Construção Paralela Lock-Free de Octrees Esparsas em GPU

Michel Brasil Cordeiro [1]

Wagner M. Nunan Zola [1]

[1]Departamento de Informática, UFPR

24 de abril de 2024

Sumário

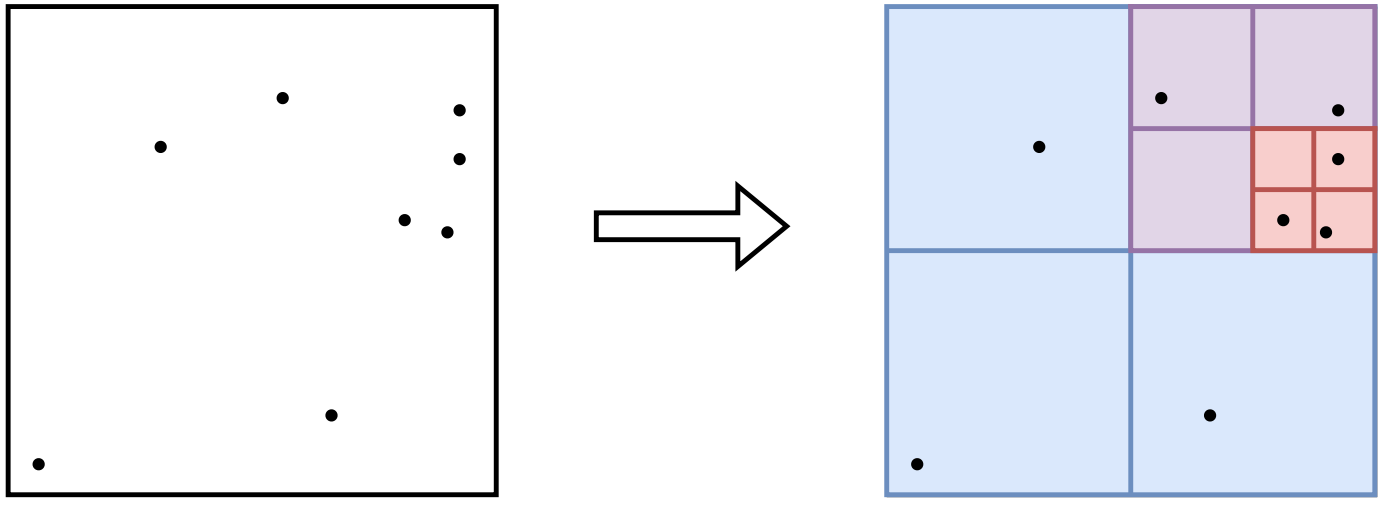
Octree (3D)

- Octree é uma estrutura de dados de árvore na qual cada nó interno possui oito filhos
- São usadas principalmente para representar e organizar dados tridimensionais
- Cada nó interno divide recursivamente o espaço tridimensional em oito octantes até que alguma condição pré-definida seja satisfeita

Octree (3D) vs Quadtree (2D)

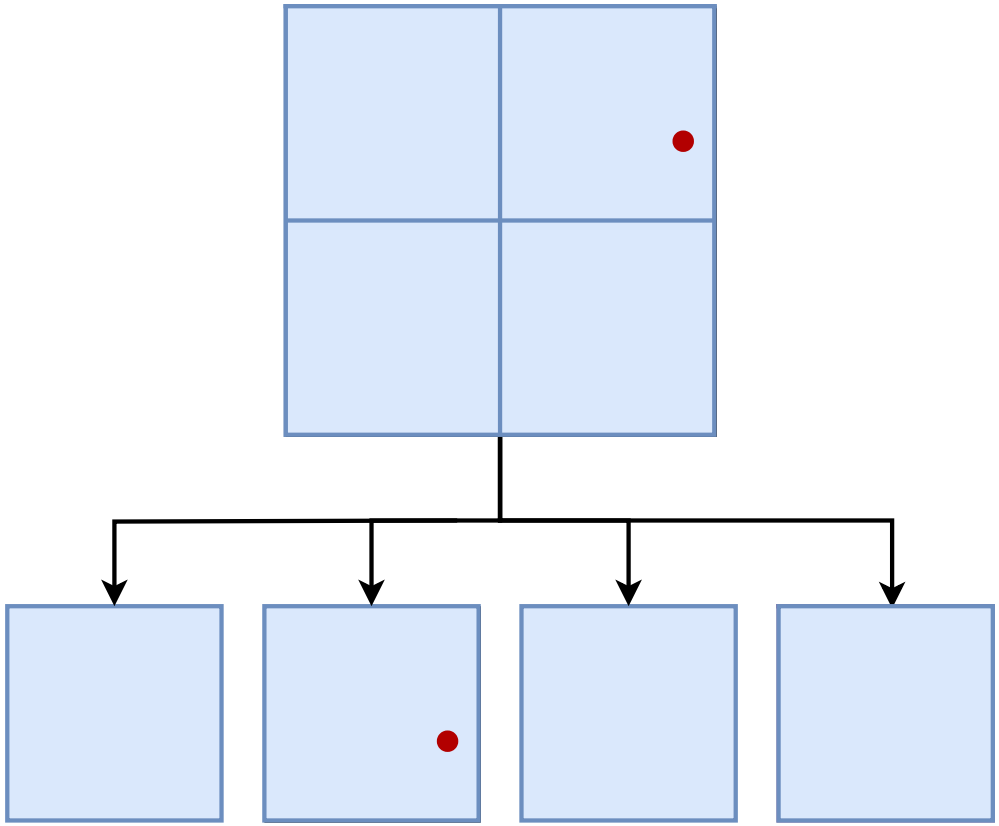
- Em uma Quadtree, cada nó interno possui **quatro** filhos
- São usadas para representar e organizar dados **bidimensionais**

Introdução



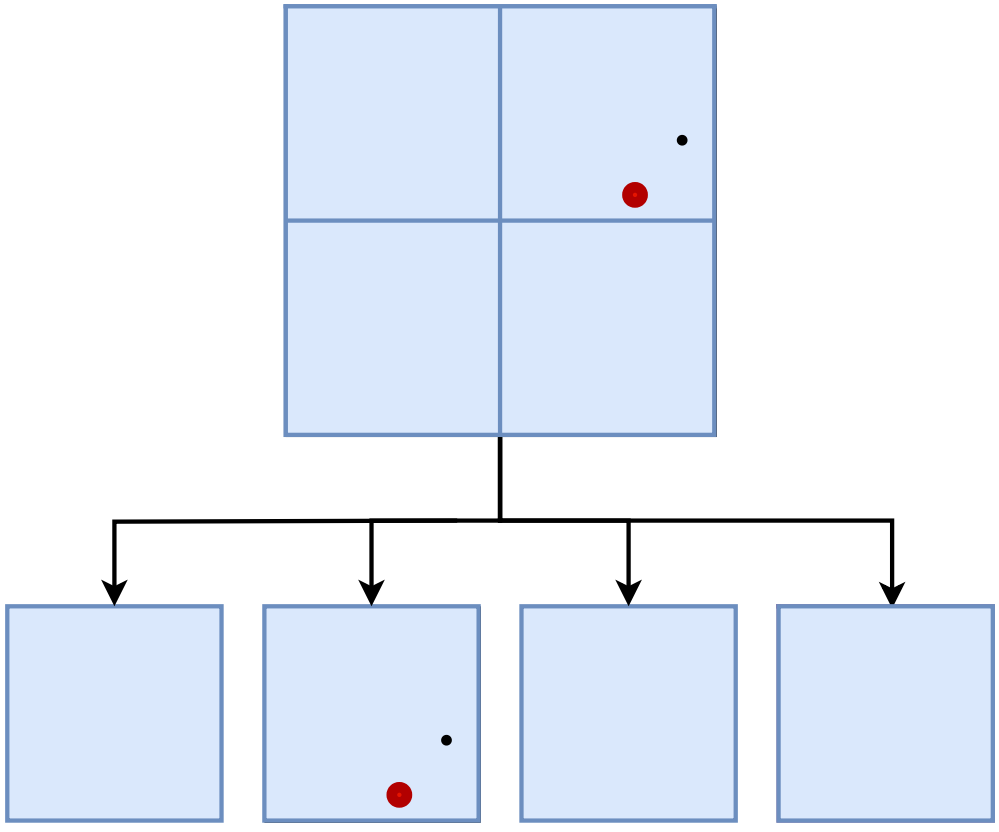
- Cada nodo folha da árvore pode possuir no máximo um corpo
- Os nodos dividem o plano (ou espaço, no caso de octrees) até que não haja mais de um corpo em cada subdivisão

