Karlsruher Institut für Technologie

Parallele Algorithmen
Einschub Shared Memory Datenstrukturen

Vorlesung · January 11, 2017

Tobias Maier

INSTITUT FÜR THEORETISCHE INFORMATIK · LEHRSTUHL ALGORITHMIK

Stand der Vorlesung

Globale Kommunikations-Muster

- Broadcast
- Reduktion
- All-to-All

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Einschub - Shared Memory

Karlsruher Institut für Technologie

Globale Kommunikations-Muster

- Trivial?
- Auch schnell?

Alternativen zu globaler Komunikation

- Asynchroner Informationsaustausch
- Datanstrukturen

Motivation

- Shared Memory vs. Distributed Memory
  $\rightarrow$ unterschiedliche Bedürfnisse
- Jeder moderne Prozessor
  ist eine Shared Memory Maschine
- häufig hybride Ansätze
  z.B. OpenMP + MPI

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Broadcast

Trivial?

```
static int cast_value = 0;
void broadcast(int value) {
  cast_value = value;
}

int receive_broadcast() {
  while (!cast_value)
  { /* wait */ }
  return cast_value;
}
```


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Broadcast

Trivial?

Atomics

```
#include <atomic>
static std::atomic_int cast_value = 0;

void broadcast(int value) {
  cast_value.store(value);
}

int receive_broadcast() {
  int temp = 0;
  while (!temp)
    temp = cast_value.load();
  return cast_value.load();
}
```


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Broadcast

Trivial?

Atomics

Sync

```
#include <atomic>
static std::atomic_int cast_value = 0;

void broadcast(int value) {
  cast_value.store(value);
  barrier();
}
int receive_broadcast() {
  int temp = 0;
  while (!temp)
    temp = cast_value.load();
  barrier();
  return temp;
}
```


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Reduktion

```
#include <atomic>
static std::atomic_int red_value = 0;

int sum_reduce(int value) {
  red_value.fetch_add(value);
  barrier();
  return red_value.load(); }
```

Verwende vorhandene atomare Reduktionen.

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Reduktion

```
#include <atomic>
static std::atomic_int red_value = 0;

int sum_reduce(int value) {
  red_value.fetch_add(value);
  barrier();
  return red_value.load(); }
```

Verwende vorhandene atomare Reduktionen.

Alternativ: Baum Reduktion mit rein lokaler Synchronisation.

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

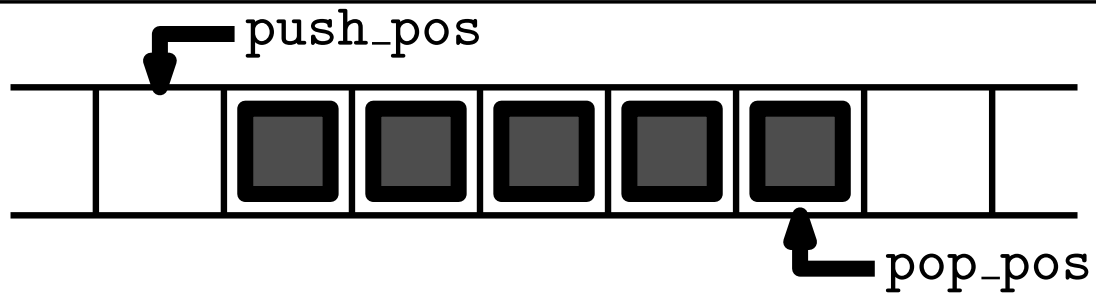
Reduktion

```
#include <atomic>
static std::atomic_int red_value = 0;

int sum_reduce(int value) {
  red_value.fetch_add(value);
  barrier();
  return red_value.load(); }
int sum_reduce(size_t valn, int* vals)
{
  int temp = 0;
  for(size_t i = 0; i < nval; ++i)
    temp += vals[i];
  red_value.fetch_add(temp);
  barrier();
  return red_value.load(); }
```


Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Task Queue

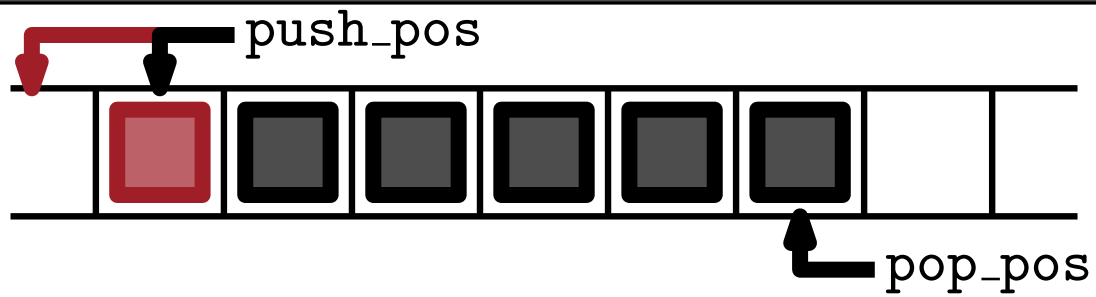
```
#include <atomic>
class circular_task_buffer {
private:
  using Task = void*;
  const static void* empty = nullptr;
  const static size_t cap = 1000;

  std::atomic<Task> task_array[cap];
  std::atomic<size_t> push_pos;
  std::atomic<size_t> pop_pos;
...
```



Institut für Theoretische Informatik
Lehrstuhl Algorithmik

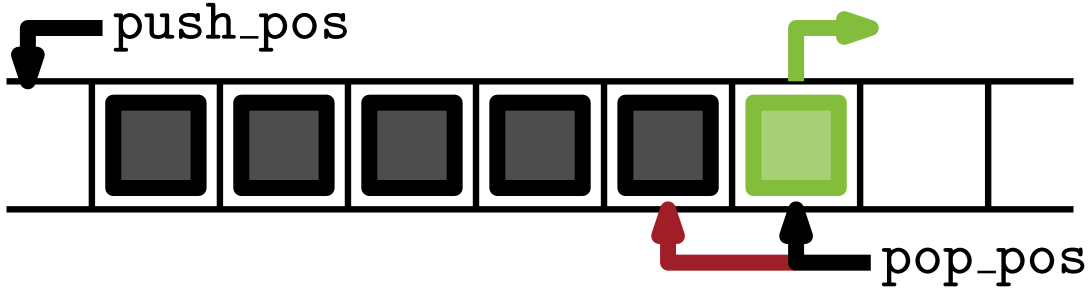
Task Queue

```
...
void push(Task t){
  size_t p = push_pos.fetch_add(1) % cap;
  bool succ = false;
  while (!succ) {
    Task expected = empty;
    bool succ = task_array[p]
    .compare_exchange_weak(expected, t);
  }
}
...
```



Task Queue

```
...
Task pop() {
  size_t p = pop_pos.fetch_add(1) % cap;
  Task temp = empty;
  while (temp == empty)
    temp = task_array[p].load();
  task_array[p]
  .compare_and_swap_strong(temp, empty);
  return temp;
}
...
```



