BALANCEAMENTO DE CARGA EM SISTEMAS PARALELOS DE VISUALIZAÇÃO VOLUMÉTRICA

Alexandre Coelho de Almeida

Dissertação submetida ao corpo docente da Faculdade de Engenharia da Universidade do Estado do Rio de Janeiro – UERJ, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia da Computação.

Orientadora: Prof[a]. Cristiana Barbosa Bentes.

Co-orientador: Prof. Ricardo Farias.

Programa de Pós-Graduação em Engenharia da Computação – Área de Concentração Geomática

Rio de Janeiro
Junho/2004

ALMEIDA, ALEXANDRE COELHO de

Balanceamento de Carga em Sistemas Paralelos de Visualização Volumétrica [Rio de Janeiro] 2004.

viii, 64 p. 29,7 cm (FEN/UERJ, M.Sc., Programa de Pós-Graduação em Engenharia de Computação - Área de Concentração Geomática, 2004)

Dissertação - Universidade do Estado do Rio de Janeiro - UERJ

1. Computação Paralela 2. Balanceamento de Carga 3. Visualização Volumétrica

I. FEN/UERJ II. Título (série)

FOLHA DE JULGAMENTO

Título: Balanceamento de Carga em Sistemas Paralelos de Visualização Volumétrica

Candidato: Alexandre Coelho de Almeida

Programa: Pós-Graduação em Engenharia de Computação – Área de Concentração Geomática

Data da defesa: 07 de junho de 2004

Aprovada por:

____

Profª. Cristiana Barbosa Bentes, D.Sc.,UERJ.

____

Prof. Ricardo Farias, Ph.D., UFRJ.

____

Profª. Maria Clícia Stelling de Castro, D.Sc., UERJ.

____

Prof. Cláudio Esperança, Ph.D., UFRJ.

Resumo da Dissertação apresentada à FEN/UERJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.).

BALANCEAMENTO DE CARGA EM SISTEMAS PARALELOS DE VISUALIZAÇÃO VOLUMÉTRICA

Alexandre Coelho de Almeida

Junho/2004

Orientadora: Profª. Cristiana Barbosa Bentes, D.Sc., UERJ.

Co-orientador: Prof. Ricardo Farias, Ph.D., UFRJ.

Programa de Pós-Graduação em Engenharia de Computação – Área de Concentração Geomática

Algoritmos de visualização volumétrica tratam os dados como se fossem compostos por um material semitransparente, permitindo mostrar detalhes do seu interior e, portanto, gerando imagens de alta qualidade. Diversas aplicações científicas se beneficiam da visualização científica de dados volumétricos. Porém, a visualização de grandes massas de dados é um problema conhecidamente dispendioso em termos computacionais. O uso de processamento paralelo e sistemas distribuídos, como *clusters* de PCs, são alternativas para se obter visualização volumétrica eficiente com baixo custo. Este tipo de aplicação, entretanto, sofre usualmente de grande desbalanceamento de carga durante a execução e o grande *overhead* de comunicação de um *cluster* de PCs piora este problema. Neste trabalho, propomos alguns algoritmos distribuídos de balanceamento de carga que podem ser aplicados a sistemas de visualização paralela. Nosso objetivo é fornecer algoritmos distribuídos que não sobrecarreguem a rede com mensagens de balanceamento de carga. Desenvolvemos três diferentes algoritmos de balanceamento de carga: Vizinho mais Próximo, Fila mais Longa e Distribuição Circular, fornecendo redistribuição dinâmica de trabalho em diferentes modos. Implementamos estes três algoritmos sobre o algoritmo *PZSweep* e nossos resultados experimentais mostraram que os algoritmos propostos de balanceamento de carga fornecem renderização com até 90% de eficiência paralela e apenas 10% de desbalanceamento de carga.

Palavras-chave: computação paralela, balanceamento de carga, visualização volumétrica.

Abstract of Dissertation presented to FEN/UERJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.).

LOAD BALANCING IN PARALLEL VOLUME RENDERING SYSTEMS

Alexandre Coelho de Almeida

June/2004

Advisors: Cristiana Barbosa Bentes and Ricardo Farias

After Graduation Program in Computer Engineering – Field of Geomatics

Volume visualization algorithms treat data as composed of a semitransparent material, allowing the analysis of the interior of the object and, therefore, generating high-quality images. A great number of scientific applications benefit from volume visualization. However, visualizing large-scale 3D datasets is a compute intensive application. The use of parallel processing and distributed systems, such as clusters of PCs, are alternatives to obtain efficient and low-cost volume rendering. These applications, however, usually suffer from high load imbalance during execution, and the high communication overhead of a cluster of PCs worsens this problem. In this work we propose some distributed load balancing algorithms that can be applied to tile-based parallel rendering system. Our goal is to provide distributed algorithms that do not overload the network with load balancing messages. We developed three different load balancing algorithms: Nearest Neighbor, Longest Queue, and Circular Distribution, providing dynamic redistribution of work in different ways. We implemented these three algorithms on top of PZSweep algorithm, and our experimental results show that the load balancing algorithms we proposed provides rendering with up to 90% of parallel efficiency and only 10% of load imbalance.

Keywords: parallel computing, load balancing, volume rendering.

SUMÁRIO

CAPÍTULO 1 –
INTRODUÇÃO..1
CAPÍTULO 2 –
VISUALIZAÇÃO VOLUMÉTRICA...5
CAPÍTULO 3 –
VISUALIZAÇÃO VOLUMÉTRICA PARALELA...9
CAPÍTULO 4 –
ALGORITMOS DE BALANCEAMENTO DE CARGA...................................20
CAPÍTULO 5 –
IMPLEMENTAÇÃO...28
CAPÍTULO 6 –
METODOLOGIA EXPERIMENTAL..49
CAPÍTULO 7 –
RESULTADOS EXPERIMENTAIS...52
CAPÍTULO 8 –
CONCLUSÕES..65

LISTA DE FIGURAS

Figura 2.1: Visualização de um objeto tridimensional real...6
Figura 3.2: Pipeline de visualização volumétrica...10
Figura 3.3: Arquiteturas paralelas. (a) Multiprocessadores com memória compartilhada fisicamente centralizada. (b) Multiprocessadores com memória compartilhada fisicamente distribuída. (c) Multicomputadores com memória distribuída...13
Figura 4.4: Algoritmo do Vizinho mais Próximo..23
Figura 4.5: Algoritmo da Fila mais Longa...24
Figura 4.6: Mecanismo de token-ring...25
Figura 4.7: Algoritmo da Distribuição Circular...27
Figura 5.8: Divisão em 8 por 8 de conjunto de dados volumétricos, produzindo 64 tiles........28
Figura 5.9: Métodos de seleção, para uma imagem dividida em 8 x 8...................................31
Figura 5.10: Métodos de distribuição estática, usando 4 nós e divisões em 16 x 16 tiles........32
Figura 6.11: Visualização volumétrica do conjunto de dados SPX..51
Figura 7.12: Comparação da seleção de todos os tiles com a seleção de tiles não vazios........53
Figura 7.13: Comparação dos métodos de distribuição estática...54
Figura 7.14: Speedups obtidos para diferentes algoritmos com o conjunto de dados SPX......57
Figura 7.15: Speedups obtidos para diferentes algoritmos com o conjunto de dados SPX1....58
Figura 7.16: Speedups obtidos para diferentes algoritmos com o conjunto de dados SPX2....58
Figura 7.17: Modificação aplicada ao algoritmo Parallel ZSweep..60
Figura 7.18: Efeito da variação da granulosidade sobre os tempos de execução.....................62
Figura 7.19: Efeito da variação da precisão da imagem sobre os tempos de execução...........64

LISTA DE TABELAS

Tabela 6.1: Informações sobre o número de vértices e o número de células nos conjuntos de dados SPX, SPX1 e SPX2..50
Tabela 7.2: Comparação dos tempos de execução, tempos de renderização e graus de desbalanceamento obtidos para cada um dos três métodos de coleta propostos......................55
Tabela 7.3: Speedups obtidos para diferentes algoritmos de balanceamento de carga............59
Tabela 7.4: Tempos de execução obtidos para os algoritmos NN, LQ e CD com diferentes granulosidades...61
Tabela 7.5: Tempos de execução obtidos para os algoritmos NN, LQ e CD com diferentes resoluções...64

CAPÍTULO 1
INTRODUÇÃO

A visualização volumétrica é uma importante técnica de computação gráfica que permite a geração de imagens bidimensionais a partir de conjuntos de dados volumétricos tridimensionais. Uma de suas características fundamentais é o tratamento de grandes quantidades de dados de natureza volumétrica, com o objetivo de apresentar suas interações, interligações e características que são consideradas complexas para serem percebidas em sua forma original. Algoritmos de visualização volumétrica, também chamados de algoritmos de renderização volumétrica, tratam os dados como se fossem compostos por um material semitransparente, permitindo mostrar detalhes do seu interior e, portanto, gerando imagens de alta qualidade. Diversas aplicações científicas se beneficiam da visualização científica de dados volumétricos, como, por exemplo, a visualização de dados coletados através de satélites e técnicas de sensoriamento remoto.

A visualização volumétrica de grandes massas de dados, entretanto, é um problema conhecidamente dispendioso em termos computacionais. Mesmo os algoritmos mais velozes gastam um tempo de execução substancial para executar em um único processador. A solução mais adotada para esse problema tem sido o uso de processamento paralelo. Dividindo o trabalho entre vários processadores trabalhando em paralelo, estes sistemas podem conseguir reduzir o tempo de processamento a uma fração do tempo gasto em um sistema com um único processador.

Vários algoritmos de renderização volumétrica paralela foram propostos na literatura, e eles conseguem obter um bom desempenho executando em máquinas paralelas de alto custo, como *SGI Power Challenge*, *IBM SP2* ou *SGI Origin 2000* [1, 2, 3, 4, 5]. Entretanto, a crescente disponibilidade e o baixo custo de tecnologias como *clusters* de PCs têm feito desta plataforma uma alternativa econômica para a execução de sistemas paralelos de visualização [6, 7, 8, 9]. Associada ao baixo custo, os *clusters* de PCs apresentam a vantagem da escalabilidade, já que, usando a tecnologia amplamente disponível de redes de computadores, é possível criar e expandir *clusters* para operar com centenas ou mesmo milhares de computadores. Além disso, eles podem ser facilmente atualizados seguindo as novas tendências em tecnologia.

Nosso objetivo neste trabalho é o de prover um sistema de renderização volumétrica paralela, para grandes massas de dados, que seja **eficiente**, porém, de **baixo custo** e, dessa forma, possa ser utilizado por cientistas de diversas áreas de conhecimento. Portanto, optamos por utilizar *clusters* de PCs como arquitetura alvo de nosso sistema. A implementação de visualização paralela em *clusters* de PC's, entretanto, esbarra nos problemas típicos desse tipo de arquitetura, como o alto custo de comunicação, o tamanho limitado da memória de cada PC e a baixa confiabilidade das máquinas envolvidas, se compararmos com máquinas paralelas de alto custo. Estamos interessados aqui em atacar especificamente o problema de desbalanceamento de carga entre os processadores, gerado devido à natureza irregular dos conjuntos de dados tridimensionais, que é uma característica das aplicações de visualização volumétrica.

Conjuntos de dados usados em visualização volumétrica possuem natureza irregular, apresentando grande variação na distribuição espacial dos objetos e produzindo imagens com algumas áreas vazias e outras áreas com grande densidade de objetos. Durante a divisão do trabalho de processamento, há uma tendência de que alguns processadores recebam grande carga de trabalho, demandando grande tempo de processamento, enquanto outros permanecem ociosos por terem recebido pouca carga de trabalho. O resultado desse desequilíbrio é o aumento do tempo de execução e o baixo desempenho do sistema. Técnicas de balanceamento de carga devem ser empregadas em conjunto com os sistemas de visualização volumétrica paralela de modo a evitar a degradação do desempenho.

Técnicas de balanceamento de carga, em geral, requerem a troca constante de informações sobre carga de trabalho entre os processadores. O desenvolvimento de algoritmos de balanceamento de carga para *clusters* de PCs torna-se um desafio na medida em que estes sistemas distribuídos não dispõem de uma área global de memória compartilhada. Nesta plataforma, toda a comunicação entre processadores se dá através da rede, o que implica em um grande custo associado à troca de mensagens.

O objetivo deste trabalho é o desenvolvimento de algoritmos distribuídos de balanceamento de carga para sistemas de visualização volumétrica paralela executando em *clusters* de PCs. Os algoritmos devem ser eficientes, proporcionando uma distribuição homogênea da carga de trabalho e evitando a sobrecarga da rede e dos processadores com a troca de mensagens relacionadas ao balanceamento de carga. A principal dificuldade a ser enfrentada é a ausência de informação global sobre a distribuição da carga de trabalho nos sistemas. Empregamos algumas técnicas conhecidas de difusão distribuída da informação para manter os processadores do *cluster* informados sobre a carga de trabalho do sistema sem o uso de mensagens de *broadcast*. Os algoritmos foram implementados em linguagem C++ na plataforma Linux usando a biblioteca de passagem de mensagens MPI.

1.1 Contribuições e Resultados

As principais contribuições desse trabalho são:

- Proposta de três diferentes algoritmos distribuídos de balanceamento de carga aplicados ao problema da visualização volumétrica paralela: algoritmo do Vizinho mais Próximo, algoritmo da Fila mais Longa e algoritmo da Distribuição Circular. Estes algoritmos se propõem a executar com eficiência o balanceamento de carga no problema da visualização volumétrica, empregando técnicas conhecidas de difusão de informação em sistemas distribuídos; e
- Avaliação do desempenho dos algoritmos. Resultados experimentais mostram que nossos algoritmos apresentam eficiência paralela de até 90%, com apenas 10% de desbalanceamento de carga, quando executados num cluster de 16 processadores, visualizando o maior conjunto de dados disponível.

1.2 Organização da Dissertação

Este trabalho está organizado da forma descrita a seguir. No Capítulo 2, abordamos os conceitos básicos relacionados à visualização volumétrica. O Capítulo 3 discute questões relevantes em sistemas paralelos de visualização volumétrica. No Capítulo 4 apresentamos detalhadamente nossos algoritmos para balanceamento de carga. O Capítulo 5 mostra aspectos da implementação dos algoritmos de balanceamento em um algoritmo de visualização volumétrica. O Capítulo 6 descreve o ambiente computacional, o sistema usado como base de comparação e os conjuntos de dados utilizados. No Capítulo 7 analisamos os resultados obtidos. Finalizamos, no Capítulo 8, com nossas conclusões e sugestões para trabalhos futuros.

CAPÍTULO 2

VISUALIZAÇÃO VOLUMÉTRICA

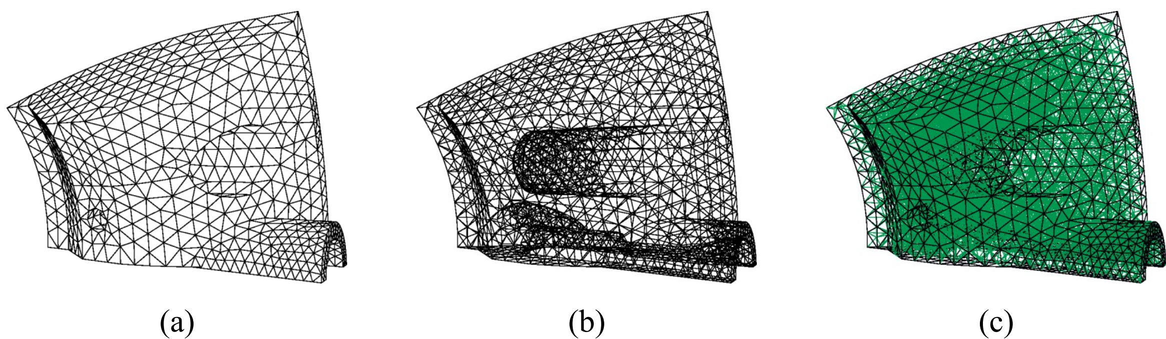
A visualização científica é uma sub-área da computação gráfica e tem recebido grande atenção da comunidade científica nos últimos anos, por permitir aos cientistas uma análise visual de seus experimentos, possibilitando uma melhor compreensão de processos físicos de difícil observação. A visualização volumétrica é uma técnica de visualização que permite a geração de imagens bidimensionais a partir de conjuntos de dados volumétricos tridimensionais. Chamamos de *rendering* volumétrico o processo pelo qual uma descrição abstrata de uma cena é convertida numa imagem. Os dados volumétricos podem ser capturados através da amostragem em três dimensões de um objeto real, usando diferentes dispositivos como: satélites, *scanners*, tomógrafos, simuladores, medidores especiais (por exemplo, bóias no oceano), entre outros dispositivos.

Um objeto tridimensional pode ser visualizado apenas pela sua superfície ou pela sua superfície e também pelo seu interior. Algoritmos de visualização de superfície empregados na computação gráfica tradicional simulam uma cena a partir de blocos básicos de construção como pontos, linhas e polígonos. Estas primitivas gráficas são descritas por atributos como posição, tamanho, orientação e cor. Para aumentar o realismo, os objetos podem receber textura e a cena pode receber iluminação. Embora a visualização de superfícies possa capturar cenas bem próximas de superfícies reais, apenas as superfícies dos objetos são consideradas, não sendo adequada para representar o interior de objetos, como, por exemplo, o interior do corpo humano.

A visualização volumétrica, em contraste, permite a representação, através de renderização, do interior de objetos de modo similar a um equipamento de raio X. Ao contrário de algoritmos tradicionais de visualização de superfície, que consideram todos os objetos como opacos ou transparentes, os algoritmos de visualização volumétrica consideram que os objetos são preenchidos por um material semitransparente e representados por uma triangulação espacial, chamada de tetraedrização. Ou seja, o interior dos objetos também contribui para a imagem final.

Na renderização volumétrica, raios imaginários são emitidos do observador, por cada *pixel* da imagem e passam através do objeto tridimensional, perfurando as células deste objeto. Na medida em que os raios atravessam os dados, eles interagem com as células do interior do objeto, acumulando valores numéricos de acordo com a intensidade, a opacidade e os atributos de cada elemento de volume atravessado, calculando-se a contribuição de cada um para a cor final do *pixel*. No término da renderização, os *pixels* gerados representam a projeção dos dados volumétricos no espaço bidimensional, compondo a imagem final.

A Figura 2.1 mostra diferentes visualizações de um objeto tridimensional real geradas a partir do conjunto de dados volumétricos SPX (cortesia de Peter Williams - LLNL), que consiste em um *grid* não estruturado composto de tetraedros e baseado em conjuntos de dados da NASA. Observamos em (a) somente as superfícies frontais do objeto. Em (b) estão representadas todas as suas superfícies. Em (c) observamos tanto a superfície quanto o interior do objeto, cujos dados internos encontram-se representados na cor verde.



(a) (b) (c)

Figura 2.1: Visualização de um objeto tridimensional real.
(a) Superfícies frontais. (b) Todas as superfícies. (c) Superfícies e interior.

Esta figura mostra como a visualização volumétrica é mais rica em detalhes, pois considera todo o conteúdo exterior e interior do objeto. Esta visualização mais detalhada tem um custo elevado e por isso os algoritmos de visualização volumétrica, em geral, apresentam custos de processamento muito superiores aos algoritmos de visualização de superfície. Por questões de desempenho, é comum a associação dos algoritmos de visualização volumétrica com técnicas de computação paralela.

2.1 Aplicações da Visualização Volumétrica

Dentre as diversas áreas de conhecimento que se beneficiam da visualização científica de dados volumétricos, podemos destacar algumas como: (1) medicina; (2) paleontologia; (3) modelagem computacional; (4) indústria; (5) geologia; ou (6) sistemas de informações geográficas.

Na área de **medicina**, a partir de dados do interior do corpo humano, capturados através de ressonância magnética, tomografia computadorizada ou ultra-sonografia, médicos podem visualizar, girar, e ampliar os dados tridimensionais, aplicar diferentes cores para diferentes tipos de tecidos ou mesmo tornar alguns tecidos transparentes de modo a permitir a visualização somente dos órgãos de interesse.