Construção da Árvore



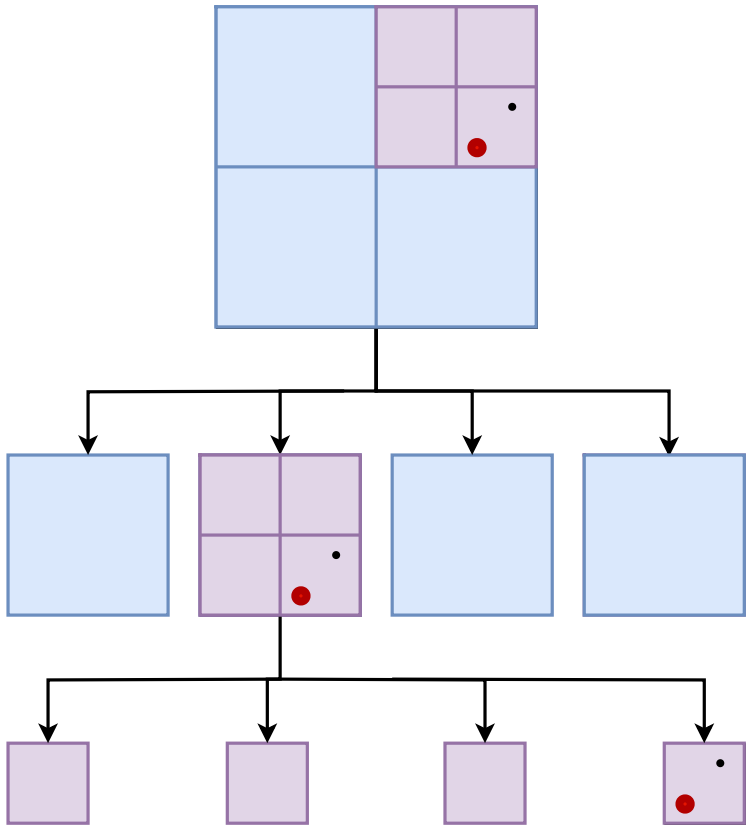
- Ao incluir um novo corpo, calcula-se o quadrante em que ele está localizado
- Se o quadrante estiver vazio, basta incluir o corpo

Construção da Árvore



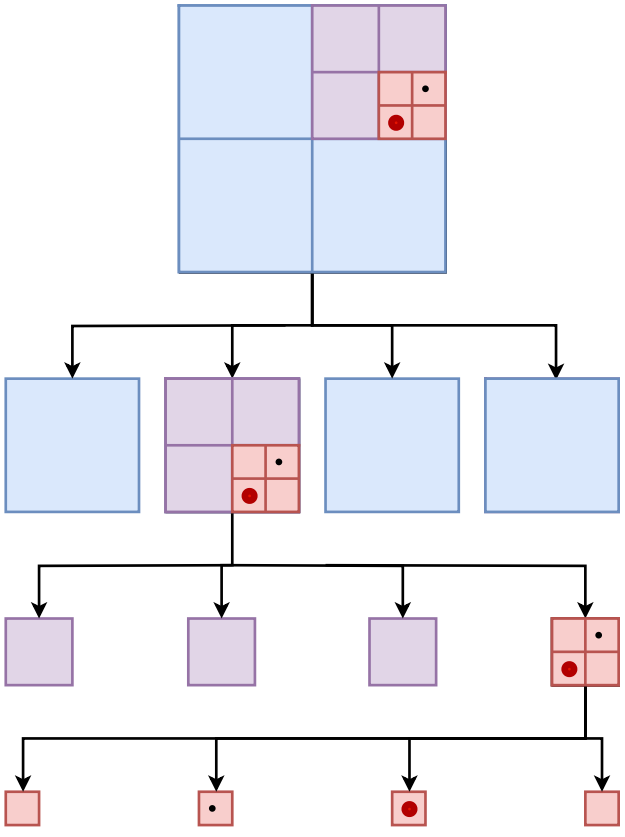
- Se o quadrante estiver ocupado, ele deve ser divido para que os corpos fiquem em quadrantes diferentes

Construção da Árvore



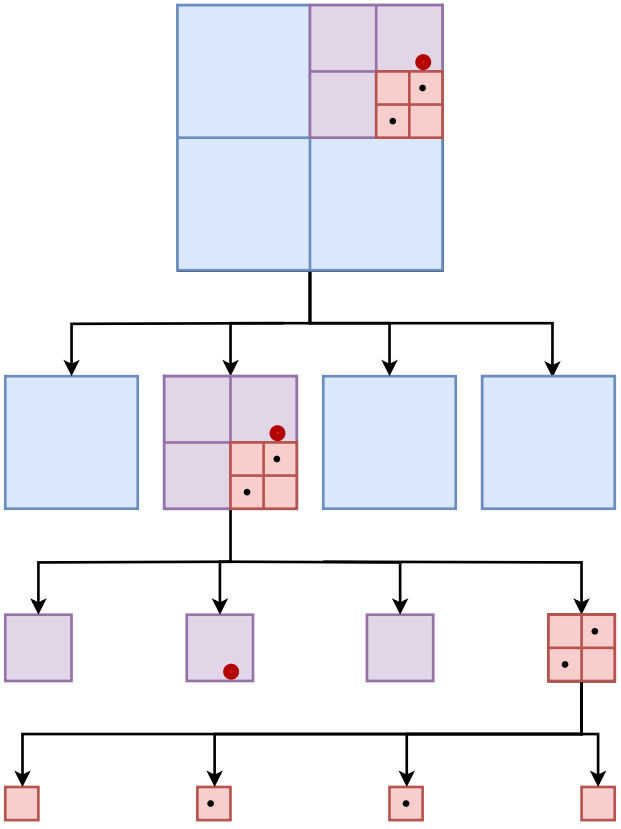
- Se após a divisão, os corpos ainda estiverem no mesmo quadrante, o novo quadrante deverá ser subdivido

