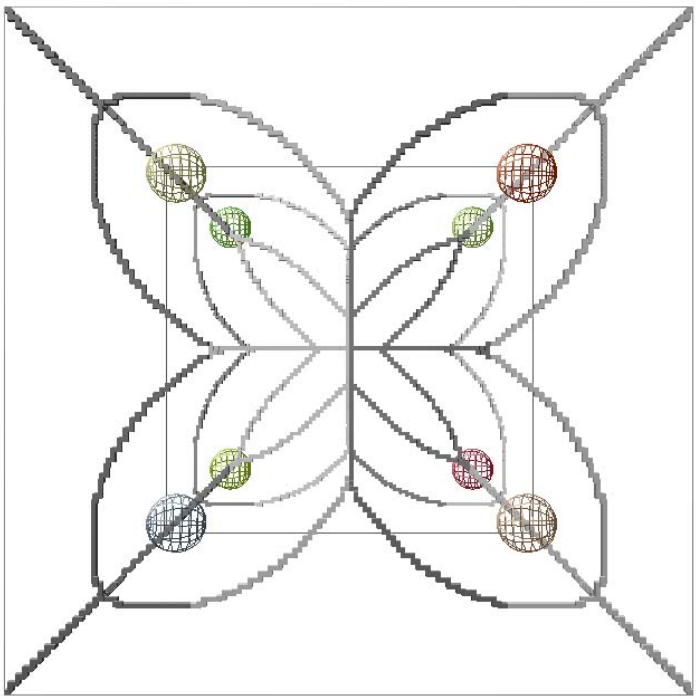

Figura 5.6: Configuração com oito obstáculos calculada por quatro processadores

6 Simulador

Neste capítulo é apresentado o simulador VrnSim para exploração de ambientes perigosos utilizando diagramas de Voronoi. Na seção 6.1 são apresentadas as principais características desse simulador.

Nas seções 6.2, 6.3 e 6.5 são apresentados os módulos principais que compõem a estrutura do simulador.

A parte gráfica do simulador, que mostra uma imagem tridimensional do estado da simulação, foi desenvolvida utilizando a biblioteca OpenGL, que será apresentada na seção 6.4.

Finalmente, na seção 6.6, são apresentados os resultados para um exemplo de ambiente explorado por um time de robôs.

6.1 Características do Simulador

O simulador VrnSim permite a um usuário estudar o comportamento de um time de robôs cuja tarefa é determinar cooperativamente, em tempo real, o traçado de um caminho seguro baseado no diagrama de Voronoi. Esse tipo de caminho é necessário para aplicações onde os obstáculos representam perigo para os robôs e é interessante mantê-los eqüidistante a eles. O simulador modela a comunicação entre os robôs por rede sem fio

Como entrada do simulador o usuário fornece a configuração dos obstáculos presentes no ambiente, o número de células por unidade de volume, a heuristíca que rege a exploração e a quantidade de robôs no time.

Durante a simulação, estão disponíveis para o usuário estatísticas sobre os robôs do time, tais como eficiência da exploração e número de células exploradas, entre outras.

O software foi desenvolvido utilizando linguagem C e as bibliotecas MPI para comunicação e a biblioteca OpenGL (vide seção 6.4) para visualização. Como compilador, no sistema operacional Windows XP, foi utilizado o Microsoft Visual Studio.NET 2005. Já no sistema operacional Linux, foi utilizado o compilador GCC[37]. A implementação da biblioteca MPI utilizada em ambos foi a MPICH[34].

6.2 Estrutura do Simulador

O simulador está estruturado de forma modular e consiste de dois módulos principais: o Robô e o Observador. O módulo Observador coleta, com uma interferência mínima, dados referentes a simulação. O módulo Robô, que é replicado para os vários robôs do time, faz com que o robô ande pelo grid e implementa o comportamento de cada um deles, desde o planejamento de trajetória até as tomadas de decisão.

Na figura 6.1 pode ser vista a configuração do sistema. Os vários módulos Robô representam o time cooperativo. Quando um robô envia uma mensagem para outro, uma cópia é enviada para o Observador. Esta é a única interferência deste módulo no sistema, o que torna esta estrutura eficiente para a utilização em uma implementação de um time de robôs cooperativos em tempo-real.

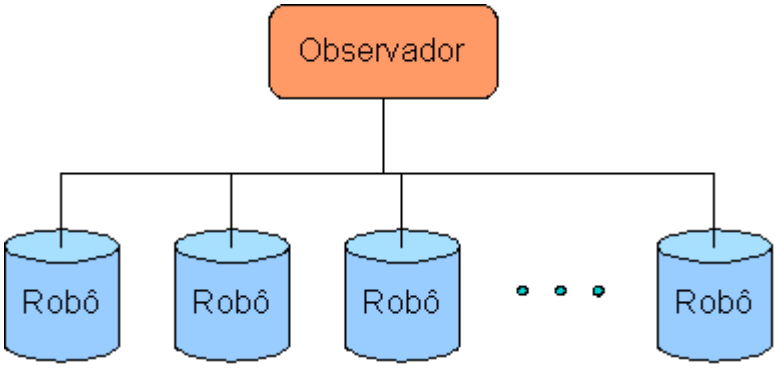

Figura 6.1: Estrutura do Simulador

6.3 Módulo Observador

O módulo Observador é responsável por apresentar, de forma gráfica tridimensional, na tela o estado da simulação. O usuário pode interagir em tempo real com a simulação, modificando o ângulo de visão e solicitando estatísticas ao sistema.

A lista a seguir mostra as estatísticas disponíveis durante a simulação:

- C_{total}: Número de células total do diagrama de Voronoi até o momento
- V_{total}: Número de vértices do diagrama de Voronoi até o momento
- E: Eficiência da exploração realizada pelo time
- C_i: Número de novas células exploradas pelo robô i
- V_i: Número de novos vértices descobertos pelo robô i
- R_i: Número de células redundantes percorridas pelo robô i

Quando o robô passa por um caminho que já foi previamente explorado, ele está realizando trabalho redundante. Esse trabalho redundante é função do número de células

```
Módulo_Observador()
{
    while(simulação não terminar) {
        recebe_mensagens_do_time();
        desenha_tela();
        mostra_estatísticas();
    }
}
```

Figura 6.2: Pseudo-código do módulo Observador

redundantes e afeta diretamente a eficiência da exploração. A eficiência da exploração é a razão entre o número de células novas exploradas pelo número de células percorridas. Este último, o número de células percorridas, é o somatório do número de células novas com o número de células redundantes, ou seja:

$$C_{total} = \sum_{i=1}^{n} C_i$$

$$E = \frac{C_{total}}{C_{total} + \sum_{i=1}^{n} R_i}$$

Onde n é o número de robôs no time.

Na figura 6.2, pode-se ver o pseudo-código do módulo observador. Nele a cada iteração do laço, o robô recebe as mensagens enviadas pelo time, desenha na tela o estado da simulação utilizando a biblioteca OpenGL, apresentada na próxima seção, e mostra as estatísticas da simulação até o presente instante.

6.4 OpenGL

6.4.1 Introdução

OpenGL é uma interface de software para utilização em hardware gráficos. Essa interface é composta por cerca de 150 funções distintas usadas para especificar objetos e operações necessárias para produzir aplicações 3D interativas.

A biblioteca foi desenvolvida para ser independente do hardware e, assim, portável para várias plataformas de hardware. Para conseguir esses objetivos, não foi incluso nenhum comando para controle de janelas ou tratamento de entrada de dados. O programador deve utilizar as facilidades do sistema operacional onde a aplicação final será compilada e utilizada.

Em computação gráfica, uma *cena* é composta por objetos, luzes e a posição do observador. Cada objeto que compõe uma cena pode ser composto por várias primitivas gráficas. Um exemplo de cena criado com OpenGL, tirado de [38] pode ser visto na figura 6.3.

Com a OpenGL, o programador precisa construir os objetos que deseja apresentar na cena usando um pequeno conjunto de primitivas gráficas: pontos, linhas e polígonos. Para cada primitiva pode ser especificada uma cor ou essa cor pode ser automaticamente determinada pela iluminação do ambiente. Com a imagem na tela, ele pode realizar transformações lineares tais como rotações e translações nos objetos desenhados para mudar o ponto de vista do observador.

