5
Resultados Experimentais

Neste capítulo são apresentados os resultados dos experimentos elaborados para validar a linha de produção gráfica distribuída e os algoritmos propostos para melhorar o desempenho das arquiteturas apresentadas. Estes foram testados em um agrupamento composto por 10 PCs, cada qual equipado com um processador Intel Pentium 4 1.8 GHz e uma placa gráfica NVIDIA Geforce 4 Ti 4200 com 128 MB de memória, utilizando o sistema operacional Linux. Os 10 PCs foram conectados por uma rede Ethernet de 1 Gigabits por segundo.

5.1
Linha de Produção Gráfica Distribuída

O sistema de renderização distribuída proposto atendeu plenamente aos objetivos traçados, servindo como bom ponto de partida para a implementação de três arquiteturas para renderização distribuída. Isso ajuda a validar as escolhas feitas no projeto da arquitetura do sistema.

Foi realizado um experimento para medir o uso dos recursos gráficos do agrupamento de PCs e a latência de resposta ao usuário no uso do sistema proposto. Foi elaborada uma aplicação hipotética, onde é simulada a renderização de uma plataforma de petróleo no agrupamento de PCs. Ao invés de renderizar a plataforma, cada nó escravo desenha um pedaço de uma imagem pré-renderizada da plataforma, a qual foi dividida igualmente entre os nós escravos. É simulada uma aplicação perfeitamente balanceada: todos os nós levam um tempo pré-fixado por quadro para desenhar a sua parte da imagem. Idealmente, como o sistema é uma linha de produção, a aplicação como um todo deve ser capaz de renderizar com uma taxa igual à taxa do estágio mais lento. Sabemos no entanto que isso não é possível devido à complexidade de uma arquitetura distribuída de renderização. Neste experimento foram utilizados 7 PCs, onde 1 foi eleito o nó mestre e 6 são nós escravos. Foram testadas resoluções de 800x600 e 300x300 *pixels*

Tempo Fixado para Renderização nos Servidores (ms)	Tempo Médio Entre Quadros Consecutivos da Aplicação (ms)	Latência de Resposta Média (ms)	Tempo Médio Rede (ms)	Tempo Médio Descompressão e Composição (ms)
50	59	143	22	23
33	43	110	23	23
25	34	94	21	23
20	29	84	19	23

Tabela 5.1: Aplicação de teste de recursos utilizando resolução 800x600 e compressão de *framebuffers*.

Tempo Fixado para Renderização nos Servidores (ms)	Tempo Médio Entre Quadros Consecutivos da Aplicação (ms)	Latência de Resposta Média (ms)	Tempo Médio Rede (ms)	Tempo Médio Descompressão e Composição (ms)
50	51	129	26	4
33	35	96	26	4
25	28	82	26	4
20	27	82	26	4

Tabela 5.2: Aplicação de teste de recursos utilizando resolução 800x600 e não utilizando compressão de *framebuffers*.

e tempos de renderização fixados em 50, 33, 25 e 20 milisegundos por quadro. (vide Figura 5.1).

Os resultados encontram-se nas Tabelas 5.1, 5.2, 5.3 e 5.4.

Pela observação dos resultados, podem ser tiradas algumas conclusões. Primeiramente, pelas quatro tabelas é possível concluir que a latência da aplicação se manteve sob controle, tendo o valor esperado de entre 2 a 3 quadros de latência. Como a latência de uma aplicação com *buffer* duplo é de 2 quadros, esse valor está dentro do aceitável para aplicações interativas.

O algoritmo de compressão utilizado não ajudou muito a reduzir o uso da rede e aumentou a latência de resposta ao usuário da aplicação, o que é observável nas quatro tabelas. Isso é decorrente do uso de um algoritmo de compressão de dados arbitrários para a compressão de imagens sem perda, o que resulta em taxas de compressão muito baixas (em torno de 50%). Essas taxas melhoram em aplicações que utilizam uma cor simples como fundo, o que é o caso de aplicações de visualização científica.

A taxa de renderização efetiva da aplicação se manteve bastante próxima da taxa de renderização dos nós escravos. Apenas nos casos em que o gargalo passou a ser a comunicação de dados pela rede isso não foi

Tempo Fixado para Renderização nos Servidores (ms)	Tempo Médio Entre Quadros Consecutivos da Aplicação (ms)	Latência de Resposta Média (ms)	Tempo Médio Rede (ms)	Tempo Médio Descompressão e Composição (ms)
50	53	115	5	6
33	37	81	5	6
25	28	64	5	6
20	23	55	5	6

Tabela 5.3: Aplicação de teste de recursos utilizando resolução 300x300 e compressão de *framebuffers*.

Tempo Fixado para Renderização nos Servidores (ms)	Tempo Médio Entre Quadros Consecutivos da Aplicação (ms)	Latência de Resposta Média (ms)	Tempo Médio Rede (ms)	Tempo Médio Descompressão e Composição (ms)
50	51	119	5	1
33	35	78	5	1
25	26	57	6	1
20	21	47	5	1

Tabela 5.4: Aplicação de teste de recursos utilizando resolução 300x300 e não utilizando compressão de *framebuffers*.

observado. Na Tabela 5.2 é possível notar que o uso da rede se torna o gargalo da aplicação quando a taxa de renderização nos servidores é de 25 e 20 milisegundos por quadro. Na Tabela 5.4, em que o tempo gasto na rede é muito pequeno, a taxa de renderização da aplicação é bem próxima da taxa de renderização nos servidores. Isso é uma indicação do bom uso dos recursos gráficos dos nós do agrupamento.

Além disso, o sistema foi pensado como uma linha de produção, em que cada estágio opera independente do outro. No entanto, em computadores com um único processador, a renderização e a compressão competem pelos ciclos da mesma CPU, tornando estas duas operações não paralelizáveis.

Em trabalhos futuros poderá ser investigado o uso de algoritmos de compressão melhores que utilizem, por exemplo, a coerência temporal que existe no processo de renderização, o que é amplamente explorado em algoritmos de compressão de vídeo. Além disso, esse sistema pode ser testado em ambientes multiprocessados, o que não é incomum em agrupamentos de PCs.