Construção da Árvore



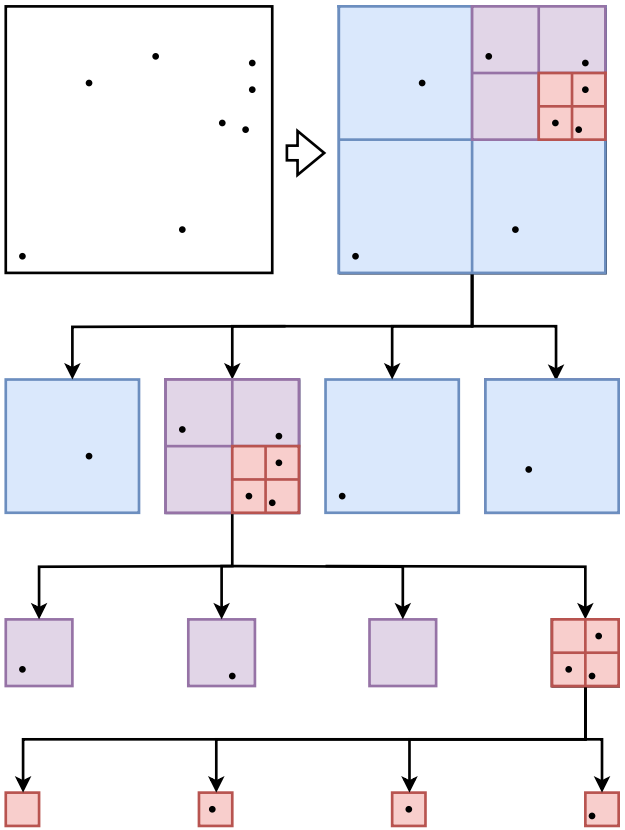
- Esse processo se repete até que os corpos fiquem em quadrante diferentes

Construção da Árvore



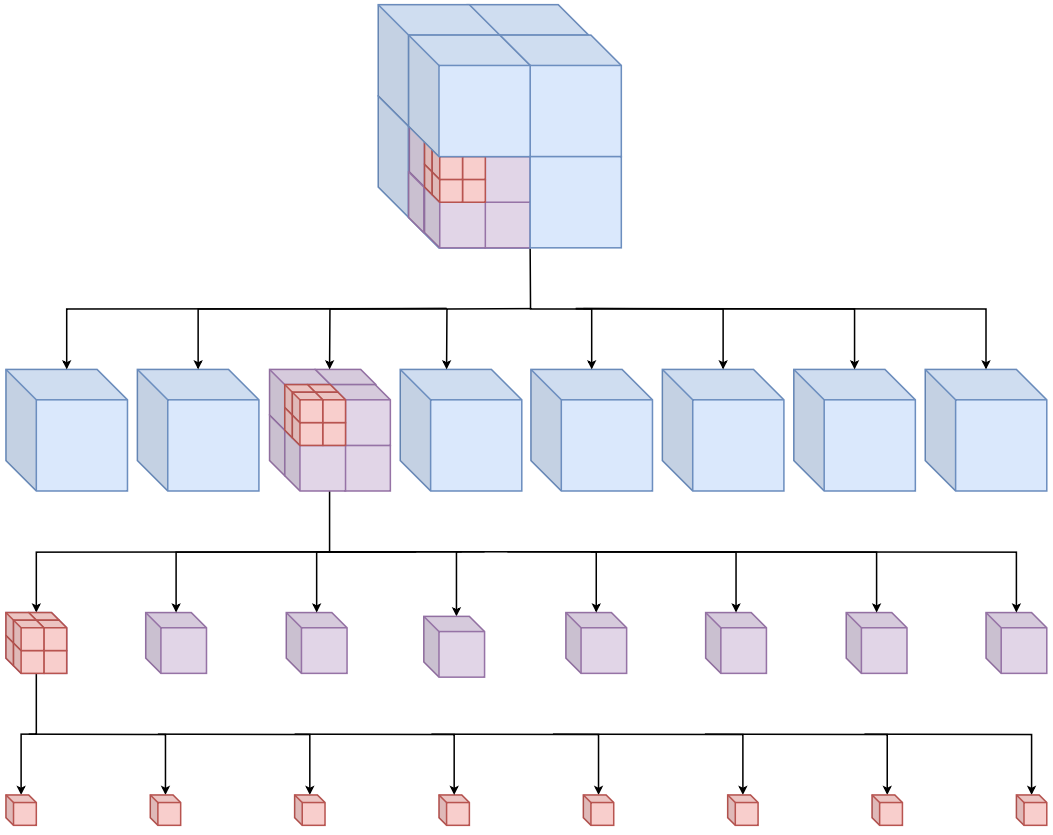
- A cada inclusão, o algoritmo deve percorrer a árvore até um nodo folha

Construção da Árvore



- Sendo assim, cada nodo pode ser:
  - Nodo interno
  - Nodo livre
  - Nodo ocupado

Construção da Octree



- A construção da Octree ocorre de forma semelhante, dividindo o espaço em octantes ao invés de quadrantes

Trabalhos Relacionados:

An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm:

- Autores: Martin Burtscher e Keshav Pingali
- Apresenta implementação eficiente do algoritmo de Barnes Hut para problemas de n-corpos usando o paralelismo em GPUs
- O algoritmo utiliza uma octree para agrupar os corpos e calcular as interações gravitacionais
- A construção da árvore é feita a cada iteração do algoritmo

Construção da Árvore (Versão Paralela)

- Cada thread recebe um corpo para inserir
- Existe condição de corrida quando mais de uma thread tenta inserir no mesmo nodo
- Para resolver esse problema, o algoritmo de Burtscher e Pingali utiliza locks com espera ocupada (em loop)
  - insere barreiras de sincronização para minorar enorme quantidade de acessos à memoria geradas pela interrogação contínua dos locks até sua liberação

Construção da Árvore (Burtscher e Pingali)

Algorithm 1 Construção da árvore de Burtscher e Pingali

```
1:  inserção_bem-sucedida ← true
2:  while existirem corpos a serem incluídos na árvore do
3:      if inserção_bem-sucedida then
4:          corpo_inserir ← novo corpo para inserir
5:          inserção_bem-sucedida ← false
6:      end if
7:      percorre a árvore até um nodo folha
8:      nodo_inserção ← nodo em que o corpo deverá ser inserido
9:      if nodo estiver livre then
10:         nodo_inserção ← corpo_inserir
11:         inserção_bem-sucedida ← true
12:     end if
```

Construção da Árvore (Burtscher e Pingali)

```
13:    if nodo estiver ocupado then
14:       if lock(nodo_inserção) then
15:          while corpos estiverem no mesmo octante do
16:             subdivide o espaço, expandindo a árvore
17:          end while
18:          unlock(nodo_inserção)
19:          inserção_bem-sucedida ← true
20:       end if
21:    end if
22:    // espera ocupada: se nodo estiver trancado, volta no loop para tentar inserir novamente
23:    sincronizaThreads()   // sincroniza para diminuir tráfego em memória na espera ocupada
24: end while
```

OctreeBuild-LF

- Este trabalho apresenta o algoritmo **OctreeBuild-LF**, que utiliza operações atômicas para remover os locks e as barreiras de sincronização
- Para isso, será utilizada a primitiva CUDA AtomicCAS (Atomic Compare and Swap), que realiza uma escrita atômica caso o valor passado por parâmetro seja igual ao valor armazenado na variável que está sendo sobrescrita

