ROBERTO DE BEAUCLAIR SEIXAS

VISUALIZAÇÃO VOLUMÉTRICA COM RAY-CASTING NUM AMBIENTE DISTRIBUÍDO.

TESE DE DOUTORADO

DEPARTAMENTO DE INFORMÁTICA

Rio de Janeiro, 9 de abril de 1997.

Roberto de Beauclair Seixas

Visualização Volumétrica com Ray-Casting num Ambiente Distribuído

Tese apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para a obtenção do título de Doutor em Ciências em Informática - Ênfase: Computação Gráfica.

Orientador: Marcelo Gattass

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 9 de Abril de 1997.

À minha esposa Maria Lúcia

Agradecimentos

À minha esposa Maria Lúcia pelo seu amor, carinho e apoio em todos os momentos, principalmente os difíceis.

À minha família pela educação e apoio que recebi e em memória do meu pai Arthur, minha irmã Vera Maria e meu padrasto Roberto, que infelizmente não puderam compartilhar deste meu momento de alegria.

Ao meu orientador, Marcelo Gattass, pelos seus conselhos, por acreditar e incentivar este trabalho e por me permitir partilhar alguns de seus conhecimentos.

Ao Luiz Martins, pela sua amizade, liderança e pela motivação e apoio ao ingresso no programa de doutorado.

Ao Luiz Fernando, pelos conselhos e incansáveis incentivos no decorrer do desenvolvimento da Tese.

Ao Luiz Henrique, por ser um exemplo a se seguir, pela sua amizade e pelas indispensáveis sugestões.

Ao LNCC, principalmente ao Olinto e Gadelha, pelo suporte e oportunidade proporcionados.

A todos do TeCGraf (Grupo de Tecnologia em Computação Gráfica da PUC-Rio), pela compreensão e paciência durante a fase de testes.

Resumo

Ray-Casting é uma técnica muito usada em visualização volumétrica para a criação de imagens médicas, a partir de dados obtidos por ressonância magnética (MRI) e tomografias computadorizadas (CT). No entanto, *ray-casting* tem um alto custo computacional que resulta em um processo de visualização lento, o que compromete a interatividade necessária para uma boa compreensão do conjunto de dados tri-dimensionais.

Este trabalho propõe estratégias para a otimização do algoritmo de *ray-casting* e para melhorar sua eficiência. Além disso, esta tese investiga o uso em um ambiente de computação distribuída, através de um protocolo de comunicação entre estações de trabalho heterogêneas e não dedicadas, conectadas em uma rede local.

As idéias propostas foram implementadas em duas versões do algoritmo, uma seqüencial e uma paralela. Os resultados obtidos com essas implementações em conjuntos de dados reais mostram que é possível obter tempo interativo com as máquinas disponíveis atualmente e em condições normais de uso da rede local por outros usuários.

Abstract

Ray-Casting is a useful volume visualization technique applied to medical images such as computer tomography (CT) and magnetic resonance image (MRI). It has, however, a high computational cost that results in a slow rendering process, which compromises the interactivity that is necessary for a good comprehension of the three-dimensional data set.

This work proposes optimization strategies to the *ray-casting* algorithm to improve its efficiency. To enhance, even further, the thesis investigates the use of a distributed computer environment through a communication protocol between heterogeneous and non-dedicate LAN-connected workstations.

The ideas proposed here were implemented in two versions of the algorithm, one sequential and one parallel. Test results, obtained with these implementations and real data sets, show that it is possible to obtain interactive time with the current available machines.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Diversas áreas do conhecimento utilizam processos que produzem dados volumétricos que representam objetos tridimensionais. Por exemplo, medicina (imagens de tomografias – CT e ressonância magnética – MRI), biologia (microscopia eletrônica), geociências (sísmica), meteorologia (tornados e furacões) e engenharia (elementos finitos). Esses dados são normalmente fornecidos em grades tri-dimensionais, onde cada nó contém um ou mais valores escalares e/ou vetoriais. A compreensão da estrutura dos volumes implicitamente representados por esses dados é um dos desafios importantes enfrentados, principalmente, pelas áreas de medicina, geologia e engenharia.

A Visualização Volumétrica, como uma das técnicas de Visualização Científica, procura transformar dados volumétricos em imagens que permitam aos pesquisadores entenderem o comportamento e a natureza do que está sendo estudado. Entretanto, a compreensão de estruturas tri-dimensionais complexas através de imagens em superfícies de visualização bi-dimensionais, como as telas de monitores, não é uma tarefa simples. Para auxiliar o entendimento, é importante que o sistema que gera a imagem possa prover ao seu usuário uma animação de movimento. Somente através da simulação de um vôo em volta e por dentro do objeto, o usuário pode compreender estruturas complexas como, por exemplo, as carótidas num exame de MRI, que possuem uma estrutura complicada com várias ramificações.

No entanto, os algoritmos para visualização volumétrica possuem um custo computacional muito alto, que torna inviável visualizações altamente interativas na grande maioria dos computadores atuais. Tipicamente, a complexidade deste algoritmo é $O(n^3+s^2n)$, onde n é a quantidade de valores em cada dimensão do volume e s é o número de *pixels* em cada dimensão da imagem, conforme mostrou Challinger [3] para o algoritmo de *ray-casting*.

Nos últimos anos, vários autores [36, 45, 47, 22, 19, 52, 20, 37, 38] apresentaram otimizações e estratégias alternativas com o objetivo de aumentar a velocidade dos algoritmos. Mesmo com as técnicas de otimizações já existentes, os algoritmos atuais, como por exemplo, o apresentado por Lacroute e Levoy [20], por exemplo, ainda não conseguem a visualização volumétrica em tempo real (*real-time*). O tempo mínimo que se conseguiu é chamado de "tempo para viabilizar interação" (*interactive-time*), indicando que os computadores atuais ainda não são rápidos o bastante para gerar imagens de alta qualidade com

estas técnicas [42].

Mais recentemente, algumas implementações utilizando computadores paralelos de alto desempenho tem sido apresentadas [32, 35]. No entanto, devido ao alto custo dos computadores paralelos e a disponibilidade de computadores de uso geral conectados em redes locais, o uso de computação distribuída tem sido explorado. Esta abordagem apresenta uma melhor relação custo/benefício do que com computadores massivamente paralelos [14, 44].

Esta tese propõe estratégias de otimizações para o algoritmo de *ray-casting* que aumentam a sua eficiência. A utilização de um ambiente de computação distribuída, através de computadores heterogêneos, não-dedicados, conectados em rede local, é também apresentada como uma contribuição na melhoria do desempenho do algoritmo.

No Capítulo 2 é feita uma revisão dos principais trabalhos na área de Visualização Volumétrica, suas formas de classificação e seus algoritmos principais.

No Capítulo 3 são descritos o algoritmo de *ray-casting*, trabalhos correlatos e as contribuições deste trabalho para o algoritmo, tanto na sua forma seqüencial quanto na paralela. Os resultados obtidos com uma implementação seqüencial destas propostas são também apresentados, seguidos de uma comparação com os resultados obtidos através do uso de uma biblioteca de domínio público para visualização volumétrica, VolPack [21], que já foram publicados em Seixas *et al* [37] e Seixas *et al* [38]. Alguns resultados obtidos com a implementação paralela também já foram publicados em Seixas e Gattass [39].

O Capítulo 4 apresenta inicialmente os conceitos e as implicações da utilização de computação paralela em ambientes distribuídos. Este capítulo também apresenta quatro estratégias de particionamento do algoritmo de *ray-casting* e uma proposta de avaliação de eficiência em ambientes distribuídos, heterogêneos e não dedicados.

No Capítulo 5 é apresentada uma implementação das idéias do Capítulo 4, juntamente com a apresentação de quatro conjuntos de dados de testes e dos resultados obtidos na visualização desses dados com essa implementação. Este capítulo apresenta ainda uma discussão dos resultados obtidos.

O Capítulo 6 apresenta as conclusões e propostas de trabalhos futuros.

Capítulo 2

Visualização Volumétrica

A Visualização Volumétrica é um conjunto de técnicas de extração de informação a partir de dados volumétricos, usando imagens e gráficos interativos, que dependem da representação, manipulação e análise destes conjuntos de dados.

Os dados volumétricos são entidades tri-dimensionais que podem conter informações no seu interior, representadas ou não por superfícies. As técnicas de Visualização Volumétrica permitem a visualização de informações extraídas do interior dos dados volumétricos, através de cortes, segmentação, classificação e transparência.

Ainda existem vários desafios no campo de Visualização Volumétrica. O tamanho típico de um conjunto de dados volumétricos é de dezenas de *megabytes*, algumas vezes até de centenas de *megabytes*, e ainda gostaríamos de poder combinar dois ou mais conjuntos de dados para a análise, numa única imagem, de algum aspecto de interesse comum.

As ferramentas para a classificação dos conjuntos de dados ainda não são eficientes e nem todas as técnicas de visualização volumétrica trabalham bem com todos os tipos de dados. Os conjuntos de dados que contêm características amorfas como nuvens, fluidos e gases, ainda são particularmente difíceis de se visualizar.

Os dados volumétricos se originam, principalmente, de dados amostrados de objetos reais ou fenômenos naturais, resultados numéricos derivados de experimentos empíricos, resultados oriundos de simulações e de dados gerados por um modelo geométrico.

Os dados amostrados de objetos reais são oriundos, por exemplo, de imagens médicas, tais como ressonância magnética (MRI), tomografica computadorizada (CT) e emissão de fótons (SPECT); de biologia (microscópios eletrônicos) e de geologia (sísmica). Como áreas que usam dados resultantes de simulações podemos citar meteorologia, para previsão de tempestades e tornados, e dinâmica de fluidos, para visualização da solução das equações de Navier-Stokes, o que permite representar com precisão o fluxo em torno de um corpo 3D (por exemplo, a pressão do vento sobre as asas de um avião) [34].

Geralmente, os dados volumétricos são amostrados ou medidos, regularmente espaçados nos três eixos ortogonais, definidos por uma **grade regular**. Se o volume ao redor dos pontos da grade tem o mesmo valor que o ponto da grade, temos uma área hexaédrica de valor constante denominada *voxel*. Se considerarmos o volume como uma coleção de hexaedros cujos vértices são os pontos da grade e cujo valor varia entre eles, utilizamos

funções de interpolação para estimar os valores dentro destas células.

Além da grade regular, são utilizadas a **grade retilínea**, onde os intervalos ao longo dos eixos podem variar, a **grade curvilínea**, onde as arestas da grade podem não ser retas, e **grades não estruturadas**, em que nenhuma geometria é definida, com células descritas por tetraedros, hexaedros, prismas, etc, utilizada, por exemplo, na análise por elementos finitos, dinâmica dos fluidos e em interpolação de dados esparsos.

2.1 Classificação

Na visualização volumétrica está implícita a criação de alguma representação intermediária, de objetos visíveis ou fenômenos que pode ser visualizada para produzir uma imagem. Esta representação intermediária pode ser feita de diversas formas: pontos, linhas, superfícies, cubos, nuvens, etc. Levoy [26] mostra que as técnicas dominantes para a exibição de dados volumétricos podem ser classificadas de acordo com a representação intermediária que empregam: *surface-based techniques*, empregando polígonos ou *patches* de superfícies curvas; *binary voxel techniques*, baseadas em cubos opacos ou nos polígonos que os definem; e *semi-transparent volume techniques*, qualquer modelo com aparência de um "gel" semi-transparente.

As técnicas de visualização volumétrica também podem ser classificadas de acordo com a ordem na qual os dados são processados. Sutherland *et al.* [26] classificaram os algoritmos de acordo com a forma como operam: *object-order*, percorrendo a lista de objetos e determinando para cada objeto quais *pixels* eram afetados; *image-order*, varrendo o espaço da tela e determinando quais objetos afetam cada *pixel*.

Elvins [8] classifica os métodos de visualização volumétrica em duas categorias (Figura 2.1): visualização volumétrica direta (*direct volume rendering* - DVR) e iso-superfícies (*surface fitting* - SF). A primeira é caracterizada pelo mapeamento dos *voxels* diretamente no espaço de visualização, sem o uso de primitivas geométricas como representação intermediária. Esta categoria é especialmente apropriada para ser utilizada com imagens médicas [26], gases e fluidos [11]. Os algoritmos principais são *ray-casting* [24, 25], *splatting* [22] e *V-buffer* [45]. A segunda, também chamada de *iso-surfacing*, utiliza primitivas geométricas tais como triângulos ou polígonos mais gerais, que coincidam com uma faixa de valores (*threshold*), definindo o contorno de uma superfície no volume. Estas técnicas são apropriadas para a visualização de campos escalares em três dimensões, tais como velocidade, temperatura e pressão, tendo como principais algoritmos *marching-cubes* e *dividing-cubes* [4, 7].

Nelson [30] sugere uma classificação muito parecida com a de Elvins [8], mas com uma categoria a mais: *multiplanar slice projection*. Esta outra categoria refere-se à extração de imagens planas de orientação arbitrária de uma localização particular em um conjunto de dados 3D. Esta é a abordagem computacional mais direta para inspeção de dados através de um volume. Esses métodos são usados por médicos e clínicos para a avaliação de dados de pacientes, pois a visualização interativa dos *slices* fornece ao clínico uma idéia da anatomia.

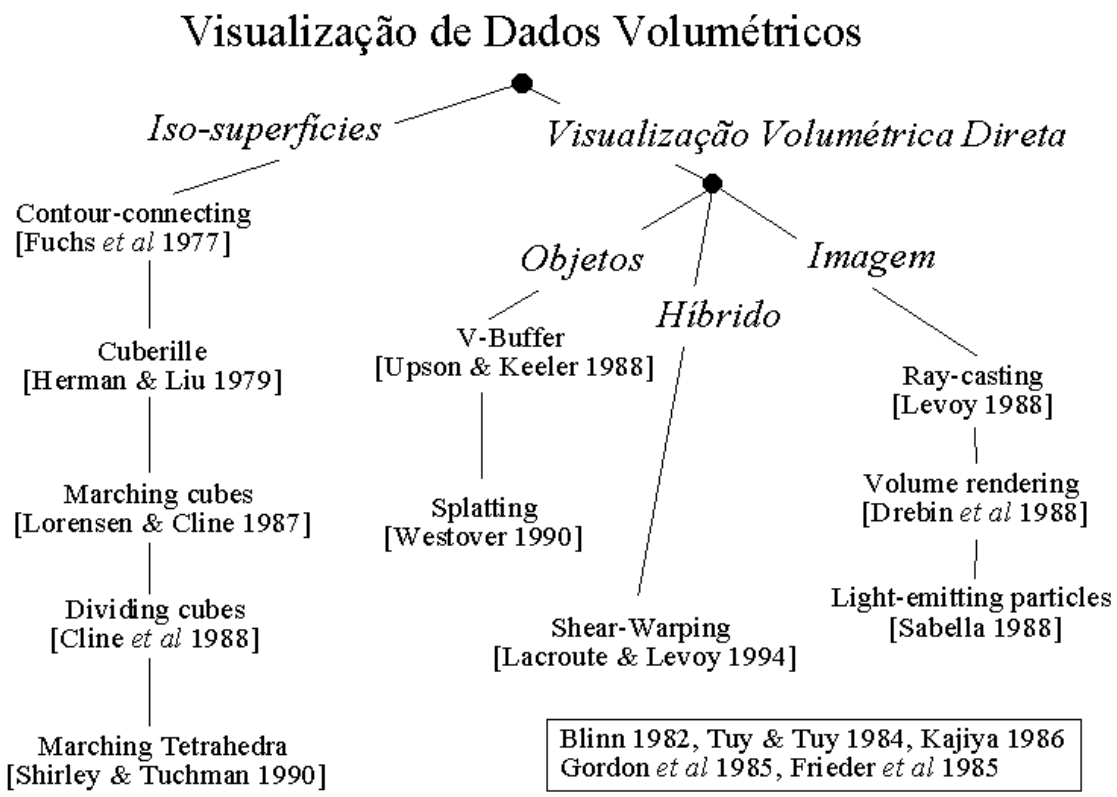

Figura 2.1: Classificação das Técnicas de Visualização Volumétrica.

2.2 Técnicas de Visualização Volumétrica

2.2.1 Iso-Superfícies

A geração de iso-superfícies é a forma mais antiga e mais bem conhecida de se estudar o conteúdo dos conjuntos de dados volumétricos. No final dos anos 70, Fuchs *et al.* [10] desenvolveu um algoritmo para gerar representações geométricas tri-dimensionais através da conexão das iso-linhas de planos adjacentes (*contour connecting*). Posteriormente, Herman [16] utilizou cubos opacos ou semi-opacos (*opaque cubes*) para representar o contorno de iso-superfícies.

Mais recentemente, Lorensen e Cline [27] apresentaram um algoritmo denominado *marching-cubes*, que gera uma iso-superfície examinando os oito vértices do *voxel* e determinando as superfícies de interseção. As interseções ao longo das arestas do *voxel* são calculadas usando-se interpolação linear ou tri-linear, e uma malha triangular para a iso-superfície é então gerada. O algoritmo de *dividing-cubes* funciona da mesma forma, mas a iso-superfície é gerada por pontos ao invés de triângulos, através de sub-divisões sucessivas [4].

