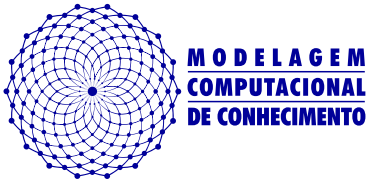


Dissertação de Mestrado

Avaliação de Desempenho de Algoritmos Paralelos de Busca de Vizinhos em Cenários com Distribuições Espaciais Distintas

Bruno Normande Lins
normandelins@gmail.com

Maceió, Outubro de 2016

Bruno Normande Lins

Avaliação de Desempenho de Algoritmos Paralelos de Busca de Vizinhos em Cenários com Distribuições Espaciais Distintas

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Modelagem Computacional de Conhecimento do Instituto de Computação da Universidade Federal de Alagoas.

Orientador: Leonardo Viana

Maceió, Outubro de 2016

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável: Helena Cristina Pimentel do Vale

L759a Lins, Bruno Normande.
Avaliação de desempenho de algoritmos paralelos de busca de vizinhos em cenários com distribuições especiais distintas / Bruno Normande Lins, 2016.
49 f. : il.

Orientador: Leonardo Viana.
Dissertação (mestrado em Modelagem Computacional de Conhecimento) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2016.

Bibliografia: f. 49.

1. Processamento paralelo (Computadores). 2. Método dos elementos discretos. 3. Detecção de contato. 4. Busca por vizinhos. I. Título.

CDU: 004.272

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Modelagem Computacional de Conhecimento
Avenida Lourival Melo Mota, Km 14, Bloco 09, Cidade Universitária
CEP 57.072-900 – Maceió – AL – Brasil
Telefone: (082) 3214-1364

Membros da Comissão Julgadora da Dissertação de Mestrado de Bruno Normande Lins, intitulada: "Avaliação de Desempenho de Algoritmos Paralelos de Busca de Vizinhos em Cenários com Distribuições Espaciais Distintas", apresentada ao Programa de Pós-Graduação em Modelagem Computacional de Conhecimento da Universidade Federal de Alagoas, em 25 de novembro de 2016, às 15h00min, no auditório do Instituto de Computação da Ufal.

COMISSÃO JULGADORA

Prof. Dr. Leonardo Viana Pereira
Ufal – Instituto de Computação
Orientador

Prof. Dr. André Luiz Lins de Aquino
Ufal – Instituto de Computação
Examinador

Prof. Dr. Orivaldo Vieira de Santana Júnior
UFRN – Escola de Ciências e Tecnologia
Examinador

Maceió, novembro de 2016.

RESUMO

Algoritmos de detecção de contatos são necessários em diferentes áreas da ciência e tecnologia, de jogos digitais e computação gráfica à simulações de alto desempenho e robótica. Esses algoritmos exigem grande esforço computacional e tendem a ser os gargalos das aplicação as quais fazem parte, principalmente em sistemas de grande escala ou em tempo real. Com a popularização das placas GPUs para uso científico e comercial, é natural que surjam implementações paralelas para esse problema. Nesse trabalho os principais algoritmos de detecção de contatos para GPU são analisados e é realizado um experimento numérico, com objetivo de descobrir qual algoritmo é o melhor em termos de desempenho computacional e uso de memória, ou se a eficiência de cada um depende das diferentes características do cenários. Para a realização do experimento, foi implementado em CUDA/C++ uma aplicação paralela do Método dos Elementos Discretos com os principais algoritmos apresentados na literatura, além desses o autor propõe e implementa a paralelização do algoritmo de detecção com ordenação e busca binária que ainda não havia sido paralelizado. Após os testes é constatado que o algoritmo com ordenação e busca é o mais eficiente para todos os cenários estudados, obtendo nos resultados um bom desempenho em tempo de execução e com uso de memória muito superior aos outros.

Palavras-chave: Detecção de Contatos. Busca por Vizinhos. Computação Paralela.

ABSTRACT

Contact detection algorithms are needed in different areas of science and technology. From digital games and computer graphics to high-performance simulations and robotics. These algorithms require great computational effort and are prone to become the bottlenecks of its applications, even more when this computation must be done in real-time or large-scale systems. With the popularization of GPU cards use for both science and business, it is only natural that parallel implementations for this problem arise in the scientific community. In this work the main contact detection algorithms are analyzed and a numerical experiment is performed, with the goal of finding out which algorithm has better computational performance and memory use, or if they efficiency depends on different scenario features. For performing the experiment, a parallel Discrete Element Method application was developed using CUDA/C++ with the main algorithms presented in literature, besides these, the author proposes and implements the Sorting Contact Detection algorithm parallelization, that hadn't been parallelized until now. The tests have found that the parallel Sorting Contact Detection algorithm is the most efficient in all studied scenarios, achieving a good performance and a superior memory usage than its peers.

Keywords: Discrete Elements Method. Contact Detection. Neighbor Search.

AGRADECIMENTOS

Agradeço ao professor Leonardo Viana, que se dedicou a me orientar sempre com muita paciência através dos inúmeros percalços que passamos.

À toda minha família com quem sempre posso contar, em especial à minha esposa, que conseguiu me aturar por esses anos de trabalho;

aos meus pais, que sem suas inspirações à busca do conhecimento eu nunca teria trilhado esse caminho;

aos meus sogros que se dedicaram a me ajudar de toda maneira possível, incluindo com a leitura do trabalho e críticas construtivas;

à minha tia, quase uma segunda mãe, minha irmã e irmão, que sempre me apoiaram e aturaram meu humor e minhas ausências.

Aos amigos, que são uma segunda família.

Aos professores que fizeram parte desse caminho, por promoverem a mediação de conhecimentos, que foram essenciais para meu desenvolvimento acadêmico.

Aos laboratórios LaCACAN e LCCV, por proporcionarem meios e a estrutura necessários para realização do meu trabalho.

À CAPES.

E por fim, à UFAL.

Bruno Normande

LISTA DE FIGURAS

LISTA DE TABELAS

Lista de Algoritmos

LISTA DE EQUAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Por mais de 50 anos a Lei de Moore (MacK, 2011) assegurou o crescimento do poder computacional, o que garantia uma certa segurança de que problemas difíceis computacionalmente eventualmente passariam a poder ser resolvidos pelo simples aumento de capacidade dos chips. Entretanto, está cada vez mais difícil assegurar esse crescimento e 2017 será o ano em que, pela primeira vez desde sua formulação em 1975, a **Intel** não conseguirá acompanhar a Lei de Moore (Mingis, 2016).

Mas não é recente que a indústria vem encontrando obstáculos para continuar nesse ritmo. Desde o início dos anos 2000 a dificuldade em aumentar o número de componentes dentro de um único núcleo de processamento fez com que outra abordagem surgisse: aumentar o número de núcleos de processamento dentro do mesmo chip, popularizando assim os processadores *multi-core* (Asanovic et al., 2006; Feinbube et al., 2011).

Estima-se que a CPU moderna terá até mil unidades de processamento (Asanovic et al., 2006). Enquanto isso, cresce no mercado o número de linguagens de programação com suporte a paralelismo, assim como a demanda por profissionais com conhecimento nesse tipo de problema (Feinbube et al., 2011). Os fabricantes de GPU, que anteriormente focavam seus produtos apenas na computação de elementos gráficos, passam a fornecer placas capazes de realizar processamento massivamente paralelo, além de fornecer as plataformas de desenvolvimento que as acompanham.

Segundo Nickolls & Dally (2010) é chegada a era da GPU, onde essas placas passaram a ser usadas, tanto pelo mercado quanto pela academia, em problemas que necessitam de grande poder computacional e otimização.

Algoritmos de detecção de contatos são essenciais para diferentes áreas da computação e engenharia (Munjiza, 2004; Ericson, 2005). Jogos digitais, computação gráfica, robótica, simulação, prototipagem e animações computadorizadas são apenas alguns exemplos que não seriam possíveis sem detecção de contato. Mas esses algoritmos são considerados verdadeiros gargalos de seus sistemas, em especial quando a etapa de busca por contato precisa ser realizada em tempo real, ou mesmo quando o número de corpos envolvidos é muito grande, chegando a tomar de 60% a 80% do tempo total de processamento (Munjiza, 2004).

Nesse trabalho o problema de detecção de contatos é abordado usando-se uma ferramenta para simulação para cenários com grandes quantidades de objetos, ou partículas.

O Método dos Elementos Discretos (DEM) (Cundall & Hart, 1992) é um método capaz de descrever o comportamento de um conjunto de corpos discretos ao longo do tempo. Muito utilizado na engenharia, sua real utilidade só é alcançada quando envolve centenas de milhares ou até milhões de corpos em uma mesma simulação (Munjiza, 2004). Justamente o caso em que a detecção de contatos passa a se tornar um problema.

A solução sequencial já foi bastante explorada, e atualmente encontram-se algoritmos com tempo de processamento $O(N)$ (Munjiza, 2004) e com uso de memória também $O(N)$, para corpos de diâmetro similares (Munjiza et al., 2004; Chen et al., 2014), ou para corpos de tamanhos distintos (Araújo et al., 2007; Perkins & Williams, 2001). Entretanto, para a realização de uma simulação, esses algoritmos são repetidos dezenas de milhares de vezes e isso, junto ao grande valor de N, torna a solução sequencial muitas vezes inviável.

Como algumas etapas desse método é inerentemente paralela, o cálculo das forças resultantes e a integração de movimento podem ser feitos para cada partícula de maneira independente, é fácil ver como esse tipo de problema pode se beneficiar de uma arquitetura massivamente paralela como da GPU. A única dificuldade resulta da etapa de detecção de contato, que, por não ser independente para cada partícula, não pode ser paralelizada de maneira direta.

Há na literatura diferentes algoritmos para essa etapa. Green (2013) implementa no *Particles* uma versão direta de checagem originalmente proposto por Kalojanov & Slusallek (2009). Zheng et al. (2012) propõe um versão diferente desse mesmo algoritmo modificando apenas a maneira que cada elemento é mapeado. Ambos usam um vetor ordenado para representar o *grid* de busca com intenção de diminuir o alto custo de memória exigido pelo algoritmo clássico de mapeamento direto (Munjiza, 2004).

Nesse trabalho é proposto a paralelização do algoritmo sequencial de detecção com ordenação e busca binária. O qual não havia ainda, até o momento da realização desse trabalho, sido implementado de maneira paralela, pois, é visto como um algoritmo de baixa eficiência computacional em arquitetura serial, devido às limitações dos algoritmos de ordenação e buscas necessários para seu funcionamento (Munjiza, 2004).

A pergunta que motiva essa pesquisa é a seguinte: há um único algoritmo paralelo de busca por contatos que seja identificado como o mais adaptado para as diferentes situações de busca?

Para encontrar a resposta foi feita a implementação paralela do Método dos Elementos Discretos (DEM), que, em seguida, foi usado em um experimento empírico para comparar os algoritmos paralelos de busca por contatos encontrados na literatura.

Foi realizada uma análise quantitativa de cada algoritmo, comparando-se a média do tempo de execução e a memória utilizada. Também foram analisados os resultados obtidos em trabalhos anteriores para as versões sequenciais dos algoritmos.

Os resultados anteriores mostraram que, para algoritmo sequenciais, quanto mais direta a representação do *grid*, melhor. Mesmo usando mais memória esse custo não era alto o suficiente para ser um impeditivo. Entretanto o algoritmo de ordenação e busca binária obteve resultados competitivos (Lins et al., 2011).

Em contrapartida, para arquiteturas GPU, memória não é um recurso abundante e a maneira como os dados são organizados possuem grande impacto no desempenho de algoritmos paralelos.

A hipótese desse trabalho é que o algoritmo proposto, versão paralela do algoritmo com ordenação e busca, se mostre adaptado para os cenários estudados devido ao seu baixo custo de memória e à eficiência que sua versão sequencial já mostrou anteriormente.

1.1 Objetivo

Essa dissertação tem dois objetivos gerais principais: propor a implementação paralela do algoritmo de detecção com Ordenação e Busca Binária (Munjiza, 2004) que não foi realizada até agora e realizar uma análise comparativa dos principais algoritmos paralelos, e sequenciais, de busca por contato em diferentes cenários de distribuição espacial de partículas. Buscando uma resposta sobre, como eles se comportam em diferentes cenários e se há um algoritmo que seja o melhor adaptado em todas situações. Os objetivos podem ser sintetizados como a seguir:

Identificar se o comportamento dos algoritmos paralelos são semelhantes aos seriais.

Visto que muito já se conhece sobre os algoritmos de busca sequenciais, se os paralelos se comportarem de maneira semelhante será possível saber previamente o que esperar desses.

Identificar se as características de empacotamento do cenário de busca influenciam no desempenho dos diferentes algoritmos.

Como os algoritmos possuem diferentes necessidades de comunicação e organização é possível que a mudança de características do cenário de busca influenciem negativamente ou positivamente de maneira individual para cada algoritmo.

Investigar a existência de um único algoritmo que seja o melhor adaptado em todos cenários, ou se cada cenário exige um diferente.