Task Queue

Karlsruher Institut für Technologie

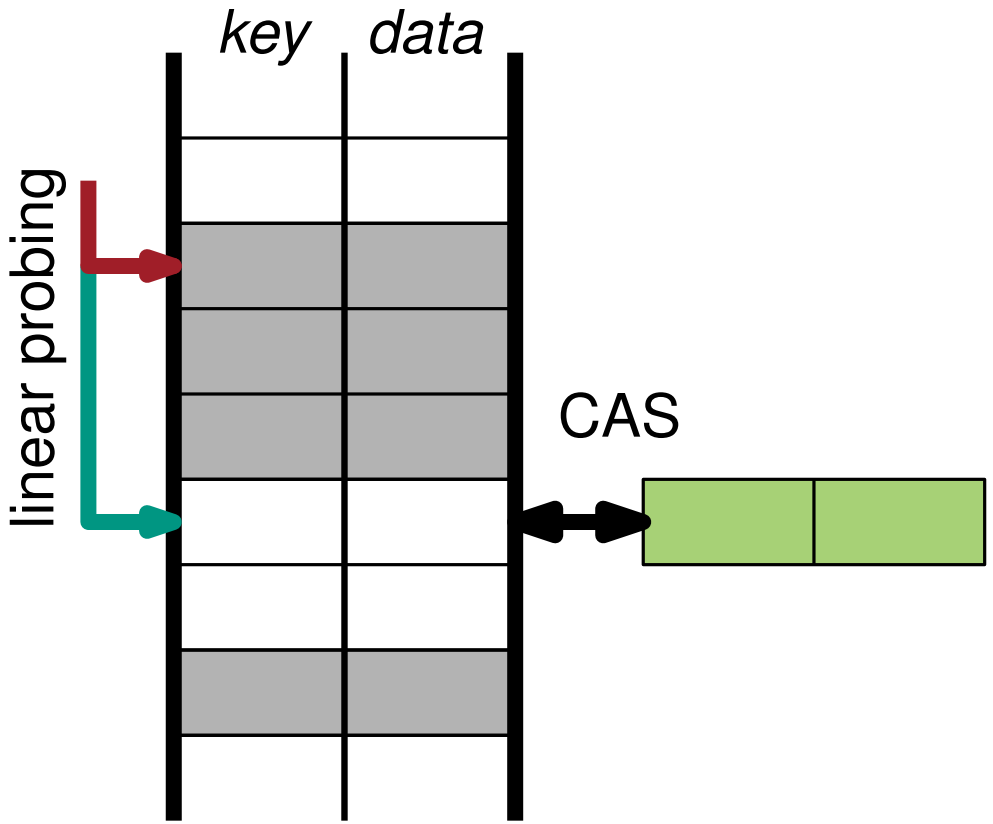
Contra

- Contention auf Countern
- Nur atomare Datentypen
- Beschränkte Größe
- Nicht Wait Free

Pro

- ”Striktes” FiFo
- Sehr schnell wenn nicht Bottleneck
- Simpel

Concurrent Hash Table



- linear probing
- compare and swap für `insert(·)` und `update(·)`
  - benötigt 2-Wort CAS
- `remove(·)` mit Dummies (nicht wiederverwendbar wegen ABA Problem)
- hashing verhindert Contention
- Memory Bus wird ausgereizt

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

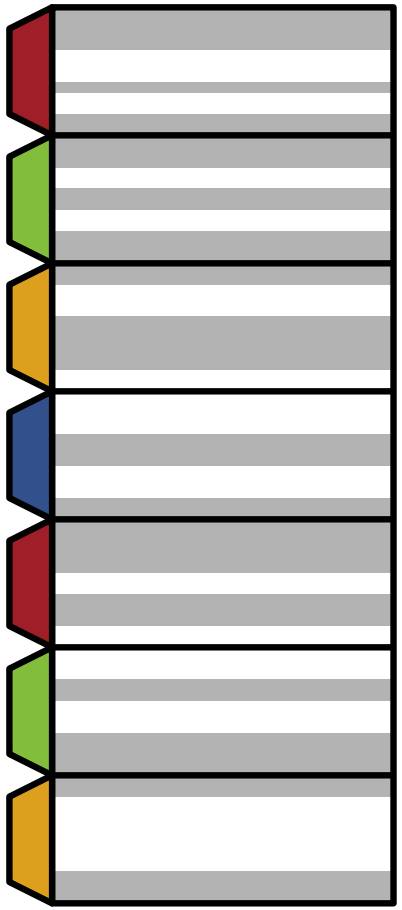
Vergrößern der Hash Tabelle

Karlsruher Institut für Technologie

Zentrale Ziele der Migration

- Asynchron
- wenig Komunikation/ Interaktion
- keine/wenig atomare Operationen
- Cache Effizienz

