Datastructures and Algorithms

Greedy Algorithms, Huffman Coding (Trees), Parallel Programming

Adel Gavranović — ETH Zürich — 2025

Overview

Learning Objectives
Huffman Coding
Greedy Choice
In-Class-Exercise (practical)
Parallel Programming
Old Exam Questions
Hints for current tasks

n.ethz.ch/~agavranovic

Material

Webpage

Mail

1. Learning Objectives

Objectives

- ☐ Be able to build a Huffman Coding Tree using the algorithm outlined in the session
- ☐ Be able to reason about simple multithreaded programs
- ☐ Understand the different aproaches to modelling performance of parallel programs (Amdahl, Gustafson)

2. Summary

Getting on the same page

- What did you see in the lectures up to now?

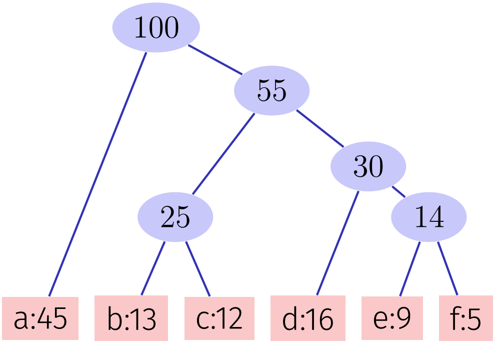
3. Huffman Coding

Huffman's Idea

From the Lecture

Tree construction bottom up

- Start with the set C of code words
- Replace iteratively the two nodes with _____ frequency by a _____. Replace iteratively the two nodes with smallest frequency by a new parent node.



Algorithm Huffman(C)

From the Lecture

Input: code words $c \in C$
Output: Root of an optimal code tree

$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ **to** $n - 1$ **do**
    allocate a new node z
    z.left $\leftarrow$ ExtractMin(Q)   // extract word with minimal frequency.
    z.right $\leftarrow$ ExtractMin(Q)
    z.freq $\leftarrow$ z.left.freq $+$ z.right.freq
    Insert(Q, z)
return ExtractMin(Q)

4. Greedy Choice

Recap: Greedy Choice

Question:

What properties must an optimization problem with a recursive solution have in order to be solvable with a greedy algorithm?

Also, give an example and a counterexample.

Recap: Greedy Choice

A problem with a recursive solution can be solved with a **greedy algorithm** if it has the following properties:

- The problem has **optimal substructure**: the solution of a problem can be constructed with a combination of solutions of sub-problems.
- The problem has the **greedy choice property**: The solution to a problem can be constructed, by using a local property that does not depend on the solution of the sub-problems.

Examples: Fractional knapsack problem, Huffman coding
Counterexamples: Knapsack problem, optimal binary search tree.

5. In-Class-Exercise (practical)

Complement the DP implementation to compute an optimal search tree. $\longrightarrow$ CodeExpert

6. Parallel Programming

Parallel Programming

Parallel Programming = perform multiple computations in parallel

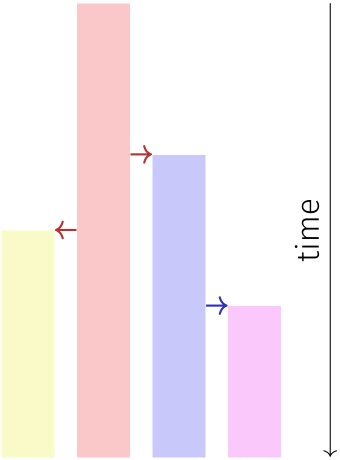
Some terminology

- **Tasks**
  are computations that need to be done. Independent computations can be done in parallel.
- **Threads**
  are parallel executions, that execute tasks.
- **Shared resources**
  anything that is needed to perform tasks, but must be shared because there isn't a resource per task. (Not the focus for this week)