A importância da descoberta de um algoritmo melhor em todas ocasiões é clara, visto que com esse conhecimento fica sanada a dúvida de qual algoritmo implementar em uma devida aplicação. Por outro lado, se as características do cenário influenciarem os algoritmos a ponto de que para cada tipo de empacotamento existir um algoritmo diferente ideal, abre-se

a possibilidade futura de desenvolvimento de algoritmos paralelos adaptativos, capazes de escolher para cada subdomínio do sistema o algoritmo que melhor se adéqua.

Esse conhecimento é importante pois, apesar de muitos algoritmos dessa área possuírem a mesma complexidade assintótica, na prática, o custo médio de tempo de execução não é calculado trivialmente, necessitando de análise empírica para sua descoberta. Outro fator importante a ser considerado é o custo de memória de cada algoritmo, que pode ser fator chave para a escolha do algoritmo uma vez que esse recurso não é barato na arquitetura de placas GPU.

1.2 Metodologia

Esse trabalho realiza uma pesquisa experimental numérica, que realiza os seguintes passos:

- Implementação de um *software* DEM paralelo;
- Experimento quantitativo dos diferentes algoritmos estudados;
- Análise de desempenho face os cenários distintos.

1.3 Estrutura da Dissertação

Essa dissertação está organizada em 6 capítulos, são eles:

Capitulo 1: introdução ao tema, motivação, objetivo, questão da pesquisa e estrutura da dissertação.

Capitulo 2: fundamentação teórica do trabalho. Apresentação do problema de detecção de contatos e do Método dos Elementos Discretos.

Capitulo 3: apresentação e análise dos algoritmos de busca por vizinhos.

Capitulo 4: apresentação da metodologia seguida durante a realização do experimento, assim como das ferramentas utilizadas.

Capitulo 5: resultados obtidos após a execução dos experimentos.

Capitulo 6: análise dos resultados e objetivos alcançados do trabalho, assim como perspectivas futuras.

2 DETECÇÃO DE CONTATOS

Nesse capitulo serão introduzidos os conceitos essenciais a esse trabalho. Na seção 2.1 será apresentado o Método dos Elementos Discretos, que é o método da engenharia que é usado na implementação do *software* desenvolvido durante esse trabalho. Em seguida, na seção 2.2 é fundamentado o conceito de algoritmo de detecção de contatos e, mais detalhadamente, a subseção 2.2.1 trata da técnica de busca por vizinhos.

2.1 Método dos Elementos Discretos

O Método dos Elementos Discretos (*Discret Elements Method*, DEM), como definido por Cundall & Hart (1992), é um método numérico capaz de descrever o comportamento mecânico de um conjunto de corpos rígidos. Para tal ele precisa satisfazer duas condições:

- Permitir o cálculo de deslocamento e rotações finitas independentes para corpos discretos;
- Reconhecer novos contatos automaticamente à medida em que ocorre a análise.

O tempo total da simulação é subdividido em pequenos passos de tempo (Δt), em cada passo é feita a verificação de contato entre todos os corpos presentes na simulação, calculado a força dos contatos, feita a integração da equação de movimento e atualizada a posição dos corpos. Esse processo é repetido para cada passo no tempo até que a simulação esteja completa como mostra a Figura 2.1.

O cálculo da força pode ser abordado de diferentes maneiras. Em geral a intensidade da força de contato é uma função do quanto uma partícula interceptou a outra. Para obter a direção da força a escolha do método utilizado pode depender da geometria dos elementos do sistema assim como da complexidade de implementação desejada. Um dos métodos usados mais comuns na literatura é o método do plano comum (Cundall, 1988).

Uma vez que temos a força que deverá ser aplicada em cada partícula, a aceleração é obtida a partir da Segunda Lei de Newton (Equação 2.1).

$$m\vec{a} = \vec{f} \tag{2.1}$$

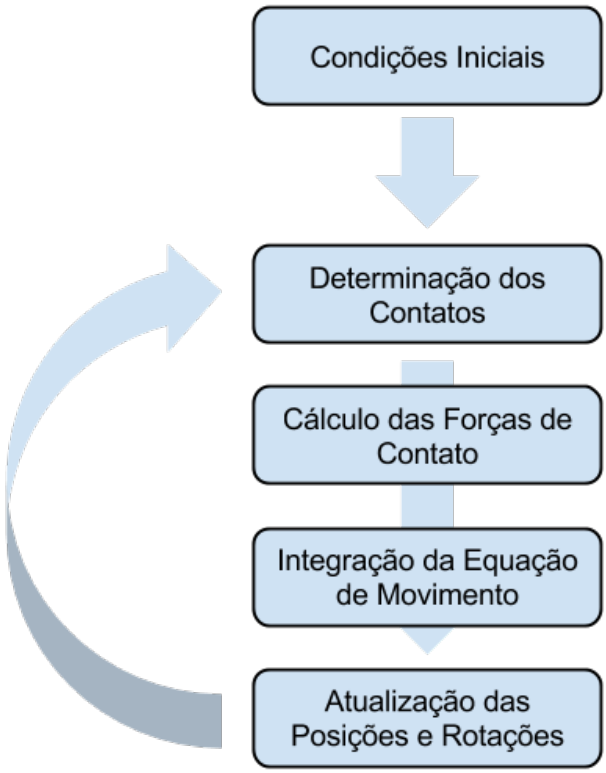


Figura 2.1 – Etapas realizadas em cada passo de tempo em uma simulação usando DEM.

A integração da equação de movimento pode ser realizada tanto analiticamente como numericamente. Para problemas DEM o método numérico é mais adequado por exigir menos hipóteses restritivas, serem mais versáteis e apresentarem respostas aproximadas satisfatórias.

Os métodos explícitos são normalmente bastante adequados ao DEM, pela característica de precisar apenas do estado em t para o cálculo do novo estado do sistema em $t + \Delta t$. Essa característica se adapta bem ao ciclo mostrado na Figura 2.1 onde cada iteração representa um incremento no tempo total de simulação.

Alguns autores escolhem utilizar métodos com amortecimento numérico devido à imprecisões geradas pelas condições de contorno e pela própria característica discreta da interação de contato, quando em um momento não há contato e abruptamente passa a existir no próximo Δt. Esses métodos usam amortecimento numérico para dissipar os erros gerados por essas imprecisões. Alguns exemplos desses tipos de métodos são os algoritmos *Meg-Alpha* (Hulbert & Chung, 1996) e Chung e Lee (Chung & Lee, 1994) ambos comumente usados em *softwares* DEM (da Silveira, 2001).

É muito comum também o uso de métodos explícitos diretos. O método de Euler simplificado (Green, 2013) pode ser usado quando se deseja uma maior simplicidade de implementação. O método das diferenças centrais também é bastante usado por apresentar baixa complexidade de processamento e resultados satisfatórios. Outras variações desses métodos

podem ser usadas ao se buscar melhoria na qualidade da análise (Munjiza, 2004).

A abordagem mais comum atualmente para a paralelização dos softwares DEM é o método da subdivisão do espaço de busca (Cintra, 2016; Bokhari, 1987; Karypis & Kumar, 1999).

Seja ele dinâmico ou estático, esse método consiste em separar grupos de partículas de acordo com sua localização, assim cada grupo é separado de acordo com o algoritmo escolhido, como por partições de grafos *k-way* (Markauskas & Kačeniauskas, 2015) ou células Voronoi (Fattebert et al., 2012) e assim cada CPU fica responsável por uma região de busca. Esse método tem a vantagem de diminuir a quantidade de comunicação entre as unidades de processamento, sendo ela necessária apenas quando uma partícula sai de uma região para a outra. A desvantagem é que em cada uma das CPUs executa na verdade o algoritmo sequencial.

Apesar de adequada para *clusters* de computadores, esse método de paralelismo não é bem aplicado para o paralelismo que se obtêm com o uso de uma GPU. Nesses casos, onde a comunicação não é tão cara, mas a memória dividida é limitada, Green (2013) aponta uma outra abordagem, a decomposição por partícula. Nesse tipo de decomposição os cálculos de cada partícula é feito independentemente em cada unidade de processamento.

Essa divisão é ideal para as etapas de integração da equação de movimento e de atualização das posições das partículas, visto que estas podem ser feitas independentemente para cada partícula sem necessidade de informação dos outros elementos do sistema.

Já para o cálculo das interações de contato é preciso verificar para cada partícula se ela está em contato com todas as outras. O custo computacional dessa checagem de contato ingênua possui alto custo computacional, complexidade $O(N^2)$ e torna seu uso inviável em uma implementação DEM que for ser usada com valor de N superior à algumas centenas de milhares de elementos, mesmo se todo esse processamento for divido entre os processadores de uma GPU (Green, 2013).

A solução usada é dividir a checagem de contato em dois passos. Para cada elemento N_i primeiro são separados os elementos próximos a ele, também chamados de vizinhos. Então a verificação de contato em sí só precisa ser realizada para os elementos da lista de vizinhos de cada partícula. A classe de algoritmos responsáveis por gerar essas listas são conhecidos como Algoritmos de Busca por Vizinhos, mas primeiro vamos falar dos conceitos gerais por trás da detecção de contato em sí (Munjiza, 2004).

2.2 Algoritmos de Detecção de Contatos

Em sistemas com grande número de elementos, a etapa de busca por contato pode chegar a ocupar de 60% a 80% do tempo total de processamento (Munjiza, 2004). Esse problema se torna ainda mais crítico se o sistema deve ser calculado em tempo real (Ericson, 2005).

É importante entender que a verificação de contato entre dois corpos é completamente

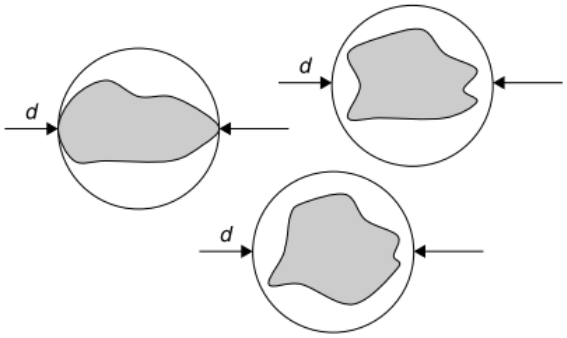


Figura 2.2 – Objeto delimitador circular (imagem retirada de Munjiza (2004)

dependente da geometria de ambos, podendo resulta em cálculos bastante complexos e custosos computacionalmente. Por esse motivo foi criado o conceito de objeto delimitador (*bounding object*), um objeto de geometria simples que é usado para conter os elementos que fazem parte da simulação e que primeiro se verifica o contato entre esse objetos para, apenas se houver esse contato, fazer o cálculo do contato em si.

Apesar de adicionar uma etapa a mais no algoritmo, essa técnica é importante pois trás dois benefícios: os algoritmos de detecção de contato criados usando esse conceito podem ser usados para objetos de qualquer geometria; e mais importante, esse técnica reduz bastante o tempo necessário para o processamento das simulações, podendo chegar a uma redução de até 50% do tempo de processamento nos casos de objetos com geometrias complexas, uma vez que o cálculo de contato entre essas geometrias podem consumir elevado tempo de processamento e o uso dos objetos delimitadores evita esse cálculo em casos onde o contato é impossível (Munjiza, 2004; Ericson, 2005).

Nesse trabalho são usados objetos delimitadores esféricos, na figura 2.2 podemos ver um exemplo desses objetos. O cálculo de contato é feito como mostrado na função (2.2): comparando a distância entre o centro das esferas com a soma de seus raios, se a inequação for verdadeira, há contato. Da equação (2.2) tiramos também que a complexidade computacional dessa operação de comparação isolada é constante, ou $O(1)$.

$$contato(A,B) = D_{AB} \leq R_A + R_B \tag{2.2}$$

A solução ingênua de se implementar um algoritmo de busca por contato é usando a função $contato()$ diretamente para cada dupla de elementos presentes no sistema. Esse algoritmo é chamado de Algoritmo de Checagem Direta, o algoritmo 2.1 é sua versão sequencial e o 2.2 é a versão paralela.