Para evitar o problema de conexão das superfícies do algoritmo de *marching-cubes*,

Shirley e Tuckman [40] criaram um algoritmo denominado *marching-tetrahedra*, que divide a célula em cinco, seis ou vinte e quatro tetraedros, com suas respectivas tabelas de interseção. Assim, apesar de gerar mais superfícies e precisar de mais memória, o algoritmo é utilizado para se reduzir as ambiguidades na conexão das mesmas, evitando a geração de superfícies falsas ou incorretas.

Uma desvantagem do uso de iso-superfícies está no fato de visualizarmos somente um subconjunto dos dados, determinado pelo valor de *threshold*. Isto pode ser contornado com o uso de iso-superfícies transparentes, uma para cada valor de *threshold*. No entanto, o entendimento diminui à medida que incluimos mais iso-superfícies.

2.2.2 Visualização Volumétrica Direta

O algoritmo de *ray-casting* baseia-se em raios lançados do ponto de vista do observador que passam através de cada *pixel* da tela e interceptam o volume. Se o raio interceptar o volume, o conteúdo do volume ao longo do raio é amostrado, transformando-se o valor escalar em cor e opacidade, resultando num valor que é atribuido ao *pixel*. Existem duas abordagens para a amostragem ao longo do raio. A primeira envolve a localização do ponto de entrada do raio no volume e, através de divisões equidistantes, a amostragem é feita até se atingir o ponto de saída do raio no volume. Dependendo do tamanho da divisão, é possível que não seja incluída a contribuição de alguns *voxels*. A segunda abordagem envolve a determinação dos pontos de entrada e saída para cada um dos *voxels* interceptados pelo raio. A contribuição para cada *voxel* é então calculada por interpolação [24, 36]. Neste algoritmo, o *loop* principal é feito através dos *pixels* da tela, onde cada raio pode ser calculado independentemente de qualquer outro e em qualquer ordem, permitindo a sua paralelização.

Os algoritmos *V-buffer* [45] e *Splatting* [47], apesar de terem o caminhamento pelos objetos, também projetam suas células ou nós no espaço da tela. Estes métodos necessitam que as células ou nós sejam ordenados do mais distante para o mais próximo (*back-to-front*) ou vice-versa (*front-to-back*).

Os algoritmos que usam *slice-shearing*, como o de *Shear-Warping* [20], primeiro rotacionam o volume na memória até que ele esteja alinhado com o plano de visualização. Depois, a visualização é feita atravessando-se o volume.

Sabella [36] modificou o algoritmo de *ray-casting* considerando cada *voxel* como um emissor de densidade variável. Através deste método, quatro valores são calculados para cada raio lançado através do volume: o maior valor encontrado ao longo do raio, a distância para este valor máximo, a intensidade da atenuação e o centro de gravidade. A imagem é gerada usando-se o modelo de cor HSV (*Hue, Saturation, Value*) da seguinte forma: o maior valor para *hue*, a intensidade da atenuação para *value* e a distância ou o centro de gravidade como *saturation*.

Na abordagem usada por Levoy [24] são utilizados os próprios valores nos *voxels* para determinar cor e opacidade. O algoritmo de *ray-casting* é utilizado para acumular o vetor de cor e de opacidades correspondente ao caminho percorrido pelo raio do ponto do vista do observador, passando através da tela até o volume.

Westover [47] propôs uma técnica, denominada *splatting*, que realiza um caminhamento ordenado pelo volume. As contribuições dos *voxels* são calculadas e compostas usando-se tabelas de mapeamento (*look-up tables*). O primeiro passo do algoritmo é determinar em qual ordem o volume deve ser percorrido, ou seja, qual o *voxel* mais próximo do plano de visualização. No segundo passo, os *voxels* são "atirados" de acordo com as suas distâncias e, a seguir, é calculada a projeção dos *voxels* no plano de visualização. Um filtro de arredondamento (*reconstruction kernel*) é usado para determinar a extensão da contribuição. A projeção do filtro no plano de visualização, denominada *footprint*, é proporcional ao tamanho do volume e ao tamanho da imagem a ser gerada (Figura 2.2).

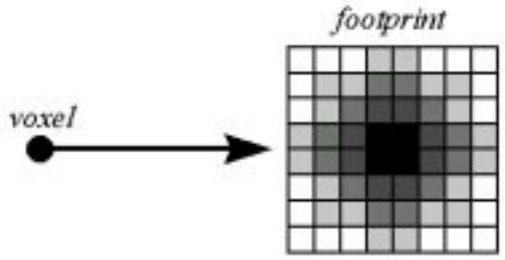

Figura 2.2: Algoritmo de Westover.

Uma otimização para o algoritmo de Westover, chamada *hierarquical splatting*, foi proposta por Laur e Hanrahan [22] e utiliza um caminhamento hierárquico nos dados e um refinamento progressivo da imagem. Um algoritmo semelhante, denominado *V-buffer*, foi apresentado por Upson e Keeler [45] e tem como principal característica o fato de basear-se em células ao invés de *voxels*. O algoritmo de *V-buffer* percorre o interior das células, interpolando os valores dos vértices e projetando cada valor interpolado no plano de visualização.

Nesta tese, optou-se pelo uso de visualização volumétrica direta através do algoritmo de *ray-casting*. A visualização volumétrica direta foi escolhida pois elimina a necessidade de classificação binária dos dados, como pertencentes ou não de uma determinada superfície. Com isso, pode-se visualizar estruturas muito pequenas e pouco definidas. O algorimo de *ray-casting* foi escolhido por permitir a visualização de imagens de alta qualidade e de possibilitar a visualização de estruturas internas através da uso de transparência nas estruturas mais externas. Por exemplo, as Figuras 2.3 e 2.4 mostram os ossos de um pé humano atribuindo-se a pele uma transparência de 90%, sem no entanto perder a noção de seu contorno.

Figura 2.3: Imagem em vista inferior.

Figura 2.4: Imagem em vista lateral.

Capítulo 3

Acelerações no Algoritmo de Ray-Casting para Visualização Volumétrica

O algoritmo de *ray-casting* para Visualização Volumétrica foi originalmente proposto independentemente por Levoy [24] e Drebin, Carpenter e Hanrahan [7] como uma técnica que permite a visualização de pequenos detalhes internos ao volume através do controle de transparência dos *voxels*, removendo trivialmente as partes escondidas atrás de partes definidas como opacas, e visualizando o volume a partir de qualquer direção. O algoritmo efetua um lançamento de raios a partir do observador, através de cada *pixel* do plano de visualização, em direção ao volume (Figura 3.1). Quando a projeção é paralela ortográfica, a direção destes raios é a da normal a este plano. Em pseudo-código, o algoritmo de *ray-casting* é:

```
para cada pixel da tela dispare um raio em direção ao volume
     para cada voxel interceptado pelo raio
          acumule a contribuição do voxel para o pixel
```

A cor final de cada *pixel* da imagem é obtida integrando as contribuições de cor $C(x)$ e transparência $\alpha(x)$ de cada *voxel* x interceptado pelo raio. Na Figura 3.2, a cor do *pixel* correspondente ao raio antes do cálculo da contribuição do *voxel* em questão é C_{in}. Computada a contribuição de um *voxel* x, a cor passa a ser C_{out}.

O cálculo da cor e da transparência de cada *voxel* usa estimativas do valor da densidade e da norma do gradiente $\nabla f(x)$ da iso-superfície correspondente a este valor de densidade. Assim, a cor é determinada através de um modelo de iluminação de superfícies onde a cor do material é obtida a partir do valor da densidade e da normal $N(x)$ corresponde ao unitário do vetor gradiente unitário.

$$N(i) = \frac{\nabla f(i)}{\mid \nabla f(i) \mid}.$$

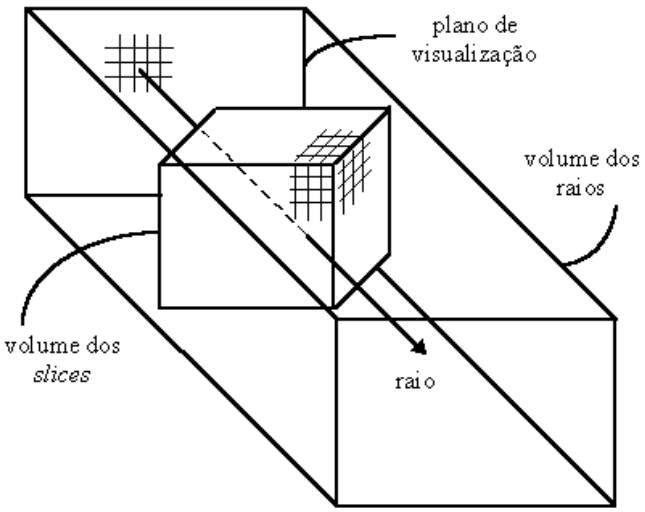

Figura 3.1: Lançamento de raios.

A estimativa da norma do gradiente é geralmente feita antes do lançamento dos raios por operadores locais do tipo diferenças finitas [24], como por exemplo, diferenças centrais:

$$\begin{aligned}\nabla f(x) = \nabla f(i,j,k) \approx \\ (&\frac{1}{2}(f(i+1,j,k) - f(i-1,j,k)), \\ &\frac{1}{2}(f(i,j+1,k) - f(i,j-1,k)), \\ &\frac{1}{2}(f(i,j,k+1) - f(i,j,k-1))).\end{aligned}$$

Levoy [24] implementou este algoritmo através de dois *pipelines* independentes: um para iluminação e um para classificação do material, concluindo com uma fase final de composição dos dois *pipelines* (Figura 3.3). Note-se que esta avaliação se dá para cada passo em todos os raios do algoritmo.

Na etapa de iluminação, as componentes RGB da cor $C(x)$ para cada *voxel* x são calculadas a partir de uma estimativa do gradiente da função densidade $D(x)$ e da intensidade de luz, usando, por exemplo, o algoritmo de Phong [9]. Na etapa de classificação, uma transparência $\alpha(x)$, baseada na densidade dos materiais $D(x)$, é associada a cada *voxel*. A composição final é realçada enfatizando-se as bordas das regiões de densidade quase uniforme e desenfatizando-se seus interiores, multiplicando-se o valor da opacidade pelo módulo do gradiente.

$$\alpha'(x) = \alpha(x) \mid \nabla D(x) \mid .$$

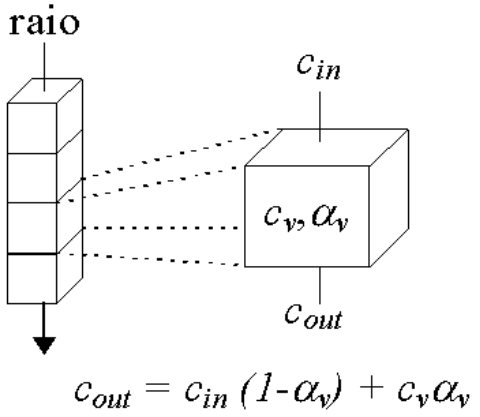

Figura 3.2: Cálculo da contribuição de cada *voxel*.

Desta forma, o volume é tratado como um "gel" iluminado com iso-superfícies determinadas para cada *voxel* [26]. A criação da imagem final é feita pela projeção bi-dimensional de $C(x)$ e $\alpha(x)$ no plano de visualização.

O algoritmo assim descrito consome muita memória e tem um alto custo computacional. Note que temos que armazenar, além do volume de dados (densidades), a opacidade e a norma do gradiente para cada *voxel*. Por exemplo, para um volume de $256 \times 256 \times 256$ (16 M*bytes*), o espaço de armazenamento final seria de 208 M*bytes* (12 *bytes* para o gradiente e 1 *byte* para a opacidade). Para contornar este problema, os cálculos são feitos em tempo de execução e, para que o algoritmo não fique muito lento, são utilizadas tabelas de mapeamento de cor e de opacidade (*look-up tables*).

3.1 Trabalhos Correlatos

O alto custo computacional do algoritmo de *ray-casting* e a necessidade de se reduzir o consumo de memória motivavam a busca de otimizações. As primeiras otimizações foram propostas pelo próprio Levoy [25], substituindo a enumeração exaustiva do volume por uma enumeração espacial hierárquica, onde *voxels* de valores semelhantes são agrupados em células, e utilizando a terminação no caminhamento do raio de uma forma adaptativa, interrompendo o cálculo das contribuições quando estas não forem mais significativas.

Lacroute e Levoy [20] apresentaram um algoritmo eficiente que combina as vantagens de outros algoritmos, utilizado a coerência no volume e na imagem, obtendo um acesso sincronizado às estruturas de dados, mantendo sempre o alinhamento entre o caminhamento no volume e no plano de visualização.

Zuffo e Lopes [52] apresentaram uma otimização significativa no *pipeline* do algoritmo original de Levoy, antecipando a reamostragem e eliminando um volume intermediário com as componentes RGB e opacidade. Além disso, as imagens finais têm uma melhor qualidade

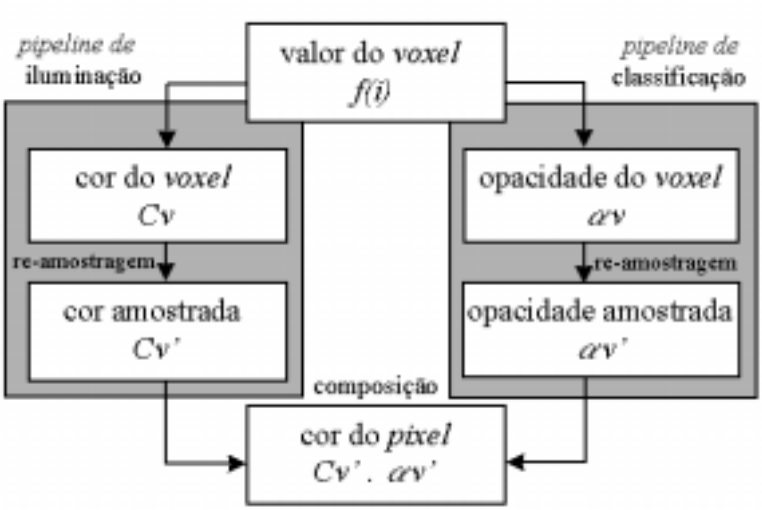

Figura 3.3: Fases do algoritmo de Levoy.

devido a uma melhor aproximação do gradiente, a partir dos dados reamostrados e não a partir dos dados originais como propôs Levoy.

3.2 Contribuições ao Algoritmo Básico

O algoritmo descrito a seguir inclui otimizações nos seguintes passos básicos do algoritmo de *ray-casting* original:

- lançamento dos raios;
- determinação dos segmentos dos raios que interceptam o volume de dados;
- acumulação da contribuição dos *voxels* ao longo do raio;
- exibição da imagem.

3.2.1 Lançamento dos Raios

A primeira otimização faz o lançamento dos raios através da determinação de pontos de referência no espaço do objeto. O caminhamento na imagem é efetuado através de incrementos previamente calculados neste plano de referência. Esta técnica é mais simples do que a proposta por Yagel e Kaufman [49], pois utiliza dois planos auxiliares de visualização (Figura 3.4).

Apesar da referência ser o *pixel* (espaço da imagem), é mais eficiente determinar pontos de referência no espaço do objeto e utilizar de incrementos, previamente calculados, para o caminhamento sobre a cena.

Este procedimento é feito através da utilização de um plano de visualização frontal e um traseiro, paralelos ao plano da tela e descritos no sistema de coordenadas do objeto. Os raios são lançados de um plano para o outro em função dos seus pontos extremos.

Os incrementos são então calculados em função do tamanho do volume de raios, em *pixels*, segundo a direção de lançamento do raio inicial, nos três eixos. A partir do raio inicial, os incrementos para os demais raios são calculados de forma proporcional.

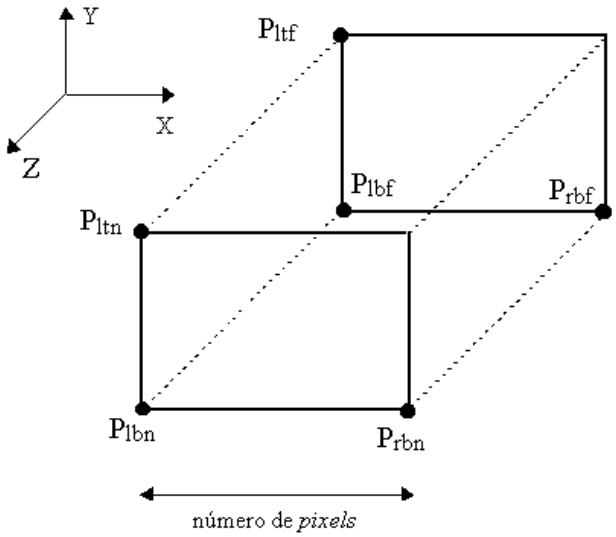

Figura 3.4: Planos Auxiliares.

3.2.2 Interseção com o Volume de Dados

Para permitir uma manipulação flexível pelo usuário, o volume de raios deve ser rotacionado, expandido e contraído. Desta forma, torna-se fundamental descartar eficientemente as partes dos raios que não atingem o volume de dados.

Para determinar a interseção dos raios com o volume de dados, é utilizado um algoritmo de *clipping* 3D. Como a maioria dos raios intercepta o volume de dados, optou-se pela utilização do algoritmo de Cyrus-Beck [9], pois este é mais eficiente na determinação dos pontos de interseção.

