Seminar Report

Rusty Parallel Traveling Salesman Problem Solver

Lars Quentin, Johann Carl Meyer

MatrNr: 21774184, 21969570

Supervisor: Dr. Artur Wachtel

Georg-August-Universität Göttingen
Institute of Computer Science

October 15, 2023

Abstract

The Travelling Salesman Problem (TSP) is one of the most studied problems in computer science and one of the most intuitive and well-known NP-complete problems. This report presents walky, a new Rust-based TSP solver. Walky supports exact solving using several highly optimized algorithms with support for sequential, multithreaded as well as distributed, MPI execution. Additionally, it supports two different approximation algorithms: The simple, easy-to-implement Nearest Neighbour approximation and the more sophisticated Christofides algorithm. Walky is fully production-ready, fully tested, and supports the most-used TSPLIB-XML format.

The Benchmarks show that, for exact solving, walky successfully uses pruning to immensely improve performance and increase the viability of exact solving. The nearest neighbour algorithm scales well with parallelism due to its minimal inter-worker communication requirements. The 1-tree lower bound also greatly benefits from parallelism. Christofides algorithm in its randomized implementation is a very quick approximation to the TSP, it can be made more reliable by utilizing parallelism. For the MST computation, the graphs tested in this setting were too small to benefit from parallelism, though the benchmarks indicated that for larger graphs a parallel implementation of Prim's algorithm would outperform its sequential counterpart.

Overall, walky shows that Rust is a valid choice for developing highly distributed High-Performance Computing (HPC) applications.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☑ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Abbreviations

ABI	Application Binary Interface
CI	Continuous Integration
CLI	Command Line Interface
FFI	Foreign Function Interface
HPC	High-Performance Computing
NN	Nearest Neighbour
NP	Nondeterministic Polynomial Time
LLVM	Low Level Virtual Machine
LTO	Link Time Optimization
MPI	Message Passing Interface
MSRV	Minimum Supported Rust Version
MST	Minimum Spanning Tree
RAII	Resource Acquisition Is Initialization
SCC	Scientific Compute Cluster
TSP	Travelling Salesman Problem
UX	User Experience

1 Introduction

1.1 Motivation

Before explaining the general motivation, a small introduction to the TSP is required.

1.1.1 Travelling Salesman Problem Definition

The TSP is easily described in colloquial language.

North Sea
Baltic Sea
DEN.
Kiel
Rostock
Neuendorf bei Wilster
Lübeck
Hamburg
Bremerhaven
Emden
Bremen
BERLIN
POL.
Hannover
NETH.
Magdeburg
Duisburg
Essen
Leipzig
Düsseldorf
Kassel
Köln
Bonn
Dresden
BEL.
Frankfurt am Main
Wiesbaden
CZECH REPUBLIC
LUX.
Mannheim
Nürnberg
Karlsruhe
Stuttgart
FRANCE
München
Danube
AUSTRIA
Zugspitze
SWITZ.
LIECH.
Rhine
Elbe
0 50 100 km
0 50 100 mi

user "Kapitän Nemo" https://commons.wikimedia.org/w/index.php?curid=5584283

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" [1]

More formally the TSP is defined as follows.

- **Input**: A weighted (only non-negative weights), undirected, and complete graph.
- **Output**: A tour (cycle that visits every vertex) on the input graph, that uses each edge at most one time.
- **The optimization problem**: Find a valid output that has minimal (cumulative) edge weight.

For a more visual introduction to the problem, see the video essay "The Traveling Salesman Problem: When Good Enough Beats Perfect" by Reducible [2].

1.1.2 Why is TSP interesting?

The TSP is interesting for us, because of a few reasons. First, the problem is well studied, thus good literature on the problem is easily found. Then, the TSP is known to be NP-hard, meaning that there can be found relatively small input data, for which the solution takes large amounts of computing power, thus suggesting that HPC can be leveraged to speed up a computation. Moreover, the problem is intuitive to understand and has practical applications in e.g. genome analysis, satellite route planning, or fiber optical network design [3], using the Concorde TSP Solver software.