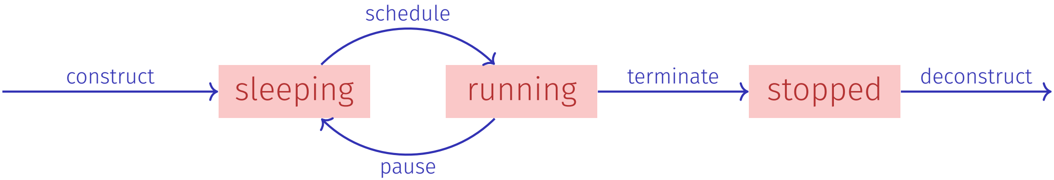
Forking Threads

Forking a thread means starting a new, concurrent computation

- Main thread forks a new thread
- Forking is done by creating a new thread object `std::thread(func, args...)`
- Main thread is the parent of its child thread
- Each thread can fork further threads



Thread Lifecycle (simplified)



The operating system's **scheduler** decides

- which thread can execute next (schedule)
- on which core to execute
- when to pause/sleep again

Switching threads on the processor, which puts the current thread to sleep and wakes up another one, is called **context switching**.

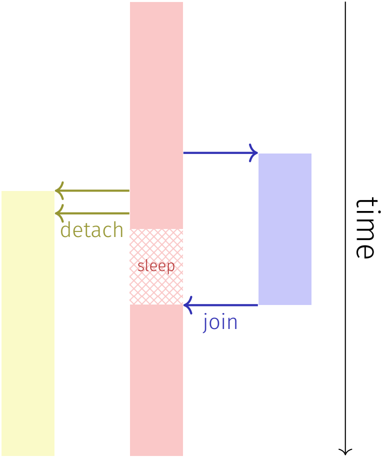
Join and Detach

`other_thread.join()` means waiting until `other_thread` has finished execution.

- The joining thread will sleep until `other_thread` terminated (if it already did, no sleeping is necessary)

`other_thread.detach()` indicates that no thread will wait for `other_thread` to finish execution

- Useful for nonterminating processes (e.g. servers), and reactive systems (e.g. GUIs)
- Terminates alongside the main thread at the latest (`int main()`)

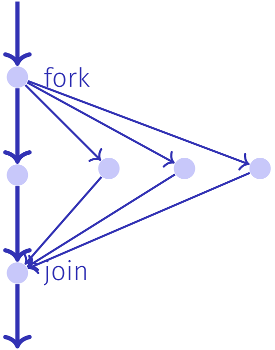


C++ Threads

From the Lecture

```
void hello(unsigned id) {
  std::cout << "hello from " << id << "\n";
}

int main() {
  std::vector<std::thread> tv(3);
  unsigned id = 0;
  for (auto& t : tv)
    t = std::thread(hello, ++id);
  std::cout << "hello from main\n";
  for (auto& t : tv)
    t.join();
}
```



Nondeterministic Execution!

From the Lecture

One execution:

hello from main
hello from 1
hello from 2
hello from 3

Other execution:

hello from 2
hello from main
hello from 1
hello from 3

Other execution:

hello from main
hello from 1
hello from hello from 2
3

Technical Details I

- Forking a function that takes a reference requires `std::ref` upon thread construction

```
void calc(std::vector<int>& very_long_vector) {
  // doing funky stuff with very_long_vector
}

// main
std::vector<int> v(1000000000);

std::thread t1(calc, std::ref(v)); // Compiler error w/o std::ref
std::thread t2([&v]{ calc(v)}; }); // Alternative
```

Technical Details II

- Threads cannot be copied

```
// --- Error ---
std::thread t1(hello);
std::thread t2;
t2 = t1; // Compiler error
t1.join();

// --- OK ---
std::thread t1(hello);
std::thread t2;
t2 = std::move(t1); // OK
t2.join();
```

- Also relevant if threads are to be stored in containers

Technical Details

Also see the corresponding “Exercise Class Example” on Code Expert with further technical details

Quiz

```
void print(char c); // Output character c

void A(char value) {
  if (value != 'D') {
    std::thread t(A, value + 1);
    print(value);
    t.join();
  }
}

int main() {
  std::thread t(A,'A');
  t.join();
}
```

possible output(s)?

