Facens	Sist. Operacionais – CP502	Turma TIN1
	Prof. Marcos Jardini	21/05/2024

Simulação de Física de Partículas em 2D

Teoria e Benefícios da Paralelização

Felipe A. A. Mariano	210045
Felipe R. de Paiva	212031
Kaique M. Govani	210170
Mateus N. V. Matos	211931

Sorocaba
2024

Lista de Figuras

Sumário

1 Introdução

A computação paralela tem se tornado uma área de grande importância e relevância na atualidade, especialmente com o advento de processadores multi-core (múltiplos núcleos) e a necessidade crescente de processamento eficiente em tempo real. Em muitas aplicações, desde simulações científicas até desenvolvimento de jogos, a capacidade de dividir tarefas complexas em subtarefas menores e executá-las simultaneamente pode resultar em melhorias significativas de desempenho.

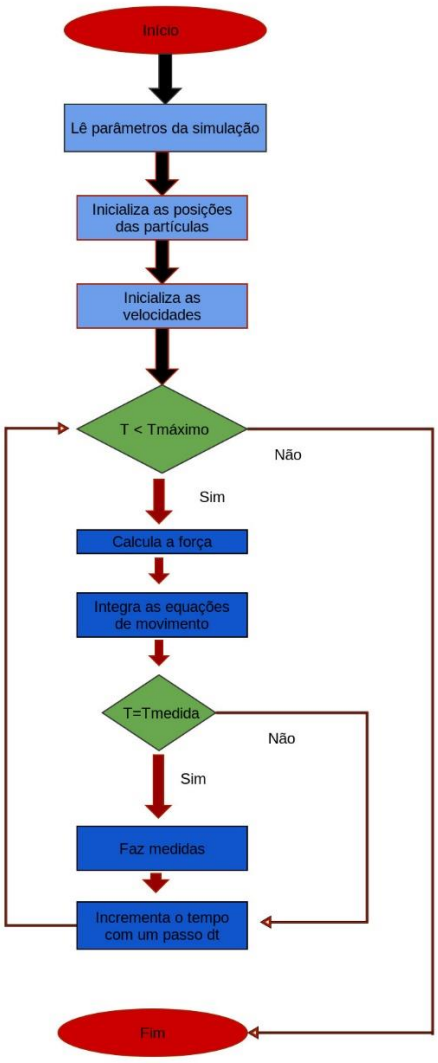
Simulações de física de partículas são ferramentas essenciais para a compreensão de sistemas complexos em que numerosas partículas interagem de acordo com leis físicas estabelecidas. Essas simulações são utilizadas em diversas áreas, como a dinâmica de fluidos, análise de colisões e estudos de materiais granulares. O objetivo principal de uma simulação de física de partículas é prever o comportamento de um sistema de partículas ao longo do tempo, levando em consideração forças como gravidade, colisões e outras interações.

Neste contexto, a utilização de *threads* e processos permite que programas aproveitem ao máximo os recursos disponíveis do hardware, realizando múltiplas operações simultaneamente e, assim, acelerando o tempo de execução e melhorando a capacidade de resposta. Este trabalho tem como objetivo explorar e demonstrar os benefícios da utilização de *threads* em um ambiente de simulação de física de partículas, utilizando a biblioteca *Pygame* para a renderização gráfica.

1.1 Simulação de Física de Partículas em 2D: Teoria e Benefícios da Paralelização

As simulações de partículas em 2D envolvem a modelagem de partículas como pontos ou pequenos discos que se movem em um plano bidimensional. Cada partícula é sujeita a forças externas (como gravidade) e interações com outras partículas (como colisões).

Figura 1 – Fluxograma do Sistema de Simulação de Física de Partículas.



Fonte: Disponível em: https://fiscomp.if.ufrgs.br/index.php/DM:_um_primeiro_programa. Acesso em 2024.

Essas simulações permitem a visualização e análise do comportamento coletivo das partículas, proporcionando *insights* valiosos sobre os fenômenos

físicos subjacentes. Hairer, Lubich e Wanner (2006) enfatizam que a integração numérica geométrica é crucial para preservar as estruturas físicas inerentes ao sistema ao longo do tempo, permitindo simulações mais precisas e estáveis.

Além disso, Metropolis e Ulam (1949) destacam a importância do método Monte Carlo na simulação de sistemas físicos complexos, onde a incerteza e a aleatoriedade desempenham papeis significativos. A abordagem Monte Carlo permite a estimativa de propriedades de sistemas físicos através de amostragem aleatória, sendo particularmente útil em contextos em que métodos analíticos são inviáveis.

Em simulações de partículas, a carga computacional é significativa devido ao grande número de cálculos necessários para atualizar as posições e detectar colisões das partículas em cada iteração. A paralelização desses cálculos por meio de *threads* pode diminuir substancialmente o tempo total de execução. Cada *thread* pode ser responsável por uma parte do espaço simulado ou por um subconjunto de partículas, permitindo que os cálculos sejam realizados simultaneamente.