1.1.3 The Implementation of this Project

As a part of this project, a TSP solver was implemented and made publicly available on GitHub[1], the software has been built using the Rust programming language. To integrate into the Rust ecosystem, the solver has been published to crates.io as well[2]. All the code is published and licensed under the permissive MIT open-source license to encourage further third-party development.

1.2 Goals and Contributions

The following goals and contributions are defined for this project.

1. Develop a CLI tool compatible with current state-of-the-art research
2. Performance and Efficiency
    - Create a blazingly fast software package
    - Provide a 100% pure Rust alternative to classical solvers
    - Support both shared and distributed memory parallelization
    - Achieve full documentation coverage
    - Achieve high unit test coverage
3. Exact Solving
    - Implement a simple, exact solver for the TSP
    - Offer several optimized versions
    - Create a shared memory parallelized version
    - Develop a distributed memory, parallelized solver based on MPI
4. Approximation Tactics
    - Include a trivial, easy-to-parallelize tactic and
    - A sophisticated, state-of-the-art tactic
    - For both:
        - Provide a shared memory parallelized solver
        - Provide a distributed memory, MPI-based parallelized solver
5. Lower Bound Calculation for TSP
    - Provide a sequential implementation
    - Develop a parallelized implementation using MPI

[1]`https://github.com/lquenti/walky`
[2]`https://crates.io/crates/walky/`

2 Methodology

2.1 Minimum Spanning Tree

For the further report, it is assumed, that the reader is familiar with the concept of an MST. For the unfamiliar reader, a look at [4, pp. 585ff.] is suggested.

For this project, the MSTs are being calculated using Prim's Algorithm.

2.1.1 Prim's Algorithm

Prim's Algorithm is a well-known algorithm used for finding MSTs in graphs. The following Definition of the algorithm is adapted from [4, p. 596].

```
// G: a graph
// G.V: List of all vertices in G
// G.Adj: G.Adj[u] is a List of all vertices that are adjacent to u
// w: w(u,v) == weight of the edge between u and v
// r: root / starting vertex for the algorithm
MST-PRIM(G, w, r)
  for each vertex u in G.V
    u.key = ∞
    u.pi = NIL
  r.key = 0
  Q = PriorityQueue::empty()
  for each vertex u in G.V
    Q.insert(u)
  while not Q.is_empty()
    u = Q.extract_min() // add u to the tree
    for each vertex v in G.Adj[u] // update keys of u's non-tree neighbors
      if v in Q and w(u, v) < v.key
        v.pi = u
        v.key = w(u,v)
        Q.update_key(v, w(u,v))
```

The implementation of the PriorityQueue can be done with different data structures. A naïve way is to use vectors with linear search, where a single `extract_min` Operation takes linear time. There are more sophisticated implementations using binary or Fibonacci heaps, that improve the asymptotic time demand of Prim's algorithm. However, when testing the different implementations on TSPLIB instances [5], it becomes clear that the graph sizes are too small to benefit from asymptotic improvements, and the simple linear search approach is the fastest. This is likely a consequence of the higher locality and simplicity of the code, giving rise to better compile-time optimizations, better branch prediction, and many other optimizations.

It is possible to parallelize linear search on a vector. However, due to the graphs for a TSP instance being relatively small in terms of nodes, there was very little benefit in using parallelization for the tested use cases. Even more so, the overhead created by the parallelization framework oftentimes outweighed the benefits of parallel computation. Nevertheless, a parallel implementation for shared memory is provided, relying on the rayon `ParallelIterator` implementation to handle the exact details. Although an implementation using MPI is possible, the results from the rayon benchmarks suggest that there is little benefit in implementing it for this use case, as can be seen in section 4.7.

2.2 Exact and Approximate Solving

Since it is known to be NP[3] complete, the TSP is very hard to solve. While the naïve implementation is $\Theta(n!)$ time and $\Theta(n^2)$ space[4], even highly optimized, dynamic programming-based algorithms such as the well-known Held-Karp algorithm [6] still require $\Theta(2^n n^2)$ time with $\Theta(n2^n)$ space complexity for caching; n being the number of nodes. This implies that, for large graphs, the computation is too time-expensive. More importantly, a sub-exponential algorithm is very unlikely to be found in the future, as it would implicitly prove that $P = NP$.

