Designing Fast and Efficient List-like Data Structures

YANNIC BONENBERGER

List-like data structures

- std::vector
- std::list
- std::deque

std::vector

- C++ version of the array-list data structure
- Backed by a C-style array
- Automatically allocates a new backing array when inserting into a "full" std::vector

std::vector

```
template <class T>
class vector {
 public:
  // constructor, accessors, ...

 private:
  size_t size_;
  size_t capacity_;
  T* data_;
};
```

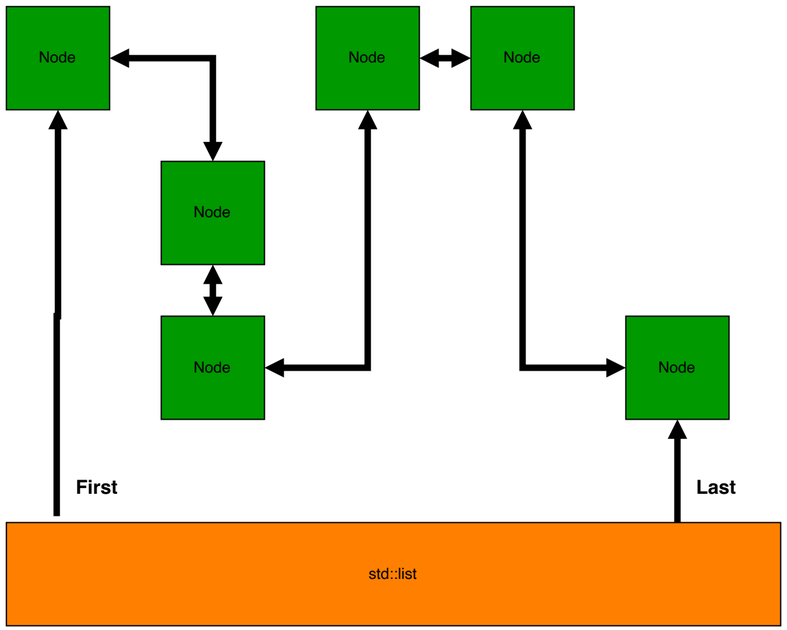
std::vector

- Compact layout - elements stored in contiguous memory
- Inserting or removing elements at the end is very cheap (O(1))
- Accessing elements in random order is very cheap (O(1))
- Inserting or removing elements at positions other than the end is expensive (O(n))

std::list

- C++ version of the linked-list data structure
- Elements stored in nodes referencing the next node

std::list



std::list

- Inserting or removing elements in the front or at the end is very cheap (O(1))
- Accessing elements in random order is expensive (O(n))