Na área de **paleontologia**, a visualização volumétrica pode ser usada para encontrar fósseis dentro de um volume de terra analisado através de equipamentos industriais de tomografia computadorizada. A visualização volumétrica permite a determinação da posição exata da espécie e a extração do fóssil sem o risco de causar a sua destruição.

Na área de **modelagem computacional**, a visualização volumétrica contribui para o estudo dos mais diversos sistemas. Na dinâmica de fluídos, por exemplo, podem ser modelados o fluxo de ar pelas asas de um avião ou o desenvolvimento de uma chama. Podem ser modelados, ainda, diferentes fenômenos naturais como turbulências oceânicas e tempestades magnéticas solares.

Na **indústria**, equipamentos de aquisição de dados em grande escala, como tomógrafos computadorizados, podem ser usados em combinação com a visualização volumétrica para identificar falhas no interior de estruturas como motores, turbinas e outras peças mecânicas.

Na área de **geologia**, a visualização volumétrica permite aos cientistas visualizar informações geológicas como porosidade de rochas, pressão, temperatura e permeabilidade, aplicando o conhecimento na descoberta e exploração de novos campos de petróleo, por exemplo. Dados sísmicos obtidos pela amostragem do subsolo são processados por técnicas de visualização volumétrica permitindo a geofísicos localizar possíveis áreas de acumulação de petróleo.

Em **sistemas de informações geográficas**, dados tridimensionais obtidos através de sensoriamento remoto, como satélites meteorológicos, podem ser renderizados através da visualização volumétrica de modo a permitir o estudo de fenômenos terrestres, oceânicos e atmosféricos, como a formação de nuvens, furacões e ciclones.

2.2 Algoritmos de Visualização Volumétrica

Um dos principais algoritmos utilizado na visualização volumétrica é baseado na técnica de *ray-casting*. Neste algoritmo, para cada *pixel* da imagem um raio imaginário é traçado entre o ponto de observação e a cena, passando pelo *pixel*. Os atributos de cada um dos objetos atravessados pelo raio são combinados para definir a cor final do *pixel*. A imagem final é composta pelo conjunto de *pixels*.

Uma importante implementação da técnica de *ray-casting* na visualização volumétrica foi proposta por Bunyk e outros em 8, na qual eles utilizam informações de adjacências das células do dado tridimensional para tornar mais eficiente a renderização de dados estruturados na forma de *grids* irregulares.

Outros algoritmos também desenvolvidos para visualização volumétrica foram baseados no paradigma de *sweep plane*. Nestes algoritmos, um plano de varredura atravessa o espaço tridimensional. Para cada posição do plano de varredura, uma seção bidimensional da cena é gerada a partir da interseção do plano de varredura com os objetos da cena. Cada seção é convertida em uma imagem, cujos *pixels* são combinados de forma acumulativa, à medida que o plano de varredura atravessa a cena, produzindo a imagem final.

Dentre os algoritmos que fazem uso do paradigma de *sweep plane* podemos destacar o *ZSweep*, desenvolvido por Farias e outros 8. O algoritmo *ZSweep* baseia-se na varredura dos dados com um plano paralelo ao plano de visualização, ao longo do eixo *z*, projetando as faces das células incidentes nos vértices a medida em que eles são encontrados pelo plano de varredura. O *ZSweep* é um algoritmo de visualização volumétrica bastante eficiente dado que ele explora a ordem global implícita aproximada que a ordenação em *z* dos vértices induz nas células incidentes a eles, de modo que apenas uma pequena quantidade de interseções é encontrada fora de ordem. Além disso, o uso de composição adiantada mantém baixa a necessidade de memória do algoritmo.

CAPÍTULO 3
VISUALIZAÇÃO VOLUMÉTRICA PARALELA

Nesse capítulo abordaremos alguns conceitos básicos sobre sistemas paralelos de visualização de volume. Discutimos a respeito do tipo de paralelismo empregado, do modelo de distribuição de tarefas e distribuição de dados, da granulosidade da divisão dos dados, do problema de balanceamento de carga e dos problemas de restrição de memória.

3.1 Tipo de Paralelismo

O tipo de paralelismo empregado define como a tarefa de visualização é subdividida em processos a serem executados em paralelo. Essa subdivisão pode ser feita de três formas diferentes: paralelismo funcional, paralelismo temporal e paralelismo de dados. Normalmente, o tipo de paralelismo empregado depende do tipo de algoritmo e aplicação. A seguir, apresentamos um breve resumo desses três tipos de paralelismo.

3.1.1 Paralelismo Funcional

O processo de renderização pode ser dividido em diferentes funções que podem ser aplicadas em série a itens individuais de dados. As funções são executadas em série de modo que os dados de saída de uma função são usados como dados de entrada da próxima função. Se cada função ou grupo de funções é atribuído a um processador distinto, forma-se um *pipeline* de renderização. Quando o *pipeline* está preenchido, os processadores trabalham em paralelo, cada um executando uma função distinta da renderização sobre um conjunto individual de dados.

A Figura 3.1 ilustra um *pipeline* de renderização típico. O número de unidades funcionais e sua ordem relativa podem variar de acordo com a implementação do sistema de visualização volumétrica.

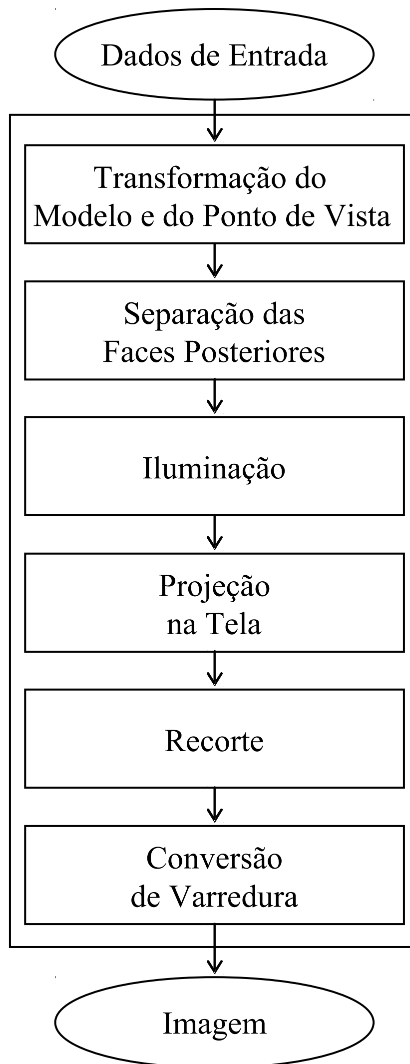


Figura 3.2: *Pipeline* de visualização volumétrica.

Um exemplo de aplicação do paralelismo funcional é o *Clark's Geometry System* [12, 13], que usa um *pipeline* de 12 estágios para executar transformações e operações de recorte em duas e três dimensões.

O paralelismo funcional apresenta duas grandes limitações. Primeiro, a velocidade do *pipeline* é limitada pelo estágio mais lento. Segundo, o número de processadores que podem ser empregados é limitado pelo número de estágios da renderização.

3.1.2 Paralelismo Temporal

É usado em seqüências de animação, onde milhares de imagens, ou quadros, devem ser produzidos para reprodução posterior. Neste tipo de paralelismo, cada conjunto de quadros é atribuído a um processador distinto, dividindo o problema no domínio do tempo e conseguindo uma redução no tempo total necessário para a renderização de todos os quadros.

3.1.3 Paralelismo de Dados

Neste tipo de paralelismo, os dados são divididos em múltiplas partições, sendo cada uma atribuída a um processador distinto. Cada processador executa todos os estágios da renderização sobre sua partição de dados. Este tipo de paralelismo não está limitado ao número de estágios da renderização, podendo ser escalonado para um grande número de processadores, de modo a prover o desempenho demandado por cenas complexas e imagens de alta resolução.

É possível aplicar o paralelismo de dados a cada estágio do paralelismo funcional ou a cada quadro do paralelismo temporal, produzindo sistemas de paralelismo híbrido.

O paralelismo de dados é uma técnica aplicável a uma gama maior de sistemas, podendo ser empregado, por exemplo, em sistemas de renderização executados em arquiteturas paralelas de uso genérico, permitindo grande escalabilidade. Por este motivo, o paralelismo de dados é o tipo de paralelismo aplicado neste trabalho.

3.2 Distribuição de Dados

Quando consideramos o paralelismo de dados, a distribuição dos dados pelos diversos elementos de processamento tem fundamental importância na implementação e na eficiência da visualização paralela. Há basicamente duas formas de se dividir os dados entre os elementos de processamento: divisão do objeto e divisão da imagem.

Na divisão do objeto, as primitivas gráficas que descrevem os objetos em uma cena são distribuídas entre os processadores para renderização individual. Cada processador renderiza os objetos que lhe foram atribuídos, produzindo resultados parciais que devem ser posteriormente integrados para produzir cada *pixel* da imagem completa.

Na divisão da imagem, o espaço da imagem é particionado e seus *pixels* são distribuídos entre os processadores. Cada processador renderiza os objetos que contribuem para os *pixels* que lhe foram atribuídos. A imagem completa é composta pelos *pixels* renderizados por cada processador.

Quando os objetos são divididos entre os elementos de processamento, a renderização individual de cada processador requer uma fase para a composição final da imagem, dado que um processador gera apenas resultados parciais para cada *pixel*. Em cada processador, a renderização dos objetos resulta em um valor parcial associado a cada *pixel*. Os valores parciais associados ao mesmo *pixel* precisam ser **combinados** na fase final de composição para produzir o valor final de cada *pixel* da imagem.

Quando o espaço da imagem é dividido entre os elementos de processamento, não há necessidade de composição final da imagem. Entretanto, os processadores devem ser capazes de obter todos os dados relativos à geração dos *pixels* que lhe foram designados. Em cada processador, a renderização resulta nos valores finais dos *pixels* de uma parte da imagem. Os *pixels* produzidos em cada processador precisam apenas ser copiados da sub-imagem local e **colados** na mesma posição da imagem final.

O processo de **colagem** de *pixels* é muito mais simples do que o processo de **combinação** de valores parciais. Por isto, neste trabalho, optamos pela divisão da imagem entre os elementos de processamento como método de partição dos dados.

3.3 Distribuição de Tarefas

Nos algoritmos de renderização paralela o problema é decomposto em tarefas. Cada tarefa é definida como um conjunto de operações sobre os dados. As tarefas são distribuídas entre os processadores de modo a dividir a carga de trabalho do sistema. O modelo de distribuição de tarefas define o padrão de acesso aos dados e está intimamente vinculado à arquitetura da máquina.

Comumente arquiteturas paralelas são classificadas, conforme o acesso à memória, em multiprocessadores e multicomputadores. Multiprocessadores possuem memória compartilhada, isto é, os processadores utilizam um espaço de endereçamento único e a memória pode estar fisicamente centralizada ou distribuída. Já os multicomputadores não possuem espaço de endereçamento único e um processador não tem acesso às memórias de outros processadores, devendo a comunicação ser feita através de mensagens pela rede.

A Figura 3.2 ilustra três modelos de arquiteturas paralelas: em 3.2(a) observamos a arquitetura de multiprocessadores com memória compartilhada fisicamente centralizada; em 3.2(b) é apresentada uma arquitetura de multiprocessadores com memória compartilhada fisicamente distribuída; em 3.2(c) observamos a arquitetura de multicomputadores com memória distribuída.

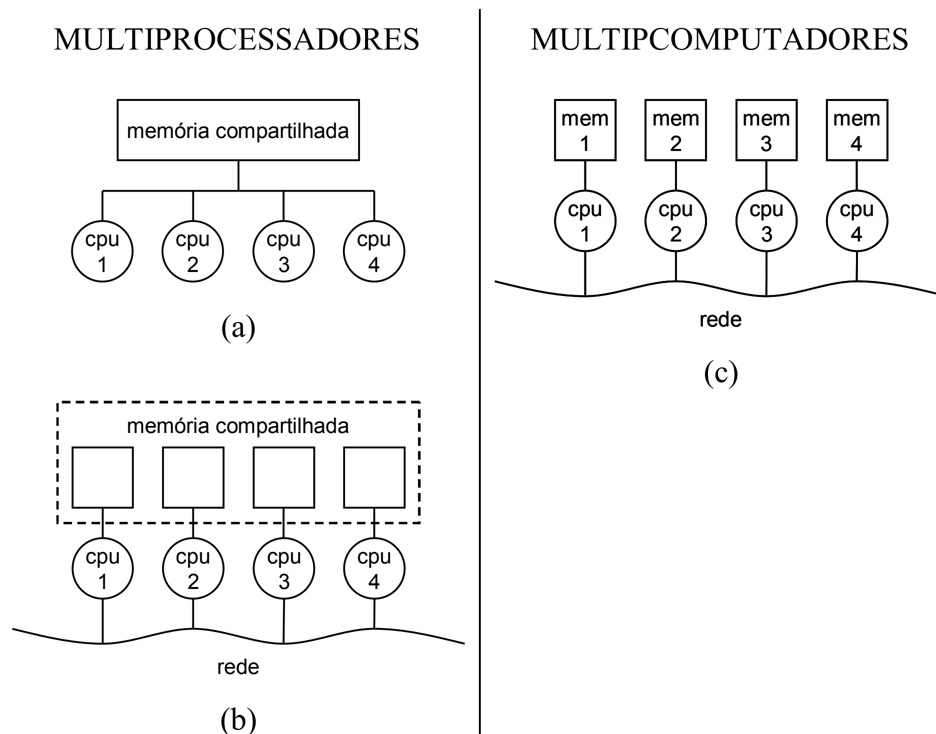


Figura 3.3: Arquiteturas paralelas. (a) Multiprocessadores com memória compartilhada fisicamente centralizada. (b) Multiprocessadores com memória compartilhada fisicamente distribuída. (c) Multicomputadores com memória distribuída.

3.3.1 Fila Única de Tarefas

Um dos modelos usados para distribuição de tarefas emprega o conceito de fila única de tarefas. Neste modelo, uma fila única de tarefas é produzida dividindo-se o trabalho a ser realizado. Cada processador retira uma tarefa da fila por vez para processamento. Ao terminar, retira uma nova tarefa da fila e assim sucessivamente, até que a fila esteja vazia.

Este tipo de distribuição provê um excelente balanceamento de carga entre os processadores, mas é mais bem empregada em arquiteturas de multiprocessadores, onde todos os processadores têm acesso à fila única de tarefas que está na memória compartilhada. Usando uma fila única de tarefas e com uma tarefa sendo retirada da fila de cada vez, o máximo de desbalanceamento que pode ocorrer no sistema equivale a uma unidade de trabalho computacional, isto é, uma tarefa, produzindo um desbalanceamento de carga mínimo.

Em arquiteturas de memória distribuída, a técnica precisa ser modificada para um modelo de mestre e escravo. A fila única de tarefas é mantida por um processador mestre, que sob demanda distribui uma tarefa de cada vez aos processadores escravos, que por sua vez recebem cada tarefa, processam-na e solicitam nova tarefa ao mestre.

O modelo de distribuição baseado em fila única de tarefas se aplica muito bem em arquiteturas de memória compartilhada, mas já não apresenta a mesma escalabilidade em arquiteturas de memória distribuída, pois na medida em que o número de processadores aumenta, pode haver diminuição na capacidade do processador mestre de responder às solicitações em tempo aceitável.

3.3.2 Distribuição Estática de Tarefas

Outro modelo usado para distribuição de tarefas baseia-se em distribuição estática. Neste modelo, após a partição inicial dos dados, as tarefas são distribuídas entre os processadores. Cada processador monta sua própria fila de tarefas e passa a processar uma tarefa de cada vez, retirando-a de sua fila.

Este método de distribuição é mais adequado para sistemas de memória distribuída, já que neste tipo de arquitetura os processadores não dispõem de uma área de memória global onde possam acessar uma fila única de tarefas. Como a arquitetura alvo de nosso trabalho é a de um *cluster* de PCs, utilizamos em nossas implementações a distribuição estática de tarefas.

A distribuição estática de tarefas também é mais adequada para a construção de sistemas tolerantes a falhas, pois o sistema não depende de um processador mestre durante o processamento das tarefas, evitando a existência de um ponto crítico de falha como ocorre no modelo mestre-escravo.

Esta distribuição, entretanto, tende a gerar um desbalanceamento de carga no sistema, tendo em vista que cada tarefa apresenta uma complexidade diferente em função da região da imagem a ser renderizada e dos objetos a serem processados. Mesmo que cada processador receba uma quantidade igual de tarefas, a carga de trabalho associada pode ser bem distinta. Este modelo de distribuição normalmente requer o emprego de técnicas de rebalanceamento dinâmico de carga. Dessa forma, em nossas implementações utilizamos inicialmente uma distribuição estática e a combinamos com técnicas de balanceamento dinâmico de carga.

3.4 Balanceamento de Carga

Em computação paralela, o desempenho global do sistema depende de uma efetiva utilização de todos os processadores disponíveis, evitando que alguns processadores fiquem ociosos enquanto outros ainda executam trabalho de processamento. Para alcançar este objetivo é necessário proporcionar uma distribuição homogênea da carga de trabalho entre os processadores.

Em sistemas de renderização paralela é difícil obter uma distribuição homogênea da carga de trabalho. Usando paralelismo de objetos, mesmo que os objetos geométricos sejam homogeneamente distribuídos entre os processadores, cada objeto contém um número variável de vértices e possui uma complexidade de renderização distinta dos demais. Usando paralelismo de imagem, mesmo que cada processador receba uma partição da imagem com área igual às demais, cada região exibe uma densidade diferente de objetos, resultando em diferentes cargas de trabalho.

Técnicas de balanceamento de carga devem ser empregadas com o objetivo de promover uma distribuição mais homogênea da carga de trabalho e uma efetiva utilização de todos os processadores disponíveis, melhorando o desempenho global do sistema.

Um método comum de balanceamento de carga consiste em promover uma distribuição estática de tarefas baseada em previsão de carga. Neste método, para cada tarefa é estimado um tempo relativo de execução com base em cálculos heurísticos, em uma fase de pré-processamento. A partir das estimativas de tempos de execução, as tarefas são estaticamente distribuídas entre os processadores, de modo a tentar igualar os tempos totais de execução previstos para as tarefas distribuídas para cada processador.

Este método depende fundamentalmente da precisão da heurística empregada na estimativa de tempo de execução. Além disso, acrescenta um custo adicional em função do tempo de pré-processamento gasto com a previsão de carga. Samanta e outros 8 propuseram três técnicas diferentes de previsão de carga de trabalho para partição dos dados, usados no balanceamento de carga de um sistema de renderização com múltiplos projetores.

Outro método comum de balanceamento de carga baseia-se na redistribuição dinâmica de tarefas entre os processadores, técnica conhecida como *work stealing*. Este método tenta minimizar o custo de pré-processamento retardando as decisões de distribuição de carga até o momento em que um ou mais processadores tornam-se ociosos. Quando um processador termina o processamento das suas tarefas locais, tornando-se ocioso, solicita tarefas de outro processador, de modo a tentar equilibrar a carga de trabalho entre os dois.

Os principais custos adicionais associados a este método envolvem a manutenção e recuperação de informações não locais de estado, a repartição de tarefas e a migração de dados entre processadores. Whitman 8 publicou um exemplo de aplicação da técnica de *work stealing* onde um processador, ao terminar de processar suas tarefas locais, procura pelo processador com a maior carga de trabalho e solicita ao outro processador metade das tarefas remotas.

Neste trabalho, estamos interessados em investigar algoritmos de balanceamento de carga baseados no princípio de *work stealing*, aplicados a sistemas com distribuição inicial estática de tarefas. Nossa proposta é determinar diferentes formas de escolha do processador alvo da solicitação de trabalho.

3.5 Granulosidade

No paralelismo de dados, os dados de entrada são divididos antes de serem distribuídos entre os processadores. Cada partição de dados representa uma unidade de trabalho, ou tarefa, que deve ser executada por algum processador. Cada processador pode receber uma ou mais unidades de trabalho para executar.