A análise assintótica de ambos algoritmos é simples, visto que ambos constituem apenas

Algoritmo 2.1 Algoritmo de checagem direta

```
1: for i de 0 a N do
2:     for j de i a N do
3:         contact(E_i, E_j)
4:     end for
5: end for
```

Algoritmo 2.2 Algoritmo de checagem direta paralelo

```
1: Inicia N processos
2: Para cada processo p_i
3:     for j de 0 a N do
4:         contact(E_i, E_j)
5:     end for
6: Finaliza execução paralela
```

de estruturas de repetição e a única função usada possui complexidade constante. A equação (2.3) é a complexidade do Mapeamento Direto sequencial, enquanto que a equação (2.4) é de sua versão paralela. É importante perceber dois fatores sobre o algoritmo paralelo: o algoritmo não se beneficia de um número de processadores maior que N e que isso raramente acontecerá em uma simulação real, onde milhões ou dezenas de milhões de objetos são usados, dessa maneira, quanto maior N mais próximo o algoritmo fica de complexidade quadrática.

$$MD = O(N^2) \tag{2.3}$$

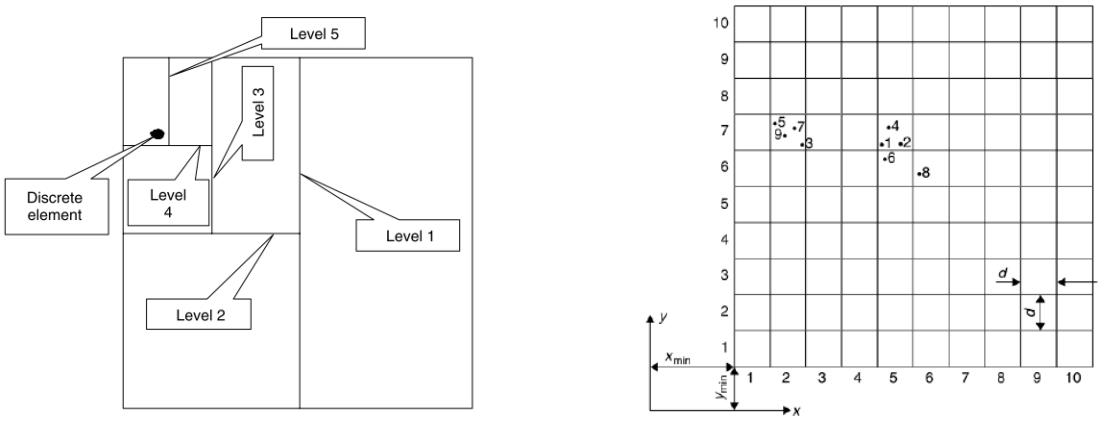
$$MDP = O\left(\left\lceil \frac{N}{P} \right\rceil \times N\right) \tag{2.4}$$

$$MDP = O(N^2) \qquad \text{se } P << N \tag{2.5}$$

Com essa complexidade, grande número de objetos e a quantidade de vezes que essa etapa precisa ser repetida durante uma simulação esse algoritmo se torna, na prática, inviável (Munjiza, 2004; Ericson, 2005).

Para combater esse gargalo computacional é preciso diminuir ao máximo o número de verificações de contato. Uma maneira de se fazer isso é deixando de verificar o contato entre partículas que estejam demasiadamente longe umas das outras a ponto que a verificação não seja necessária.

A classe de algoritmos que realiza essa operação é conhecida como Algoritmos de Busca por Vizinhos. Esse algoritmos consistem em realizar um pré-mapeamento da localização dos objetos para, após o mapeamento, testar o contato apenas entre os que estão próximos, daí o nome de Busca por Vizinhos.



(a) Divisão do espaço do tipo árvore (b) Divisão do espaço do tipo grid

Figura 2.3 – Diferença na divisão do espaço de árvore para *grid* (Figuras tiradas de Munjiza (2004))

2.2.1 Algoritmos de Busca por Vizinhos

Os algoritmos de busca por vizinhos podem ser divididos em duas categorias. Algoritmos de árvore, são aqueles que usam estruturas de árvores hierárquicas para representar o espaço de busca, como pode ser visto na Figura 2.3(a); e algoritmos de *grid* são os que separam o espaço de busca em células de igual tamanho. Cada célula possui o lado de tamanho d e pode ser endereçada por coordenadas x, y e z, assim como é visto na Figura 2.3(b).

Para simulações de DEM temos que métodos de busca em *grid* são mais eficientes que os algoritmos baseados em árvores, principalmente quando tamanhos dos elementos presentes na simulação é homogêneo (Ericson, 2005; Han et al., 2007). Além disso a estrutura de *grid* se assemelha à arquitetura paralela das GPUs, enquanto que a necessidade de reconstrução das árvores em cada passo faz com que esse tipo de algoritmo perca sua eficiência quando aplicado ao DEM, que exige milhares de reconstruções durante uma mesma simulação (Zheng et al., 2012). Dessa maneira esse trabalho foca seus experimentos na categoria de algoritmos de *grid*.

Esses algoritmos usam o modelo de *grid* para mapear a posição dos elementos no espaço de busca, mas se diferenciam entre si na maneira escolhida para representar esse mapeamento na memória. A diferença entre eles pode ser determinada pelas seguintes características:

- A estrutura de dados usada para representar o *grid*;
- O tamanho escolhido para o lado da célula, representado nesse trabalho por d;
- Se um mesmo elemento pode ser mapeado para mais de uma célula.

Existem duas maneiras principais de se fazer a correspondência entre as coordenadas de um elemento e as do *grid*. Pode-se atribuir cada elemento a apenas uma célula, usando como

base o ponto central do objeto delimitador do elemento. Ou, se cada elemento pode pertencer a mais de uma célula, usamos um paralelepípedo que contenha o objeto delimitador do elemento para determinar a faixa de células nas quais o devemos mapear.

Nos algoritmos do primeiro tipo, onde cada elemento só pode estar mapeado a uma célula cada, é importante que o tamanho d para a célula seja definido como o tamanho do maior objeto delimitador do sistema. Assim é garantido que ao atribuir cada elemento as suas células, nenhum poderá está em contato com um elemento a duas células de distância.

Nesses casos o cálculo para as posições X, YeZ do *grid* é como apresentado nas equações (2.6), onde (x_p, y_p, z_p) é a posição da partícula e (x_0, y_0, z_0) é a posição zero do espaço de busca.

$$\begin{aligned} X &= \left\lfloor \frac{x_p - x_0}{d} \right\rfloor \\ Y &= \left\lfloor \frac{y_p - y_0}{d} \right\rfloor \\ Z &= \left\lfloor \frac{z_p - z_0}{d} \right\rfloor \end{aligned} \tag{2.6}$$

Se o segundo método é usado, as células que irão conter o mapeamento do elemento podem ser obtidas percorrendo a partir da célula que contém o vértice inferior, esquerdo e atrás do paralelepípedo até a que contém o vértice superior, direito e frente do mesmo. Para calcular esses dois pontos basta usar a equação (2.7), onde r é o raio da esfera delimitadora.

$$\begin{aligned} X_0 &= \left\lfloor \frac{x_p - x_0}{d} - r \right\rfloor & X_1 &= \left\lceil \frac{x_p - x_0}{d} + r \right\rceil \\ Y_0 &= \left\lfloor \frac{y_p - y_0}{d} - r \right\rfloor & Y_1 &= \left\lceil \frac{y_p - y_0}{d} + r \right\rceil \\ Z_0 &= \left\lfloor \frac{z_p - z_0}{d} - r \right\rfloor & Z_1 &= \left\lceil \frac{z_p - z_0}{d} + r \right\rceil \end{aligned} \tag{2.7}$$

Ambas as equações, 2.6 e 2.7, possuem complexidade $O(1)$ e são usadas para o mapeamento dos elementos no *grid*. Como não há diferença entre elas em termo de complexidade, a função $mapear(E)$ é usada para representar ambas nas análises dos algoritmos.

A técnica básica de *grid* consiste em criar uma estrutura de dados onde cada célula do *grid* esteja representado diretamente por um *array*, ou uma lista encadeada, no qual possamos colocar os elementos mapeados para aquela célula. Chamaremos esse algoritmo de Mapeamento Direto (MD). Em sua versão paralela, para que não haja erro de concorrência, a ação de adição de elemento em um *array* tem que ser implementado como uma operação atômica. Amba as versões, sequencial e paralela, possuem um alto custo de memória, pois todo o *grid* precisa ser representado como um *array* multidimensional (Munjiza, 2004).

A implementação DEM da SDK CUDA, o *Particles* (Green, 2013), usa um *array* com os elementos ordenados de acordo com sua posição no *grid*. Para evitar a busca no vetor

ordenado, ele usa também mais dois *arrays* multidimensionais, com as mesmas dimensões do *grid*, onde são armazenados os índices referentes ao inicio e ao fim de cada célula no vetor ordenado (Green, 2013). Tal algoritmo foi inicialmente proposto por Kalojanov & Slusallek (2009) que o aplicou para realizar traçado de raios (*ray tracing*). Chamaremos esse algoritmo de Mapeamento com Ordenação (MO).

O algoritmo de Mapeamento por Célula (Araújo et al., 2007; Cundall, 1988) é semelhante ao Mapeamento Direto, mas, enquanto o MD necessita que o tamanho a célula d seja equivalente ao diâmetro do maior elemento do sistema, o Mapeamento por Célula (MC) permite escolher um tamanho arbitrário para d. Para que isso seja possível é preciso permitir que cada elemento passe a ser mapeado para mais de uma célula no sistema.

Zheng et al. (2012) propõe a versão paralela do MC. Apesar de melhorar o desempenho, em casos onde os elementos que compõe a simulação possuem tamanhos muito variados, o custo de memória, assim como o próprio desempenho, podem sofrer mudanças drásticas a depender do tamanho d escolhido para célula.

A representação de todo o *grid* de busca como um *array* pode ter um custo muito caro de memória. Uma alternativa é o algoritmo de Ordenação e Busca (OB), que utiliza-se de vetores de tamanho N para armazenar a posição dos elementos de maneira ordenada e, na etapa de busca por contato, usa o algoritmo de Busca Binária para encontrar os elementos das células vizinhas. Essa etapa de busca faz com que a busca por contato com os vizinhos tenha complexidade $O(N\log(N))$ e por esse motivo esse algoritmo não é muito popular na academia (Munjiza, 2004).

Como não foi possível achar na literatura uma versão paralela desse algoritmo, nesse trabalho é proposta a sua implementação paralela para comparação com os outros algoritmos paralelos, já que sua versão sequencial já se mostrou eficiente em trabalhos passados (Lins et al., 2011).

O *No Binary Search* (NBS) é um algoritmo que usa uma estrutura de lista encadeadas para representar o *grid* (Munjiza & Andrews, 1998), mas sem precisar da etapa de busca binária (daí o nome). Para evitar a busca binária ele tem como requisito que a busca por contato seja feita de maneira sequencial, em uma iteração sobre os elementos da simulação, e por isso não é possível ser paralelizado.

Apesar dos algoritmos apresentados não serem os únicos existentes na literatura, esses são os mais gerais. Outros algoritmos, como por exemplo o apresentado por Harada (2007), que subdivide o espaço de busca em faixas , geralmente tem o uso muito restrito a configuração dos cenários de busca (Goswami & Schlegel, 2010) ou possuem um custo de memória tão alto como o MD, mas sem ganho de desempenho (Zheng et al., 2012).

Em síntese, os principais algoritmos que podem ser aplicados de maneira geral a diferentes cenários de busca são estes apresentados na tabela 2.1. No próximo capitulo cada um desses algoritmos é explicado em detalhes e analisado.

Algoritmo	Serial	Paralelo
Mapeamento Direto	X	X
Mapeamento com Ordenação		X
Mapeamento por Célula	X	X
Ordenação e Busca	X	X
No Binary Search	X	

Tabela 2.1 – Pricipais algoritmos de busca por vizinhos

3 ALGORITMOS DE BUSCA DE VIZINHOS

Nesse capitulo é feita a análise dos algoritmos usados na experimentação desse trabalho. Para que essa análise seja realizada é preciso primeiro definir que o escopo desse trabalho é o comportamento dos algoritmos em cenários com partículas de tamanho homogêneas, dessa maneira será considerado que os objetos delimitadores presentes em uma mesma simulação possuem todos o mesmo tamanho.

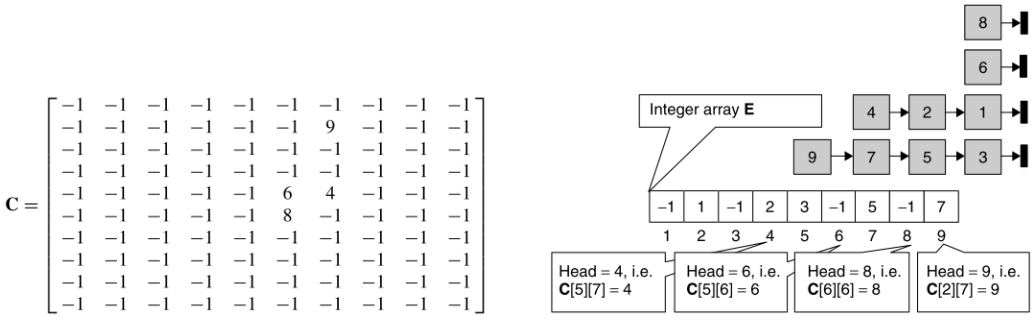
3.1 Algoritmos Seriais

3.1.1 Mapeamento Direto

No algoritmo de Mapeamento Direto é feito uma relação direta entre o *grid* de busca e um *array* multidimensional de listas encadeadas. As listas são usadas para armazenar os elementos mapeados para cada célula do *grid*. O valor para lado da célula d é escolhido como o diâmetro do maior objeto.

A Figura 3.1 mostra um exemplo das estruturas de dados usadas para representar o *grid*, e como elas estariam preenchidas a partir do cenário demostrado na figura 2.3(b).

As estruturas de dados necessárias para esse algoritmos são a matriz C com as mesmas dimensões D_x, D_y e D_z do *grid*, além do vetor E de tamanho N. Sendo assim sua complexidade de memória é como descrita na equação (3.1).



(a) O *Array* C representa as células do *grid*.

(b) Lista encadeada com os elementos.

