UNIVERSIDADE FEDERAL FLUMINENSE

DANIEL LUIZ ALVES MADEIRA

G-MPP
MÉTODO PARA SIMULAÇÃO MASSIVA PARTÍCULA-PARTÍCULA DE N-CORPOS EM CLUSTER DE GPUS

NITERÓI

2015

UNIVERSIDADE FEDERAL FLUMINENSE

DANIEL LUIZ ALVES MADEIRA

G-MPP
MÉTODO PARA SIMULAÇÃO MASSIVA PARTÍCULA-PARTÍCULA DE N-CORPOS EM CLUSTER DE GPUS

Tese de doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Doutor. Área de concentração: Computação Visual.

Orientador:
ESTEBAN WALTER GONZALEZ CLUA

NITERÓI

2015

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

M181 Madeira, Daniel Luiz Alves
G-MPP : método para simulação massiva partícula-partícula de n-corpos em cluster de GPUS / Daniel Luiz Alves Madeira. – Niterói, RJ : [s.n.], 2015.
81 f.

Tese (Doutorado em Computação) - Universidade Federal Fluminense, 2015.
Orientador: Esteban Walter Gonzalez Clua.

1. Unidade de processamento gráfico. 2. Sistema de computação em grade. 3. Problema dos n-corpos. I. Título.

CDD 006.6

DANIEL LUIZ ALVES MADEIRA

G-MPP

MÉTODO PARA SIMULAÇÃO MASSIVA PARTÍCULA-PARTÍCULA DE N-CORPOS EM CLUSTER DE GPUS

Tese de doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Doutor. Área de concentração: Computação Visual.

BANCA EXAMINADORA

Prof. Esteban Walter Gonzalez Clua / IC-UFF
(Presidente)

Prof. Renato Antônio Celso Ferreira / DCC-UFMG

Prof. Ricardo Farias / PESQ/COPPE-UFRJ

Profa. Aura Conci / IC-UFF

Prof. Anselmo Antunes Montenegro / IC-UFF

Niterói

2015

"Mais do que criar o inédito, inovar é dar vitalidade ao antigo."

Mário Sérgio Cortella

À Deus, meus pais e irmãos, minha família, minha Aline, aos grandes amigos Leo, Elisa, João Gabriel, Sara, Vinícius, Douglas, Crys, Marco e Marcelo Caniato e ao professor Esteban. Vocês acreditaram em mim e me ajudaram a chegar até aqui.

Agradecimentos

À Deus, por me dar forças para perseguir meu objetivos.

Aos meus pais e meus irmãos, e à toda a minha família, por todo apoio, amizade, companheirismo e diversão durante todos esses anos.

À Aline, por ter mantido o seu sorriso, mesmo muitas vezes eu nem merecendo, e me ajudado a lembrar do meu. A nossa história só está começando.

Ao Leo e à Elisa, porque sem a amizade e os puxões de orelha de vocês, talvez meu caminho tivesse sido bem mais tortuoso do que já foi.

Aos grandes amigos Sara, Vinícius, Douglas, Crys, Marco Aurélio e Marcelo Caniato, e todos os outros que posso ter esquecido de citar, por todos os encontros, cuja diversão é certa, e por cada brinde!

Aos meus alunos da UFSJ, que me apoiaram muito nos meus pouco mais de 4 anos de casa. Demorei, mas consegui! (afinal o prazo é até às 23:59, né?)

Ao João Gabriel, por todos os bate-papos e aos "republicanos" Thales, Lucas, Tássio, João Paulo, pelos anos de república e pelo apoio em todas as idas à Niterói.

Ao Instituto de Computação da UFF e todos seus professores e funcionários, pela oportunidade de desenvolver meus estudos.

A todas as pessoas cujos nomes não estão aqui, mas que fizeram parte desta caminhada.

Resumo

O algoritmo de N-corpos é aplicado a uma série de problemas em Física Computacional, como gravitação, física molecular, simulação de multidão, entre outros, bem como em Computação Gráfica e Robótica. Os n corpos compõe um sistema e, enquanto o algoritmo itera, as propriedades de cada corpo são atualizados considerando interações entre o corpo em questão com todos os outros. Tal abordagem tem complexidade $O(n^2)$, uma vez que considera todas as interações de pares possíveis.

Várias abordagens foram propostas para diminuir a complexidade do problema a $O(N \log N)$. Embora estas abordagens tenham sido vitais para que as simulações fossem efetivamente utilizadas em produção nas décadas de 1980 e 1990, elas incluíram erros de aproximação também. Os erros foram deixados de lado, à medida que o tempo de execução diminuía. Com a popularização da computação paralela através de clusters e grids, esses métodos também foram implementadas nesses cenários, enquanto o método direto caiu em desuso devido a sua alta complexidade computacional. Mesmo que se paralelizado, o método direto ainda era muito mais lento, devido à falta de aceleradores atuais como GPUs. No entanto, com o atual desenvolvimento das GPUs, é possível revisitar o método e adaptá-lo às arquiteturas modernas, tornando viável para uso em produção.

Este estudo visa adaptar o método direto para clusters de GPUs. Usando várias GPUs em conjunto, o algoritmo pode ser optimizado de forma significativa, minimizando o tempo de execução. No entanto, com o uso destes clusters de GPUs, o custo extra tráfego de rede deve ser levado em conta. Portanto, é necessário um estudo detalhado sobre o algoritmo para uma implementação eficiente. Este estudo foi orientado por um problema sintético para simular o comportamento do algoritmo N-corpo em cluster, tornando possível observar o impacto como cada algoritmo passo colabora para o tempo de execução total. Após esta etapa, uma versão eficiente do algoritmo de N-corpos é apresentada e testada em um cluster.

Palavras-chave: Computação de alto desempenho; Computação em grade; Simulação Física; N-corpos; GPU

Abstract

The N-body algorithm is applied to a range of problems in Computational Physics, as gravitation, molecular physics, crowd simulation, among others, as well as in Computer Graphics and Robotics. The n bodies composes a system and, as the algorithm iterates, properties of each body are updated considering interactions between the body in question with all the others. Such an approach has $O(n^2)$ complexity, as it considers all pairwise interactions possible.

Several approaches has been used to decrease the problem complexity to $O(n\ logn)$. While these approaches were vital for n-body simulations to be effectively used in production in the 1980s and 1990s, they included approximation errors too. The errors were put aside, as the execution time decreased. With the popularization of parallel computing through clusters and grids, these methods have also been implemented in those scenarios, while the direct method has been left out due to their high computational complexity. Even if it were parallelized, the direct method was still much slower, due to lack of current accelerators like GPUs. However, with the current development of GPUs, it is possible to revisit the method and adapt it to modern architectures, making it viable for production use.

This study aims to adapt the direct method for GPUs clusters. Using multiple GPUs together, the algorithm can be significantly optimized, minimizing execution time. However, with the use of GPUs clusters, the extra network traffic cost has to be counted. Therefore, a detailed study on the algorithm is required for an efficient implementation. This study was guided by a synthetic problem to simulate the behavior of N-body algorithm in cluster, making possible to watch the impact how each algorithm step collaborates to total execution time. After this step, an efficient version of the N-body algorithm is presented and tested in a cluster.

Keywords: High-Performance Computing; Grid Computing; Physics Simulations; N-Body simulation; GPU

Lista de Figuras

2.1 Exemplo do método hierárquico de Barnes-Hut [5]. 12

2.2 Exemplo do Método Partícula-Malha 13

2.3 Exemplo de um programa paralelo. 17

2.4 Execução de cada categoria de programas paralelos segundo a taxonomia de Flynn 18

2.5 Quatro partículas dispostas em um grid, com a força calculada para cada par. 22

2.6 Primeiro método paralelo para simulação de N-Corpos em GPU utilizando CUDA. 23

3.1 Divisão simples da simulação em múltiplas GPUs 29

3.2 Divisão da simulação em pequenos ladrilhos 30

3.3 Esquema de divisão de trabalho entre GPUs 33

3.4 Estrutura da comunicação entre os nós em anel 34

3.5 Sequência de atualização de cada ladrilho com a topologia de anel 35

3.6 Diferença da execução com fluxo padrão e vários fluxos 36

4.1 Tempo de execução, variando o número de streams, para simulações entre 65536 e 262144 partículas. 47

4.2 Tempo de execução, variando o número de streams, para simulações entre 327680 e 1048576 partículas. 48

4.3 Taxa de crescimento esperada x alcançada. 53

Lista de Tabelas

4.1 Parâmetros para baixa e alta demanda 44

4.2 Resultados dos testes com o Problema Amostra 45

4.3 Tempos normalizados de execução de um e quatro fluxos e a melhora alcançada. 49

4.4 Tempo de execução por iteração (em segundos) variando o número de fluxos no G-MPP 51

4.5 Tempo de execução por iteração (em segundos) do G-MPP e da aplicação original [46]. 52

Sumário

1 Introdução **1**

1.1 Contribuições da Tese 4

1.2 Organização 5

2 Background **6**

2.1 Modelagem Matemática 6

2.2 Simulação de N-Corpos 8

2.2.1 Partícula-Partícula 9

2.2.2 Percurso de Árvore 10

2.2.3 Partícula-Malha 13

2.2.4 Partícula-Partícula/Partícula-Malha 14

2.3 Conceitos Básicos 16

2.3.1 Computação Paralela 16

2.3.2 Computação Paralela em GPU 19

2.4 Simulação de N-Corpos Paralela 21

2.4.1 Método Direto Paralelo 21

2.4.2 Outros Métodos Paralelos 24

2.5 Sumário 25

3 G-MPP

Simulador de N-Corpos Massivo em GPU 27

3.1 Simulação de N-Corpos Paralela em GPU 29

3.2 Gerenciamento de Memória 33

3.2.1 Comunicação entre nós do cluster 34

3.2.2 Fluxos de dados em GPU 35

3.3 Detalhes de Implementação 37

3.4 Sumário 40

4 Avaliação Experimental 41

4.1 Problema Amostra 41

4.1.1 Avaliação do Problema Amostra 44

4.2 Análise de Fluxos de Dados em GPU 46

4.3 Avaliação do G-MPP 50

5 Conclusão 54

5.1 Trabalhos Futuros 55

Referências **57**

Apêndice A – Capturas de Tela do Simulador **62**

Capítulo 1

Introdução

Em Computação, o uso de simulações numéricas oferece uma maneira prática de se estudar diversos problemas, ao permitir que os mesmos sejam observados em um ambiente conhecido e controlável. Tais simulações são computacionalmente difíceis de serem executadas. A grande quantidade de dados e a complexidade matemática impulsionaram o uso de paradigmas paralelos de programação, onde a aplicação é dividida em tarefas menores, que podem ser resolvidas paralelamente, e também o uso de aceleradores como as GPUs, que são altamente paralelos e otimizados para cálculos matemáticos.

Dentre os vários problemas que podem ser resolvidos através de simulações, destaca-se neste trabalho o problema de N-Corpos, proposto por Newton [42]. O problema de N-Corpos é o problema de predizer os movimentos individuais de grupos de objetos(partículas, corpos) interagindo entre si. Resolver este problema pode, por exemplo, permitir entender melhor o comportamento de multidões ou o movimento de astros celestes, como o Sol, Lua, planetas e estrelas visíveis. Exemplificando, ao modelar o problema de N-Corpos para estudar o movimento de astros, cuja interação se dá pela atração gravitacional entre eles, pode-se estudar os mecanismos de formação de galáxias e outras

estruturas [18]. Já ao modelar o problema para estudar comportamento de multidões, utilizando a influência de cada indivíduo como força de interação, pode-se entender como bandos são formados ou o padrão de migração dos grupos [65].

No século 20, impulsionados principalmente pelo desenvolvimento do campo da astrofísica e da necessidade de entender melhor o Universo, a simulação de N-Corpos passou a ter maior importância. A simulação de N-Corpos é a simulação de um sistema dinâmico de partículas que interagem entre si. As primeiras simulações de N-Corpos foram realizadas na década de 1970, mas se estudo vêm de ainda antes [29]. Esses primeiros simuladores envolviam um código para simulação da interação através do campo gravitacional e se utilizavam da soma direta em um esquema de interação conhecido como PP (Partícula-Partícula), sendo conhecidos como métodos diretos. Este esquema envolve o cálculo de todas as forças entre cada par de partículas, possuindo assim complexidade de tempo $O(n^2)$. Outros simuladores que utilizam esta técnica são [50, 64].

A segunda geração de simuladores introduziu esquemas de interação alternativos, que diminuíram a complexidade do problema para $O(n \log n)$. Ao utilizar um processo de aproximação do agrupamento das partículas em estruturas hierárquicas (p.ex. árvores), pode-se aproximar a influência de um grupo sobre uma partícula, diminuindo a quantidade de operações a serem realizadas na simulação. O algoritmo de árvore mais utilizado foi introduzido em 1986, por Barnes e Hut [5]. Estes métodos tornaram computacionalmente viáveis o uso das simulações de N-Corpos em produção. Porém, a partir de duas partículas, este sistema se torna caótico [4], o que significa que mesmo pequenos erros podem crescer exponencialmente com o tempo. Por isso, os erros introduzidos pelas aproximações realizadas pelos métodos baseados em árvore devem ser estritamente

controlados.

A partir da década de 1990, já baseados no esquema de interação hierárquicos, houve o início do movimento de paralelização dos métodos [22, 31, 26], permitindo que simulações ainda maiores fossem realizadas. Já a partir da última década, com o desenvolvimento da arquitetura das unidades de processamento gráfico (GPU) [17], os métodos hierárquicos passaram a ser portados para esta arquitetura. Houve também o aumento do uso das GPUs organizadas em clusters e grids, aumentando ainda mais o poder computacional. Clusters e grids de GPUs apresentam a vantagem de manter uma melhor proporção de consumo de energia por desempenho e ocupar menos espaço físico [16, 33].

O método direto, da primeira geração de simuladores, foi implementado em GPU com o início da arquitetura CUDA [45, 46]. Enquanto o método direto apresenta complexidade de tempo maior do que os métodos hierárquicos, apresenta também resultados numericamente mais precisos, devido à ausência de aproximações. Suas características permitem uma implementação simples e que casa perfeitamente com a arquitetura da GPU: cada interação entre um par de partículas é completamente independente das outras. Assim, pode-se utilizar eficientemente a arquitetura da GPU, que segue o paradigma SIMT [36].

As limitações do método direto envolvem dois pontos: a complexidade de tempo e o consumo de memória. Devido a complexidade $O(n^2)$ do método direto, o tempo de execução do código cresce muito rapidamente conforme se aumenta o número de partículas do sistema. Além disso, ainda que o tempo não seja fator limitante, há o limite do espaço de memória disponível, já que o algoritmo necessita conhecer todas as partículas durante toda a execução.

Este trabalho se propõe a adaptar o método direto para cluster de GPUs. Utilizando múltiplas GPUs, o tempo de execução do algoritmo pode ser significativamente diminuído. Porém, com o uso do cluster, tem-se também os custos extras de comunicação entre cada nó. No primeiro passo, para quantificar o impacto da comunicação dos dados e da execução do algoritmo na GPU, foi modelado um problema sintético, derivado do método direto de simulação de N-Corpos. A partir deste problema um algoritmo eficiente para a simulação de N-Corpos em cluster, um simulador de N-Corpos Partícula-Partícula massivo (G-MPP, GPU Massive Particle-Particle simulator), é apresentado. O G-MPP propõe uma divisão de trabalho visando dividir o impacto do tráfego de dados dentro do cluster, ao enviar informações em pacotes menores, enquanto a GPU processa cada pacote de dados separadamente. Assim, permite-se a sobreposição do tempo de execução do algoritmo com o tempo de cópia de dados, aumentando o fluxo de dados do algoritmo. Além disso, ao permitir que a GPU opere sobre um pacote de dados menor, contorna-se o limite da memória da GPU. Embora o G-MPP tenha sido modelado para clusters, o algoritmo também será aplicado em outros dois cenários: um único nó com múltiplas GPUs e um único nó com uma única GPU. Assim, podemos avaliar o nosso algoritmo com a literatura.

1.1 Contribuições da Tese

Esta tese apresenta o G-MPP: um algoritmo otimizado para simulações de N-Corpos Partícula-Partícula em cluster de GPUs. A modelagem inicial do problema sintético permitiu um estudo mais detalhado do impacto das cópias de memória e da execução da GPU no tempo total do algoritmo.

A partir do estudo do problema sintético, pode-se desenvolver o G-MPP com foco na eficiência na comunicação de dados, permitindo que o algoritmo, mesmo não diminuindo a complexidade geral do problema, melhore o seu desempenho, permitindo que o mesmo seja utilizado em aplicações reais. Assim, o método proposto também atende à falta de métodos exatos para simulação massivas de N-Corpos em cluster de GPUs, já que na literatura o foco é grande em torno dos métodos hierárquicos.

1.2 Organização

Este trabalho é dividido da seguinte maneira: no capítulo 2, são apresentados os conceitos relacionados, o problema de N-corpos e os métodos computacionais mais conhecidas da literatura. No capítulo 3 é apresentado a modelagem do G-MPP, o algoritmo desenvolvido neste trabalho. O capítulo 4 traz a avaliação experimental e a análise dos resultados obtidos. O capítulo 5 conclui o trabalho e apresenta ideias de trabalhos futuros.

Capítulo 2

Background

Resumidamente, o problema de N-corpos é o problema de predizer o movimento de um grupo de objetos que interagem entre si [35]. A solução deste problema, através da simulação de N-Corpos, é motivada pela necessidade de entender o movimento de estrelas [37, 18, 53, 54], comportamento de moléculas [2, 7], movimentação de robôs ou multidões [59, 11], entre outros. Neste capítulo, serão apresentados os conceitos relacionados do trabalho, como a modelagem matemática do problema de N-Corpos e os principais métodos para resolução da simulação.

2.1 Modelagem Matemática

Para entender melhor os métodos de simulação de N-Corpos, é necessário entender a modelagem matemática do problema. A modelagem a seguir é apresentada em [39]. Informalmente, pode-se definir o problema de N-corpos como:

> Dada somente as posições e velocidades de um grupo de corpos, prediga seus movimentos para o tempo futuro, e deduza seu movimento no tempo anterior.

Ao trabalhar mais nesta definição, chega-se a uma definição mais precisa. Com um pouco mais de rigor matemático, define-se o problema de N-Corpos como a seguir:

> Considere-se N massas pontuais $m_1, m_2, ..., m_n$, no espaço físico tridimensional. Supondo que a força de atração entre cada par de objetos é Newtoniana. Então, se as posições e velocidades iniciais são especificadas para todas as partículas em um tempo t_0, determine a posição de cada partícula em todo momento futuro ou passado [52].