Vergrößern der Hash Tabelle

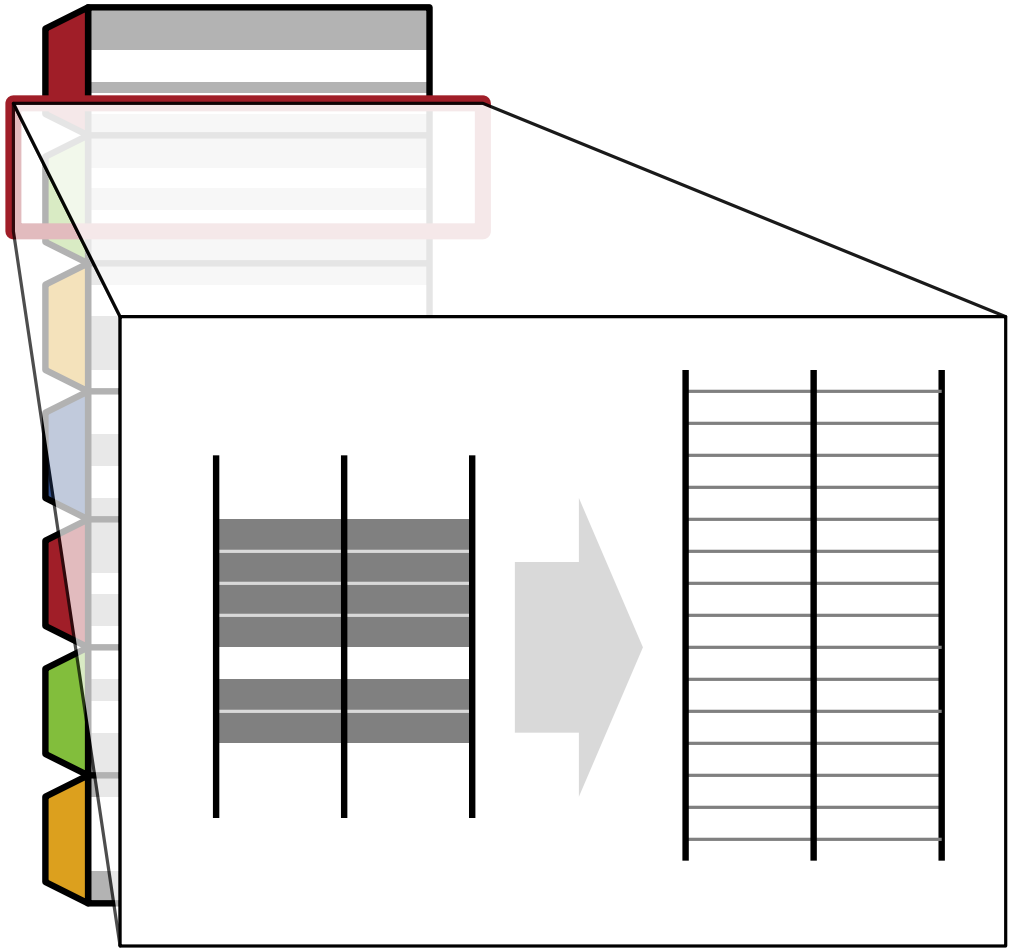


Verteilung der Blöcke
mit einem Atomic
und `fetch_add(·)`

- **Aufteilen in Blöcke**
- Beobachtung über Cluster
- Implizite Cluster-Blöcke
- Block Migration