Figura 3.1 – Estruturas de dados usadas pelo Mapeamento Direto para representar o *grid* de busca. Imagens retiradas de Munjiza (2004)

Para representar a lista encadeada temos que -1 representa o fim da lista. A matriz C possuí o índice do primeiro elemento de cada lista (*head*); o vetor E armazena os índices dos próximo elementos da lista. Usando a Figura como exemplo, a célula $x = 2, y = 7$ teriam os elementos indicados na lista $C[2][7]$, sendo o primeiro elemento $id = 9$, o próximo será o $E[9]$, que é 7 então o próximo seria o $E[7]$... e assim segue até o último elemento que é o 3 e $E[3] = -1$, o que indica o fim da lista.

$$MEM_{MD} = D_x \times D_y \times D_z + N \tag{3.1}$$

Algoritmo 3.1 Mapeamento Direto

Require: C, E, N
1: **for** Para cada objeto o_i de 0 a N **do**
2: $x_o, y_o, z_o \leftarrow mapeamento(o_i)$
3: adiciona o_i na lista C_{x_o,y_o,z_o}
4: **end for**
5: **for** Para cada objeto o_i de 0 a N **do**
6: $x_o, y_o, z_o \leftarrow mapeamento(o_i)$
7: **for** Para x de $x_o - 1$ a $x_o + 1$ **do**
8: **for** Para y de $y_o - 1$ a $y_o + 1$ **do**
9: **for** Para z de $z_o - 1$ a $z_o 1$ **do**
10: **for** Para cada e na lista $C_{x,y,z}$ **do**
11: $contato(o_i, e)$
12: **end for**
13: **end for**
14: **end for**
15: **end for**
16: **end for**

O algoritmo completo de Mapeamento Direto (Munjiza, 2004) é como visto em 3.1. O primeiro *for*, linhas 1-4, contém duas operações, *mapeamento* e adição na lista encadeada, ambas possuem tempo constante e por isso a repetição tem complexidade $O(N)$.

O segundo *for* contém uma sequência de repetições que aparece em vários dos algoritmos: a busca por contato entre as células vizinhas. Ao todo são 27 células, cada uma pode conter no máximo 8 elementos, se eles estiverem exatamente nos vértices daquela célula, mas então a célula vizinha poderá conter apenas 4 elementos e uma terceira que seja vizinha de ambas poderá ter apenas duas. Dessa maneira o total de operações feitas por esses *loops* será 64 operações, que reflete o número máximo de elementos mapeados para as células vizinhas. Assim a complexidade do *for* de busca nos vizinhos é de $O(64N) = O(N)$.

Então a complexidade assintótica do algoritmo Mapeamento Direto é:

$$COMP_{MD} = O(N) \tag{3.2}$$

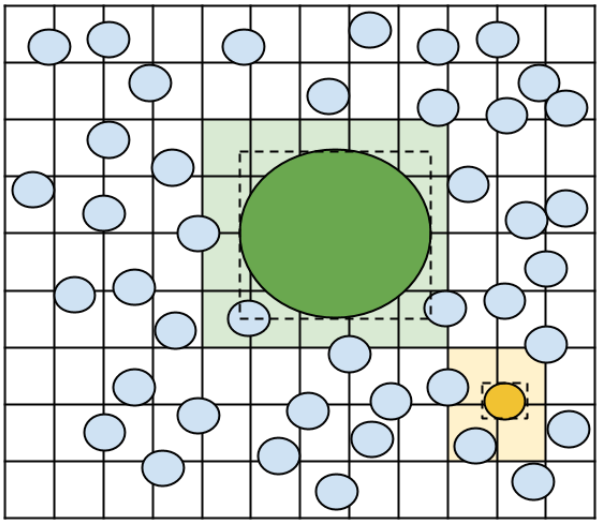


Figura 3.2 – Exemplo do mapeamento no algoritmo de Mapeamento por Células.

3.1.2 Mapeamento por Células

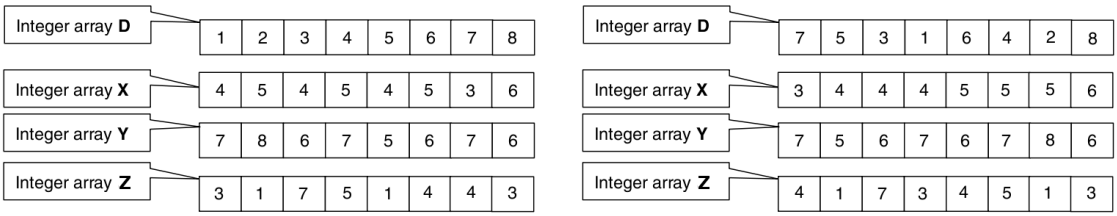
O algoritmo de Mapeamento por Células (MC) (Araújo et al., 2007; Cundall, 1988) é muito semelhante ao MD, diferenciando-se em dois detalhes: o tamanho escolhido para d pode ser arbitrário e cada elemento pode ser mapeado para mais de uma célula. À parte disso, as estruturas de dados usadas e o algoritmo são os mesmos.

A única mudança é que, como cada elemento pode ser mapeado para mais de uma célula, quando o tamanho dos objetos é homogêneo e o valor de d escolhido como o diâmetro desses objetos, um elemento poderá ser mapeado para no máximo 8 células, no caso do centro do mesmo coincidir com o vértice da célula. Dessa maneira tanto o custo de memória referente às listas encadeadas é multiplicado por 8, equação (3.3), como a quantidade de *repetições* na busca por contato, o que não altera a complexidade assintótica do algoritmo, equação (3.4).

A Figura 3.2 é um exemplo de como é feito o mapeamento: o elemento verde que será mapeado para as células em verde enquanto o elemento amarelo é mapeado para as células em amarelo.

$$MEM_{MC} = D_x \times D_y \times D_z + 8 \times N \tag{3.3}$$

$$COMP_{MC} = O(N) \tag{3.4}$$



(a) Mapeamento de cada elemento. (b) Vetores de mapeamento após ordenação.

Figura 3.3 – Exemplos de mapeamento e ordenação no algoritmo de Ordenação e Busca, o vetor D armazena o id de cada elemento, enquanto X, Y, eZ armazenam as posições do mesmo e são usados como base da ordenação (Figuras tiradas de (Munjiza, 2004)).

3.1.3 Ordenação e Busca

O custo de memória dos algoritmos MD e MC podem se tornar um empecilho quando a simulação tiver um cenário de busca muito vasto, já que seus custos de memória dependem diretamente das dimensões do *grid* de busca. O algoritmo de Ordenação e Busca (OB) (Munjiza, 2004) foi criado justamente para cenários onde a memória é um recurso escasso.

Usando apenas 4 vetores de tamanho N como estrutura de dados, esse algoritmo consiste em armazenar as coordenadas de cada elemento nesse conjunto de vetores para em seguida ordena-lo de acordo com os valores de X, Y e Z, a Figura 3.3 representa esse processo. A dimensão para d escolhida é o diâmetro do maior objeto e cada elemento pode ser mapeado apenas para uma célula. Na etapa de busca é usado o algoritmo de busca binária, sobre os vetores X, Y e Z, para se localizar os elementos de cada célula. O algoritmo 3.2 apresenta-o passo a passo, no algoritmo é usada a função $achar_primeiro(x,y,z)$, que é implementada como uma busca binária para encontrar a primeira aparição de um elemento mapeado para (x,y,z).

A complexidade assintótica desse algoritmo é limitado pelos algoritmos de ordenação e busca presentes no mesmo. A equação (3.5) apresenta sua complexidade. O custo de memória desse algoritmo é como apresentado em (3.6).

$$COMP_{OB} = O(N\log(N)) \tag{3.5}$$

$$MEM_{OB} = 4 \times N \tag{3.6}$$

Algoritmo 3.2 Ordenação e Busca

Require: D, X, Y, Z
1: **for** Para cada objeto o_i de 0 a N **do**
2: $x_o, y_o, z_o \leftarrow mapeamento(o_i)$
3: $D_i \leftarrow i$
4: $X_i \leftarrow x_o$
5: $Y_i \leftarrow y_o$
6: $Z_i \leftarrow z_o$
7: **end for**
8: $ordenar(D, X, Y, Z)$
9: **for** Para cada objeto o_i de 0 a N **do**
10: $x_o, y_o, z_o \leftarrow mapeamento(o_i)$
11: **for** Para x de $x_o - 1$ a $x_o + 1$ **do**
12: **for** Para y de $y_o - 1$ a $y_o + 1$ **do**
13: **for** Para z de $z_o - 1$ a $z_o + 1$ **do**
14: $e \leftarrow achar\_primeiro(x, y, z)$
15: **while** e em (x, y, z) **do**
16: $contato(o_i, e)$
17: $e++$
18: **end while**
19: **end for**
20: **end for**
21: **end for**
22: **end for**

3.1.4 No Binary Search

O NBS foi criado por Munjiza & Andrews (1998) com intenção de usar uma estrutura de vetores, como o OB, mas evitando os algoritmos de ordenação e busca binária. Para tal ele utiliza três listas encadeadas, uma para cada dimensão do *grid*, para representar o espaço de busca.

Esse algoritmo é tão complexo que alguns autores simplesmente não o implementam em seus estudos (Han et al., 2007). Em seu livro Munjiza & Andrews (1998) explica detalhadamente seu funcionamento, assim como a prova de que sua complexidade assintótica é proporcional a N. A equação (3.7) apresenta a complexidade desse algoritmo e a (3.8) seu custo de memória em relação às dimensões do espaço de busca.

$$COMP_{NBS} = O(N) \tag{3.7}$$

$$MEM_{NBS} = D_x + D_y + D_z + 5 \times N \tag{3.8}$$

3.2 Algoritmos Paralelos

3.2.1 Mapeamento Direto Paralelo

Para implementar a versão paralela do Mapeamento Direto é preciso modificar as estruturas de dados usadas por sua versão sequencial. Devido ao alto custo, ou até impossibilidade, de alocação dinâmica de memória na GPU, e também para evitar problemas de concorrência, o uso de listas encadeadas para armazenar os elementos mapeados para cada célula tem que ser descartado.

Em seu lugar são usados dois *arrays* de inteiros: C será o vetor que servirá como contador de quantos objetos estão atribuídos para cada célula, enquanto que o *array* G armazenará os *ids* desses elementos. Assim, para adicionar um novo elemento a uma célula, primeiro é feito uma soma atômica[1] no índice da célula no vetor C e depois é adicionado o *id* do elemento em G.

Para garantir que todos os elementos possam ser corretamente mapeados, é preciso que G tenha como tamanho o total de células do *grid* multiplicado pelo número máximo possível de objetos em cada célula. A Figura 3.4 ilustra como as estruturas de dados apresentadas representam o *grid* de busca.

[1] Operações atômicas são aquelas que garantem que não haverá erro de concorrência durante sua execução

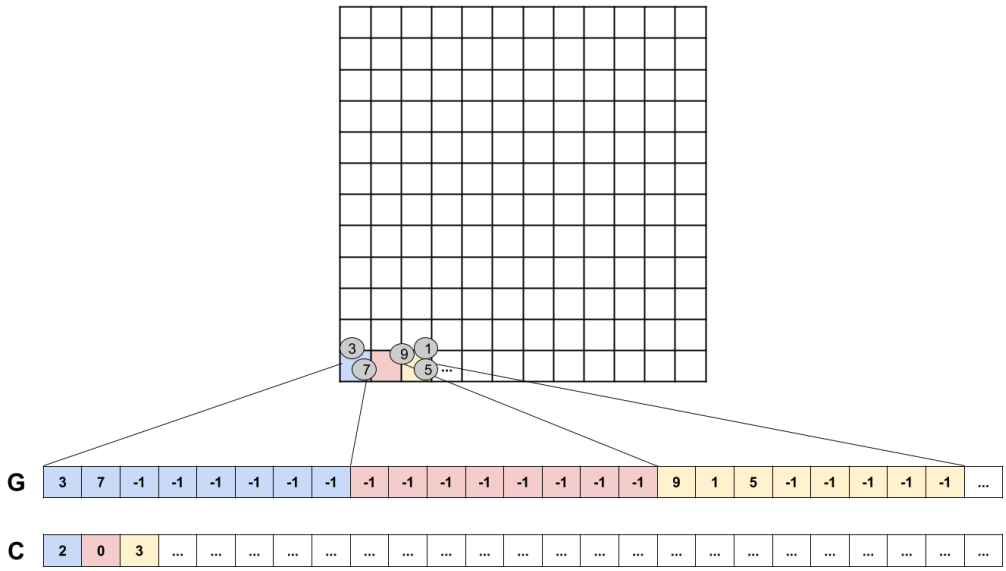


Figura 3.4 – Representação do *grid* de busca no Mapeamento Direto Paralelo

Da Figura 3.4 podemos ver uma característica importante desse algoritmo: como todo espaço de busca tem que ser representado na estrutura, se a simulação tiver uma baixa densidade de elementos, a maior parte da matriz G será usada para representar espaço vazio.