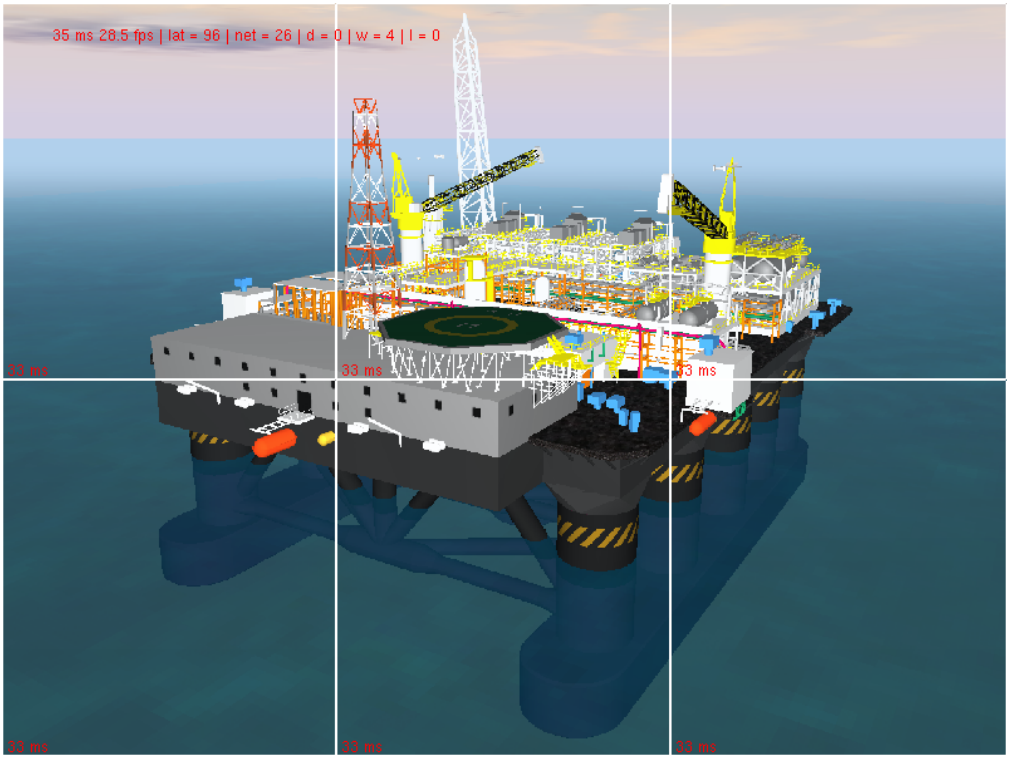

Figura 5.1: Foto de tela do programa de teste do uso de recursos gráficos do agrupamento de PCs.

5.2 Arquitetura com Ordenação no Início

O sistema com ordenação no início e os algoritmos propostos para essa arquitetura descritos no Capítulo 4 foram testados utilizando 10 PCs do agrupamento descrito acima, dentre os quais 1 foi eleito o nó mestre e 9 são nós escravos.

Foram desenvolvidas duas aplicações, mostradas nas Figuras 5.2 e 5.3:

- *Platforms*: uma aplicação com gargalo na geometria, composta por seis plataformas de petróleo. A cena inteira possui 1.884.844 triângulos distribuídos entre 11.811 objetos;
- *VolRender*: uma aplicação com gargalo na rasterização, composta por um dado volumétrico que representa um motor. O dado está armazenado em uma textura 3D de dimensões 256x256x110, visualizada por 379 fatias quadrangulares perpendiculares ao observador.

O modelo de *Platforms* foi visualizado utilizando-se um caminho pré-determinado, no qual o observador navega pela cena, aproximando-se de uma plataforma entre $t = 60s$ e $t = 75s$. Já ao modelo de *VolRender* foi

aplicado um conjunto de rotações e translações, fazendo com que ele seja visualizado a partir de diferentes pontos de vista.

Na tentativa de não limitar o desempenho da aplicação por causa do tráfego na rede, foi utilizada uma resolução de 600x600 *pixels* para *Platforms* e 800x600 *pixels* para *VolRender*. A compressão de *framebuffers* foi desligada nesses testes.

Figura 5.2: Aplicação *Platforms*, composta por milhões de polígonos.

5.2.1 Algoritmo de Balanceamento de Carga

Para testar o algoritmo de balanceamento proposto, foi medido o desbalanceamento de carga obtido na visualização de ambos os modelos. Assim como Mueller [11], medimos o desbalanceamento de carga como a razão entre a carga no processador mais lento e a carga média entre os processadores. Mueller [11] considerou razoável um desbalanceamento de carga menor que 1,5. Conforme mostrado na Figura 5.4, o algoritmo proposto obteve uma razão menor que 1,5 para ambos os modelos em todo o caminho, com um valor médio inferior a 1,2.