O conceito de granulosidade se refere à quantidade de divisões ou partições que são efetuadas sobre os dados de entrada antes de sua distribuição para os processadores. Quanto maior o número de partições, ou unidades de trabalho, resultantes da divisão dos dados de entrada, mais fina é a granulosidade do sistema. Em um sistema com granulosidade grossa, cada processador recebe uma única ou poucas unidades de trabalho, estando cada unidade de trabalho associada a uma grande carga de processamento. Em um sistema com granulosidade fina, cada processador pode receber muitas unidades de trabalho, estando cada unidade de trabalho associada a uma pequena carga de processamento como, por exemplo, um único *pixel* da imagem.

A granulosidade apresenta uma relação direta com o desempenho do sistema. Granulosidade fina, geralmente, acrescenta maiores custos de gerenciamento de tarefas e comunicação entre processos, mas resulta em um melhor balanceamento de carga entre processadores. A granulosidade grossa apresenta menores custos de gerenciamento de tarefas e comunicação, mas pode levar a um severo desbalanceamento de carga no sistema.

Para cada sistema pode ser determinada a granulosidade ideal, avaliando-se experimentalmente o desempenho do sistema usando diferentes granulosidades.

3.6 Restrições de Memória

A visualização volumétrica de grandes conjuntos de dados, em computadores com quantidade de memória limitada, representa um problema a ser tratado. Muitas vezes, os dados são gerados em supercomputadores, mas precisam ser visualizados em estações de trabalho comuns. Para resolver este problema faz-se necessário o desenvolvimento de técnicas de renderização *out-of-core*.

Nas técnicas de renderização *in-core*, todos os dados de entrada são carregados do disco para a memória principal do sistema, onde são mantidos e diretamente acessados durante toda a renderização, requerendo disponibilidade de grande quantidade de memória.

Nas técnicas de renderização *out-of-core*, os dados são carregados do disco na medida em que são necessários para cada etapa da renderização, sendo mantida na memória principal, a cada instante, apenas a quantidade mínima de dados necessária para processamento.

Apesar de apresentarem desempenho, relativo aos tempos de execução, inferior às técnicas *in-core*, as técnicas de renderização *out-of-core* permitem a visualização volumétrica de conjuntos de dados de tamanho muito superior à quantidade de memória principal disponível.

O *Out-Of-Core ZSweep* 8 é um algoritmo de visualização volumétrica *out-of-core* desenvolvido por Farias e Silva, cuja idéia básica é a divisão do conjunto de dados em partes de tamanhos iguais, que podem ser renderizadas de forma independente. A tela onde é projetada a imagem também é divida e, para cada região da tela, são renderizadas apenas as partes do conjunto de dados que se projetam naquela região.

3.7 Trabalhos Relacionados

Há uma grande quantidade de trabalhos na área de renderização paralela. Aqui sumarizamos alguns desses trabalhos, concentrando nossas discussões na pesquisa mais diretamente relacionada ao problema de balanceamento de carga em sistemas de renderização paralela.

Para a renderização de polígonos, em hardware de processamento paralelo de uso genérico, um dos primeiros sistemas desenvolvidos foi o *Pixel-Planes* 8 de Fuchs e Poulton. Outro sistema de referência é o *Pixel Machine* 8, da AT&T, que faz uso combinado de paralelismo funcional e paralelismo de dados.

Para a renderização baseada na técnica de *ray-tracing* foi desenvolvida a arquitetura SIGHT 8, onde o espaço da imagem é divido entre os processadores, sendo cada processador responsável por traçar os raios que emanam dos *pixels* que lhe foram atribuídos.

Para a renderização volumétrica paralela faz-se necessário o desenvolvimento de estruturas de memória que permitam acesso múltiplo aos dados de volume. O sistema *Cube* 8, desenvolvido por Kaufman e Bakalash, introduziu uma técnica de acesso a cubos de dados volumétricos. Knittel e Straßer 8 desenvolveram uma arquitetura de renderização volumétrica baseada em paralelismo funcional.

Whitman 8 propôs um método adaptativo de tarefas, baseado no particionamento dinâmico da quantidade de trabalho computacional restante em cada processador. Quando um processador termina seu trabalho previamente atribuído, ele divide o trabalho de outro processador em duas tarefas, ficando com a primeira tarefa para si e deixando a segunda permanecer no outro processador. Este esquema foi proposto para uma arquitetura de memória compartilhada.

Samanta et al 8 desenvolveram um sistema de visualização paralela usando uma rede de PCs, placas gráficas disponíveis no mercado consumidor e um sistema de vídeo com múltiplos projetores. O foco do trabalho era encontrar uma boa estratégia de divisão do espaço da tela, associando aos computadores as regiões resultantes da divisão da tela. Foram obtidos resultados interessantes, mas focados na visualização de polígonos, e não na visualização volumétrica.

Nieh e Levoy 8 desenvolveram um sistema de visualização paralela volumétrica que usa distribuição estática de tarefas associada a *work stealing* quando necessário. Entretanto, o trabalho foi baseado em um algoritmo paralelo de *ray-tracing*, executando em uma arquitetura de multiprocessadores com memória compartilhada fisicamente distribuída, onde há um único espaço de endereçamento visível por todos os processadores.

Meißner et al 8 desenvolveram um sistema de visualização volumétrica paralela para um *cluster* de PCs. Eles usaram um algoritmo de *ray-casting* e implementaram balanceamento de carga usando um modelo de mestre e escravo.

Farias e Silva 8 desenvolveram o *Parallel ZSweep*, uma técnica de visualização volumétrica paralela baseada no *Out-Of-Core ZSweep*, em que a tela onde é projetada a imagem é dividida e cada região da tela a ser renderizada constitui uma unidade de trabalho ou tarefa. As tarefas são mantidas em uma fila gerenciada por um processador mestre, que recebe solicitações dos demais processadores. A cada solicitação, uma tarefa é retirada da fila pelo processador mestre e enviada ao solicitante. O processador solicitante recebe a tarefa, renderiza a região da tela correspondente, e então envia nova solicitação ao mestre. O processo continua até que toda a imagem tenha sido renderizada.

CAPÍTULO 4

ALGORITMOS DE BALANCEAMENTO DE CARGA

Neste capítulo apresentamos os algoritmos de balanceamento de carga que estamos propondo para sistemas de visualização volumétrica paralela em arquitetura de memória distribuída. Os algoritmos foram desenvolvidos usando os princípios de paralelismo de dados, divisão da imagem entre os processadores e mecanismos de distribuição dinâmica e de distribuição estática combinada com *work stealing* para balanceamento da carga. Eles são razoavelmente genéricos e, portanto, podem ser aplicados a uma grande variedade de sistemas de visualização paralela.

O principal objetivo considerado na elaboração dos algoritmos é o de conseguir uma distribuição homogênea da carga de trabalho, com baixo *overhead* e baixa utilização da rede, em um ambiente de *cluster* de PCs. A principal dificuldade enfrentada foi manter em cada nó do *cluster* informações atualizadas sobre a carga de trabalho e o estágio de renderização dos demais nós, sem a utilização de uma área de memória global compartilhada.

Foram desenvolvidos três algoritmos diferentes de balanceamento de carga: Vizinho mais Próximo (NN - *Nearest Neighbor*), Fila mais Longa (LQ - *Longest Queue*) e Distribuição Circular (CD - *Circular Distribution*). Os dois primeiros empregam distribuição inicial estática de tarefas combinada com técnicas de redistribuição dinâmica de carga, enquanto o último algoritmo usa apenas um método de distribuição dinâmica de tarefas.

4.1 Estratégias de Distribuição de Tarefas

A distribuição de tarefas entre os processadores é uma fase do processo determinante para a estratégia de balanceamento de carga a ser adotada no sistema. Podem ser empregados métodos dinâmicos e estáticos de distribuição tarefas.

Na distribuição dinâmica, existe uma única fila de tarefas global. Cada processador retira uma tarefa da fila de cada vez para processamento, até que tenha terminado e esteja pronto para receber a próxima tarefa. A distribuição de tarefas ocorre simultaneamente com a renderização.

Na distribuição estática, cada processador recebe uma quantidade igual de tarefas e as mantém em uma fila de tarefas local, cada processador com sua própria fila. A distribuição de tarefas só acontece no início da execução, antes da renderização. Este é um método muito simples e pode ser executado até mesmo sem causar nenhum tráfego de rede. Cada processador pode montar sua própria lista de tarefas, desde que sejam previamente conhecidos: o número total de processadores; o número total de tarefas a serem distribuídas; e o tipo de distribuição a ser usada.

A distribuição estática tende a causar um desequilíbrio da carga de trabalho entre os processadores, tendo em vista que, como cada tarefa requer uma quantidade de processamento diferente, cada processador recebe uma carga de trabalho diferente, mesmo que tenha recebido uma quantidade igual de tarefas para processar. Para compensar esse desequilíbrio, devem ser empregados métodos de balanceamento de carga.

A distribuição dinâmica constitui, por si só, uma estratégia para distribuição equilibrada de carga entre os processadores, não sendo necessária a utilização de nenhum outro método de balanceamento de carga. Usando uma fila de tarefas única e com cada processador retirando uma tarefa da fila por vez, o resultado é um sistema muito bem equilibrado, ficando o desbalanceamento limitado no máximo ao tamanho da unidade mínima de trabalho, ou seja, uma tarefa.

4.2 Estratégias de Balanceamento de Carga

Os métodos de distribuição estática criam filas de tarefas independentes, uma para cada processador. Antes da fase de renderização, todas as filas possuem tamanhos iguais. Mas cada tarefa está associada à renderização de uma determinada parte da tela, com complexidade distinta das demais partes, requerendo uma quantidade de processamento diferente. A tendência é que, durante a fase de renderização, com algumas tarefas sendo executadas mais rapidamente e outras mais lentamente, cada processador passe a ter uma fila de tarefas de tamanho diferente. Até o ponto em que alguns processadores terminam o processamento das suas tarefas, encontrando-se ociosos e com uma fila de tarefas vazia, ao mesmo tempo em que outros processadores encontram-se em pleno trabalho de renderização e mantendo longas filas de tarefas.

Este desequilíbrio de carga de trabalho, resultado natural dos métodos de distribuição estática, influencia diretamente o tempo total de execução e o desempenho do sistema. Em um sistema bem balanceado, os processadores terminam a renderização em tempos bem próximos, e o tempo de execução se aproxima do tempo médio de renderização. Em um sistema desbalanceado, um processador pode terminar sua renderização em um tempo próximo de zero, se a ele forem atribuídas apenas áreas vazias da tela, enquanto que outro pode terminar sua renderização, por exemplo, com o dobro do tempo médio de renderização. Neste exemplo, o tempo de execução seria superior ao dobro do tempo médio de renderização, pois a execução só termina quando a imagem foi completamente gerada, o que não ocorre até que todos os processadores, até o último deles, tenham terminado a renderização.

Quanto melhor o balanceamento da carga de trabalho, maior o desempenho do sistema e, nesse sentido, os métodos de distribuição estática precisam estar sempre associados a algum método de balanceamento de carga. Conforme descrito anteriormente, um dos métodos mais usados para balanceamento de carga é chamado de *work stealing*. Neste método, um processador retira uma ou mais tarefas da fila de outro processador e os adiciona à sua própria fila, com o objetivo de tentar equilibrar a carga de trabalho entre estes dois processadores. Se as filas passarem a ter tamanhos iguais após o *work stealing*, maior é a probabilidade de que a carga de trabalho esteja mais equilibrada.

Nas nossas implementações do *work stealing*, um processador envia uma requisição a outro processador que possua uma fila com mais tarefas. Caso a fila de tarefas do primeiro processador esteja vazia, o segundo processador enviará como resposta metade das tarefas da sua fila. De modo mais genérico, se um processador que possui uma fila de tarefas com n tarefas envia uma solicitação a outro processador com uma fila de m tarefas, e supondo $m > n$, o segundo processador responderá ao primeiro enviando $(m - n) / 2$ tarefas de sua própria fila, sempre com o objetivo de igualar o tamanho das filas de tarefas e aumentar a probabilidade de ter uma distribuição de carga mais equilibrada. Caso o resultado da divisão (m - n) / 2 não seja inteiro, arbitrariamente optamos por arredondar o resultado para o número inteiro imediatamente superior, pois testes preliminares indicaram que o tipo de arredondamento empregado, para mais ou para menos, não tem influência sobre o desempenho do sistema.

Já os métodos de distribuição dinâmica, como a distribuição circular descrita a seguir neste capítulo, garantem por si só uma distribuição bem equilibrada da carga de trabalho entre os processadores, constituindo também uma estratégia de balanceamento de carga.

4.3 Algoritmo do Vizinho mais Próximo (NN)

Neste algoritmo de balanceamento de carga, durante a etapa de renderização, cada processador *p*, ao determinar que sua fila de tarefas encontra-se vazia, envia uma mensagem de solicitação de tarefas ao processador *p* + 1, que responde retirando de sua fila e enviando de volta metade de suas tarefas. O processador solicitante recebe as tarefas, adiciona-as à sua fila e continua com o processamento das tarefas recebidas. A Figura 4.1 ilustra este processo, que se repete até que todas as tarefas de todos os processadores tenham sido processadas.

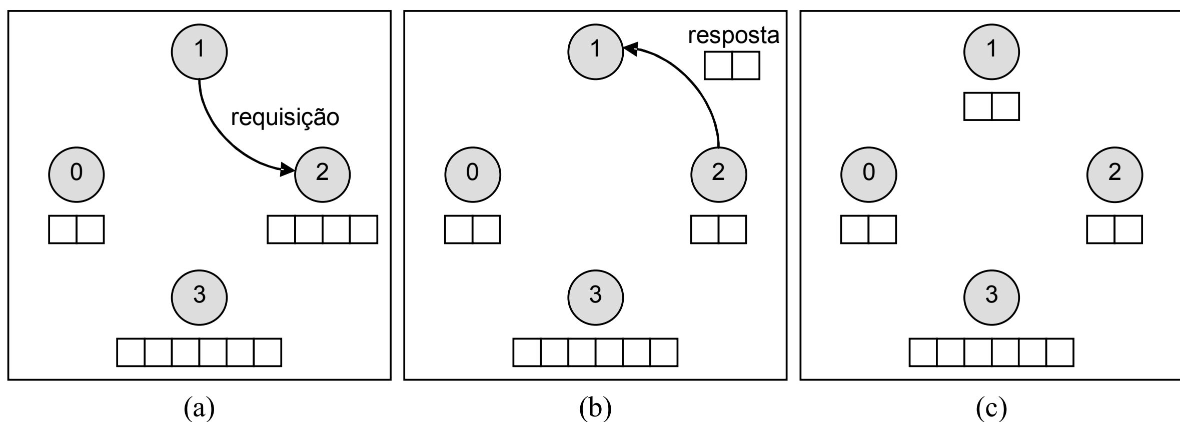


Figura 4.4: Algoritmo do Vizinho mais Próximo.

Os círculos representam processadores e os quadrados representam tarefas na fila de cada processador. (a) O processador 1 envia uma requisição ao vizinho mais próximo 2. (b) O processador 2 envia como resposta metade das tarefas da sua fila. (c) A carga de trabalho é homogeneamente redistribuída entre os processadores 1 e 2.

Este algoritmo, embora seja bastante simples, evita o envio excessivo de mensagens através da rede, permitindo somente a comunicação entre processadores vizinhos. Vizinho, no sentido aqui empregado, refere-se ao próximo processador numa seqüência numerada de processadores. Não necessariamente se refere ao processador fisicamente mais próximo, embora o algoritmo tenha sido empregado em pequenas redes de computadores em que a distância física entre os processadores é praticamente idêntica. A seqüência numerada é considerada como sendo circular, de modo que o vizinho mais próximo do processador (*n* - 1) é o processador 0.

Um potencial problema deste algoritmo encontra-se na localidade das transferências de carga. Se a distribuição inicial de trabalho não for razoavelmente homogênea, podem surgir grupos em seqüência de processadores com muita carga de trabalho e outros grupos distantes com pouca carga. Como cada processador só transfere carga para seu vizinho, esta localidade das transferências pode dificultar o equilíbrio de carga entre processadores que se encontram distantes.

O princípio básico de funcionamento do algoritmo do Vizinho mais Próximo, ou *Nearest Neighbor*, foi originalmente proposto em 8.

4.4 Algoritmo da Fila mais Longa (LQ)

Neste algoritmo, durante a etapa de renderização, cada processador, ao encontrar-se ocioso por ter terminado de processar a última tarefa da sua fila, envia uma mensagem de solicitação de tarefas ao processador com a mais longa fila de tarefas, que responde retirando de sua fila e enviando de volta metade de suas tarefas. O processador solicitante recebe as tarefas, adiciona-as à sua fila, e continua o processamento das tarefas recebidas. Este processo, ilustrado na Figura 4.2, se repete até que todas as tarefas de todos os processadores tenham sido processadas.

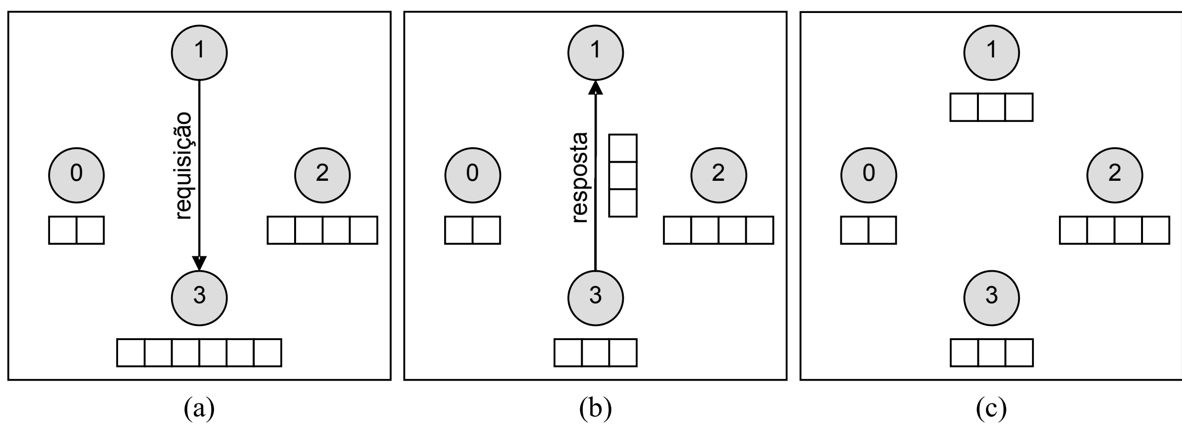


Figura 4.5: Algoritmo da Fila mais Longa.
(a) O processador 1 envia uma requisição ao processador 3, que possui a fila de tarefas mais longa. (b) O processador 3 envia como resposta metade das tarefas da sua fila. (c) A carga de trabalho é homogeneamente redistribuída entre os processadores 1 e 3.

Esta abordagem requer que sejam compartilhadas entre os processadores informações atualizadas sobre os tamanhos de todas as filas de tarefas. Somente com base nestes dados é possível determinar que processador possui a maior fila de tarefas. Para este fim, é usado um mecanismo de *token-ring*.

Um *token* contendo uma estrutura de dados com os tamanhos de todas as filas de tarefas circula entre os processadores, como ilustrado na Figura 4.3. Cada processador, ao receber o *token*, atualiza-o com a informação sobre o tamanho de sua própria fila de tarefas. Se este processador estiver ocioso, ele usa a informação atualizada que acabou de receber para determinar o processador com a maior fila de tarefas, para o qual envia uma solicitação. Depois, repassa o *token* ao próximo processador.