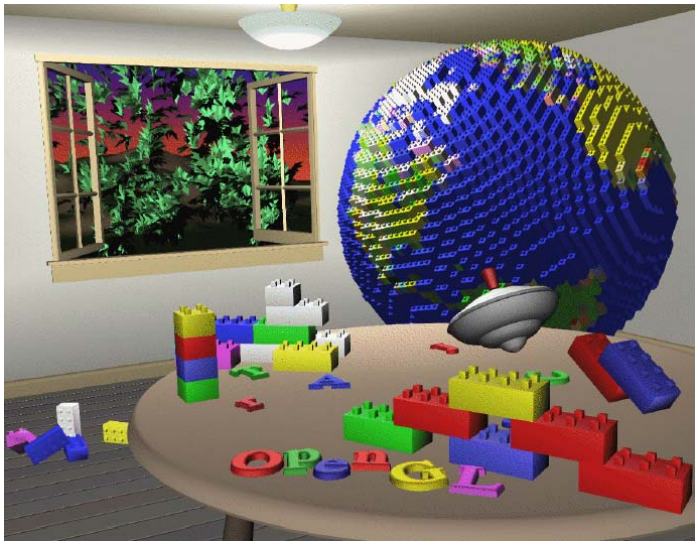

Figura 6.3: Exemplo de cena 3D

6.4.2 A máquina de estados da OpenGL

A OpenGL é, basicamente, uma máquina de estados. O programador a coloca em vários desses estados, e eles ficam ativos até que a máquina receba um comando para trocar de estado.

Um exemplo de variável de estado é a cor de desenho. Quando uma cor é selecionada, todas as primitivas gráficas são desenhadas com elas até que um comando mude a cor.

A cor atual é somente uma das muitas variáveis de estado que a OpenGL controla. Outras controlam, por exemplo, o modo de desenho dos polígonos, característica das luzes de uma cena, transformações de projeção, etc. Muitas variáveis de estado são habilitadas e desabilitadas utilizando os comandos `glEnable()` e `glDisable()`.

6.4.3 GLUT

Como já foi dito, a OpenGL contém comandos para desenho de cenas, mas foi desenvolvida para ser independente do sistema de janela de qualquer sistema operacional. Conseqüentemente não há comandos para lidar com teclado e mouse.

Para lidar com a criação e manutenção das janelas necessárias para desenhar uma cena, além de lidar com a entrada de dados, é utilizada a biblioteca *GLUT* (*OpenGL Utility Toolkit*).

Além de lidar com janelas e entradas de dados, a GLUT ainda possui funções para desenho de objetos 3D mais complicados, tais como esferas, toróides e cubos.

6.4.4 Projeção Perspectiva

Para mostrar uma cena 3D construída em OpenGL na tela do computador é necessário obter um *volume de visualização*. Esse volume determina como os objetos serão projetados na tela. Outra função desse volume é determinar quais objetos ou suas partes estarão ou não presentes na imagem final.

A forma definida pelo volume de visualização e utilizada para projetar os objetos no presente trabalho é a projeção perspectiva. A característica mais marcante dessa projeção é que quanto mais próximo da tela, maior o objeto. Quanto mais distante, menor.

Esse fato ocorre porque o volume de visualização da projeção perspectiva é um tronco de pirâmide. Nesse caso, um tronco de pirâmide é uma pirâmide cujo topo foi cortado por um plano paralelo à sua base. Objetos que se encontram dentro do volume de visualização são projetados em direção ao vértice superior da pirâmide, onde está localizado o espectador.

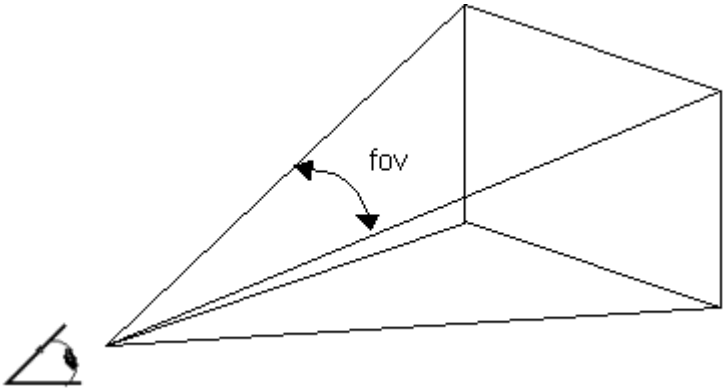

Figura 6.4: Parâmetro fov de gluPerspective()

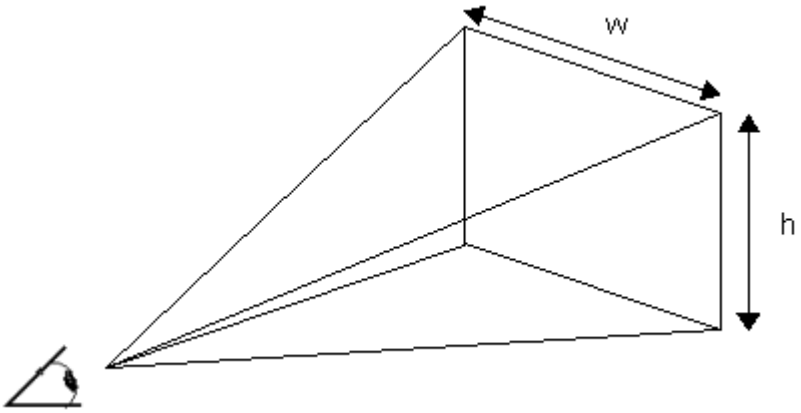

Figura 6.5: w e h para o cálculo do parâmetro aspect

Para definir o volume de visualização da projeção perspectiva é utilizada a seguinte função:

```
gluPerspective(fov, aspect, near, far)
```

O parâmetro `fov` do inglês *Field Of View* é o ângulo do campo de visão no plano dos eixos x-z, como visto na figura 6.4; seu valor deve ser no intervalo $[0.0, 180.0]$ graus.

Já o parâmetro `aspect` é dado pela largura (w) do tronco da pirâmide dividida pela sua altura (h). Na figura 6.5 pode ser visto o significado desses dois valores.

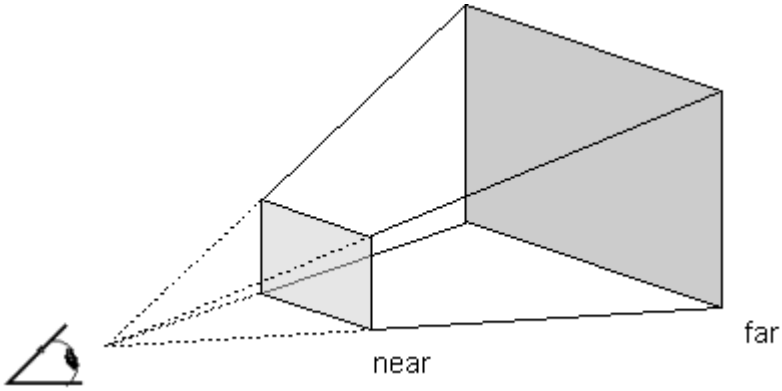

Figura 6.6: Planos de corte do tronco da pirâmide

Finalmente, os parâmetros `near` e `far` são as distâncias entre o espectador e os planos que definem o topo e o fundo do tronco da pirâmide. Esses valores devem sempre ser positivos. Na figura 6.6 podem ser vistos os planos `near` e `far`.

A projeção perspectiva é muito utilizada em simulações visuais, animação e outras aplicações que necessitam de um certo grau de realismo. A impressão de realismo vem da semelhança entre essa projeção e a forma como o olho humano e uma câmera funcionam.

6.4.5 Primitivas Gráficas

Cada objeto geométrico que compõe uma cena em OpenGL é descrito por uma série de coordenadas de vértices. Cada vértice é especificado com a função `glVertex3f()` ou alguma de suas variantes. Os parâmetros dessa função são as coordenadas x, y e z do vértice que se deseja criar.

O desenho dos objetos é feito em dois passos. O primeiro é escolher o tipo de primitiva gráfica que se deseja usar utilizando a função `glBegin()`. No segundo, todos os vértices do objeto são criados pela função `glVertex3f()`.