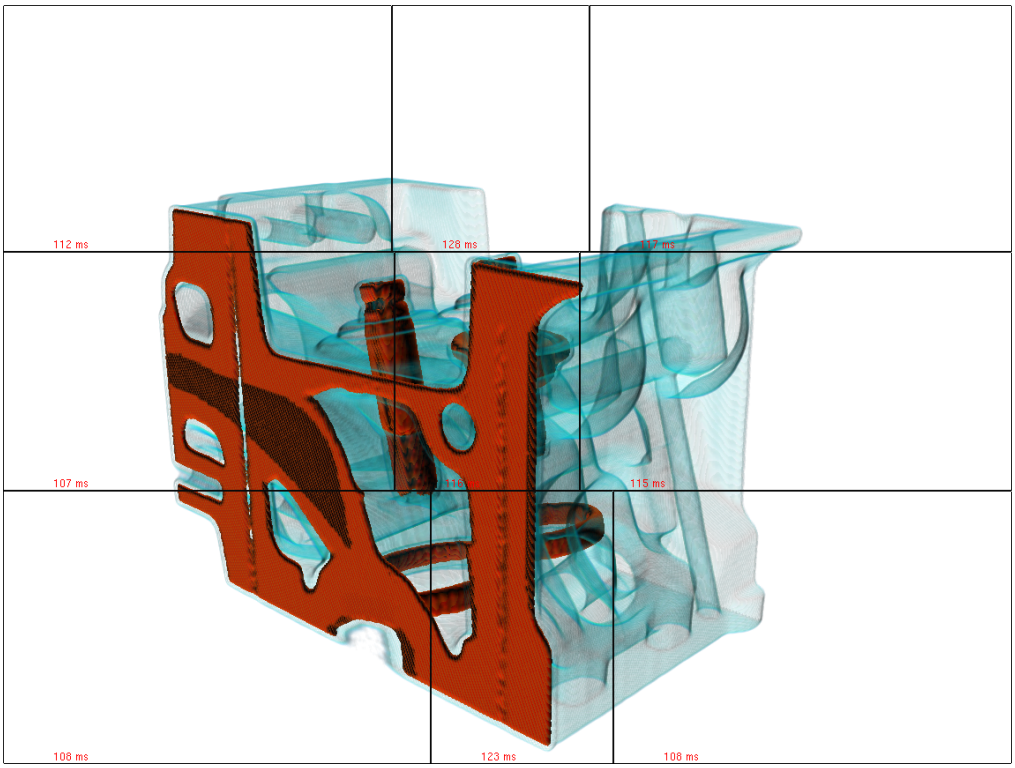

Figura 5.3: Aplicação *VolRender*, que renderiza um dado volumétrico de dimensões 256x256x110.

Posteriormente foi testada a escalabilidade do sistema. A escalabilidade da arquitetura com ordenação no início é limitada principalmente pelo número de primitivas que devem ser processadas de modo redundante em mais de um ladrilho [13, 22, 11, 15, 14]. Na análise da escalabilidade do sistema implementado, é medido para a aplicação com gargalo na geometria (*Platforms*) o valor médio do número de interseções que cada primitiva tem com todos os ladrilhos para diferentes números de ladrilhos: 4, 16, 36 e 64 (é considerado que uma primitiva que está em apenas 1 ladrilho intersecta 1 ladrilho). A Figura 5.5 mostra que esse valor se mantém menor que 2 em todo o caminho, até mesmo na configuração com 64 ladrilhos, exceto no intervalo onde é feita a aproximação a uma plataforma.

O algoritmo de balanceamento proposto reduziu satisfatoriamente a diferença entre os tempos de renderização nos nós de renderização, melhorando o desempenho da aplicação como um todo. Obviamente, por se tratar de uma heurística, o algoritmo não é capaz de calcular partições perfeitamente balanceadas. Além disso, o algoritmo apenas tenta igualar o esforço entre os nós, não levando em conta a possibilidade de minimização do esforço total para renderizar a cena.

Os resultados apresentados comprovam que o algoritmo se comporta

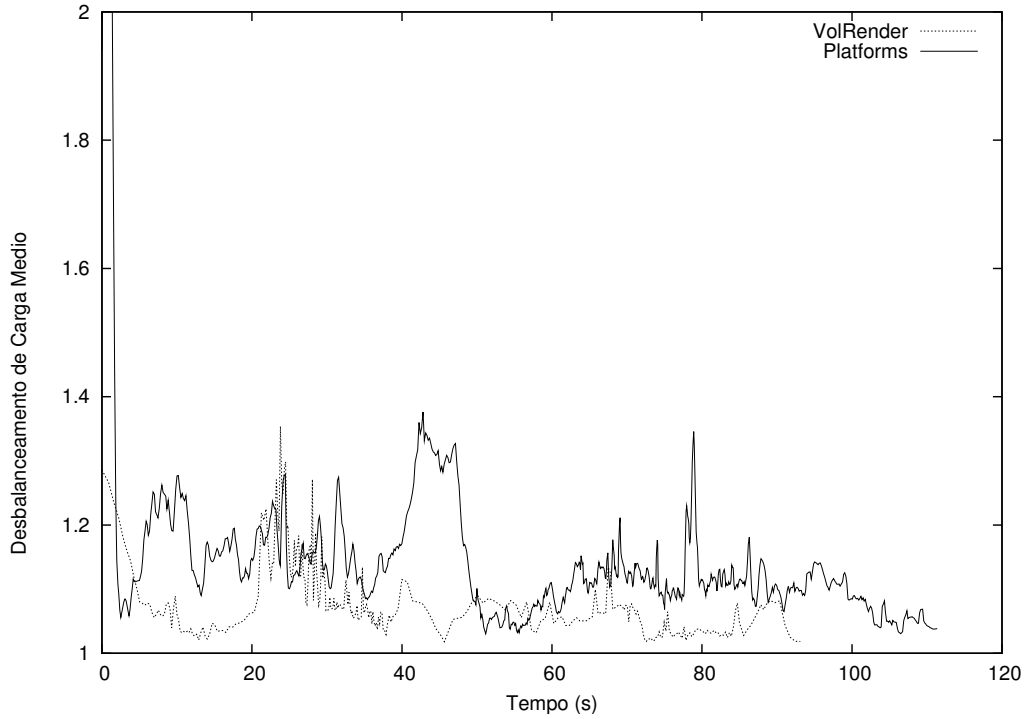

Figura 5.4: Desbalanceamento de carga em *Platforms* e *VolRender*.

bem, obtendo resultados satisfatórios em aplicações com gargalo tanto na geometria quanto na rasterização.

5.2.2 Estratégia de Distribuição de Ladrilhos

Para ambas as aplicações, foi feita uma comparação entre o desempenho da aplicação que combina a estratégia de distribuição de ladrilhos com o algoritmo de balanceamento de carga, o desempenho da aplicação que apenas utiliza o algoritmo de balanceamento de carga e o desempenho de uma aplicação local. As Figuras 5.6 e 5.7 mostram a comparação para *Platforms* e *VolRender*, respectivamente. É possível notar uma redução e suavização dos tempos de renderização com o uso da estratégia de distribuição de ladrilhos.

Foi também medida a latência adicional introduzida pelo uso da estratégia de distribuição de ladrilhos. A comparação encontra-se nas Figuras 5.8 e 5.9. Para ambas as aplicações, a latência adicional foi bastante pequena, com valores médios de 2.6 ms para *Platforms* e 4.8 ms para *VolRender* e desvios padrão de 2.4 ms e 7.6 ms, respectivamente.