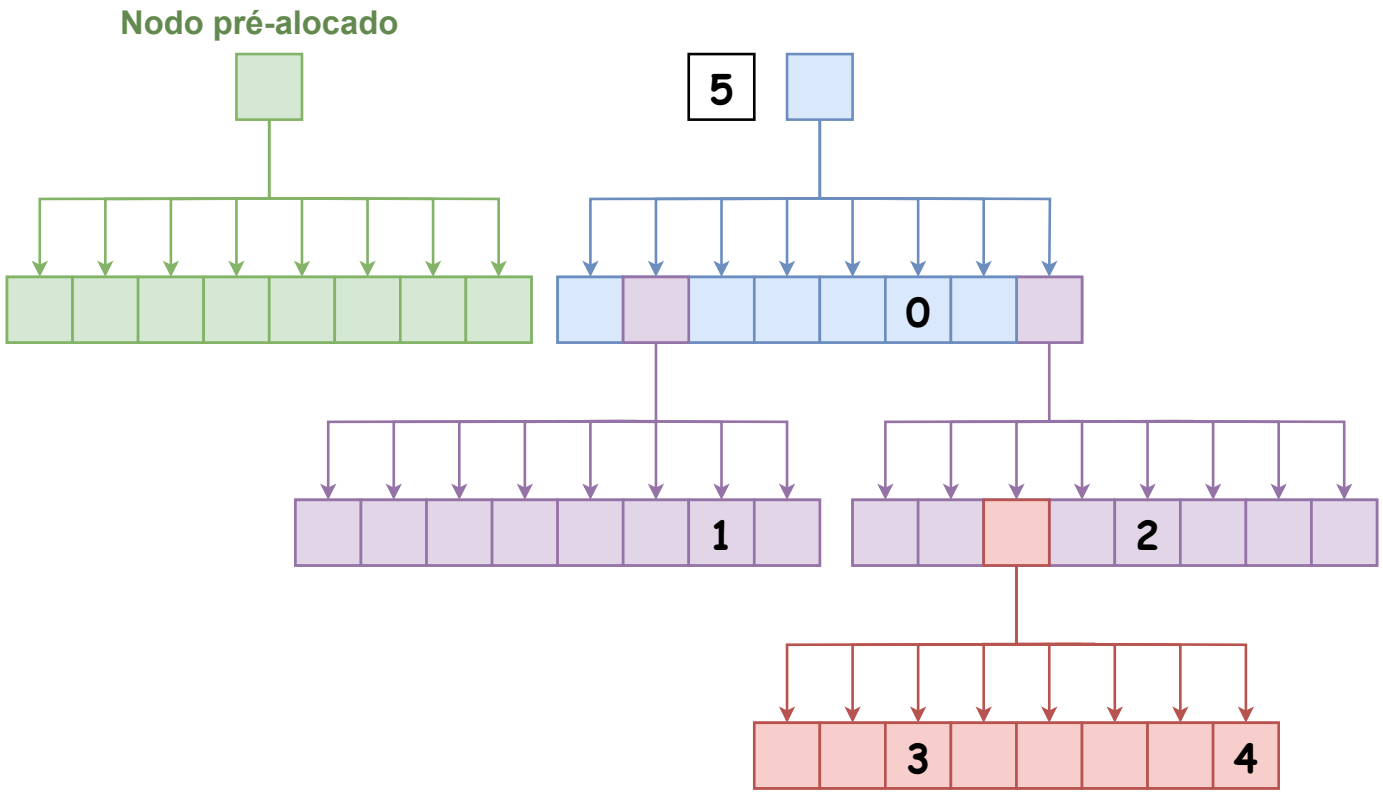
OctreeBuild-LF

- atomicCAS(address, compare, val)
  - **address**: é o endereço da memória que será atualizado
  - **compare**: é o valor a ser comparado com o valor atual no endereço
  - **val**: é o valor que será escrito no endereço se o valor atual for igual a compare
- atomicCAS retorna o valor armazenado em **address**

OctreeBuild-LF

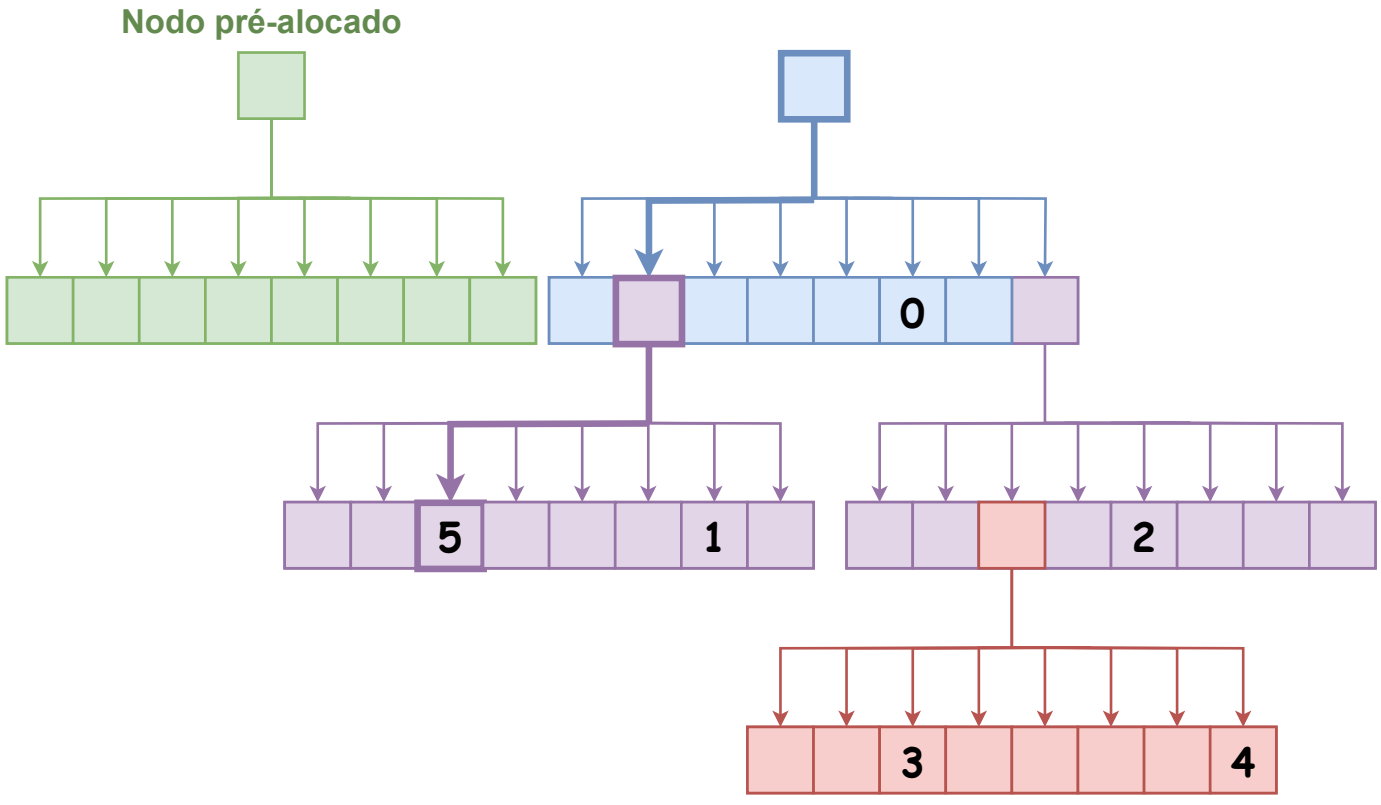
- Utilizando a construção:
  atomicCAS(&*nodo*, *valor_lido_pela_thread*, *corpo_a_ser_inserido***)** == *valor_lido_pela_thread*
- é possível alterar os nodos da árvore de forma atômica
- Se a comparação for verdadeira, a alteração foi realizada na árvore
- Se a comparação for falsa, a alteração não pode ser feita, pois o valor armazenado no nodo não é igual ao valor do nodo que tinha sido lido pela thread (ou seja, o nodo foi alterado)