O algoritmo de Cyrus-Beck baseia-se na determinação de um parâmetro t da interseção do raio com a superfície do volume de dados, e na classificação deste ponto como "potencialmente entrando" (PE) ou "potencialmente saindo" (PS), de acordo com a superfície de interseção. O valor t é dado por:

$$t = \frac{-N_i \cdot [P_0 - P_i]}{N_i \cdot [P_1 - P_0]},$$

face	ponto de interseção
left	$t = \frac{X_0 - X_{min}}{-\Delta X}$
right	$t = \frac{X_{max} - X_0}{\Delta X}$
bottom	$t = \frac{Y_0 - Y_{min}}{-\Delta Y}$
top	$t = \frac{Y_{max} - Y_0}{\Delta Y}$
near	$t = \frac{Z_0 - Z_{min}}{-\Delta Z}$
far	$t = \frac{Z_{max} - Z_0}{\Delta Z}$

Tabela 3.1: Tabela do cálculo de interseções.

onde N_i é a normal da face, P_0 é o ponto inicial do raio, P_1 o ponto final do raio e P_i é um ponto abitrário a ser classificado como PE ou PS. Desta forma, se $den > 0$, então tem-se o caso PS e, se $den < 0$, tem-se o caso PE.

Os pontos de interseção com o volume ficam assim determinados por uma tabela de cálculo de interseções (Tabela 3.1).

Na tabela 3.1, X_{min}, X_{max}, Y_{min}, Y_{max}, Z_{min} e Z_{max}, representam as dimensões do volume dos dados. Os pontos de partida dos raios no plano de visualização são (X_0, Y_0, Z_0) e ΔX, ΔY e ΔZ, são os tamanhos nos respectivos eixos.

3.2.3 Caminhamento no Raio

Uma vez descobertos os pontos de interseção do raio com o volume de dados, o algoritmo de *ray-casting* prossegue percorrendo o raio e acumulando a contribuição de cada *voxel* pelo qual o raio passa. A forma usual é percorrer o raio em pontos de amostragem equidistantes. No entanto, este procedimento possui como problema a determinação do *voxel* correspondente a cada amostragem.

Para evitar tal problema, inicialmente foi utilizado o algoritmo de Bresenham [9], originalmente desenvolvido para desenhar linhas em um dispositivo matricial, permitindo uma forma incremental de caminhamento do raio [37]. Desta forma, evita-se o problema de localização e garante-se que, para um mesmo raio, não existam duas amostragens para um mesmo *voxel*.

Entretanto, o algoritmo de Bresenham não garante a contribuição de todos os *voxels* interceptados por um raio, conforme ilustrado na Figura 3.5. Dependendo do método de interpolação e do algoritmo de *shading* adotados, esta característica pode ser bastante prejudicial na visualização volumétrica pois pode levar a visualização de partes que normalmente não seriam visualizadas ou até a visualização de artefatos[1].

[1] O termo "artefatos" é utilizado aqui para identificar a presença de pontas ou furos na imagem, gerados a partir de falhas ou erros no algoritmo.

Figura 3.5: Algoritmo de Bresenham 2D.

Assim, o algoritmo de caminhamento deve garantir a amostragem de todos os *voxels* interceptados pelo raio. Esta abordagem utiliza o conceito de *tripod* proposto por Cohen [5], onde cada *voxel* ao longo do raio é adjacente, através de uma de suas faces, ao seu predecessor. Ou seja, o *voxel* seguinte a ser amostrado pode ser determinado pela face pelo qual o raio atravessa quando deixa o *voxel*. A face interceptada pode ser determinada pela relação entre as arestas que partilham o vértice comum e o raio, mostrado na Figura 3.6.

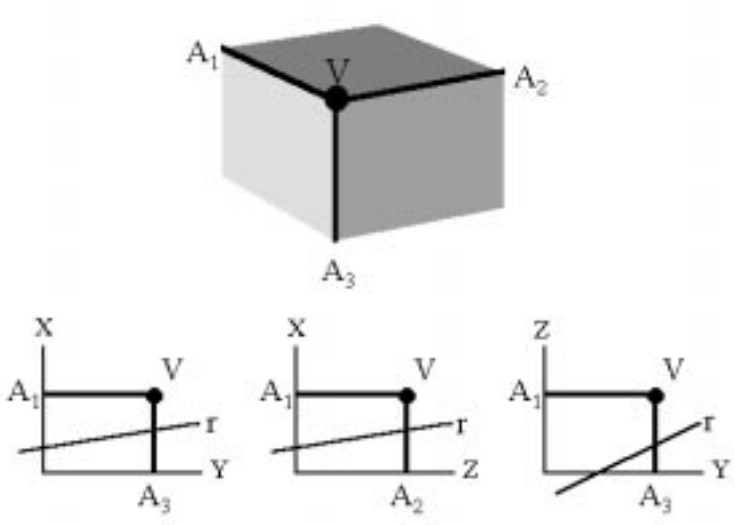

Figura 3.6: Conceito de *Tripod*.

A diferença entre as duas abordagens pode ser claramente verificada, comparando-se a Figura 3.7, que utiliza o algoritmo de Bresenham, e a Figura 3.8, que utiliza o conceito de *tripod*.

Figura 3.7: Algoritmo de Ray-Casting usando Bresenham.

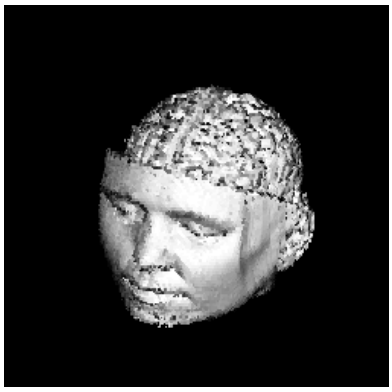

Figura 3.8: Algoritmo de Ray-Casting usando *Tripod*.

3.2.4 Refinamento Sucessivo

Para obter uma visualização rápida, têm sido utilizadas estruturas de dados hierárquicas, do tipo "pirâmides" [22]. Neste tipo de estrutura, decompõe-se o volume de N^3 *voxels* em $\log_2(N)$ volumes, onde o primeiro é o volume original e o segundo é uma média de $2 \times 2 \times 2$ *voxels* do original, resultando num volume com 1/8 da resolução. Este processo se repete até serem definidos $\log(N)$ volumes. Uma vez construída, a visualização é feita percorrendo a pirâmide no nível correspondente à resolução desejada.

Para evitar o uso excessivo de memória e obter uma maior interatividade com o usuário, apresentamos a imagem utilizando a técnica de refinamentos progressivos [18]. A idéia é percorrer uma grade regular sobre o plano de visualização, diminuindo a cada etapa o passo na grade. Assim, a imagem é mostrada rapidamente em baixa resolução, sendo refinada até a resolução final (Figura 3.9). Cada *pixel* da imagem é calculado uma única vez, ficando o tempo de cálculo da cor dos *pixels* inalterado.

Desta forma, obtemos o mesmo efeito pretendido quando na utilização de estruturas do tipo pirâmides, sem, entretanto, ser necessária qualquer memória adicional. O único problema desta técnica é quando a exibição de áreas preenchidas é muito mais lenta do

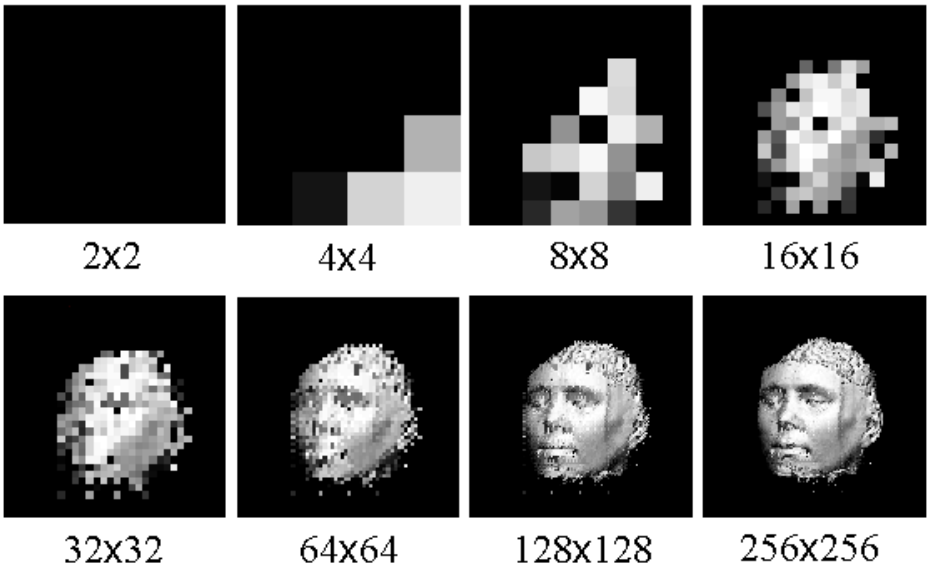

Figura 3.9: Refinamento Sucessivo.

que a exibição de um *pixel*.

3.2.5 Cálculo da Norma do Gradiente

Durante o desenvolvimento do algoritmo de *ray-casting* verificou-se também uma forma de se otimizar o cálculo da norma do gradiente, considerando as duas alternativas clássicas [15]: em vez da "norma 2" (raiz quadrada da soma dos quadrados), pode-se utilizar a "norma 1" (soma dos valores absolutos) ou a "norma infinito" (maior valor absoluto).

No entanto, apesar do ganho de tempo ser significativo, 40% na "norma 1" e 36% na "norma infinito", a qualidade final das imagens ficou prejudicada, o que levou ao abandono dessas alternativas. A degradação da imagem pode ser observada na Figura 3.10.

3.3 Resultados Comparativos Iniciais

Para avaliar as otimizações propostas, os resultados obtidos com a implementação do algoritmo de *ray-casting* com as otimizações propostas foram comparados com os resultados gerados utilizando-se a biblioteca VolPack, que usa o algoritmo de *Shear-Warp* [20]. As comparações foram feitas em uma estação de trabalho SGI Indigo 2 utilizando um conjunto de dados MRI de uma cabeça humana com $128 \times 128 \times 84$ *voxels*, distribuído juntamente com o VolPack.

Para obter os valores característicos dos algoritmos não foi utilizado nenhum sistema gráfico ou *hardware* específico. Assim, ao invés de exibir diretamente a imagem resultante,

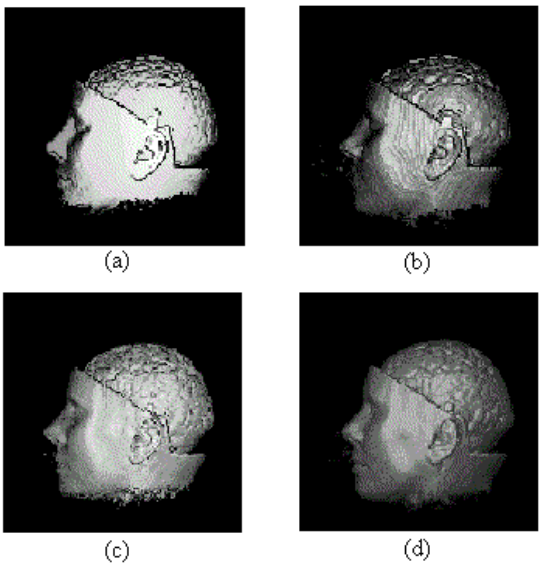

Figura 3.10: Imagens geradas com cálculos alternativos da norma do gradiente: (a) norma infinito, (b) norma 1, (c) norma 2 e (d) norma 2 com *depth cueing*.

	método proposto	VolPack
leitura dos dados	3 s	-
cálculo do gradiente	16 s	-
rendering	26 s	-
gravação da imagem	< 1 s	-
tempo total	45 s	44

Tabela 3.2: Comparação do método proposto com o VolPack

geramos o resultado na resolução final (256×256) num arquivo no formato do VolPack, de modo que a exibição da imagem não interfira na comparação.

É importante observar que a escolha dos mapeamentos de cor e opacidade dependem fortemente do conjunto de dados e da necessidade de observação específica de alguma faixa de valores. Assim, a determinação destes mapeamentos requer um grande conhecimento das características dos dados, uma vez que um mapeamento errado pode acarretar numa grande alteração da imagem, produzindo artefatos ou omitindo detalhes importantes.

A Tabela 3.2 mostra o custo computacional dos dois métodos. Para o método proposto, o tempo de cada fase é explicitado, enquanto que para o VolPack somente o tempo total é apresentado por impossibilidade da obtenção dos valores parciais. Note-se que após a primeira visualização desejarmos rotacionar o volume, apenas o tempo de *rendering* é necessário para as demais visualizações.

As otimizações apresentadas aqui possuem uma implementação simples e mostraram-se apropriadas para a visualização volumétrica, como comprovam os tempos e as imagens

obtidas. Estas otimizações, aliadas à técnica de refinamento sucessivo para exibição, permitem uma forma quase interativa de manipulação e visualização de dados volumétricos para pequenos volumes de dados. No entanto, para grandes volumes de dados, os tempos ainda não são suficiente para possibilitar esta interatividade (Figura 3.11).

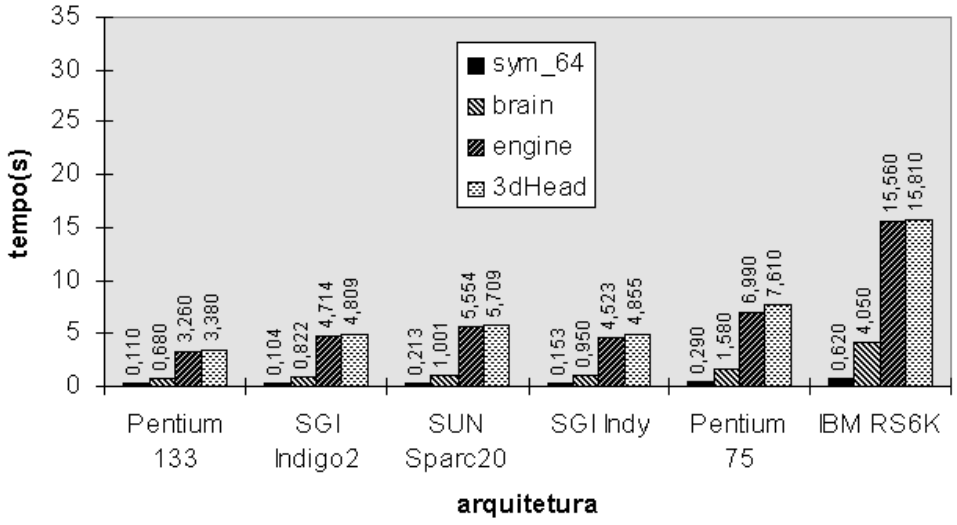

Figura 3.11: Tempos finais do algoritmo seqüencial em cada arquitetura.

Capítulo 4

Visualização Volumétrica com Computação Distribuída

Como os computadores que manipulam volumes ainda estão em desenvolvimento, como por exemplo *IBM Power Visualization System* e o Cube-3 [35], a tendência para se obter uma visualização volumétrica direta rápida é utilizar os computadores paralelos disponíveis. Alguns destes computadores já possuem pacotes de visualização de dados volumétricos [48]. Infelizmente, além de caros, estes computadores possuem características muito particulares, fazendo com que o desenvolvimento de algoritmos e técnicas que se aproveitem totalmente do alto desempenho do *hardware* seja muito complexo e resulte em programas não protáteis.

Para contornar este problema e permitir o uso de algoritmos interativos de visualização volumétrica direta, optou-se por utilizar computadores interligados em redes locais, usando processamento distribuído, fazendo com que cada um funcione como um "processador" (o termo "processador" será utilizado para designar um computador que atue como um elemento de processamento). A vantagem desta abordagem é que podem-se agregar a este "computador pararelo virtual" vários computadores de fabricantes e arquiteturas diferentes.

Nos últimos anos, o uso de paralelismo em visualização vem crescendo rapidamente. Em função disso, foi realizado, em 1993, o primeiro *Parallel Rendering Symposium* — PRS'93, mostrando que existem pesquisas para *hardware* específicos ou para computadores de uso geral interligados por uma rede local (LAN), denominados *clusters*.

Diversos trabalhos apresentados no último *Parallel Rendering Symposium* — PRS'95 mostraram que ainda existem vários problemas a serem resolvidos em áreas como escalabilidade, balanceamento e composição da imagem.

4.1 Computação Paralela em Redes Locais

Normalmente, um maior poder computacional é obtido pelo uso de paralelismo. No entando, o desempenho de algoritmos em paralelo é bastante afetado pela topologia da rede e pelas características da conexão entre os processadores. Estas informações interferem di-

retamente no comportamento e na estrutura dos algoritmos paralelos, podendo, inclusive, torná-los impraticáveis.

De maneira geral, o tempo total necessário para uma visualização volumétrica direta pode ser descrito como a soma dos tempos de decomposição, carga dos dados, processamento, comunicação e exibição.

Um fator que tem grande impacto no desempenho da computação distribuída é a forma de conexão dos processadores. Na arquitetura de *clusters* de computadores, a forma utilizada neste trabalho é a de memória distribuída (*distributed memory* MIMD), onde cada processador tem a sua própria memória, que pode ser acessada, indiretamente, por outros processadores. O acesso a esta memória por outros processadores depende dos tipos diferenciados de acesso providos pela rede a diferentes processadores.