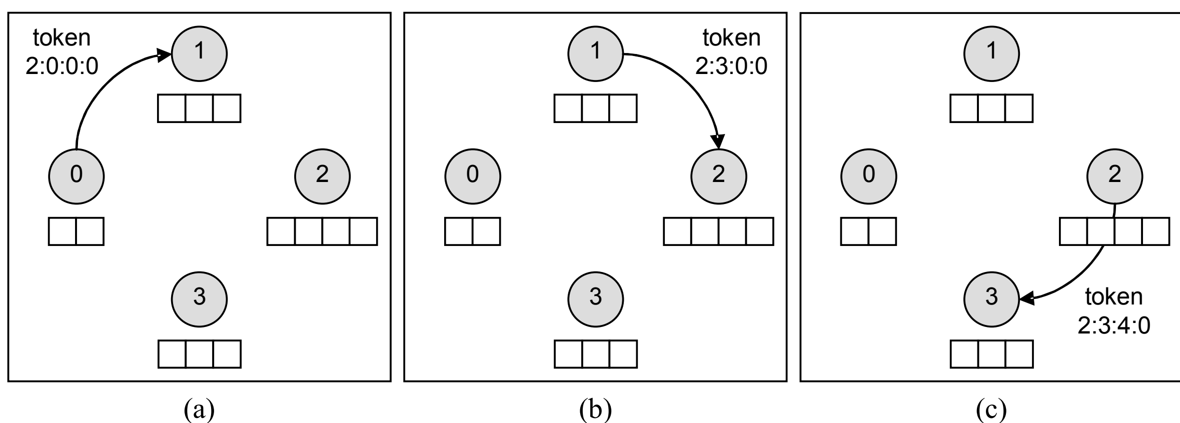


Figura 4.6: Mecanismo de *token-ring*.
(a) Processador 0 atualiza o *token*, informando que sua fila possui 2 tarefas, passando-o a seguir ao processador 1. (b) Processador 1 atualiza o *token*, informando que sua fila possui 3 tarefas, repassando-o a seguir ao processador 2. (c) O processo se repete do processador 2 para o processador 3, e assim sucessivamente.

Este algoritmo evita a localidade das transferências tentando sempre promover o equilíbrio entre os processadores com a menor e com a maior carga de trabalho. Entretanto, como as informações que circulam no *token-ring* podem ficar desatualizadas até que atinjam seu destino, podem acontecer transferências de tarefas entre processadores com pouca carga de trabalho, dificultando o balanceamento de carga.

O maior problema desta abordagem está na dificuldade de se conseguir manter as informações sobre as filas de tarefas realmente atualizadas. Desde o momento em que um processador atualiza o *token* com o tamanho da sua própria fila, até o momento em que outro processador distante recebe o *token* com essa informação, ela pode estar desatualizada. A fila de tarefas pode estar menor, caso nesse intervalo o processador tenha processado alguma tarefa da fila, ou caso tenha enviado tarefas para outro processador após receber uma solicitação. A fila de tarefas também pode estar maior, caso nesse intervalo tenha recebido tarefas de outro processador após o envio de uma solicitação. Digamos que, em um determinado momento, o *token* esteja realmente atualizado, havendo um alvo com uma longa fila de tarefas e que neste momento alguns processadores iniciem o *work stealing*. Naquele momento, todos estes processadores escolhem um mesmo alvo. Ao atender a primeira solicitação, o alvo envia metade das tarefas da sua fila. Ao atender a segunda solicitação, envia metade das tarefas que sobraram, fica com apenas um quarto do tamanho original da fila. Ao atender a terceira solicitação, ficará com apenas um oitavo das suas tarefas. Enquanto isso, o *token* que circula informa a todos os nós que o alvo continua com sua fila de tarefas do tamanho original. Até que o *token* passe novamente pelo alvo e seja atualizado, qualquer processador que inicie o *work stealing* escolhe o mesmo alvo, mesmo que este não tenha mais tarefas para enviar.

Para evitar este problema, o *token* deveria circular muito rapidamente. Mas mesmo que se conseguisse isto, o gerenciamento do *token-ring* passaria a tomar muito tempo do processamento da renderização propriamente dita.

O algoritmo da Fila mais Longa, ou *Longest Queue*, é uma evolução do algoritmo denominado simplesmente de *Token-Ring*, proposto em 8.

4.5 Algoritmo da Distribuição Circular (CD)

Este algoritmo propõe uma forma alternativa para executar a distribuição de tarefas. Um *token* contendo o número da próxima tarefa a ser executada circula entre os processadores, através de um mecanismo de *token-ring*. Cada processador que esteja com a fila de tarefas vazia, ao receber o *token*, deve adicionar a tarefa referenciada no *token* à sua fila de tarefas, incrementar o valor do *token* e repassá-lo ao próximo processador. Caso o processador não esteja com a fila de tarefas vazia, ele deve simplesmente receber o *token* e repassá-lo ao próximo processador imediatamente. Este processo é ilustrado na Figura 4.4.

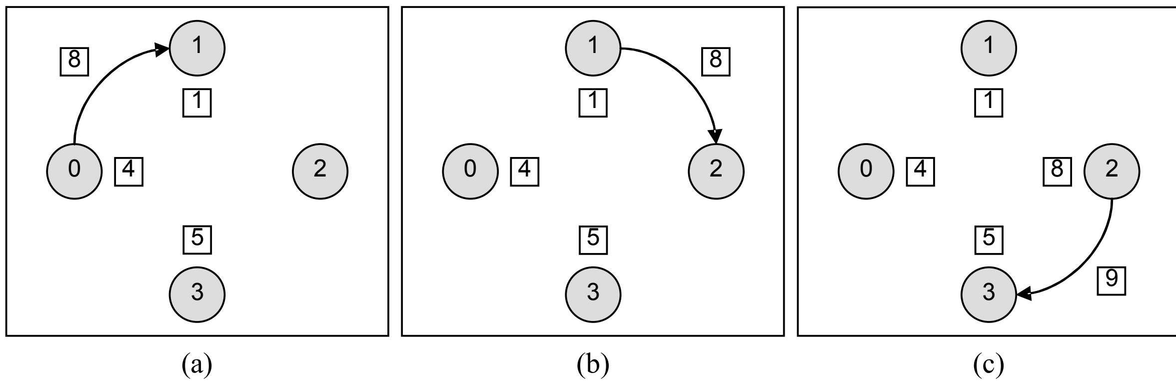


Figura 4.7: Algoritmo da Distribuição Circular.
(a) O processador 0 repassa ao processador 1 o *token* contendo o número da próxima tarefa a ser executada. (b) O processador 1 repassa o *token* ao processador 2, cuja fila de tarefas encontra-se vazia. (c) O processador 2 recebe o *token*, acrescenta a tarefa indicada à sua fila, incrementa o número da próxima tarefa a ser executada, e repassa o *token* ao processador 3.

Enquanto não estiver com a fila de tarefas vazia, cada processador deve executar a renderização processando a tarefa da sua fila. O término da renderização ocorre quando o valor do *token*, incrementado sucessivas vezes, se iguala ao número total de tarefas, indicando que não há mais tarefas para processar.

Busca-se com este algoritmo obter um melhor balanceamento de carga fazendo com que as tarefas sejam distribuídas dinamicamente, uma a uma, durante a renderização, na medida em que cada processador termina de renderizar sua tarefa atual. Evita-se com isso que um processador fique com uma longa fila de tarefas ao mesmo tempo em que outro estaria ocioso. O principal problema consiste na velocidade de circulação do *token-ring*, que caso seja baixa demais pode fazer com que um *token* demore a chegar até um processador ocioso, desperdiçando tempo.

O algoritmo da Distribuição Circular, ou *Circular Distribution*, foi originalmente proposto em 8.

CAPÍTULO 5

IMPLEMENTAÇÃO

Para avaliar os algoritmos de balanceamento de carga propostos, utilizamos como base o algoritmo *Parallel ZSweep* (*PZSweep*) 8, que é um algoritmo de renderização paralela projetado para execução em sistemas com arquitetura de memória compartilhada. O *PZSweep* implementa a paralelização do algoritmo *Out-Of-Core ZSweep* (*OOC-ZSweep*) 8, um eficiente algoritmo de visualização volumétrica *out-of-core*.

5.1 Paralelização do Algoritmo de Visualização Volumétrica

O algoritmo *OOC-ZSweep* divide a tela horizontalmente e verticalmente produzindo regiões retangulares denominadas *tiles*. Cada *tile* representa uma unidade computacional de trabalho e recebe um número de identificação. A partir do número de identificação de um *tile*, é possível determinar que parte do conjunto de dados deve ser renderizada e que região retangular da tela corresponde à sub-imagem resultante da renderização. A Figura 5.1 ilustra a divisão da tela em *tiles*.

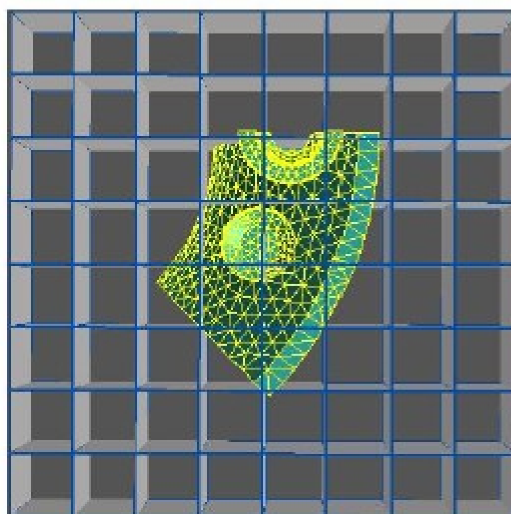


Figura 5.8: Divisão em 8 por 8 de conjunto de dados volumétricos, produzindo 64 *tiles*.

Neste algoritmo, o conjunto de dados de entrada é particionado numa etapa de pré-processamento para produzir uma representação hierárquica dos dados numa estrutura denominada *octree*. As células e vértices do conjunto de dados são distribuídos nas folhas da *octree*. O tempo de pré-processamento gasto na geração da *octree* é desprezível se comparado ao tempo total de renderização. Usando uma única *octree* é possível visualizar o mesmo conjunto de dados em vários ângulos diferentes de rotação.

Como a estrutura *octree* é carregada inteiramente na memória, com uma busca simples na *octree* é possível determinar que folhas se projetam dentro da região da tela associada a um determinado *tile*, determinando assim os vértices e células dentro deste *tile*.

A rederização de um *tile* então se procede: encontrando as folhas da *octree* que interceptam este *tile*; determinando os vértices de todas as faces que interceptam quaisquer das folhas encontradas; e projetando as faces na tela em ordem de profundidade com o algoritmo seqüencial *ZSweep* 8.

Na paralelização do *OOC-ZSweep*, os *tiles* são renderizados individualmente em diferentes processadores. Cada *tile* renderizado produz uma sub-imagem correspondente a uma região da tela. As sub-imagens geradas devem ser coletadas por um único processador, chamado coletor, onde são coladas para produzir a imagem final completa.

O algoritmo *PZSweep* foi o primeiro a implementar a paralelização do *OOC-ZSweep*, usando um algoritmo centralizado de balanceamento de carga, baseado em uma única fila de tarefas contendo os *tiles* a serem renderizados. Esta fila é gerenciada por um processador mestre, que distribui um a um cada *tile* aos demais processadores, chamados de escravos, que executam a renderização e enviam de volta cada sub-imagem produzida ao processador mestre, que também atua como coletor.

Nossos algoritmos propostos – Vizinho mais Próximo (NN), Fila mais Longa (LQ) e Distribuição Circular (CD) – apresentam estratégias distribuídas para a paralelização e balanceamento de carga do algoritmo *OOC-ZSweep*, em contraste com a estratégia centralizada no processador mestre do algoritmo PZSweep. A fila única de tarefas do modelo mestre-escravo foi substituída por um esquema de distribuição inicial estática de tarefas, combinada com três estratégias distintas de redistribuição dinâmica de trabalho, uma para cada algoritmo.

5.2 Fases da Execução

A implementação dos algoritmos aqui propostos na versão paralela do *OOC-ZSweep* requer a divisão da sua execução em seis fases diferentes: seleção, distribuição, renderização, balanceamento, finalização e coleta. Cada uma destas fases está descrita e definida a seguir:

- **Seleção**: define quais *tiles* devem ser incluídos na fila de tarefas para processamento. Normalmente todos os *tiles* devem ser renderizados, mas muitos deles podem estar vazios e, dessa forma, podem ser removidos da fila através de um trabalho inicial de seleção;
- **Distribuição**: define como os *tiles* são distribuídos entre os processadores;
- **Renderização**: é a fase principal, na qual o algoritmo *OOC-ZSweep* é usado para processar os dados de entrada em três dimensões e gerar uma imagem de saída em duas dimensões. Cada *tile* é renderizado separadamente dos demais;
- **Balanceamento**: é a fase onde são empregados os algoritmos propostos. Esta fase define como alguns *tiles* são transferidos entre filas de tarefas de diferentes nós durante a fase de renderização, com o objetivo de equilibrar a carga de trabalho de todos os processadores, visto que cada *tile* exige uma quantidade de processamento diferente;
- **Finalização**: define um método para que todos os nós possam determinar de forma distribuída que não há mais nenhum *tile* a ser renderizado em nenhum outro nó, encerrando a fase de renderização; e
- **Coleta**: define como os *tiles* renderizados em cada processador são enviados ao processador coletor, que coleta e cola esses *tiles* para formar uma única imagem completa.

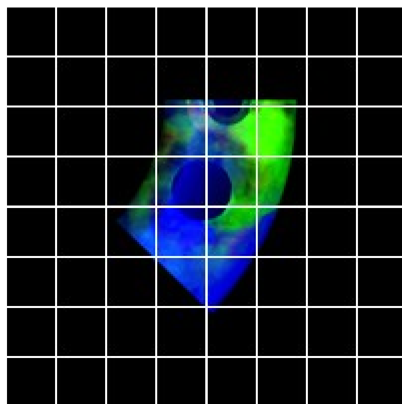
5.2.1 Seleção de *Tiles*

Imagens de objetos renderizados costumam conter muitas áreas vazias, exibindo apenas a cor de fundo. Isso ocorre por exemplo, quando o objeto renderizado for razoavelmente menor que a tela. Nestes casos, a divisão da tela faz com que muitos *tiles* fiquem vazios, ou seja, não contenham nenhuma parte do objeto a ser renderizado.

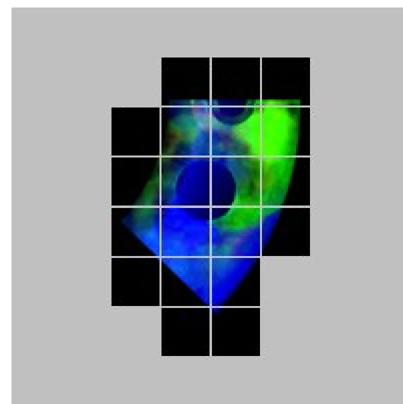
Esses *tiles* vazios constituem um problema para uma distribuição de *tiles* equilibrada. Mesmo que dois processadores recebam uma quantidade igual de *tiles* para renderizar, um deles pode receber uma quantidade bem maior de *tiles* vazios, causando um grande desequilíbrio na distribuição da carga de trabalho.

A fase de seleção determina quais *tiles* devem ser renderizados. São apresentadas duas possibilidades para a fase de seleção: todos os *tiles* ou *tiles* não vazios.

A Figura 5.1 compara os métodos de seleção de *tiles* aplicados a uma imagem dividida em 8 x 8. A seleção de *tiles* não vazios elimina a maior parte dos *tiles*, já que muitos deles não possuem interseções com objetos da cena.



(a) selecionar todos os *tiles*



(b) selecionar os *tiles* não vazios

Figura 5.9: Métodos de seleção, para uma imagem dividida em 8 x 8.
Com o método (a), todos os 64 *tiles* são selecionados. Com o método (b), apenas 20.

5.2.1.1 Seleção de Todos os *Tiles*

No primeiro método, todos os *tiles* são adicionados à fila de tarefas para renderização. Este modo é o mais simples e não apresenta nenhum custo adicional de processamento para o sistema. Entretanto, pode causar um maior desbalanceamento de carga no sistema.

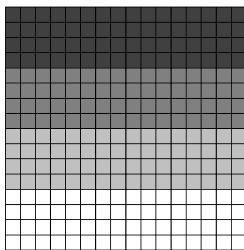
5.2.1.2 Seleção de *Tiles* Não Vazios

No segundo método, um processador, chamado distribuidor, deve analisar todos os *tiles* e determinar quais estão vazios. Apenas os *tiles* não vazios são distribuídos igualmente entre os processadores. Desta forma, a distribuição da carga de trabalho entre os processadores pode ser mais homogênea. Como desvantagem, este método envolve um custo adicional de pré-processamento. Entretanto, esta desvantagem é minimizada pelo fato de que é possível determinar se um *tile* está vazio a um custo muito baixo, com uma operação de consulta à estrutura de dados *octree*, que é mantida na memória.

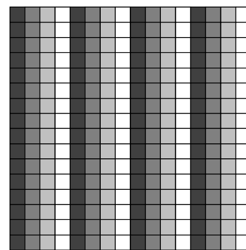
5.2.2 Distribuição de Tarefas

Propomos quatro métodos de distribuição de tarefas, sendo um dinâmico – distribuição circular – e três estáticos – distribuição contínua, distribuição intercalada e distribuição aleatória.

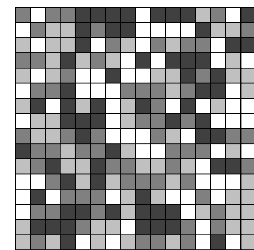
O método dinâmico da Distribuição Circular (CD) foi apresentado no Capítulo 4. Neste capítulo apresentamos três métodos de distribuição estática: contínua, intercalada e aleatória. Os métodos de distribuição estática, ilustrados na Figura 5.3, são usados pelos algoritmos do Vizinho mais Próximo (NN) e da Fila mais Longa (LQ). A seguir discutimos os métodos de distribuição estática.



(a) distribuição contínua



(b) distribuição intercalada



(c) distribuição aleatória

Figura 5.10: Métodos de distribuição estática, usando 4 nós e divisões em 16 x 16 tiles. Cada cor representa um nó. Cada *tile* foi representado na cor do nó que o recebeu.

5.2.2.1 Distribuição Estática Contínua

Neste método, cada processador monta sua própria lista de tarefas. Inicialmente é calculada a quantidade x de *tiles* que cada processador deve receber com base na razão entre o número total de *tiles* e o número de nós. Os *tiles* são então distribuídos de modo contínuo entre os nós. O nó 0 monta sua fila de tarefas com os primeiros x *tiles*, o nó 1 monta sua fila com os próximos x *tiles*, e assim sucessivamente, de modo que o nó (n - 1) monta sua fila com os últimos x *tiles*. Caso o número total de *tiles* não seja múltiplo do número de nós, a distribuição é feita de modo que os primeiros nós, em número equivalente ao resto da divisão, recebem (x + 1) *tiles*, enquanto os demais nós recebem x *tiles*.

No caso da seleção prévia dos *tiles* não vazios, a distribuição contínua ocorre de forma ligeiramente diferente. Um processador, chamado distribuidor, determina quais são os *tiles* não vazios e calcula a quantidade x de *tiles* que cada processador deve receber com base na razão entre o número de *tiles* não vazios e o número de nós. A seguir, envia para o nó 0 os primeiros x *tiles*, para o nó 1 os próximos x *tiles*, e assim sucessivamente. Cada nó monta sua lista de tarefas com base nos *tiles* recebidos do processador distribuidor. Aplica-se a mesma consideração do parágrafo anterior caso o número de *tiles* não vazios não seja múltiplo do número de nós.

Em muitas imagens produzidas por renderização, áreas contínuas no espaço tendem a ter uma densidade semelhante de objetos. Deste modo, pela distribuição contínua alguns processadores podem receber *tiles* contínuos de áreas pouco densas da tela, enquanto que outros processadores podem receber *tiles* contínuos de áreas muito densas da tela, causando um grande desequilíbrio de carga de trabalho no sistema.

5.2.2.1 Distribuição Estática Intercalada

Uma das tentativas de diminuir o desbalanceamento de carga produzido pela distribuição contínua é a distribuição intercalada. Neste método, cada processador monta sua própria lista de tarefas. Os *tiles* são distribuídos de forma intercalada entre os nós, de modo que o *tile* de número 0 seja adicionado à lista de tarefas do nó 0, o *tile* 1 para o nó 1, o *tile* 2 para o nó 2, e assim sucessivamente. Considerando um total de n nós: o nó 0 monta sua lista de tarefas com os *tiles* de número 0, n, $2n$, $3n$, ...; o nó 1 monta sua lista com os *tiles* 1, $n + 1$, $2n + 1$, $3n + 1$, ...; o nó 2 monta sua lista com os *tiles* 2, $n + 2$, $2n + 2$, $3n + 2$, e assim por diante.