OctreeBuild-LF



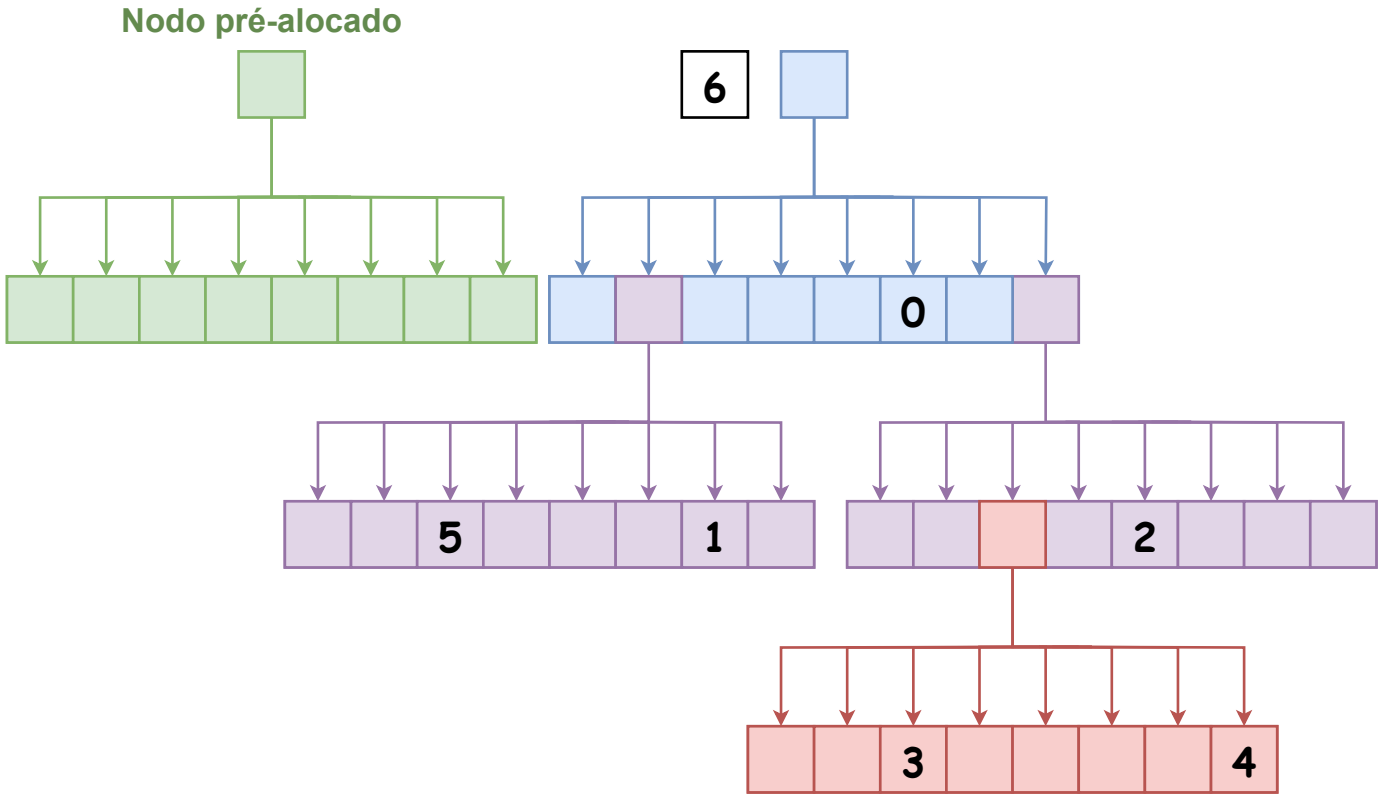
- No início do algoritmo, cada thread possui um nodo pré-alocado
- Cada thread recebe um corpo para inserir na árvore

OctreeBuild-LF



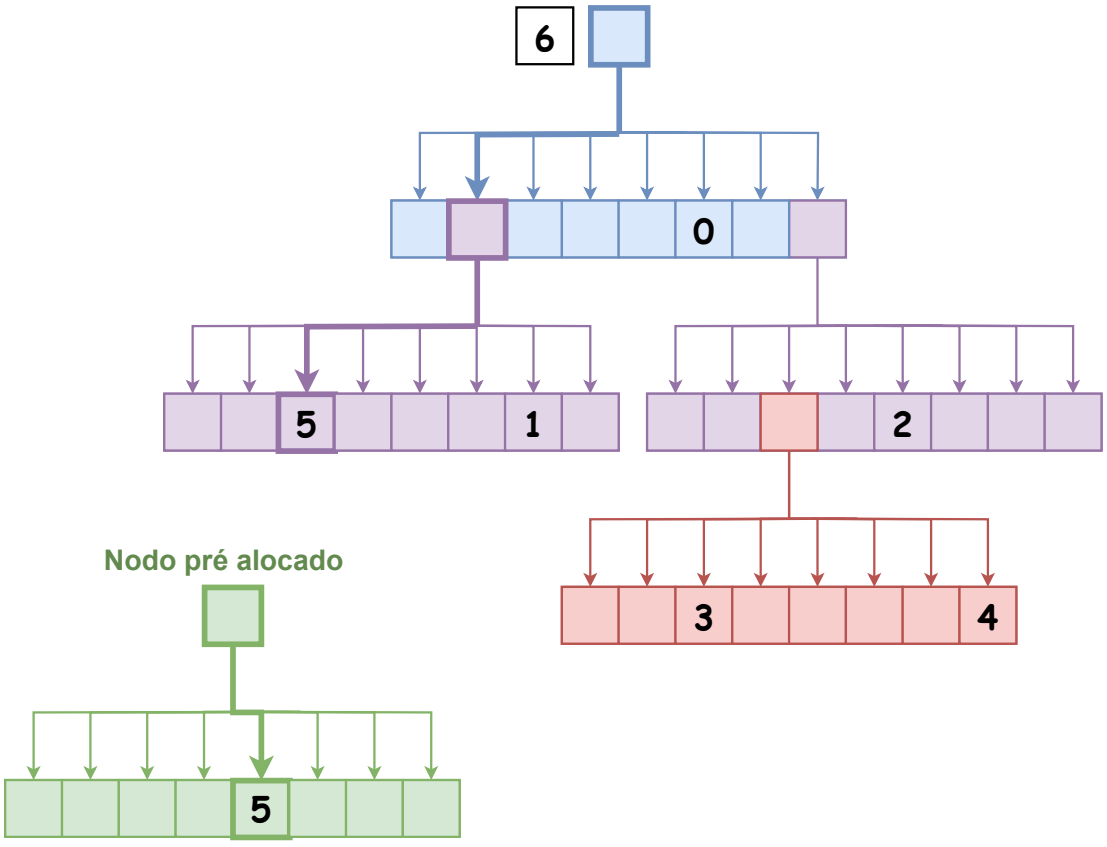
- A thread então percorre a árvore até encontrar um nodo folha