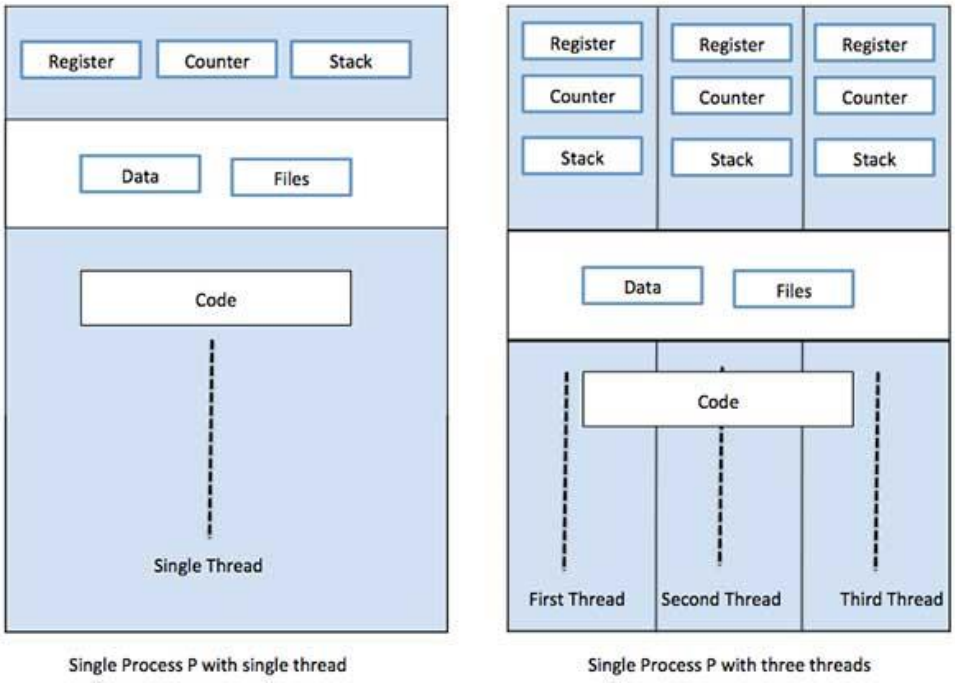
Bjarne Stroustrup, em seu livro The C++ Programming Language (2013), enfatiza que a programação *multithreaded* (múltiplas *threads*) é uma técnica poderosa para melhorar o desempenho dos programas, especialmente em sistemas com múltiplos núcleos de processamento. A capacidade de executar tarefas em paralelo permite que programas façam um uso mais eficiente do *hardware* disponível, reduzindo o tempo necessário para completar tarefas computacionalmente intensivas.

De maneira similar, Andrew S. Tanenbaum e Herbert Bos, em Modern Operating Systems (2014), discutem como a criação de processos e *threads* pode ser utilizada para aproveitar o processamento paralelo. Eles destacam que, enquanto processos oferecem isolamento e segurança, *threads* oferecem uma maneira mais leve de paralelizar tarefas, permitindo que recursos sejam compartilhados mais facilmente entre diferentes linhas de execução dentro do mesmo processo.

1.2 *Threads* e Processos: Conceitos e Funcionamento

Threads são sequências de execução que compartilham o mesmo espaço de memória dentro de um processo. Elas permitem que múltiplas tarefas sejam executadas simultaneamente dentro de um mesmo programa, o que é especialmente útil para operações que podem ser realizadas de maneira independente ou em paralelo. Em linguagens como *Python*, o a biblioteca *threading* fornece uma maneira simples de criar e gerenciar *threads*.

Figura 2 – Comparação entre um processo com uma e várias threads.



Fonte: Disponível em https://www.tutorialspoint.com/operating_system/os_multi_threading.htm, Acesso em 2024.

Threads são ideais para tarefas que requerem comunicação rápida e frequente entre diferentes partes do programa, pois os *threads* dentro de um processo compartilham o mesmo espaço de memória. Isso facilita a troca de dados e a coordenação entre os *threads*, reduzindo a sobrecarga associada à comunicação entre processos separados. No entanto, esse compartilhamento de memória também pode levar a problemas de sincronização, como condições de corrida, onde duas ou mais *threads* tentam acessar e modificar o mesmo recurso simultaneamente.

Processos são unidades de execução independentes, cada uma com seu próprio espaço de memória. A criação de processos filhos através do *fork* é uma técnica comum em sistemas UNIX. O comando *fork* cria um processo que é uma cópia do processo pai. Embora isso permita um isolamento completo entre processos, a comunicação entre eles requer mecanismos como *pipes* ou *sockets*, o que pode introduzir complexidade adicional.