Conclui-se que a estratégia de distribuição de ladrilhos aumentou o uso efetivo do poder de processamento gráfico das GPUs do sistema,

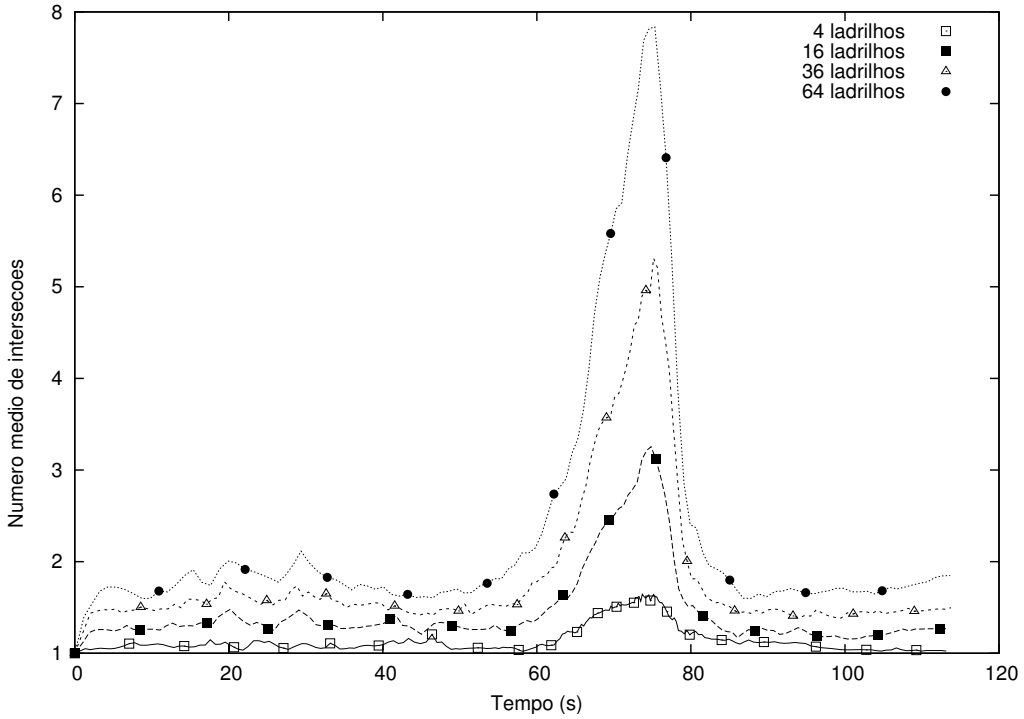

Figura 5.5: Número médio de interseções de primitivas com os ladrilhos em *Platforms*.

mantendo a latência de resposta ao usuário limitada e bastante parecida com a latência do sistema que não usa tal estratégia. Além disso, foi observada a suavização esperada nas taxas de renderização, a qual é desejável em aplicações interativas.

5.3 Arquitetura com Ordenação no Fim para Renderização Volumétrica

O sistema de renderização volumétrica distribuída com ordenação no fim foi testado utilizando 9 PCs do agrupamento descrito acima, dentre os quais 1 foi eleito o nó mestre e 8 são nós escravos.

Neste teste, são comparadas as arquiteturas com ordenação no início e com ordenação no fim empregadas na mesma aplicação (*VolRender*). Foi utilizada uma resolução de 700x700 *pixels* e foi desligada a compressão de *framebuffers*.

Foram utilizados dois modelos:

– *Engine*: o mesmo motor visualizado nos testes dos algoritmos para a arquitetura com ordenação no início: o dado está armazenado em uma textura 3D de dimensões 256x256x110, visualizada por 379

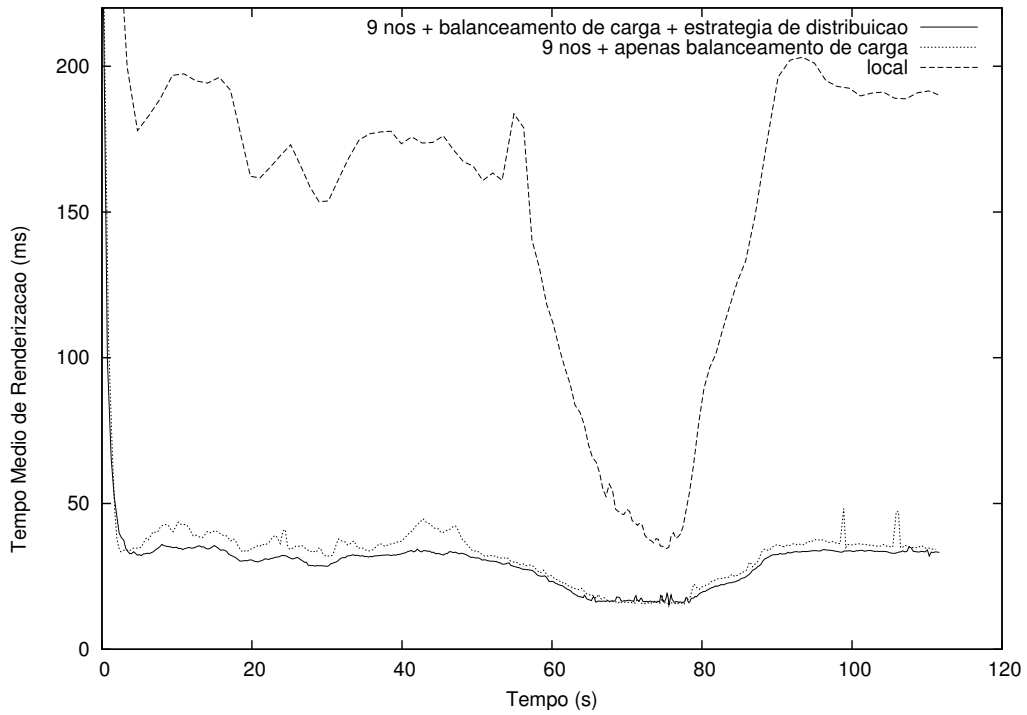

Figura 5.6: Tempo médio de renderização em *Platforms*.

fatias quadrangulares perpendiculares ao observador. Este modelo cabe inteiro na memória de textura das placas gráficas utilizadas;

– *Domo512*: um modelo sintético, cujo dado está armazenado em uma textura 3D de dimensões 512x512x512, visualizada por 886 fatias quadrangulares perpendiculares ao observador. Este modelo não cabe na memória de textura das placas gráficas utilizadas.