With such a hard problem, which, as mentioned in the introduction, has many applications in real life, the need for sub-exponential approximation algorithms becomes obvious. Walky supports both exact solving and finding approximate solutions. In this chapter, the focus will first be on exact solving, starting with the naïve $\Theta(n!)$ algorithm which will then be optimized and parallelized using traditional performance optimization methods. After that, two different approximation algorithms will be covered. Lastly, two more algorithms for computing the lower bound will be presented.

2.3 Exact Solving

This section will focus on the exact solving. Starting with the most naïve, sequential implementation, this report will iteratively optimize the runtime using traditional performance engineering. Note that many of the ideas are inspired by the related TSP lecture from the MIT 6.172 course “Performance Engineering of Software Systems” [7].

2.3.1 Sequential Algorithms

This section will use pseudo-code whenever possible to simplify the algorithmic concepts explained.

Version 0: Naïve implementation: For a n-vertex graph, with vertices numbered $1, \dots, n$, a tour can be described as an n-dimensional vector where each number $[1, n]$ can occur only once. Thus, testing all possible tours for the shortest one is equivalent to trying out all permutations of the vector $(1, \dots, n)$. In pseudocode:

[3]Nondeterministic Polynomial Time (NP)
[4]Which is the minimum to hold the graph itself.

```
def tsp_solver(G):
  path = [1..n]
  least_cost = inf
  while (has_next_permutation(path)):
    current_cost = evaluate(path)
    if (current_cost < least_cost):
      least_cost = current_cost
    path = next_permutation(path)
  return least_cost
```

Figure 1: Naïve implementation of a TSP solver in pseudocode.

Since there are $(\prod_{i=1}^{n} i) = n!$ possible permutations, this algorithm has a time complexity of $\Theta(n!)$. For enumeration, Nayuiki's iterative next lexicographical permutation algorithm was used [8].

Version 1: Fix the first element: The TSP is defined as the shortest path traveling through all cities *and returning to the origin.* Since a tour is cyclic, the vectors $(1, 2, 3)$, $(3, 1, 2)$, and $(2, 3, 1)$ all represent the same underlying tour. Thus, one can reduce the time complexity to $\Theta((n-1)!)$ by fixing the first element.

Version 2 and 3: Prune if the subpath is non-optimal: Version 2 caches the subpaths, reducing the amount of additions required to compute the tour cost.

Starting with version 3, the next algorithm iterations will heavily rely on decision tree pruning. This is best explained by an example:

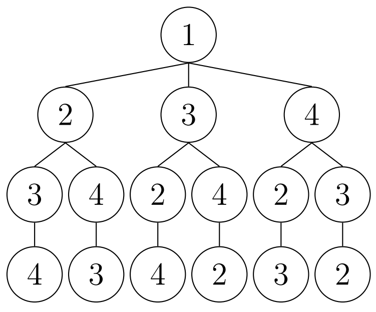
Example 2.1. *Pruning Let G be an n-dimensional, fully connected, undirected symmetric graph for which the TSP is being solved. W.l.o.g let 35 be the current optimal known cost.*

Let us further assume that the path $(4, 5, 6)$ has cost 36. Since edge costs can't be negative, this means that all paths with the prefix $(4, 5, 6)$ can't be better than the already known optimum, thus they can be pruned.

Hence, instead of iterating over all possible solutions as a whole, the partial sums of the subtours will be cached using **recursive enumeration**:

```
def rec_enum(xs, n):
  """Recursively enumerate xs"""
  if len(xs) == n: # Full path
    print(xs)
  for i in range(n):
    if i not in xs:
      rec_enum(xs + [i], n)
```

(a) Pseudocode of recursive enumeration



(b) All permutations of $(1, 2, 3, 4)$, with 1 fixed as the first element, visualized as a decision tree.