A atração entre duas partículas é regida de acordo com a lei da gravitação universal. Portanto, dada uma partícula i, com uma posição inicial x_i e velocidade v_i, podemos calcular a atração gravitacional a uma partícula arbitrária j através da equação 2.1:

$$f_{ij} = G\frac{m_i m_j}{||\overrightarrow{r}_{ij}||^2} \times \frac{\overrightarrow{r_{ij}}}{||\overrightarrow{r_{ij}}||} \tag{2.1}$$

Nesta equação, temos que G é a constante gravitacional, $\overrightarrow{r_{ij}}$ é o vetor da partícula i ao j e m_i e m_j são as massas das respectivas partículas i e j. Como existem N partículas no sistema, todas exercendo força de atração entre si, a força resultante F_i é a soma de todas as interações entre i e todas as outras partículas, como na equação 2.2:

$$F_i = Gm_i \sum_{1 \leq j \leq N}^{i \neq j} \frac{m_j \overrightarrow{r_{ij}}}{||\overrightarrow{r_{ij}}||^3} \tag{2.2}$$

Conforme os corpos se aproximam, a força de atração entre eles cresce para o infinito, gerando um comportamento não desejável. Para contornar essa situação, um fator de suavização ϵ^2 é adicionado na 2.2, resultando na 2.3:

$$F_i \approx Gm_i \sum_{1 \leq j \leq N} \frac{m_j \, \overrightarrow{r_{ij}}}{(||\overrightarrow{r_{ij}}||^2 + \epsilon^2)^{3/2}} \tag{2.3}$$

Sem este fator ϵ, deve-se tomar o cuidado de não calcular a força de uma partícula i com ela mesma. Nesse caso, o vetor r_{ij} seria nulo e ocorreria uma divisão por zero. Considerando então $\epsilon^2 > 0$ e a força $f_{ii} = 0$, a condição de $i \neq j$ não é mais necessária. Este fato permite uma simplificação no algoritmo, apresentado nas seções a seguir.

Para a atualização da velocidade e posição das partículas, é necessária a aceleração. De acordo com a segunda Lei de Newton, tem-se que $F_i = m_i \, a_i$. Portanto, podemos cancelar o termo m_i na equação 2.3, obtendo ao final a aceleração resultante, conforme a equação 2.4.

$$F_i \approx G \sum_{1 \leq j \leq N} \frac{m_j \, \overrightarrow{r_{ij}}}{(||\overrightarrow{r_{ij}}||^2 + \epsilon^2)^{3/2}} \tag{2.4}$$

O problema de N-corpos gravitacional consiste em resolver esta equação para todos as partículas, em cada passo de tempo, atualizando as suas posições e velocidades conforme as partículas interagem entre si. Este problema é computacionalmente resolvido através da simulação de N-Corpos, que será discutida a seguir.

2.2 Simulação de N-Corpos

A simulação de N-Corpos visa resolver o problema de N-Corpos, apresentado na seção anterior. Os métodos de simulação foram sendo refinados ao longo do tempo. Essa evolução ocorreu principalmente pela necessidade simulações com maior número de partículas e se desenvolveu ao longo das últimas décadas. O método original e os refinamentos mais

conhecidos serão apresentados nas subseções a seguir.

2.2.1 Partícula-Partícula

O primeiro método foi desenvolvido inicialmente por Sebastian von Hoerner [60], na década de 1960. Por se utilizar da soma direta da interação entre todas as partículas, é conhecido como método direto ou Partícula-Partícula.

Possuindo um sistema com muitas partículas, todas sob influência mútua de suas forças gravitacionais, a simulação de N-corpos Partícula-Partícula (PP) integra numericamente todas as forças sem qualquer tipo de simplificação ou aproximação, o que proporciona o máximo de precisão na simulação. Esta precisão vem às custas do desempenho, já que para atualizar cada partícula, a informação de todas as outras é necessária. Suavizando um pouco o impacto no desempenho, em um sistema com N partículas, ao invés de calcular N vezes a interação com as outras $N-1$ partículas, o método permite o cálculo de somente $\frac{N*(N-1)}{2}$ forças. Isto se deve ao fato de que, devido à equação 2.4, a força entre um par de corpos é mútua, possuindo módulo igual, mesmo sentido e somente direção inversa.

O algoritmo 1 mostra a implementação do método Partícula-Partícula. Deve-se percorrer todas as partículas (linhas 2 a 9 no algoritmo). Então, para cada partícula, deve-se percorrer todas as outras, calculando a sua interação com a partícula atual do laço externo (linhas 5 a 7). Com isso, podemos observar que a complexidade é de $O(N^2)$. A atualização da partícula (linha 8) somente altera sua posição e velocidade, com complexidade constante.

Esse método então se mostra preciso, pois calcula a força de maneira exata, mas

Algoritmo 1 Método Partícula-Partícula.

Input: Posição e Velocidade das partículas / Número de passos da simulação
Output: Partículas com posição e velocidade atualizadas

```
1:  function CPU(*particles, *force, maxSimSteps)
2:      while simStep < maxSimSteps do                ▷ Em cada passo da simulação
3:          for all particle i in particles do                    ▷ Para cada partícula
4:              force = 0
5:              for all particle j in particles do          ▷ Itere sobre todas as outras
6:                  force += bodybodyInteraction(i, j)     ▷ Calcule a força entre o par
7:              end for
8:              updateParticle(i, force)                          ▷ Atualize a partícula
9:          end for
10:         simStep++
11:     end while
12: end function
```

possui desempenho baixo, devido a sua complexidade. Esse equilíbrio entre precisão e desempenho levou à criação de métodos onde o foco fosse o desempenho, ainda que em detrimento à precisão. Estes métodos serão apresentados nas subseções a seguir.

2.2.2 Percurso de Árvore

Enquanto no método Partícula-Partícula a precisão matemática é alta, todos as partículas fazem parte dos cálculos, não importando a distância entre si. Em casos onde esta distância é muito grande, a força resultante é ínfima. Assim, em busca de melhor desempenho da aplicação, estas forças ínfimas podem ser desprezadas, utilizando somente uma força resultante destas. Esta abordagem é a utilizada pelo método hierárquico de percurso de árvore, desenvolvido inicialmente por Barnes e Hut [5].

Seja a o raio de um disco D, de modo que um conjunto de partículas P esteja contido nesse disco. Muitos sistemas físicos possuem a propriedade de que o campo gerado pelo conjunto de partículas P é complexo dentro do disco, porém mais simples fora do mesmo. A força gravitacional possui essa característica. Essa propriedade é

utilizada para simplificar os cálculos da simulação de N-corpos: grupos de partículas a partir de uma distância pré-definida d de uma partícula arbitrária são consideradas como uma única partícula para a soma.

Este método é efetivo pois permite tratar um grupo de partícula como uma única.. Assim, o espaço é subdivido em uma árvore, onde cada folha contém no máximo uma partícula. Cada nó interno desta árvore possui uma partícula no centro de massa, calculada a partir de todos seus filhos.

A subdivisão do espaço cria uma *kd-tree* [47], uma árvore de subdivisão espacial que pode ser construída em tempo $O(n \log(n)$ [61] e possui altura máxima $\log(n)$. Após a construção da árvore, para calcular a força em cada partícula, basta percorrer, para cada partícula, a árvore a partir da raiz. Então, este método reduz a complexidade de $O(n^2)$ para $O(n \log(n)$.

O método proposto por Barnes e Hut, em [5], possui alguns ingredientes principais: uma divisão espacial virtual regular, onde cada célula é dividida em 8 células iguais; a construção da árvore de células a partir da divisão virtual, utilizando somente as células não-vazias e aceitando as células com somente uma partícula; e a reconstrução total da árvore a cada passo de tempo. Para cada célula não-vazia, tanto nós folha quanto interno da árvore, é criada uma pseudo-partícula que contém a soma das massas da célula, localizada no centro de massa das partículas que ela contém. Assim, resumidamente, os passos para execução do algoritmo são:

1. Construir a árvore, subdividindo o espaço recursivamente.

2. Percorrer a árvore das folhas para a raiz, computando o centro de massa e a massa

total de cada nó pai.

3. Para cada partícula, percorrer a árvore a partir da raiz, calculando a força durante o percurso.

A figura 2.1 é um exemplo de aplicação do método de Barnes-Hut. Para atualizar a área em vermelho, as partículas das áreas vizinhas são consideradas uma a uma, como o método Partícula-Partícula. Já nas áreas mais afastadas, somente a partícula resultante é avaliada. Assumindo uma distribuição normal das partículas, pode-se observar que à medida que a distância aumenta há um ganho de desempenho grande, porém uma perda também considerável de acurácia, devido ao tamanho das áreas unificadas em um único ponto.

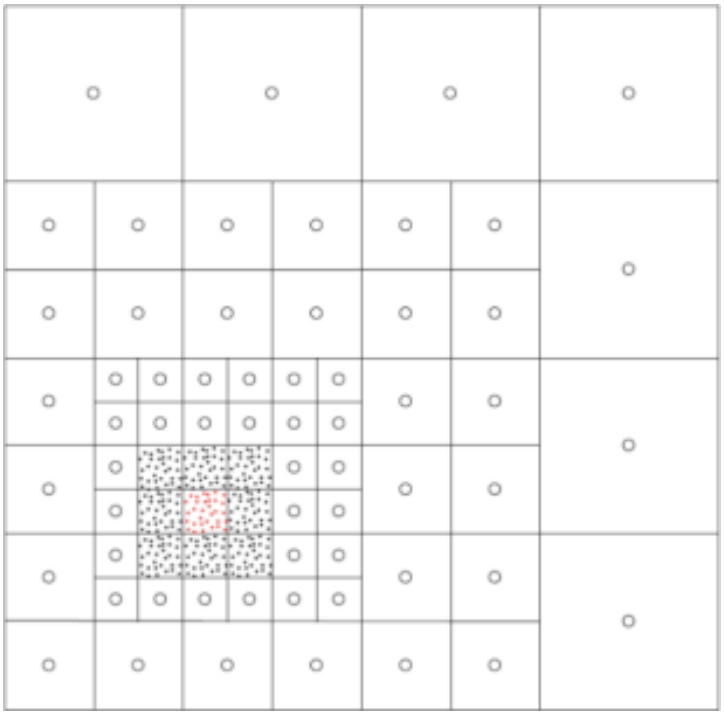


Figura 2.1: Exemplo do método hierárquico de Barnes-Hut [5].

2.2.3 Partícula-Malha

Nas simulações previamente descritas, o tempo já está num espaço discreto, sendo incrementado de um δt a cada passo da simulação. O método Partícula-Malha [28], PM, discretiza também o espaço. Esta discretização permite que o cálculo da força resultante em cada partícula se resuma ao cálculo do potencial em cada ponto da malha. A seguir, passa-se à atualização de cada partícula.

O método funciona criando uma malha sobre o espaço que contém as partículas a serem simuladas. Em seguida, basta resolver a equação do potencial nos pontos desta malha e calcular a força em cada ponto da malha pelo gradiente do potencial. Para encontrar a força em uma partícula fora dos pontos da malha, pode-se considerar a mais próxima ou a interpolação dos potenciais vizinhos [28]. A figura 2.2 ilustra este cálculo. Para calcular a força resultante da partícula, é necessário o potencial no ponto contido na célula 2, que contém a partícula a ser calculada, ou a interpolação dos pontos das células 1, 2, 3 e 4, que são as células mais próximas à partícula.

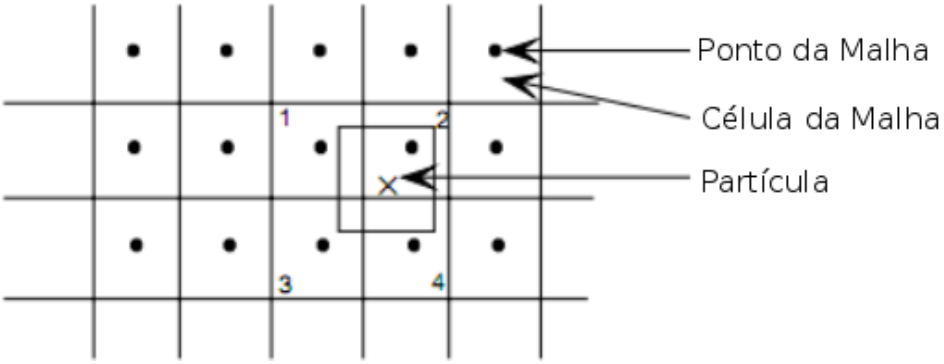


Figura 2.2: Exemplo do Método Partícula-Malha. Para calcular a força na partícula, pode-se utilizar o ponto da malha mais próximo ou a média dos pontos da malha vizinhos.

Para o cálculo do potencial, é necessário calcular a densidade (quantidade de massa/unidade de volume) em cada ponto da célula. Diversas abordagens podem ser utilizadas, como: **a)**

adicionar cada massa ao ponto da malha mais próximo; **b)** interpolar a massa, dividindo-a proporcionalmente entre os quatro pontos de malha mais próximos.

Por fim, para resolver a equação do potencial, é necessário utilizar a Transformada de Fourier [13]. O tempo para este cálculo é de $O(M \log(M))$ [62], onde M é o número de pontos na malha. Para que o método ganhe em desempenho, é necessário que $M < N$.

Como cada partícula é visitada uma vez para o cálculo da densidade e para a atualização, o método possui complexidade de $O(n)$ em relação às partículas, sendo a FFT a etapa mais lenta. Em troca do alto desempenho, o método cria uma limitação espacial que impede que fenômenos pequenos sejam modelados corretamente. A aproximação do potencial da célula impede que interações somente entre partículas dentro da mesma sejam corretamente calculadas.

2.2.4 Partícula-Partícula/Partícula-Malha

O método Partícula-Malha/Partícula-Partícula [28](PPPM, ou P^3M) visa corrigir o problema do método Partícula-Malha com os fenômenos que ocorrem na escala menor que o tamanho da célula, também chamados de fenômenos de curto-alcance. A ideia deste método é simples: separar o cálculo da força em forças de curto alcance e forças de longo alcance. O método Partícula-Malha é utilizado para as forças de longo alcance, e o método Partícula-Partícula é utilizado para corrigir o método Partícula-Malha nas distâncias curtas.

O método Partícula-Malha trata as partículas como uma nuvem de massas, onde cada partícula oferece uma parcela que depende de detalhes do método Partícula-Malha. Sendo $e(x_0, x_i)$, a parcela da partícula x_i sobre a partícula x_0, realiza-se uma subtração

dessa parcela no método Partícula-Partícula, e no final, soma-se o resultado do método Partícula-Partícula para obter a força final calculada. Em caso de massas pontuais, a equação 2.5 mostra a aproximação da força final calculada pelo método:

$$F_i \approx G \sum_{1 \leq j \leq N} \frac{m_j \, \overrightarrow{r_{ij}}}{(||\overrightarrow{r_{ij}}||^2 + \epsilon^2)^{3/2} - e(x_i, x_j)} \tag{2.5}$$

O algoritmo 2 ilustra o funcionamento do método. No primeiro passo, a primeira parcela da força é calculada pelo método Partícula-Malha (linha 4 do algoritmo), apresentado na subseção 2.2.3. A seguir, são selecionadas as partículas mais próximas da partícula i a ser atualizada (linha 5). Com estas partículas, uma segunda força é calculada, através do método Partícula-Partícula, apresentado na subseção 2.2.1 (linha 6), conforme a equação 2.5. A força final é dada pela soma das duas parcelas (linhas 7 e 8).

Algoritmo 2 Método PPPM.

```
 1: function PPPM(maxSimSteps)
 2:     while simStep < maxSimSteps do
 3:         for all particle i in particles do                ▷ Para cada partícula
 4:             F = calculatePM()                             ▷ Método Partícula-Malha
 5:             set P = getParticlesCloser(i, r_0)            ▷ Separa as partículas mais próximas
 6:             F_s = calculatePP(P)                          ▷ Método Partícula-Partícula
 7:             F_t = F + F_s
 8:             updateParticle(i, F_t)                        ▷ Atualiza posição e velocidade
 9:         end for
10:         simStep++
11:     end while
12: end function
```

Esse método apresenta um novo equilíbrio acurácia/desempenho em relação ao método Partícula-Malha. Ao corrigir o problema dos fenômenos de curto alcance, através da adição do método Partícula-Partícula, sua eficácia aumenta. Porém, devido à complexidade do método Partícula-Partícula, o desempenho é impactado. Por isso, o tempo de execução do método P^3M é facilmente dominado pela parcela Partícula-Partícula. Além

disso, por ainda possuir uma parte onde o cálculo é aproximado (parcela Partícula-Malha), sua precisão também é inferior ao método Partícula-Partícula.

2.3 Conceitos Básicos

Diversas implementações paralelas dos métodos de simulação de N-Corpos já foram publicadas na literatura. Esta seção se dedicada a apresentar alguns conceitos básicos de Computação Paralela e Computação em GPU, para que a seguir as vantagens e as limitações destas implementações possam ser discutidas. A próxima subseção apresenta alguns conceitos de Computação Paralela. Em sequência, conceitos básicos sobre GPUs também são apresentados.

2.3.1 Computação Paralela

Podemos definir Paralelismo como uma técnica usada para decompor aplicações complexas em tarefas menores, obtendo resultados com maior rapidez [51]. A Figura 2.3 ilustra um modelo de divisão de uma aplicação. A aplicação inicial é dividida em tarefas menores T1, T2 e T3, que são enviadas aos processadores disponíveis P1, P2 e P3. Esses processadores executam cada tarefa de forma independente ou comunicativa, trocando informações e sincronizando seu processamento.

Dentre os objetivos da programação paralela e distribuída, podemos destacar:

- diminuir o tempo de processamento, permitindo resolver desafios computacionais complexos, que demandariam muito tempo para serem executadas;
- dividir a quantidade de dados processados por cada unidade. Assim, uma tarefa global que não podia ser executada por limitação de memória pode passar a ser

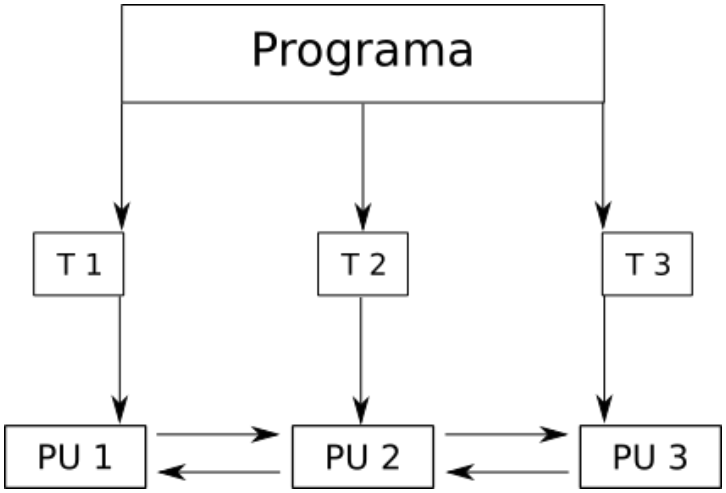


Figura 2.3: Exemplo de um programa paralelo. Cada processador recebe e executa uma tarefa, podendo comunicar e sincronizar com os outros processadores.