As várias constantes que representam as primitivas gráficas disponíveis na biblioteca

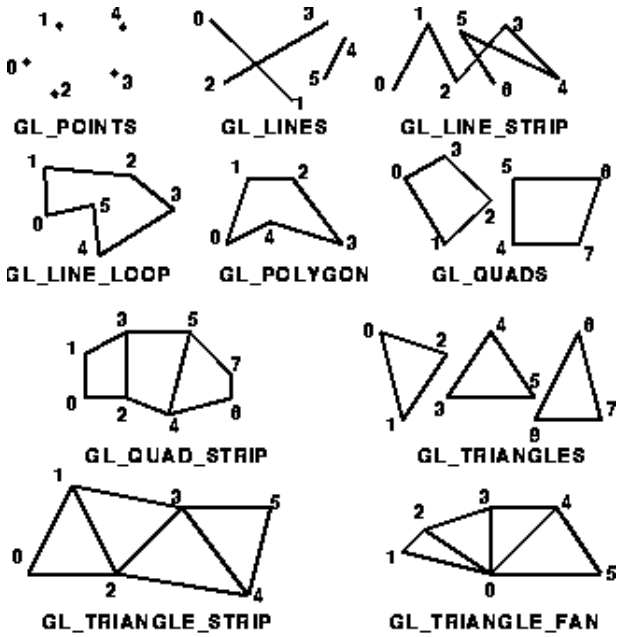

Figura 6.7: Primitivas gráficas em OpenGL

OpenGL podem ser vistas na figura 6.7. Os números na figura representam os vértices e a ordem em que foram criados. A seguir, são mostradas as descrições de cada uma das primitivas.

- `GL_POINTS` são pontos individuais
- `GL_LINES` cada par de vértices é interpretado como um segmento de linha
- `GL_LINE_STRIP` séries de segmentos de linha conectados
- `GL_LINE_LOOP` mesmo do acima, só que com um segmento do último vértice para o primeiro
- `GL_POLYGON` vértices formam um polígono fechado
- `GL_QUADS` a cada quatro vértices, um polígono de quatro lados

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 3.0, 0.0);
    glVertex3f(4.0, 3.0, 1.0);
    glVertex3f(6.0, 1.5, 1.0);
    glVertex3f(4.0, 0.0, 2.0);
glEnd();
```

Figura 6.8: Exemplo de código para o desenho de um polígono

- `GL_QUAD_STRIP` polígono de quatro lados com um lado em comum
- `GL_TRIANGLES` a cada três vértices, um triângulo
- `GL_TRIANGLE_STRIP` triângulos com um lado em comum
- `GL_TRIANGLE_FAN` triângulos com um vértice em comum

Na figura 6.8 pode-se ver um exemplo de código para desenho de um polígono com cinco vértices. O parâmetro `GL_POLYGON` na chamada da função `glBegin()` especifica que a próxima primitiva a ser desenhada será um polígono. Depois, as cinco chamadas a `glVertex3f()` criam os seus vértices. Para avisar a OpenGL que o polígono terminou de ser criado, a função `glEnd()` é chamada.

6.4.6 Transformações

A biblioteca OpenGL fornece funções para transladar e rotacionar os objetos de uma cena. A sintaxe das chamadas as duas funções responsáveis por essas transformações são, respectivamente:

```
glTranslatef(x, y, z)
```

```
glRotatef(angle, x, y, z)
```

Na primeira, os três argumentos indicam quantas posições em cada um dos três eixos o objeto vai se movimentar. Já na segunda, o parâmetro `angle` permite que um ângulo em graus da rotação seja especificado. A rotação irá ocorrer `angle` graus no sentido anti-horário em relação ao vetor indicado pelos parâmetros `x`, `y` e `z`.

6.5 Módulo Robô

O time de robôs cooperativos é formado pela interação entre vários módulos do tipo Robô executando paralelamente. Cada um deles é um processo separado, comunicando-se pela biblioteca MPI. Juntos eles realizam a exploração do ambiente determinado pelo usuário. Cada um desses módulos é responsável pelo planejamento de trajetória e pela tomada de decisão individual de cada membro do time. Além disso, esse módulo também é responsável pela cooperação entre eles. A exploração funciona como descrito na próxima seção.

6.5.1 Exploração do Ambiente

Quando um robô vai explorando o ambiente pelo diagrama de Voronoi, ele vai encontrando vértices do diagrama. Com esses vértices ele vai montando um grafo. Ao sair de um vértice e chegar em outro, o robô cria uma aresta nesse grafo e armazena na sua estrutura o comprimento dessa aresta, para quando necessário utilizá-lo na melhor escolha da próxima aresta não explorada a ser perseguida.

Um robô começa a exploração do ambiente partindo de uma célula do canto do grid. Esta célula inicial, além de todas as outras células dos cantos do grid, são tratadas, especialmente, como vértices do diagrama. O robô consegue determinar, utilizando as mesmas

técnicas do algoritmo numérico 3D serial, quais faces desse vértice são atravessadas pelo diagrama, ou seja, para quais células as arestas do diagrama saindo daquele vértice seguem. Essas arestas, que ainda não foram percorridas por nenhum robô são chamadas de arestas *não exploradas*, pois sabe-se da sua existência mas não do seu destino, ou seja, em qual vértice ela chegará. O objetivo da exploração do ambiente é eliminar todas as arestas não exploradas.

Ao chegar em um vértice, o robô verifica no seu grafo se este vértice já foi ou não descoberto antes. No caso de não ter sido descoberto antes, o robô adiciona esse vértice ao seu grafo. Ele também adiciona no grafo uma aresta ligando esse vértice novo ao último vértice onde esteve.

Todas as arestas desse novo vértice, menos a qual veio o robô, são não exploradas. Nesse caso, o robô, por meio de uma heurística, escolhe uma dessas arestas não exploradas para seguir e continua a exploração. Nesse ponto, o robô envia uma mensagem para os outros membros do time avisando-os sobre o novo vértice e a nova aresta a ser adicionada no grafo. O robô também indica para o time qual aresta não explorada ele irá seguir. Isto é necessário para evitar que dois robôs explorem uma mesma aresta realizando assim trabalho redundante. Quando um robô recebe uma mensagem desse tipo, ele faz as devidas modificações no seu grafo.

Se o vértice no qual o robô chegou já tinha sido descoberto antes, ele verifica a existência de arestas não exploradas. Existindo mais de uma, ele usa uma heurística para decidir qual delas seguir e envia uma mensagem para os outros membros do time avisando. No caso de não existir nenhuma aresta não explorada, o robô descobre, por meio do algoritmo de Dijkstra, qual é o vértice mais próximo dele com arestas não exploradas. Ao aplicar esse algoritmo, o robô descobre, na forma de uma seqüência de vértices e arestas, o menor caminho para chegar no destino. O algoritmo de Dijkstra será visto na seção seguinte.

Quando o robô chega em um vértice já descoberto e não encontra mais nenhuma aresta não explorada no grafo, a exploração do ambiente está terminada.

6.5.2 Algoritmo de Dijkstra

6.5.2.1 Grafos

Um *grafo* $G(V, E)$ é composto por um conjunto finito não-vazio V e por um conjunto E de pares não-ordenados de elementos distintos de V. Os elementos de V são os *vértices* e os de E são as *arestas* de G respectivamente. Cada aresta $e \in E$ será denotada pelo par de vértices $e = (v, u)$. Neste caso, os vértices v, u são os extremos da aresta e, sendo denominados adjacentes. A aresta e é dita incidente a ambos v, u. Duas arestas que possuem um extremo comum são chamadas de adjacentes.

Pode-se visualizar um grafo através de uma representação geométrica, na qual seus vértices correspondem a pontos distintos do plano em posições arbitrárias, enquanto que a cada aresta (v, u) é associada uma linha arbitrária unindo os pontos correspondentes a v, u.

Cada uma das arestas que compõem o grafo pode ter associada a si um valor chamado *peso*. O peso é uma função $w(u, v)$ tal que $w : E \rightarrow \mathbb{R}$.

6.5.2.2 Algoritmo

O Algoritmo de Dijkstra [39], batizado em homenagem ao cientista que o propôs, Edsger Dijkstra, é um algoritmo que resolve o problema do caminho mínimo com uma única origem para um grafo com arestas que possuam pesos não negativos.

Por exemplo, se os vértices do grafo representarem cidades e o peso das arestas as distâncias rodoviárias entre pares de cidades conectados por estradas, o algoritmo de

Dijkstra pode ser usado para encontrar a menor rota entre duas cidades.

A entrada do algoritmo consiste de um grafo $G(V, E)$ e de s, o vértice de origem pertencente ao conjunto V de G. Todas as arestas de G possuem pesos não negativos, ou seja tem uma função $w : E \rightarrow [0, \infty]$. A função $w(u, v)$ pode ser vista como o custo não negativo de se mover do vértice u para o vértice v, ou seja, a distância entre esses dois vértices. O custo de um caminho p passando pelos vértices $v_0, v_1, \ldots, v_k$ é a soma dos pesos das arestas que compõem esse caminho:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Para um dado par de vértices s e t em V, o algoritmo encontra o caminho de s para t que tenha o menor custo. Ele também pode ser usado para encontrar os custos do menor caminho de um vértice de origem para todos os outros vértices no grafo.

6.5.2.3 Descrição do Algoritmo

O algoritmo funciona mantendo, para cada vértice v, o custo $d[v]$ do menor caminho encontrado até momento entre s e v. Inicialmente, esse valor é zero para o vértice de origem s ($d[s] = 0$), e infinito para todos os outros vértices, já que ainda não se conhece nenhum caminho indo para esses vértices ($d[v] = \infty, \forall\ v \in V - \{s\}$). Quando o algoritmo termina, $d[v]$ será o custo do menor caminho de s para v, ou infinito, se não existir um caminho.

A operação básica realizada pelo algoritmo de Dijkstra é o relaxamento das arestas: se existe uma aresta de u para v, então o menor caminho conhecido de s para u ($d[u]$) pode ser estendido para um caminho de s para v simplesmente adicionando a aresta (u, v) no seu fim. Esse caminho terá comprimento $d[u] + w(u, v)$. Se esse novo valor for menor

que o valor atual de $d[v]$, o valor atual de $d[v]$ é substituido por ele.

O relaxamento das arestas é aplicado até todos os valores $d[v]$ representarem o custo do menor caminho de s até v. O algoritmo foi desenvolvido de forma que cada aresta (u, v) é relaxada somente uma vez, quando $d[u]$ atinge seu valor final.

O algoritmo mantém dois conjuntos de vértices S e Q. O conjunto S contém todos os vértices para os quais sabe-se que o valor de $d[v]$ é o custo final do menor caminho e conjunto Q contém todos os outros vértices. O conjunto S começa vazio e, a cada passo do algoritmo, um vértice é movido de Q para S. O vértice escolhido para mudar de conjunto possui o menor valor $d[u]$ de todo o conjunto Q. Quando um vértice u é movido para S, o algoritmo relaxa cada uma de suas arestas (u, v).

6.5.2.4 Pseudo-código do Algoritmo

No pseudo-código mostrado na figura 6.9, a função `u = Min_D(Q)` retorna o vértice u com menor valor $d[u]$ dentro do conjunto Q. O vértice retornado é removido de Q e retornado ao usuário.

Se o único interesse é encontrar o menor caminho entre dois vértices s e t, a busca pode ser terminada se o vértice u retornado por `Min_D(Q)` for igual a t.

Para saber quais vértices compõem o caminho de s para t basta utilizar os valores de `previous[]` obtidos após a execução do algoritmo. O pseudo-código para essa tarefa pode ser visto na figura 6.10, onde a seqüência L é a lista dos vértices contidos no menor caminho entre s e t.