OctreeBuild-LF



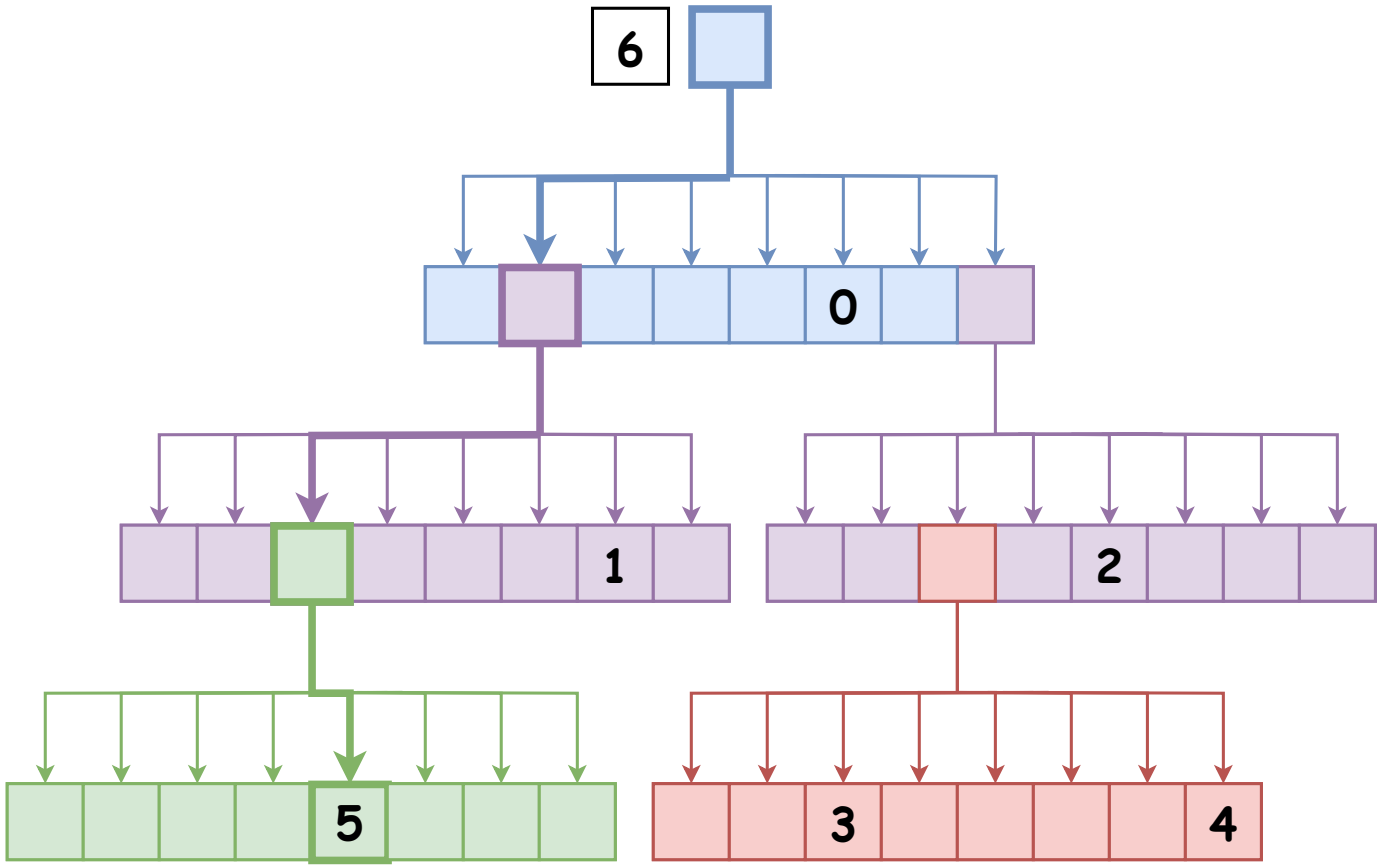
- Se o nodo estiver livre, basta inserir utilizando a operação atomicCAS

OctreeBuild-LF



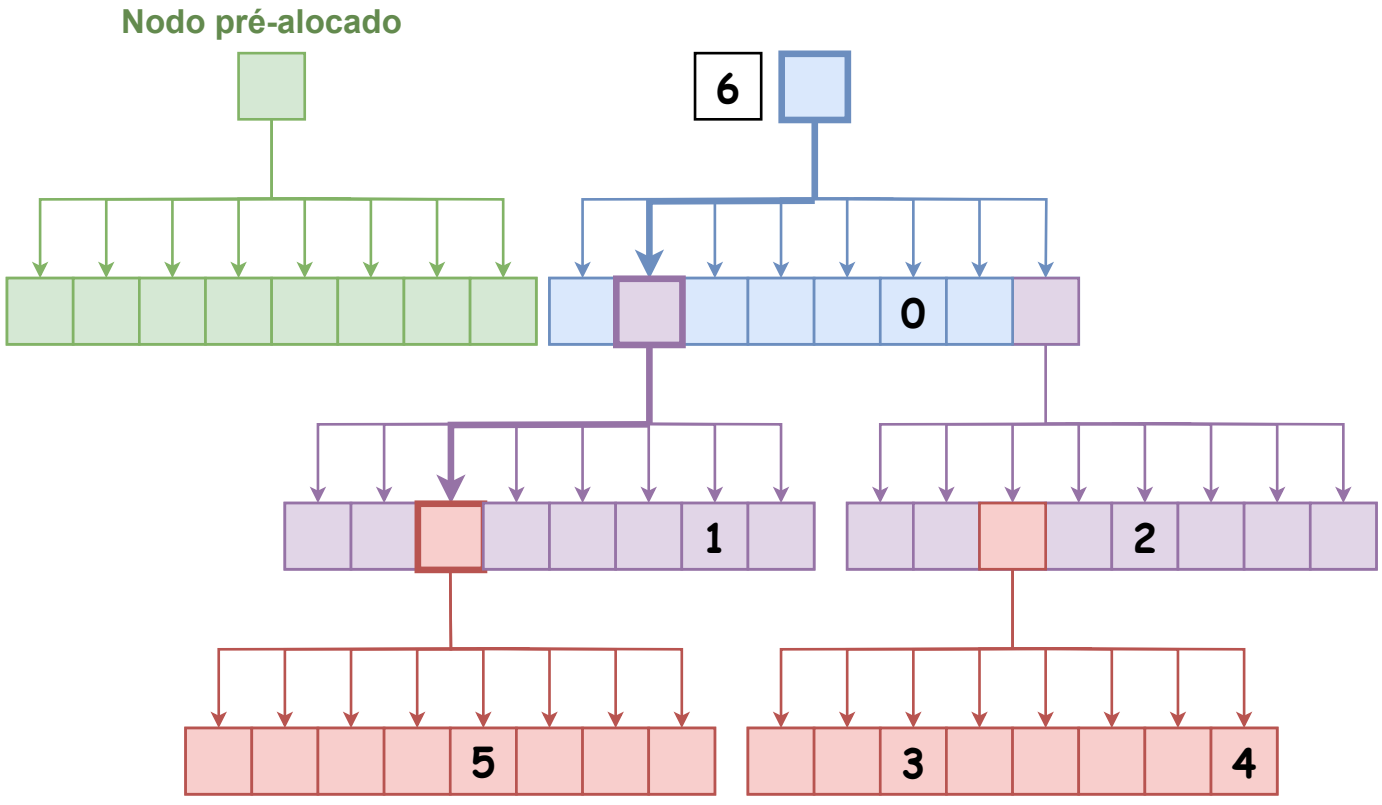
- Se o nodo estiver ocupado por um corpo, a thread armazena o corpo encontrado no nodo pré-alocado