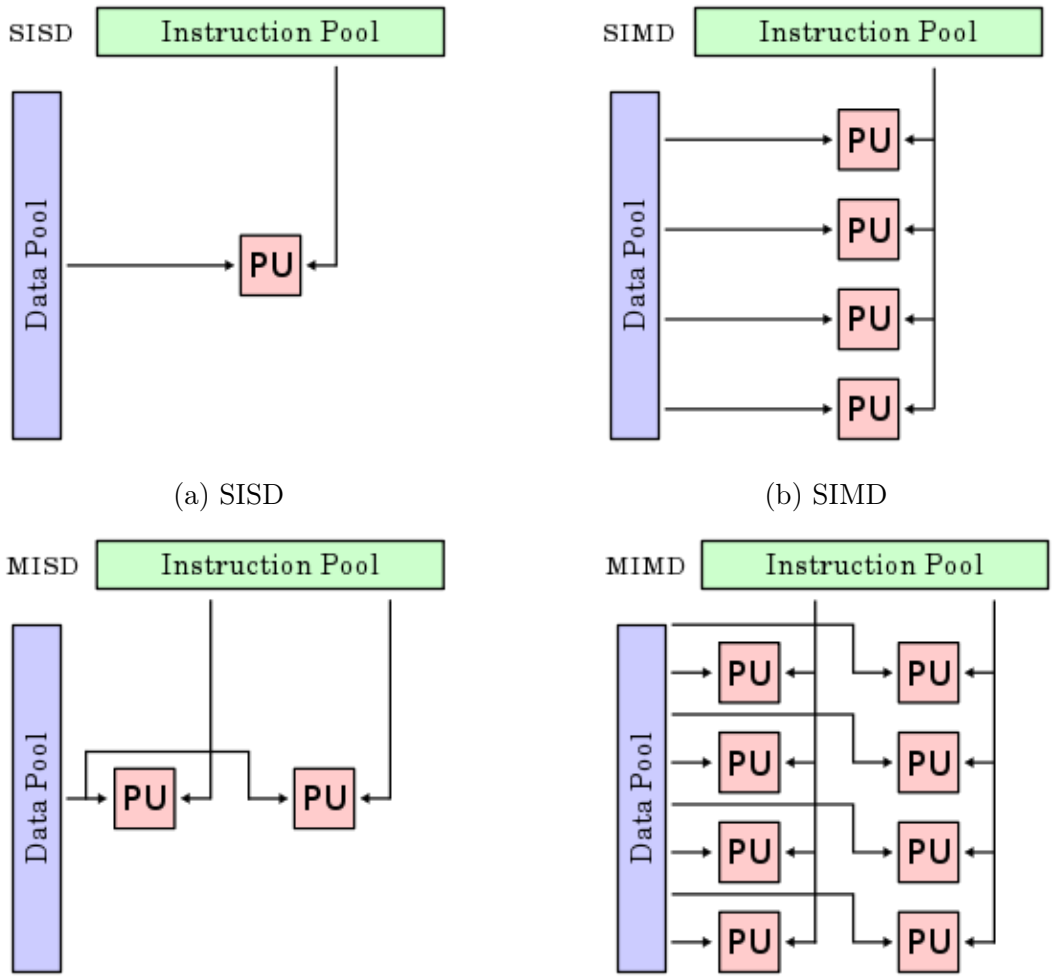
factível.

De acordo com a taxonomia de Flynn [20, 15, 58], as arquiteturas paralelas podem ser divididas em quatro classes, baseadas na quantidade de instruções (ou controles) concorrentes e quantidade de fluxo de dados disponíveis na arquitetura.

- SISD - *Single Instruction, Single Data Stream*: programa sequencial, que não aproveita de nenhum paralelismo, seja de instruções, seja de dados. Os processadores de núcleo único entram nessa categoria.

- SIMD - *Single Instruction, Multiple Data Streams*: se aproveita de múltiplos fluxos de dados para processá-los paralelamente, utilizando o mesmo fluxo de instruções em todos os dados. É a arquitetura utilizada em processadores vetoriais e GPUs.

- MISD - *Multiple Instructions, Single Data Stream*: arquitetura onde várias instruções operam sobre o mesmo fluxo de dados. Esta arquitetura incomum é utilizada principalmente para tolerância à falhas.



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figura 2.4: Exemplo de execução em cada uma das categorias de programas paralelos segundo a taxonomia de Flynn. Cada processador *PU* recebe as suas instruções da *Instruction Pool* e os dados da *Data Pool* para execução.

- MIMD - *Multiple Instructions, Multiple Data Streams*: vários processadores independentes operam sobre fluxos de dados também independentes. Processadores superescalares multi-core são exemplo desta arquitetura.

Alguns autores [10, 38] ainda dividem a categoria MIMD em diversas outras. Na figura 2.4 temos um exemplo para cada uma das arquiteturas citadas.

Cada uma das classes possui aplicações onde são mais eficientes do que as outras. No caso específico da aplicação desta tese, a arquitetura SIMD casa perfeitamente com o

método Partícula-Partícula. A atualização de cada partícula (em um espaço de memória separado) envolve um conjunto de passos iguais - calcular a interação da partícula com as outras, computar a força resultante e atualizar a partícula. Este é o comportamento ideal para a arquitetura SIMD: cada processador se ocupa com uma unidade da memória diferente, porém todas tem sua execução guiada pela mesma instrução corrente.

2.3.2 Computação Paralela em GPU

Uma GPU (*Graphical Processor Unit*) é um processador originalmente dedicado ao processamento gráfico. É composta de diversos processadores paralelos (de centenas a milhares), provendo desempenho muito maior do que CPUs modernas em diversos cenários e alcançando número muito maior de operações por segundo. Em um exemplo, a Tesla K20 [44] pode sustentar até 4,7 TFLOPS/s contra 60 GFLOPS [3] médio encontrado nas CPUs [30].

A arquitetura da GPU é especialmente criada para processar tarefas com alta complexidade matemática, com pouca troca de memória, exigindo alto desempenho aritmético, enquanto não utiliza a memória constantemente. Sua arquitetura, nomeada SIMT [24] (*Single Instruction, Multiple Threads*), é uma variação da arquitetura SIMD, onde os múltiplos dados são acessados através de múltiplas *threads* que executam suas tarefas recebendo todas a mesma instrução a cada passo.

Esta arquitetura mais o fato da quantidade de ULAs disponíveis em uma GPU exige que a aplicação desenvolvida para ela utilize um paradigma de programação diferente do modelo tradicional sequencial para CPU. Por exemplo, um processador moderno tem entre 2 e 6 núcleos, enquanto a Tesla K20 possui 2496. Para aproveitar ao máximo o

poder de processamento da GPU, o desenvolvedor precisa adaptar as tarefas para este paradigma altamente paralelo. Além disso, ao utilizar clusters, deve conciliar também a comunicação entre cada nó do cluster e sua GPU. Isto se deve ao fato das GPUs serem somente uma unidade auxiliar de processamento, necessitando da CPU para controle de execução.

A GPU é utilizada no PC como um acelerador para processar dados e executar tarefas computacionalmente caras. Para isto, se utiliza de *frameworks* como CUDA [12] e OpenCL [23]. Estes *frameworks* facilitam o uso da GPU como um coprocessador de propósito geral.

A maior vantagem da utilização destes é a flexibilidade dada ao desenvolvedor. Anteriormente ao desenvolvimento do CUDA e do OpenCL, o uso das GPUs como processadores genéricos envolvia traduzir o problema para uma linguagem dedicada a gráficos (linguagens de shader), e muitas vezes eram necessários operadores que não existiam nessas linguagens, como alguns operadores relacionais, com números inteiros e booleanos [48]. Enquanto o OpenCL pode ser utilizada em hardware de qualquer fabricante, tem a desvantagem de ser mais lento que o CUDA [14, 1]. Já o CUDA, apesar do melhor desempenho, só é compatível com placas da fabricante Nvidia.

A Computação em GPU (*GPU Computing*) está se tornando cada vez mais comum e sendo aplicada em áreas diversas. Além das já citadas gravitação [37, 18, 53, 54], física molecular [2, 7], simulação de multidões [11], podemos destacar as áreas de medicina [40] e geologia [32, 8].

Atualmente, não só se utiliza a GPU para processamento paralelo, como também se utilizam de múltiplas GPUs em um mesmo computador. Além disso, diversos clusters

de GPUs figuram entre os mais rápidos supercomputadores do mundo [57]. É tarefa do desenvolvedor gerenciar os dados e distribuí-los entre os vários nós e GPUs, garantindo a consistência entre todas as cópias dos dados disponíveis, tornando a tarefa mais complicada. Entre vários trabalhos que utilizam várias GPUs, destacam-se muitos relacionados à processos matemáticos, como cálculo da Transformada Rápida de Fourier [43] ou o Gradiente Conjugado [9]. Esta tese também utiliza um cluster de GPUs.

2.4 Simulação de N-Corpos Paralela

As abordagens paralelas da simulação de N-Corpos permitiram que o tempo utilizado no processamento fosse amplamente reduzido, trazendo a possibilidade de se investir em simulações com número cada vez maior de partículas. A tese foca no método Partícula-Partícula para GPUs, porém não é a primeira implementação do mesmo. Esta seção discute a paralelização já conhecida do método Partícula-Partícula em GPU e discute seus pontos positivos e negativos. Também é apresentado um resumo sobre paralelização dos métodos hierárquicos mais conhecidos, também avaliando pontos positivos e negativos.

2.4.1 Método Direto Paralelo

Nyland e Harris apresentam em [46] uma primeira paralelização do método Partícula-Partícula em GPUs. De acordo com os autores, uma paralelização simples do método direto cria um grid de tamanho $n \times n$, contendo todas as interações entre os n corpos. A força gravitacional final sobre um corpo i qualquer é a soma de todas as forças da linha i. A figura 2.5 ilustra um caso com 4 partículas. Note que não é necessário calcular a força da iteração entre a partícula e ela mesma.

	a	b	c	d
a		F(a,b)	F(a,c)	F(a,d)
b	F(b,a)		F(b,c)	F(b,d)
c	F(c,a)	F(c,b)		F(c,d)
d	F(d,a)	F(d,b)	F(d,c)	

Figura 2.5: Quatro partículas dispostas em um grid, com a força calculada para cada par.

Como cada força é completamente independente, é possível calcular todas as interações entre os corpos em paralelo. A desvantagem em calcular todas as forças em paralelo é que os resultados parciais precisam ser armazenados em uma matriz, utilizando um espaço de memória de $O(n^2)$. Além disso, uma operação de redução é necessária para somar as forças em cada linha, para obter o resultado final. Esta paralelização tem seu uso limitado pela quantidade de memória da GPU, geralmente muito menor que a memória disponível para a CPU.

Para limitar essa demanda de memória, em [46] uma paralelização em linhas é proposta: ao invés de calcular todas as forças em paralelo, cada tarefa consiste em calcular a interação de uma partícula com todas as outras. A figura 2.6 ilustra este passo. Cada linha do grid é processada em paralelo, realizando um loop por todas as colunas e atualizando somente uma partícula no final. Com o resultado final, basta atualizar a partícula com a força resultante ao final da execução. Com esse método, somente é necessário armazenar um valor por partícula com a força resultante. Isto diminui a complexidade de memória de $O(n^2)$ para $O(n)$, tornando-o mais adaptável para a GPU.

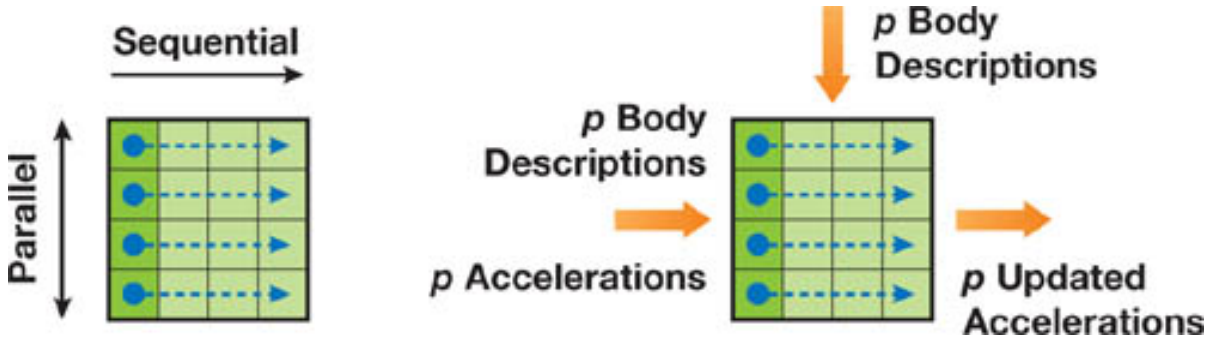


Figura 2.6: Primeiro método paralelo para simulação de N-Corpos em GPU utilizando CUDA. Cada linha é processada em paralelo, enquanto as colunas são processadas sequencialmente. [46]

As forças $F(i,j)$ e $F(j,i)$ são idênticas em módulo e direção, mudando somente o sentido. Quando há a paralelização total do grid, é simples evitar realizar duas vezes este cálculo, bastando inserir nas posições (i,j) e (j,i) a mesma força calculada $F(i,j)$, alternando somente o sentido quando necessário.

Porém, quando trabalhamos na paralelização por linhas, esta tarefa se torna mais complicada. Com somente um valor de força resultante para cada partícula, seria necessário um esquema de sincronização. Ao se programar em GPUs, este é um comportamento extremamente desaconselhável, devido a arquitetura SIMT [12].

Como só há um único controle de instrução, se várias threads chamarem operações atômicas, cada uma delas será executada sequencialmente, mesmo que tais operações ocorram em espaços separados de memória. Para evitar este problema, e aproveitando o desempenho da GPU em cálculos matemáticos, é preferível calcular separadamente as forças $F(i,j)$ e $F(j,i)$.

Em relação ao número de partículas máximo que a simulação suporta, é necessário que todas as partículas sejam conhecidas pela GPU a todo instante. Portanto, mesmo que o requisito de memória, conforme mostrado anteriormente, a memória da GPU limita

efetivamente a simulação.

2.4.2 Outros Métodos Paralelos

Um dos primeiros trabalhos a desenvolver um simulador paralelo escalável baseado no método de Barnes-Hut foi [63]. Grama *et al.* [22] trouxe uma análise sobre diferentes paralelizações em CPU do mesmo. Estes trabalhos se utilizavam de várias CPUs para a paralelização. Além disso, alguns hardwares especializados foram utilizados, como o GRAPE [37]. Este hardware paralelo executava de maneira similar a uma GPU (arquitetura SIMD), e conseguiu melhorar o desempenho do método Partícula-Partícula em relação aos processadores comuns. Pesa negativamente contra o uso de hardware especializado como este o alto custo para manter um sistema dedicado.

No início dos anos 2000, foram criados os primeiros simuladores em GPU. Em [45, 49], há a utilização da linguagem CG [19], apropriada para gráficos. Programar nas linguagens específicas de gráficos envolvia traduzir o problema para um problema na área de Computação Gráfica ou Processamento de Imagens, o que torna seu uso mais complicado, já que envolvia uma etapa de pré-processamento e pós-processamento que demandava além de conhecimentos do problema, domínio em áreas usualmente não relacionadas aos problemas.

Outros trabalhos também implementaram o método Partícula-Partícula, como [26]. Em [21], o método Partícula-Partícula foi adaptado para a GPU de forma a imitar o comportamento do hardware especializado GRAPE [37]. Como o GRAPE era um processador vetorial, a tradução entre GPU e GRAPE é simplificada. As limitações desta implementação eram similares ao simulador desenvolvido por Nyland em [46],

apresentado na subseção 2.4.1.

Em [6, 41, 56], os métodos hierárquicos foram implementados na GPU. Porém estes trabalhos ainda utilizavam a CPU para algumas tarefas, como a criação da árvore e a separação das partículas (pelo caminhamento da árvore) a serem utilizadas em cada atualização. Para a GPU ficava então a tarefa de calcular as forças e atualizar a partícula, similarmente ao método Partícula-Partícula. Esta abordagem limitava o desempenho, já que algumas de suas tarefas ainda eram realizadas de maneira sequencial na CPU.

Em [21, 27, 31, 34], tem-se implementações em múltiplas GPUs. Em [21], há a paralelização em múltiplas GPUs, porém somente no caso de várias GPUs em um único host. Em [27], há também o uso de clusters homogêneos de GPUs, onde todas as GPUs disponíveis são idênticas. O uso de clusters homogêneos permite contornar mais simplificadamente os problemas de desbalanceamento de carga e transferência de memória entre nós do cluster. Em [31] é realizado um estudo para escalar o trabalho hierárquico em clusters homogêneos. Os trabalhos anteriores citados que utilizam clusters de GPUs são relacionados aos métodos hierárquicos. Não há, até onde a pesquisa se estendeu, nenhum trabalho que trate do método Partícula-Partícula em um cluster de GPUs.

2.5 Sumário

Neste capítulo, foram apresentados e discutidos as principais abordagens e métodos para a simulação de N-Corpos. Os pontos principais, positivos ou negativos, foram levantados para cada abordagem. Após a apresentação de cada método, também discutiu-se as publicações mais relevantes sobre a paralelização dos métodos, apresentando também os pontos fracos e fortes. O próxima capítulo vai apresentar o trabalho desenvolvido nesta

tese, o G-MPP, para a simulação de N-Corpos em cluster de GPUs.

Capítulo 3

G-MPP
Simulador de N-Corpos Massivo em GPU

A simulação de N-Corpos é utilizada em muitos campos de pesquisa, e em especial na Física Computacional e Astrofísica. Como mostrado no capítulo 2, o método direto possuía complexidade $O(n^2)$. Com os métodos hierárquicos, essa complexidade caiu para $O(n \log n)$, porém trazendo aproximações nos cálculos.

Com o avanço do poder computacional das GPUs, novas implementações dos métodos hierárquicos foram desenvolvidas. Com o custo cada vez mais acessível, o uso de clusters de GPUs também cresceu. Novamente, o foco do desenvolvimento foram os métodos hierárquicos. Porém, com todo o poder computacional das GPUs, há novamente a possibilidade de se utilizar o método exato, eliminando todo tipo de aproximação nas operações.

No capítulo 2 apresentamos os principais problemas dos métodos já conhecidos de simulação de N-Corpos. Retomando a discussão, o método Partícula-Partícula possuía a limitação do baixo desempenho do método em uma única GPU. O método hierárquico de travessia de árvore melhorou o desempenho ao permitir que partículas fossem

agrupadas em uma única pseudopartícula. A melhora do desempenho, porém, trouxe simplificações nos cálculos, que levam a erros. O método Partícula-Malha e o P^3M se utilizaram de outra discretização, desta vez do espaço, para melhorar o desempenho, mas também simplificaram a soma total de forças, ocorrendo também nos erros trazidos pela simplificação.

Enquanto não faz nenhuma simplificação, o método Partícula-Partícula tem como desvantagem seu desempenho. A paralelização deste método em GPU já foi proposta, porém trazia um outro limitador: há a necessidade de toda as partículas estarem na memória da GPU a todo momento, o que limita o tamanho da simulação que pode ser feita. Além disso, o uso de uma única GPU, mesmo com a melhora de desempenho trazida pela tecnologia, se torna outro limitador para simulações massivas, com grande número de partículas, já que o tempo de execução cresce mais rápido do que o aumento do número de partículas.

Esta tese apresenta o G-MPP, simulador Partícula-Partícula massivo de N-Corpos, visando contornar estes dois limitadores. Com o uso de clusters de GPUs, podemos melhorar o desempenho do método, dividindo os cálculos entre várias GPUs. Além disso, com uma organização eficiente de memória e de comunicação entre CPU e GPU, o método proposto só precisa da informação de uma parte das partículas a cada instante, surpassando também a limitação da quantidade de memória da GPU. A seguir, discutiremos como a divisão de trabalho é feita e como a memória é organizada para otimizar o método.

3.1 Simulação de N-Corpos Paralela em GPU

Este subseção irá discutir a adaptação do método paralelo para clusters de GPUs. A abordagem mais simples envolve simplesmente copiar todas as partículas em cada uma das GPUs. Porém, invés de calcular todas as partículas, cada GPU ficaria responsável por atualizar um subgrupo destas, como mostra a Figura 3.1. Ao final de um passo da execução, há a necessidade de cada GPU comunicar todas as outras as alterações nas suas partículas, através de uma operação de broadcast. Isto torna a comunicação cada vez mais complexa conforme o número de nós aumenta.