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

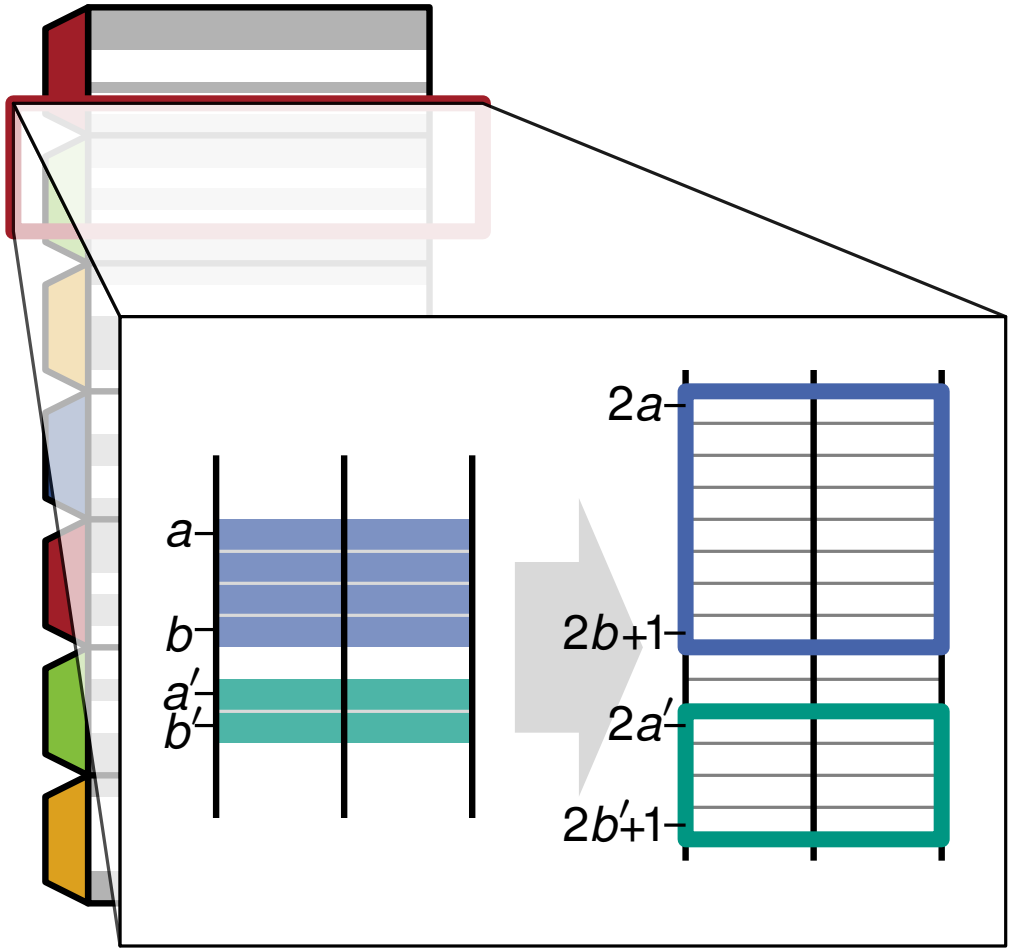
Vergrößern der Hash Tabelle



- Aufteilen in Blöcke
- **Beobachtung über Cluster**
- Implizite Cluster-Blöcke
- Block Migration

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

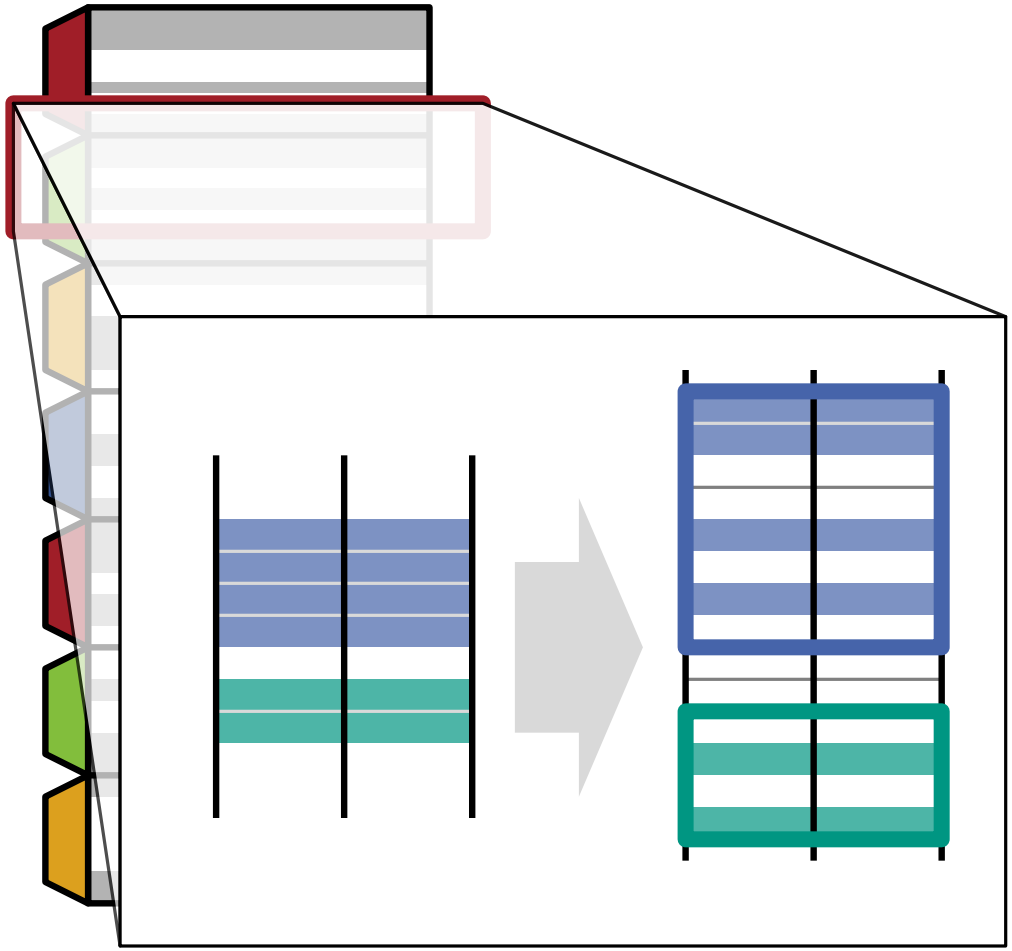
Vergrößern der Hash Tabelle



- Aufteilen in Blöcke
- **Beobachtung über Cluster**
- Implizite Cluster-Blöcke
- Block Migration