Os processos são particularmente úteis em situações em que o isolamento é crítico, como em servidores *web* onde cada solicitação de cliente pode ser tratada por um processo separado, garantindo que uma falha em um processo não afete os outros. No entanto, a criação de processos é mais custosa em termos de recursos do sistema do que a criação de *threads*, devido à necessidade de duplicar o espaço de memória e outros recursos do processo pai.

1.3 Integração de Verlet na Simulação de Física de Partículas

A integração de Verlet recebe seu nome em homenagem ao físico francês Loup Verlet, que a introduziu em 1967. Verlet desenvolveu este método enquanto estudava a dinâmica de líquidos utilizando simulações computacionais. Seu trabalho pioneiro foi fundamental para a área de dinâmica molecular, permitindo simulações mais precisas e estáveis de sistemas de partículas.

1.3.1 Introdução à Integração de Verlet

A integração de Verlet é um método numérico utilizado para resolver equações diferenciais que descrevem o movimento de partículas. É amplamente empregada em simulações de dinâmica molecular e outros sistemas físicos devido à sua simplicidade e precisão na conservação de energia ao longo do tempo. O método é especialmente eficaz para problemas em que forças são derivadas de potenciais, como no caso de partículas sujeitas à gravidade e colisões elásticas.

1.3.2 Funcionamento da Integração de Verlet

A integração de Verlet calcula a posição das partículas em momentos discretos de tempo, usando as posições em tempos anteriores.

Este método utiliza as posições passadas e a aceleração atual para calcular a nova posição, evitando a necessidade de calcular velocidades explicitamente. Isso resulta em maior estabilidade e conservação de energia no sistema simulado.

As equações para x(t) e v(t), representando a posição da partícula e a velocidade dado o tempo t e variação do tempo Δt, respectivamente, podem ser descritas da seguinte forma:

$$x(t + \Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 \quad (1)$$

$$v(t + \Delta t) = v(t) + \frac{a(t) + a(t + \Delta t)}{2} * \Delta \mathrm{t} \quad (2)$$

Assim, pode-se substituir a velocidade da equação 1 com a equação 2, e aplicando o cálculo num espaço vetorial de 2 dimensões, obtém-se a seguinte formula para a posição vetorial $\vec{r}$ da partícula:

$$\vec{r}(t + \Delta t) = 2\vec{r}(t) - \vec{r}(t - \Delta t) + \vec{a}(t) * \Delta t^2 \quad (3)$$

Onde:

- $\vec{r}(t)$: é $a\ posição\ da\ partícula\ no\ tempo\ t$,
- Δt: é $o\ passo\ de\ tempo$,
- $\vec{a}(t)$: é $a\ aceleração\ da\ partícula\ no\ tempo\ t$.

1.3.3 Vantagens da Integração de Verlet

A integração de Verlet possui várias vantagens que a tornam adequada para simulações de física de partículas:

Estabilidade e Conservação de Energia: A integração de Verlet é conhecida por sua capacidade de conservar a energia total do sistema, o que é crucial para simulações de longo prazo. Essa característica é particularmente importante em sistemas onde a precisão da energia é essencial para a fidelidade da simulação.

Simplicidade de Implementação: O algoritmo de Verlet é simples de implementar, pois requer apenas as posições e acelerações das partículas, sem a necessidade de armazenar ou calcular velocidades diretamente.

Baixa Sensibilidade a Erros Numéricos: A integração de Verlet tende a ser menos sensível a erros numéricos acumulados ao longo do tempo, o que resulta em simulações mais precisas e estáveis.

1.3.4 Aplicação na Simulação de Física de Partículas

Na simulação de física de partículas em 2D, a integração de Verlet é utilizada para atualizar as posições das partículas a cada passo de tempo. A força resultante sobre cada partícula, que pode incluir forças de gravidade, força

de colisão e outras interações, é calculada para determinar a aceleração atual. Em seguida, as novas posições das partículas são determinadas usando a fórmula de Verlet.

1.3.5 Benefícios da Paralelização com Integração de Verlet

O uso de *threads* na simulação pode aumentar significativamente a eficiência computacional. Cada *thread* pode ser responsável por um subconjunto de partículas, calculando suas novas posições e acelerações em paralelo. Como a integração de Verlet é computacionalmente leve e não requer cálculos de velocidade explícitos, ela se presta bem à paralelização.

Ao distribuir a carga de trabalho entre múltiplas *threads*, a simulação pode aproveitar melhor os recursos do sistema, reduzindo o tempo total de execução. Isso é particularmente vantajoso em simulações de grande escala, onde o número de partículas é elevado e os cálculos são intensivos.