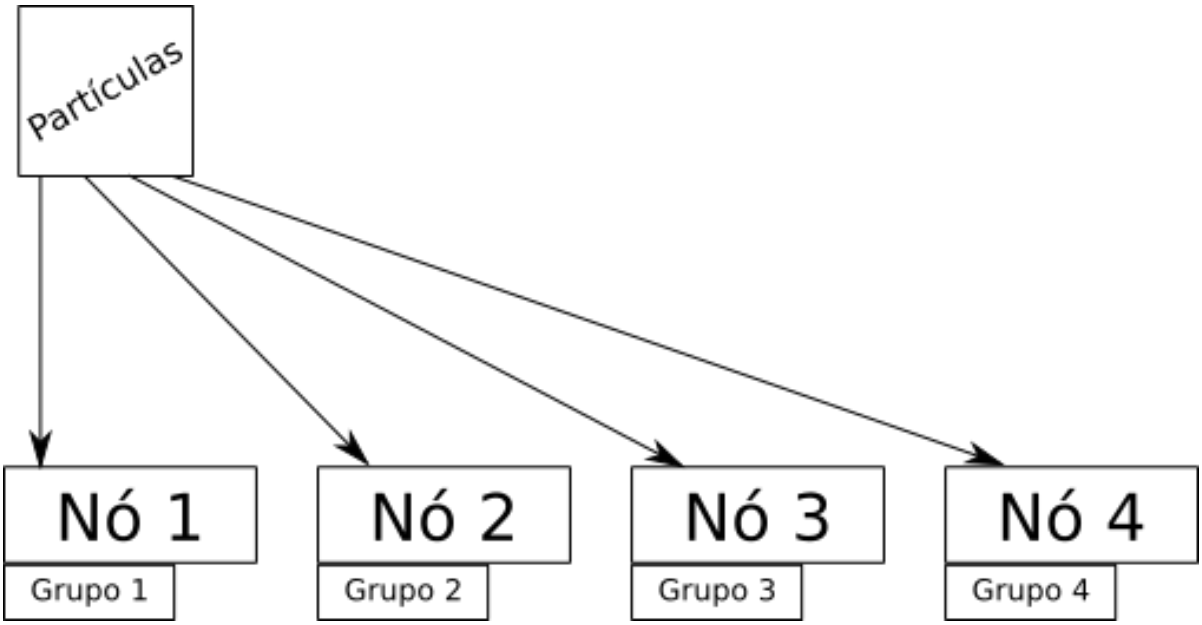


Figura 3.1: Divisão simples da simulação em múltiplas GPUs. Embora cada GPU conheça inicialmente todas as partículas, somente um subconjunto é calculado por cada uma. Ao final da execução, porém, todos as partículas de cada GPU devem ser substituídas pelas novas.

Esta abordagem possui implementação mais simples, mas ainda não elimina o limitador para simulações massivas: já que cada GPU precisa conhecer todas as partículas, o limite de partículas que podem ser simuladas é proporcional à memória de uma única GPU. Se levarmos em conta um cluster com GPUs diferentes em cada nó, o limite é

proporcional à GPU com menos memória disponível, o que pode levar a um desperdício de recursos computacionais ainda maior.

Antes de discutir como simular grandes quantidades de partículas, se faz necessário introduzir o conceito de subdivisão do trabalho da simulação. Na figura 2.5, tem-se as forças que devem ser calculadas para a solução do problema com 4 partículas. Podemos observar essa tabela de forças como um grid discreto, onde cada linha e cada coluna representa uma única partícula. Cada par (x, y) indica uma única força definida pela atuação da partícula y em x. A força é sempre não-nula para $x \neq y$, e nula caso contrário.

Esta tese então propõe uma subdivisão regular do grid de partículas. Cada subespaço deste grid é tratado como uma única simulação parcial de N-Corpos. A Figura 3.2 ilustra o processo. Cada GPU só precisa conhecer um pequeno subconjunto da simulação em cada passo.

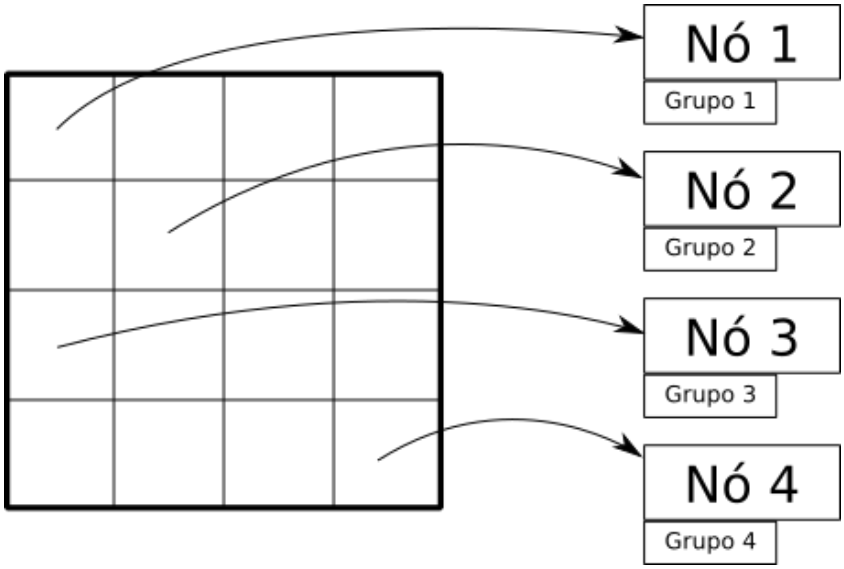


Figura 3.2: Divisão da simulação em pequenos ladrilhos. Cada GPU recebe um pequeno conjunto de trabalho a cada passo. Quando todos os conjuntos forem calculados, as forças resultantes em cada uma das partículas estará calculada e pode-se passar para o passo de integração.

Ao agruparmos as partículas desse grid em conjuntos maiores, como na figura 3.2,

temos um conjunto onde cada ladrilho é uma pequena simulação de N-Corpos parcial. Seguindo a figura, na primeira linha, tem-se 4 sub-simulações. Assumindo um total de 64 partículas no sistema representado na figura 3.2, cada sub-simulação contém 16×16 partículas. Assim, para a primeira linha, o primeiro ladrilho representa a interação entre as partículas da linha (as primeiras 16) com elas memas. Cada ladrilho a seguir representa a sub-simulação das mesmas partículas da linha com as outras do sistema, sempre em pequenos grupos. A soma dos resultados de cada linha dá a força resultante para atualizar as partículas correspondentes.

Resumidamente, o método então subdivide a tabela $n \times n$ de partículas em $n/A \times n/B$ ladrilhos. Cada ladrilho envolve o cálculo da força resultante das influência das B partículas (colunas) nas partículas das A linhas. Cada ladrilho pode ser calculado de maneira independente pela GPU. Ao final de cada ladrilho, o resultado é enviado de volta à CPU e acumulado para posterior atualização das partículas.

Destaca-se que nesse passo, somente o cálculo da força é realizado. Enquanto na paralelização original no mesmo kernel a força resultante é calculada e a partícula é atualizada, este método precisa separar o passo da integração. Este passo é mais simples, e cada GPU somente precisa conhecer as partículas a serem atualizadas e as forças resultantes agindo sobre cada uma.

Quando todos os ladrilhos acabarem, vem a etapa de integração, onde somente se atualizam as propriedades de cada partícula. A cada nova sub-simulação na GPU, a mesma recebe todos os dados que precisa para seus cálculos naquele momento, ignorando a existência de outras partículas no sistema. O algoritmo 3 apresenta os passos do método em alto nível a partir do método em CPU.

Algoritmo 3 Algoritmo para a simulação de N-corpos.

```
function CPU(*pos, *vel, *newpos, *newvel, *accell, A, B, N, maxSimSteps)
    while ( dosimStep < maxSimSteps)
        for all row R ∈ N/A rows do                    ▷ Para cada conjunto de partículas
            for all tile T ∈ N/B tiles do in parallel   ▷ Receba um ladrilho
                CalculateInteractions(T)                ▷ Calcule o ladrilho
            end for
            IntegrateRow(R, newpos, newvel) ▷ Após todos os ladrilhos, atualize as partículas
        end for
        pos = newpos
        vel = newvel
        simStep++
    end while
end function
```

Até este ponto, foi definido como cada GPU processa uma sub-simulação, representada por um ladrilho. Como este método foi desenvolvido tendo em mente um cluster de GPUs, é necessário definir também seu comportamento no cluster. O mesmo comportamento da GPU, que não conhece todas as partículas a cada passo, pode então ser estendido também aos nós do cluster: cada nó só precisa ter em sua memória um subconjunto das partículas do sistema. Então, quando um nó receber um ladrilho para ser calculado, recebe também os dados das partículas dos nós que a contém.

Cada nó do cluster inicialmente é responsável por salvar em sua memória um subconjunto das partículas. Utilizando novamente o exemplo de 64 partículas, utilizando 4 nós, cada um receberia 16 partículas ao todo. A seguir, os ladrilhos são distribuídos para serem processados. A cada ladrilho, é necessário solicitar as partículas dos nós correspondentes, executar a sub-simulação na GPU e entregar o resultado para o nó correto para atualizar as partículas. Após o término do cálculo de todos os ladrilhos, passa-se para a etapa de integração, que atualiza as partículas. Estes passos se repetem para cada passo de simulação, conforme ilustrado pela Figura 3.3.

Esta abordagem cria a necessidade de comunicação constante entre nós. Na Fi-

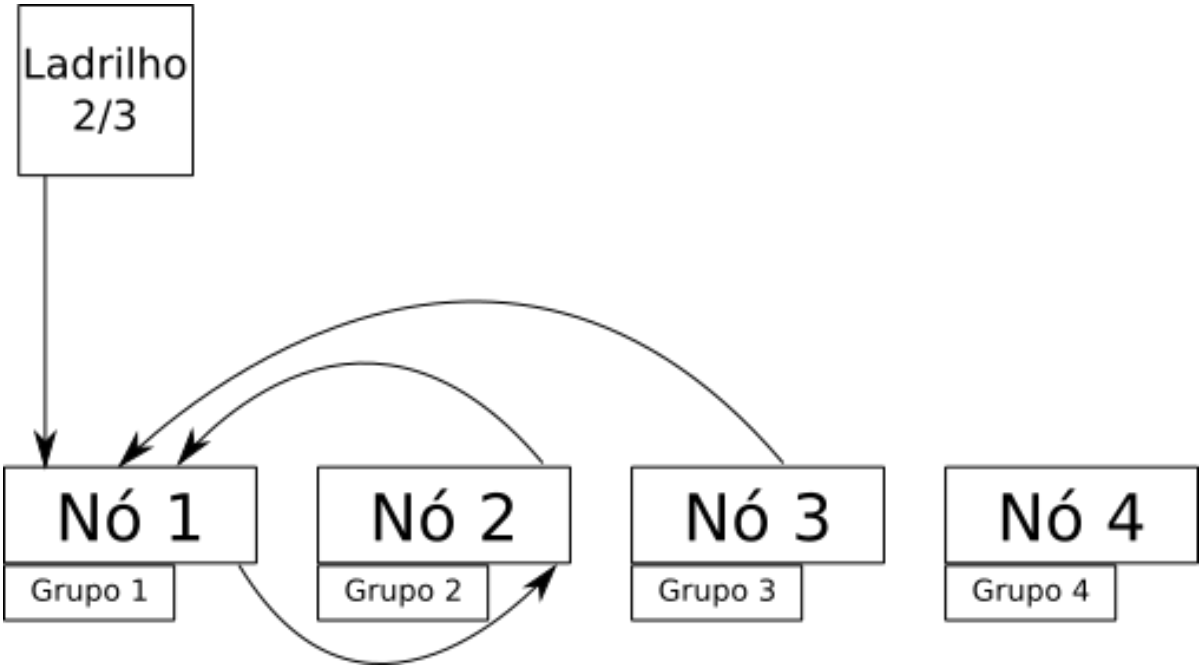


Figura 3.3: Esquema de divisão de trabalho entre GPUs. Cada nó do cluster carrega um conjunto de partículas. Quando receber algum ladrilho para calcular, recebe também os dados vindos dos nós que os contém. Ao final, devolve o valor calculado ao nó correto.

gura 3.3, o nó 1 recebe o ladrilho 2/3 para ser executado. Assim, ele precisa receber os dados das partículas dos nós 2 e 3, realizar o cálculo e repassar o resultado ao nó 2, que é o nó responsável pela atualização das partículas a serem calculadas por este ladrilho. Esta abordagem pode ser otimizada para explorar localidade de dados, diminuindo o tráfego de dados. É esta a abordagem utilizada no G-MPP e que será apresentada e discutida na Seção 3.2, a seguir.

3.2 Gerenciamento de Memória

O método descrito na subseção anterior pode ser organizado para aproveitar melhor a localidade dos dados durante sua execução. Esta subseção trata desta organização. Gerenciar eficientemente as trocas de mensagens entre nós do cluster e a transferência de dados para a GPU é papel vital para o desempenho deste método.

3.2.1 Comunicação entre nós do cluster

O método descrito na subseção anterior descreve o comportamento do método sem otimização em relação à quantidade de dados trafegada pela rede. Para o cluster de GPUs, é possível diminuir a quantidade de mensagens trafegadas entre os nós. Como todas as partículas precisam passar por todos os nós para atualização completa do sistema de partículas, podemos organizar o tráfego de memória criando um caminho para a passagem desses dados. Para isso, utilizamos uma estrutura de anel, conforme a Figura 3.4. Cada nó possui dois vizinhos: um vizinho anterior e um vizinho posterior. Cada nó carrega o seu grupo de partículas em memória. A seguir, recebe o grupo de partículas do nó anterior e envia seu grupo atual para o nó posterior. Assim, a cada novo grupo de partículas, uma sub-simulação é executada. Estes passos se repetem até que todos os grupos de partículas tenham passado por cada um dos nós. Assim, só há necessidade de enviar e receber um único conjunto de dados e o resultado de cada ladrilho pode ser armazenado sempre localmente, evitando comunicações desnecessárias.

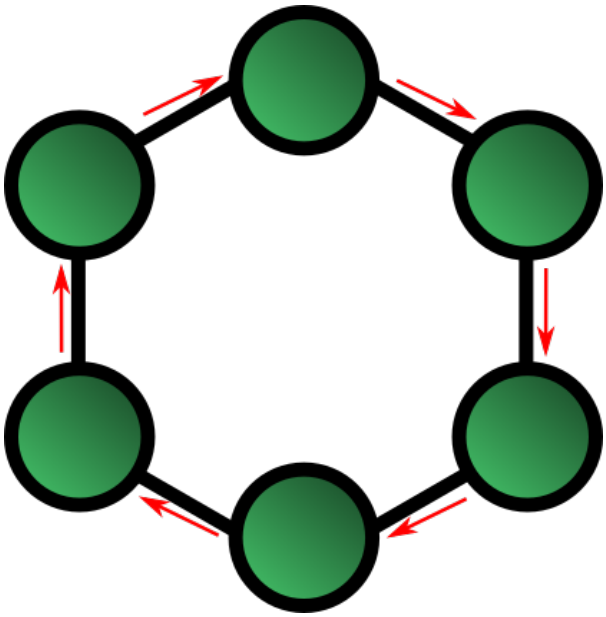


Figura 3.4: Estrutura da comunicação entre os nós em anel. Cada nó só se comunica com seus vizinhos anterior e posterior.

Fixar um conjunto de dados cria uma sequência para o cálculo dos ladrilhos. Cada nó (e consequentemente cada GPU) passa a ser responsável por calcular uma linha inteira do grid de ladrilhos. A Figura 3.5 demonstra isto. Cada GPU é responsável por toda a linha, e recebe as partículas dos outros nós para executar cada sub-simulação e atualizar totalmente o grupo local.

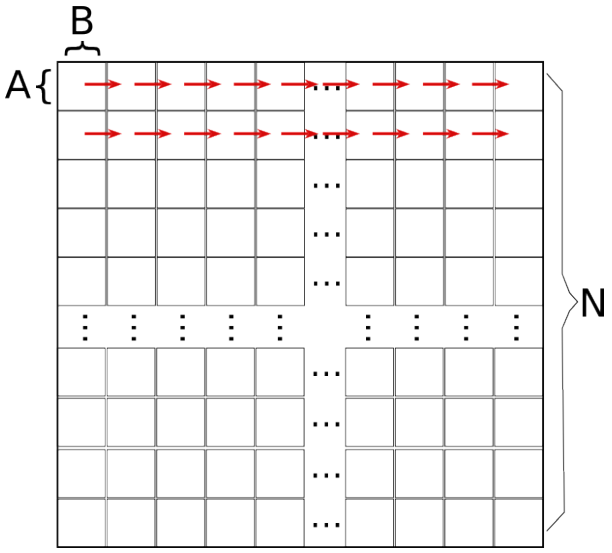


Figura 3.5: Sequência de atualização de cada ladrilho. Cada GPU calcula uma linha inteira de ladrilhos, atualizando as partículas do seu grupo local

Ao utilizar a localidade espacial dos dados, podemos adaptar o método Paricula-Partícula para cluster de GPU sem sobrecarregar a rede com tráfego exagerado de dados. Esta abordagem, juntamente com os fluxos de dados em GPU, apresentados a seguir, possibilita eficiência do G-MPP nas trocas de mensagens no cluster.

3.2.2 Fluxos de dados em GPU

Um fluxo CUDA (CUDA Stream) é uma sequência de operações que executam no dispositivo na ordem em que elas são colocadas no código. Enquanto as operações dentro de um fluxo são executadas garantidamente na ordem prescrita, as operações em fluxos diferentes podem ser intercaladas ou até mesmo executarem concorrentemente, quando

possível.

Por padrão, toda operação na GPU (tanto kernel quanto transferência de dados) são executados em um fluxo. Se nenhum fluxo é especificado, o fluxo padrão é utilizado. Ao se criar fluxos não-padrão, podemos intercalar e otimizar as cópias e execuções de kernel na GPU. Na arquitetura Fermi, até 16 kernels podem ser executados concorrentemente, enquanto 2 cópias podem ser realizadas simultaneamente, desde que seja cada uma em uma direção. Enquanto isso, a CPU também fica desbloqueada para processamento.

A Figura 3.6 ilustra a diferença entre o fluxo padrão e a utilização de múltiplos fluxos. Ao invés de esperar toda a cópia terminar, executar o código e novamente acessar os dados na GPU, pode-se quebrar a cópia em partes menores. Ao término de uma operação de cópia, um kernel é lançado para executar naquele espaço de memória, enquanto um segundo conjunto é copiado para a GPU. A seguir, esse segundo espaço de memória pode ser utilizado por outro kernel e o primeiro já pode ser copiado de volta. Esta interação pode diminuir significativamente o tempo de execução na GPU.