Parallel Performance

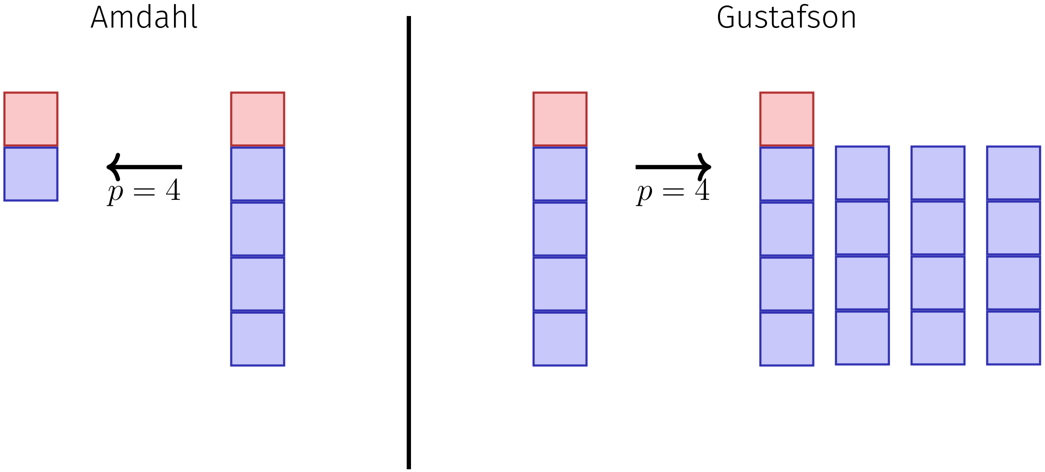
Given

- fixed amount of computing work W (number computing steps)
- Sequential execution time T_1
- Parallel execution time on p CPUs T_p

	runtime	speedup	efficiency
perfection (linear)	$T_p = T_1/p$	$S_p = p$	$E_p = 1$
loss (sublinear)	$T_p > T_1/p$	$S_p < p$	$E_p < 1$
sorcery (superlinear)	$T_p < T_1/p$	$S_p > p$	$E_p > 1$

Amdahl vs. Gustafson

From the Lecture



Amdahl vs. Gustafson, or why do we care?

Amdahl	**Gustafson**
pessimist	optimist
strong scaling	weak scaling

$\Rightarrow$ need to develop methods with smallest sequential protion possible.

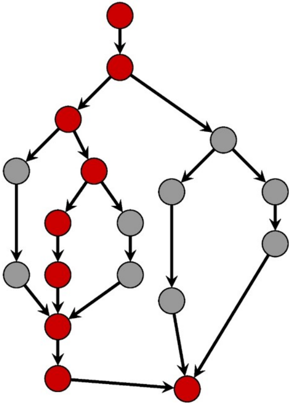
Performance Model

From the Lecture

- T_1: **work**: time for executing total work on one processor
- T_p: Execution time on p processors
- T_∞: **span**: critical path, execution time on ∞ processors. Longest path from root to sink.
- T_1/T_∞: **Parallelism:** wider is better
- Lower bounds:

$$T_p \geq T_1/p \quad \text{Work law}$$
$$T_p \geq T_\infty \quad \text{Span law}$$

Task graph:



Greedy Scheduler

From the Lecture

Greedy scheduler: at each time it schedules as many available tasks as possible.

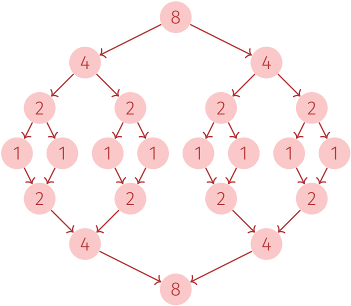
Theorem 1

On an ideal parallel computer with p processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time

$$T_p \leq T_1/p + T_\infty$$

Quiz: Scheduling

The following figure shows a task-graph of some algorithm. The number in each of the nodes denotes the execution time per task step.