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

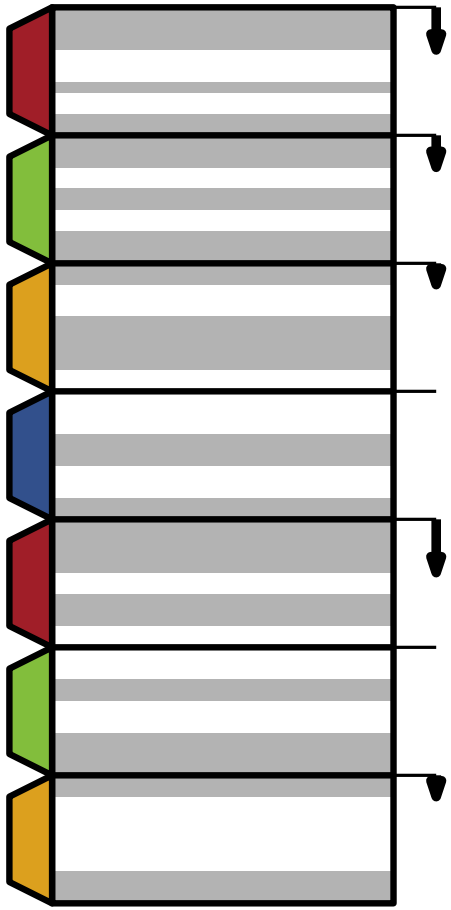
Vergrößern der Hash Tabelle



- Aufteilen in Blöcke
- **Beobachtung über Cluster**
- Implizite Cluster-Blöcke
- Block Migration

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

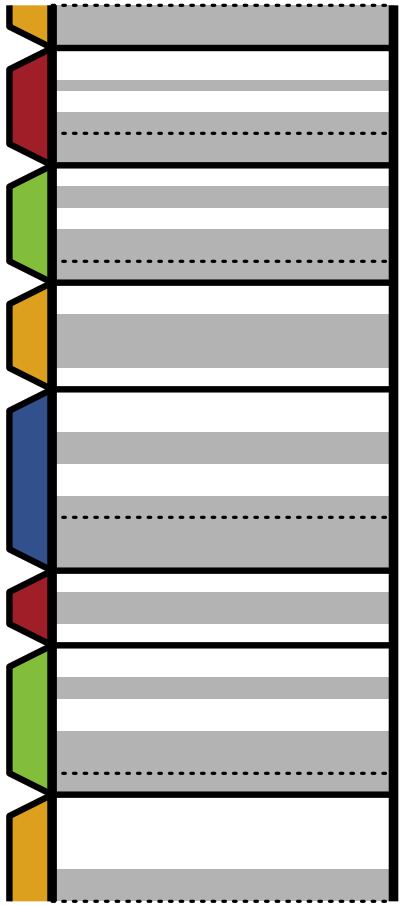
Vergrößern der Hash Tabelle



- Aufteilen in Blöcke
- Beobachtung über Cluster
- **Implizite Cluster-Blöcke**
- Block Migration

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

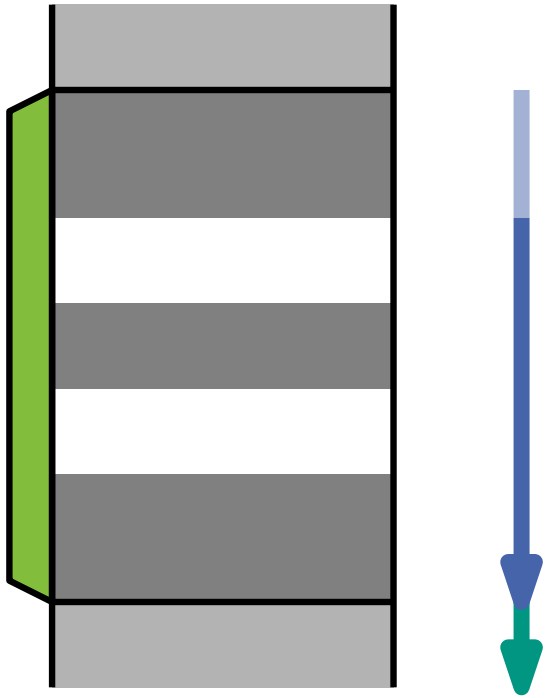
Vergrößern der Hash Tabelle



- Aufteilen in Blöcke
- Beobachtung über Cluster
- **Implizite Cluster-Blöcke**
- Block Migration