```
Dijkstra(G, w, s)
{
    // G: grafo; s: vértice de origem;
    // w: função do peso; d[]: custo;
    // previous[]: vértice anterior no caminho;
    // S: conjunto de vértices com d[] determinado
    // Q: conjunto com os outros vértices

    // Inicialização
    for(cada vértice v em V[G])
        d[v] = Infinito;
        previous[v] = NULL;

    d[s] = 0;
    S = Vazio;
    Q = Todos os vértices de G;
    while(Q não é Vazio)
        // Min_D(Q): retorna vértice com menor custo em Q
        u = Min_D(Q);
        S = S união {u};
        for(cada aresta (u,v) saindo de u)
            if(d[v] > d[u] + w(u,v))   // Relaxe (u,v)
                d[v] = d[u] + w(u,v);
                previous[v] = u;
}
```

Figura 6.9: Pseudo-código do algoritmo de Dijkstra

6.5.2.5 Implementação

Na implementação, os vértices do grafo são armazenados utilizando uma lista encadeada. Cada elemento da lista representa um vértice e contém as informações sobre cada uma de suas arestas. Essas informações indicam o peso da aresta e o vértice sobre o qual ela incide.

A opção pela lista encadeada como estrutura se deve ao desconhecimento total do grafo

```
Caminho(t, previous[])
{
    // L: lista com os vértices do menor caminho de s a t
    // u: variável de iteração no laço
    // previous[]: vért. anterior calculado em dijkstra()

    L = Lista vazia;
    u = t;
    while(u not NULL)
        insere u no começo de L;
        u = previous[u];
}
```

Figura 6.10: Encontrando a seqüência de vértices de s para t

quando a simulação começa. Os robôs vão construindo dinamicamente o grafo conforme vão explorando a área de trabalho. Se fosse utilizada uma matriz de adjacência ao invés da lista encadeada, a cada novo vértice descoberto seria necessário criar uma nova matriz na memória, copiar todo o conteúdo da antiga matriz para a nova área e adicionar as informações do novo vértice. Já com a lista, basta alocar um novo elemento e inserir suas informações.

A indicação de que um vértice pertence ao conjunto S ou ao conjunto Q é dada por um valor *booleano* presente em cada elemento da lista onde estão armazenados todos os vértices do grafo. A função `Min_D()` é implementada como uma busca linear nessa lista buscando somente os elementos pertencentes ao conjunto Q.

Devido a essa forma de implementação, o algoritmo de Dijkstra possui tempo de execução de $O(n^2)$. Sendo n o número de vértices no grafo.

6.5.3 Máquina de Estados do Módulo Robô

O módulo Robô funciona, basicamente, na forma de uma máquina de estados. Essa máquina possui três estados possíveis: Aresta, Vértice e Concluído. O robô atinge o estado Concluído quando não existe mais nenhuma aresta não explorada no grafo, ou seja, a exploração do ambiente terminou.

Quando o robô se encontra no estado Aresta, ele está em uma célula que pertence a uma aresta do diagrama de Voronoi. Em uma célula de aresta, somente duas células vizinhas pertencem ao diagrama. O trabalho do robô nesse estado é, sabendo a célula em que o robô estava no passo anterior, ir para a outra célula. O robô continua nesse estado até chegar em uma célula que possui um vértice do diagrama. No estado Aresta, o robô não envia mensagens para nenhum outro componente do time, embora consiga receber mensagens destinadas a ele.

Ao chegar em uma célula vértice, o robô muda para o estado Vértice. Nesse estado, ele realiza o comportamento descrito na seção 6.5.1, ou seja, realiza as devidas modificações no seu grafo, envia informações para os outros membros do time sobre suas descobertas e escolhe a próxima aresta a ser explorada.

6.5.4 Heurísticas

Quando o robô chega em um vértice que possui mais de uma aresta não explorada, ele utiliza uma heurística para decidir que aresta seguir para continuar a exploração. No simulador foram implementadas as seguintes heurísticas:

Aleatória Escolhe aleatoriamente uma aresta para ser seguida pelo diagrama.

Vértices Visíveis Determina quantos vértices do grafo são visíveis da face da aresta não

explorada. Depois, escolhe seguir a aresta cuja face consegue ver o menor número de vértices, o que significa andar na direção de uma região menos explorada. O teste de visibilidade é realizado utilizando o produto escalar do vetor normal à face atravessada pela aresta de Voronoi e o vetor com origem nesta face e destino no vértice cuja visibilidade está sendo testada.

Antes de iniciar a simulação é permitido ao usuário escolher qual das heurísticas o time de robô ira utilizar.

6.5.5 Pseudo-código do Algoritmo de Exploração

Cada robô do time cooperativo é implementado pelo módulo Robô. Nesse módulo está presente o código para realizar a exploração do ambiente. Como já dito, esse módulo funciona como uma máquina de estados. Seu pseudo-código pode ser visto na figura 6.11.

Uma função externa chama a função `Módulo_Robô()` de tempos em tempos. Nela, a primeira tarefa executada pelo robô é receber as mensagens enviadas pelos outros membros do time, se houver alguma. Em seguida, uma ação baseada no estado atual do robô, representado pela variável `robo.estado`, é tomada. Se o robô estiver em um vértice do diagrama de Voronoi, ele chama a função `Robo_em_Vértice()`, que será mostrada mais adiante. No caso do estado atual ser Aresta, o robô decide a próxima célula para qual vai se movimentar. Já é conhecido o fato de que toda célula de uma aresta do diagrama possui somente duas células vizinhas pertencentes ao diagrama. Então, sabendo qual delas o robô esteve no passo anterior, é fácil determinar para qual ir no próximo passo.

Após determinado para qual célula ele deve ir, o robô se movimenta fisicamente utilizando a chamada a função `move_na_direção()`. Finalmente, já na nova célula, é chamada a função `descobre_próximo_estado()` para calcular o próximo estado do robô. O