$T_\infty = ?$ $T_1 = ?$ $T_4 \leq ?$ $T_8 \leq ?$

7. Old Exam Questions

Old Exam Questions

Question

The analysis of a program has shown a speed-up of 2 when running on 9 processor cores. What is the serial fraction according to Gustafson's law?

Answer

Using Gustafson's law formula $S_p = p - \lambda \cdot (p - 1)$, we substitute the given values $S_p = 2$ and $p = 9$ to get $2 = 9 - \lambda \cdot 8$. Rearranging gives $7 = \lambda \cdot 8$. Solving for λ (the serial fraction), we find $\lambda = \frac{7}{8} = 0.875$.

Old Exam Questions

Question

You make a measurement of your program using a very large number of processor cores. The measurements suggest that the speed-up (using arbitrarily many processor cores) is bounded from above by $S_\infty = 2.5$. What is the best possible upper bound on the speed-up using 6 cores, assuming that Amdahl's law holds for your problem?

Answer

Using Amdahl's law formula $S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$ and $S_\infty = \frac{1}{\lambda} = \frac{5}{2}$, we find $\lambda = \frac{2}{5}$.

Substituting λ and $p = 6$ into Amdahl's law gives $S_6 \leq \frac{1}{\frac{0.4}{1} + \frac{0.6}{6}} = 2$.

8. Hints for current tasks

Huffman Coding

Huffman: Frequencies

Use `std::unordered_map` **(**`#include <unordered_map>`**)**

```
std::unordered_map<char, int> frequencies;
// ...
++frequencies['a'];
++frequencies['x'];
++frequencies['a'];

// A map is a container of key-value pairs (std::pair).
// Output all entries:
for (auto x:observations){
  std::cout << "observations of " << x.first << ":" << x.second << '\n';
}
```

Huffman: Min Heap

Use `std::priority_queue` **(**`#include <queue>`**)**

```
struct MyClass {
  int x;
  MyClass(int X): x{X} {}
};

struct compare {
  bool operator() (const MyClass& a, const MyClass& b) const {
    return a.x < b.x;
  }
};

std::priority_queue<MyClass, std::vector<MyClass>, compare> q;
q.push(MyClass(10));
```

Huffman: Shared Pointers [optional]

Shared Pointers `std::shared_ptr` (`#include <memory>`)

```
struct SNode {
  int value;
  std::shared_ptr<SNode> left;
  std::shared_ptr<SNode> right;
  SNode(int v): value{v}, left{nullptr}, right{nullptr} {}
};

// A graph in which node 7 is shared:   //     0
SNode* root = new SNode(0);             //    / \
root->left = new SNode(1);              //   1   2
root->right = new SNode(2);             //      / \
root->right->left = new SNode(7);       //      \ /
root->right->right = root->right->left; //       7

root->left = nullptr; // Node 1 can and should be deallocated (deleted) now
root->right->left = nullptr; // Node 7 must not yet be deallocated
root->right->right = nullptr; // Node 7 can and should be deallocated now
```

Automated memory management, see Code Expert example

Huffman: Tree Nodes

```
using SharedNode = std::shared_ptr<Node>;

struct Node {
  char value;
  int frequency;
  SharedNode left;
  SharedNode right;

  // constructor for leafs
  Node(char v, int f):
    value{v}, frequency{f}, left{nullptr}, right{nullptr}
  {}

  // constructor for inner nodes
  Node(SharedNode l, SharedNode r):
    value{0}, frequency{l->frequency + r->frequency}, left{l}, right{r}
  {}
};
```

9. Outro

General Questions?

See you next time!

Have a nice week!