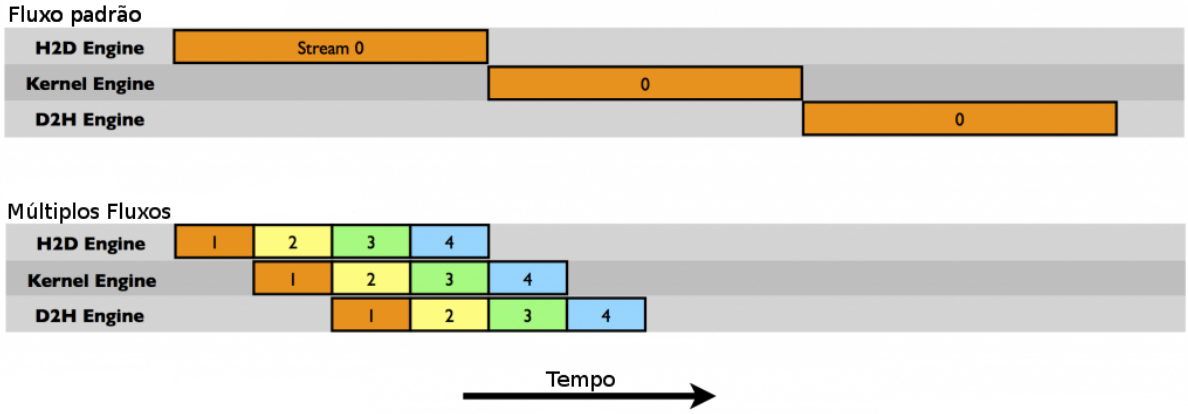


Figura 3.6: Diferença da execução com fluxo padrão e vários fluxos. Com os fluxos dividindo as operações, o kernel em um fluxo pode ser executado enquanto uma cópia é executada em outro.

No G-MPP, utilizamos esta característica para otimizar o tráfego de dados entre CPU

e GPU. Ao criarmos vários fluxos, calculamos forças parciais relativas a cada cópia, que vão sendo acumuladas pedaço a pedaço. Ao final de todas as cópias e execuções de kernel concorrentes, temos o resultado final esperado. No capítulo 4 avaliamos o impacto do uso de fluxos com detalhes.

Ao utilizar os fluxos de dados em GPU, a GPU não mais necessita de conhecer inicialmente todas as partículas e também pode executar concorrentemente cópias e execuções de kernel, otimizando ao máximo o uso da GPU. No Capítulo 4 é realizada uma análise do impacto desta abordagem no G-MPP.

3.3 Detalhes de Implementação

Esta seção discute detalhes da implementação do G-MPP, apresentando como ele faz a comunicação entre os nós e realiza a atualização das partículas.

Inicialmente, são necessários utiliza dois grupos de partículas e um vetor de aceleração, utilizado para computar a força resultante. Cada grupo de partículas são compostos de sua posição e velocidade (3D) e massa. O primeiro grupo de partículas representa o grupo de partículas que será atualizado pela GPU, e é denominado `local`. O segundo grupo representa as partículas que serão recebidas dos outros nós, para atualização das partículas locais, denominado então `temporário`. O Algoritmo 4 detalha mais a implementação do G-MPP.

Inicialmente, cada nó calcula a interação entre seu conjunto local de partículas com ele mesmo (linha 4 do algoritmo 4). Este passo pode ser realizado logo que o conjunto inicial de partículas for carregado, já que não depende de dados contidos fora do nó. Então temos o laço das sub-simulações com os dados externos, de acordo com a quantidade de

Algoritmo 4 Algoritmo para simulação de N-Corpos

```
1: function G-MPP(*local, *temporario, *newlocal, *aceleracao, qtNos, maxSimSteps)
2:     while (simStep < maxSimSteps) do
3:         LoadLocal(local)                                        ▷ Carrega o conjunto local
4:         CalculateInteractions(local, local, aceleracao)         ▷ Calcula a interação local-local
5:         i = 1
6:         while (i < qtNos) do
7:             sendTemporario()                                    ▷ Envia o temporário atual
8:             receiveTemporario(temporario)                       ▷ Recebe o novo temporário
9:             CalculateInteractions(local, temporario, aceleracao)
10:            i++
11:        end while
12:        stepSize = CalculateStepSize(aceleracao)                ▷ Calcula variáveis para integração
13:        IntegrateParticles(local, aceleracao, stepSize)         ▷ Atualiza as partículas
14:        simStep++
15:    end while
16: end function
```

nós disponíveis. A cada passo do laço, cada nó recebe um novo conjunto `temporário` do seu nó anterior (linha 7) e envia seu conjunto para o nó posterior (linha 8), dentro da estrutura de anel já apresentada. A seguir, o grupo `temporário` é enviado para a GPU, que calcula a interação entre as forças através da função `CalculateInteractions`(linha 9), detalhada no Algoritmo 5.

Algoritmo 5 Algoritmo para cálculo da interação das partículas

```
function CalculateInteractions(*local, *temporario, *aceleracao)
    tid = threadIdx
    myposicao = local.posicao[tid]
    acc = {0, 0, 0}
    for all (particles b_j ∈ temporario.posicao) do
        acc+ = bodyBodyInteraction(myposicao, b_j, acc)
    end for
    aceleracao[tid] = acc
end function
```

O Algoritmo 5 calcula a aceleração resultante acumulando as interações entre cada par de partículas, através da equação de Gravitação (2.4). Após calcular a força resultante, utilizando todas as partículas do sistema, é necessário agora atualizar as partículas em si. Para isso, são necessários dois passos, realizados pelas funções:

- `CalculateStepSize`
- `IntegrateParticles`

Na primeira função, `CalculateStepSize` (linha 12), cada nó envia para um nó central, pré-definido, quaisquer valores necessários para cálculo das variáveis necessárias para a integração. No exemplo implementado, do problema de gravitação, utilizamos o integrador Leap-Frog [25] que precisa de um cálculo a partir da menor aceleração entre todas. Assim, cada nó envia a menor aceleração encontrada para um nó central, que novamente filtra a menor de todas, realiza seus cálculos e envia para todos os outros nós a sua resposta.

Na segunda função, `IntegrateParticles` (linha 13), é que ocorre a integração em si. De posse de todas as variáveis necessárias, a GPU atualiza as características da partícula. No exemplo de gravitação, posição e velocidade são atualizadas. O Algoritmo 6 ilustra resumidamente o processo.

Algoritmo 6 Algoritmo para integração das partículas

function INTEGRATEPARTICLES($*local, *aceleracao, stepSize$)
  $tid = threadIdx$
  acc = aceleracao[tid]
  local.velocidade[tid] += $acc * stepSize$
  local.posicao[tid] += local.velocidade[tid]
end function

Portanto, o algoritmo deve enviar e receber os dados para execução e calcular a força acumulada, repetindo este passo até haver acessado todas as partículas do sistema. Após este passo, uma sincronização de variáveis acontece, devido ao método utilizado para atualização das partículas. Ao final é que ocorre a atualização das partículas propriamente dita, alterando sua posição e velocidade para o próximo passo da simulação.

Os resultados experimentais da organização em anel e do G-MPP serão apresentados e discutidos no capítulo a seguir.

3.4 Sumário

Este capítulo abordou a estratégia utilizada no G-MPP, para adaptação e otimização do método Partícula-Partícula de simulação de N-Corpos em clusters de GPUs. O método resolve as principais limitações das implementações já conhecidas: o limite de memória da GPU e a divisão de trabalho entre GPUs distribuídas em máquinas conectadas em rede.

Desta maneira, o G-MPP se mostra ideal para simulações massivas, onde o número de partículas é muito alto para armazenamento na memória da GPU e onde o tempo de execução deve ser reduzido ao máximo, porém sem as aproximações dos métodos hierárquicos.

No próximo capítulo as estratégias para otimização do gerenciamento de memória serão avaliadas e discutidas.

Capítulo 4

Avaliação Experimental

A avaliação do G-MPP ocorreu em dois passos. No primeiro, um problema sintético de amostra foi utilizado para analisar o comportamento padrão do algoritmo em um cluster de GPUs. Com estes dados em mãos, o G-MPP foi modelado e testado em diferentes cenários. Todos os testes foram realizados em um cluster de 4 nós equipados com processador Intel Core i5-3570 3.40GHz, 16GB de RAM e uma GPU GeForce GT640, ligados por uma rede Gigabit Ethernet dedicada.

4.1 Problema Amostra

Esta seção apresenta o problema amostra desenvolvido para avaliar o comportamento do algoritmo de simulação de N-Corpos em clusters de GPUs. A utilização de um problema amostra possibilita ilustrar as características mais importantes para a análise de desempenho do algoritmo, sem se ater ao resultado da simulação. Portanto o mesmo oferece um mecanismo interessante para observar o comportamento padrão da simulação de N-Corpos, permitindo modelar melhor a solução final. Portanto, o problema amostra é semelhante ao problema de N-Corpos, mas não é feita validação física dos resultados ao

final. O objetivo deste teste é observar o comportamento do algoritmo, sem prendê-lo a um problema específico, permitindo a adaptação a uma gama maior de aplicações. Com isso, podemos observar o comportamento do cluster em diferentes cenários.

Nesta arquitetura composta por várias GPUs e requisitando grande comunicação entre os nós e entre as GPUs, um fator determinante para eficiência é o tempo total de execução e a análise do quanto a comunicação e o processamento impactam neste tempo total. A hipótese é que há casos onde o tempo de rede pode ser insignificante em relação ao tempo de GPU, enquanto em outros casos, o tempo de tráfego de informação na rede vai dominar totalmente o tempo da GPU.

Para este trabalho, por tempo de rede se entende a soma dos tempos de envio de dados entre os nós do cluster (CPU-CPU) e entre o nó e sua GPU (CPU-GPU). Então, esta aplicação sintética foi desenvolvida para análise de dois parâmetros: a quantidade de memória transferida e a complexidade do processamento na GPU. Para isso, criamos 4 grupos de aplicações:

- Grupo 1: Tráfego baixo, complexidade da GPU baixa.
- Grupo 2: Tráfego baixo, complexidade da GPU alta.
- Grupo 3: Tráfego alto, complexidade da GPU baixa.
- Grupo 4: Tráfego alto, complexidade da GPU alta.

Para simular a demanda da rede, um buffer com valores aleatórios é enviado para cada nó. Quanto maior esse buffer, maior é a comunicação entre cada nó. Para a complexidade do código da GPU, utilizamos uma modificação do kernel da simulação,

que permite modificação na duração do laço de execução. Quanto mais vezes o laço é executado, maior o tempo gasto na GPU, simulando alta complexidade das operações. Com menos repetições do laço, simula-se uma tarefa mais simples na GPU.

O Algoritmo 7 apresenta como o problema amostra é avaliado. O passo 1, das linhas 4 à 5 do algoritmo coordena a troca de dados entre os nós, segundo a estrutura de anel, conforme apresentado na subseção 3.2.1. A seguir, no passo 2 (linha 6), o kernel é executado na GPU. Estes passos devem ser repetidos até que os dados de cada nó passem por todos os outros. O passo 3 (linha 9) envolve qualquer tipo de sincronização que seja necessária entre os nós. Esta etapa simula a necessidade de processamento conjunto de um dado pequeno de cada nó, como uma variável acumulada. Todos esses valores são enviados a um nó centralizador que será responsável pelo cálculo. A seguir, o resultado é enviado para todos os nós, que atualizam seus parâmetros de acordo. Por fim, no último passo (linha12), um segundo kernel de GPU é invocado, para terminar seu processamento com os parâmetros novos recebidos. Esta aplicação, mesmo sendo sintética, se comporta exatamente como o algoritmo 4, já apresentado anteriormente.

Algoritmo 7 Algoritmo do Problema Amostra

```
 1: function ToyProblem(memSize, gpuWLoad)
 2:     while !endSimulation do
 3:         while c_j < nnodes do
 4:             SendBufferToRight()                    ▷ Início do passo 1
 5:             ReceiveBufferFromLeft()                ▷ Fim do passo 1
 6:             CalculateIterations()                  ▷ Passo 2
 7:             c_j++
 8:         end while
 9:         SendDataToMaster()                         ▷ Início do passo 3
10:         BroadcastFromMaster()
11:         UpdateParameters()                         ▷ Fim do passo 3
12:         IntegrateParticles()                       ▷ Passo 4
13:     end while
14: end function
```

4.1.1 Avaliação do Problema Amostra

Nesta subseção apresentamos os resultados encontrados pela carga de trabalho mencionada acima. Para os testes, é necessário primeiro escolher os parâmetros para a demanda da rede e da GPU. Os parâmetros escolhidos estão na Tabela 4.1. Em relação aos parâmetros da GPU, os kernels utilizam a seguinte configuração: uma thread é lançada para cada byte do conjunto de dados de entrada, e cada bloco contém 256 threads.

Para a demanda de rede, inicialmente, o tamanho da GPU limita o tamanho máximo do bloco a ser trafegado pela rede. Na próxima seção ilustramos os testes para evitar este limite.

Para a demanda de GPU, quanto maior o contador do laço, maior a quantidade de cálculos a ser realizados. Os parâmetros foram escolhidos empiricamente, para executarem o mais próximo possível das aplicações reais.

	Baixa demanda	Alta demanda
Rede	50 MB	512 MB
GPU	10^4 iterações	10^5 iterações

Tabela 4.1: Parâmetros para baixa e alta demanda

Com estes parâmetros, os grupos detalhados anteriormente foram criados. Todas as simulações foram executadas 10 vezes, com 128 interações cada. Os tempos apresentados na Tabela 4.2 são a média por iteração de todas as execuções.

Os resultados permitem algumas considerações sobre o comportamento da simulação em um cluster. No caso onde se imaginava que o tempo de GPU poderia dominar o tempo de rede (grupo 2: baixa demanda de rede e alta de GPU), os tempos foram equivalentes. Como os dados estão divididos entre os nós, é perceptível que mesmo uma interface

Tempo (em s)	Total	Rede (%)	GPU (%)
Grupo 1	5.31	86.27	13.81
Grupo 2	12.34	40.84	59.16
Grupo 3	53.82	86.06	13.94
Grupo 4	122.92	39.47	60.53

Tabela 4.2: Resultados dos testes com o Problema Amostra. Todos os tempos estão em segundos e correspondem ao tempo médio por iteração do algoritmo.

Gigabit consegue suportar todo o tráfego: mesmo em simulações massivas dificilmente um único nó vai receber uma quantidade tão alta de dados que a porta não suporte e leve o tempo de comunicação a crescer não-linearmente. Isto pode ser visto pelos grupos 3 e 4, onde cada nó enviava 512MB de dados a cada vez. Ao invés disso, pode-se somente adicionar mais nós ao cluster, e com isso diminuir o impacto do tempo de tráfego na rede.

Considerando que o tempo de kernel depende não só da complexidade do mesmo, mas também da quantidade de dados a serem calculados, é perceptível o impacto da maior quantidade de dados na tabela. Ao aumentar a quantidade de dados trafegados, também o tempo de processamento da GPU subiu na mesma proporção.

O tempo de cópia dos dados entre CPUs e o tempo de execução do kernel podem ser sobrepostos, já que, enquanto a GPU está processando um conjunto de partículas, a CPU está livre para receber o próximo conjunto. Com isso, tem-se a sobreposição dos tempos de rede e execução da GPU. O máximo equilíbrio entre desempenho da rede e da GPU é alcançado quando os tempos de execução dos mesmos puderem ser sobrepostos, como aconteceu mais claramente no grupo 2 e com um pouco de desbalanceameto no grupo 4. Com isso, podemos observar que problemas com grande quantidade de dados podem se utilizar de kernels mais complexos na GPU sem impactar no resultado final

da execução. A partir disso, podemos otimizar as aplicações que pertençam ao grupo 3, com grande uso de rede, porém com baixa complexidade em GPU, dividindo os dados para diversos nós. Isso permite utilizar mais processamento da GPU, enquanto não se desequilibra a sobreposição dos tempos de rede e processamento.

Com estes resultados, pode-se observar também que, quando não for possível adicionar outro nó, o primeiro passo é explorar melhorias no tráfego de dados. Na parte de tráfego de dados, destaca-se principalmente o fluxo de dados entre CPU e GPU. A otimização desta parte de tráfego de dados será analisada na seção a seguir.

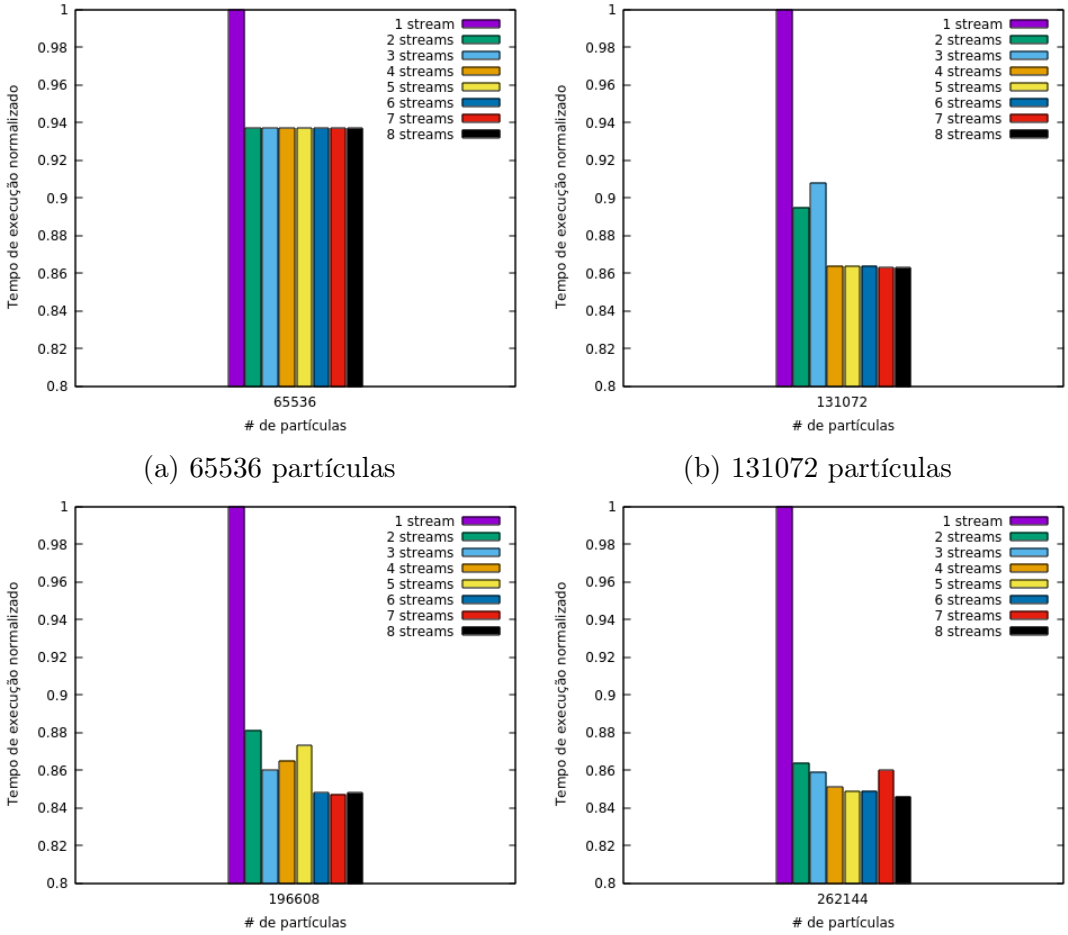
4.2 Análise de Fluxos de Dados em GPU

Neste teste, o objetivo é analisar o impacto de paralelizar as cópias de memória da CPU para a GPU através de fluxos CUDA (CUDA Streams). Para utilizar diversos fluxos, o conjunto de dados local é mantido na GPU por toda a execução, enquanto os subconjuntos temporários necessários para cada sub-simulação são copiados um a um para a GPU. O kernel realiza o cálculo da força resultante para este subconjunto, enquanto outro subconjunto é copiado para a GPU.

Com essa abordagem, cada kernel pode executar seu trabalho em menos tempo, ao permitir a sobreposição da execução dos kernels. Também há a diminuição da latência da GPU, ao permitir que execuções e cópias de memória sejam realizadas em paralelo. Este comportamento é semelhante ao visto na figura 3.6. Paralelamente, o consumo de memória também é controlado, já que a memória utilizada é proporcional ao número de fluxos utilizados.