2 Preparação para desenvolvimento

2.1 Ambiente de programação

Para a implementação da simulação de física de partículas em 2D, será utilizado *Python* como a linguagem de programação principal, aproveitando-se do módulo *threading* para a criação e gerenciamento de *threads*. O ambiente de desenvolvimento utilizara a biblioteca Pygame, para renderização, e incluirá bibliotecas adicionais como *NumPy* para cálculos numéricos eficientes e *Matplotlib* para visualização gráfica.

O trabalho será particionado com base nas seguintes diretrizes:

Divisão de Espaço de Simulação: O espaço 2D será dividido em sub-regiões, cada uma gerenciada por uma *thread* separada. Isso permitirá que os cálculos de movimento e colisão sejam realizados em paralelo para diferentes partes do espaço simulado.

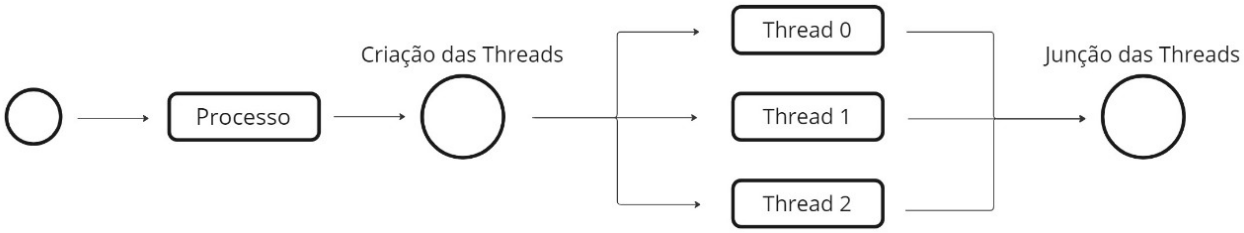
Atribuição de Partículas: As partículas serão distribuídas entre as *threads* com base em suas posições iniciais. Cada thread será responsável por atualizar as posições e detectar colisões para as partículas em sua sub-região.

Sincronização e Comunicação: As *threads* se comunicarão periodicamente para trocar informações sobre partículas que se movem entre sub-regiões. Esta comunicação será gerenciada através de filas de mensagens compartilhadas.

Controle de Execução e Comunicação: O controle da execução das *threads* será gerenciado por uma *thread* principal que coordena a inicialização e a sincronização das *threads* trabalhadoras. As mensagens entre *threads*, que incluem informações sobre partículas que atravessam as fronteiras das sub-regiões, serão trocadas em intervalos regulares utilizando filas de mensagens seguras para *threads*.

A Figura 2, ilustra a separação das *threads* e fluxo do programa:

Figura 3 – Diagrama funcionamento de threads.



Fonte: Elaborada pelos autores, 2024.

2.2 Especificações de Partição e Controle

Para garantir um desempenho ótimo, o trabalho de simulação será particionado de forma que cada *thread* tenha aproximadamente a mesma carga de trabalho. Isso será monitorado dinamicamente, redistribuindo partículas se necessário para balancear a carga entre as *threads*. O controle da execução incluirá verificações periódicas do estado de cada *thread* e ajustes na alocação de recursos conforme necessário.

2.3 Monitoramento e Ajustes Dinâmicos

O monitoramento do desempenho será feito através de métricas como o tempo médio de execução por iteração e a utilização de *CPU* de cada *thread*. Ajustes serão realizados automaticamente pelo sistema para redistribuir partículas entre *threads* se um desequilíbrio significativo for detectado.

2.4 Diagramas e Figuras

Os diagramas e fluxogramas incluídos no texto ajudam a visualizar a organização e o fluxo de controle do sistema de *threads*. Eles são fundamentais para entender como as *threads* se comunicam e sincronizam, garantindo uma execução eficiente e precisa da simulação.

Em conclusão, a preparação cuidadosa e a organização estruturada do ambiente de programação são essenciais para o sucesso da implementação de uma *engine* de simulação de física de partículas em 2D, utilizando técnicas de paralelização para maximizar a eficiência e reduzir o tempo de execução.

3 Referências

Metropolis, N., & Ulam, S. 1949. *The Monte Carlo Method. Journal of the American Statistical Association.*

Verlet, L. 1967. *"Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules." Physical Review, 159(1), 98-103.*

Hairer, E., Lubich, C., & Wanner, G. 2006. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations. Springer.*

Tanenbaum, A. S., & Bos, H. 2014. *Modern Operating Systems. Pearson.*

Stroustrup, B. 2013. *The C++ Programming Language. Addison-Wesley.*

Burns, A., & Wellings, A. 2016. *Concurrent and Real-Time Programming in Ada. Cambridge University Press.*

Schroeder, D. V. 2022. *Physics Simulations in Python: A Lab Manual. Departamento de Física, Weber State University. Disponível em https://physics.weber.edu/schroeder/scicomp/PythonManual.pdf.*