A rede de comunicação entre os processadores também pode ter um grande impacto no desempenho. O estudo de algortimos para computadores paralelos ou distribuídos depende fortemente das características da rede de comunicação [6].

Além disso, as redes locais podem ser heterogêneas, compostas de processadores de várias arquiteturas e/ou desempenho diferentes, e homogêneas, quando compostas de processadores idênticos.

Para maximizar o uso de tempo ocioso em laboratórios existentes, é importante usar redes heterogêneas.

4.2 Indicadores de Desempenho

Quando se projeta um algoritmo para computadores paralelos, deseja-se que ele seja eficiente e utilize toda a potência computacional. Entretanto, a inclusão de cada vez mais processadores em um computador paralelo nem sempre implica em uma redução no tempo de execução [31]. A métrica para a avaliação de eficiência e potência computacional se dá através de **granulosidade**, **aceleração** e **eficiência** do algoritmo paralelo em relação ao seqüencial, além da técnica de distribuição dos dados entre os processadores, denominado **particionamento** (estes termos serão definidos mais adiante).

Uma vez que o desempenho dos algoritmos paralelos depende da arquitetura, apenas leva-se em consideração o tempo de decomposição, o tempo de processamento e o tempo para comunicação e sincronização dos processadores para se estimar o desempenho de um algoritmo em uma determinada arquitetura. Os tempos de carga e exibição são praticamente os mesmos do algoritmo seqüencial, não interferindo na medida de desempenho. Desta forma, assume-se que o algoritmo possui várias tarefas e que cada uma delas pode ser paralelisada, sendo processada concorrentemente.

À relação entre o tempo necessário para o processamento dessa tarefa (R) e o tempo necessário à comunicação (C), dá-se o nome de **granulosidade** e pode ser usada para determinar o ponto de desempenho ótimo de uma determinada arquitetura:

- *coarse-grain*: se R/C é grande, então há pouca comunicação entre os processadores;
- *fine-grain*: se R/C é pequeno, então um tempo considerável é gasto em comunicação.

Como regra geral, pode-se dizer que, quando a relação é grande, podemos usar mais processadores na expectativa de que a comunicação seja distribuída entre os processadores. Se a relação é muito pequena, então normalmente o número ótimo de processadores é menor.

Outra importante medida de desempenho de um algoritmo paralelo é a sua **aceleração** (*speedup*). Alguns autores definem aceleração como a relação entre o tempo do melhor algoritmo seqüencial e o tempo do algoritmo paralelo. Outros autores a definem como a relação entre o tempo do algoritmo paralelo em um processador e o tempo do algoritmo em n processadores de mesma arquitetura [41]. Neste trabalho, utiliza-se como medida de aceleração a primeira relação.

A principal dificuldade é que os algoritmos têm seções facilmente paralelizáveis e outras que são inerentemente seqüenciais. Quando um número de processadores está disponível, as seções paralelizáveis são rapidamente executadas, mas as seções seqüenciais tornam-se "gargalos" computacionais [2]. Esta observação é conhecida como "Lei de Amdahl" e pode ser quantificada da seguinte maneira:

> "Se um algoritmo tem duas seções, uma inerentemente seqüencial e uma totalmente paralelizável, e se a seção seqüencial consome uma fração f do tempo total de processamento, então a sua aceleração é limitada pela seguinte fórmula:
>
> $$speedup \leq \frac{1}{f + \frac{(1-f)}{p}}, \quad \forall p, \tag{4.1}$$
>
> onde p é o número de processadores envolvidos."

Num caso hipotético ideal em que conseguíssemos usar um número infinitos processadores ($p \to \infty$), ainda assim estaríamos limitados a uma aceleração máxima de acordo com a equação 4.2.

$$speedup \leq \frac{1}{f}. \tag{4.2}$$

Para se ter uma idéia da importância da Lei de Amdahl, vamos exemplificá-la com um algoritmo que possua uma seção seqüencial que consuma 25% ($f = 0.25$) do tempo de processamento.

Utilizando a equação 4.2 para o caso hipotético ideal de infinitos processadores, a aceleração máxima obtida seria de 4. Se, por outro lado, utilizássemos 10 processadores, a aceleração máxima seria de aproximadamente 3.08, o que mostra que, independentemente do número de processadores que se utilize, é muito importante que as seções seqüenciais do algoritmo sejam reduzidas ao mínimo necessário.

A **eficiência** de uma paralelização é determinada como uma relação entre a aceleração obtida e o número de processadores necessários para obtê-la, isto é:

$$e = \frac{speedup}{p}.$$

Portanto, pela lei de Amdahl, a eficiência é no máximo:

$$e_{max} = \frac{1}{f\ p}.$$

4.3 Balanceamento e Particionamento

Além das medidas de desempenho discutidas anteriormente, o projeto e a implementação de algoritmos paralelos dependem da forma de distribuição das tarefas entre os processadores, denominada de **balanceamento** (*load balancing*).

Nenhum algoritmo paralelo funcionará bem sem um balanceamento adequado, pois o desempenho geral ficará comprometido pelos processadores mais lentos e por aqueles que receberem tarefas maiores [31, 41].

Assim, para obter-se um bom balanceamento, algumas regras têm que ser seguidas:

- nenhum processador deve ficar disponível ou esperando por outro, sem estar realizando alguma tarefa;
- deve-se evitar a latência da rede fazendo com que os processadores recebam e transmitam mais de um dado de cada vez;
- deve-se reduzir a consulta a outros processadores e o acesso a dados via rede durante a etapa de processamento local da tarefa.

Todos os algoritmos paralelos de visualização volumétrica devem partilhar dois conjuntos de dados entre os processadores. O primeiro conjunto é formado pelos dados a serem visualizados, ou seja, os dados volumétricos propriamente ditos. O outro conjunto contém os dados da imagem, ou seja, posição (x, y) e cor (r, g, b) de cada *pixel*. Assim, os algoritmos paralelos trabalham baseando-se em um dos dois conjuntos. Esta distinção de estratégias cria duas classes de algoritmos paralelos [31]:

particionamento da imagem: a imagem é dividida entre os processadores. Cada processador é responsável por uma partição da imagem e todos os processadores executam as mesmas tarefas concorrentemente em diferente regiões da imagem (Figura 4.1). O processador responsável pela exibição simplesmente coloca cada sub-imagem no lugar apropriado da tela.

particionamento dos dados: o volume de dados é dividido entre os processadores. Cada processador obtém uma imagem intermediária com a partição do volume de sua responsabilidade. O processador responsável pela exibição precisa combinar as sub-imagens para obter a imagem final (Figura 4.2).

Para obter um balanceamento mais flexível e independente das características do volume de dados, utilizamos o particionamento da imagem onde os processadores dividem entre si os *pixels* da tela para gerar a imagem completa. O balanceamento uniforme é mais simples de ser obtido neste tipo de particionamento pois, através da duplicação do volume

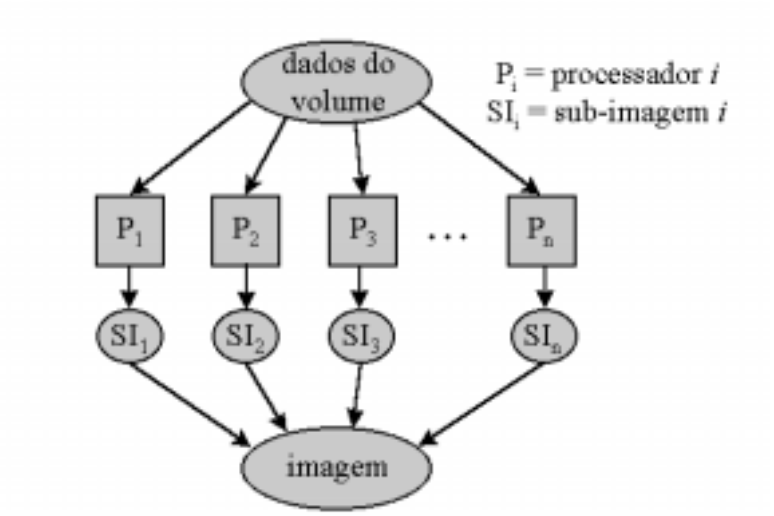

Figura 4.1: Particionamento da Imagem.

de dados pelos processadores, permite que se elimine quase totalmente a comunicação entre os processadores. Esta comunicação fica restrita ao processador responsável pela exibição da imagem e cada processador. Este tipo de estratégia é essencial para minimizar os custos de comunicação em ambientes não dedicados ou que possuam uma rede de baixa velocidade.

No particionamento dos dados, a técnica de terminação do caminhamento no raio quanto se atinge um certo valor máximo torna-se menos eficiente à medida que são agregados mais processadores, pois um percentual cada vez maior de processadores deixa de ser utilizado. Outra desvantagem é que o tempo gasto com a sincronização entre os processadores é maior, pois vários processadores compõem a mesma área da imagem, através do cálculo de sub-imagens, fazendo com que um erro na ordem de composição gere resultados absurdos. No particionamento da imagem isto não ocorre pois cada processador é o único responsável por uma área da imagem.

Além desta facilidade de balanceamento, o particionamento no espaço da imagem em geral não impõe restrições ao aproveitamento de técnicas de aceleração ou aumento de realismo desenvolvidas para algoritmos seqüenciais. Este tipo de particionamento pode ser feito de forma estática ou dinâmica (estes termos são explicados mais adiante), e de várias maneiras (Figura 4.3): *pixels* individuais, blocos de *scanlines* ou blocos retangulares.

No caso de redes locais, o particionamento por *pixels* acarreta uma comunicação excessiva. Por exemplo, para um conjunto de dados de $128 \times 128 \times 84$ *voxels* visualizados em uma SGI Indigo2, o tempo do algoritmo usando o particionamento estático de *pixels* foi de $14,9$ segundos e de $105,6$ segundos para o particionamento dinâmico dos *pixels*, enquando que os demais tipos apresentaram tempos em torno de 1 segundo, como ilustra a Figura 4.4. O mesmo comportamento foi verificado nas demais arquiteturas; e por isso, nos exemplos escolhidos, o particionamento por *pixels* não foi utilizado.

Para o lançamento de raios, o particionamento por *scanlines* permite que se pré-calculem

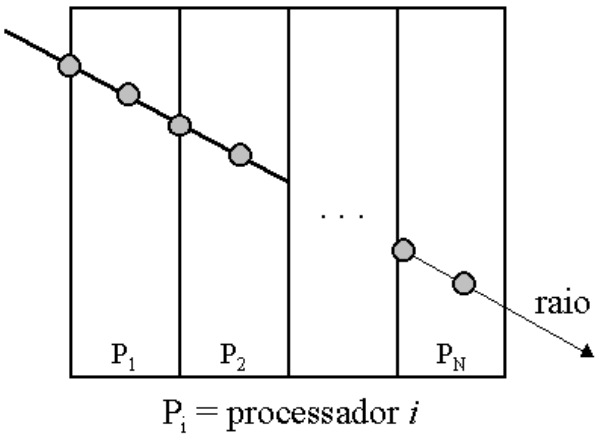

Figura 4.2: Particionamento dos Dados.

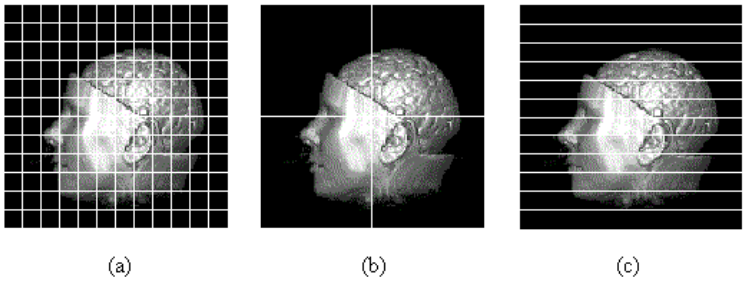

Figura 4.3: Formas de Particionamento: (a) *pixels* individuais, (b) blocos retangulares e (c) blocos de *scanlines*.

os valores iniciais e de incremento de cada raio da *scanline*, devido à coerência nos dados. No particionamento por blocos isto também é possível se os blocos tiverem o mesmo tamanho.

Considerando que a tarefa básica é calcular um grupo contíguo de *pixels*, agrupados na forma de *scanlines* ou blocos de imagens intermediárias, temos que escolher uma forma de atribuir estes grupos aos vários processadores, tendo o cuidado de maximizar o balanceamento, visto que o tempo final será o tempo de processamento do processador mais lento, acrescido do tempo necessário à de comunicação. Para tal, os processadores mais rápidos recebem mais grupos de *pixels* e os mais lentos menos, de forma que o tempo de processamento por processador seja equivalente. Esta atribuição pode ser feita de três formas:

estática contígua: a imagem é dividida em blocos de *scanlines* ou blocos retangulares contíguos, que são atribuídos a cada processador (Figura 4.5). O número de blocos é igual ao número de processadores;

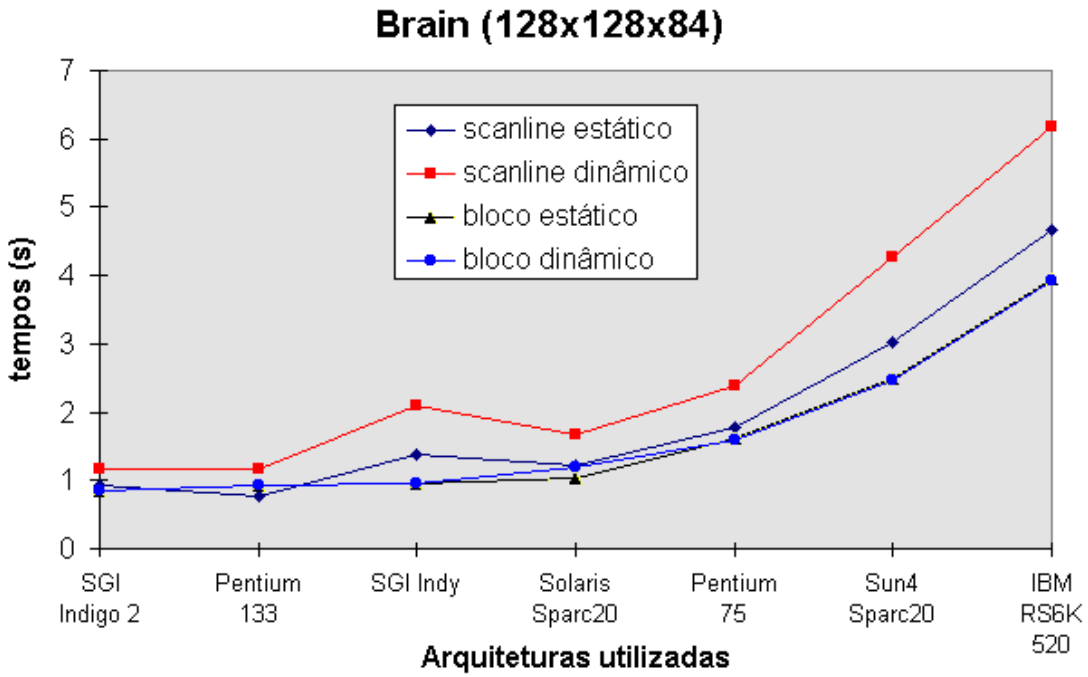

Figura 4.4: Tempos obtidos em cada tipo de particionamento, por arquitetura.

estática intercalada: a imagem é dividida em blocos de *scanlines* ou blocos retangulares e cada um é atribuído a um processador, de forma intercalada (Figura 4.6). O número de blocos é igual ao quadrado do número de processadores. Note-se que, no caso dos blocos retangulares, a atribuição para os processadores deve ser feita de forma circular, de modo a evitar um particionamento contíguo vertical da imagem;

dinâmica: a imagem é dividida em blocos de *scanlines* ou blocos retangulares, que são atribídos a cada processador de acordo com as suas disponibilidades de processamento, durante o tempo de processamento.

O método utilizado baseia-se na distribuição estática contígua até que todos os processadores recebam um grupo de *pixels* a calcular (Figura 4.7). A partir deste ponto, o método passa a fazer o particionamento dinâmico: à medida que for terminando uma tarefa, cada processador disponível receba um novo grupo de *pixels* (Figura 4.8).

A principal vantagem desta técnica é que ela permite minimizar a influência de processadores lentos no desempenho final, fazendo com que os mais rápidos processem mais *pixels*. Outra vantagem é que pode-se obter uma "avaliação" de desempenho de cada processador a partir das primeiras partições calculadas.

Outro fato importante é que este método se adapta ao número de processadores existentes, ou seja, se o número de processadores for maior ou igual ao de partições, então somente o particionamento estático será utilizado. Se, por outro lado, um único processador for

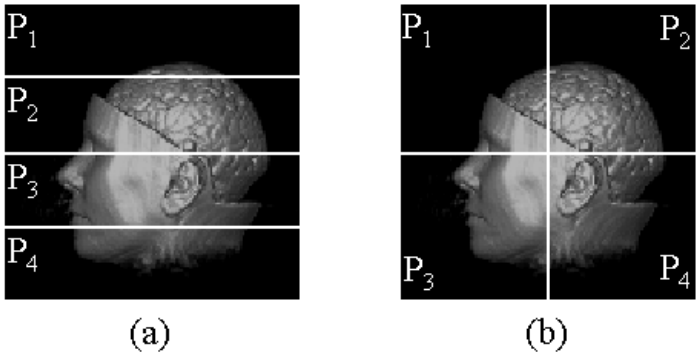

Figura 4.5: Particionamento estático contíguo.