No caso da seleção prévia dos *tiles* não vazios, a distribuição intercalada ocorre de forma ligeiramente diferente. O processador distribuidor determina quais são os *tiles* não vazios e, dentre os *tiles* não vazios selecionados, envia para o nó 0 os *tiles* 0, n, $2n$, ...; para o nó 1 os *tiles* 1, $n + 1$, $2n + 1$, ...; e assim sucessivamente. Cada nó monta sua lista de tarefas com base nos *tiles* recebidos do processador distribuidor.

A distribuição intercalada, em geral, produz uma distribuição um pouco mais homogênea da carga de trabalho entre os nós do que a distribuição contínua. Mesmo assim, não produz uma distribuição de carga muito bem equilibrada. Quando o número de divisões é múltiplo do número de nós, por exemplo, a única diferença entre a distribuição contínua e a intercalada é que, na primeira, cada processador recebe uma larga faixa horizontal contínua da tela para renderizar, enquanto que, na segunda, cada processador recebe algumas faixas verticais contínuas com um *tile* de largura cada.

5.2.2.3 Distribuição Estática Aleatória

Um método com maior probabilidade de produzir uma distribuição estática com carga de trabalho razoavelmente equilibrada entre os nós é a distribuição aleatória. Neste método, o processador distribuidor monta uma lista de *tiles* a serem distribuídos usando um dos dois métodos de seleção descritos anteriormente. A seguir, reposiciona aleatoriamente os *tiles* na lista. Calcula então a quantidade x de *tiles* que cada processador deve receber com base na razão entre o número de *tiles* selecionados e o número de nós. Por último, envia para o nó 0 os primeiros x *tiles*, para o nó 1 os próximos x *tiles*, e assim sucessivamente. Cada nó monta sua lista de tarefas com base nos *tiles* recebidos do processador distribuidor. Caso o número total de *tiles* selecionados não seja múltiplo do número de nós, a distribuição é feita de modo que os primeiros nós, em número equivalente ao resto da divisão, recebem $(x + 1)$ *tiles*, enquanto os demais nós recebem x *tiles*.

Este método, ao contrário das distribuições contínua e intercalada, depende sempre da atuação do processador distribuidor na fase de distribuição. Ele apresenta uma probabilidade maior de produzir uma distribuição estática de carga mais equilibrada, embora adicione um pequeno custo de pré-processamento e tráfego de rede à fase de distribuição.

5.2.3 Renderização

Esta é a fase principal. Ocorre em cada nó de forma independente. O processador retira um *tile* da fila de tarefas de cada vez e executa sua renderização usando o algoritmo *Out-Of-Core ZSweep*.

O resultado da renderização de um *tile* é uma sub-imagem, que na fase de coleta é enviada ao processador coletor para combinar com as demais sub-imagens e produzir a imagem final completa. A renderização prossegue processando cada um dos *tiles* remanescentes até que não haja mais *tiles* para renderizar.

5.2.4 Balanceamento de Carga

A fase de balanceamento de carga ocorre simultaneamente com a fase de renderização. Uma *thread* secundária é implementada para executar o balanceamento de carga, enquanto a *thread* principal executa a renderização. A estratégia de balanceamento de carga adotada pode ser baseada na distribuição dinâmica de tarefas, usada no algoritmo da **Distribuição Circular (CD)**, ou na distribuição inicial estática de tarefas combinada com mecanismos de *work stealing*, usada nos algoritmos do **Vizinho mais Próximo (NN)** e da **Fila mais Longa (LQ)**.

Duas questões relacionadas ao balanceamento de carga baseado em *work stealing* devem ser respondidas na implementação do algoritmo: quando iniciar o *work stealing* e para que nó enviar uma solicitação.

O início do *work stealing* pode ser determinado por um limite inferior de *tiles* na fila de tarefas, ou seja, quando o tamanho da fila de tarefas for inferior a um limite pré-estabelecido, o nó deve buscar *tiles* em outros nós. Adotamos o valor zero como limite inferior, ou seja, apenas quando a fila de tarefas estiver vazia é iniciado o balanceamento de carga. Desta forma, garante-se que um nó sempre busca *tiles* em outro nó com maior carga de trabalho.

Que nó deve receber uma solicitação de tarefas é a segunda questão a ser respondida. Apresentamos a seguir três propostas para responder esta questão: requisitar a um nó fixo; requisitar a um nó aleatório; requisitar a um nó selecionado durante a execução.

5.2.4.1 Requisitar a um Nó Fixo

Neste método cada nó direciona todas as suas solicitações para um único outro nó previamente estabelecido. Esta estratégia foi adotada no algoritmo do **Vizinho mais Próximo (NN)**, em que cada nó de número i envia solicitações sempre para o nó $i + 1$. Ou seja, o nó 0 solicita ao nó 1, que solicita ao nó 2, que solicita ao nó 3, e assim sucessivamente, sendo que o nó $n - 1$ solicita ao nó 0.

5.2.4.2 Requisitar a um Nó Aleatório

Neste método o alvo do *work stealing* é escolhido de forma completamente aleatória dentre todos os nós, exceto obviamente o próprio solicitante. Para cada solicitação, um novo alvo aleatório é escolhido usando a função *rand()* da linguagem C++.

Procura-se com isso eliminar a localidade das transferências de *tiles*, dispersando-as aleatoriamente por todos os nós. Todos os nós podem transferir *tiles* entre si. Entretanto, a eficiência deste método depende de que os alvos escolhidos aleatoriamente sejam também os alvos ideais, ou seja, nós com longas filas de tarefas. E muitas vezes não é o que ocorre. Um processador ocioso pode escolher aleatoriamente como alvo, por exemplo, outro processador ocioso, mesmo que existam vários processadores com *tiles* disponíveis em suas filas de tarefas.

Testes preliminares indicaram que este método não apresentou resultados satisfatórios, se comparado com os demais métodos propostos, sendo, portanto, descartado como método de balanceamento de carga.

5.2.4.3 Requisitar a um Nó Selecionado Durante a Execução

Neste método algum critério objetivo de seleção deve ser usado para a escolha do alvo do *work stealing*. Esta estratégia foi adotada como base do algoritmo da **Fila mais Longa (LQ)**, em que é selecionado como alvo o nó com a maior fila de tarefas.

5.2.5 Finalização da Renderização

Durante a execução do sistema, haverá um momento em que todos os *tiles* terão sido renderizados, não havendo mais trabalho de renderização a realizar. Este momento define o término da fase de renderização. A fase de finalização define o método através do qual todos os nós podem determinar quando a fase de renderização se encerrou.

Neste trabalho propomos duas alternativas para a fase de finalização distribuída: finalização com primeira resposta negativa e finalização baseada em *token-ring*.

5.2.5.1 Finalização com Primeira Resposta Negativa

Quando um nó inicia o *work stealing*, ele envia uma mensagem de solicitação a outro nó escolhido como alvo. O nó alvo pode responder à solicitação de dois modos: enviando uma mensagem contendo uma parte de seus *tiles*, caso sua fila de tarefas esteja cheia - uma resposta positiva; ou enviando algum tipo de mensagem de retorno indicando que não possui *tiles* para fornecer - uma resposta negativa.

Na abordagem da finalização baseada na primeira resposta negativa, cada nó inicia o *work stealing* e aguarda uma resposta. Caso receba uma resposta positiva, continua a renderização com os *tiles* recebidos, podendo inclusive enviar novas solicitações caso seja necessário mais adiante. Mas caso receba uma única resposta negativa, termina a renderização.

Este é um método simples, mas pouco eficiente. O fato de um nó ter recebido uma única resposta negativa não significa que não haja mais *tiles* para renderizar em outros nós. Cada nó que receber a primeira resposta negativa finaliza a renderização e permanece ocioso, enquanto os demais nós continuam renderizando, limitando demais a eficiência do balanceamento de carga.

Para um balanceamento de carga ideal, cada nó somente deveria desistir de solicitar *tiles* quando tivesse certeza de que não há mais *tiles* para renderizar em nenhum outro nó. Para isso é necessário que cada nó saiba quantos *tiles* já foram renderizados em cada um dos outros nós. Quando a soma dos *tiles* renderizados em cada nó for igual ao total de *tiles*, a fase de renderização pode ser terminada. Faz-se necessário que todos os nós troquem entre si informações sobre a quantidade de *tiles* renderizados por cada nó. Para implementar esta idéia é proposto o método de finalização baseado em *token-ring*.

5.2.5.2 Finalização Baseada em *Token-Ring*

Neste método existe uma estrutura de dados contendo informações sobre a quantidade de *tiles* renderizados em cada um dos nós. Esta informação é atualizada e compartilhada entre os nós através do mecanismo de *token-ring*.

A própria estrutura de dados constitui o *token*, que é passado de nó a nó através do algoritmo *token-ring*. Cada nó, ao receber o *token*, atualiza os dados com informações sobre a quantidade de *tiles* que ele renderizou. Calcula, então, a soma das quantidades de *tiles* renderizados em todos os nós. Caso a soma seja igual ao total de *tiles*, a fase de renderização é finalizada e o nó que estiver com o *token* envia uma mensagem de finalização a todos os demais nós. Caso contrário, o nó passa o *token* adiante.

Esta abordagem garante que o balanceamento de carga seja executado em todos os nós até que não existam mais *tiles* disponíveis para renderização em nenhum dos nós.

5.2.6 Coleta Final

Nesta fase, os *tiles* que foram renderizados em cada nó produziram sub-imagens correspondentes a diferentes regiões da tela, que precisam ser enviadas para o processador coletor. Este, por sua vez, coleta as sub-imagens recebidas e as combina, através de um simples processo de colagem, para produzir uma única imagem completa, que posteriormente é armazenada em disco como um arquivo de saída.

Um mesmo processador pode ser usado como distribuidor e coletor, gerenciando tanto a fase de distribuição estática de tarefas quanto a fase de coleta final de sub-imagens. Nossas implementações usam arbitrariamente o processador de número zero para desempenhar as funções de distribuidor e coletor.

Neste trabalho, são propostas três alternativas para executar a coleta final: coleta de tela inteira; coleta de *tiles* não vazios; e coleta simultânea. As duas primeiras ocorrem após a fase de renderização, enquanto que na última a coleta ocorre durante a fase de renderização, na medida em que os *tiles* vão sendo renderizados.

5.2.6.1 Coleta de Tela Inteira

Este é um método bem simples de ser implementado. Cada processador possui um *buffer* de tela onde a imagem, ou parte dela, é gerada durante a renderização. Ao terminar a renderização, cada processador envia o conteúdo inteiro do seu *buffer* de tela para o processador coletor. O processador coletor recebe os *buffers* de tela inteira de cada um dos outros processadores, e os combina com seu próprio *buffer* de tela, aplicando a operação *OR* sobre o valor de cada um dos *pixels*, produzindo a imagem completa.

Uma vantagem deste método é que o próprio *buffer* de tela pode ser usado, também, como *buffer* de mensagem, evitando cópias de dados na memória durante o envio e o recebimento de mensagens. Além disso, o processo de montagem da imagem final é simplificado pelo uso do operador *OR* na combinação dos *buffers* de tela.

Por outro lado, embora o *buffer* de tela de cada processador contenha espaço de memória suficiente para armazenar toda a imagem, este *buffer* é usado em cada nó para gerar apenas alguns pedaços da imagem, correspondentes aos *tiles* processados localmente. Isto significa que, ao final da renderização, em cada um dos nós, o *buffer* de tela contém muito mais áreas vazias, correspondentes a *tiles* que foram renderizados por outros nós, do que pedaços da imagem final. Mesmo assim, na coleta de tela inteira, os *buffers* completos são enviados através da rede, com todas as suas áreas vazias consumindo desnecessariamente tempo de tráfego de dados e tempo de combinação de *pixels* vazios no processador coletor.

5.2.6.1 Coleta de *Tiles* Não Vazios

Uma alternativa potencialmente mais eficiente do que a coleta de tela inteira é a coleta de *tiles* não vazios. Durante a renderização, o número de identificação de cada *tile* determinado como sendo não vazio é acrescentado a uma lista de *tiles* renderizados. Os *tiles* que foram submetidos à renderização mas que foram identificados como vazios, não são acrescentados à lista.

Ao final da renderização, cada nó monta um *buffer* para transmissão de mensagem da seguinte forma: para cada item da lista de *tiles* renderizados, o *buffer* recebe o número identificador do *tile* e o conteúdo da sub-imagem correspondente ao *tile*, copiada do *buffer* de tela. O *buffer* de mensagem é, então, transmitido ao processador coletor.

O processador coletor recebe os *buffers* de mensagem de cada um dos outros processadores. Para cada *buffer* de mensagem recebido, o processador coletor determina com base no tamanho do *buffer* quantos identificadores de *tiles* ele contém. Então, para cada *tile* recebido, ele lê do *buffer* o número identificador do *tile* e o conteúdo da sub-imagem correspondente, que é então colada no *buffer* de tela local do processador coletor. Ao final, o processador coletor terá recebido mensagens contendo o conteúdo de todas as regiões não vazias da tela, que combinadas resultaram na imagem completa.

O número identificador do *tile* é usado para determinar a localização exata da região da tela correspondente ao *tile*, de modo a permitir operações precisas de recorte e colagem de sub-imagens.

A vantagem deste método sobre a coleta de tela inteira é que apenas as regiões da tela efetivamente renderizadas são enviadas através da rede, diminuindo o tempo gasto com tráfego de dados e cópia de *pixels*, e, conseqüentemente, diminuindo o tempo gasto com a fase de coleta.

5.2.6.2 Coleta Simultânea

Este método funciona de forma semelhante à coleta de *tiles* não vazios, de modo que apenas as regiões da tela correspondentes aos *tiles* efetivamente renderizados são enviadas ao processador coletor.

A diferença é que, na coleta simultânea, os dados são enviados ao processador coletor na medida em que os *tiles* vão sendo renderizados. Uma *thread* secundária é criada para enviar as regiões da tela já renderizadas ao processador coletor, ao mesmo tempo em que a *thread* principal renderiza mais *tiles*. No processador coletor, a *thread* secundária coleta as sub-imagens recebidas que são coladas no *buffer* de tela local, ao mesmo tempo em que a *thread* principal executa a renderização. A fase de coleta ocorre simultaneamente com a fase de renderização.

Quando um processador termina a renderização e envia todos os *tiles* renderizados, envia ao processador coletor uma mensagem de finalização. O processador coletor só encerra a coleta após receber mensagens de finalização de todos os demais processadores.

Uma vantagem deste método é que a fase de coleta terminará quase que simultaneamente com a fase de renderização, pois a maioria dos *tiles* renderizados já terão sido enviados e coletados antes do final da renderização. Nos outros métodos, a coleta somente se inicia quando a renderização termina. Por outro lado, a *thread* secundária responsável pelo envio e coleta compete por tempo de processamento com a *thread* principal, tornando mais lenta a renderização.

5.3 Implementação dos Algoritmos

A implementação da renderização paralela baseada no algoritmo *OOC-ZSweep* em um sistema distribuído, segundo as fases de execução apresentadas anteriormente, foi realizada de forma modular em linguagem C++ e totalmente orientada a objetos. Dessa forma, os diferentes métodos utilizados para seleção de *tiles*, distribuição de tarefas, balanceamento de carga e coleta final podem ser implementados e avaliados independentemente. Além disso, o sistema é flexível para a inclusão de novos métodos e algoritmos para as suas diferentes fases de execução. A seguir, apresentamos nosso sistema como uma biblioteca de classes.

5.4 A Biblioteca *PZSweep*

Chamamos de biblioteca *PZSweep* ao conjunto de classes desenvolvido com o objetivo de encapsular as operações paralelas no *OOC-ZSweep* e permitir a fácil reutilização das principais rotinas. Ela foi escrita em complementação à biblioteca gráfica *ZSweep* (que contém o código do algoritmo de renderização volumétrica) e contém métodos para envio e recebimento de mensagens, distribuição, coleta, renderização de *tiles*, gerenciamento da fila de tarefas, gerenciamento de *threads*, *work stealing*, compartilhamento de informações, monitoração de tempo, geração de *logs*, entre outros procedimentos.

Na biblioteca *PZSweep* são definidas quatro classes, das quais apenas a primeira precisa ser instanciada no trecho de código principal: *PZSweep*, *WorkInfo*, *RandomList* e *ListItem*. A classe *PZSweep* é a classe principal da biblioteca e fornece os métodos necessários para a construção dos algoritmos de balanceamento de carga. As demais classes são usadas internamente pela classe *PZSweep*. A classe *WorkInfo* fornece acesso a informações sobre a quantidade de *tiles* na fila de tarefas e a quantidade de *tiles* renderizados em cada processador. A classe *RandomList* é uma estrutura de dados usada para armazenar a fila de tarefas, possuindo ainda um método de distribuição aleatória de seus itens. A classe *ListItem* representa um item da fila de tarefas, contendo informações sobre o número de identificação do *tile* e sua posição na fila.

5.4.1 Usando a Biblioteca PZSweep

Para usar a biblioteca PZSweep, o arquivo de cabeçalho C++ correspondente deve ser adicionado ao código fonte através da inclusão da linha:

```
#include "pzs_pzsweep.hh"
```

Todos os arquivos de cabeçalho e de código fonte da biblioteca PZSweep são identificados por nomes iniciados pelo prefixo "pzs_", como ilustrado a seguir:

```
pzs_define.hh          pzs_pzsweep.hh        pzs_thread.cpp
pzs_distribute.cpp     pzs_randomlist.cpp    pzs_workdone.cpp
pzs_extern.hh          pzs_randomlist.hh     pzs_workinfo.cpp
pzs_message.cpp        pzs_render.cpp        pzs_worktodo.cpp
pzs_pzsweep.cpp
```

O procedimento principal da renderização paralela é a função *work_loop*:

```
void work_loop(Octree *octree, ViewPlane *view,
        char *tfname, float *brightness);
```

Dentro da função *work_loop* deve ser instanciada a classe *PZSweep*, de modo que a estrutura básica de qualquer aplicação usando a biblioteca PZSweep pode ser definida pelo seguinte trecho de código:

```
#include "pzs_pzsweep.hh"

void work_loop(Octree *octree, ViewPlane *view,
        char *tfname, float *brightness)
{
        PZSweep pzsweep(octree, view, tfname, brightness);
        // insert the code here
}
```

5.4.2 A Classe PZSweep

A seguir apresentamos a descrição detalhada da classe *PZSweep* e a definição de seus métodos públicos:

```
class PZSweep {
public:
        PZSweep(Octree *octree, ViewPlane *view,
                char *tfname, float *brightness);
        ~PZSweep();

        bool isEmptyTile(int tile);
        bool renderTile(int tile);
        bool renderTile();

        void addWorkToDo(int tile);
        bool hasWorkToDo();
        int  getWorkToDo();
        int  sendWorkToDo(int node);
        int  sendWorkToDo(int node, int count);
        int  recvWorkToDo(int node);

        void addWorkDone(int tile);
        bool hasWorkDone();
        int  sendWorkDone(int node);
        int  sendWorkDone(int node, int tile);
        int  recvWorkDone(int node);
        void sendScreen(int node);
        void recvScreen(int node);
```

```
	void sendWorkInfo(int node);
	void recvWorkInfo(int node);
	int countWorkToDo();
	int countWorkDone();
	int getWorkToDoInfo(int node);
	int getWorkDoneInfo(int node);
	int getNodeWithMaxWorkToDo();
	int getNodeWithMinWorkToDo();

	enum distribution {
		continuous,
		interleaved,
		random
	};

	void distribute(distribution mode);
	void distributeNonEmpty(distribution mode);

	void createThread(void *(*function)(void *));
	void waitThread();
	void stop();
	void go();
	bool isStopped();

	enum message {
		msgRequest,
		msgWorkToDo,
		msgWorkDone,
		msgWorkInfo,
		msgScreen,
		msgTime,
		msgQuit
	};

	void sendMessage(void *buffer, int count, MPI_Datatype type,
		int dest, message tag);
	void sendMessage(int dest, message tag);
	void recvMessage(void *buffer, int count, MPI_Datatype type,
		int source, message tag);
	void recvMessage(int source, message tag);
	void waitMessage(int *source, message tag);
	bool hasMessage(int *source, message tag);
};
```

A seguir são descritos resumidamente os membros da classe *PZSweep*:

- *PZSweep*: construtor da classe. Recebe os mesmos parâmetros fornecidos pela biblioteca *ZSweep* quando da chamada da função *work_loop*.
- *~PZSweep*: destruidor da classe. Executa medições de tempo e gera arquivos de imagem e de *log* com monitoração de tempo.