Na aplicação que utiliza a arquitetura com ordenação no início, o modelo inteiro é visualizado em cada nó de renderização, enquanto que na arquitetura com ordenação no fim cada modelo foi dividido em 8 sub-volumes iguais, de dimensões 128x128x55 e 256x256x256 para os modelos *Engine* e *Domo512*, respectivamente. Estes sub-volumes cabem inteiros na memória da placa gráfica utilizada.

Os modelos foram visualizados utilizando-se um caminho pré-determinado, no qual foi aplicado aos modelos um conjunto de rotações e translações, finalizando com uma aproximação a um dos cantos do modelo no trecho entre $t = 35$ e $t = 45s$.

A comparação entre as arquiteturas com ordenação no início e com ordenação no fim para o modelo *Engine* encontra-se na Figura 5.10. Pode-se notar que a arquitetura com ordenação no início tem um desempenho ligeiramente melhor, o que só é possível porque o modelo cabe inteiro na

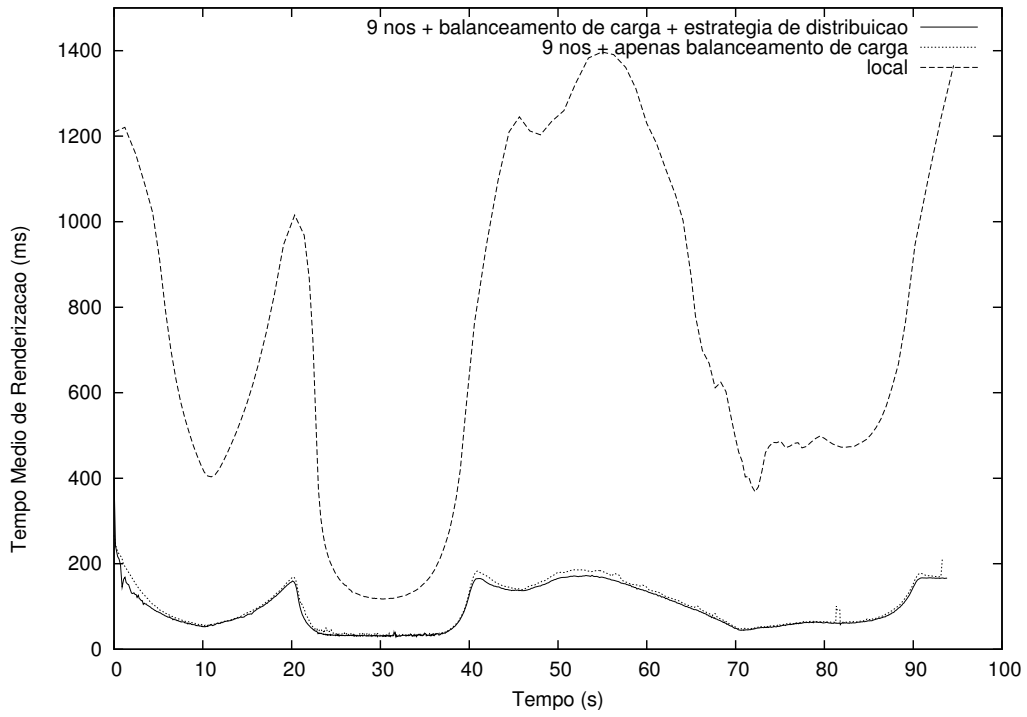

Figura 5.7: Tempo médio de renderização em *VolRender*.

memória de uma placa gráfica. A vantagem da arquitetura com ordenação no início se explica pelo seu menor desbalanceamento de carga, conforme ilustrado na Figura 5.11. A arquitetura com ordenação no fim possui um desbalanceamento maior no trecho do caminho da câmera em que há uma aproximação a um dos cantos do modelo, pois neste trecho apenas um sub-volume é visível.

Já em relação ao modelo *Domo512*, que não cabe inteiro na memória de textura de uma placa gráfica, a visualização interativa do modelo não foi possível na arquitetura com ordenação no início, pois ela requer que o modelo inteiro seja visualizado. O desempenho com essa arquitetura foi péssimo, tendo cada quadro levado mais de 20 segundos para ser renderizado. Na arquitetura com ordenação no fim foi possível uma visualização interativa, pois os sub-volumes cabem na memória de textura dos nós de renderização. Um gráfico com o desempenho dessa arquitetura para a renderização desse modelo encontra-se na Figura 5.12.

Esses resultados comprovam que a divisão do modelo entre os nós do agrupamento é uma opção que viabiliza a visualização interativa de dados volumétricos muito grandes.