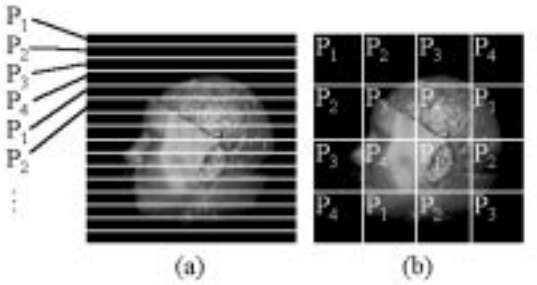

Figura 4.6: Particionamento estático intercalado.

utilizado, então o particionamento dinâmico funcionará de forma idêntica ao algoritmo seqüencial, onde uma *scanline* ou um bloco são processados um após o outro.

4.4 Indicadores de Desempenho em Ambientes Heterogêneos Não Dedicados

As medidas tradicionais de aceleração (*speedup*) descritas anteriormente são adequadas para ambientes homogêneos e dedicados. Em um ambiente compartilhado e heterogêneo, como o escolhido, o tempo do melhor algoritmo seqüencial depende do processador utilizado e de seu processamento local. Neste trabalho, a métrica adotada corresponde ao tempo medido no melhor processador da rede sem o compartilhamento com outros usuários.

Além disso, a análise de tempos em redes heterogêneas não dedicadas requer uma medida que indique a perda devida à comunicação e ao desempenho individual de cada máquina. Para este fim, este trabalho propõe uma medida de tempo denominada "tempo ideal".

O tempo ideal em cada tipo de particionamento é o tempo que o algoritmo usaria

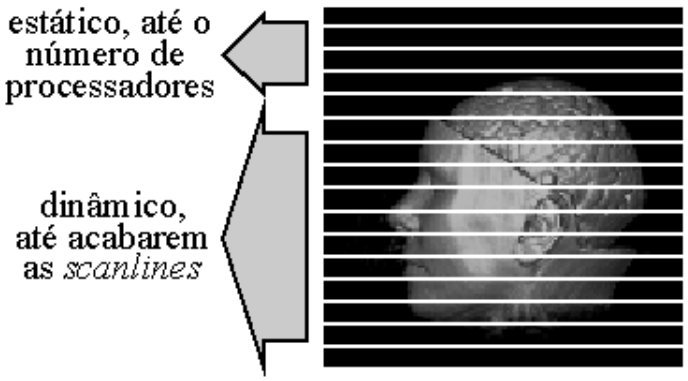

Figura 4.7: Particionamento proposto por *scanlines*.

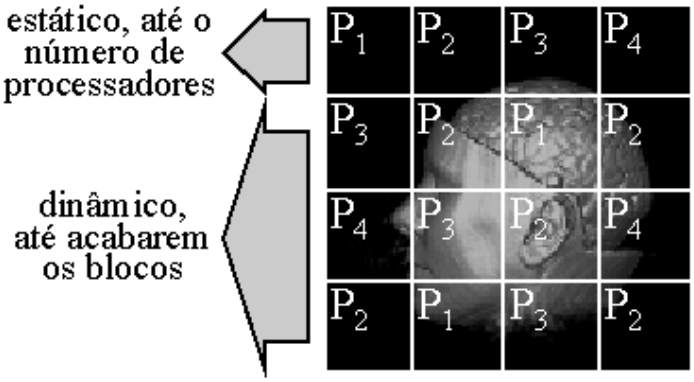

Figura 4.8: Particionamento proposto por blocos.

se não existissem perdas (*overheads*) devidas à carga nos processadores e à comunicação entre eles, e baseia-se no tempo obtido com o algoritmo seqüencial em todas as arquiteturas utilizadas.

Para os particionamentos estáticos, seja por *pixel*, *scanline* ou blocos, o tempo ideal é o tempo que a arquitetura mais lenta leva para processar o seu percentual do conjunto de dados: nesta arquitetura, cada processador recebe a mesma quantidade de dados e o processo só acaba quando o último processador terminar. Assim, este tempo é obtido por:

$$t_{ideal} = \frac{\max(t_i)}{np}, \tag{4.3}$$

onde t_i é o tempo que o processador i leva para calcular toda a imagem e np é o número de processadores.

Por exemplo, se tivermos 3 processadores diferentes, que executam o algoritmo seqüencial em 1, 2 e 3 segundos, respectivamente, processando 100 blocos, então o cálculo do tempo ideal para o particionamento estático de uma imagem particionada em 100 blocos seria de 1 segundo, determinado pelo tempo da arquitetura mais lenta, 3 segundos, dividido pelo número de processadores, 3.

Para os particionamentos dinâmicos, seja por *pixel*, *scanline* ou bloco, o tempo ideal é determinado calculando-se o balanceamento ótimo, que é proporcional ao desempenho do algoritmo seqüencial em cada arquitetura. Assim, o balanceamento ótimo é obtido quando todos os processadores terminam simultaneamente. O tempo ideal neste caso é dado por:

$$t_{ideal} = q_1 \times \frac{t_1}{N} = q_2 \times \frac{t_2}{N} = \cdots = q_n \times \frac{t_n}{N}, \tag{4.4}$$

onde q_i é a quantidade de *pixels*, *scanlines* ou blocos a serem calculados e t_i o tempo de processamento na arquitetura i para uma quantidade total de N *pixels*, *scanlines* ou blocos.

Como

$$\sum_{i=1}^{np} q_i = N \tag{4.5}$$

e pela equação 4.4

$$q_i = q_j \frac{t_j}{t_i}, \tag{4.6}$$

temos

$$N = \sum_{i=1}^{np} q_j \frac{t_j}{t_i} = q_j \sum_{i=1}^{np} \frac{t_j}{t_i} \tag{4.7}$$

e

$$q_j = \frac{N}{\sum_{i=1}^{np} \frac{t_j}{t_i}}. \tag{4.8}$$

Substituindo-se a equação 4.8 na equação 4.4 tem-se o tempo ideal em função apenas dos tempos dos processadores:

$$t_{ideal} = \frac{1}{\sum_{i=1}^{np} \frac{1}{t_i}}. \tag{4.9}$$

Utilizando-se o mesmo exemplo do particionamento estático, temos que o tempo ideal é de 0.54 segundos. Assim, se neste ambiente obtivéssemos um tempo de 0.6 segundos, então o desempenho relativo seria de

$$\frac{t_{ideal}}{t_{obtido}} \times 100\% = \frac{0.54}{0.6} \times 100\% = 90.90\%.$$

Capítulo 5

Implementação e Resultados

Este capítulo apresenta o ambiente utilizado (*hardware* e *software*), a arquitetura do programa distribuído desenvolvido, os exemplos escolhidos para teste, o cálculo dos tempos ideais, os resultados obtidos para os diversos tipos de particionamentos e uma análise destes.

5.1 Ambiente Utilizado

Para a implementação das idéias deste trabalho, escolheu-se o PVM – *Parallel Virtual Machine* – que é um ambiente de desenvolvimento e execução de aplicações em paralelo que envolvem componentes ou tarefas, relativamente independentes, que interagem entre si [13].

O PVM propõe-se a atuar sobre um grupo heterogêneo de computadores interconectados por uma ou mais redes locais. É composto por uma biblioteca de rotinas, ou primitivas, que podem ser incorporadas às linguagens de programação existentes (C e Fortran). Entre outras facilidades, essas rotinas admitem a inicialização e o término de tarefas dentro da rede, a comunicação e a sincronização entre elas.

No modelo PVM, cada computador pode acessar somente sua memória e toda a comunicação é efetuada através de troca de mensagens. Assim, para se fazer uma troca de dados, devemos acessá-los na memória de um processador providenciando informações de "como" e "para onde" serão enviados; só então os dados são fisicamente transmitidos e postos na memória do computador receptor. O PVM implementa a parte de comunicação de dados através do controle de envio destas mensagens entre os processadores. Como consequência, temos dois fatores intrínsecos de degradação da eficiência da aplicação: o tempo de comunicação, que é o tempo necessário para a troca de informações; e o tempo de sincronização, que é o tempo de espera para que certas partes do algoritmo sejam completadas a fim de liberar a continuidade da execução. A comunicação e a sincronização entre os processadores é controlada pelo próprio PVM, garantindo a confiabilidade na entrega das mensagens. Ainda como vantagem, fazendo uso do PVM, garante-se a portabilidade do programa para diversas plataformas, no que diz respeito à comunicação via rede.

A Figura 5.1 mostra o tempo médio gasto para a troca de mensagens entre dois computadores cresce com o tamanho da mensagem. Assim, mensagens menores garantirão uma transferência de dados mais rápida entre os processadores.

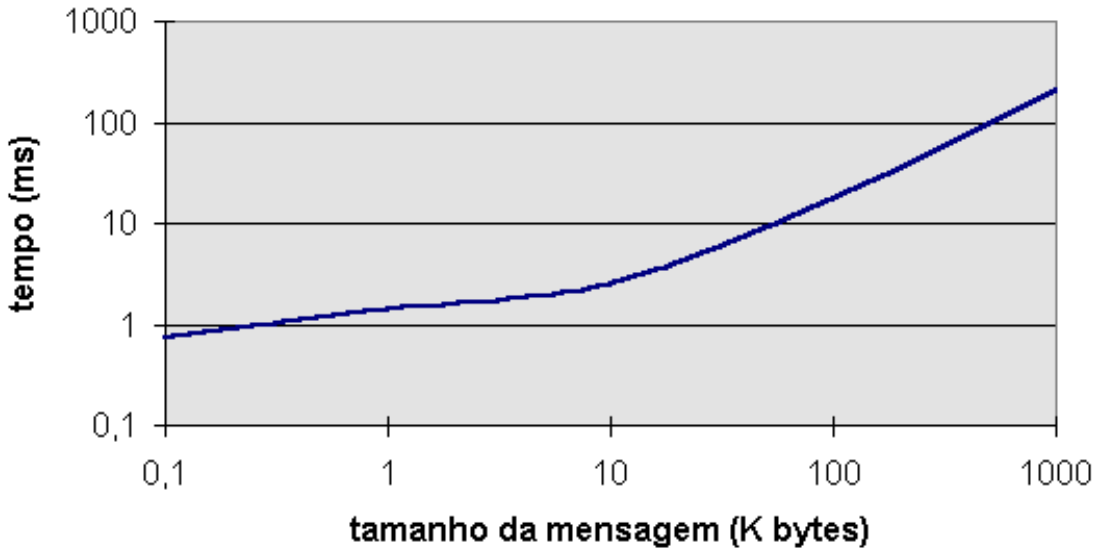

Figura 5.1: Tempo gasto pelo PVM para a troca de mensagens.

A utilização do PVM permitiu a criação de um ambiente de computação distribuída que, praticamente, envolveu todos os processadores disponíveis nos laboratórios do TeCGraf. Os processadores utilizados foram:

- 1 Silicon Graphics Indigo 2 com 64 M*bytes*;
- 7 Silicon Graphics Indy com 32 M*bytes*;
- 2 IBM RS/6000 520 com 32 M*bytes*de memória;
- 3 SUN SPARCstation 20 com 64 M*bytes*de memória;
- 2 Microcomputadores Pentium 133 Mhz (Linux) com 16 M*bytes*de memória;
- 1 Microcomputador Pentium 75 Mhz (Linux) com 16 M*bytes*de memória.

Pode-se notar que são vários os fabricantes e várias as configurações que compõem este ambiente. Além disso, os processadores não estão dedicados ao uso de computação distribuída, possuindo na maioria das vezes usuários locais e remotos executando programas.

A escolha da ordem de agregação dos processadores leva em conta a disponibilidade média dos processadores na rede: os processadores mais utilizados ou mais lentos são escolhidos por último. As figuras da seção 5.5 apresentam o tempo total (decomposição, processamento, comunicação e exibição) em função do número de processadores, agregados segundo a ordem descrita na Tabela 5.1, para cada tipo de particionamento.

número de processadores	arquiteturas utilizadas
2	Pentium 133 Mhz (2×)
3	SGI Indigo2
4	SUN Sparc20
5	SUN Sparc20
6	SGI Indy
7	SGI Indy
8	SGI Indy
9	SGI Indy
10	SGI Indy
11	SGI Indy
12	SGI Indy
13	Pentium 75 Mhz
14	SUN Sparc20
15	IBM RS6K
16	IBM RS6K

Tabela 5.1: Ordem de agregação dos processadores.

5.2 Implementação

No método inicialmente implementado, cada *scanline* ou bloco calculado era imediatamente exibido. Infelizmente, logo nos primeiros testes este esquema não se mostrou eficiente, visto que o tempo para se exibir uma *scanline* ou bloco não é desprezível nos sistemas de janelas baseados em X11, pois o processador que acabou de calcular uma *scanline* ou bloco tem que esperar que esta seja exibida antes de receber uma nova para calcular, pois a tarefa de visualização está presente no *master*. Assim, se o processo de exibição dos resultados for lento, todo o mecanismo de paralelização será prejudicado.

A solução adotada para este problema foi a separação das tarefas de particionamente e visualização em processadores diferentes (Figura 5.2). Desta forma, o tempo necessário à exibição de cada *scanline* ou bloco não interfere no particionamento e submissão de uma nova *scanline* ou bloco para o processador que está disponível.

Uma outra vantagem obtida com esta separação é que o processador responsável pela visualização pode executar um método de *preview* da imagem, de forma seqüencial, enquanto a imagem com alta qualidade está sendo calculada pelos demais processadores, aumentando a interatividade com o usuário.

Deste modo, criou-se um ambiente (Figura 5.2) no qual podem ser testadas técnicas de particionamento independentemente das técnicas de visualização ou das técnicas de processamento, e vice-versa.

Desta forma, podem-se utilizar num mesmo ambiente, variações de técnicas sem que

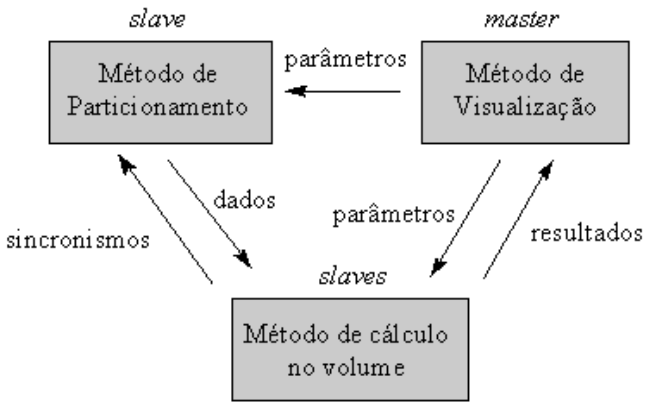

Figura 5.2: Ambiente Final de Desenvolvimento.

haja a necessidade de alteração de sua arquitetura, desde que o protocolo (Figura 5.2) seja mantido. Assim, os resultados dos processamentos são enviados para o processador responsável pela visualização (*viewer*), determinado durante a inicialização dos processadores *slaves*. Para um particionamento diferente, bastaria informar o novo processador responsável por esta tarefa (*main*) através de parâmetros globais transmitidos entre os processadores pelo protocolo de mensagens. Para um novo método de cálculo do volume (*raycast*), basta que o processador responsável pelo particionamento seja informado de quais os processadores que devem ser utilizados na distribuição das tarefas.

A Figura 5.3 mostra a interface do ambiente para Visualização Volumétrica Distribuída – DVV – implementado.

5.3 Dados de Exemplo

Foram analisados quatro conjuntos de dados que estão disponíveis na Internet:

"brain" $(128 \times 128 \times 84)$**:** cabeça humana com parte do cérebro exposto, apresentado na seção 3.3 (Figura 5.4), distribuído junto com a biblioteca VolPack;

"3dHead" $(256 \times 256 \times 109)$**:** cabeça humana (Figura 5.5) obtida na *State University of New York at Stone Brook*;

"engine" $(256 \times 256 \times 110)$**:** peça mecânica (Figura 5.6), também distribuída juntamente com o VolPack;

"sym64" $(64 \times 64 \times 64)$**:** conjunto de dados gerado sinteticamente (Figura 5.7), obtido na *Ohio State University*.

mensagem	origem	destino	descrição
INIT	*viewer*	*main, raycast*	dados do ambiente
START	*viewer*	*main*	inicia particionamento
QUERY_PIXEL	*main*	*raycast*	*pixel* dinâmico
QUERY_PIXELS	*main*	*raycast*	*pixel* estático
QUERY_SCANLINE	*main*	*raycast*	*scanline* dinâmica
QUERY_SCANLINES	*main*	*raycast*	*scanline* estática
QUERY_BLOCK	*main*	*raycast*	bloco dinâmico
QUERY_BLOCKS	*main*	*raycast*	bloco estático
OK	*main, raycast*	*viewer*	inicialização ok
VIEW	*viewer*	*main*	dados de particionamento
PARAM	*viewer*	*raycast*	dados de visualização
RESULT_PIXEL	*raycast*	*viewer*	*pixel* calculado
RESULT_SCANLINE	*raycast*	*viewer*	*scanline* calculada
RESULT_BLOCK	*raycast*	*viewer*	bloco calculado
IDLE	*raycast*	*main*	processador disponível
DONE	*main*	*viewer*	processamento terminado

Tabela 5.2: Protocolo Final de Troca de Mensagens.