```
Módulo_Robô()
{
    // Verifica se existem mensagens a serem recebidas
    receber_mensagens();

    if(robo.estado == VERTICE) {
        Robo_em_Vértice();
    }
    else if(robo.estado == ARESTA) {
        próxima = próxima célula da aresta;
        move_na_direção(próxima);
        descobre_próximo_estado();
    }
    else {
        // robô está no estado CONCLUÍDO
        imprimir("Exploração terminada!");
    }
}
```

Figura 6.11: Pseudo-código do módulo Robô

pseudo-código dessa função será mostrado mais adiante. O valor retornado por ela será utilizado na próxima chamada a `Módulo_Robô()`.

No caso do robô não se encontrar nem no estado Aresta, nem no estado Vértice, ele estará no estado Concluído, onde é determinado o fim da exploração.

Agora, retornando a explicação do estado Vértice, foi mencionada uma chamada a função `Robo_em_Vértice()`. O pseudo-código dessa função pode ser visto na figura 6.12. Nela o robô verifica se o vértice no qual se encontra possui ou não arestas não exploradas.

Se possui, o algoritmo utiliza a função `heurística()`, que implementa uma das heurísticas expostas na seção 6.5.4 para decidir qual das arestas seguir. Tendo escolhido uma, o robô marca essa aresta com a função `marca()` e envia uma mensagem para o time com o objetivo de evitar que outros robôs do time tentem explorar essa mesma aresta.

```
Robo_em_Vértice()
{
    if(vértice atual tem aresta não explorada) {
        aresta = heurística();
        // marca a aresta escolhida pela
        // heurística como explorada
        marcar(aresta);
        // avisa o time sobre a aresta escolhida
        enviar_mensagem(time, aresta);
    }
    else {   // não tem aresta não explorada
        if(ainda existe aresta não explorada no grafo) {
            // usa Dijkstra para encontrar proxima aresta
            aresta = Dijkstra();
        }
        else { // não existe mais nenhuma aresta não explorada
            robo.estado = CONCLUÍDO;
            return;
        }
    }
    move_na_direção(aresta);
    descobre_próximo_estado();
}
```

Figura 6.12: Pseudo-código da função Robo_em_Vértice()

No caso do vértice atual não possuir arestas não exploradas, o robô verifica no grafo se ainda existe alguma aresta a ser explorada em algum vértice. Se existe, o algoritmo de Dijkstra é chamado para encontrar o caminho mínimo para o vértice mais próximo do atual com arestas não exploradas. Se não existe mais nenhuma aresta não explorada no grafo, a função muda o estado do robô para Concluído e retorna.

Finalmente, depois de decidida qual aresta seguir, o robô se movimenta para a próxima célula chamando a função `move_na_direção()` e o próximo estado é calculado com a função `descobre_próximo_estado()`.