Figure 2: A pseudocode and visual representation of a recursive enumeration

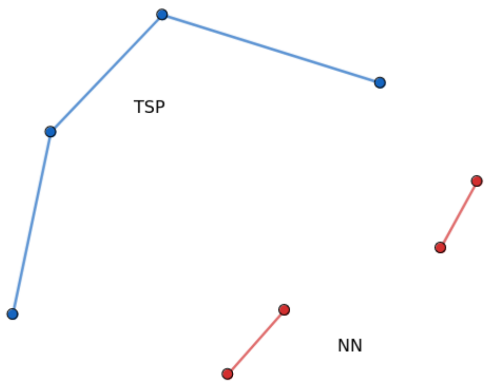
Therefore, version 3 uses recursive enumeration for the tour generation, computing the sub-tour cost on each level, and pruning whenever the partial sum is bigger than the previously known optimum by not recursing deeper.

Analogous to this approach, the next versions will use pruning extensively to achieve better performance by reducing the number of permutations to compute the cost for.

Version 4: Prune using NN Metric: In version 3, the subgraph was pruned iff it already costs more than the previously known minimum. To prune more aggressively, now a subgraph will be pruned iff its cost plus a lower bound on the remaining vertices is bigger than the current minimum. A graph qualifies as a lower bound if its cumulative cost is below the sum of the costs of the vertices added to the TSP solution;

Here, the Nearest Neighbour lower bound will be used[5]. The NN graph will be computed by connecting each vertex to its nearest neighbour. Note that the resulting graph can have multi-edges and doesn't have to be fully connected.

Let p be our recursively enumerated subpath, $v_1, \dots, v_m$ be the free vertices (i.e. $v_i \notin p$), and $c : V \times V \to \mathbb{R}$ be the cost function. The nearest neighbour graph can be computed in $\Theta(m^2)$ using the following formula:



(a)

$$c_{NN} := \sum_{i \in \{1,\dots,m\}} \min_{\substack{j \in \{1,\dots,m\} \\ i \neq j}} c(i,j)$$

(b)

Figure 3: An example subgraph with a NN lower bound (a) and the formal for computing the total cost of an NN (b).

Note that this is an obvious improvement over the previous lower bound for pruning since $p + c_{NN}$ is a tighter bound than $p + 0$.

Version 5: Prune using the MST Metric: This version uses the same approach as version 4, but instead of computing a NN graph, it computes the previously defined MST from the remaining vertices. This is an even tighter bound.

[5]Not to be confused with the Nearest Neighbour algorithm used for approximating a solution.

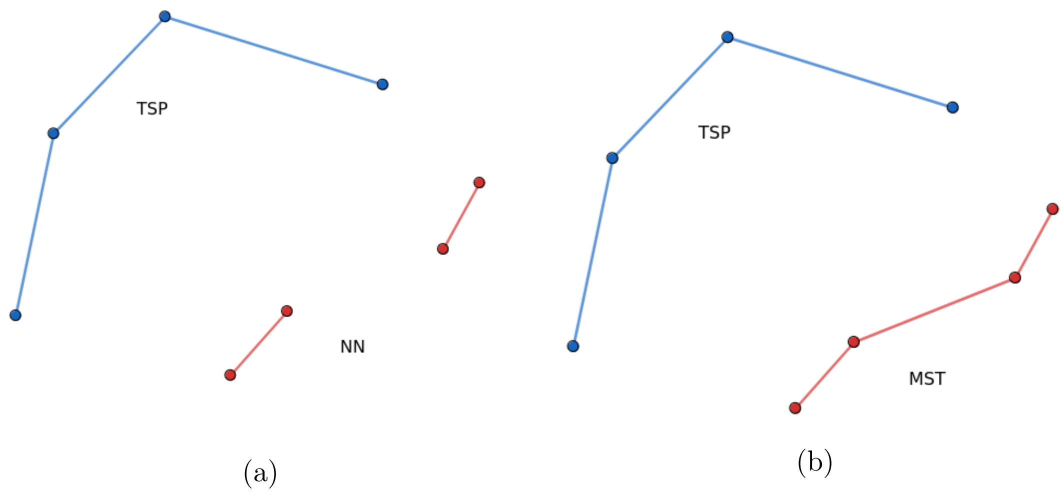


Figure 4: Comparison between the NN (a) and MST (b) graph of the remaining vertices for an example graph.