- *isEmptyTile*: determina se um *tile* é vazio ou não.
- *renderTile*: renderiza um *tile* chamando a função equivalente da biblioteca *ZSweep*.
- *addWorkToDo*: adiciona um *tile* à fila de tarefas.
- *hasWorkToDo*: determina se a fila de tarefas não está vazia.
- *getWorkToDo*: retira um *tile* da fila de tarefas.
- *sendWorkToDo*: envia *tiles* a outro nó.
- *recvWorkToDo*: recebe *tiles* de outro nó.
- *addWorkDone*: adiciona um *tile* à lista de *tiles* renderizados.
- *hasWorkDone*: determina se a lista de *tiles* renderizados não está vazia.
- *sendWorkDone*: envia a outro nó o conteúdo do *buffer* de tela correspondente aos *tiles* renderizados.
- *recvWorkDone*: recebe de outro nó o conteúdo do *buffer* de tela correspondente aos *tiles* renderizados.
- *sendScreen*: envia a outro nó todo o conteúdo do *buffer* de tela.
- *recvScreen*: recebe de outro nó todo o conteúdo do *buffer* de tela.
- *sendWorkInfo*: envia a outro nó informações sobre o número de *tiles* de cada fila de tarefas e o número de *tiles* renderizados em cada nó.
- *recvWorkInfo*: recebe de outro nó informações sobre o número de *tiles* de cada fila de tarefas e o número de *tiles* renderizados em cada nó.
- *countWorkToDo*: calcula a soma dos *tiles* nas filas de tarefas de todos os nós.
- *countWorkDone*: calcula a soma dos *tiles* renderizados em todos os nós.
- *getWorkToDoInfo*: retorna o tamanho da fila de tarefas de um determinado nó.
- *getWorkDoneInfo*: retorna o número de *tiles* renderizados por um determinado nó.
- *getNodeWithMaxWorkToDo*: determina qual nó possui a maior fila de tarefas.
- *getNodeWithMinWorkToDo*: determina qual nó possui a menor fila de tarefas.
- *distribute*: distribui todos os *tiles* entre os nós.
- *distributeNonEmpty*: distribui apenas os *tiles* não vazios entre os nós.
- *createThread*: cria uma *thread* secundária.
- *waitThread*: aguarda o fim da execução da *thread* secundária.
- *stop*: suspende a execução da *thread* atual.
- *go*: continua a execução da outra *thread*.
- *isStopped*: determina se a execução da outra *thread* foi suspensa.
- *sendMessage*: envia uma mensagem.
- *recvMessage*: recebe uma mensagem.
- *waitMessage*: suspende a execução da *thread* atual até a chegada de uma mensagem.

• *hasMessage*: determina se há alguma uma mensagem no *buffer* para ser recebida.

5.4.3 Implementação do Algoritmo Seqüencial *OOC-ZSweep*

Com a biblioteca *PZSweep*, a implementação do algoritmo seqüencial *OOC-ZSweep* é realizada de forma simples e elegante com os seguintes comandos:

```
#include "pzs_pzsweep.hh"

void work_loop(Octree *octree, ViewPlane *view,
        char *tfname, float *brightness)
{
        PZSweep pzsweep(octree, view, tfname, brightness);

        NODES = 1;
        if (rank == MASTER)
                for (int tile = 0; tile < TILES; tile++)
                        pzsweep.renderTile(tile);
}
```

O código é auto-explicativo: renderize cada *tile* um a um. Mesmo que este simples algoritmo não utilize computação paralela, a biblioteca *PZSweep* pode simplificar o código executando automaticamente as rotinas de monitoração de tempo parcial e total, geração do arquivo de imagem e geração do arquivo de *log* contendo os tempos medidos, data e hora da execução e os parâmetros utilizados, como arquivo de entrada, resolução e número de divisões.

O tempo de execução deste código seqüencial é utilizado como base para o cálculo dos *speedups* atingidos com os nossos algoritmos paralelos. O *speedup*, ou aceleração, é definido como a razão entre o tempo de execução da aplicação de visualização volumétrica paralela e o tempo de execução da versão seqüencial, monoprocessada, da mesma aplicação, para um mesmo conjunto de parâmetros de execução.

5.4.4 Implementação do Algoritmo *Parallel ZSweep* Original

A versão original do *Parallel ZSweep* foi escrita usando como base do algoritmo o modelo mestre-escravo de distribuição dinâmica de tarefas. Ela é usada como base de comparação dos nossos algoritmos distribuídos, conforme é explicado no Capítulo 6. Com a biblioteca *PZSweep*, a implementação deste algoritmo, também, pode ser realizada de forma bastante simples:

```
#include "pzs_pzsweep.hh"

void work_loop(Octree *octree, ViewPlane *view,
   char *tfname, float *brightness)
{
   PZSweep pzsweep(octree, view, tfname, brightness);
   int tile, source;

   if (rank == MASTER) {
      for (int i = 0; i < TILES; i++) {
         tile = i + NODES - 1;
         pzsweep.waitMessage(&source, PZSweep::msgWorkDone);
         pzsweep.sendMessage(&tile, 1, MPI_INT, source,
            PZSweep::msgWorkToDo);
         pzsweep.recvWorkDone(source);
      }
   }
   else {
      tile = rank - 1;
      while (tile < TILES) {
         pzsweep.renderTile(tile);
         pzsweep.sendWorkDone(MASTER, tile);
         pzsweep.recvMessage(&tile, 1, MPI_INT, MASTER,
            PZSweep::msgWorkToDo);
      }
   }
}
```

Neste caso, a biblioteca *PZSweep* também se encarrega de coletar os tempos de renderização de todos os nós, calcular os tempos máximo, mínimo e médio, e acrescentar estes dados ao arquivo de *log*, bem como o número de nós usados.

Cada processador escravo renderiza um *tile*, envia o resultado ao processador mestre e aguarda receber um novo *tile*, enquanto houver *tiles* disponíveis. O processador mestre aguarda uma mensagem de solicitação dos demais processadores, envia ao solicitante o próximo *tile* a ser renderizado e recebe o resultado da renderização do *tile* anterior, até que ele tenha coletado todos os *tiles* renderizados. Observa-se que no algoritmo do *Parallel ZSweep* o processador mestre não renderiza *tiles*, sendo usado apenas para gerenciar a distribuição dinâmica e a coleta.

5.4.5 Implementação da Distribuição Estática

A implementação da renderização paralela com distribuição estática de tarefas, também, é realizada de forma modular usando a biblioteca *PZSweep*. O código a seguir, por exemplo, se refere à implementação da distribuição contínua:

```
#include "pzs_pzsweep.hh"

void work_loop(Octree *octree, ViewPlane *view,
        char *tfname, float *brightness)
{
        PZSweep pzsweep(octree, view, tfname, brightness);

        pzsweep.distribute(PZSweep::continuous);

        while (pzsweep.hasWorkToDo())
                pzsweep.renderTile();

        if (rank == MASTER)
                for (int node = 1; node < NODES; node++)
                        pzsweep.recvScreen(MPI_ANY_SOURCE);
        else
                pzsweep.sendScreen(MASTER);
}
```

Conforme podemos observar, a distribuição contínua é usada para montar a fila de tarefas de cada nó. Então, cada *tile* é renderizado um a um até que a fila de tarefas esteja vazia. Finalmente, o *buffer* de tela de cada nó é coletado pelo processador coletor.

Nossos algoritmos – NN, LQ e CD – partem deste pequeno trecho de código fonte para construir os sistemas distribuídos de balanceamento de carga propostos.

CAPÍTULO 6

METODOLOGIA EXPERIMENTAL

Neste capítulo é apresentada a metodologia usada para avaliação dos algoritmos de balanceamento de carga propostos.

6.1 Ambiente

O ambiente de execução usado na obtenção dos resultados experimentais consiste de um *cluster* com 16 nós, com um processador *Intel Pentium III* de 800 MHz e 512 MB de memória em cada nó. Os nós encontravam-se conectados através de placas de rede *Fast Ethernet* de 100 Mbps. Todos os nós executavam o sistema operacional *Linux* com *kernel* versão 2.4.20. As comunicações entre nós foram estabelecidas usando a biblioteca de passagem de mensagens MPI versão 1.2.5 8.

O *cluste*r encontrava-se dedicado para execução exclusiva durante as monitorações de tempo. Cada experimento foi repetido 70 vezes e os resultados apresentados no capítulo 7 expressam a média dos tempos obtidos em cada execução.

6.2 Base de Comparação

Usamos o algoritmo *Parallel ZSweep*, que aqui chamamos de PZS, como base de comparação para nossos algoritmos. As primeiras implementações deste algoritmo executavam em máquinas paralelas de memória compartilhada SGI Onyx, SGI Origin 2000 e SGI Origin 3000 8. Embora o modelo de programação mestre-escravo do PZS, originalmente projetado para arquitetura de memória compartilhada, não seja o mais adequado para um sistema distribuído, com apenas 16 processadores a comunicação com o processador mestre não cria um gargalo na execução. Por outro lado, o esquema de distribuição dinâmica do PZS, baseado em uma fila única de tarefas, fornece uma excelente estratégia de balanceamento de carga, já que um escravo ocioso é imediatamente reabastecido com um *tile* pelo mestre, para sistemas com poucos processadores.

Assim, o PZS pode servir como base para comparação com nossos algoritmos propostos. Além disso, como nossos algoritmos são baseados na difusão distribuída de informação de carga, é interessante compará-los com um esquema que fornece informação centralizada de carga.

Com o aumento do número de processadores, a tendência é que o algoritmo PZS, embora apresente um excelente balanceamento de carga, crie um gargalo para a execução em função da fila única de tarefas estar centralizada no processador mestre. Os nossos algoritmos propostos não apresentam este ponto de estrangulamento, tornando-se potencialmente mais escaláveis para sistemas com muitos processadores.

6.3 Conjunto de Dados Volumétricos

Em nossos experimentos foram usados três diferentes conjuntos de dados volumétricos, denominados SPX, SPX1 e SPX2. Eles são versões baseadas em tetraedros de conjuntos de dados da NASA. SPX (cortesia de Peter Williams - LLNL) é um *grid* não estruturado composto de tetraedros. Os arquivos contendo os conjuntos de dados foram copiados para o disco rígido local de cada máquina. A Tabela 6.1 lista o número de vértices e células em cada conjunto de dados. A Figura 6.1 mostra uma imagem produzida a partir da renderização volumétrica do conjunto de dados SPX.

Quando não forem especificados outros parâmetros, os resultados experimentais apresentados no Capítulo 7 referem-se à renderização do conjunto de dados SPX2, usando uma divisão da tela em *tiles* de 32 por 32, para produzir uma imagem com resolução de 1024 x 1024 *pixels*.

Dados	**Vértices**	**Células**
SPX	2.9K	13K
SPX1	20K	103K
SPX2	150K	803K

Tabela 6.1: Informações sobre o número de vértices e o número de células nos conjuntos de dados SPX, SPX1 e SPX2.

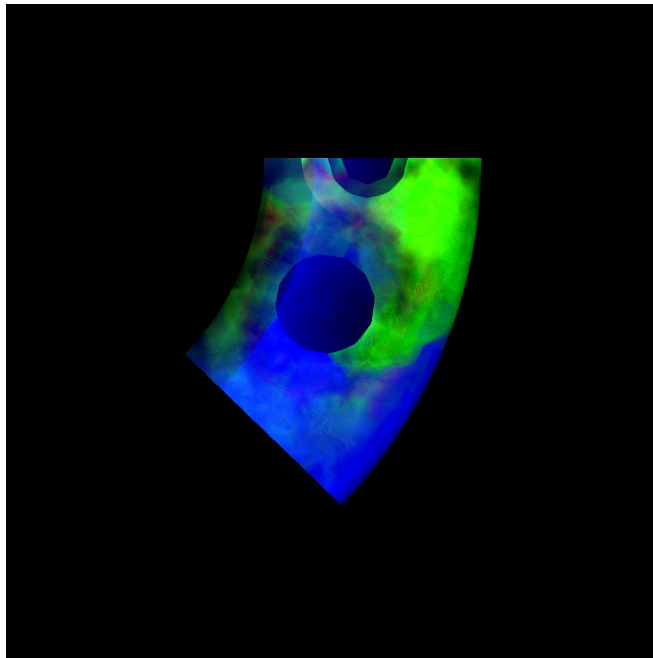


Figura 6.11: Visualização volumétrica do conjunto de dados SPX.

CAPÍTULO 7
RESULTADOS EXPERIMENTAIS

Neste capítulo avaliamos os resultados obtidos pela execução dos nossos algoritmos de balanceamento de carga. Apresentamos inicialmente uma comparação das estratégias de seleção de *tiles* e de distribuição estática de tarefas, mostrando que em qualquer das estratégias utilizadas, o balanceamento dinâmico de carga é absolutamente necessário para o desempenho do sistema. Comparamos, também, as estratégias de coleta final de *tiles* e, em seguida, utilizando as melhores estratégias de seleção de *tiles*, distribuição estática e coleta final, avaliamos o desempenho dos nossos algoritmos de balanceamento de carga em relação ao bom balanceamento implementado no *Parallel ZSweep*. Mostramos ainda como a granulosidade da divisão e a resolução da imagem afetam os nossos resultados.

7.1 Efeito da Seleção de *Tiles*

Inicialmente, em nosso primeiro experimento, avaliamos o desempenho dos dois diferentes métodos de seleção propostos: todos os *tiles* e *tiles* não vazios. Os métodos de seleção de *tiles* foram avaliados em combinação com a distribuição estática contínua e a coleta de tela inteira, sem o emprego de qualquer estratégia de balanceamento de carga.

Na Figura 7.1 apresentamos em (a) os tempos de execução obtidos e em (b) o grau de desbalanceamento de carga para cada um dos métodos de seleção. O desbalanceamento foi calculado pela fórmula proposta por Ma 8:

$$1 - \frac{t_{avg}}{t_{max}}$$

Nesta fórmula, t_{avg} representa a média dos tempos de renderização obtidos em cada processador, e t_{max} indica o tempo de renderização do último processador a terminar.

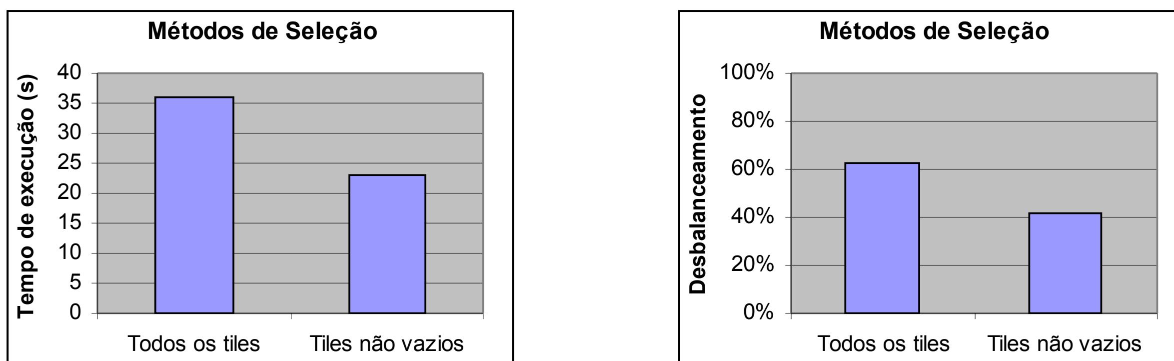


Figura 7.12: Comparação da seleção de todos os *tiles* com a seleção de *tiles* não vazios. (a) Tempos de execução. (b) Grau de desbalanceamento de carga.

Observamos que, conforme esperado, a seleção prévia de *tiles* não vazios contribuiu decisivamente para um menor desbalanceamento da carga de trabalho no sistema, reduzindo, em conseqüência, o tempo de execução. Removendo os *tiles* vazios, evitou-se que alguns processadores pudessem receber muitas regiões vazias da tela para renderizar enquanto outros recebiam regiões com alta densidade de objetos. O custo de pré-processamento, acrescentado pela seleção de *tiles* não vazios, é largamente compensado pela melhor distribuição de carga, tendo em vista que o custo de determinação de um *tile* vazio é muito reduzido, envolvendo apenas uma consulta à estrutura de dados *octree*, armazenada na memória.

7.2 Efeito da Distribuição Estática

No nosso segundo experimento, avaliamos o desempenho das três diferentes estratégias de distribuição estática propostas: contínua, intercalada e aleatória. As estratégias de distribuição estática foram avaliadas em combinação com a seleção de todos os *tiles* e a coleta de tela inteira, sem o emprego de qualquer estratégia de balanceamento de carga.

Na Figura 7.2(a) apresentamos o tempo de execução em segundos das três diferentes versões de distribuição estática de carga: contínua, intercalada e aleatória. Na Figura 7.2(b) apresentamos o percentual de desbalanceamento de carga obtido para cada uma das três distribuições estáticas, respectivamente.

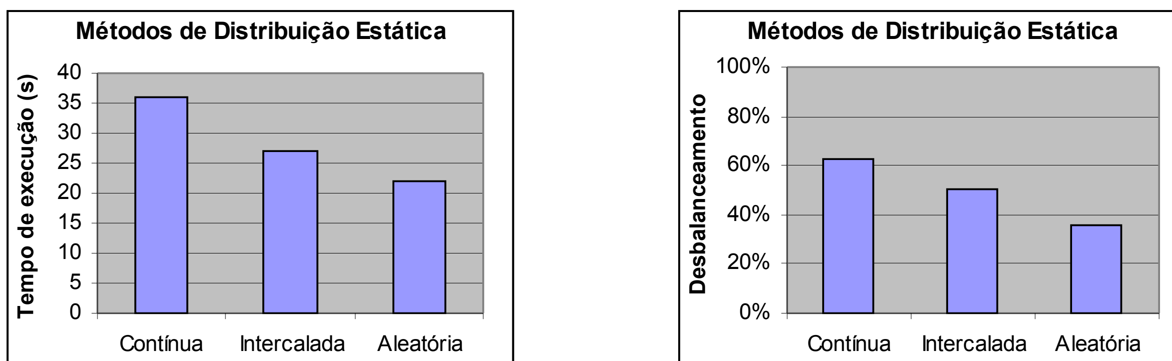


Figura 7.13: Comparação dos métodos de distribuição estática.
(a) Tempos de execução. (b) Grau de desbalanceamento.

Analisando a Figura 7.2 em detalhes, observamos que os tempos de execução, mostrados em (a), diminuem na medida em que é reduzido o grau de desbalanceamento de carga do sistema, mostrado em (b). Observamos ainda que a distribuição estática contínua proporciona o maior grau de desbalanceamento de carga e, conseqüentemente, o maior tempo de execução. Nesta distribuição, cada processador recebe uma região contínua da tela para renderizar. Em cenas reais, é comum que algumas regiões da tela contenham objetos próximos, enquanto outras regiões encontrem-se vazias. Nestes casos, a distribuição contínua fará com que alguns processadores recebam regiões com alta densidade de objetos, enquanto outros fiquem encarregados de renderizar apenas o fundo da imagem, produzindo elevado desbalanceamento de carga.