O primeiro e o último conjunto de dados são relativamente pequenos comparados com os demais. Esta variação foi escolhida para analisar a influência do tamanho do volume dos dados nas técnicas de particionamento propostas (veja seção 5.6).

Como o tempo do algoritmo de *ray-casting* é influenciado pelo ângulo de visualização, optou-se por posicionar o plano de visualização numa posição genérica, mostradas nas Figuras 5.5 a 5.7, evitando-se posições alinhadas que favorecessem os resultados. As posições adotadas são as apresentadas nas figuras anteriores, que também permitem uma avaliação dos mapeamentos adotados para as transformações de densidade em cor e opacidade.

5.4 Tempos Ideais

O cálculo dos tempos ideais, definidos na seção 4.4, toma como base o tempo do algoritmo seqüencial executado em cada tipo de arquitetura. A Tabela 5.3 mostra estes tempos obtidos, executando-se localmente em cada arquitetura sem outros usuários. No entanto, os serviços de rede continuaram disponíveis (NFS, NIS, etc).

Com base na equação 4.3 para os particionamentos estáticos e na equação 4.9 para os particionamentos dinâmicos, a Tabela 5.4 mostra os tempos ideais em função do número de processadores, para cada conjunto de dados.

Podemos notar que, nos particionamentos estáticos, a inclusão de um processador lento faz com que o tempo ideal piore, pois a distribuição das tarefas entre os processadores é

dado	Pentium 133	SGI Indigo2	SUN Sparc20	SGI Indy	Pentium 75	IBM RS6K
brain	0,680	0,822	1,001	0,950	1,580	4,050
3dHead	3,380	4,809	5,709	4,855	7,610	15,810
engine	3,260	4,714	5,554	4,523	6,990	15,560
sym64	0,110	0,104	0,213	0,153	0,290	0,620

Tabela 5.3: Tempos do algoritmo seqüencial para o cálculo do tempo ideal.

número de	brain		3dHead		engine		sym64	
processadores	est.	din.	est.	din.	est.	din.	est.	din.
2	0,340	0,340	1,690	1,690	1,630	1,630	0,055	0,055
3	0,274	0,241	1,603	1,251	1,571	1,211	0,037	0,036
4	0,250	0,194	1,427	1,026	1,389	0,994	0,053	0,031
5	0,200	0,162	1,142	0,870	1,111	0,843	0,043	0,027
6	0,167	0,139	0,951	0,737	0,926	0,711	0,035	0,023
7	0,143	0,121	0,816	0,640	0,793	0,614	0,030	0,020
8	0,125	0,107	0,714	0,566	0,694	0,541	0,027	0,018
9	0,111	0,096	0,634	0,507	0,617	0,483	0,024	0,016
10	0,100	0,088	0,571	0,459	0,555	0,436	0,021	0,014
11	0,091	0,080	0,519	0,419	0,505	0,398	0,019	0,013
12	0,083	0,074	0,476	0,386	0,463	0,366	0,018	0,012
13	0,122	0,071	0,585	0,367	0,538	0,348	0,022	0,012
14	0,113	0,066	0,544	0,345	0,499	0,327	0,021	0,011
15	0,270	0,065	1,054	0,338	1,037	0,320	0,041	0,011
16	0,235	0,064	0,988	0,331	0,972	0,314	0,039	0,011

Tabela 5.4: Tempos ideais (em segundos) em função do número de processadores, para cada conjunto de dados.

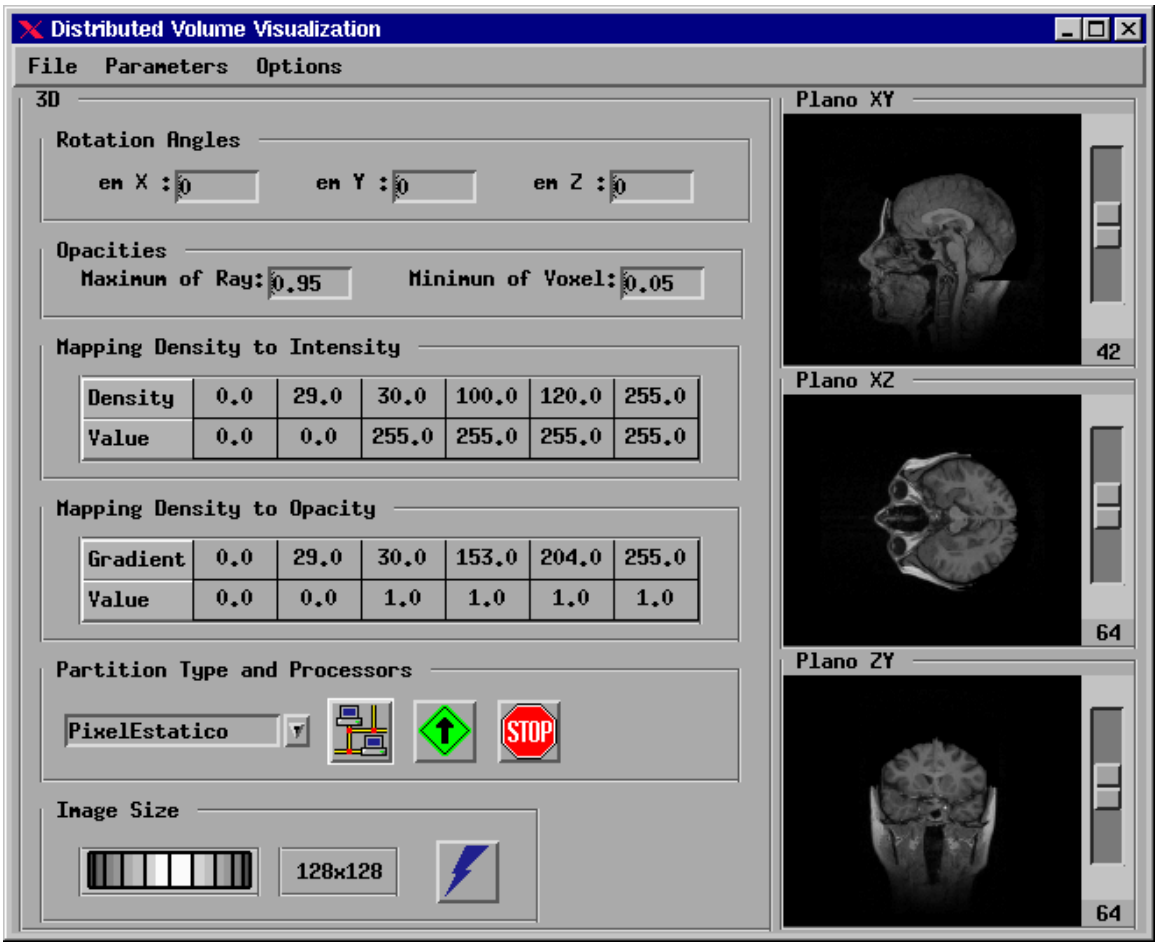

Figura 5.3: Interface do ambiente implementado (DVV).

uniforme, fazendo com que o processador mais lento determine o tempo final do processo.

5.5 Resultados Obtidos

Os resultados aqui apresentados foram obtidos a partir de um arquivo de *log* da versão distribuída do algoritmo. Este arquivo registra para cada execução do programa: data, hora, arquivo de dados, tipo de particionamento, número de processadores, tempo de parede, tempo de exibição e tempo de cálculo. Nos particionamentos estáticos, o registro inclui ainda: tempo de cálculo, tempo de *cpu*, número de *swaps* e número de *page faults*, para cada processador. Nos particionamentos dinâmicos, o registro inclui a quantidade de blocos calculados, por processador.

Para reduzir a influência do tempo de exibição, os testes foram feitos sem a exibição de imagens parciais que ocorrem no refinamento sucessivo, descrito na Seção 3.2.4.

O arquivo de *log* aqui analisado possui aproximadamente dois mil registros, obtidos

Figura 5.4: Imagem do dado “brain”.

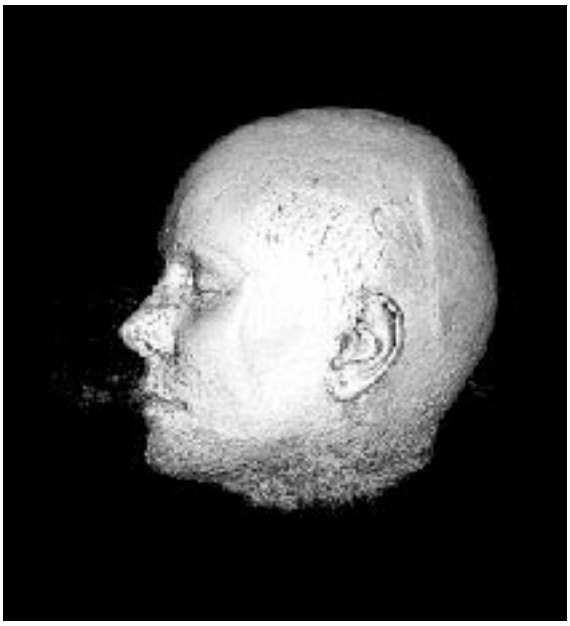

Figura 5.5: Imagem do dado “3dHead”.

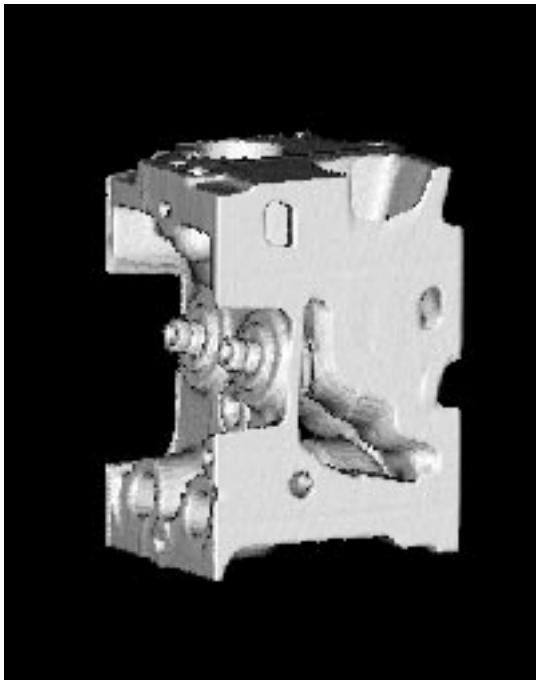

Figura 5.6: Imagem do dado “engine”.

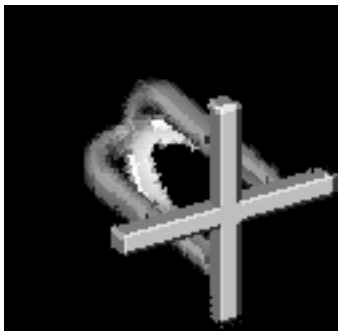

Figura 5.7: Imagem do dado “sym64”.

com uma média de 30 execuções em horários e dias diferentes, para cada tipo de particionamento, em cada um dos conjunto de dados, para cada grupo de processadores.

No entanto, a apresentação de todos estes dados neste documento seria desnecessariamente extensa e maçante. Por isso, optou-se inicialmente por selecionar os campos mais relevantes do registro de *log*: arquivo de dados, tipo de particionamento e tempo de parede.

Os campos de data e hora foram eliminados utilizando-se o valor mínimo das diversas medidas para cada tipo de dado e cada tipo de particionamento. Esta escolha é justificada com base na pequena variação do conjunto, medida pelo desvio padrão da amostra, mostrado na Figura 5.8. Excetuam-se alguns tempos obtidos com os particionamentos estáticos, que atribuídos a processadores sobrecarregados, apresentaram tempos de até 16 vezes maiores do que o valor médio observado.

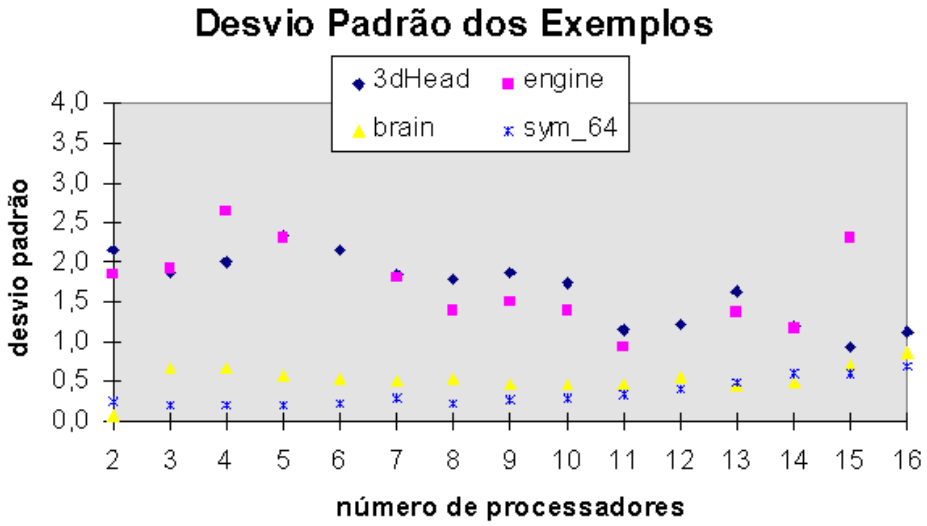

Figura 5.8: Desvio Padrão da Amostra.

Os campos específicos por processador foram utilizados apenas para identificar comportamentos atípicos, como o dos particionamentos estáticos descritos anteriormente.

O tempo de parede é o mais relevante para o usuário e engloba os tempos de particionamento, cálculo, comunicação e exibição. Estes tempos são difíceis de serem obtidos individualmente nos particionamentos dinâmicos, pois a inclusão desta tarefa de supervisão no particionador acarretaria uma alteração no comportamento do algoritmo, interferindo na medida.

Os tempos mínimos obtidos para cada particionamento (bloco dinâmico, bloco estático, *scanline* dinâmica e *scanline* estática), para cada exemplo ("3dHead", "engine", "brain" e "sym64") e para cada configuração de $2, 3, 4, \ldots, 16$ processadores são mostrados nas Figuras 5.9, 5.10, 5.11 e 5.10.

As Figuras 5.13, 5.14, 5.15 e 5.16 mostram os tempos obtidos para cada exemplo em função dos particionamentos e do número de processadores. As Figuras 5.17, 5.18, 5.19 e

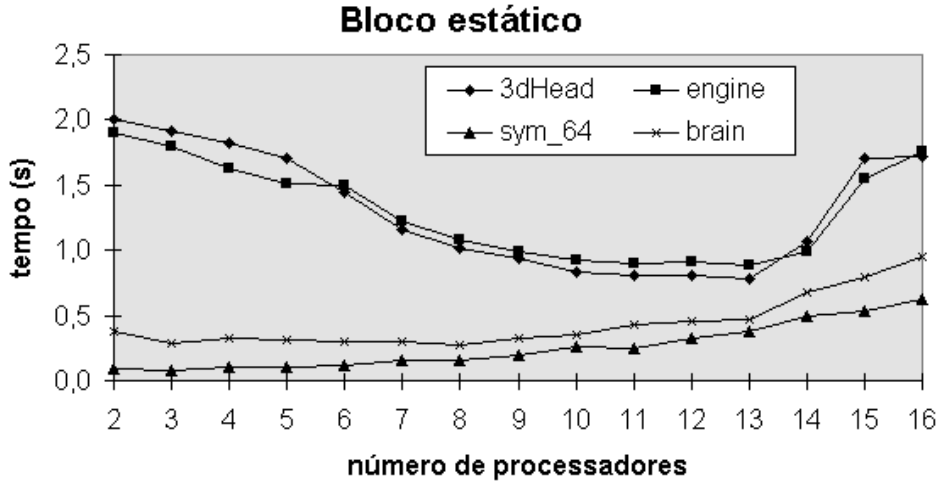

Figura 5.9: Tempos obtidos para o particionamento estático por blocos.

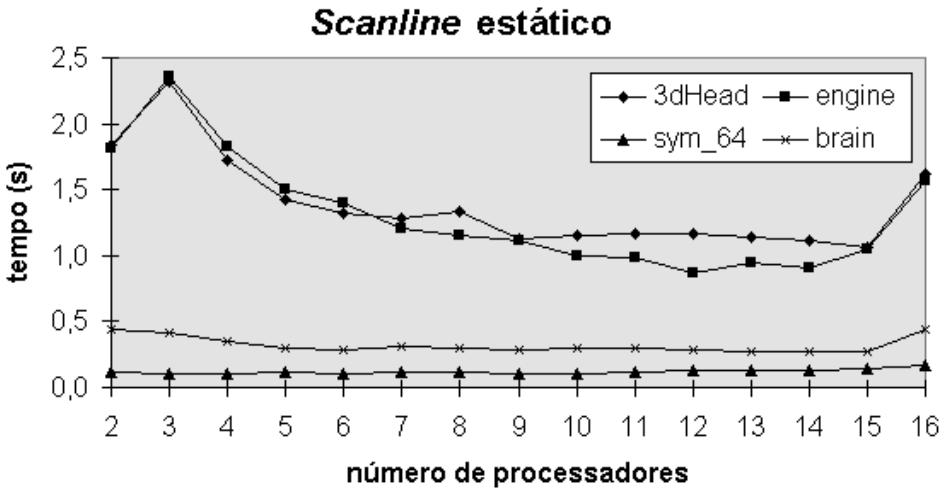

Figura 5.10: Tempos obtidos para o particionamento estático por *scanline*.