Para este teste, a carga de trabalho varia de 65536 a 1048576 partículas. Inicial-

mente, para analisar o impacto da paralelização da cópia de memória através de fluxos concorrentes, a carga total é dividida em blocos contendo 32768 partículas cada. O número de fluxos varia de 1 a 8. Para este teste, os kernels utilizam a mesma configuração anterior: uma thread para cada byte do conjunto de dados de entrada, cada bloco com 256 threads. Os resultados estão apresentados na Figura 4.1 para as simulações até 262144 partículas e na Figura 4.2 para o restante das simulações teste.



(a) 65536 partículas

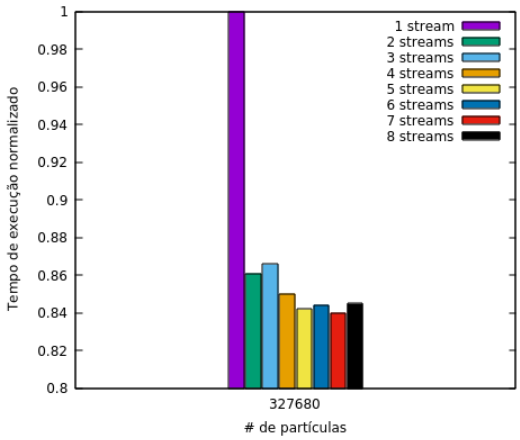
(b) 131072 partículas

(c) 196608 partículas

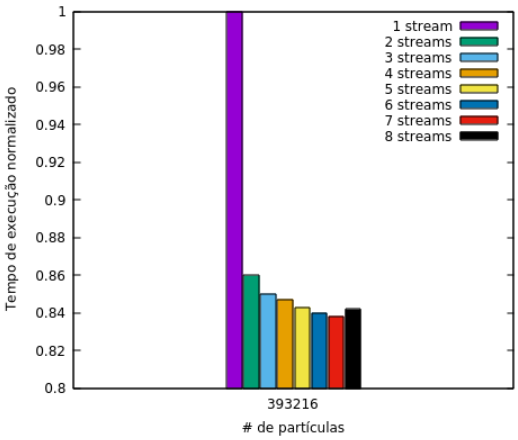
(d) 262144 partículas

Figura 4.1: Tempo de execução da simulação, variando o número de streams. Os tempos foram normalizados para facilitar a comparação entre cada configuração.

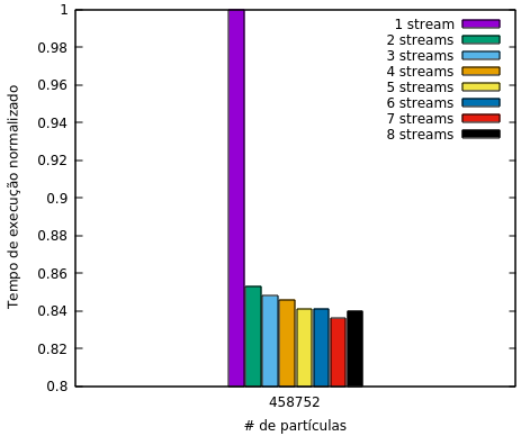
Este teste mostra que o uso de fluxos melhora o tempo de execução. Ao utilizarmos dois fluxos, os ganhos em relação a um único fluxo são significantes. A partir daí há



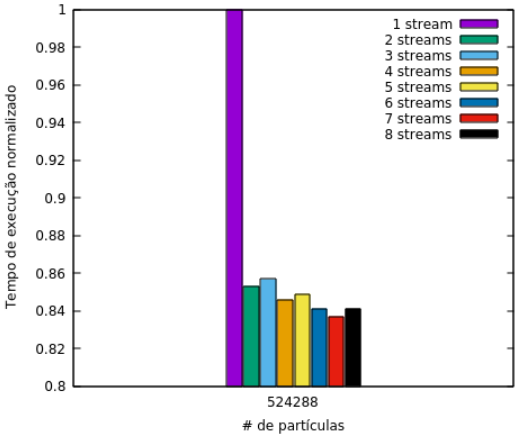
(a) 327680 partículas



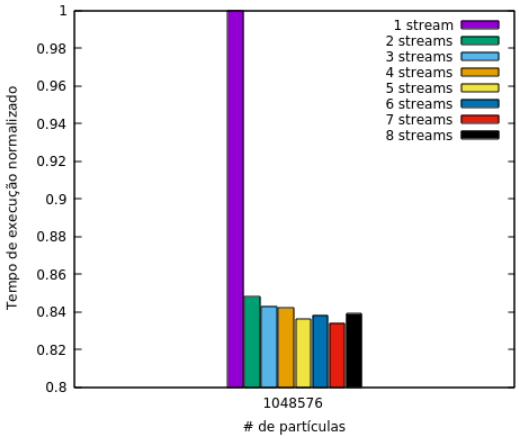
(b) 393216 partículas



(c) 458752 partículas



(d) 524288 partículas



(e) 1048576 partículas

Figura 4.2: Tempo de execução da simulação, variando o número de streams, para simulações entre 327680 e 1048576 partículas. Os tempos foram normalizados para facilitar a comparação entre cada configuração.

# de partículas	1 fluxo	4 fluxos	Melhora (%)
65536	1.000	0.937	6.724
131072	1.000	0.864	15.741
196608	1.000	0.864	15.741
262144	1.000	0.851	17.509
327680	1.000	0.850	17.647
393216	1.000	0.847	18.064
458752	1.000	0.846	18.203
524288	1.000	0.846	18.203
1048576	1.000	0.842	18.765

Tabela 4.3: Tempos normalizados de execução de um e quatro fluxos e a melhora alcançada. Podemos observar que quanto maior a quantidade de partículas, maior o ganho ao usar fluxos.

um ganho menor, especialmente nas simulações maiores. O ganho de desempenho nas simulações maiores ficaram entre 15 e 18%. Essa diferença vem da sobreposição do tempo de cópia de dados para a GPU e a execução de dados. Como destaque, a Tabela 4.3 mostra o ganho obtido ao aumentar de 1 para 4 fluxos.

Podemos observar pela tabela 4.3 que o ganho aumenta conforme o tamanho da simulação aumenta, demonstrando que o algoritmo se beneficia do uso dos fluxos de dados. Este comportamento vem da possibilidade de sobrepor operações de cópia e execuções de kernel, minimizando a latência da GPU. Como o algoritmo é mais demorado que a cópia de dados necessária para execução, o tempo de cópia fica completamente escondido por trás da execução do kernel. Há também o benefício de não sobrecarregar o barramento da GPU, garantindo um *throughput* constante de dados. Esta abordagem ainda permite realizar simulações massivas em uma única GPU, já que só é necessário o conjunto de dados para parte da simulação a cada chamada. Esta abordagem não diminui o tempo total da simulação, mas permite que a simulação seja limitada pela memória disponível no host.

4.3 Avaliação do G-MPP

Nesta seção discutimos os resultados da execução do G-MPP. Primeiramente, um teste variando o número de fluxos utilizados pela GPU foi realizado. Com o melhor caso escolhido, pudemos também comparar os resultados da execução com o código original do método Partícula-Partícula em GPU [46]. A configuração de execução dos kernels da GPU são as mesmas dos testes anteriores. Para proporcionar interpretação visual do método, no Anexo A apresenta diversas capturas em intervalos diferentes. Para melhor visualização, o exemplo apresentado contém somente 4096 partículas.

Na tabela 4.4 apresentamos os tempos de execução por iteração do método G-MPP variando o número de fluxos. Verifica-se que o melhor caso ocorreu com 7 fluxos para quase todos os casos, como já verificado na análise do impacto do uso de fluxos, na seção anterior. Verifica-se com o uso dos fluxos uma melhora de desempenho em média de 15%, comparando o caso com 1 e 7 fluxos.

A seguir, a tabela 4.5 mostra o ganho em desempenho em relação à aplicação original [46]. Mesmo em casos onde o número de partículas é pequeno (65536 ou 131072 partículas, por exemplo), o G-MPP já se comporta melhor. Em cenários mais complexos, com maior número de partículas, o G-MPP apresenta melhora ainda maior em relação à aplicação original, mostrando ser eficiente na solução de simulações massivas de N-Corpos.

Ao analisar o tempo de execução do G-MPP, observou-se também que a taxa de crescimento do tempo de execução alcançada é ligeiramente inferior à n^2, ordem de tempo esperada de acordo com a complexidade do algoritmo. A ordem esperada é representada

	1 fluxo	2 fluxos	3 fluxos	4 fluxos	5 fluxos	6 fluxos	7 fluxos	8 fluxos
65536 partículas	0,11	0,11	0,11	0,11	0,11	0,11	0,11	0,11
131072 partículas	0,45	0,40	0,40	0,39	0,38	0,38	0,38	0,38
196608 partículas	1,00	0,88	0,86	0,86	0,87	0,85	0,85	0,85
262144 partículas	1,77	1,53	1,52	1,51	1,51	1,51	1,53	1,50
327680 partículas	2,77	2,39	2,40	2,36	2,34	2,34	2,33	2,34
393216 partículas	3,99	3,43	3,39	3,38	3,36	3,35	3,34	3,36
458752 partículas	5,43	4,63	4,60	4,59	4,56	4,56	4,54	4,56
524288 partículas	7,09	6,05	6,07	6,00	6,02	5,96	5,93	5,96
1048576 partículas	28,32	24,01	23,86	23,85	23,69	23,73	23,62	23,76

Tabela 4.4: Tempo de execução por iteração (em segundos) do algoritmo variando o número de fluxos no G-MPP. A utilização do tempo por iteração permite verificar o desempenho do algoritmo independentemente das condições de parada, como estabilidade das partículas no sistema.

# de partículas	G-MPP	Original	Gap (%)
65536	0.105	0.195	-46.154
131072	0.384	0.775	-50.452
196608	0.847	1.724	-50.870
262144	1.526	3.068	-50.261
327680	2.329	4.772	-51.194
393216	3.342	6.880	-51.424
458752	4.539	9.336	-51.382
524288	5.933	12.222	-51.456
1048576	23.616	48.798	-51.605

Tabela 4.5: Tempo de execução por iteração (em segundos) do G-MPP e da aplicação original [46]. A última coluna indica o gap entre os tempos obtidos pelo G-MPP e pela aplicação original.

pela linha vermelha do gráfico. A linha verde exibe a taxa de crescimento alcançada pelo G-MPP. Com isso, pode se perceber o bom desempenho do G-MPP em realizar grandes simulações.

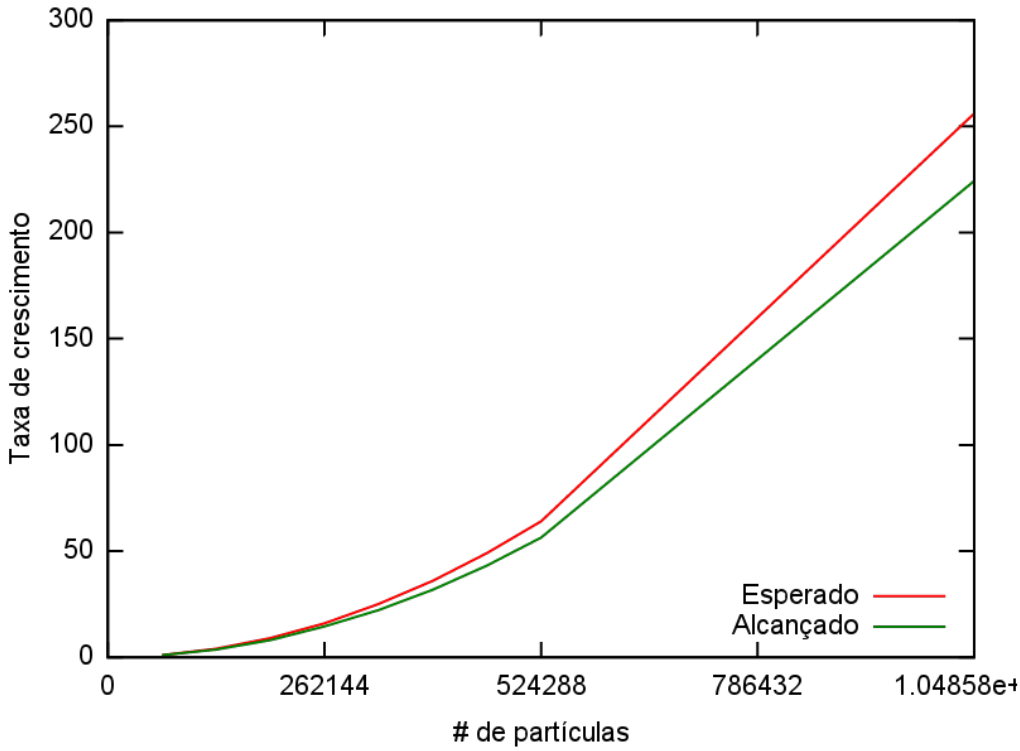


Figura 4.3: Taxa de crescimento esperada x alcançada. Nota-se que o G-MPP escala ligeiramente melhor do que a taxa esperada. Esta taxa melhora ainda mais à medida que a entrada cresce, mostrando que o algoritmo se comporta melhor sob cargas grandes de dados.

Capítulo 5

Conclusão

De forma geral, este trabalho apresentou um método de simulação de N-Corpos adaptado para cluster de GPUs. A gama de simuladores disponíveis na literatura não tratavam da simulação de N-Corpos em clusters de GPUs pelo método Partícula-Partícula que, embora com maior complexidade, traz resultados mais precisos. Enquanto os métodos hierárquicos ganham em desempenho, trazem arredondamentos que provocam pequenos erros acumulados na simulação. Com o desenvolvimento das GPUs, é possível voltar a métodos mais precisos, aproveitando o desempenho destas plataformas. Assim, foi desenvolvido o G-MPP, método adaptado para cluster de GPUs. O uso de clusters de GPUs cresceu nos últimos anos devido à necessidade cada vez maior de simulações e tarefas mais complexas, além da vantagem econômica das GPUs: permitem maior poder computacional por preço, consumo energético e uso de espaço físico menores.

O G-MPP, nas avaliações realizadas, obteve resultados satisfatórios, ao permitir que grandes simulações fossem realizadas em menos tempo que as soluções já encontradas na literatura. O maior limitador do método é a quantidade de dados trafegadas na rede entre os nós. Enquanto houver a possibilidade de manter essa quantidade baixa,

o método se comporta de maneira mais eficiente. Este problema pode ser facilmente contornado com a adição de mais nós ao cluster, o que levaria a uma divisão de tarefas maior entre os nós.

Vale destacar que mesmo no caso de simulações pequenas, em que uma única GPU seria suficiente, o método se comportou de maneira satisfatória, sempre com vantagem em relação ao método otimizado para uma GPU.

5.1 Trabalhos Futuros

Diversos outros trabalhos podem surgir a partir deste. Este método foi desenvolvido com o foco na aplicação de simulação gravitacional, porém outras simulações podem ser adaptadas para este método, como simulação de multidões ou de movimentação de robôs.

Alguns pontos podem ser ainda modificados no próprio método. A comunicação entre os nós pode seguir outros modelos, a serem estudados, de acordo com as características de cada problema a ser simulado. Além disso, um nó pode enviar e receber dados de múltiplos nós, gerando assim uma movimentação de memória em paralelo maior. O estudo nesse ponto é válido para verificar o impacto desta movimentação na rede.

Também podem ser desenvolvidos simuladores para uso em produção, especialmente na área de Física Computacional. Em Astrofísica, o estudo de movimentação de estrelas e galáxias não necessita de infraestrutura para ser realizada em tempo real, porém necessita de grande precisão matemática, já que a escala dos eventos é muito grande. Assim, este método pode ser utilizado para garantir desempenho e precisão para estas simulações. Pesquisas sobre o tema mostram que há publicações iniciais sobre o tema, como o grupo

de pesquisa do INPE, que busca examinar o efeito de deformações gravitacionais na evolução de estruturas de larga escala [55].

Referências

[1] Adams, B.; Pauly, M.; Keiser, R.; Guibas, L. J. Adaptively sampled particle fluids. In *ACM Transactions on Graphics (TOG)* (2007), vol. 26, ACM, p. 48.

[2] Aichelin, J. "quantum" molecular dynamics—a dynamical microscopic n-body approach to investigate fragment formation and the nuclear equation of state in heavy ion collisions. *Physics Reports 202*, 5 (1991), 233–360.

[3] Aila, T.; Laine, S.; Karras, T. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02* (2012).

[4] Alligood, K. T.; Sauer, T. D.; Yorke, J. A. *Chaos*. Springer, 1996.

[5] Barnes, J.; Hut, P. A hierarchical O(N log N) force-calculation algorithm. *Nature Publishing Group 324*, 6096 (1986), 446–449.

[6] Bédorf, J.; Gaburov, E.; Portegies Zwart, S. A sparse octree gravitational< i> n</i>-body code that runs entirely on the gpu processor. *Journal of Computational Physics 231*, 7 (2012), 2825–2839.

[7] Berendsen, H. J.; van der Spoel, D.; van Drunen, R. Gromacs: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications 91*, 1 (1995), 43–56.

[8] Brandao, D.; Zamith, M.; Clua, E.; Montenegro, A.; Bulcao, A.; Madeira, D.; Kischinhevsky, M.; Leal-Toledo, R. C. Performance evaluation of optimized implementations of finite difference method for wave propagation problems on gpu architecture. In *Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2010 22nd International Symposium on* (2010), IEEE, pp. 7–12.

[9] Cevahir, A.; Nukada, A.; Matsuoka, S. Fast conjugate gradients with multiple gpus. In *Computational Science–ICCS 2009*. Springer, 2009, pp. 893–903.

[10] Cornell Theory Center. SPMD or Manager/Worker. `http://web0.tc.cornell.edu/Services/Education/Topics/Parallel/Design/SPMD.aspx`. Acessado em: 21-02-2015.

[11] Courty, N.; Raupp Musse, S. Simulation of large crowds in emergency situations including gaseous phenomena. In *Computer Graphics International 2005* (2005), IEEE, pp. 206–212.

[12] Cuda, C. *Programming guide*, 2012.

[13] Darden, T.; York, D.; Pedersen, L. Particle mesh ewald: An n log (n) method for ewald sums in large systems. *The Journal of chemical physics 98*, 12 (1993), 10089–10092.

[14] Du, P.; Weber, R.; Luszczek, P.; Tomov, S.; Peterson, G.; Dongarra, J. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing 38*, 8 (2012), 391–407.

[15] Duncan, R. A survey of parallel computer architectures. *Computer 23*, 2 (1990), 5–16.

[16] Enos, J.; Steffen, C.; Fullop, J.; Showerman, M.; Shi, G.; Esler, K.; Kindratenko, V.; Stone, J. E.; Phillips, J. C. Quantifying the impact of gpus on performance and energy efficiency in hpc clusters. In *Green Computing Conference, 2010 International* (2010), IEEE, pp. 317–324.

[17] Fatica, M.; Luebke, D. High Performance Computing with {CUDA}. Supercomputing 2007 tutorial. In Supercomputing 2007 tutorial notes, 2007.

[18] Felix Soehr, S. D. M. W. *Simulation of Galaxy Fomation and Large Scale Structure*. Phd thesis, Ludwig-Maximilian Universitat Munchen, 2003.

[19] Fernando, R.; Kilgard, M. J. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[20] Flynn, M. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on 100*, 9 (1972), 948–960.