A distribuição intercalada consegue diminuir um pouco o desbalanceamento, alternando as regiões da tela atribuídas aos processadores. Entretanto, a distribuição estática aleatória consegue os menores tempos de execução, proporcionando uma distribuição inicial mais homogênea da carga de trabalho entre os processadores. Como a distribuição de tarefas é realizada de forma aleatória, minimiza-se a possibilidade de um processador receber apenas regiões densas da tela enquanto outro recebe apenas áreas vazias.

Em função dos resultados desses dois primeiros experimentos, foram escolhidos como estratégias de seleção de *tiles* e de distribuição estática de tarefas, a serem adotadas na avaliação dos nossos algoritmos de balanceamento de carga, a seleção de *tiles* não vazios e a distribuição aleatória.

Apesar dos ganhos de desempenho obtidos, os altos índices de desbalanceamento mostram que mesmo os métodos mais eficientes de seleção e distribuição não garantem balanceamento de carga satisfatório, sendo necessária a utilização de métodos dinâmicos de balanceamento de carga.

7.3 Efeito da Coleta Final

No nosso terceiro experimento, avaliamos como a coleta final influencia o tempo total de execução, comparando os três métodos de coleta propostos: coleta de tela inteira; coleta de *tiles* não vazios; e coleta simultânea. Os resultados obtidos referem-se à implementação das estratégias de coleta final em combinação com a seleção de todos os *tiles* e com a distribuição estática contínua de tarefas, ainda sem o uso de balanceamento dinâmico de carga.

Método de coleta final	Tempo de execução	Tempo de renderização			Grau de desbalanceamento
		Máximo	Médio	Mínimo	
Coleta de tela inteira	35,8s	35,4s	13,3s	0,0s	62,3 %
Coleta de *tiles* não vazios	35,4s	35,4s	13,3s	0,0s	62,3 %
Coleta simultânea	35,6s	35,6s	13,4s	0,0s	62,3 %

Tabela 7.2: Comparação dos tempos de execução, tempos de renderização e graus de desbalanceamento obtidos para cada um dos três métodos de coleta propostos.

Na Tabela 7.1 observamos os tempos de execução, os tempos de renderização e os graus de desbalanceamento obtidos para cada um dos três métodos de coleta propostos. Como esperado, os graus de desbalanceamento obtidos são rigorosamente iguais, visto que a fase de coleta final de sub-imagens não afeta a distribuição de carga. Os tempos de execução obtidos encontram-se bem próximos, indicando que a fase de coleta não representa uma fração muito significativa do tempo total de execução. Apesar das estratégias de coleta serem bem distintas, é muito pequena a diferença resultante nos tempos de execução.

Observamos que a coleta de tela inteira apresenta um tempo total de execução ligeiramente superior à coleta de *tiles* não vazios, devido ao maior tráfego de dados na rede e ao processamento de uma maior quantidade de *pixels* durante a combinação da imagem final, já que, na primeira estratégia de coleta, as telas inteiras de cada processador são enviadas, coletadas e combinadas.

A coleta simultânea, também, apresenta um tempo de execução ligeiramente superior à coleta de *tiles* não vazios, proporcionalmente à pequena diferença apresentada nos tempos médios de renderização. O tempo médio de renderização da coleta simultânea é ligeiramente superior devido à competição entre a *thread* principal, que executa a renderização, e a *thread* secundária, que executa o balanceamento.

Desta forma, apesar de termos obtidos tempos de execução quase idênticos, a coleta de *tiles* não vazios foi escolhida para avaliar os algoritmos de balanceamento de carga propostos, em função de ter reduzido ligeiramente a quantidade de dados enviados, recebidos e processados durante a coleta.

7.4 Análise dos Algoritmos de Balanceamento

No nosso quarto experimento, avaliamos os algoritmos de balanceamento propostos. O algoritmo do Vizinho mais Próximo (NN) e o algoritmo da Fila mais Longa (LQ) foram implementados usando a seleção de *tiles* não vazios combinada com a distribuição estática aleatória de tarefas. Estas estratégias de seleção e distribuição inicial de tarefas se mostraram as mais eficientes, através dos resultados apresentados. Já o algoritmo da Distribuição Circular (CD) introduz um outro paradigma de distribuição de tarefas: a distribuição dinâmica baseada em *token-ring*. Os três algoritmos de balanceamento de carga foram implementados usando a coleta de *tiles* não vazios como estratégia de coleta final de sub-imagens.

7.4.1 Algoritmos Distribuídos x Algoritmo Centralizado

Primeiramente, comparamos a execução dos nossos algoritmos – Vizinho mais Próximo (NN), Fila mais Longa (LQ) e Distribuição Circular (CD) – com a execução do algoritmo original *Parallel ZSweep* (PZS). Nesse experimento, estamos interessados em investigar o quanto as nossas propostas de mecanismos totalmente distribuídos de balanceamento de carga se aproximam do excelente balanceamento obtido pelo modelo de fila única de tarefas empregado pelo *Parallel ZSweep*. Embora o modelo de fila única de tarefas do *Parallel ZSweep* tenha sido desenvolvido para uma arquitetura com memória compartilhada, com até 16 processadores ele apresenta desempenho bastante satisfatório, não gerando gargalo de execução. Portanto, o objetivo a ser alcançado é que os nossos algoritmos distribuídos gerem um balanceamento de carga tão bom quanto aquele gerado pela fila única de tarefas.

A Tabela 7.2 mostra os *speedups* obtidos para cada um dos algoritmos, com 4, 8 e 16 processadores renderizando os conjuntos de dados SPX, SPX1 e SPX2. Os gráficos das Figuras 7.3, 7.4 e 7.5 ilustram os dados da Tabela 7.2, mostrando os *speedups* obtidos com os conjuntos de dados SPX, SPX1 e SPX2, respectivamente. Os *speedups* são relativos ao tempo seqüencial, medido usando apenas uma máquina do *cluster*.

Observamos que os algoritmos distribuídos NN, LQ e CD conseguem obter *speedups* semelhantes aos obtidos com o algoritmo PZS, proposto originalmente para arquitetura de memória compartilhada. Os *speedups* se aproximam especialmente para conjuntos de dados maiores como o SPX2. Visualmente, podemos observar que as linhas dos gráficos convergem na medida em passamos da Figura 7.3 para a Figura 7.4, até chegarmos à Figura 7.5, que mostra os resultados da renderização do conjunto de dados SPX2, onde os algoritmos distribuídos de balanceamento de carga NN, LQ e CD apresentam desempenho quase igual ao algoritmo centralizado PZS, mesmo quando é usada a quantidade máxima de processadores disponíveis, ou seja, 16 nós. Na medida que a quantidade de dados cresce, aumentando a quantidade de trabalho a ser realizada, vemos que o balanceamento de carga tem efeito decisivo no desempenho dos sistemas distribuídos, fazendo-os apresentar um grau de desbalanceamento tão baixo quanto aquele apresentado pelo modelo de fila única de tarefas.

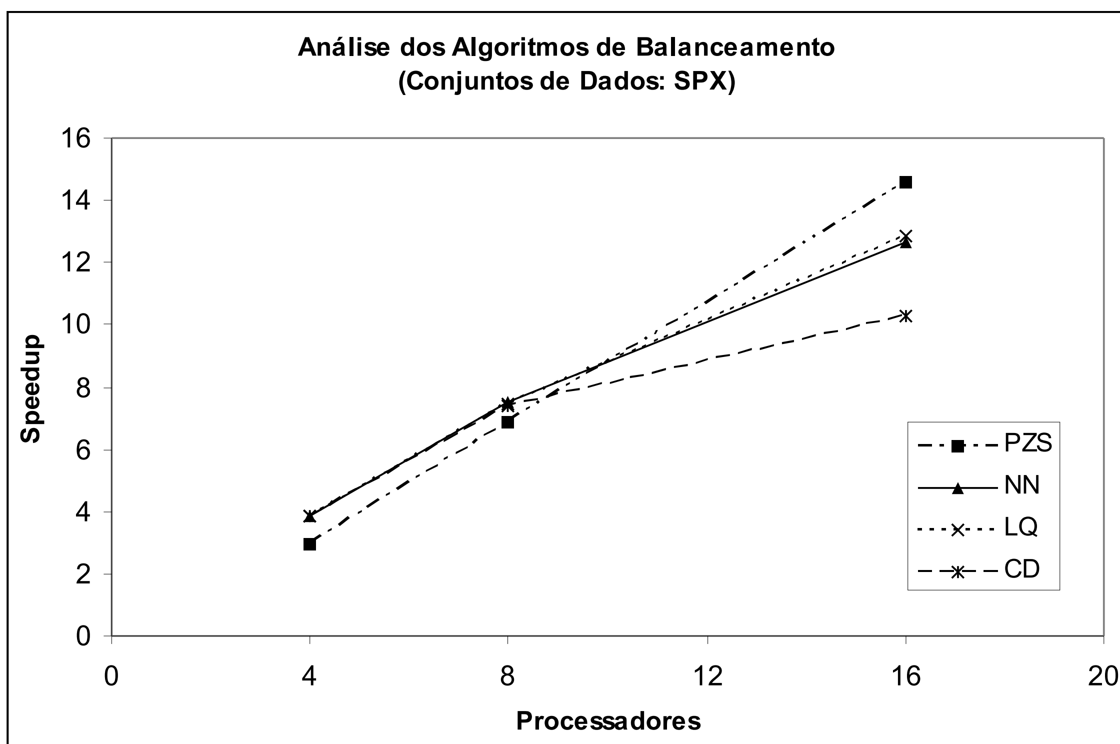


Figura 7.14: *Speedups* obtidos para diferentes algoritmos com o conjunto de dados SPX.

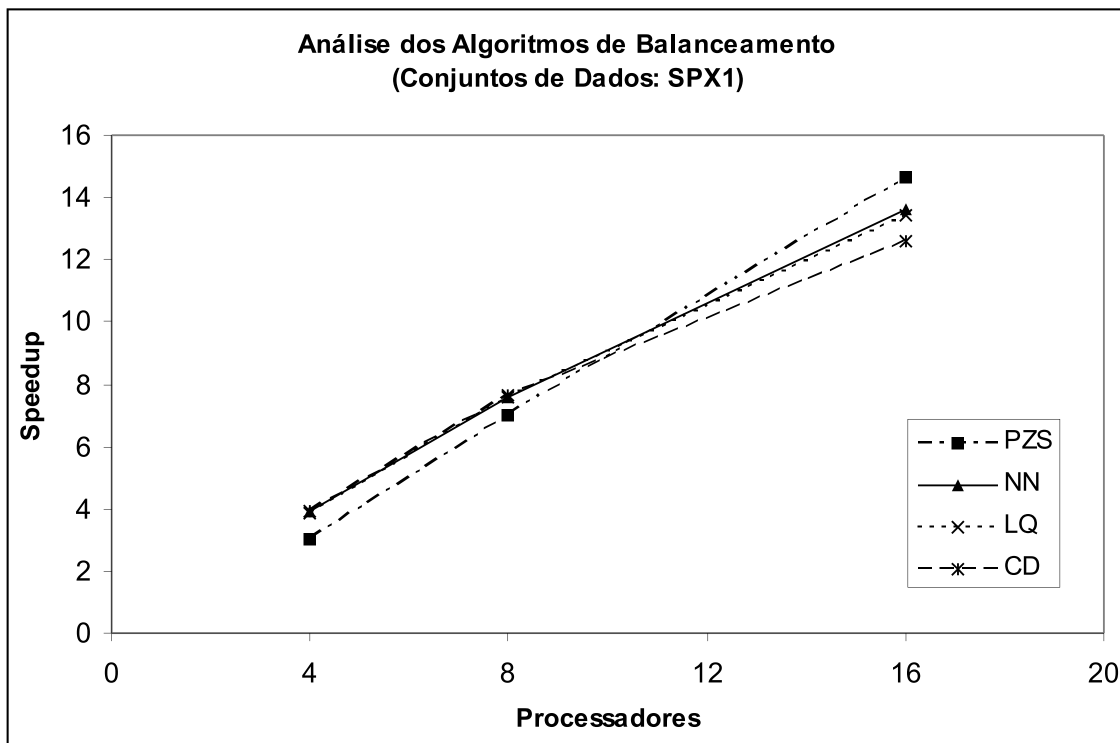


Figura 7.15: *Speedups* obtidos para diferentes algoritmos com o conjunto de dados SPX1.

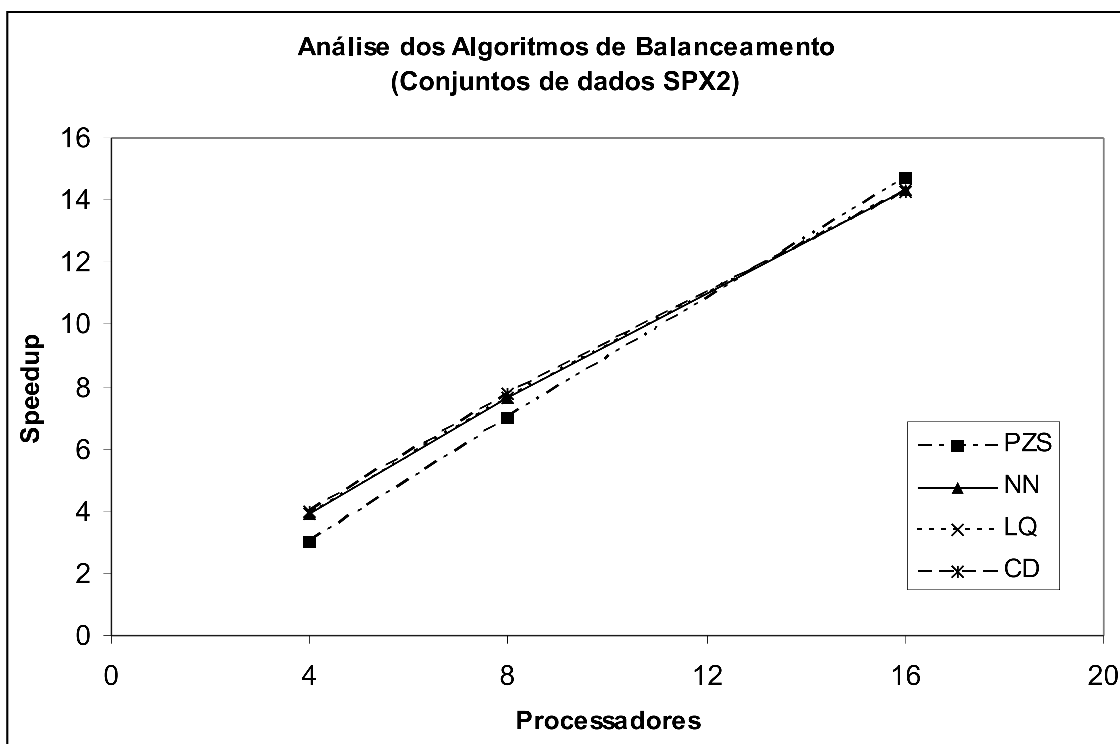


Figura 7.16: *Speedups* obtidos para diferentes algoritmos com o conjunto de dados SPX2.

Algoritmo	*Speedup*								
	SPX			**SPX1**			**SPX2**		
	4 Nós	**8 Nós**	**16 Nós**	**4 Nós**	**8 Nós**	**16 Nós**	**4 Nós**	**8 Nós**	**16 Nós**
PZS	2,95	6,85	14,56	2,99	6,98	14,66	3,00	7,00	14,70
NN	3,88	7,49	12,68	3,89	7,56	13,61	3,91	7,63	14,36
LQ	3,86	7,43	12,82	3,88	7,58	13,41	3,90	7,66	14,31
CD	3,86	7,36	10,29	3,92	7,66	12,62	3,96	7,80	14,26

Tabela 7.3: *Speedups* obtidos para diferentes algoritmos de balanceamento de carga.

Conclui-se que os algoritmos distribuídos propostos apresentam desempenho satisfatório para arquiteturas de memória distribuída processando grandes volumes de dados, obtendo eficiência de paralelismo de até 90%. Além dos resultados dos algoritmos distribuídos NN, LQ e CD se aproximarem do algoritmo centralizado PZS, eles possuem ainda a vantagem de uma maior escalabilidade da arquitetura distribuída com relação à arquitetura centralizada.

Os algoritmos distribuídos NN, LQ e CD foram desenvolvidos para sistemas com muitos processadores, e seus resultados podem ser comparados aos do algoritmo centralizado PZS a partir de 16 processadores. Os resultados obtidos com 4 e 8 processadores mostram-se desfavoráveis ao PZS pelo fato de que, neste algoritmo, o processador mestre é usado apenas para coordenar o processamento, resultando em um processador a menos trabalhando na renderização. Este efeito passa a ter menos importância na medida em que aumenta o número de processadores, como mostra a Figura 7.6. Este gráfico mostra o desempenho obtido pelo algoritmo original PZS em comparação com uma versão deste mesmo algoritmo modificada para usar o processador mestre na renderização. Nesta versão modificada são implementadas duas *threads* no processador mestre: uma executa a renderização enquanto a outra executa a função de distribuidora de tarefas.

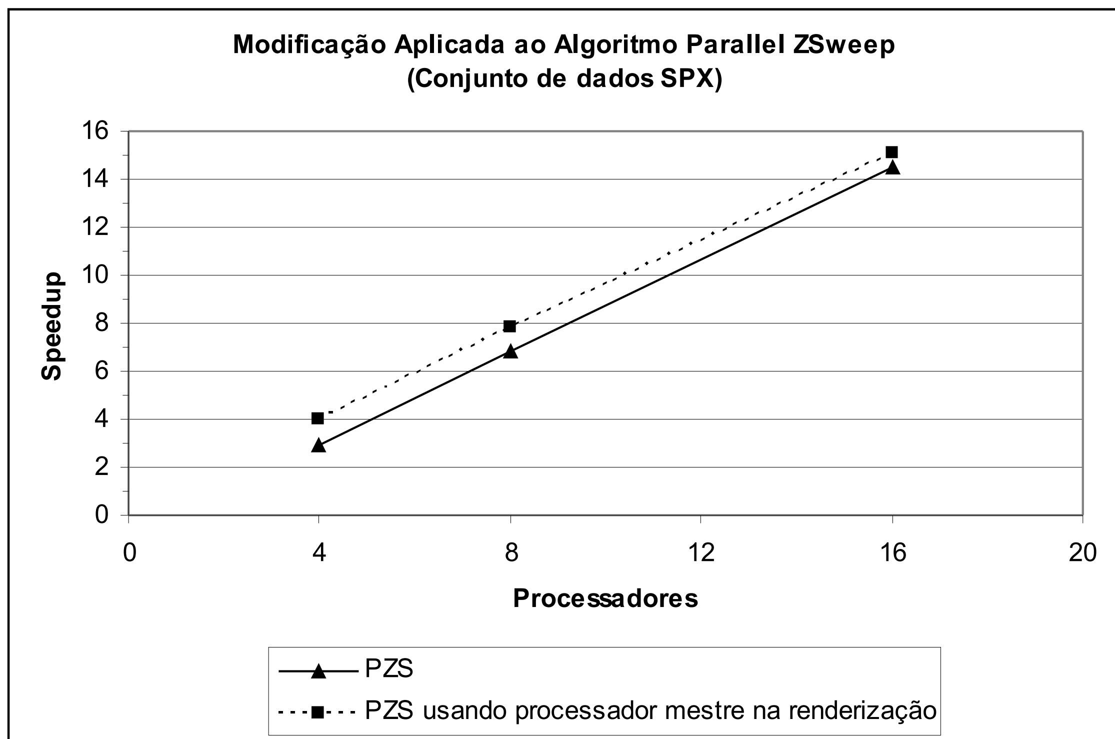


Figura 7.17: Modificação aplicada ao algoritmo *Parallel ZSweep*.

Observamos na Figura 7.6 que, para 4 processadores, o uso do processador mestre na renderização resulta em um aumento de uma unidade de *speedup* em relação ao algoritmo PZS original, o que corresponde exatamente a um processador a mais. Na medida em que o número de processadores aumenta, esta diferença cai, chegando a 0,6 unidade de *speedup* com 16 processadores. A tendência exibida pelo gráfico é que, com mais processadores, as linhas se cruzem, porque o processador mestre não conseguirá desempenhar tão bem a função de distribuidor de tarefas enquanto está ocupado renderizando.