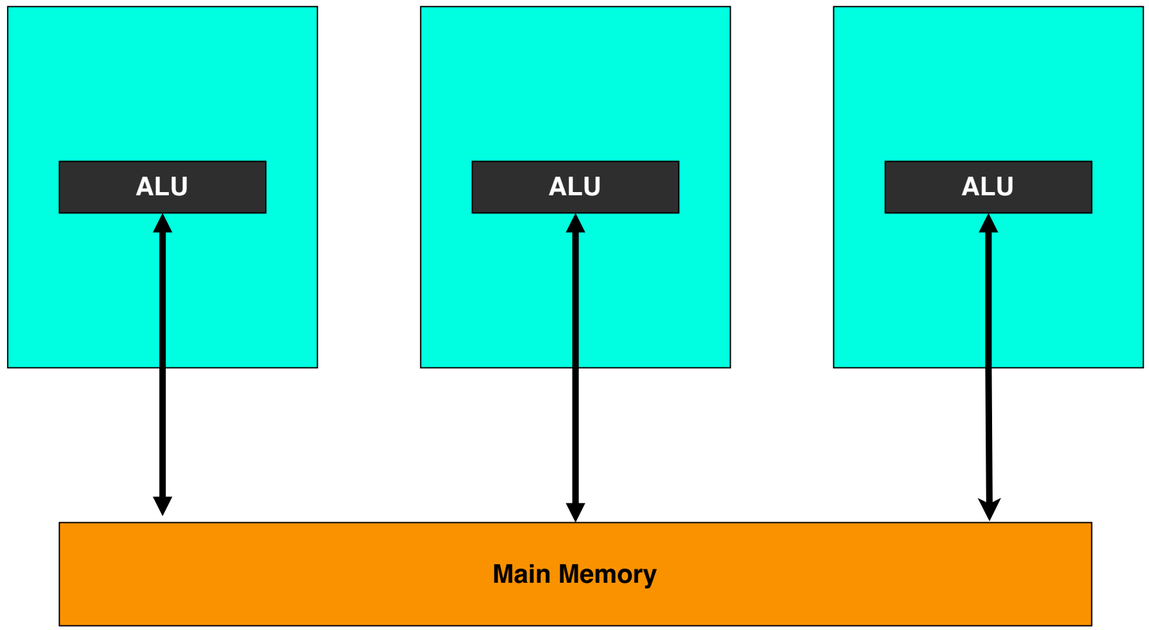
std::deque

- Conceptually a table of vectors

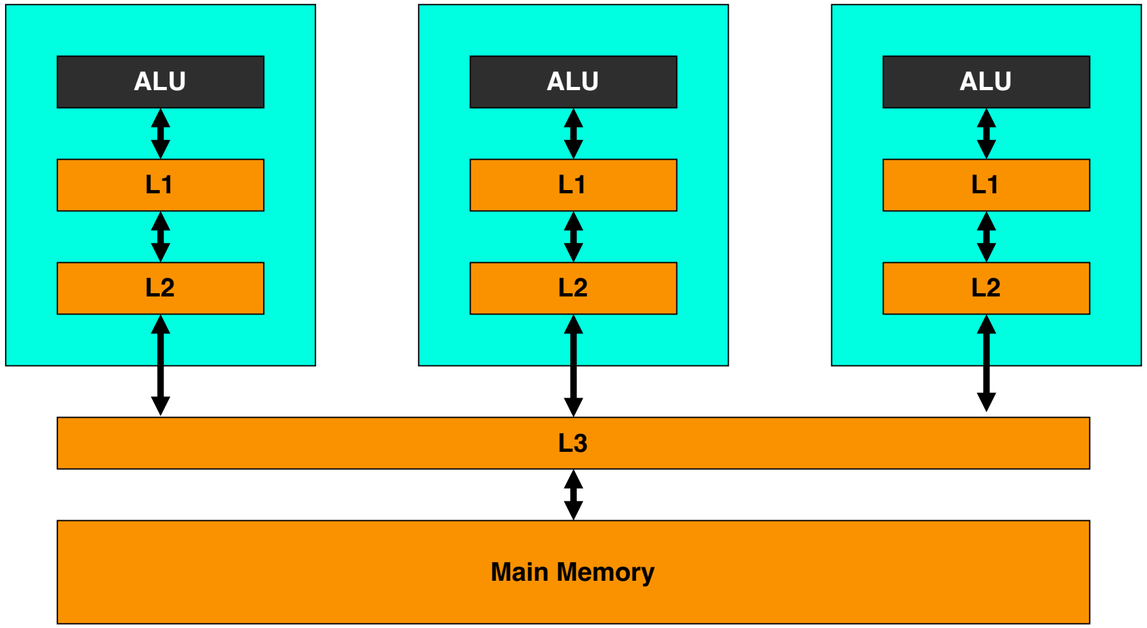
std::deque

- Stores elements in chunks of contiguous memory
- Inserting or removing elements in the front or at the end very cheap (O(1))
- Accessing elements in random order is cheap (O(1))
- Inserting or removing elements at positions other than the front or end is expensive (O(n))
- Requires additional indirection for every access