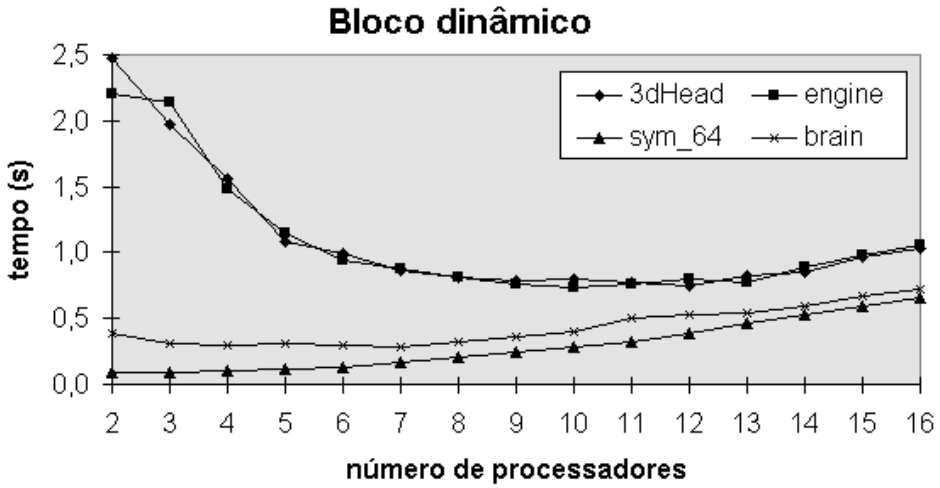

Figura 5.11: Tempos obtidos para o particionamento dinâmico por blocos.

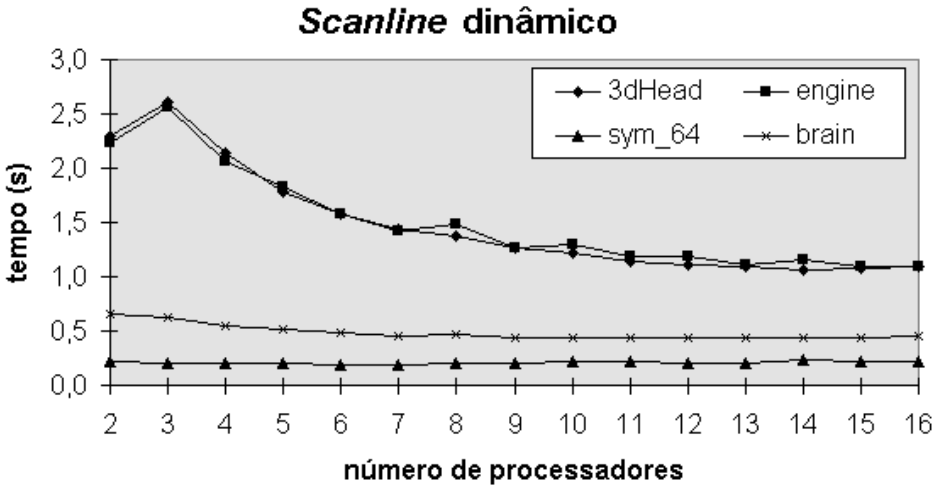

Figura 5.12: Tempos obtidos para o particionamento dinâmico por *scanline*.

5.20 mostram a eficiência medida como uma percentagem do tempo ideal sobre o tempo obtido.

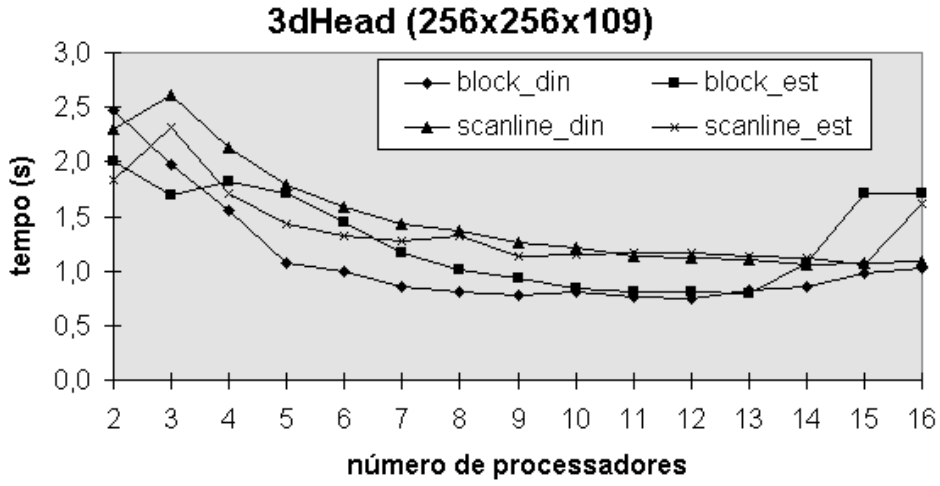

Figura 5.13: Tempos obtidos para os dados "3dhead".

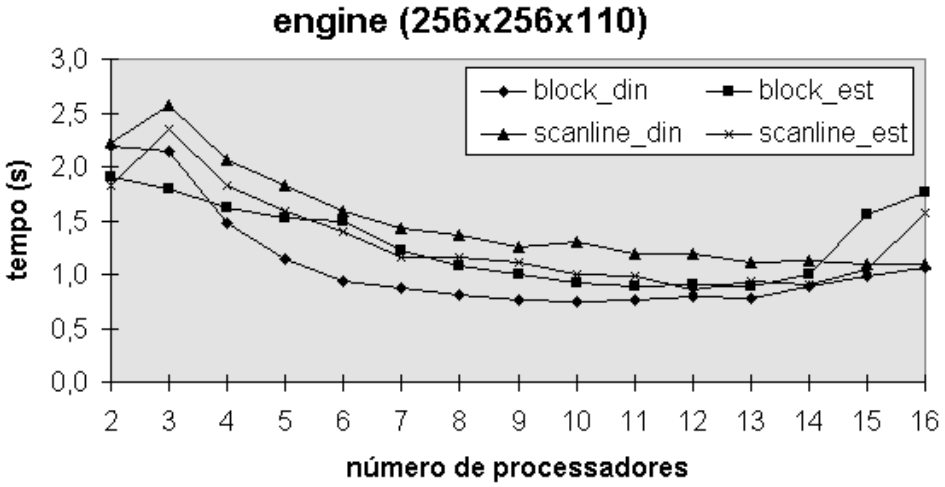

Figura 5.14: Tempos obtidos para os dados "engine".

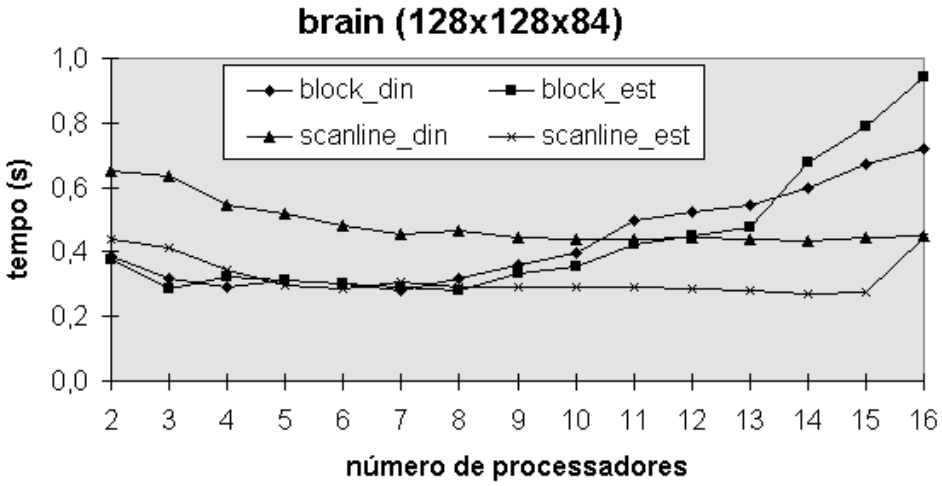

Figura 5.15: Tempos obtidos para os dados "brain".

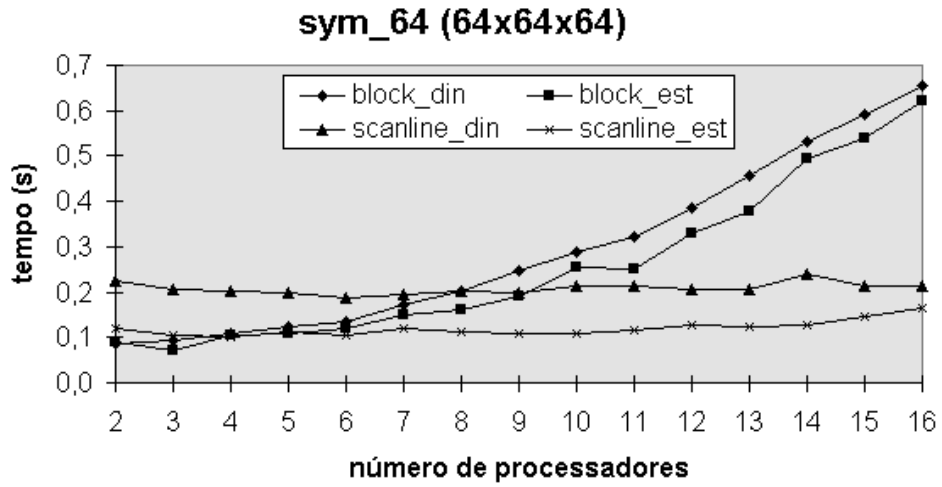

Figura 5.16: Tempos obtidos para os dados "sym64".

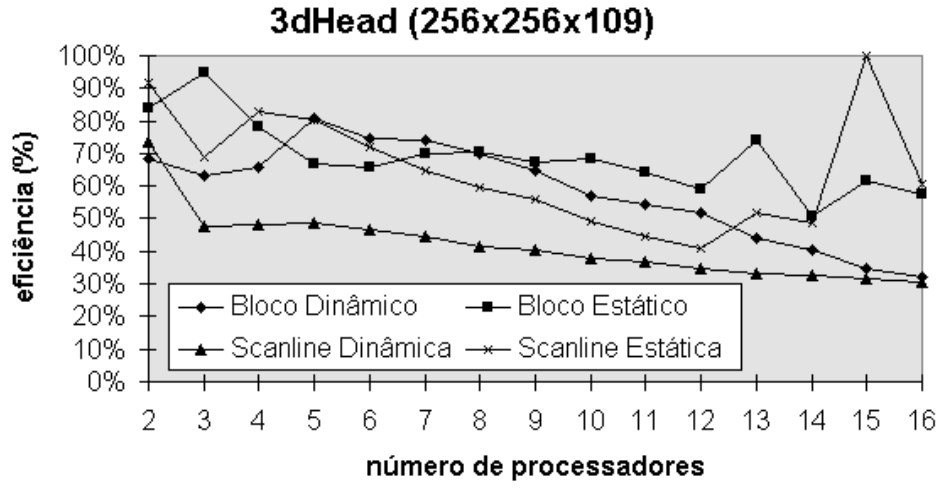

Figura 5.17: Eficiências obtidas para os dados "3dhead".

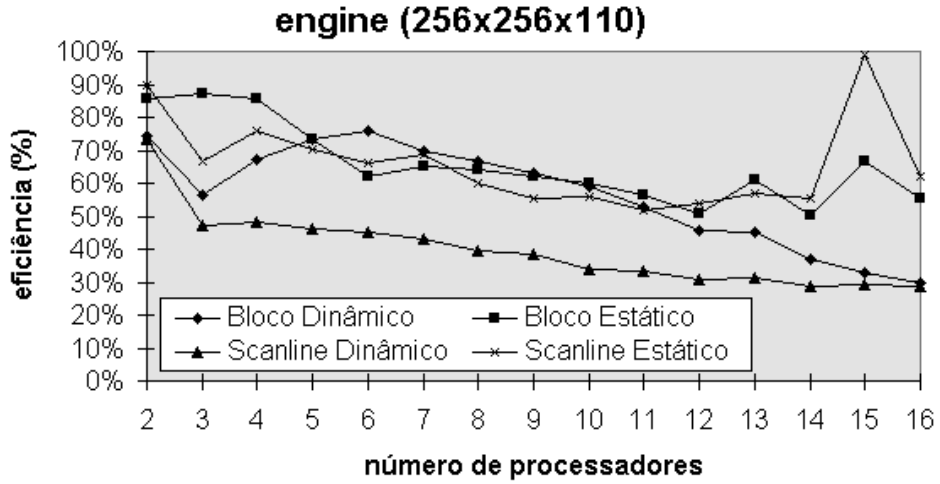

Figura 5.18: Eficiências obtidas para os dados "engine".

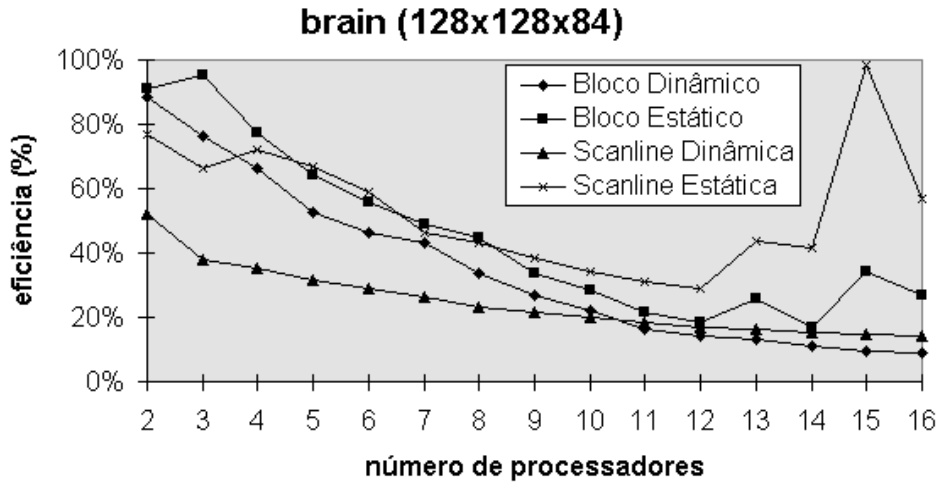

Figura 5.19: Eficiências obtidas para os dados "brain".

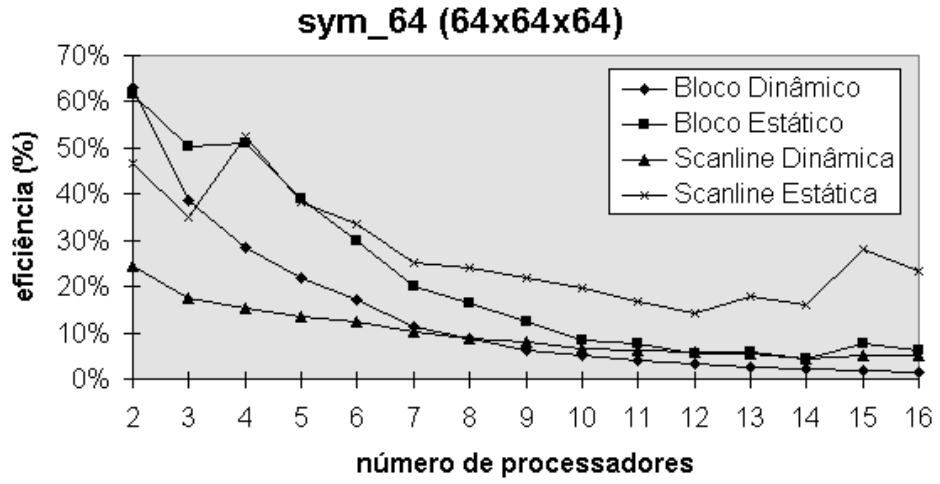

Figura 5.20: Eficiências obtidas para os dados "sym64".

5.6 Análise dos Resultados

Conforme ilustram as Figuras 5.9, 5.10, 5.11 e 5.12, existem dois tipos de comportamento: dados grandes (3dhead e engine) e dados pequenos (brain e sym64)[1]. Nos dados grandes, a agregação inicial de novos processadores reduz o tempo final, pois os sub-volumes a calcular são menores. No entanto, conforme se agregam mais processadores, a comunicação aumenta, fazendo com que esta tendência se altere, pois o tempo de comunicação, seja para enviar resultados ou para solicitar novos dados, começa a ser superior ao tempo de cálculo do sub-volume. Este crescimento é mais acentuado nos particionamentos estáticos do que nos particionamentos dinâmicos. A conjugação destas duas tendências, diminuição e crescimento, sugere um valor ótimo entre 10 e 12 processadores, no ambiente utilizado, para este tipo de dado.