O custo de memória do MDP é proporcional ao tamanho do espaço de busca. Como nesse algoritmo o tamanho da célula ,d, é definido como o diâmetro do maior objeto delimitador, então cada célula pode conter no máximo 8 elementos, cada um centralizado exatamente em cada um dos vértices da célula. Assim o custo de memória das estruturas será como definido na equação 3.9, considerando os *arrays* C (contador de quantos elementos tem em cada célula) e G (para cada célula, um vetor de 8 espaços).

$$MEM_{MDP} = D_x \times D_y \times D_z \times 9 \tag{3.9}$$

Tanto a etapa de mapeamento quando a de busca são realizadas paralelamente para cada partícula. O algoritmo 3.3 demostra como acontecem ambas as etapas. A primeira não possui nenhuma estrutura de repetição, sendo composta apenas da função de mapeamento, uma soma atômica e algumas atribuições, portanto essa parte do código possui complexidade $O\left(\frac{N}{p}\right)$. Os *loops* internos da segunda etapa são exatamente como o de sua versão sequencial que, como foi visto, possui complexidade constante e, como resultado, a complexidade dessa etapa também é $O\left(\frac{N}{p}\right)$. Como ambas acontecem de maneira sequencial então a complexidade assintótica delas é mantida (equação (3.10)).

$$COMP_{MDP} = O\left(\frac{N}{P}\right) \tag{3.10}$$

Algoritmo 3.3 Mapeamento Direto Paralelo

Require: C, G
1: **Inicia** N processos
2: Para cada processo p_i ▷ Mapeamento
3: $x_o, y_o, z_o \leftarrow mapeamento(o_i)$
4: $n \leftarrow atomicAdd(C[x_o, y_o, z_o], 1)$
5: $G[x_o, y_o, z_o][n] \leftarrow i$
6: **Finaliza** execução paralela
7: **Inicia** N processos
8: Para cada processo p_i ▷ Busca por contato
9: $x_o, y_o, z_o \leftarrow Mapeamento(o_i)$
10: **for** x de $x_o - 1$ a $x_o + 1$ **do**
11: **for** y de $y_o - 1$ a $y_o + 1$ **do**
12: **for** z de $z_o - 1$ a $z_o + 1$ **do**
13: $total \leftarrow C[x_o, y_o, z_o]$
14: **for** j de 0 a $total$ **do**
15: $other \leftarrow G[x, y, z][j]$
16: $contato(o_i, other)$
17: **end for**
18: **end for**
19: **end for**
20: **end for**
21: **Finaliza** execução paralela

3.2.2 Mapeamento com Ordenação Paralelo

O algoritmo de Mapeamento com Ordenação Paralelo (Kalojanov & Slusallek, 2009) foi criado com o objetivo de evitar o uso de operações atômicas do MDP, assim como para evitar seu alto custo de memória. Para que isso seja alcançado ele usa vetores ordenados como estrutura de mapeamento dos elementos. Ou seja, no lugar de um *array* representando todo o espaço de busca, são usados quatro vetores de tamanho N que armazenam o id e as coordenadas (x, y, z) de cada elemento, de maneira similar ao algoritmo Ordenação e Busca.

Para que não seja necessário o uso de um algoritmo de busca ao se procurar os elementos de cada célula no *array* ordenado, Kalojanov & Slusallek (2009) propõe o uso de dois vetores para armazenar o índice dos elementos que marcam o início e fim de cada célula. Assim as estruturas usadas para representar o *grid* de busca serão os quatro *arrays* tamanho N e dois de dimensão $D_x \times D_y \times D_z$.

A etapa de mapeamento é dividida em três partes. Primeiro armazena-se as coordenadas de cada partícula, e seu *id*, nos *arrays* D, X, Y e Z. Em seguida o *array* é ordenado de acordo com suas coordenadas no *grid*. Então os índice de início e fim de cada célula são armazenados nos vetores I e F, respectivamente.

Após a etapa de ordenação, fazemos a busca pelo início e fim de cada célula no vetor ordenado. Essa etapa é bastante simples, basta, para cada elemento no *array* ordenado, verificar se o elemento anterior a ele está na mesma célula. Se estiver, nada é feito. Mas, se o elemento anterior estiver mapeado para uma célula diferente, então isso quer dizer que ele é o último elemento de sua célula, e o atual é o primeiro da nova célula. Enfim se faz a busca nas células vizinhas por contato. O algoritmo 3.4 mostra todas etapas do MOP.

Algoritmo 3.4 Algoritmo Mapeamento com Ordenação Paralelo

Require: D, X, Y, Z, I, F
Inicia N processos
Para cada processo p_i ▷ Mapeamento
    $x_o, y_o, z_o \leftarrow Mapeamento(o_i)$
    $D_i \leftarrow i$
    $X_i \leftarrow x_o$
    $Y_i \leftarrow y_o$
    $Z_i \leftarrow z_o$
Finaliza execução paralela
$sort(D, X, Y, Z)$ ▷ Ordenação Paralela
Inicia N processos
Para cada processo p_i ▷ Busca pelo início e fim das células
    if $X_i \neq X_{i-1}$ || $Y_i \neq Y_{i-1}$ || $Z_i \neq Z_{i-1}$ **then**
        $I[X_i, Y_i, Z_i] \leftarrow i$
        $F[X_{i-1}, Y_{i-1}, Z_{i-1}] \leftarrow i - 1$
    end if
Finaliza execução paralela
Inicia N processos
Para cada processo p_i ▷ Busca por contato
    $x_o, y_o, z_o \leftarrow Mapeamento(o_i)$
    for x de $x_o - 1$ a $x_o + 1$ **do**
        for y de $y_o - 1$ a $y_o + 1$ **do**
            for z de $z_o - 1$ a $z_o + 1$ **do**
                $ini \leftarrow I[x,y,z]$
                $fim \leftarrow F[x,y,z]$
                for j de ini a fim **do**
                    $contato(o_i, D[j])$
                end for
            end for
        end for
    end for
Finaliza execução paralela

As etapas de mapeamento, busca pelo início e fim das células e busca por contato apresentam complexidade $O\left(\frac{N}{P}\right)$, visto que só são compostas de operações de tempo constante. Já o algoritmo de ordenação usado é o *Merge Sort*, implementado pela biblioteca *Thrust* (Bell & Hoberock, 2012), que possui complexidade $O\left(\frac{N\log(N)}{P}\right)$ (Cole, 1986; Satish et al., 2009). A

complexidade do algoritmo por completo é apresentado da equação (3.11).

$$COMP_{MOP} = O\left(\frac{N\log(N)}{P}\right) \tag{3.11}$$

O custo de memória do MOP é depende to tamanho do espaço de busca assim como do número de partículas no sistema. Assim, o custo de memória das estruturas será como definido na equação 3.12, considerando os *arrays* I, F, D, X, Y e Z.

$$MEM_{MOP} = D_x \times D_y \times D_z \times 2 + 4 \times N \tag{3.12}$$

3.2.3 Mapeamento por Células Paralelo

Semelhante ao Mapeamento Direto Paralelo, a principal diferença desse algoritmo para sua versão sequencial está na estrutura usada para representar o *grid*, que também passa a ser um vetor multidimensional no lugar de listas encadeadas. Zheng et al. (2012), ao propor esse algoritmo, aponta como sua principal vantagem a possibilidade de escolha do tamanho de d, que pode ser definido de acordo com as necessidades do sistema.

As estruturas de dados usadas para representar o *grid* são as mesmas que no MDP, com a diferença que cada elemento poderá ser mapeado para mais de uma célula. O algoritmo 3.5 mostra o processo de mapeamento para um elemento, que é adicionado nas células que fazem intersecção com o cubo que sobre-escreve o elemento.

Como esse algoritmo mapeia os elementos para todas as células que contêm o objeto delimitador, então para checar o contato basta percorrer essas células realizando o teste de contato com os outros elementos mapeados na mesma, como demostra o algoritmo 3.5. Sua complexidade é apresentada na equação (3.13).

$$COMP_{MCP} = O\left(\frac{N}{P}\right) \tag{3.13}$$

Considerando o tamanho máximo de partículas por célula como 8, no caso de elementos de tamanhos homogêneos, o custo total de memória para esse algoritmo é mostrado na equação 3.14.

$$MEM_{MCP} = D_x \times D_y \times D_z \times 9 \tag{3.14}$$

Algoritmo 3.5 Algoritmo de Mapeamento por Celulas Paralelo

Require: C, G, raio
1: **Inicia** N processos
2: Para cada processo p_i ▷ Mapeamento
3: $x_o, y_o, z_o \leftarrow Mapeamento(o_i)$
4: $URB \leftarrow (x_o + raio, y_o + raio, z_o + raio)$
5: $DLF \leftarrow (x_o - raio, y_o - raio, z_o - raio)$
6: **for** (x, y, z) de DLF a URB **do**
7: $n \leftarrow atomicAdd(C[x,y,z], 1)$
8: $G[z,y,x][n] \leftarrow i$
9: **end for**
10: **Finaliza** execução paralela
11: **Inicia** N processos
12: Para cada processo p_i ▷ Busca por contato
13: $x_o, y_o, z_o \leftarrow Mapeamento(o_i)$
14: $URB \leftarrow (x_o + raio, y_o + raio, z_o + raio)$
15: $DLF \leftarrow (x_o - raio, y_o - raio, z_o - raio)$
16: **for** (x, y, z) de DLF a URB **do**
17: **for** j de 0 a $C[x,y,z]$ **do**
18: $contato(E_i, G[z,y,x][j])$
19: **end for**
20: **end for**
21: **Finaliza** execução paralela

3.3 Proposta do Algoritmo de Ordenação e Busca Paralela

Nenhum dos algoritmos estudados realizam etapa de busca no vetor ordenado, usando no lugar estruturas auxiliares para identificar o inicio e fim de cada célula. Essas estruturas auxiliares terminam por resultar no aumento do custo de memória dos mesmos. Recurso esse que possui alto custo, tanto de armazenamento quanto de acesso, em arquiteturas GPU.

Nesse trabalho é proposto a versão paralela do algoritmo de Ordenação e Busca sequencial. Para realizar a paralelização é usando o algoritmo *merge sort* paralelo para a ordenação dos elementos a partir de suas posições e, na etapa de busca é usada busca binária de maneira simultânea nos núcleos paralelos.

O Ordenação e Busca Paralelo requer as mesmas estruturas de dados de sua versão sequencial, e seu custo de memória é como descrito na equação (3.15). Por ser proporcional apenas ao tamanho de N, o custo de memória desse algoritmo não vai depender do espaço de busca, o que poderá se mostrar vantajoso, principalmente em cenários de busca com elementos esparsos.

$$MEM_{OBP} = 4 \times N \tag{3.15}$$

O algoritmo completo é como descrito no Algoritmo 3.6. Para a etapa de ordenação é usado a função *sort* da biblioteca *Thrust* (Bell & Hoberock, 2012), já a busca é realizada paralelamente para cada objeto usando-se o algoritmo de busca binária implementada pelo autor. A complexidade algorítmica desse algoritmo (equação (3.16)) é limitado pelas dos algoritmos de ordenação e busca binária, ambos $O\left(\frac{N\log N}{p}\right)$, uma vez que a busca binária ($O(\log N)$) deverá ser realizada paralelamente para todas N partículas.

Apesar de ter uma complexidade, no pior caso, restritiva. Esse algoritmo pode obter vantagem em seu caso médio por dois motivos: uma vez ordenado, espera-se que o vetor se mantenha bem comportado, com poucas modificações, já que de um passo ao outro as partículas não realizam movimentos drásticos e, assim, ele não precise ser reordenado todo passo; O outro fator que pode se mostrar positivo é que como esse algoritmo usa uma única estrutura de memória, o vetor ordenado, e que os elementos que estiverem próximos na simulação o estarão também no vetor, isso permitirá que a GPU otimize seu uso de cache, permitindo menos acessos à memória principal da placa e tornando o algoritmo mais eficiente.

Algoritmo 3.6 Algoritmo de Ordenação e Busca Paralela

Require: D, X, Y, Z
Inicia N processos
Para cada processo p_i ▷ Mapeamento
    $x_o, y_o, z_o \leftarrow Mapeamento(o_i)$
    $D_i \leftarrow i$
    $X_i \leftarrow x_o$
    $Y_i \leftarrow y_o$
    $Z_i \leftarrow z_o$
Finaliza execução paralela
$sort(D,X,Y,Z)$ ▷ Ordenação Paralela
Inicia N processos
Para cada processo p_i ▷ Busca por contato
    $x_o, y_o, z_o \leftarrow mapeamento(o_i)$
    for x de $x_o - 1$ a $x_o + 1$ **do**
        for y de $y_o - 1$ a $y_o + 1$ **do**
            for z de $z_o - 1$ a $z_o + 1$ **do**
                $e \leftarrow achar\_primeiro(x, y, z)$
                while e em (x, y, z) **do**
                    $contato(o_i, e)$
                    $e++$
                end while
            end for
        end for
    end for
Finaliza execução paralela

$$COMP_{OBP} = O\left(\frac{N\log N}{p}\right) \tag{3.16}$$

3.4 Resumo dos Algoritmos Estudados

As tabelas 3.1 e 3.2 contém um resumo dos custos, de memória e computacional, de cada algoritmo estudado nesse trabalho.