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Vergrößern der Hash Tabelle



- Aufteilen in Blöcke
- Beobachtung über Cluster
- Implizite Cluster-Blöcke
- **Block Migration**

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

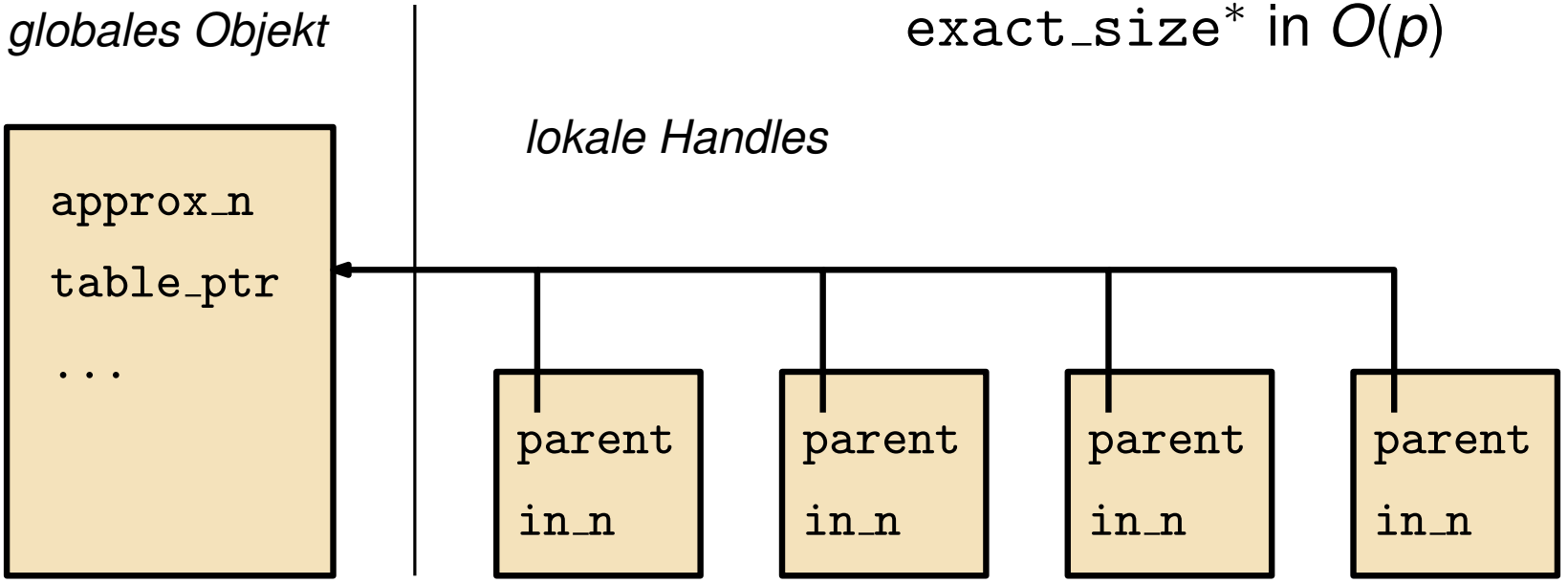
Approximate Element Count

Problem: Elemente zählen erzeugt Contention auf einem Atomic

- p einzelne Counter
- ein globaler Counter + lazy Updates
- Counter werden in Handles gespeichert

`approx_size` in O(1)

`exact_size`* in $O(p)$



* `exact_size` race conditions mit `inserts`

Deallocation Problem

Problem: deallokieren der Tabelle nach Migration

- Thread kann angehalten werden
  $\Rightarrow$ während Operation auf der alten Tabelle
- Migration ist Asynchron

Lösung: Shared Pointer

- keine Atomics für `std::shared_ptr`
- regelmäsiges Kopieren ist langsam
- Cachen im Handle

Institut für Theoretische Informatik
Lehrstuhl Algorithmik

Good Practices for Shared Memory

- Verstehen versteckter Komplexität
  `shared_ptr`, `fetch_add` ...
- Verhindere Thread Rescheduling
  Core-Pinning oder Thread Affinitäten
- Verhindere False Sharing
  `alignas`
- Vermeide Allokationen
  Allgemein Betriebssystem Interaktion
- Beachte ABA Probleme
- Häufiger Bottleneck Datenbandbreite