OctreeBuild-LF



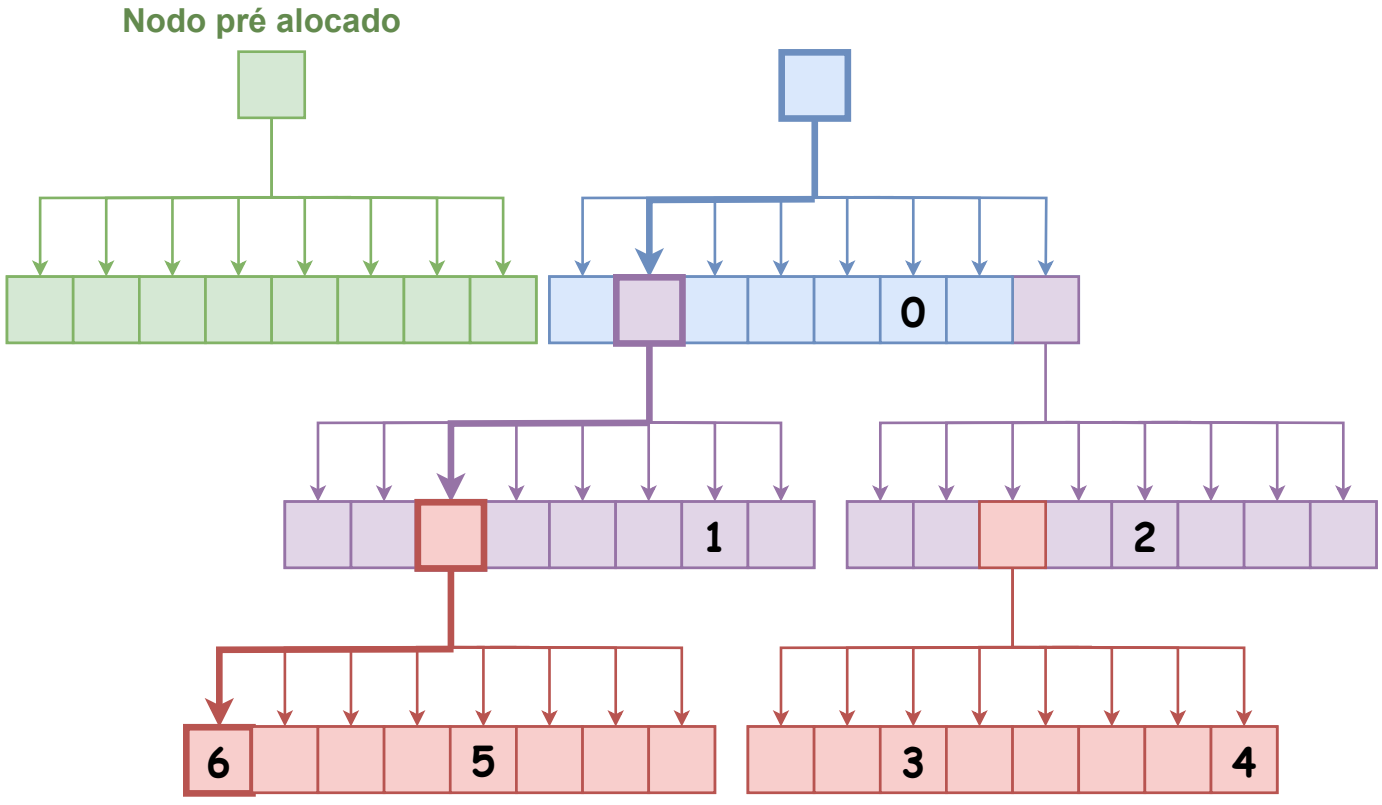
- Então, a thread insere atomicamente o nodo pré-alocado na árvore, utilizando a operação atomicCAS, e pré-aloca um novo nodo

OctreeBuild-LF



- Então, uma nova tentativa de inserção é realizada

OctreeBuild-LF



- Até que o corpo seja inserido em nodo que esteja livre

OctreeBuild-LF: Vantagens

- As vantagens de construir a árvore dessa forma são:
  - Não há necessidade de sincronizar as threads e, portanto, as threads não ficam esperando em uma barreira a cada tentativa de inserção
  - Atualizar a árvore a cada subdivisão permite que as alterações fiquem visíveis para as threads muito mais rapidamente, aumentando a cooperação entre elas