7.4.2 Algoritmos de Balanceamento

Comparando entre si os algoritmos distribuídos de balanceamento de carga, observamos, pelos dados apresentados, que os algoritmos NN e LQ, baseados no mecanismo de *work stealing*, apresentam desempenhos muito próximos, independente do conjunto de dados e do número de processadores utilizados. Embora os mecanismos de balanceamento destes algoritmos sejam muito diferentes, as estratégias de seleção de *tiles* não vazios e de distribuição estática aleatória provêem uma distribuição inicial de tarefas razoavelmente equilibrada, facilitando o trabalho de redistribuição de carga executado pelos algoritmos.

O algoritmo CD, embora tenha um desempenho semelhante aos algoritmos NN e LQ processando grandes conjuntos de dados, apresenta resultados menos expressivos para pequenos conjuntos de dados. Para conjuntos de dados muito pequenos, a velocidade de circulação do *token-ring*, usado na distribuição dinâmica de tarefas, acaba sendo inferior à velocidade de renderização de *tiles*. Para grandes conjuntos de dados, que são o objetivo principal dos algoritmos de balanceamento de carga propostos, a situação se inverte, e o *token-ring* consegue circular em velocidade suficiente para garantir uma distribuição de tarefas bem equilibrada enquanto os processadores executam a renderização.

Concluímos que os três algoritmos distribuídos conseguem resultados experimentais semelhantes e satisfatórios, para a renderização de grandes conjuntos de dados em *clusters* de PCs com 16 nós, e possivelmente mais, apesar de adotarem estratégias bem distintas de balanceamento de carga.

7.4.3 Efeito da Variação da Granulosidade e Precisão da Imagem

No nosso último experimento, analisamos como os parâmetros de execução afetam os tempos de execução dos algoritmos. Os parâmetros analisados são: a granulosidade das divisões da tela e a precisão da imagem. Foram realizadas monitorações de tempo com divisões de 8x8, 16x16, 32x32 e 64x64 conjuntos de *tiles*, além de medições de tempo para resoluções de imagem de 512x512, 1024x1024 e 2048x2048 *pixels*.

7.4.3.1 Variação da Granulosidade

A Tabela 7.3 mostra os tempos de execução, em segundos, obtidos para os algoritmos NN, LQ e CD para diferentes granulosidades (8x8, 16x16, 32x32 e 64x64) e diferentes números de processadores (4, 8 e 16 nós).

	Tempos de execução (segundos)								
	4 processadores			8 processadores			16 processadores		
Divisões	**NN**	**LQ**	**CD**	**NN**	**LQ**	**CD**	**NN**	**LQ**	**CD**
8x8	40,6	41,0	44,6	25,6	25,5	35,2	22,0	21,7	21,9
16x16	41,3	41,0	40,9	22,3	22,5	22,4	13,2	13,3	12,9
32x32	56,1	56,3	55,5	28,8	28,7	28,2	15,3	15,4	15,4
64x64	106,3	106,4	105,9	53,3	53,3	53,1	27,6	27,0	28,9

Tabela 7.4: Tempos de execução obtidos para os algoritmos NN, LQ e CD com diferentes granulosidades.

A análise da variação da granulosidade mostra que há um valor ideal para o número de divisões em cada sistema analisado. Um número muito pequeno de divisões resulta em cargas de trabalho muito desbalanceadas entre os processadores, devido aos *tiles* de grande tamanho. Um número muito alto de divisões resulta em muitas células sendo processadas mais de uma vez, por encontrarem-se nas fronteiras entre os *tiles*, que terão tamanhos reduzidos. Observamos que, para os parâmetros usados, o valor ideal do número de divisões é: entre 8x8 e 16x16 para 4 processadores; 16x16 para 8 processadores; e entre 16x16 e 32x32 para 16 processadores. Há uma tendência de aumento da granulosidade ideal com o aumento do número de processadores.

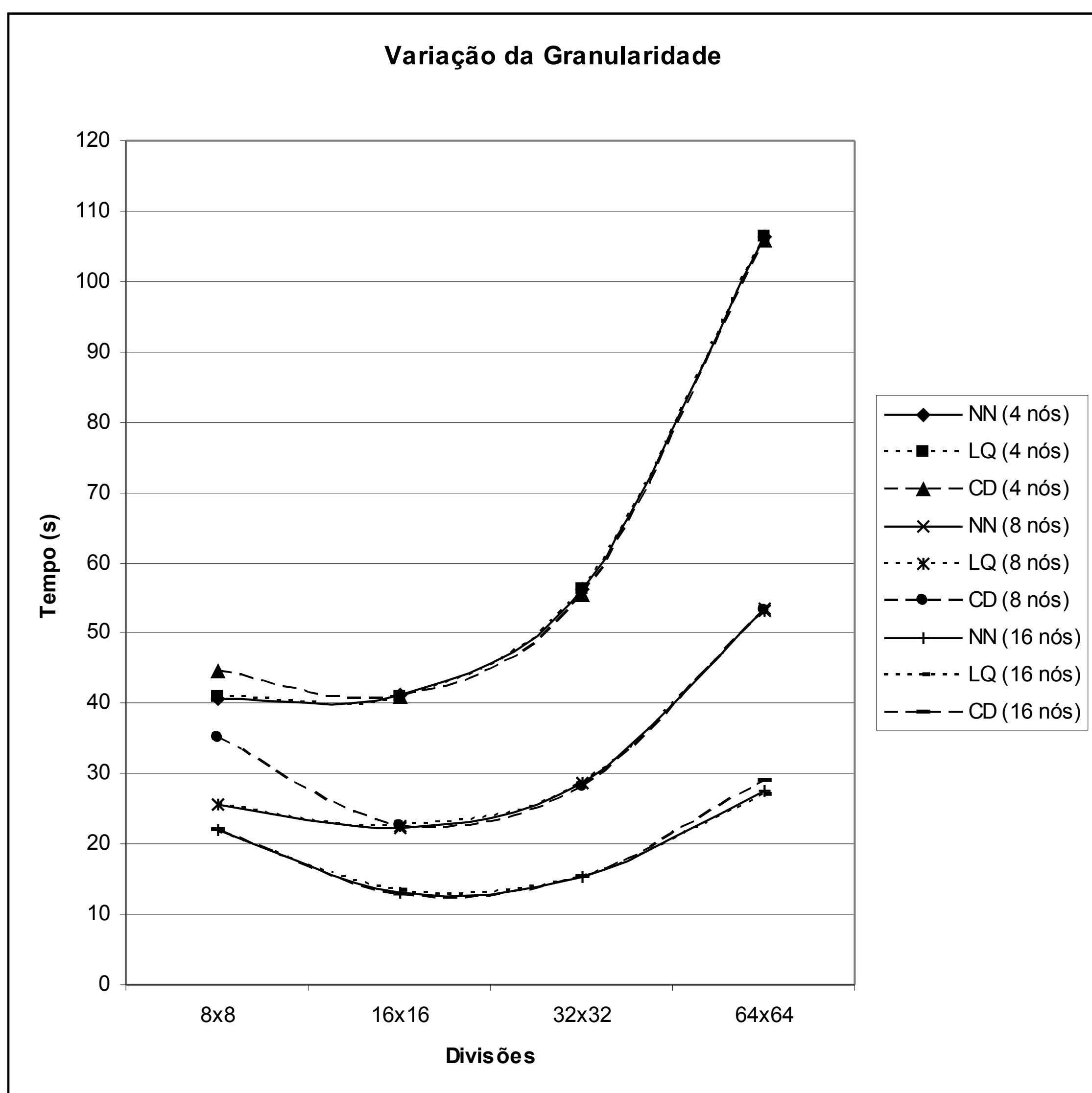


Figura 7.18: Efeito da variação da granulosidade sobre os tempos de execução.

O gráfico apresentado na Figura 7.7 ilustra os dados da Tabela 7.3, apresentando para cada algoritmo a variação do tempo de execução em função da variação da granulosidade. Observamos que as curvas agrupam-se de acordo com a quantidade de processadores: um grupo de curvas posicionadas abaixo, com menores tempos de execução, para o maior número de processadores, 16 nós; um grupo de curvas ao centro, para 8 nós; e um grupo de curvas posicionadas acima, com maiores tempos de execução, para o menor número de processadores, 4 nós. As posições relativas dos grupos de curvas expressam o resultado natural de que quanto maior o número de processadores, maior o grau de paralelismo e menores os tempos totais de execução.

Observamos ainda que, para cada número de processadores, as curvas obtidas para os diferentes algoritmos praticamente se sobrepõem, confirmando que os algoritmos propostos, embora distintos, apresentam resultados semelhantes mesmo com a variação da granulosidade da divisão da tela.

7.4.3.2 Variação da Precisão da Imagem

A análise da variação da precisão da imagem mostra como o tempo de execução aumenta na medida em que a resolução é ampliada de 512x512 para 1024x1024 e, a seguir, para 2048x2048. Note que o custo de renderização está diretamente relacionado à precisão da imagem.

A Tabela 7.4 mostra os tempos de execução obtidos para os diferentes algoritmos propostos, usando resolução de imagem de 512x512, 1024x1024 e 2048x2048 *pixels*. O gráfico apresentado na Figura 7.8 ilustra os dados da Tabela 7.4, mostrando, conforme esperávamos, que o tempo de execução aumenta à medida que aumenta a precisão da imagem, para todos os algoritmos.

Os resultados obtidos mostram que os algoritmos NN, LQ e CD apresentam comportamentos semelhantes frente à variação dos parâmetros de execução. Os algoritmos apresentam tempos de execução bem próximos, independente dos parâmetros de execução usados.

	Tempos de execução (segundos)								
	4 processadores			8 processadores			16 processadores		
Resolução	**NN**	**LQ**	**CD**	**NN**	**LQ**	**CD**	**NN**	**LQ**	**CD**
512x512	37,6	37,6	37,1	19,4	19,2	18,8	10,2	10,5	10,5
1024x1024	56,1	56,3	55,5	28,9	28,7	28,2	15,3	15,4	15,4
2048x2048	132,8	133,0	131,5	67,5	67,5	66,5	35,0	35,6	35,5

Tabela 7.5: Tempos de execução obtidos para os algoritmos NN, LQ e CD com diferentes resoluções.

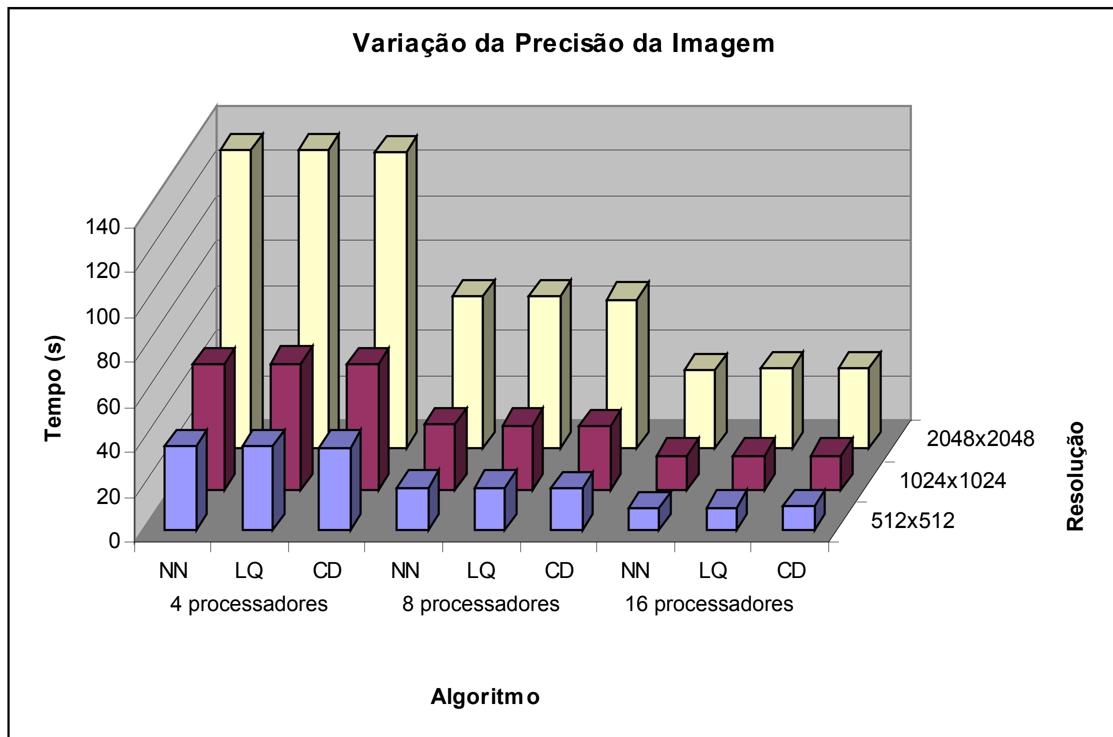


Figura 7.19: Efeito da variação da precisão da imagem sobre os tempos de execução.

CAPÍTULO 8
CONCLUSÕES

Neste trabalho abordamos a implementação da renderização volumétrica paralela de grandes volumes de dados num *cluster* de PCs, de modo a se obter um sistema eficiente e de baixo custo para visualização científica. Nosso objetivo foi estudar o problema de balanceamento de carga gerado pela renderização paralela. Propusemos três algoritmos distribuídos de balanceamento de carga: Vizinho mais Próximo, Fila mais Longa e Distribuição Circular. Os algoritmos empregam técnicas conhecidas de difusão de informação em sistemas distribuídos e executam o balanceamento de carga de diferentes modos.

O algoritmo do Vizinho mais Próximo implementa o balanceamento executando transferências de carga de trabalho entre processadores vizinhos. O algoritmo da Fila mais Longa promove transferências de carga entre processadores com grande carga de trabalho e processadores ociosos, usando um mecanismo de *token-ring* para distribuir informações sobre a carga de trabalho de cada processador. O algoritmo da Distribuição Circular adota uma estratégia de distribuição dinâmica da carga de trabalho, combinando os conceitos de distribuição de tarefas e balanceamento de carga, simulando a circulação de uma fila de tarefas na rede.

Implementamos os três algoritmos distribuídos usando como base o algoritmo *PZSweep*, que é um algoritmo centralizado de balanceamento de carga projetado para arquitetura de memória compartilhada. O *PZSweep* foi modificado para trabalhar de forma distribuída, usando uma estratégia de distribuição inicial estática aleatória de tarefas, combinada com um processo preliminar de seleção de *tiles* não vazios. Nossos resultados mostraram que esta versão distribuída do *PZSweep* apresentava um alto grau de desbalanceamento de carga.

Os resultados experimentais mostraram que a aplicação de nossos algoritmos de balanceamento de carga sobre a versão distribuída do *PZSweep* resultou em ganhos de até 90% de eficiência paralela, com apenas 10% de desbalanceamento de carga, num *cluster* de 16 processadores executando a renderização do maior conjunto de dados disponível.

Comparando nossos algoritmos distribuídos com o algoritmo centralizado *PZSweep*, que fornece um excelente balanceamento de carga para sistemas com poucos processadores, concluímos que nossos algoritmos apresentam um desempenho quase igual ao do algoritmo *PZSweep*. Os resultados mostraram que os algoritmos distribuídos apresentaram desempenho de um algoritmo centralizado, com a vantagem de serem mais escaláveis para utilização em problemas e arquiteturas maiores.

Comparando os nossos três algoritmos entre si, concluímos que eles apresentam desempenho semelhante com os vários parâmetros de execução empregados, reduzindo satisfatoriamente o desbalanceamento de carga do sistema, embora utilizem técnicas completamente diferentes de balanceamento de carga. Os algoritmos são genéricos e podem ser adaptados para outros sistemas de visualização paralela, fornecendo uma alternativa de baixo custo com a execução em *clusters* de PC's.

Os trabalhos futuros incluem: um estudo mais detalhado da escalabilidade dos algoritmos usando maiores conjuntos de dados e *clusters* com mais processadores; a análise da influência da velocidade de circulação *token-ring* no desempenho do sistema; e a implementação dos nossos algoritmos em outros sistemas de visualização paralela.

BIBLIOGRAFIA

[1] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 81-88, November 1993.

[2] C. Hofsetz and K. L. Ma. Multi-threaded rendering unstructured-grid volume data on the sgi origin 2000. In *Third Eurographics Workshop on Parallel Graphics and Visualization*, 2000.

[3] L. Hong and A. Kaufman. Accelerated ray-casting for curvilinear volumes. In *IEEE Visualization '98*, pages 247-254, October 1998.

[4] K. L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *IEEE Parallel Rendering Symposium*, pages 23-30, October 1995.

[5] K. L. Ma and T. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *IEEE Parallel Rendering Symposium*, pages 95-104, November 1997.

[6] S. Muraki, E. Lum, K Ma, M. Ogata and X. Liu. A PC Cluster System for Simultaneous Interactive Volumetric Modeling and Visualization. In *2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2003.

[7] R. Samanta, J. Zheng, T. Funkhouser, K. Li and J.P. Singh. Load balancing for multi-projector rendering systems In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 107-116, August 1999.

[8] M. Meißner and T. Hüttner and W. Blochinger and A.Weber. Parallel Direct Volume Rendering on PC Networks. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, July 1998.

[9] R. Samanta, T. Funkhouser, K. Li and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[10] P. Bunyk, A. Kaufman and C. Silva. Simple, fast and robust ray casting of irregular grids. In *Scientific Visualization, Proceedings of Dagstuhl '97*, pages 30-36, 2000.

[11] R. Farias, J. Mitchell and C. Silva. ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. In *2000 Volume Visualization Symposium*, pages 91-99, October 2000.

[12] J. Clark. A VLSI Geometry Processor for Graphics. In Computer, Vol. 13, No. 7, pages 59-68, July 1980.

[13] J. Clark. The Geometry Engine: A VLSI Geometry System for Graphics. In *Computer Graphics*, Vol. 16, No. 3, pages 127-133, July 1982.

[14] S. Whitman. Dynamic Load Balancing for Parallel Polygon Rendering. In *IEEE Computer Graphics and Applications*, Vol.14, No. 4, pages 41-48, July 1994.

[15] R. Farias and C. Silva. Out of Core Rendering of Large Unstructured Grids. In *Special Issue of Computer Graphics and Applications - Large-Scale Data Visualization*, July/August 2001.

[16] H. Fuchs and J. Poulton. Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine. In *VLSI Design*, pages 20-28, 1981.

[17] M. Potmesil and E. M. Hoffert. The Pixel Machine: A Parallel Image Computer. In *Computer Graphics*, Vol. 23, No. 3, pages 69-78, July 1989.

[18] T. Naruse, M. Yoshida, T. Takahashi, and S. Naito. SIGHT - A Dedicated Computer Graphics Machine. In *Computer Graphics Forum*, Vol. 6, No. 4, pages 327-334, December 1987.

[19] A. Kaufman and R. Bakalash. Memory and Processing Architecture for 3D Voxel-Based Imagery. In *IEEE Computer Graphics and Applications*, Vol. 8, No. 6, pages 10-23, November 1988.

[20] G. Knittel and W. Straßer. A Compact Volume Rendering Accelerator. In *Proceedings 1994 Symposium on Volume Visualization*, ACM SIGGRAPH, pages 67-74, October 1994.

[21] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24, October 1992.

[22] R. Farias and C. Silva. Parallelizing the ZSWEEP Algorithm for Distributed-Shared Memory Architectures. In *International Workshop on Volume Graphics*, pages 91-99. October 2001.

[23] R. Farias, C. Bentes, A. Coelho, S. Guedes and L. Gonçalves. Work Distribution for Parallel Zsweep Algorithm. In *XI Brazilian Symposium on Computer Graphics and Image Processing*, pages 107-114, October 2003.

[24] A. Coelho, C. Bentes and R. Farias. Distributed Load Balancing Algorithms for Parallel Volume Rendering on Cluster of PCs. To be published in *IASTED International Conference on Computers Graphics and Imaging (CGIM 2004)*, August 2004.

[25] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker and J. Dongarra. MPI - The Complete Reference. *The MIT Press, Cambridge, Massachusetts, London, England*, 1996.