Modern CPU architectures



Modern CPU architectures

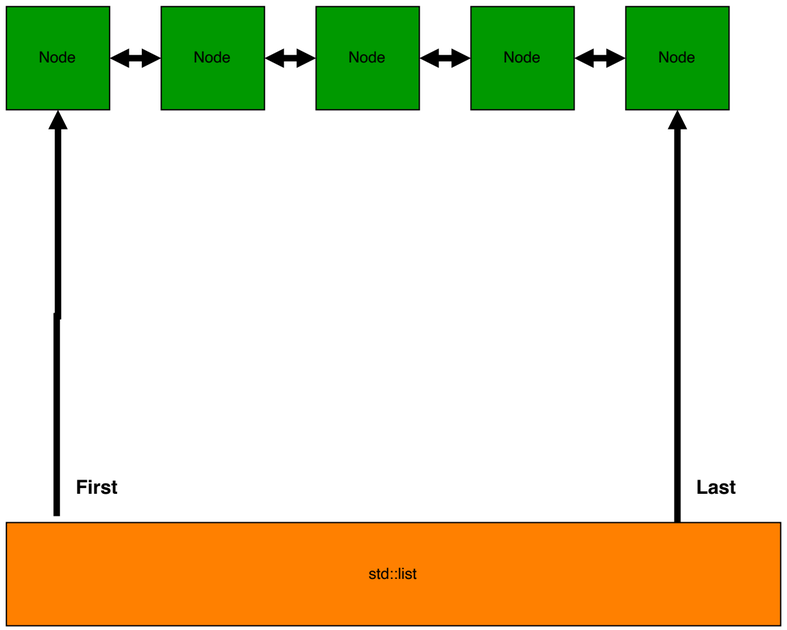


Modern CPU architectures

std::list with custom allocator

- Idea: minimize cache misses by grouping consecutive nodes into contiguous memory

std::list with custom allocator



std::list with custom allocator

- Packing sequences of nodes into a contiguous chunk of memory provides better cache locality
- However: having a wrapper for each element containing a pointer to the previous and next node is high overhead

FixedStack

- Stack implementation with a (compile-time) fixed upper bound for the number of elements it can hold
- Completely avoids heap allocations for small amount of small objects

FixedStack

```
template <class T, size_t capacity>
class FixedStack {
 public:
  FixedStack& Push(const T& element);
  ElementContainer Pop();

 private:
  Page<T, capacity> data_;
};
```

Memory Pages

```
template <class T, size_t page_size>
class Page {
 public:
  T& Front();
  T& Back();
  T& At(size_t position);

  Slot<T>::ElementContainer RemoveFront();
  Slot<T>::ElementContainer RemoveBack();

 private:
  size_t front_slot_;
  size_t back_slot_;
  Slot<T> slots_[page_size];
};
```

Memory Slots

```
template <class T>
using Slot = SlotImpl<
    T,
    (sizeof(T) <= (ABSL_CACHELINE_SIZE / 8)) ||
        ((sizeof(T) <= (ABSL_CACHELINE_SIZE / 2)) &&
        ((ABSL_CACHELINE_SIZE % sizeof(T)) == 0))>;
```

Memory Slots

```
template <class T>
class alignas(T) SlotImpl<T, true> {
 public:
  // constructor, accessors, ...

 private:
  alignas(T) char storage_[sizeof(T)];
};
```

Memory Slots

```
template <class T>
class alignas(T) SlotImpl<T, false> {
 public:
  // constructor, accessors, ...

 private:
  T* storage_;
};
```

Queue

```
template <class T, size_t page_size = 128>
class Queue {
 public:
  Queue& Push(const T& element);
  ElementContainer Pop();

 private:
  PageList<T, page_size> pages_;
  PageList* first_;
  PageList* last_;
};
```

Real-World use-case of Queue

```
class TaskQueue {
 public:
  Add(OnceCallback<void()> task) {
    task_queue_.Push(std::move(task));
  }

 private:
  void Run() {
  while (true) {
    std::move(task_queue_.Pop()).Run();
  }
  }

  ThreadSafeQueue<OnceCallback<void()>> task_queue_;
};
```

Outlook and Future Work

Early results show that TaskQueue can pass tasks between threads with 50% * less overhead than std::deque and avoids most of the syncronization needed for std::deque.

* Measured on a MacBook Pro, Apple M1 Max with 64GiB of memory

Outlook and Future Work

However: Especially the dynamic-sized and thread-safe containers including TaskQueue are still experimental.

Outlook and Future Work

Publishing on GitHub still in progress. Planned for later this year.

Questions?

Thank You!