Algoritmo	**Memória**	**Desempenho**
Mapeamento Direto	$D_x \times D_y \times D_z + N$	$O(N)$
Mapeamento com Ordenação	-	-
Mapeamento por Célula	$D_x \times D_y \times D_z + 8 \times N$	$O(N)$
Ordenação e Busca	$4 \times N$	$O(N\log(N))$
No Binary Search	$D_x + D_y + D_z + 5 \times N$	$O(N)$

Tabela 3.1 – Resumo das complexidades dos algoritmos sequenciais estudados, não há versão sequencial do algoritmo de Mapeamento com Ordenação.

Algoritmo	**Memória**	**Desempenho**
Mapeamento Direto	$D_x \times D_y \times D_z \times 9$	$O\left(\frac{N}{p}\right)$
Mapeamento com Ordenação	$D_x \times D_y \times D_z \times 2 + 4 \times N$	$O\left(\frac{N\log(N)}{p}\right)$
Mapeamento por Célula	$D_x \times D_y \times D_z \times 9$	$O\left(\frac{N}{p}\right)$
Ordenação e Busca	$4 \times N$	$O\left(\frac{N\log N}{p}\right)$
No Binary Search	-	-

Tabela 3.2 – Resumo das complexidades dos algoritmos paralelos estudados, não há versão paralela do algoritmo NBS.

4 EXPERIMENTO NUMÉRICO

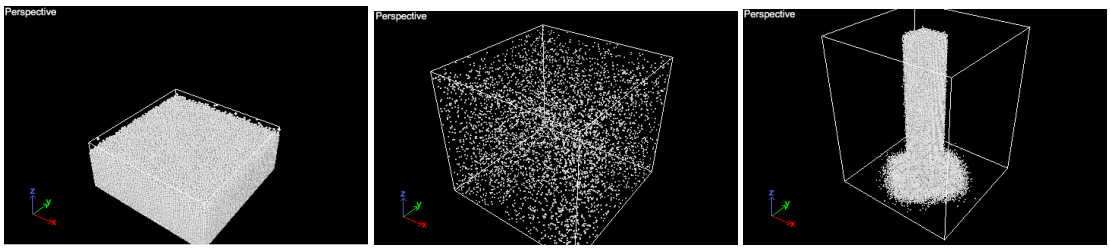
Para responder as questões levantadas nesse trabalho, foi criado um experimento numérico capaz de analisar cada um dos algoritmos nos diferentes cenários de distribuição espacial. A classificação desses cenários é feita de acordo com o agrupamento, ou empacotamento, dos elementos presentes na simulação. Segue uma descrição dos diferentes tipos de empacotamento:

Empacotamento Denso: durante todo tempo de simulação as partículas continuam sempre próximas umas das outras, sua maioria em contato com ao menos uma outra partícula durante toda a simulação, normalmente associado ao estado sólido da matéria, um exemplo pode ser visto na Figura 4.1(a);

Empacotamento Esparso: as partículas entram raramente em contato nesse tipo de cenário normalmente elas "vagam" pelo espaço de busca durante boa parte da simulação sem encontrar outro elemento, normalmente associado ao estado gasoso da matéria, um exemplo pode ser visto na Figura 4.1(b);

Empacotamento Misto: nesse tipo é possível encontrar, em diferentes grupos de partículas, as características tanto do empacotamento denso quanto do esparso, esse empacotamento é normalmente associado ao estado líquido da matéria. A figura 4.1(c) exemplifica esse tipo.

Os testes dos algoritmos sequenciais foram apresentados em trabalho anterior pelo autor (Lins et al., 2011) no qual foram comparados os resultados obtidos para cada algoritmo. Nesse



(a) Empacotamento Denso (b) Empacotamento Esparso (c) Empacotamento Misto

Figura 4.1 – Exemplos dos três tipos de empacotamento (Figuras geradas pelo autor).

trabalho o autor implementou os algoritmos de Ordenação e Busca e o NBS no software DEMOOP (LCCV, n.d.).

4.1 Implementação do Software Particles-Extended

Para a realização dos experimentos em GPU, o autor desenvolveu um *software* DEM paralelo, nomeado *Particles-Extended* (ou PEX). O PEX foi baseado no *software Particles* (Green, 2013), mas com ênfase no desempenho e reuso do código. Todos os algoritmos de detecção de contatos descritos nesse trabalho foram implementados nesse *software*.

A implementação foi feita usando-se C++ e a plataforma CUDA (Nickolls et al., 2008), para os algoritmos de busca foi usada a biblioteca *Thrust* (Bell & Hoberock, 2012), ambos disponíveis para download. Todo o código fonte e casos de teste usados nesse trabalho podem ser encontrados no repositório do autor no *GitHub*[1].

Para simplificação do código e foco na detecção de contato o PEX possui apenas a opção de geometria esférica para os elementos presentes na simulação e apenas o método de Euler foi adicionado, por sua simplicidade e baixo custo computacional. Por esse motivo, também não é feito o cálculo do deslocamento rotacional das partículas. Para o cálculo das forças de contatos é utilizado uma simplificação do método do plano comum (Cundall, 1988) onde a normal do plano é obtida a partir do vetor que liga os pontos centrais das partículas que estão interagindo.

4.2 Execução do Experimento

Todos os testes foram executados em um *cluster* computacional, cada computador com processador *Intel Xeon E5-2670* e uma placa NVIDIA *Tesla M2075* com 6GB de memória e 448 núcleos de processamento.

Nas simulações foram usadas partículas de tamanho homogêneo. Todos algoritmos foram testados usando-se as mesmas condições iniciais e os mesmos intervalos de N para cada empacotamento, denso, esparso e misto. A variação do número de elementos para cada cenário pode ser vista na Tabela 4.1.

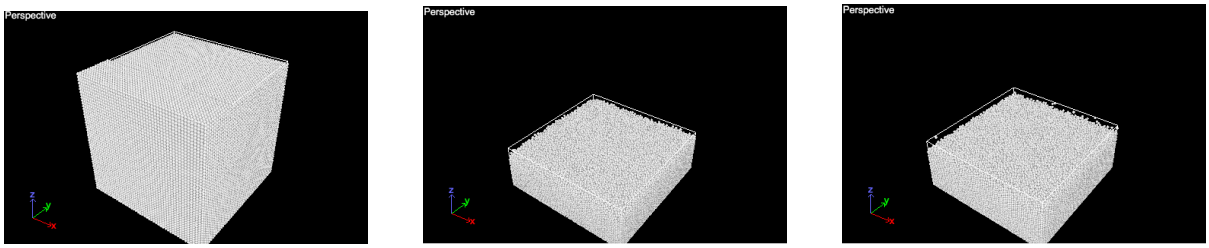
Empacotamento	Número de Partículas
Denso	1×10^5 a 5×10^6
Esparso	1×10^5 a 10^6
Misto	1×10^5 a 10^6

Tabela 4.1 – Quantidade de partículas por tipo de empacotamento

[1] https://github.com/jollyrog3r/pex

Cada simulação foi executada 14 vezes e a média do tempo total de execução foi calculada para cada valor de N. Todas foram programadas para realizar 1500 passos no tempo, realizando dessa maneira, na prática, 21 mil repetições de cada algoritmo, para cada N e em cada cenário diferente, de maneira a obter um baixo desvio padrão nos resultados.

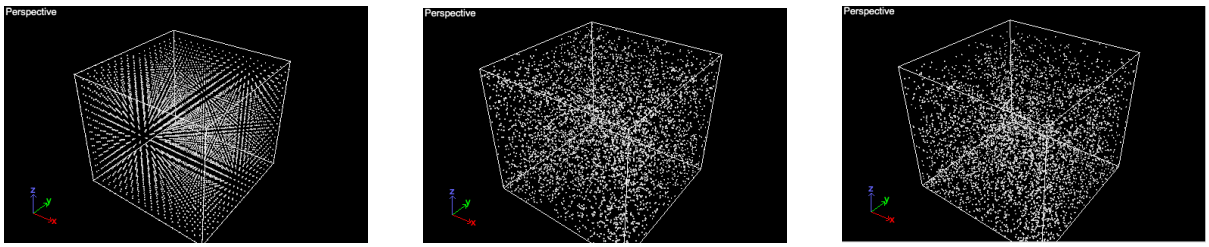
Para as simulações de empacotamento denso foi criado um cenário onde todas as partículas se encontram dentro de um cubo fechado, em estado inicial de repouso. Ao iniciar a simulação a força gravitacional tira as partículas do estado de repouso e elas caem até se estabilizarem. A Figura 4.2 demostra três momentos diferentes de uma simulação, usando o PEX, com empacotamento denso e 10×10^5 partículas.



(a) Estado inicial (b) Durante a simulação (c) Estado final

Figura 4.2 – Estados da simulação do empacotamento denso

As simulações com empacotamento esparso foram feitas em um cenário similar, com três mudanças: a distância entre as partículas passa a ser de dez vezes o diâmetro das mesmas, com uma pequena variação aleatória, a gravidade é diminuída para $\frac{1}{30}$ do valor original e, ao invés de iniciarem em repouso, elas possuem velocidade inicial com direção e intensidades também aleatórias. Dessa maneira, as partículas são lançadas em movimento para todas as direções desde o inicio e continuam assim durante todo tempo de simulação. A figura 4.3 apresenta três momentos diferentes de uma simulação com cinco mil partículas em empacotamento esparso.

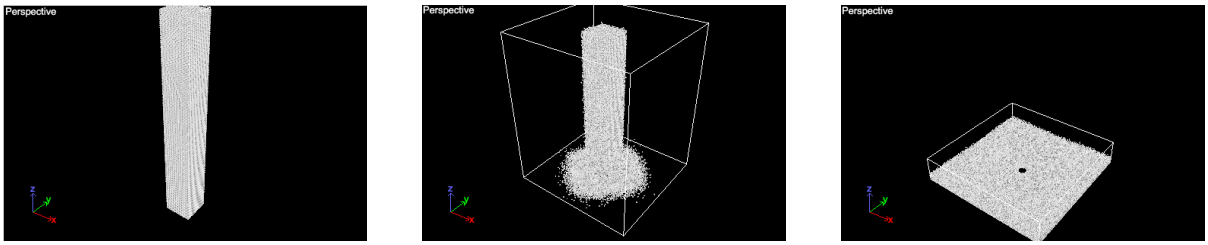


(a) Estado inicial (b) Durante a simulação (c) Estado final

Figura 4.3 – Estados da simulação do empacotamento esparso

O cenário mais diferente é o misto. Nesse caso os elementos partem de uma posição com densidade alta, mas o espaço de busca configurado tem dimensões equivalentes ao

esparso. Ao partir do repouso as partículas caem, puxadas pela gravidade, de encontro a um objeto esférico posicionado abaixo das mesmas. Essa colisão espalha as partículas para todos os lados enquanto elas continuam a cair, obtendo assim diferentes configurações de agrupamentos na mesma simulação. A figura 4.4 mostra três momentos diferentes de uma simulação com cinquenta mil elementos em empacotamento esparso.



(a) Estado inicial (b) Durante a simulação (c) Estado final

Figura 4.4 – Estados da simulação do empacotamento misto

5 RESULTADOS

Nesse capitulo é realizada a análise e comparação dos dados obtidos com as simulações. Na seção 5.1 é discutido os resultados para empacotamento denso, o mesmo é feito em seguida nas seções 5.2 e 5.3 onde são expostos os resultado para empacotamentos esparsos e mistos.

5.1 Empacotamento Denso

Os resultados obtidos nos testes com empacotamento denso podem ser vistos na Figura 5.1, onde a média de execução de todos os algoritmos para cada N é comparada. O desvio padrão para os algoritmos Mapeamento Direto Paralelo (MDP), Mapeamento com Ordenação Paralelo (MOP) e Ordenação e Busca Paralelo (OBP) ficaram abaixo de 2 segundos para todos valores de N, apenas o algoritmo Mapeamento por Célula Paralelo (MCP) se mostrou mais instável a medida que o número de elementos crescia, chegando a apresentar um desvio padrão de 290 segundos (4,8 minutos) para 5 milhões de partículas, mas, ainda assim, é considerado um desvio aceitável já que a sua média de tempo de execução nesse estágio foi de 12644 segundos (3,5 horas).

Analisando a Figura 5.1 é visto que o Mapeamento Direto possui um desempenho superior ao dos outros algoritmos, sendo igualado apenas pelo Ordenação e Busca, em contradição ao que seria esperado levando em conta suas complexidades assintóticas. Ambos obtêm médias muito próximas em toda simulação, para $N = 5 \times 10^6$ por exemplo, o MDP tem uma média de 943 segundos enquanto o OBP fica ligeiramente melhor com 919 segundos.

É possível perceber também que todos algoritmos possuem comportamento próximo ao linear, mas que há uma grande perda de desempenho para os algoritmos MOP e MCP em relação aos MDP e OBP. Para $N = 5 \times 10^6$ a média do MOP foi 6 vezes pior que o do OPB enquanto o MCP, que obteve o pior desempenho entre eles, chegou a ser 13 vezes pior que o OBP. Essa é a diferença entre a execução do código demorar 15 minutos (919s), uma hora e meia (5.725s) ou 3 horas e 50 minutos (12.644s).