```
descobre_próximo_estado()
{
    if(célula atual é vértice) {
        if(novo vértice foi descoberto) {
            incluir_no_grafo(novo vértice);
            // avisa o time sobre a descoberta
            envia_mensagem(time, novo vértice);
        }
        robo.estado = VÉRTICE;
    }
    else {
        robo.estado = ARESTA;
    }
}
```

Figura 6.13: Pseudo-código da função descobre_próximo_estado()

O pseudo-código da função `descobre_próximo_estado()` pode ser visto na figura 6.13. Nela, o robô verifica se a célula na qual se encontra é um vértice ou uma aresta. No caso de ser aresta, o estado do robô é modificado para Aresta e a função retorna.

Se a célula na qual o robô se encontra for um vértice, ele verifica no grafo se este já foi encontrado antes ou se acabou de ser descoberto. Nesse caso, o vértice novo é incluído no grafo do robô. O time é avisado do descobrimento do novo vértice por uma mensagem enviada pela função `envia_mensagem()`. Finalmente, o estado do robô se torna Vértice e a função retorna.

6.6 Resultados

Nesta seção é apresentado um experimento simulado da exploração de um ambiente por um time de robôs cooperativos. O time é composto por oito robôs, todos inicialmente localizados nos cantos do grid. Para essa simulação, foram utilizadas quatro estações de

trabalho. Uma delas executa o módulo Observador e dois módulos Robô. As outras três, executam dois módulos Robô cada.