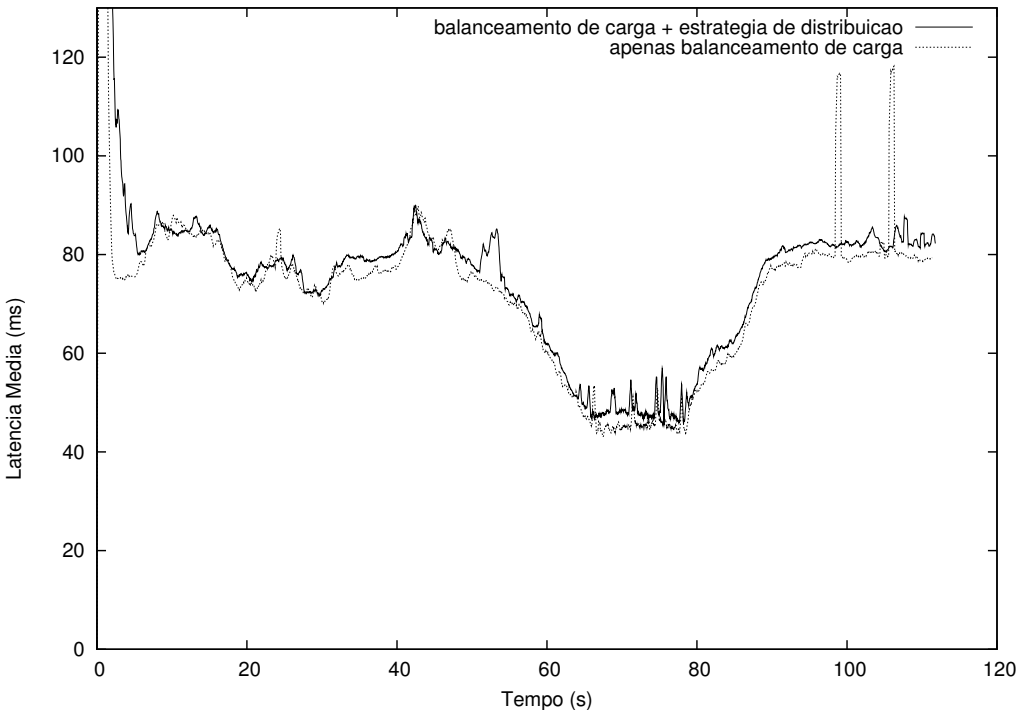

Figura 5.8: Latência média de resposta ao usuário em *Platforms*.

5.4 Arquitetura Híbrida com Ordenação no Início e no Fim

O sistema de arquitetura híbrida com ordenação no início e no fim descrito no Capítulo 4 foi testado utilizando 9 PCs do agrupamento descrito acima, dentre os quais 1 foi eleito o nó mestre e 8 são nós escravos.

Foi utilizada a aplicação *Platforms* com um modelo mais complexo, composto por 21 plataformas de petróleo, as quais possuem 5.730.106 triângulos distribuídos entre 41.463 objetos. O modelo foi visualizado utilizando-se um caminho pré-determinado, no qual o observador navega pela cena, aproximando-se de uma plataforma entre $t = 60s$ e $t = 75s$. Foi utilizada uma resolução de 600x600 *pixels* e foi desligada a compressão de *framebuffers*.

A arquitetura híbrida foi testada com uma faixa máxima de 50 *pixels* para a leitura do *Z-Buffer*.

Foram comparados o valor médio do número de interseções de primitivas com os ladrilhos e o tempo de renderização médio da aplicação durante o caminho da câmera para a arquitetura com ordenação no início e a arquitetura híbrida com uma faixa de 50 *pixels*. Os gráficos se encontram nas Figuras 5.13 e 5.14, respectivamente.

Pode-se observar no gráfico de interseções que, mesmo com o uso de

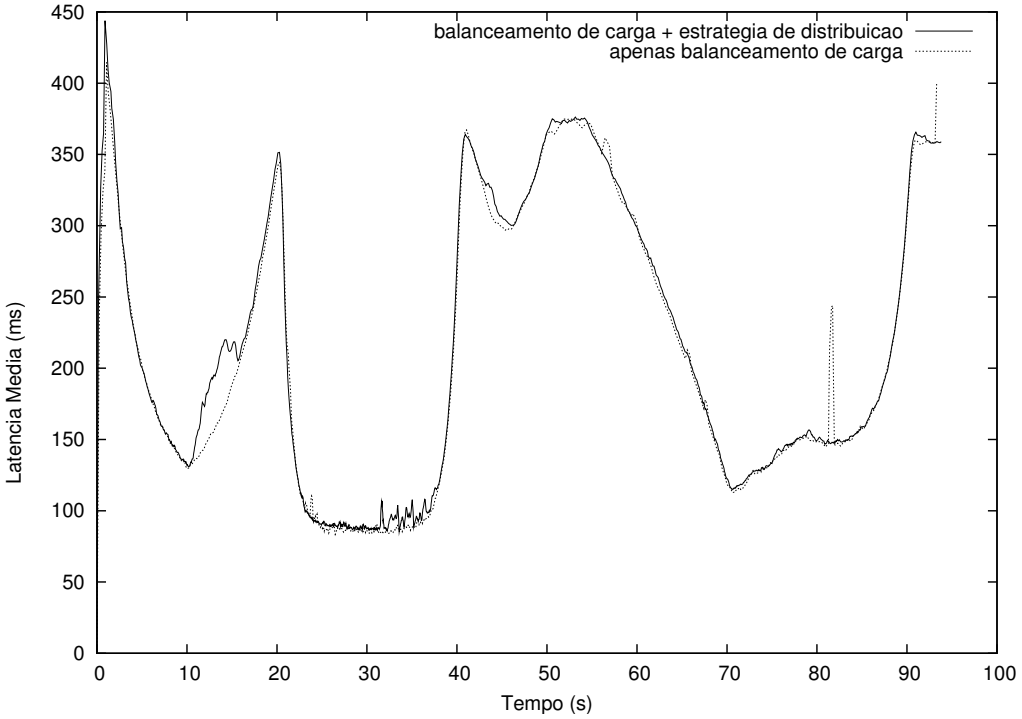

Figura 5.9: Latência média de resposta ao usuário em *VolRender*.

uma faixa fixa de 50 *pixels*, o fator de interseção de primitivas da estratégia híbrida se encontra bem próximo de 1, exceto no trecho da câmera em que há aproximação a uma plataforma. Isto demonstra que a estratégia híbrida proposta é válida. No entanto, a diminuição do número total de primitivas desenhadas não resultou num melhor desempenho da aplicação. As prováveis razões para isso são os atrasos causados pela leitura do *Z-Buffer* e escrita de *framebuffers* com informação de *Z-Buffer*, além do maior uso de largura de banda da rede, que se torna o gargalo da aplicação no trecho de aproximação à plataforma. O uso de faixas maiores ou menores para a leitura do *Z-Buffer* não trouxe melhorias ao desempenho da aplicação.

Espera-se que futuramente, com barramentos entre a CPU e a GPU mais poderosos, sejam possíveis leituras do *Z-Buffer* e escritas de *framebuffers* com informação de *Z-Buffer* mais eficientes, o que tornará essa arquitetura mais atraente.