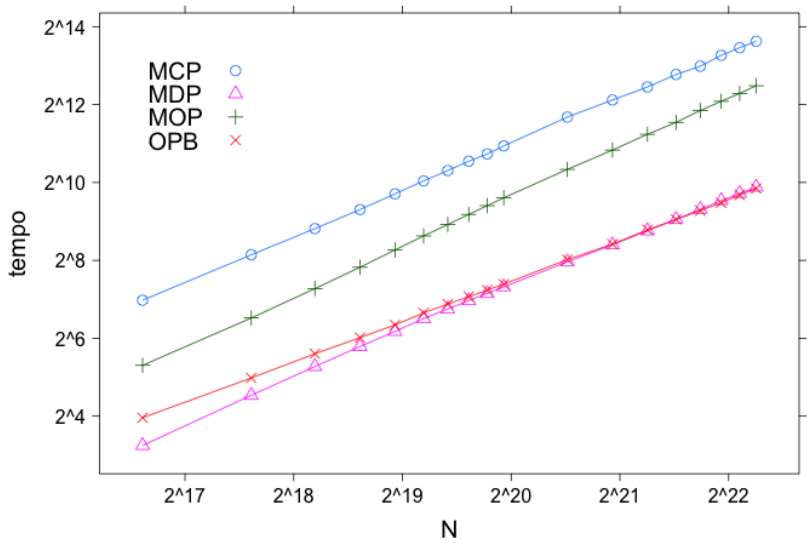


Figura 5.1 – Resultados obtidos para empacotamento denso. Os algoritmos de Mapeamento Direto e Ordenação e Busca apresentam desempenhos muito próximos, enquanto o Mapeamento com Ordenação toma 6 vezes mais tempo que eles e o Mapeamento por Células chega a ser 13 vezes mais lento.

Na Figura 5.2 é apresentada uma comparação entre os resultados dos algoritmos MDP e OBP. Ambos obtiveram desempenho semelhantes, mas é importante notar que o OBP passa a ter desempenho superior ao MDP a partir de $N = 3 \times 10^6$, mesmo o primeiro tendo complexidade algorítmica inferior.

Isso confirma a hipótese levantada na Sessão 3.3, que o algoritmo de Ordenação e Busca Binária iria se beneficiar tanto do algoritmo de ordenação trabalhar com um vetor semi-ordenado na maioria dos casos quanto do uso eficiente da memória. Acentuando o quão crítico é o uso da memória em arquiteturas GPU.

O custo de memória de cada algoritmo é visto na Figura 5.3, o MDP e MCP tiveram o mesmo custo de memória, que chega a ser mais que três vezes o custo do OBP. A tabela 5.1 sumariza os resultados obtidos no maior valor de N para cada algoritmo. Dessa forma fica evidenciado que as dimensões do *grid* possuem um impacto maior no custo de memória do que o tamanho de N, para os algoritmos em que o custo dependem dos dois fatores.

Algoritmo	Número Máximo de Partículas	Desempenho (Segundos)	Memória (MB)
Ordenação e Busca P.	5×10^6	919 s	518
Mapeamento Direto P.	5×10^6	943 s	1.699
Mapeamento com Ordenação P.	5×10^6	5.725 s	793
Mapeamento por Células P.	5×10^6	12.644 s	1.699

Tabela 5.1 – Número máximo de partículas, tempo de execução e custo de memória para cada algoritmo paralelo no empacotamento denso, ordenação crescente a partir do desempenho.

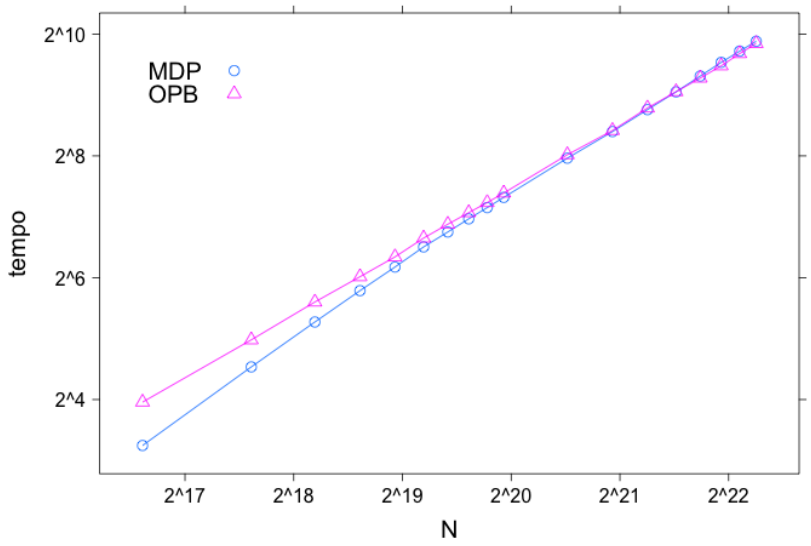


Figura 5.2 – Detalhe nos resultados do Mapemanto Direto e Ordenação e Busca, os dois algoritmos obtiveram os melhores desempenhos. A partir de $N = 10^6$ o OBP passa a mostrar comportamento superior ao MDP.

5.2 Empacotamento Esparso

Nos experimentos com empacotamento esparso foi possível constatar o real problema do alto custo de memória de alguns dos algoritmos estudados. O Mapeamento Direto, Mapeamento com Ordenação e Mapeamento por Células não chegaram a executar todos os testes, pois estouraram o limite da memória da GPU. As Figuras 5.4 e 5.5 expõem os resultados do tempo de execução e do custo de memória de cada algoritmo.

Para $N = 10^6$ o OBP obteve uma média de 136 segundos, menor que seu resultado com empacotamento denso, essa diferença era esperada já que quando as partículas estão distribuídas de maneira esparsa poucos contatos precisam ser calculados em cada passo no tempo. Esse algoritmo foi o único capaz de executar todos os testes nesse empacotamento, como seu custo de memória é proporcional a N e completamente independente do espaço de busca, ele é o único realmente adequado para esse tipo de cenário. Novamente o que apresenta o pior resultado foi o MCP.

5.3 Empacotamento Misto

Como o cenário misto também possui uma grande área de busca, ele recai no mesmo problema do esparso. Os algoritmos que têm o maior custo de memória, Mapeamento Direto e Mapeamento por Células, não puderam ser executados para $N > 6 \times 10^5$. A Figura

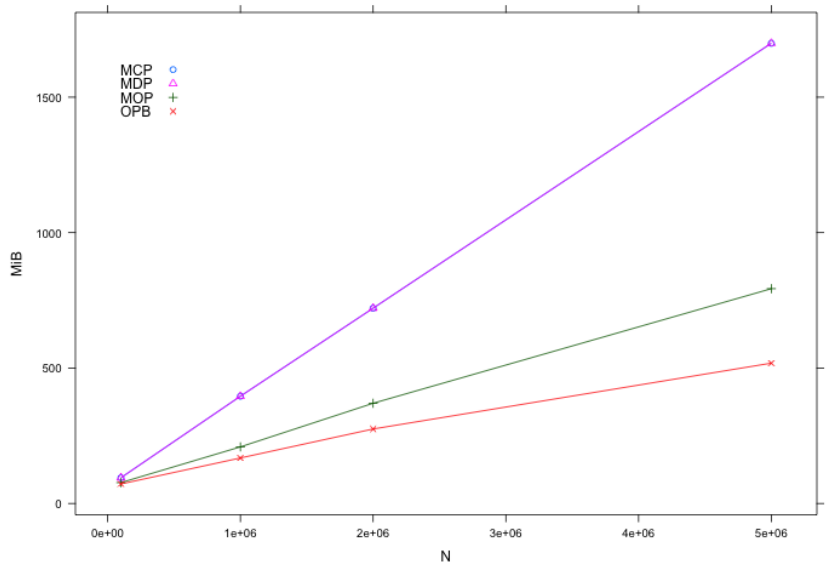


Figura 5.3 – Custo de memória dos diferentes algoritmos em empacotamento denso, apesar dos algoritmos Mapeamento Direto e Mapeamento por Células possuírem custo de memória mais alto que os outros, a diferença não é tão grande devido ao limitado espaço de busca nesse tipo de empacotamento.

Algoritmo	Número Máximo de Partículas	Desempenho (Segundos)	Memória (MB)
Ordenação P. e Busca	10^6	136 s	168
Mapeamento Direto P.	10^5	11 s	3.806
Mapeamento com Ordenação P.	6×10^5	120 s	4.929
Mapeamento por Células P.	10^5	62 s	3.806

Tabela 5.2 – Número máximo de partículas, tempo de execução e memória gasta para cada algoritmo paralelo no empacotamento esparso

5.6 exibe o desempenho dos algoritmos para esse cenário e a 5.7 seus custos de memória. Novamente o Mapeamento por Células obtêm desempenho consideravelmente inferior aos outros, apresentando comportamento próximo ao quadrático.

Também nesse cenário o OPB se mostra tão eficiente quanto nos outros (154 segundos para $N = 10^6$). Como o Mapeamento com Ordenação também possui um menor requisito de memória ele foi capaz de executar a busca para 1 milhão de elementos, mas fica claro que seu desempenho é inferior ao OPB, tendo levado em média o dobro do tempo de processamento deste, 306 segundos, para o maior valor de N experimentado.

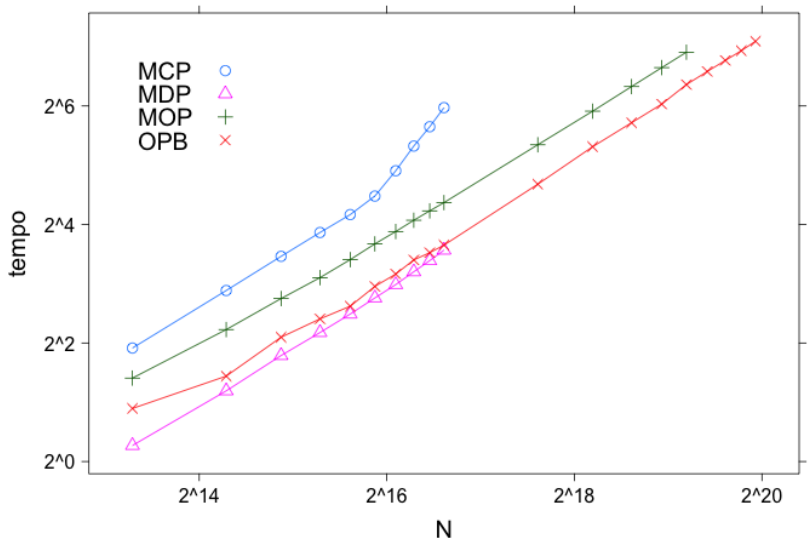


Figura 5.4 – Nos resultados obtidos para empacotamento esparso apenas o algoritmo Ordenação e Busca executou todos os testes, enquanto os outros atingiram o limite de memória da GPU.

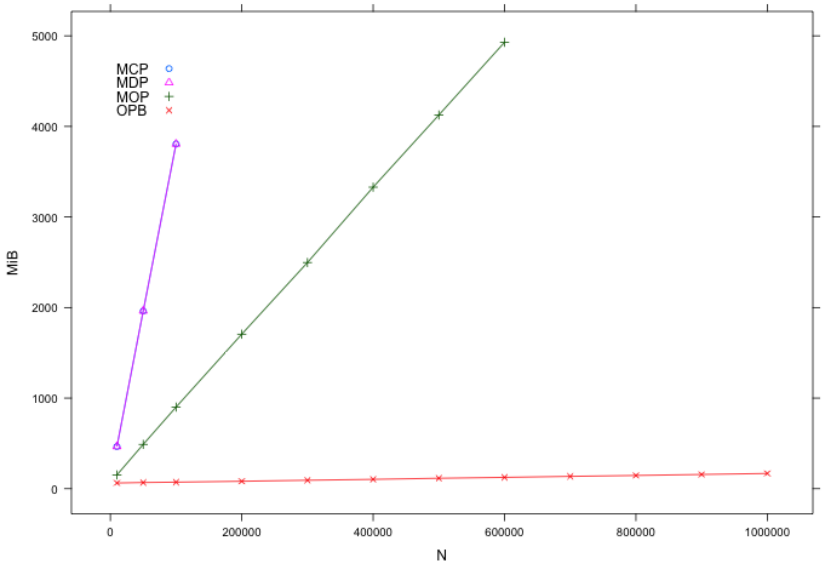


Figura 5.5 – Custo de memória empacotamento esparso, em contraste com empacotamento denso, a diferença dos custos de memória entre os algoritmos é clara.