A exploração, logo após o seu início, pode ser vista na figura 6.14. Depois, os robôs exploram a área de trabalho como pode ser visto nas figuras 6.15 a 6.19. O resultado final é apresentado na figura 6.20.

As estatísticas apresentados pelo simulador após o término da exploração são mostradas na figura 6.21. O diagrama da exploração contem 1133 células, sendo 23 delas vértices. A exploração teve uma eficiência de 51.69% e levou 274 passos de simulação para terminar. Na tabela com a estatísticas individuais de cada robô as colunas vistas são: NC (novas células), NV (novos vértices) e Red (trabalho redundante).

A exploração desse mesmo ambiente com um time de dois robôs consegue uma eficiência de 62.38%, mas, em compesação, leva 913 passos de simulação. Com quatro robôs obtem-se eficiência de 69.33% e gasta-se 410 passos de simulação.

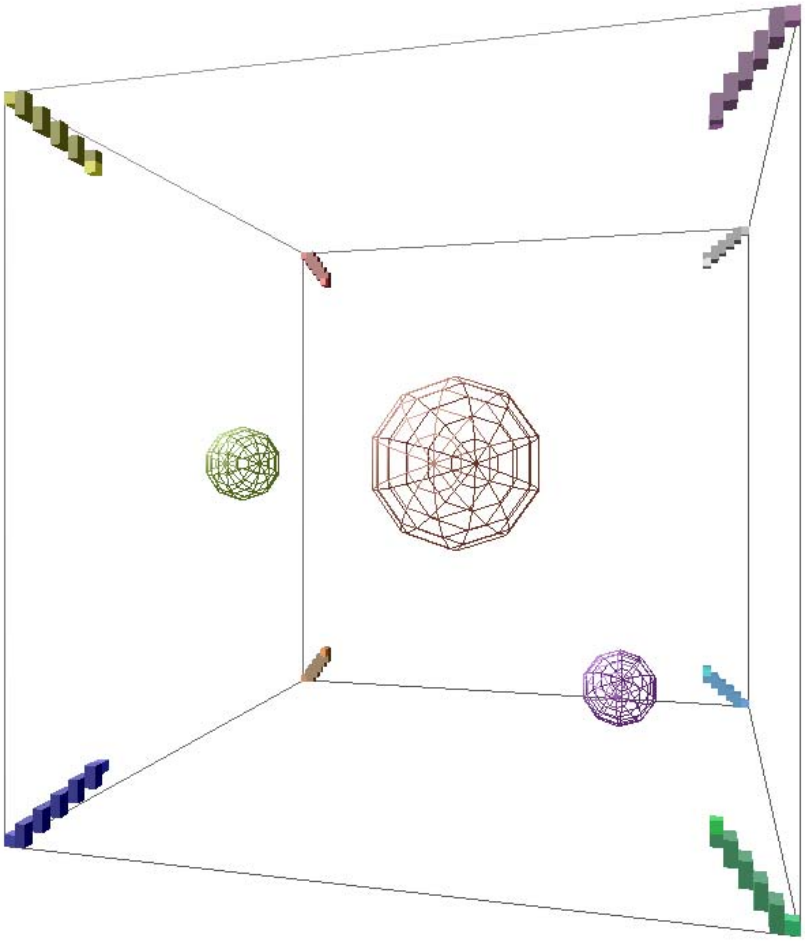

Figura 6.14: Logo após o passo inicial da simulação

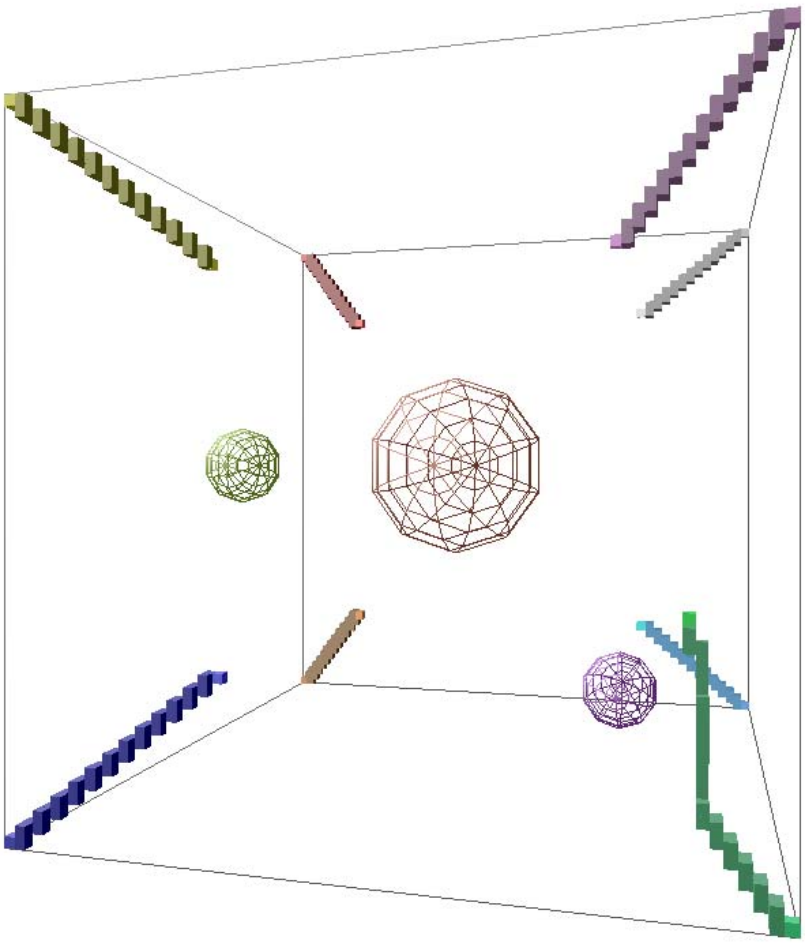

Figura 6.15: Time de robôs explorando a área de trabalho (parte 1)

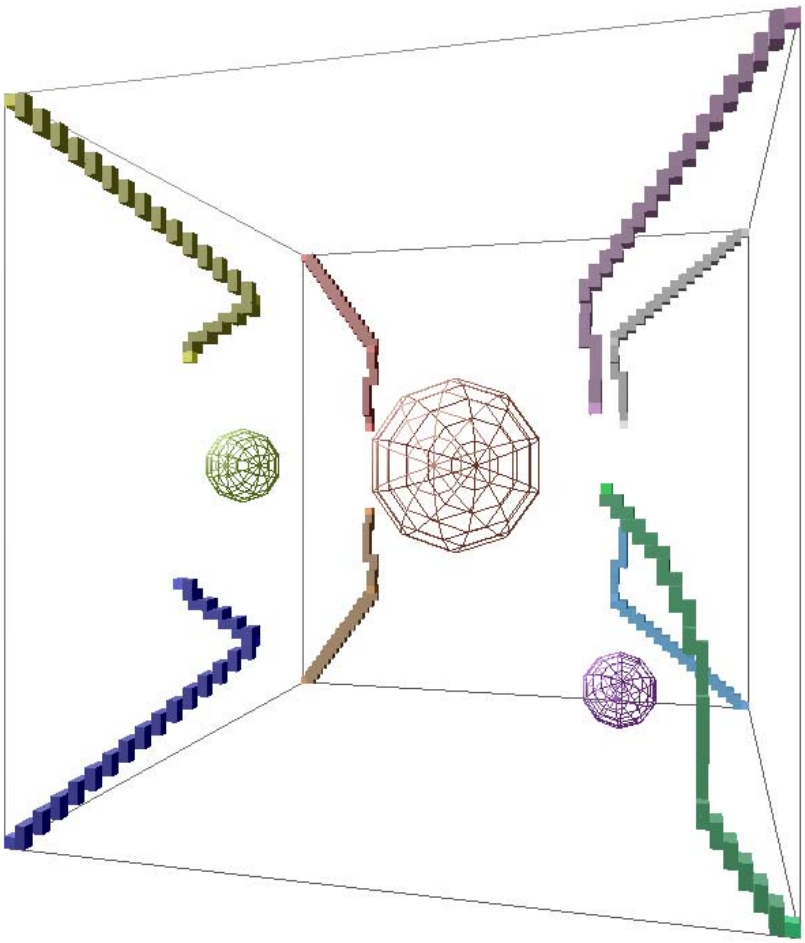

Figura 6.16: Time de robôs explorando a área de trabalho (parte 2)

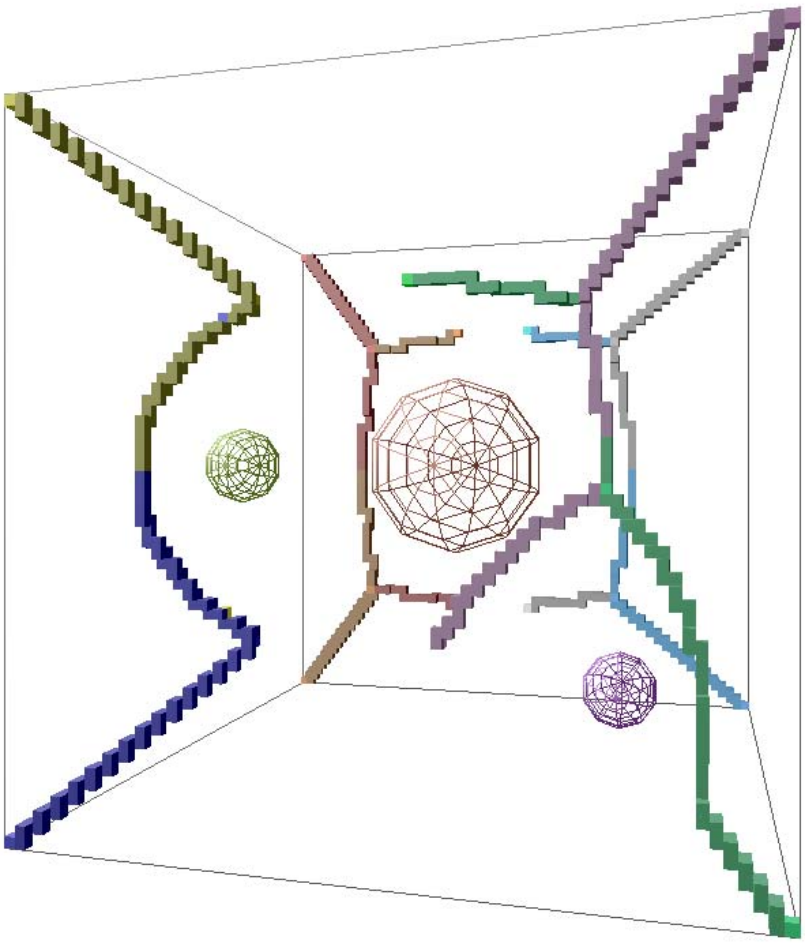

Figura 6.17: Time de robôs explorando a área de trabalho (parte 3)

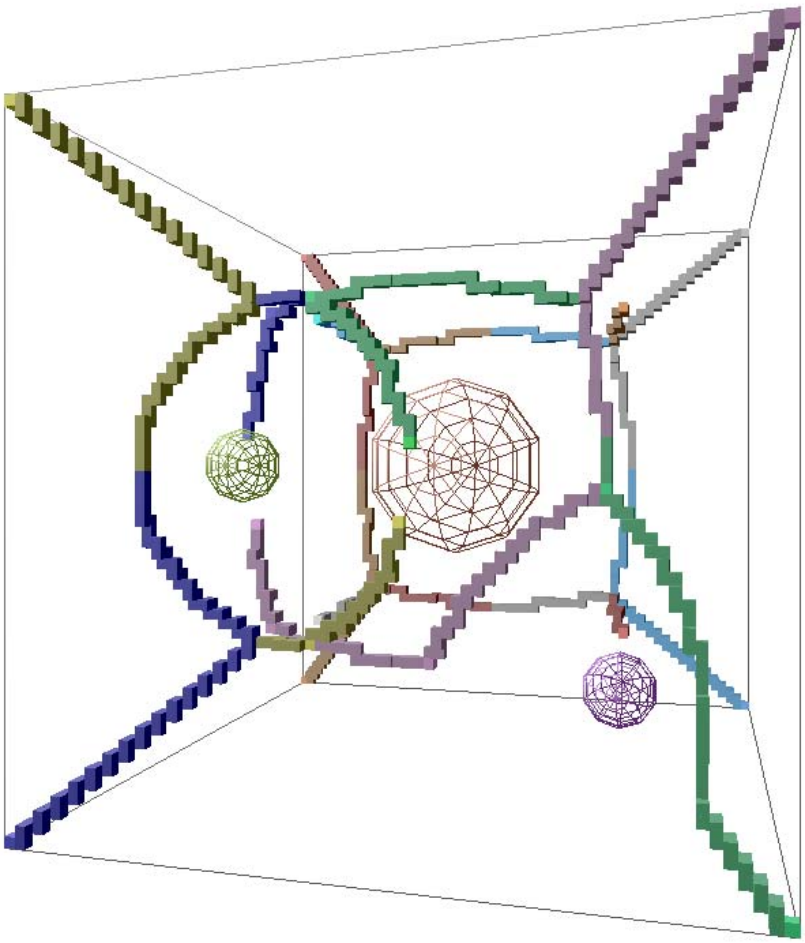

Figura 6.18: Time de robôs explorando a área de trabalho (parte 4)

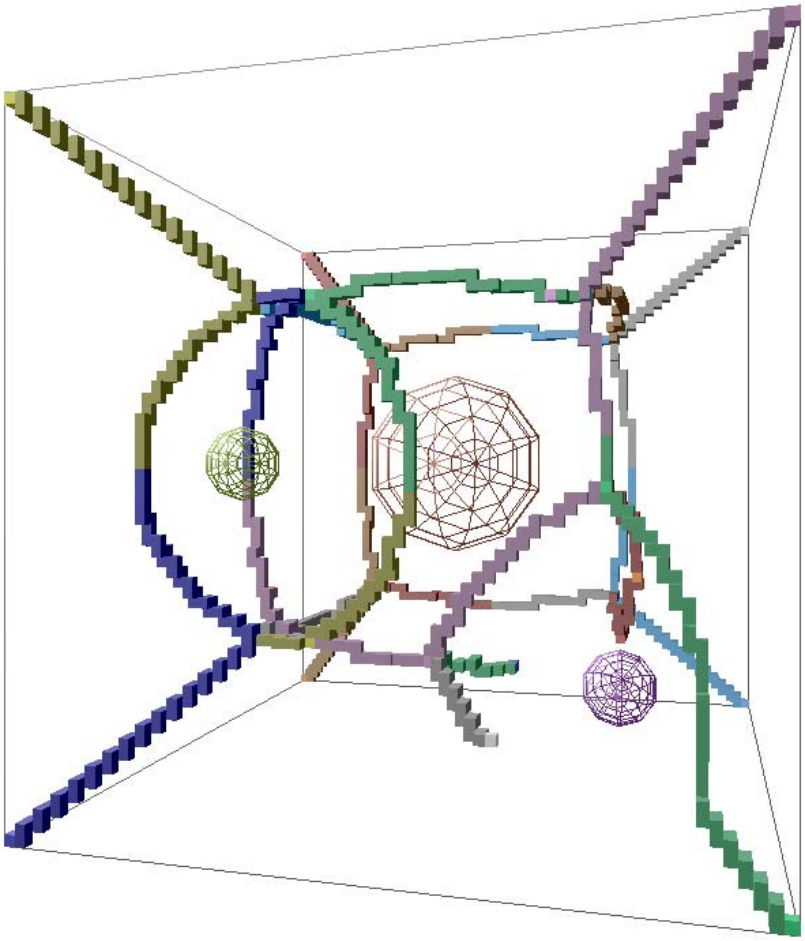

Figura 6.19: Time de robôs explorando a área de trabalho (parte 5)

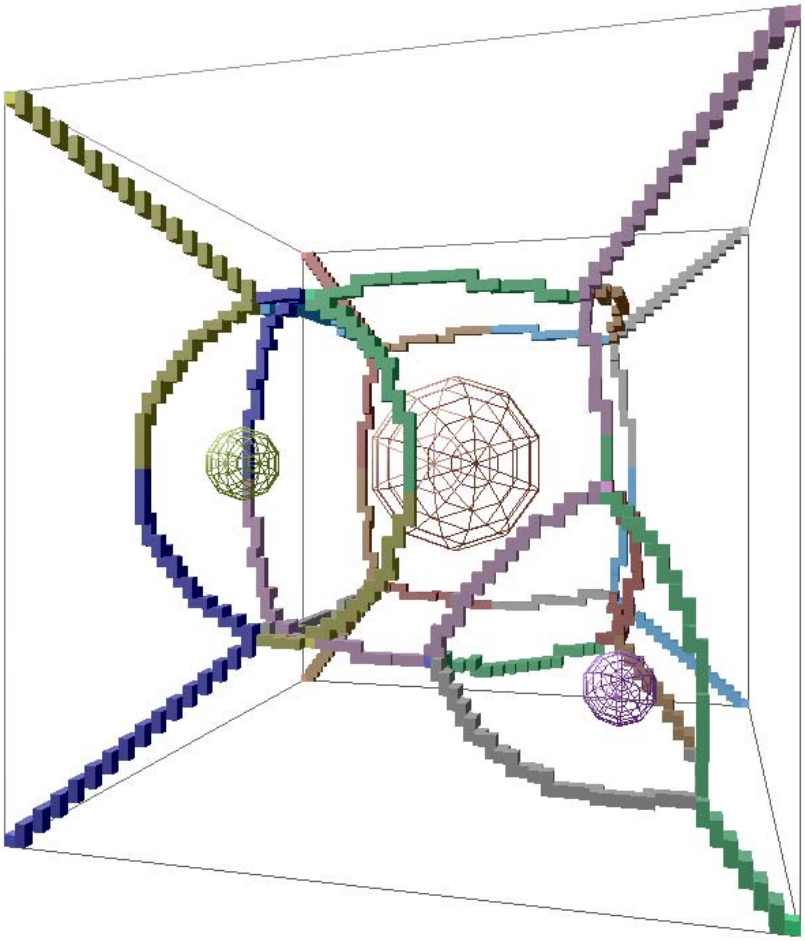

Figura 6.20: Área de trabalho totalmente explorada pelo time