Nas Figuras 5.13 e 5.14 pode-se observar que, devido ao tamanho dos conjuntos de dados, a tendência nos dois tipos de particionamento foi a mesma, mostrando que o tempo de comunicação não foi muito relevante. Isto também pode ser verificado nas Figuras 5.17 e 5.18, que mostram que a eficiência se manteve acima de 50%, com exceção do particionamento dinâmico por *scanline*, que apesar de ter apresentado a mesma tendência, se manteve sempre abaixo deste valor. Conforme mais processadores são agregados, a tendência é de melhoria no tempo obtido. No entanto, os tempos obtidos com particionamentos estáticos apresentam saltos de piora quando se agrega um processador lento. Essa tendência é confirmada pelas medidas obtidas, pois estes tipos de particionamento também são mais sujeitos à carga local nos processadores. O desvio padrão nestes tipos de particionamento também foi maior.

O menor tempo para o "3dhead" foi de $0.745s$, obtido com o particionamento dinâmico por blocos com 12 processadores, com uma eficiência de 52%. O menor tempo do "engine" foi de 0.739, obtido com o mesmo tipo de particionamento, com 10 processadores e com um eficiência de 59%. Com estes valores, as acelerações máximas obtidas foram de 4.5 e 4.4 para "3dhead" e "brain", respectivamente.

Note-se que, se o tempo ideal mostrado na Tabela 5.4 pudesse ser alcançado, estas acelerações seriam de $\frac{3.38}{0.386} = 8.75$ e $\frac{3.26}{0.436} = 7.48$, respectivamente.

Nos dados pequenos, o tempo de comunicação supera rapidamente o tempo de cálculo, conforme se agregam os processadores. Assim, a identificação de um valor ótimo é mais difícil.

Pode-se verificar nas Figuras 5.15 e 5.16, que os tempos obtidos no particionamento por *scanlines* segue uma tendência decrescente, o que não ocorre no particionamento por blocos. Essa diferença ocorre porque o número de *scanlines* só é proporcional ao tamanho da imagem, enquanto que o número de blocos é determinado pelo quadrado do número de processadores. Na Figura 5.15, pode-se observar que o cruzamento do particionamento por blocos e o particionamento por *scanline* dinâmico ocorre quando o número de *scanlines* é aproximadamente igual ao número de blocos, ou seja, com 11 processadores, que é apro-

[1]Naturalmente, esta classificação dos dados em "grandes" e "pequenos" depende não só do tamanho dos dados, mas também do ambiente utilizado.

ximadamente $\sqrt{128}$. Na Figura 5.16, o cruzamento ocorre com 8 processadores, que é $\sqrt{64}$.

Os gráficos de eficiência dos dados pequenos, Figuras 5.19 e 5.20, confirmam que o tempo de comunicação é mais relevante que o tempo de cálculo.

Na Figura 5.19, as eficiências menores que 50% ilustram que a comunicação entre os processadores é excessiva, fazendo com que, mesmo que se obtenha um tempo menor, esta redução não seja significativa. Desta forma, o particionamento estático por blocos atingiu um tempo mínimo de $0.287s$ usando 4 processadores com uma eficiência de 95%, que pode ser comparado ao menor tempo geral, de $0.272s$, obtido com o particionamento estático de *scanlines* usando 14 processadores com uma eficiência de apenas 41%. Os particionamentos dinâmicos mostraram uma queda de eficiência muito rápida, indicando que o tempo gasto em comunicação é excessivo, não compensando o seu uso neste conjunto de dados.

Para os dados "sym64", o tempo máximo foi obtido no particionamento estático por blocos com 3 processadores em $0.073s$, com uma eficiência de 50%.

As acelerações obtidas neste caso são de $2,5$ e $1,4$ para o "brain" e o "sym64", respectivamente. As acelerações obtidas dos tempos ideais são $\frac{0.68}{0.25} = 2.72$ e $\frac{0.104}{0.037} = 2.8$. As obervações sobre a relação de tempo ideal e aceleração citadas anteriormente se repetem.

A Figura 5.20 confirma a baixa eficiência dos dados pequenos usando-se muitos processadores.

Capítulo 6

Conclusões

A compreensão dos dados volumétricos por meio de imagens requer experimentação. Os algoritmos existentes ainda são muito lentos para prover interatividade nas estações de trabalho atuais; por outro lado, os computadores paralelos com vários processadores ainda são muito caros para estarem disponíveis para uso geral.

Como resultado final do trabalho, obteve-se um sistema interativo de visualização volumétrica utilizando processamento distribuído em redes locais convencionais. Através da utilização deste sistema nos exemplos apresentados, procurou-se avaliar a viabilidade de se utilizar redes heterogêneas e não dedicadas para acelerar o algoritmo de *ray-casting*. Ainda como contribuição deste trabalho, foram propostas e avaliadas técnicas de particionamento e indicadores de desempenho.

Com estes exemplos, mostrou-se que nos ambientes de estações de trabalho conectadas via redes locais, presentes na maioria das instuições de pesquisa e universidades, é possível utilizar os tempos disponíveis e não utilizados de outros computadores e obter acréscimos significativos de desempenho.

As acelerações de aproximadamente 4.5 obtidas para os grandes conjuntos de dados, reduziram significativamente o tempo de cálculo e exibição. Espera-se que, com as melhorias nas estações de trabalho e na velocidade da rede, brevemente seja possível obter um tempo que permita a animação.

A pequena variância dos tempos obtidos mostrou que, apesar da rede apresentar uma utilização bastante variada, o seu efeito na execução do programa distribuído não é maior do que ocorre nos aplicativos comumente utilizados nestes ambientes.

Os indicadores de desempenho propostos, tempo ideal e eficiência, se adequaram bastante bem à heterogenidade da rede e forneceram indicações equivalentes às obtidas em redes homogêneas.

Um ponto importante a ser destacado é que o tempo seqüencial utilizado no cálculo da aceleração no ambiente heterogêneo pressupõe a disponibilidade do processador mais rápido, para que este fique dedicado ao processamento. Assim, nestes ambientes, acelerações não muito altas também podem significar ganhos importantes para o usuário.

Nos dados grandes, as técnicas de particionamento estático, apesar de apresentarem acelerações semelhantes às técnicas de particionamento dinâmico, são mais sensíveis ao

processamento local dos processadores utilizados. Sugere-se, portanto, o uso dos particionamentos dinâmicos, principalmente o por blocos.

Nos dados pequenos, a melhor indicação também é o uso do particionamento dinâmico por blocos com o uso de poucos processadores.

6.1 Trabalhos Futuros

Um tema importante para um trabalho futuro é a criação de uma técnica que permitisse uma visualização combinada de visualização volumétrica direta e iso-superfícies. Com isto poderíamos obter técnicas de segmentação e classificação para a identificação automática de regiões, como por exemplo a extração de iso-superfícies de órgãos e tumores em imagens médicas.

Ainda na área médica, a utilização de técnicas de visualização volumétrica, objetivando a reconstrução tri-dimensional, a partir de dados de ultrassonografia ainda é muito pouco explorada. A visualização tri-dimensional dos exames em várias etapas da gestação permitiria o acompanhamento do desenvolvimento do feto e a identificação de problemas de formação e crescimento.

A utilização de um particionamento estático baseada no tempo ideal, ou seja, basear o particionamento no balanceamento dos desempenhos dos processadores utilizados. Além disso, a aplicação das idéias propostas com o uso de computação distribuída para o algoritmo de *shear-warping*.

A operação aritmética matricial entre os volumes e a sua posterior visualização permitiria que fossem examinados resultados de exames de ressonância magnética funcionais, permitindo a reconstrução e visualização tri-dimensional das áreas ativadas pelos estímulos.

Além disso, a disponibilização efetiva de *hardwares* específicos para a visualização de volume permitirá a sua aplicação em outras áreas, como, por exemplo, no método de elementos finitos, visualizando-se iso-volumes em malhas não estruturadas.

De maneira geral, existe uma ampla bibliografia sobre Visualização Volumétrica no que se refere a algoritmos específicos, aplicações científicas, projetos comerciais e tópicos avançados, e uma falta de informação quando se trata de *hardwares* especiais, métodos para manipulação de dados não-cartesianos, otimização e aperfeiçoamento de algoritmos, interface com o usuário, animação (apenas citada como um processo crítico da visualização científica) e, principalmente, implementações de uso geral ou de domínio público.

Referências Bibliográficas

[1] D. Badouel, K. Bouatouch, T. Priol. Distributing Data and Control for Ray Tracing in Parallel. *IEEE Computer Graphics and Applications* **4** (14), pp. 69–77, 1994.

[2] D. P. Bertsekas, J. N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall International Editions, 1989.

[3] J. A. Challinger. Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids. Ph.D. Thesis, *University of California at Santa Cruz*, 1993.

[4] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, B. C. Teeter. Two Algorithms for the Three-Dimensional Reconstruction of Tomograms. *Medical Physics* **15** (3), pp. 320–327, 1988.

[5] D. Cohen. Voxel Traversal along a 3D Line. *Graphics Gems IV*, Academic Press, pp. 366–369, 1994.

[6] J. Dongarra, P. Messina, D. C. Sorensen, R. G. Voigt. Parallel Processing for Scientific Computing. *Proceedings of the Fourth SIAM Conference*, December, 1989.

[7] R. A. Drebin, L. Carpenter, P. Hanrahan. Volume Rendering. *Proceedings of SIGGRAPH'88*, *Computer Graphics* **22** (4), pp. 65–74, 1988.

[8] T. Elvins. A Survey of Algorithms for Volume Visualization. Course Notes 1, *SIGGRAPH'92*, pp. 3.1–3.14, 1992.

[9] J. Foley, A. van Dam, S. Feiner, J. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[10] H. Fuchs, Z. M. Kedem, S. P. Uselton. Optimal Surface Reconstruction for Planar Contours. *Communications of the ACM* **20** (10), pp. 693–702, 1977.

[11] R. Gallagher, J. Nagtegall. An Efficient 3D Visualization Technique for Finite Element Models and Other Coarse Volumes. *Computer Graphics* **23** (3), pp. 185–194, 1989.

[12] R. S. Gallagher. *Computer Visualization.* CRC Press, 1995.

[13] A. Geist, V. S. Sunderam. PVM 3: User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1994.

[14] C. Giertsen, J. Petersen. Parallel Volume Rendering on a Network of Workstations. *IEEE Computer Graphics and Applications*, pp. 16–23, 1993.

[15] R. C. Gonzalez, P. Wintz. Digital Image Processing. Addison-Wesley, 1987.

[16] G. T. Herman, H. K. Liu. Three-Dimensional Display of Human Organs from Computer Tomograms. *Computer Graphics and Image Processing* **9** (1), pp. 1–21, 1979.

[17] B. Hibbard, P. J. Moran, M. L. Norman. Distributed Scientific Visualization on High-Performance Networks. Course Notes 7, *SIGGRAPH'92*, 1992.

[18] S. Hollasch. Progressive Image Refinement vis Gridded Sampling. Graphics Gems III, *Academic Press*, pp. 352–361, 1992.

[19] A. E. Kaufman, W. Lorensen, R. Yagel. Volume Visualization: Algorithms and Applications. Course Notes 1, *Volume Visualization Symposium'94*, 1994.

[20] P. G. Lacroute, M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Proceedings of SIGGRAPH'94*, pp. 451–458, 1994.

[21] P. G. Lacroute. VolPackUser's Guide. *Stanford University (Computer Graphic Lab)*, http://www-graphics.stanford.edu:/software/volpack, 1994.

[22] D. Laur, P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Proceedings of SIGGRAPH'91*, *Computer Graphics* **25** (4), pp. 258–288, 1991.

[23] S. T. Leutenegger, Xian-He Sun. Distributed Computing Feasibility in a Non-Dedicated Homogeneos Distributed System. Technical Report ICASE/93-65, Institute for Computer Applications in Science and Engineering, 1993.

[24] M. Levoy. Display of Surface from Volume Data. *IEEE Computer Graphics and Applications* **8** (3), pp. 29–37, 1988.

[25] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transaction on Graphics* **9**, pp. 29–37, 1988.

[26] M. Levoy. A Taxonomy of Volume Visialization Algorithms. Course Notes 11, *SIGGRAPH'90*, pp. 6–12, 1990.

[27] W. E. Lorensen, H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics* **21** (4), pp. 163–169, 1987.

[28] K. Ma, J. S. Painter, C. D. Hansen, M. F. Krogh. Parallel Volume Rendering Using Binary Swap Compositing. *IEEE Computer Graphics and Applications* **4** (14), pp. 59–68, 1994.

[29] B. H. McCormick, T. A. Defanti, M. Brown. Visualization in Scientific Computing. *Computer Graphics* **21** (6), 1987.

[30] T. R. Nelson, T. T. Elvins. Visualization of Ultrasound Data. *IEEE Computer Graphics and Applications* **13** (6), pp. 50–57, 1993.

[31] U. Neumann. Communication Costs for Parallel Volume Rendering Algorithms. *IEEE Computer Graphics and Applications* **4**, pp. 49–58, 1994.

[32] J. Nieh, M. Levoy Volume Rendering on Scalable Shared-Memory MIMD Architectures. *Proceedings of the 1992 Workshop on Volume Visualization*, ACM Press, pp. 17–24, 1992.

[33] G. M. Nielson, B. Shriver, L. J. Rosenblum. *Visualization in Scientific Computing.* IEEE Computer Society Press.

[34] G. M. Nielson. Modeling and Visualization Volumetric and Surface-on-Surface Data. Course Notes 4, *SIGGRAPH'94*, pp. 31–97, 1994.

[35] H. Pfister, A. Kaufman, T. Chiueh. Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization. *Proceedings of the 1994 Symposium on Volume Visualization*, ACM SIGGRAPH, pp. 75–82, 1994.

[36] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Proceedings of SIGGRAPH'88*, *Computer Graphics* **22** (4), pp. 160–165, 1988.

[37] R. de B. Seixas, M. Gattass, L. H. de Figueiredo, L. F. Martha. Otimização do Algoritmo de Ray-Casting para Visualização de Tomografias. Caderno de Comunicações do *VII SIBGRAPI*, pp. 5–8, 1994.

[38] R. de B. Seixas, M. Gattass, L. H. de Figueiredo, L. F. Martha. Visualização Volumétrica com Otimizações de Ray-Casting e Detecção de Bordas. Anais do *VIII SIBGRAPI*, pp. 281–286, 1995.

[39] R. de B. Seixas, M. Gattass. Visualização Volumétrica em um Ambiente de Computação Distribuída. Anais do *IX SIBGRAPI*, pp. 15–22, 1996.

[40] P. Shirley, A. Tuckman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics* **24** (5), pp. 63–70, 1990.

[41] C. T. Silva, A. E. Kaufman. Parallel Performance Measures for Volume Ray Casting. Reports of *State University of New York at Stony Brook*, 1994.

[42] C. T. Silva, F. M. Lok, A. E. Kaufman. Interactive Parallel Volume Rendering Using PVR System. Technical Report of *State University of New York at Stony Brook*, 1995.

[43] L. Sobierajski, A. E. Kaufman. Volumetric Ray Tracing. *Proceedings of the 1994 Symposium on Volume Visualization*, pp. 11–18, 1994.

[44] K. Sung, J. L. J. Shiuan, L. Ananda. Ray Tracing in a Distributed Environment. *Computer Graphics* **20** (1), pp. 41–49, 1996.

[45] C. Upson, M. Keeler. V-Buffer: Visible Volume Rendering. *Proceedings of SIGGRAPH'88*, *Computer Graphics* **22** (4), pp. 59–64, 1988.

[46] A. Watt, M. Watt. *Advanced Animation and Rendering Techniques.* Addison-Wesley, 1992.

[47] L. Westover. Footprint Evaluation for Volume Rendering. *Computer Graphics* **24** (4), pp. 367–376, 1990.

[48] S. Whitman, C. D. Hansen, T. W. Crockett. Recent Development in Parallel Rendering. *IEEE Computer Graphics and Applications* **14**, pp. 21–22, 1994.

[49] R. Yagel, A. Kaufman. Template-Based Volume Viewing. *Proceedings of Eurographics'92*, pp. 153–167, 1992.

[50] R. Yagel, R. Machiraju. Data-Parallel Volume Rendering Algorithms. Reports of *The Ohio State University*, 1994.

[51] M. K. Zuffo, E. T. Santos, R. de D. Lopes, C. A. P. S. Martins. Visualização de Alto-Desempenho. Anais do *Workshop* sobre Computação de Alto-Desempenho e Processamento de Sinais, pp. 142–161, 1993.

[52] M. K. Zuffo, R. Lopes. A High-Performance Direct Volume Rendering Pipeline. Anais do *VII SIBGRAPI*, pp. 241–248, 1994.

Visualização Volumétrica com Ray-Casting num Ambiente Distribuído.

Tese de Doutorado apresentada por **Roberto de Beauclair Seixas** em 09 de Abril de 1997 ao Departamento de Informática da PUC-Rio e aprovada pela Comissão Julgadora, formada pelos seguintes professores:

Prof. Marcelo Gattass – Orientador
Departamento de Informática – PUC-Rio

Prof. Roberto de Alencar Lotufo
UNICAMP

Prof. Luiz Henrique de Figueiredo
LNCC

Profa. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. Marcelo Dreux
Departamento de Mecânica – PUC-Rio

Prof. Paulo Cezar Carvalho
IMPA

Prof. Luiz Fernando Campos Ramos Martha
Departamento de Engenharia Civil – PUC-Rio

Visto e permitida a impressão.
Rio de Janeiro, de de 1997.

Coordenador dos Programas de Pós-Graduação e
Pesquisa do Centro Técnico Científico