Algoritmo	Número Máximo de Partículas	Desempenho (Segundos)	Memória (MB)
Ordenação P. e Busca	10^6	154 s	169
Mapeamento Direto P.	6×10^5	66 s	4.752
Mapeamento com Ordenação P.	10^6	306 s	1.881
Mapeamento por Células P.	6×10^5	498 s	4.752

Tabela 5.3 – Número máximo de partículas, tempo de execução e memória gasta para cada algoritmo paralelo no empacotamento misto

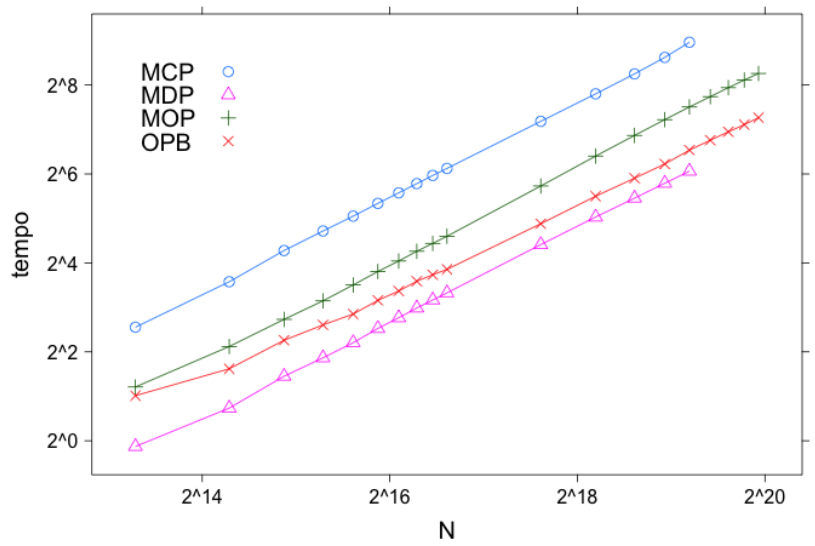


Figura 5.6 – Resultados obtidos para empacotamento misto, o Mapeamento por Células apresenta comportamento próximo ao quadrático. Mapeamento com Ordenação obtêm desempenho pior que Odenação e Busca, chegando a tomar o dobro do tempo de execução para o valor máximo de N.

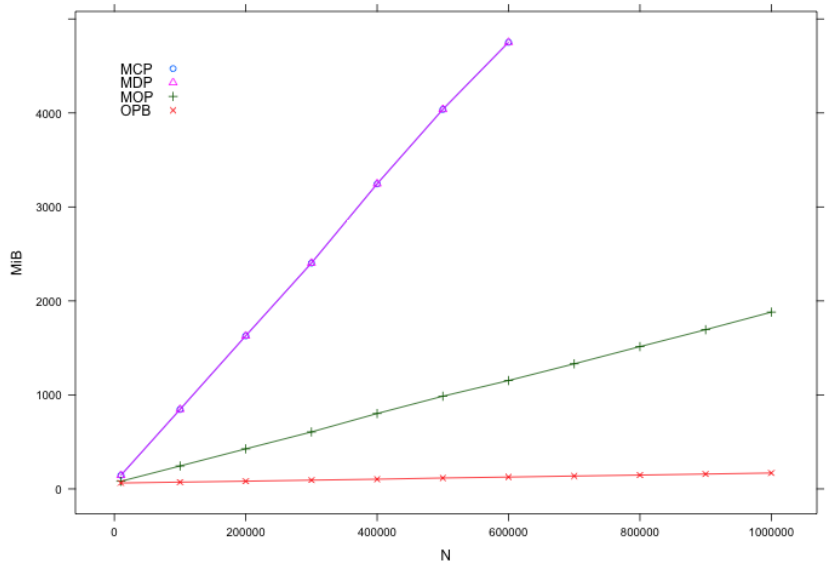


Figura 5.7 – Custo de memória empacotamento misto. Devido ao tamanho do espaço de busca, os algoritmos Mapeamento Direto e Mapeamento por Células ultrapassaram o limite de memória da placa para $N > 6 \times 10^5$.

6 CONSIDERAÇÕES FINAIS

O crescimento de poder computacional nos últimos anos está diretamente ligado ao aumento do número de núcleos nas unidades de processamento, ainda assim, a própria Intel declarou que em 2017 não conseguirá mais acompanhar a Lei de Moore. Enquanto isso cresce a popularidade do uso de placas gráficas para além da renderização de imagens, capazes de realizar computação massivamente paralela, cada dia são mais usadas em áreas que necessitam de grande poder computacional, como jogos digitais, robótica, engenharia, computação científica e simulação.

Algoritmos de detecção de contato estão presentes em todas essas áreas e são responsáveis por identificar colisões entre objetos presentes em seus sistemas. No Método dos Elementos Discretos esses algoritmos chegam a ser responsáveis por até 80% do tempo total de computação em uma simulação.

Nesse trabalho, foi feita a proposta da implementação paralela de um algoritmo sequencial de detecção de contato tradicional. Ademais é realizada também uma revisão bibliográfica dos principais algoritmos na literatura. Foi realizada um análise algorítmica de cada um e, em seguida, uma experimentação numérica para efetuar estudo comparativo de seus desempenhos em diferentes cenários de busca.

A pergunta que esse trabalho busca responder é: há um algoritmo que possa ser identificado como o mais adaptado nos diferentes cenários de busca?

Os resultados mostraram que o algoritmo proposto responde satisfatoriamente a essa pergunta. Com desempenho competitivo em relação ao mais rápido dos algoritmos, obtendo por vezes velocidades até melhores que o outro, e com baixo custo de memória, sendo essa proporcional apenas ao número de elementos, o algoritmo de Ordenação e Busca Paralelo se mostrou como o mais adaptado às diferentes situações dentre seus pares.

Resultado similar foi encontrado em estudo anterior para os algoritmos sequenciais, onde o Ordenação e Busca se mostrou mais eficiente que o esperado mas, devido à maior capacidade de memória de *desktops* e ao menor custo do Mapeamento Direto sequencial, o OB encontra sua utilidade restrita a casos de baixíssima densidade de elementos e em sistemas sensíveis ao custo de memória.

Os algoritmos de Ordenação e Busca, tanto o paralelo quanto o sequencial, se beneficiam de três fatores que explicam o bom desempenho apesar da complexidade algorítmica. O

primeiro é o fato de utilizarem os algoritmos de ordenação das bibliotecas padrões, que, via de regra, são extremamente otimizados; o segundo é que o vetor ordenado é bastante comportado ao longo da simulação, normalmente mantendo a ordem da maior parte dos elementos; por último, mas talvez mais importante, vem o fato da única estrutura de dados usada no algoritmo ser o vetor ordenado, e os elementos a serem consultados estão sempre próximos uns aos outros na memória, beneficiando o uso da *cache* na maioria das arquiteturas.

Por outro lado o Mapeamento por Células mostrou um péssimo desempenho assim como alto custo de memória. Criado para cenários compostos de elementos com grande diferença de tamanhos entre sí, esse algoritmo deve ser completamente evitado em ambientes com partículas homogêneas.

Futura pesquisa pode ser realizada acerca do uso do *cache* das placas GPU e como a arquitetura de memória pode ser melhor explorada para otimizar ainda mais esses algoritmos. É preciso também investigar algoritmos feitos para elementos de tamanhos distintos. Além disso, as ultimas gerações de placas GPU estão vindo com possibilidade de alocação dinâmica de memória e lançamento de *kernel* dentro de *kernel*, o que pode gerar o surgimento de novos algoritmos de detecção de contatos, ou até a modificação de alguns dos algoritmos estudados nesse trabalho.

REFERÊNCIAS

Araújo, J. P. N., Carvalho Júnior, H., Lira, W. W. M. & Ramos Junior, A. S. (2007), Estudo e implementação de um algoritmo de busca de contatos para problemas com elementos discretos planos de formas e dimensões arbitrárias, *in* '28st Iberian Latin American Congress on Computational Methods in Engineering', Porto, Portugal, pp. 1–14.

Asanovic, K., Catanzaro, B. C., Patterson, D. a. & Yelick, K. a. (2006), 'The Landscape of Parallel Computing Research : A View from Berkeley', *EECS Department University of California Berkeley Tech Rep UCBEECS2006183* **18**, 19.

Bell, N. & Hoberock, J. (2012), 'Thrust: A Productivity-Oriented Library for CUDA', *GPU Computing Gems Jade Edition* pp. 359–371.

Bokhari, S. H. (1987), 'A Partitioning Strategy for Nonuniform Problems on Multiprocessors', *IEEE Transactions on Computers* **C-36**(5), 570–580.

Chen, H., Lei, Z. & Zang, M. (2014), 'LC-Grid: a linear global contact search algorithm for finite element analysis', *Computational Mechanics* pp. 1285–1301.

Chung, J. & Lee, J. M. (1994), 'A new family of explicit time integration methods for linear and non-linear structural dynamics', *International Journal for Numerical Methods in Engineering* **37**(23), 3961–3976.

Cintra, D. T. (2016), Metodologia de paralelização híbrida do DEM com controle de balanço de carga baseado em curva de Hilbert, PhD thesis, Universidade Federal de Pernambuco.

Cole, R. (1986), 'Parallel Merge Sort.', *Annual Symposium on Foundations of Computer Science (Proceedings)* pp. 511–516.

Cundall, P. (1988), 'Formulation of a three-dimensional distinct element model—Part I. A scheme to detect and represent contacts in a system composed of many polyhedral blocks', *International Journal of Rock Mechanics and Mining ...* **25**(3), 107–116.

Cundall, P. A. & Hart, R. D. (1992), 'Numerical modelling of discontinua', *Engineering Computations* **9**, 101–103.

da Silveira, E. S. S. (2001), Análise Dinâmica de Linhas de Ancoragem com Adaptação no Tempo e Subciclagem, PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro.

Ericson, C. (2005), *Real-time collision detection*, elsevier ed., Morgan Kaufmann Publishers.

Fattebert, J.-L., Richards, D. & Glosli, J. (2012), 'Dynamic load balancing algorithm for molecular dynamics based on Voronoi cells domain decompositions', *Computer Physics Communications* **183**(12), 2608–2615.

Feinbube, F., Tröger, P. & Polze, A. (2011), 'Joint forces: From multithreaded programming to GPU computing', *IEEE Software* **28**(1), 51–57.

Goswami, P. & Schlegel, P. (2010), Interactive SPH simulation and rendering on the GPU, *in* 'Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation', pp. 55–64.

Green, S. (2013), Particle Simulation using CUDA, Technical Report September, NVIDIA.

Han, K., Feng, Y. T. & Owen, D. R. J. (2007), 'Performance comparisons of tree-based and cell-based contact detection algorithms', *Engineering Computations* **24**(1-2), 165–181.

Harada, T. (2007), Sliced Data Structure for Particle-Based Simulations on GPUs, *in* 'Graphite', number 5, pp. 55–62.

Hulbert, G. & Chung, J. (1996), 'Explicit time integration algorithms for structural dynamics with optimal numerical dissipation', *Computer Methods in Applied Mechanics and Engineering* **137**(2), 175–188.

Kalojanov, J. & Slusallek, P. (2009), 'A parallel algorithm for construction of uniform grids', *Proceedings of the Conference on High Performance Graphics* pp. 1–6.

Karypis, G. & Kumar, V. (1999), 'Parallel multilevel series k-way partitioning scheme for irregular graphs', *Siam Review* **41**(2), 278–300.

LCCV (n.d.), *Demoop - Manual Teórico*.

Lins, B. N., Viana, L. P. & Silva Júnio, M. A. P. (2011), Performance Evaluation of Neighbor Detection Algorithms in Diverse Scenarios, *in* 'CILAMCE', Ouro Preto.

MacK, C. A. (2011), 'Fifty years of Moore's law', *IEEE Transactions on Semiconductor Manufacturing* **24**(2), 202–207.

Markauskas, D. & Kačeniauskas, A. (2015), 'The comparison of two domain repartitioning methods used for parallel discrete element computations of the hopper discharge', *Advances in Engineering Software* **84**, 68–76.

Mingis, K. (2016), 'Intel falls behind Moore's Law, hopes to catch up in '17', *Computerworld* (April), 3–6.

Munjiza, A. (2004), *The Combined Finite-Discrete Element Method*, John Wiley & Sons.

Munjiza, A. & Andrews, K. R. F. (1998), 'NBS contact detection algorithm for bodies of similar size', *International Journal for Numerical Methods in Engineering* **43**(1), 131–149.

Munjiza, A., Bangash, T. & John, N. W. M. (2004), 'The combined finite-discrete element method for structural failure and collapse', *Engineering Fracture Mechanics* **71**(4-6), 469–483.

Nickolls, J. & Dally, W. J. (2010), 'The GPU computing era', *IEEE Micro* **30**(2), 56–69.

Nickolls, J., Buck, I., Garland, M. & Skadron, K. (2008), 'Scalable parallel programming with CUDA', *AMC Queue* **6**(April), 40–53.

Perkins, E. & Williams, J. R. (2001), 'A fast contact detection algorithm insensitive to object sizes', *Engineering Computations* **18**(1-2), 48–61.

Satish, N., Harris, M. & Garland, M. (2009), 'Designing efficient sorting algorithms for manycore gpus', *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium* (May), 1–10.

Zheng, J., An, X. & Huang, M. (2012), 'GPU-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations', *Computers and Structures* **112-113**, 193–204.

Este trabalho foi redigido em LaTeX utilizando uma modificação do estilo IC-UFAL. As referências bibliográficas foram preparadas no programa Mendeley e administradas pelo BibTeX com o estilo LaCCAN. O texto utiliza fonte Fourier-GUT*enberg* e os elementos matemáticos a família tipográfica Euler Virtual Math, ambas em corpo de 12 pontos.