```
-- Stats --------------------------------------------
# of cells      : 1133
# of verts      : 23

Robot#  NC      NC%     NV      NV%     Red     Red%
1       186     16.42   4       17.39   88      32.12
2       113     9.97    3       13.04   161     58.76
3       126     11.12   2       8.70    148     54.01
4       146     12.89   3       13.04   128     46.72
5       149     13.15   2       8.70    125     45.62
6       142     12.53   4       17.39   132     48.18
7       145     12.80   3       13.04   129     47.08
8       126     11.12   2       8.70    148     54.01

Efficiency      : 51.69
Steps           : 274
-----------------------------------------------------
```

Figura 6.21: Estatísticas geradas pelo simulador para o experimento apresentado

7 Conclusão

Esse trabalho teve como objetivo o estudo e a implementação de diversas alternativas para a solução do problema de planejamento de trajetórias utilizando diagramas de Voronoi. Inicialmente, uma extensão ao algoritmo proposto em[1] que calcula o diagrama de Voronoi para ambientes tridimensionais é apresentada. A paralelização desse algoritmo utilizando a técnica de decomposição regular de domínio e comunicação através da biblioteca MPI foi implementada e discutida. O *speed-up* da solução paralela proposta em relação a versão serial foi estudado e discutido.

Finalmente, um simulador foi desenvolvido para estudar o problema do planejamento cooperativo de trajetórias utilizando um time de robôs. Nesse simulador um time de robôs trabalha em conjunto para encontrar, em tempo real, o caminho seguro a ser seguido em um ambiente tridimensional. A área de trabalho do time possui obstáculos modelados em forma de esferas. O simulador funciona de forma distribuída e um do nós apresenta o resultado da simulação enquanto os outros simulam as tarefas realizadas pelos robôs na exploração. O simulador modela a comunicação dos robôs por rede sem fio. Um dos nós, chamado de módulo Observador é responsável pela apresentação dos resultados da simulação de forma gráfica tridimensional utilizando a biblioteca OpenGL. Os outros nós, através do módulo Robô, modelam a exploração do ambiente por cada robô na busca da solução do diagrama de Voronoi. Experimentos simulados foram realizados e apresentados.

7.1 Trabalhos Futuros

O algoritmo para o cálculo do diagrama de Voronoi em 3D utiliza, como aproximação dos obstáculos na área de trabalho, uma esfera com o menor raio possível. O objetivo dessa aproximação é permitir que o algoritmo possa ser utilizado para obstáculos com qualquer formato, bastando estarem envoltos com uma esfera. Como extensão dessa aproximação, poderia ser utilizado o fecho convexo do obstáculo, diminuindo assim o erro com relação a aproximação usando esferas.

O planejamento de trajetórias apresentado neste trabalho segue o modelo de planejamento clássico. Nesse modelo, todos os obstáculos são conhecidos inicialmente. Poderia ser realizada uma extensão desse algoritmo com técnicas de planejamento de trajetórias baseados em sensores. Neste caso os robôs vão descobrindo os obstáculos do ambiente com seus sensores. Devido a incerteza do ambiente, a definição do diagrama de Voronoi precisaria ser relaxada.

Um estudo mais completo do algoritmo paralelo pode ser realizado aumentando o número de estações de trabalho disponíveis para o cálculo. Assim, os gargalos existentes na rede podem ser melhor identificados e avaliados.

Finalmente, o modelo de exploração cooperativa proposto poderá ser implementado no futuro time de robôs do laboratório CONTROLAB.

Referências

[1]MENDES, F.S. AUDE, E.P.L. AUDE, J.S. Construction of the Voronoi diagram by a team of cooperative robots. In: INTERNATIONAL CONFERENCE ON INFORMATICS IN CONTROL, AUTOMATION AND ROBOTICS, 1., 2004, Setubal. **Proceedings ...** Setúbal, 2004. ICINCO´04.

[2] AUDE, E.P.L. et al. Real-time obstacle avoidance performed by an autonomous vehicle throughout a smooth trajectory using an electronic stick. In: IEEE/RSJ INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS, 2003, Las Vegas. **Proceeding …** Las Vegas, 2003. IROS'03

[3] KIM, J. ; KHOSLA, P. Real-time obstacle avoidance using harmonic potential functions. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1991, Sacramento. **Proceedings…** Sacramento, 1991. p. 790-796. ICRA'91.

[4] AUDE, E.P.L. et al. CONTROLAB MUFA: A multi-level fusion architecture for intelligent navigation of a telerobot. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1999, Detroit. **Proceedings…** Detroit, 1999. v. 1. p. 465-472. ICRA'99.

[5] LATOMBE, J.-C. **Robot motion planning**. Boston: Kluwer Academic Publishers, 1991.

[6] LOPES, E.P. et al. Obstacle avoidance strategy based on adaptive potential fields generated by an electronic stick. In: IEEE INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS, 2005, Edmonton. **Proceedings …** Edmonton, 2005. IROS'05.

[7] PÉREZ, T. L. ; MICHAEL, W. R. An algorithm for planning collision-free paths among polyhedral obstacles. **Communications of the ACM**, New York, vol. 22, n. 10, p. 560–570, Oct. 1979.

[8] COSTA, C.R.F. ; AUDE, E.P.L. Cooperation of multiple robots to solve maze tasks. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED AUTONOMOUS ROBOTIC SYSTEMS, 5., 2000, Knoxville. **Proceedings…** Knoxville, 2000.

[9] COSTA, C.R.F. ; AUDE,. E.P.L. Application of a cooperative team of robots to solve maze tasks. In: IASTED INTERNATIONAL CONFERENCE ON APPLIED INFORMATICS, 2001, Innsbruck. **Proceedings…** Innsbruck, 2001. AI 2001.

[10] AURENHAMMER, Franz. Voronoi Diagrams – A survey of a fundamental geometric data structure. **ACM Computing Surveys**, New York, vol. 23, n. 3, p.345–405, 1991.

[11] SHAMOS, M. ; HOEY, D. Closest-point problems. In: ANNUAL IEEE SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 16.,1975, Berkeley. **Proceedings …** Berkeley, 1974. p.151–162.

[12] FORTUNE, S. A Sweepline algorithm for Voronoi diagrams. In: ANNUAL ACM SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 2., 1986, Yorktown Weights. **Proceedings …** Yorktown Heights, 1986. p. 313–322.

[13] CHOSET, H. ; BURDICK, J. Sensor based planning: The Generalized Voronoi graph. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1995. Nagoya. **Proceedings …** Nagoya, 1995. Part I. p. 1649–1655 ICRA'95.

[14] CHOSET, H. ; BURDICK, J. Sensor based planning: . Incremental construction of the generalized Voronoig graph .In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1995. Nagoya. **Proceedings …** Nagoya, 1995. Part II. p 1643– 1648. ICRA' 95.

[15]HOFF III, K. E. et al. Fast computation of generalized Voronoi diagrams using graphics hardware. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 26., 1999, Los Angeles. In: **Proceedings …** New York: ACM Press, 1999. p. 277–286. SIGGRAPH '99.

[16] JARDOSH, Amit et al. Towards realistic mobility models for mobile ad hoc networks. In: ANNUAL INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, 9., 2003, San Diego. **Proceedings …** New York: ACM Press, 2003. p. 217-229. MobiCom'03.

[17] HARRINGTON, Brian ; HUANG, Yan. In-network surface simplification for sensor fields. In: ANNUAL ACM INTERNATIONAL WORKSHOP ON GEOGRAPHIC INFORMATION SYSTEMS, 13., 2005, Bremen. **Proceedings …** New York, 2005. p. 41–50. GIS '05:

[18]TZIONAS, P.G. Collision-free path planning for a diamond-shaped robot using two dimensional cellular automata. **IEEE Transactions on Robotics and Automation**, Hoes Lane, v. 13, n. 2 p. 237–250, Apr. 1997.

[19] SUDHA, N. ; NANDI, S. ; SRIDHARAN, K. A parallel algorithm to construct Voronoi diagram and its VLSI architecture. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1999, Detroit. **Proceedings…** Detroit, 1999. p. 1683-1688. ICRA'99.

20] VORONOI, Georgy. M. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. **Journal für die Reine und Angewandte Mathematik**, Berlin, v. 133, p. 97–178, 1908.

[21] MENDES, F.S. **Construção do diagrama de Voronoi por um time de robôs cooperativos**. 2005. Projeto Final (Curso de Informática) - Universidade Federal do Rio de Janeiro, Rio de Janeiro.

[22] OHYA, T. , IRI, M. ; MUROTA, K. Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms. **Journal of the Operational Research Society**, Oxford, Eng., v. 27, n. 4 p.306–336, 1984.

[23] SAAD, Yousef **Iterative methods for sparse linear systems**. [s.l.]: PWS Publishing Company, 1996.

[24] BLOOM, Burton H. Space/time trade-offs in hash coding with allowable errors. **Communications of the ACM**, New York, v. 13, n. 7, p.422–426, 1970.

[25] THARP, Alan L. **File organization and processing**. John Wiley and Sons, 1988.

[26] METCALFE, R.M. ; BOGGS, D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. **Communications of the ACM**, New York, v. 19, n. 7, p. 395–404, July 1976.

[27] GROPP, W. ;, LUSK, E. ; SKJELLUM, A. **Using MPI**: Portable Parallel Programming with the Message-Passing Interface.2.ed. Cambridge: The MIT Press, 1999 .(Scientific and Engineering Computation Series)

[28] SNIR, M. et al. **MPI**: The Complete Reference. Cambridge: The MIT Press, 1996.

[29] BEGUELIN, A. et al. **A user's guide to PVM**: Parallel virtual machine. , Oak Ridge: Technical Oak Ridge National Laboratory. 1991.(Report TM-11826).

[30] BOMANS, L. ; ROOSE, D. ; HEMPEL, R. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. **Parallel Computing**, Amsterdam, v. 15, n. 1-3, p. 119–132, 1990.

[31] BURNS, Greg ; DAOUD, Raja ; VAIGL, James LAM: An Open Cluster Environment for MPI. In: SUPERCOMPUTING SYMPOSIUM, 1994, Toronto. **Proceedings …** Toronto, 1994. p. 379–386.

[32] SQUYRES, Jeffrey M. ; LUMSDAINE, Andrew. A Component Architecture for LAM/MPI. In DONGARRA, J.; LAFORENZA, D. ; ORLANDO, S. (Eds.) **Recent Advances in Parallel Virtual Machine and Message Passing Interface** Berlin: Springer, 2003. p. 379-387. (Lecture Notes in Computer Science, 2840) Proceedings of the 10th European PVM/MPI User's Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings

[33] GROPP, W. et al. A high-performance, portable implementation of the MPI message passing interface standard. **Parallel Computing**, Amsterdam, v. 22, n. 6, p. 789–828, Sept. 1996.

[34] GOUGH, B.J**. An Introduction to GCC**. [s.l.]: Network Theory, 2004.

[35] WOO, Mason et al. **OpenGL programming guide**. 2.ed. Readings: Addison-Wesley Publishing, 1997.

[36] DIJKSTRA, E. W. A Note on two problems in connexion with graphs. **Numerische Mathematik**, Berlin, v. 1, n. 1, p. 269–271, Dec. 1959.