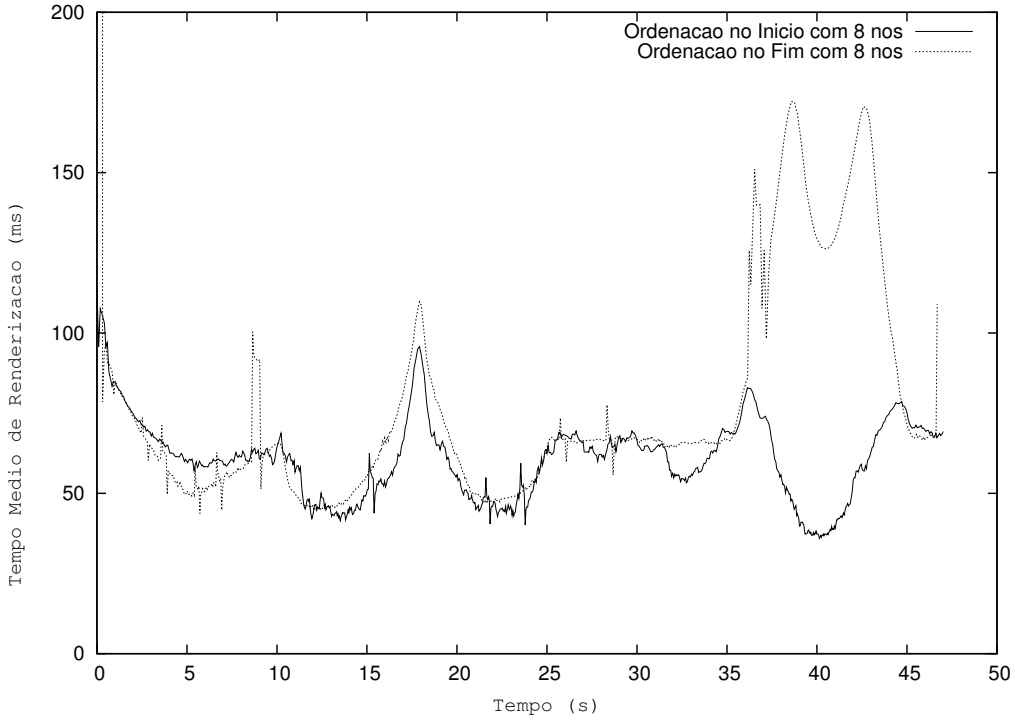

Figura 5.10: Tempo médio de renderização do modelo *Engine*.

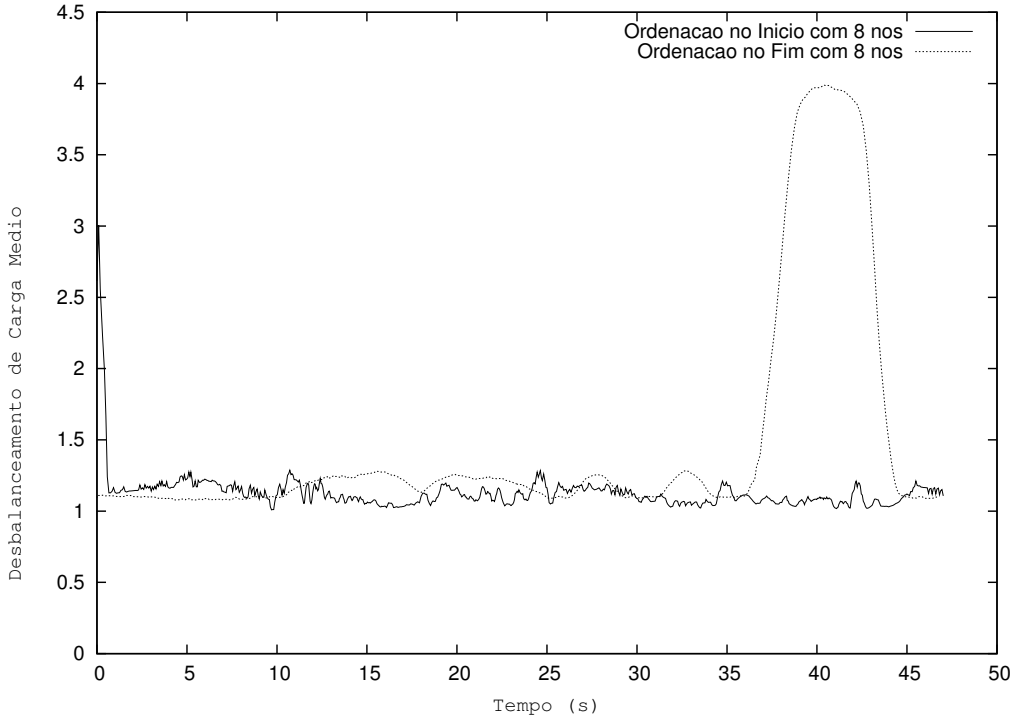

Figura 5.11: Desbalanceamento de carga médio na renderização do modelo *Engine*.