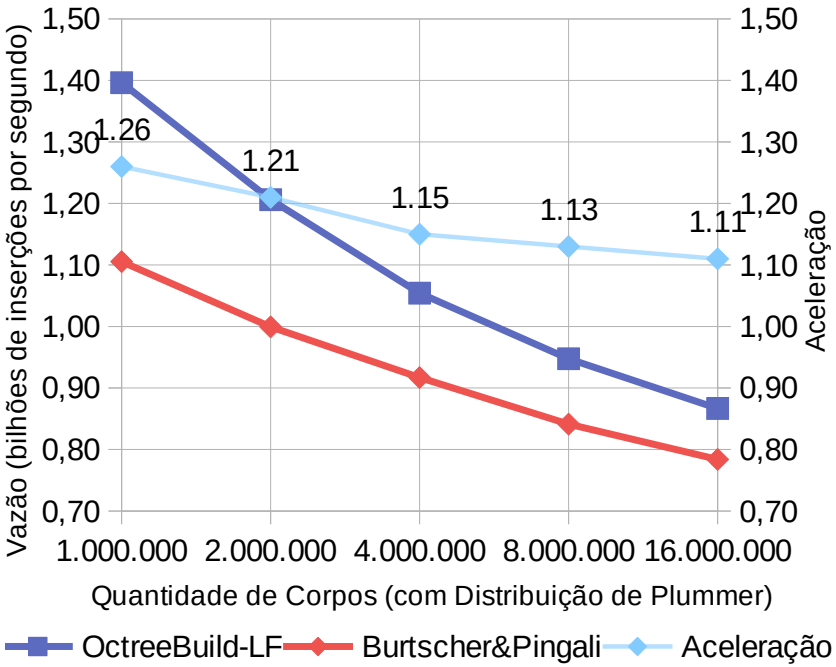
Metodologia

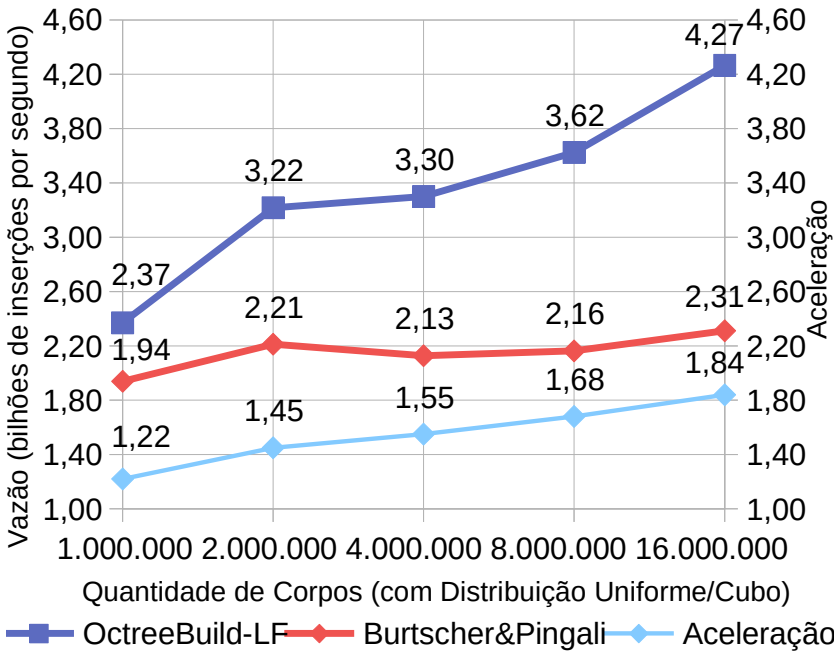
- A máquina utilizada para realizar os experimentos possui:
  - processador Intel Xeon Silver 4314 CPU @ 2.40GHz com 16 núcleos
  - 32GB de RAM
  - GPU NVIDIA A4500 que possui 56 multiprocessadores CUDA
- Cada experimento foi realizado 30 vezes e então reportada a vazão média de consultas por segundo
- Também foram calculados os intervalos com nível de confiança de 95%, mas não foram observadas valores maiores que 2% em relação a média

Metodologia

- Dois cenários são considerados:
  - *i*) os corpos estão distribuídos seguindo a distribuição de Plummer, com os dados dispostos aleatoriamente na memória e densidade variável;
  - *ii*) os corpos estão organizados em cubo uniforme no espaço, onde corpos próximos no espaço possuem dados próximos na memória.
- A distribuição *ii* é interessante, pois avalia cenários onde a entrada de dados apresenta maior localidade na distribuição e maior concorrência entre threads
- Os experimentos foram executados com entrada de 1, 2, 4, 8 e 16 milhões de corpos
- O OctreeBuild-LF foi comparado com o algoritmo de Burtscher e Pingali

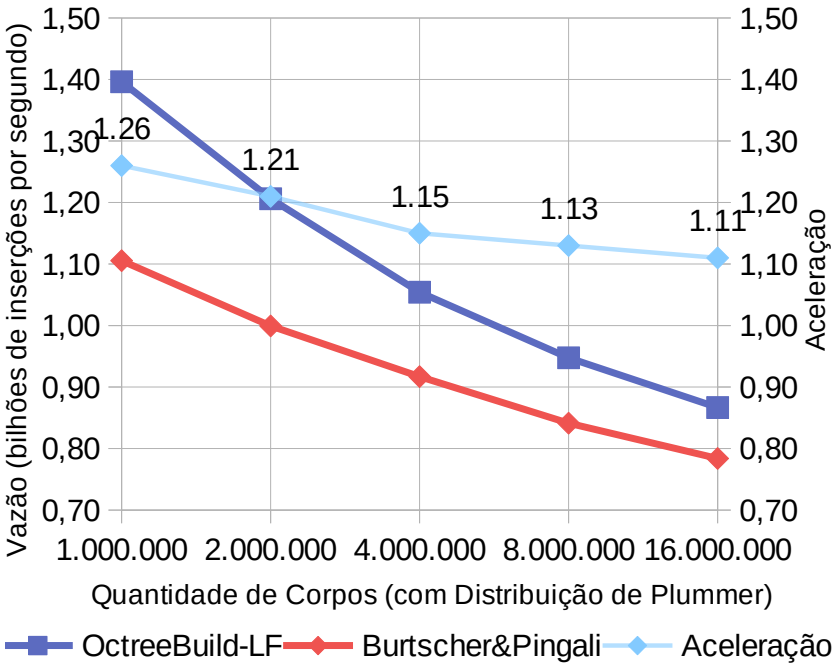
Resultados: bilhões de inserções / segundo

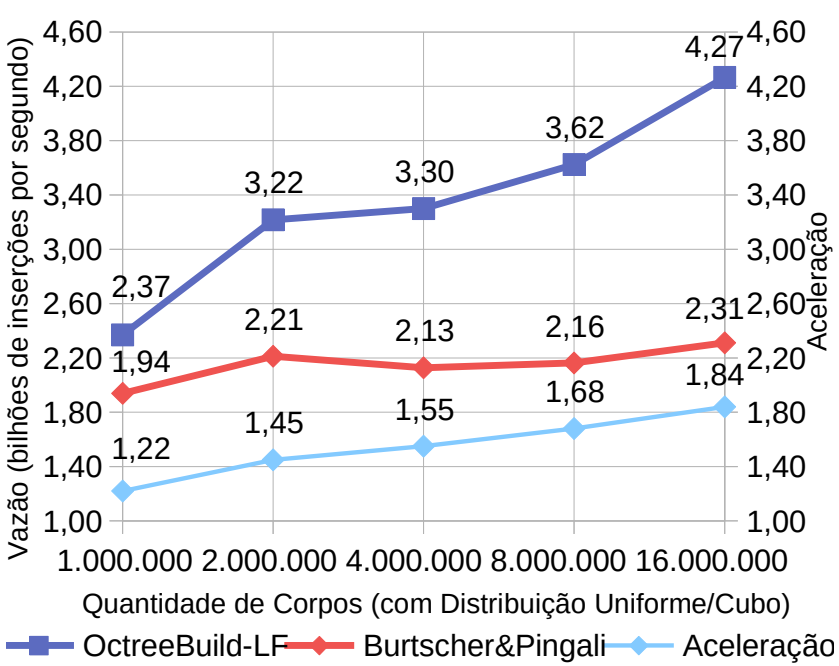




- OctreeBuild-LF apresenta maior vazão em ambos os cenários avaliados, alcançando uma aceleração de até 1,84 para 16 milhões de corpos na distribuição uniforme
- Isso demonstra a eficiência do algoritmo, principalmente quando há uma maior concorrência entre as threads

Resultados: bilhões de inserções / segundo





- Quando os dados estão armazenados aleatoriamente na memória, a concorrência diminui conforme a quantidade de corpos aumenta, reduzindo a aceleração obtida
- O aumento no tamanho da árvore também causa um aumento na quantidade de acessos irregulares à memória, diminuindo a vazão para ambos os algoritmos

Conclusões

- Este trabalho apresentou o algoritmo **OctreeBuild-LF** para construção de octrees utilizando paralelismo em GPUs
- O OctreeBuild-LF **elimina a necessidade de locks e barreiras** ao utilizar operações atômicas para resolver as condições de corridas
- O algoritmo proposto foi capaz de alcançar **aceleração de até 1,84** em relação ao algoritmo de Burtscher e Pingali, que utilizam locks e barreiras

Trabalhos Futuros

- Testar os algoritmos em outras distribuições, incluindo testes com base de dados reais
- Utilizar a memória compartilhada da GPU para obter maior desempenho

Muito Obrigado