[21] Gaburov, E.; Harfst, S.; Zwart, S. P. SAPPORO: A way to turn your graphics cards into a GRAPE-6. *New Astronomy 14*, 7 (2009), 630–637.

[22] Grama, A. Y.; Kumar, V.; Sameh, A. Scalable parallel formulations of the barnes-hut method for n -body simulations. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing - Supercomputing '94* (New York, New York, USA, 1994), ACM Press, p. 439.

[23] Group, K. O. W., et al. The opencl specification. *version 1*, 29 (2008), 8.

[24] Habermaier, A.; Knapp, A. On the correctness of the simt execution model of gpus. In *Programming Languages and Systems*. Springer, 2012, pp. 316–335.

[25] Hairer, E.; Lubich, C.; Wanner, G. Geometric numerical integration illustrated by the störmer-verlet method, 2003.

[26] Hamada, T.; Iitaka, T. The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units. *arXiv preprint astro-ph/0703100* (Mar. 2007).

[27] Hamada, T.; Narumi, T.; Yokota, R.; Yasuoka, K.; Nitadori, K.; Taiji, M. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 62.

[28] Hockney, R. W. *Computer Simulation Using Particles.* McGraw-Hill, 1988.

[29] Holmberg, E. On the clustering tendencies among the nebulae. ii. a study of encounters between laboratory models of stellar systems by a new integration procedure. *The Astrophysical Journal 94* (1941), 385.

[30] Intel. Intel Core Processors in the LGA2011 and LGA1150 Sockets. `http://www.intel.com.br/content/dam/www/public/us/en/documents/platform-briefs/4th-gen-core-desktops-brief.pdf`. Acessado em 21.02.2015.

[31] Jetley, P.; Wesolowski, L.; Gioachin, F.; Kalé, L. V.; Quinn, T. R. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[32] Kadlec, B.; Tufo, H.; Dorn, G. Knowledge-assisted visualization and segmentation of geologic features using implicit surfaces.

[33] Kindratenko, V. V.; Enos, J. J.; Shi, G.; Showerman, M. T.; Arnold, G. W.; Stone, J. E.; Phillips, J. C.; Hwu, W.-m. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), IEEE, pp. 1–8.

[34] Lashuk, I.; Chandramowlishwaran, A.; Langston, H.; Nguyen, T.-A.; Sampath, R.; Shringarpure, A.; Vuduc, R.; Ying, L.; Zorin, D.; Biros, G. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), Proceedings of the 2009 ACM/IEEE conference on Supercomputing - Supercomputing '09, ACM, pp. 58:1–58:12.

[35] Leimanis, E.; Minorsky, N. Dynamics and nonlinear mechanics. some recent advances in the dynamics of rigid bodies and celestial mechanics. *Dynamics and nonlinear mechanics. Some recent advances in the dynamics of rigid bodies and celestial mechanics, by Leimanis, E.; Minorsky, Nicholas. New York, Wiley [1958]. Surveys in applied mathematics; 2 1* (1958).

[36] Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J. Nvidia tesla: A unified graphics and computing architecture. *Ieee Micro 28*, 2 (2008), 39–55.

[37] Makino, J.; Taiji, M. *Scientific Simulations with Special-Purpose Computers–the GRAPE Systems*, vol. 1. Wiley, Apr. 1998.

[38] Math.nist.gov. Nist sp2 primer: Distributed-memory programming. `http://math.nist.gov/~KRemington/Primer/distrib.html`. Acessado em: 21.02.2015.

[39] Meyer, K.; Hall, G.; Offin, D. *Introduction to Hamiltonian Dynamical Systems and the N-Body Problem: A Mechanised Logic of Computation*, vol. 90. Springer Science & Business Media, 2008.

[40] Muyan-Ozcelik, P.; Owens, J. D.; Xia, J.; Samant, S. S. Fast deformable registration on the gpu: A cuda implementation of demons. In *Computational Sciences and Its Applications, 2008. ICCSA'08. International Conference on* (2008), IEEE, pp. 223–233.

[41] Nakasato, N. Oct-tree Method on GPU : $ 42/Gflops Cosmological Simulation. *arXiv preprint arXiv:0909.0541* (2009).

[42] Newton, I. *Newton's Principia : the mathematical principles of natural philosophy*, vol. 1. New-York : Published by Daniel Adee, 1846.

[43] Nukada, A.; Maruyama, Y.; Matsuoka, S. High performance 3-d fft using multiple cuda gpus. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units* (2012), ACM, pp. 57–63.

[44] NVidia. Tesla K20 GPU Active Accelerator Board Specification. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v02.pdf. Acessado em: 21.02.2015.

[45] Nyland, L.; Harris, M.; Prins, J. N-body simulations on a gpu. In *ACM Workshop on General-Purpose Computing on Graphics Processors* (2004), A. Press, Ed., pp. C–37.

[46] Nyland, L.; Harris, M.; Prins, J. Fast n-body simulation with cuda. *GPU gems 3*, 1 (2007), 677–696.

[47] Ooi, B. C.; McDonell, K. J.; Sacks-Davis, R. Spatial kd-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC* (1987), vol. 87, p. 85.

[48] Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (2007), vol. 26, Wiley Online Library, pp. 80–113.

[49] Portegies Zwart, S.; Mcmillan, S.; Nualláin, B. O.; Heggie, D.; Lombardi, J.; Hut, P.; Banerjee, S.; Belkus, H.; Fragos, T.; Fregeau, J.; Fuji, M.; Gaburov, E.; Glebbeek, E.; Groen, D.; Harfst, S.; Izzard, R.; Jurić, M.; Justham, S.; Teuben, P.; Bever, J.; Yaron, O.; Zemp, M. A multiphysics and multiscale software environment for modeling astrophysical systems. *New Astronomy 14*, 4 (2009), 369–378.

[50] Press, W. H.; Schechter, P. Formation of galaxies and clusters of galaxies by self-similar gravitational condensation. *The Astrophysical Journal 187* (1974), 425–438.

[51] Quinn, M. J. *Parallel computing: theory and practice*, vol. 8. McGraw-Hill New York, 1994.

[52] Rosenberg, R. On nonlinear vibrations of systems with many degrees of freedom. *Advances in applied mechanics 9* (1966), 155–242.

[53] Springel, V. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society 364*, 4 (Dec. 2005), 1105–1134.

[54] Stalder, D. H. Um novo simulador de n-corpos para cosmologia computacional utilizando tecnologia GPU/CUDA. Dissertação de mestrado, Instituto Nacional de Pesquisas Espaciais, 2013.

[55] Stalder, D. H.; Rosa, R. R.; da Silva Junior, J. R.; Clua, E.; Ruiz, R. S. R.; Velho, H. F. C.; Ramos, F. M.; Araújo, A. D. S.; Conrado, V. G. A new gravitational n-body simulation algorithm for investigation of cosmological chaotic advection. *THE SIXTH INTERNATIONAL SCHOOL ON FIELD THEORY AND GRAVITATION-2012. AIP Conference Proceedings 1483* (2012), 447–452.

[56] Stock, M. J.; Gharakhani, A. Toward efficient gpu-accelerated n-body simulations. *AIAA paper 608* (2008), 7–10.

[57] Strohmaier, E.; Dongarra, J.; Simon, H.; Meuer, M.; Meuer, H. Top500 Supercomputing sites. `http://www.top500.org/lists/2014/11/`.

[58] Tanenbaum, A. S.; Zucchi, W. L. *Organização estruturada de computadores.* Pearson Prentice Hall, 2009.

[59] Van Den Berg, J.; Guy, S. J.; Lin, M.; Manocha, D. Reciprocal n-body collision avoidance. In *Robotics research.* Springer, 2011, pp. 3–19.