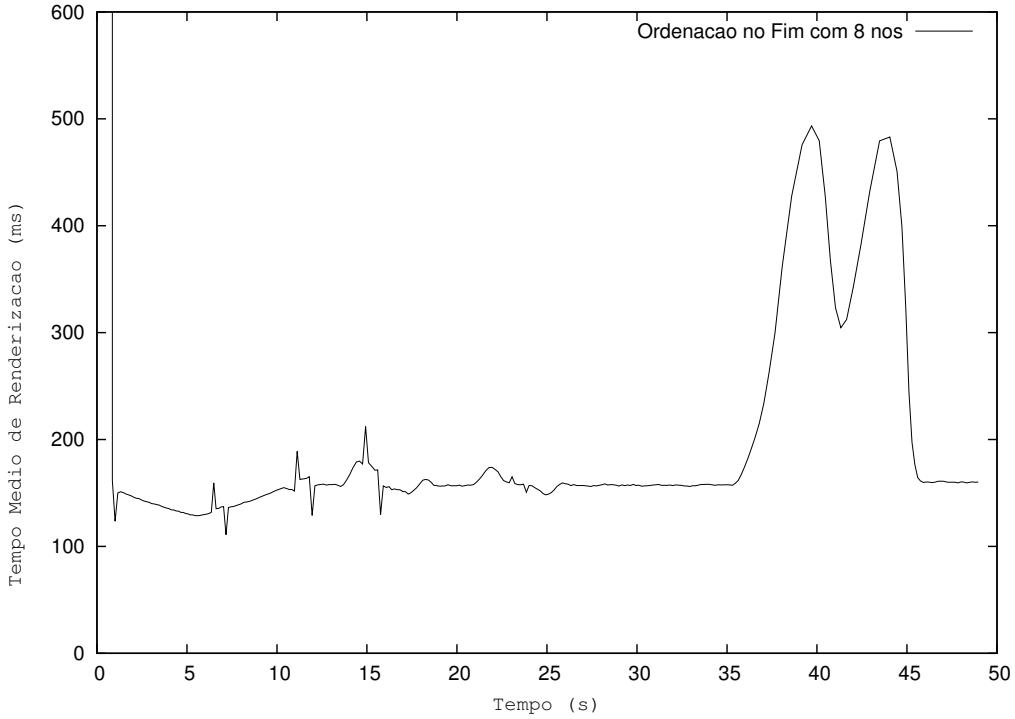

Figura 5.12: Tempo médio de renderização do modelo *Domo512*.

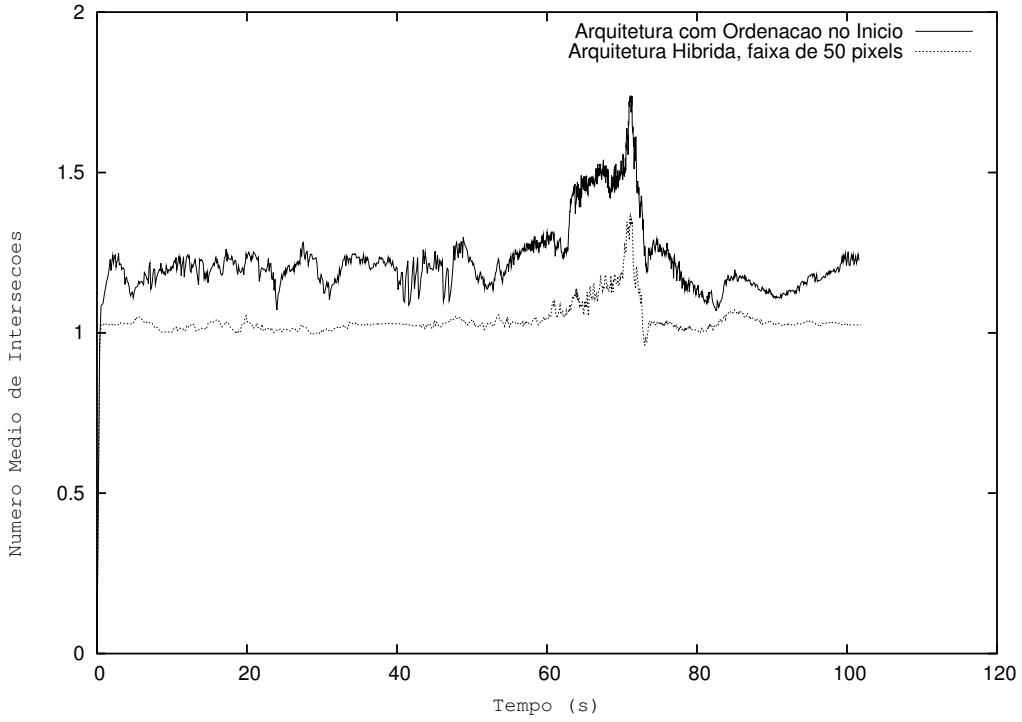

Figura 5.13: Número de interseções de primitivas em *Platforms*.

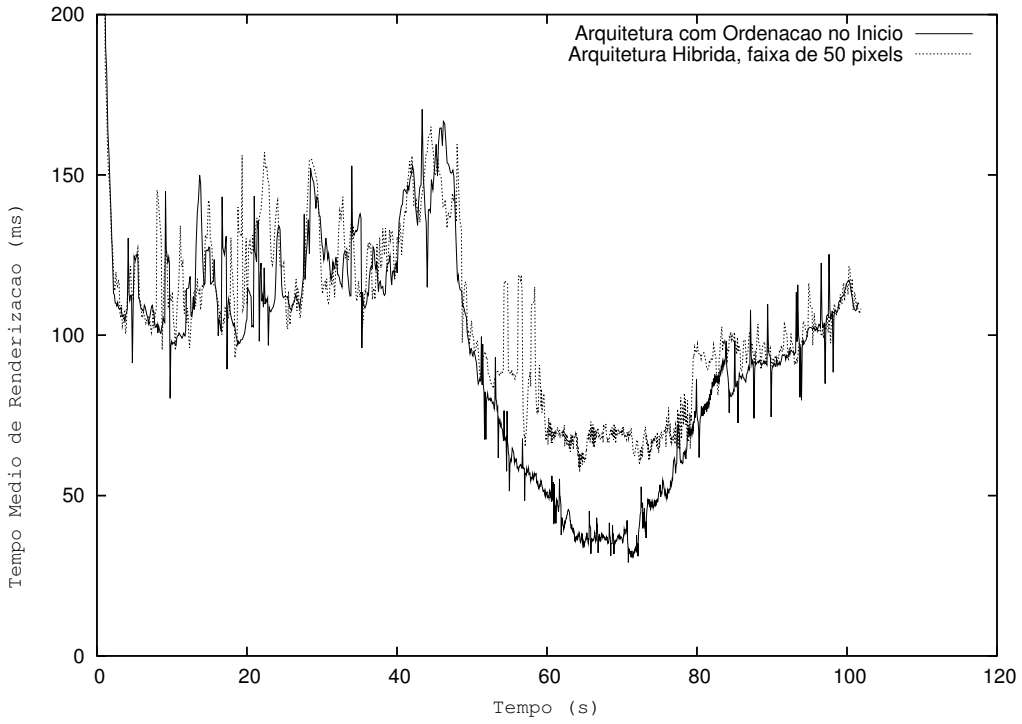

Figura 5.14: Tempo médio de renderização em *Platforms*.