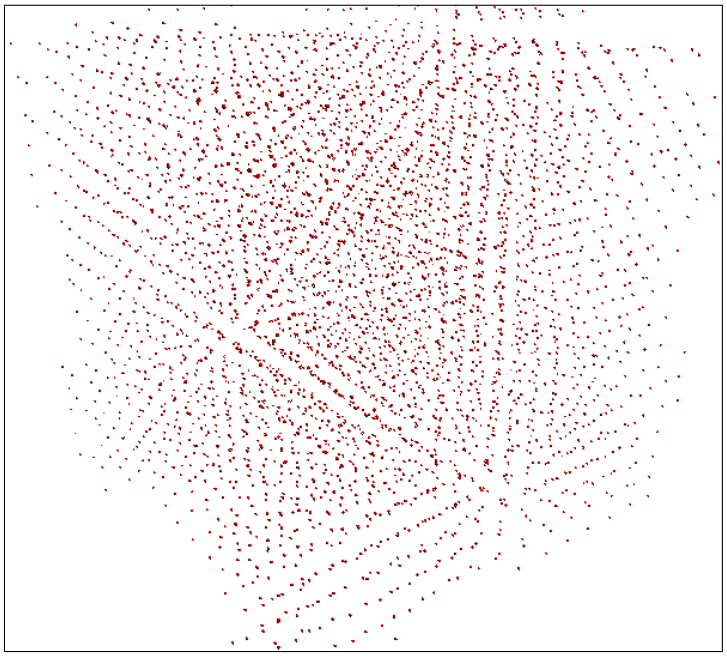
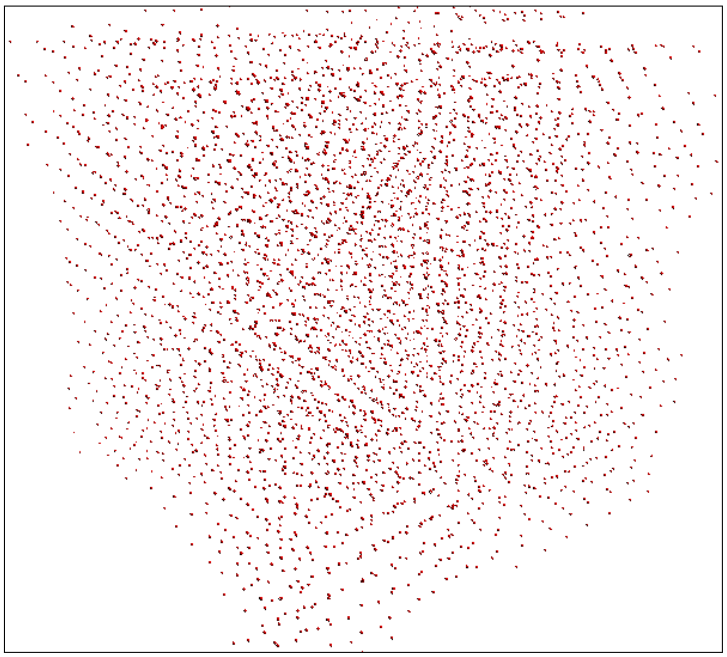
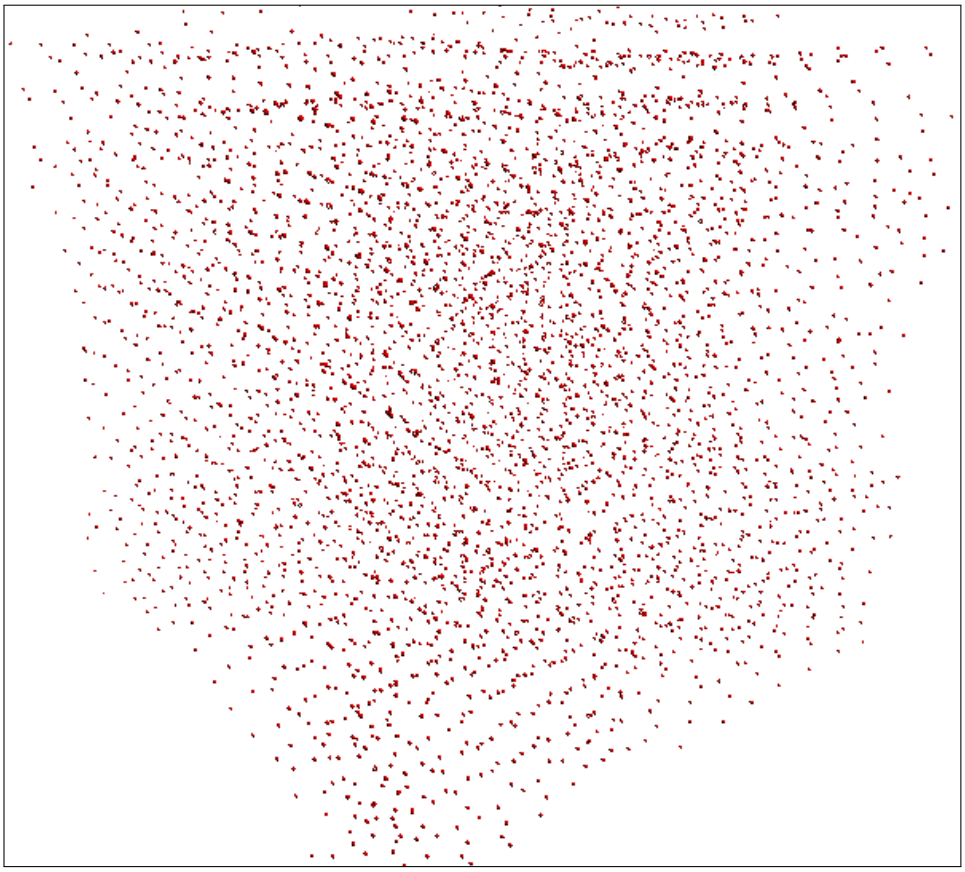
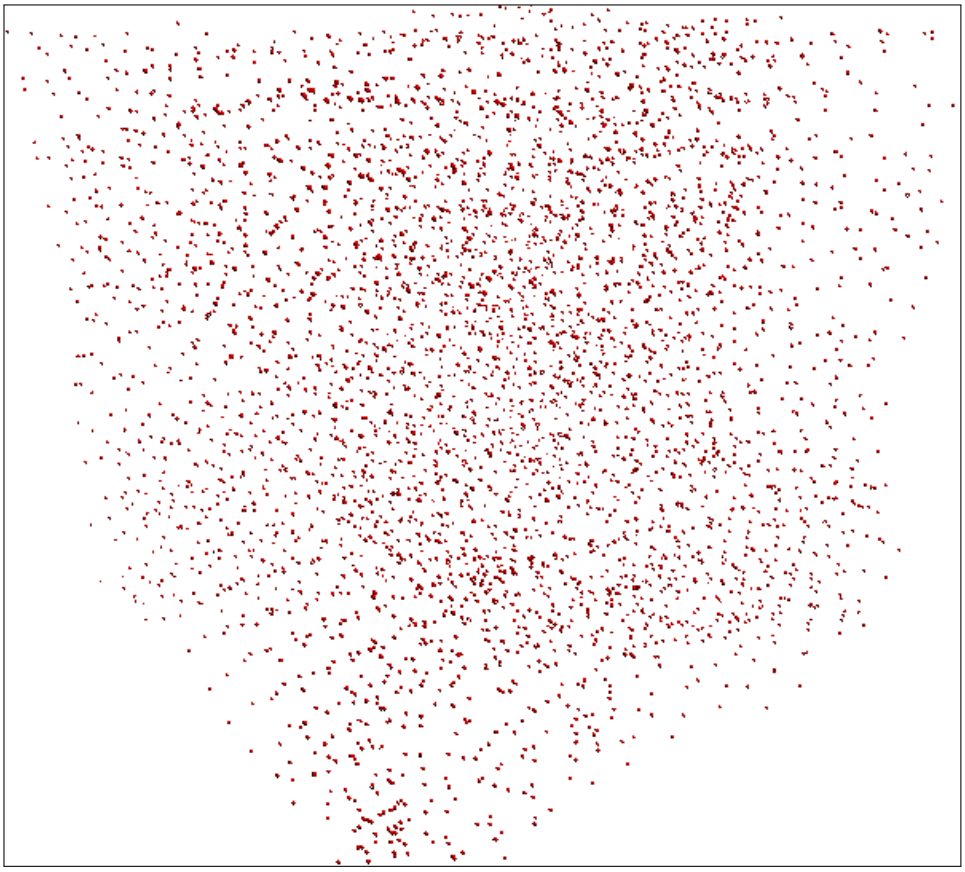
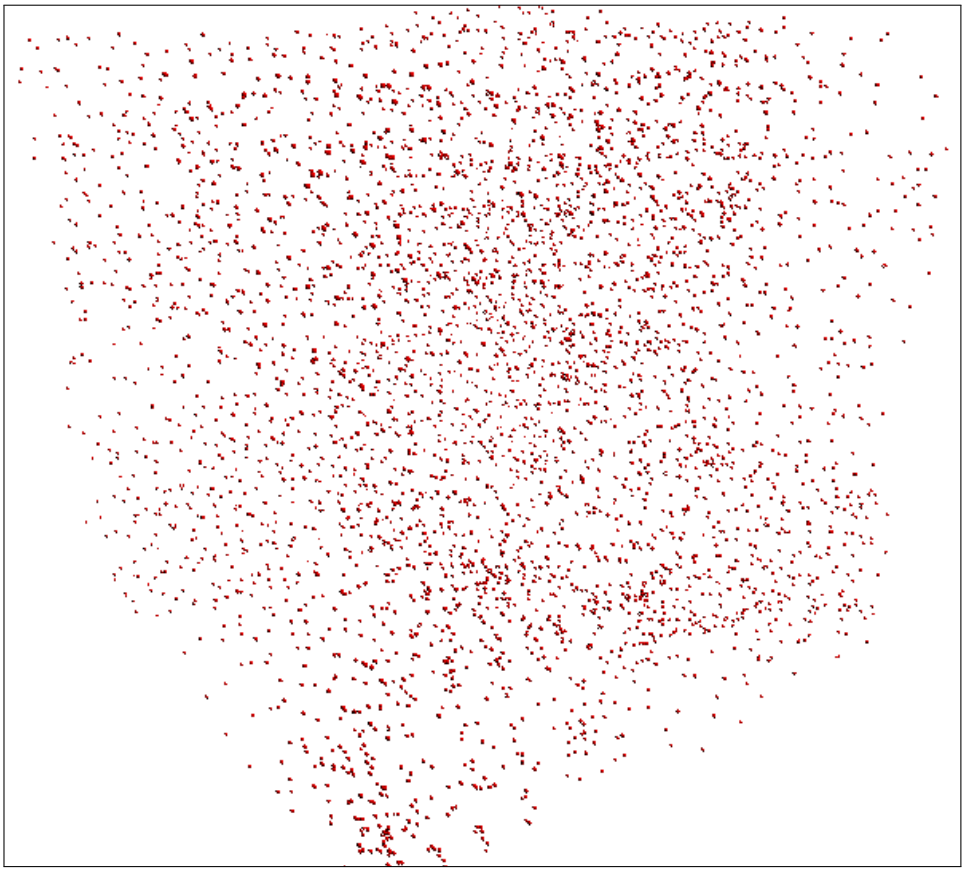
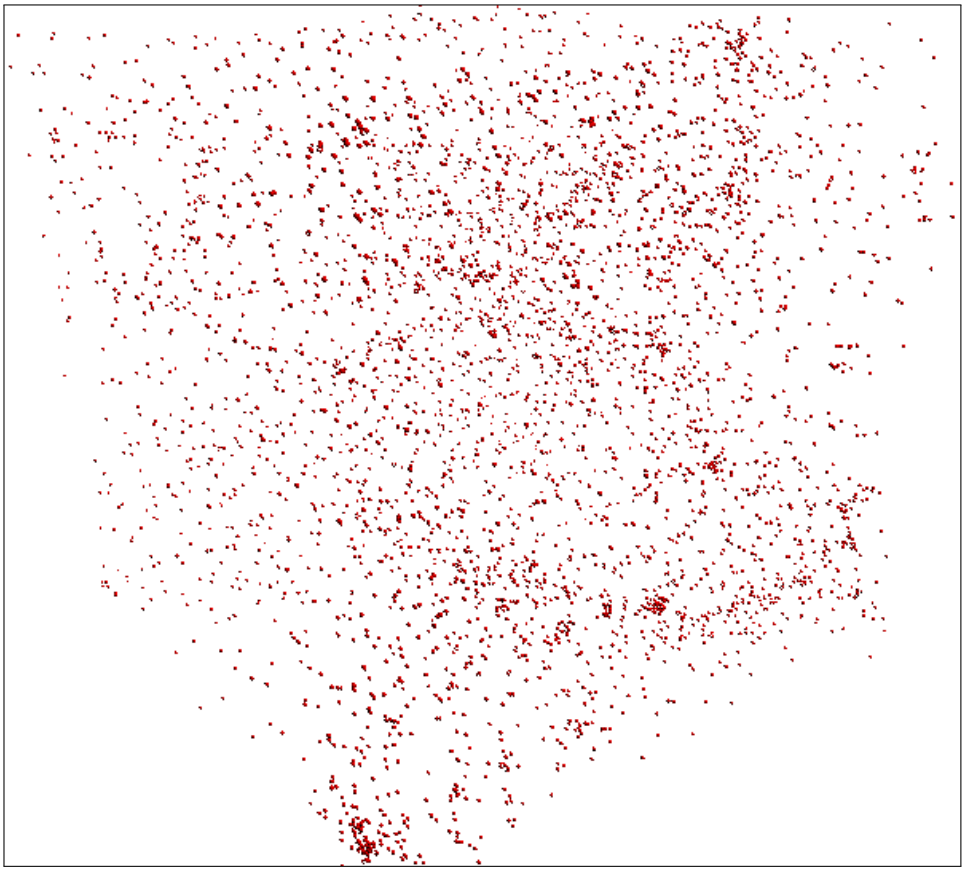
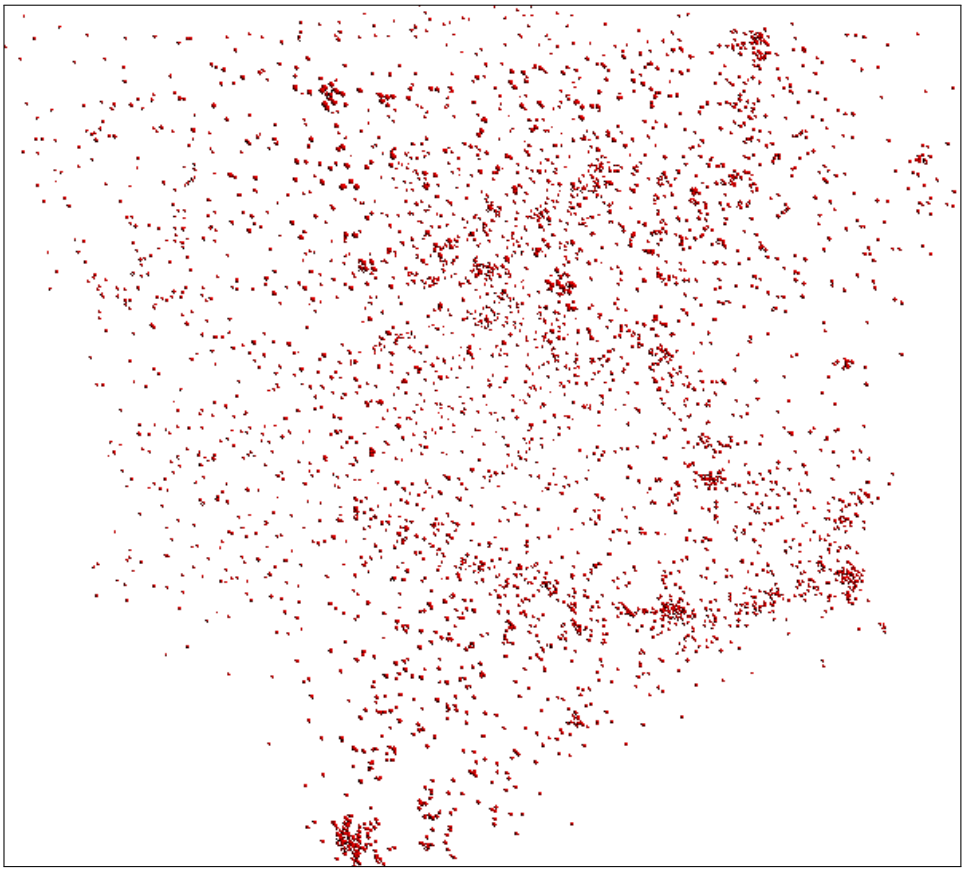
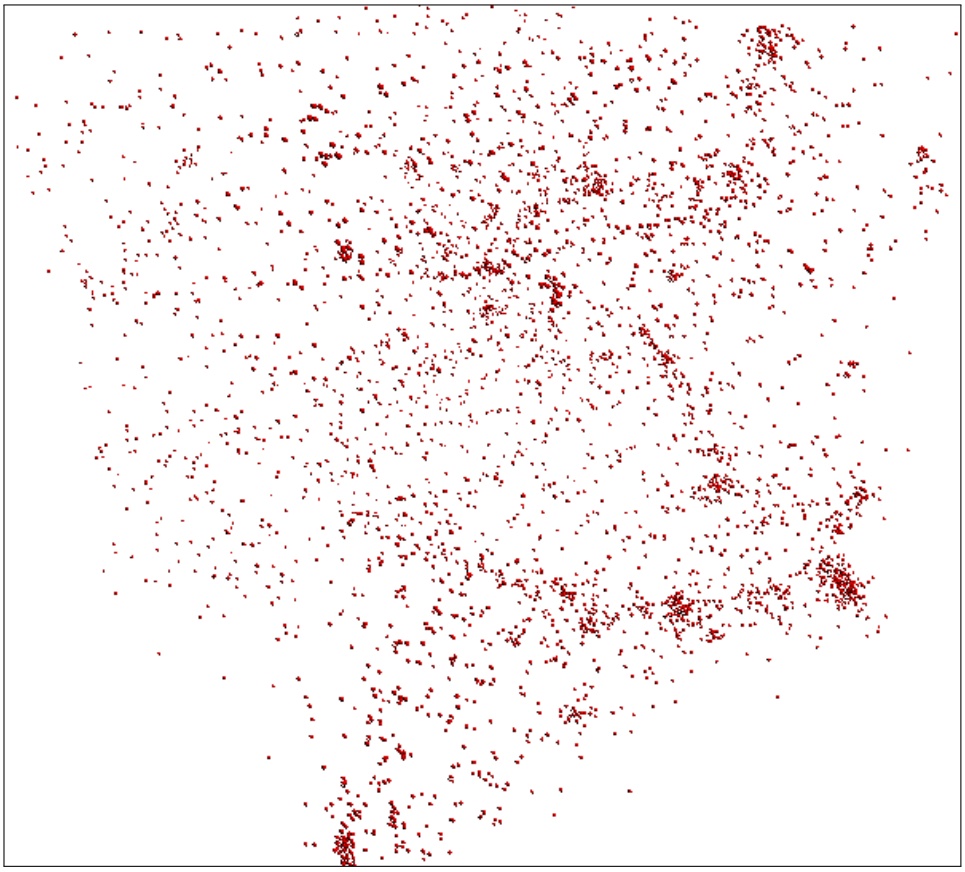
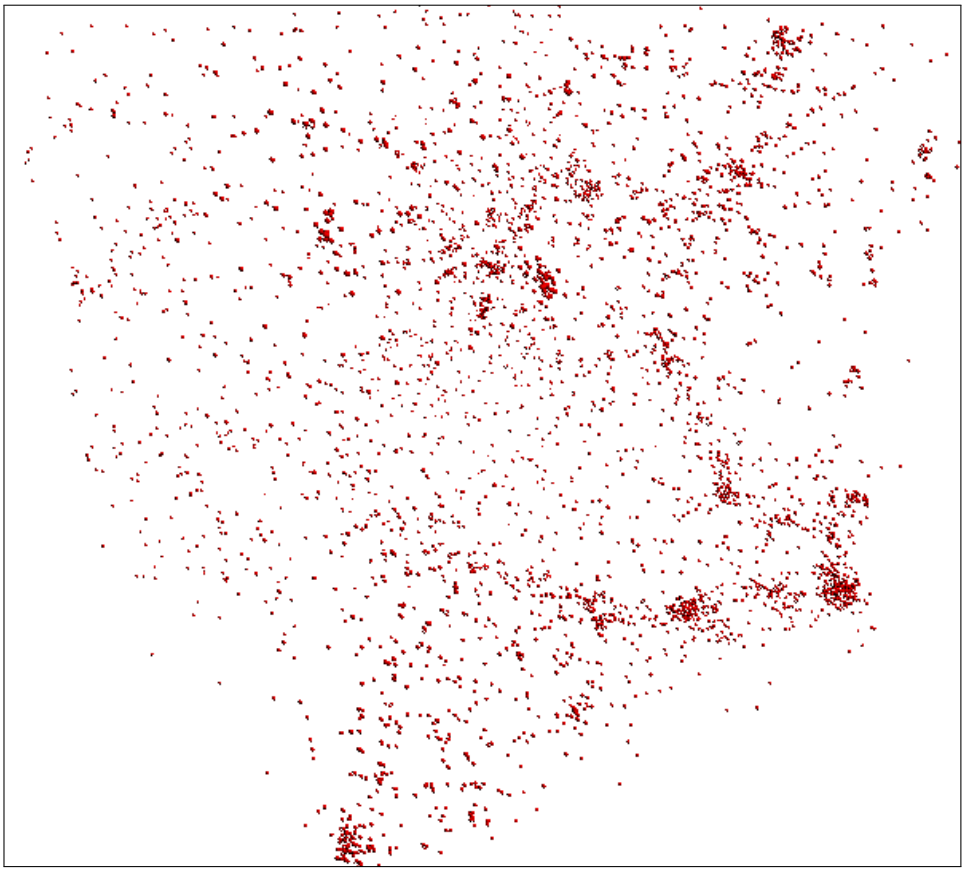
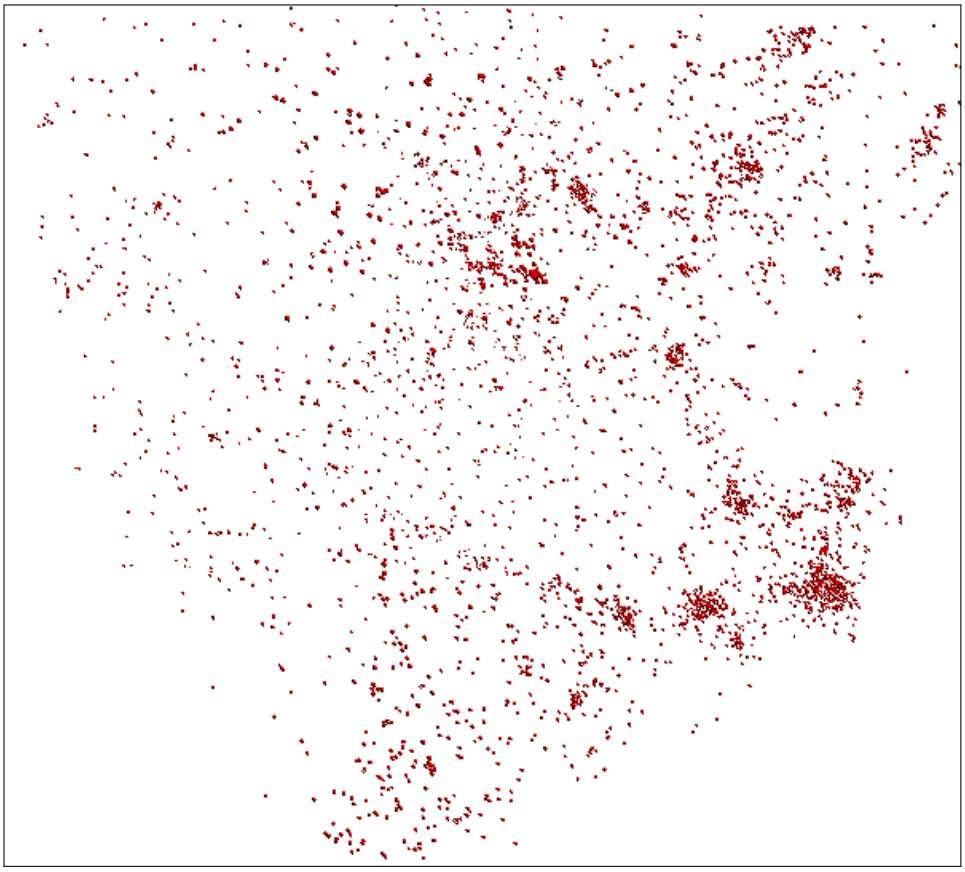
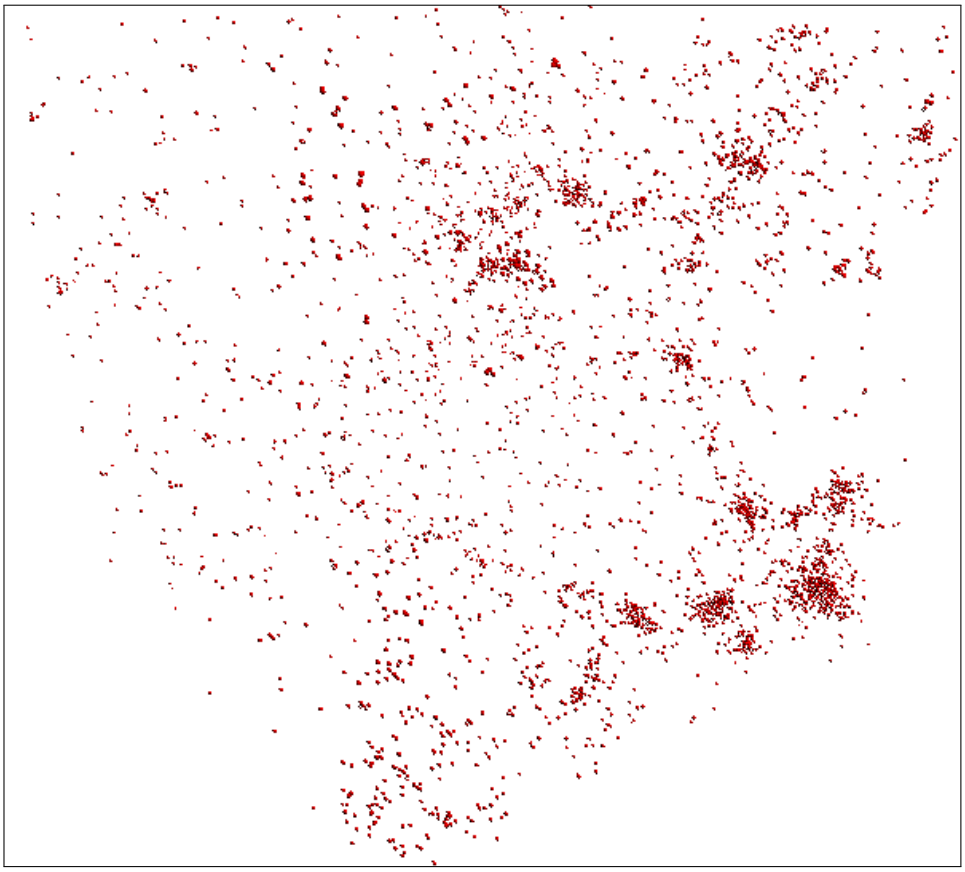
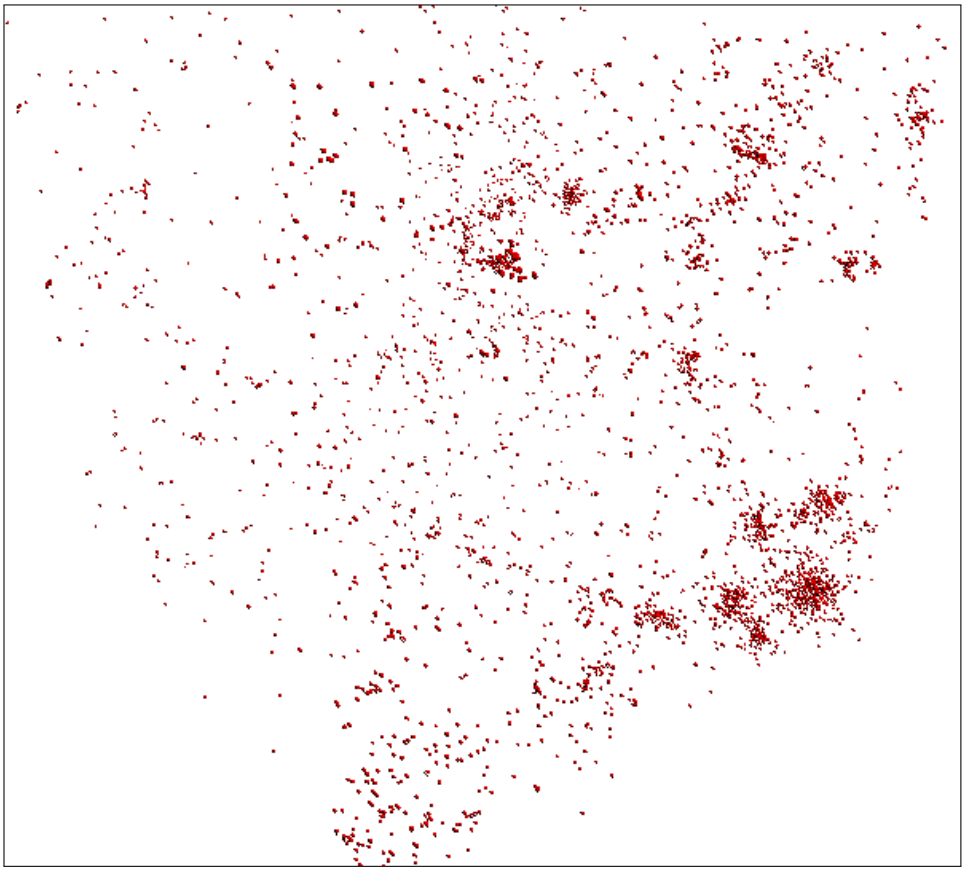
[60] von Hoerner, S. Die numerische integration des n-körper-problemes für sternhaufen. i. *Z. Astrophys. 50* (1960), 184–214.

[61] Wald, I.; Havran, V. On building fast kd-trees for ray tracing, and on doing that in o (n log n). In *Interactive Ray Tracing 2006, IEEE Symposium on* (2006), IEEE, pp. 61–69.

[62] Walker, J. S. *Fast fourier transforms*, vol. 24. CRC press, 1996.

[63] Warren, M.; Salmon, J. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing* (1992), IEEE Computer Society Press, pp. 570–576.

[64] White, S. D. The dynamics of rich clusters of galaxies. *Monthly Notices of the Royal Astronomical Society 177*, 3 (1976), 717–733.

[65] Zhou, B.; Zhou, S. Parallel simulation of group behaviors. In *Proceedings of the 36th conference on Winter simulation* (2004), Winter Simulation Conference, pp. 364–370.

APÊNDICE A – Capturas de Tela do Simulador