Note that it is not obvious that this algorithm will perform better than the previous versions. While it improves its pruning ability, it also adds the cost of computing all MSTs. In the next and last version, those MSTs will be cached to reduce redundant computations.

Version 6: Cache the previous MSTs: The last version builds upon version 5, but caches the MSTs so that the amount of redundant compute is reduced. This requires a fast MST lookup; in our code a HashMap was used, resulting in $O(1)$ average lookup time. Furthermore, instead of using the default HashMap, a cryptographically insecure but overall more performant HashMap was used [9][6].

2.3.2 Prefix Space Partitioning

For explaining both the shared and distributed parallelization algorithms, the concept of **prefix space partitioning** is required.

As explained before, a TSP solution of an n-vertex graph can be viewed as an n-dimensional vector. Furthermore, since the paths are generated using recursive enumeration, the prefix of a path can be used to compute all subpaths containing that prefix. This means that the prefixes of a given length form an equivalence relation on the set of all paths. Since all equivalence classes have the same size, one can evenly partition the work by dividing the number of prefixes by the number of workers.

This mapping is archived by interpreting a prefix as a n-ary number.

Definition 2.2. *Mapping paths to n-ary numbers: A path $v := (v_1, \dots, v_m)$ of an n-vertex graph can be mapped to a number by interpreting the i-th element as the i-th digit of an n-ary number. Formally, the mapping function $\rho_{n,m}$ can be defined as*

$$\rho_{n,m} : \{1, \dots, n\}^m \to \mathbb{N}$$

$$\rho_{n,m}(v_1, \dots, n_m) := \sum_{i=1}^{m} n^{i-1} v_i$$

[6]As the name implies, `rustc-hash` is also used in the compiler itself and maintained by the Rust core team.

Let $\rho_{n,m}^{-1}$ be defined as the inverse, i.e. $\rho_{n,m}^{-1}(\rho_{n,m}(x)) = x$.

Example 2.3. *This is analogous (but reversed) to how natural numbers in base* 10 *can be defined.*

$$(12345)_{10} = 5 \cdot 10^0 + 4 \cdot 10^1 + 3 \cdot 10^2 + 2 \cdot 10^3 + 1 \cdot 10^4 = \rho_{10,5}(5, 4, 3, 2, 1)$$

Example 2.4. *This is also how binary numbers are converted into the decimal system.*

$$(10001)_2 = 1 \cdot 2^0 + 1 \cdot 2^4 = 9 = \rho_{2,5}(1, 0, 0, 0, 1)$$

Now, with this number mapping in mind, the prefix space partitioning algorithm can be properly defined:

Definition 2.5. *Prefix Space Partitioning Algorithm: Given a graph with n vertices, a prefix length of m and p workers, the i-th worker can compute his prefix range as follows:*

1. *Compute the total number of prefix values, including invalid paths: n^m.*
2. *Calculate the i-th chunk of all path ids: $[l, r) := [n^m/p \cdot (i - 1), n^m/p \cdot i)$*
3. *Map and return the prefixes associated with those bounds:*
   - *Starting Value: $\rho_{n,m}^{-1}(l)$*
   - *Ending Value (exclusive): $\rho_{n,m}^{-1}(r)$*

Remark. Note that this prefix space partitioning can be done locally on each worker without any communication needed by using its rank and the world size.

2.3.3 Shared Memory Parallelization

With the idea of prefix space partitioning in mind, the shared memory parallelization algorithm is straightforward:

1. Start n threads. n can be manually specified, otherwise it defaults to `std::thread::available_parallelism`.
2. Each thread computes its prefix space using prefix space partitioning.
   The prefix length can be manually specified, otherwise it defaults to 3.
3. Each thread computes each valid tour in its prefix space.

The threads prune using the MST lower bound of the version 5 sequential algorithm. The current minimum is potentially updated after every tour. Synchronization is done via a mutex, of which all threads get an atomically counted reference. Note that we do not cache the MSTs as the performance boost was negligible while resulting in a lot of locking.

2.3.4 Statically Partitioned Distributed Memory Parallelization

The statically partitioned, MPI-based solver works analogously to the multi-threaded version, using both the MST lower bound as well as the prefix space partitioning.

This means that it

- divides up the work through prefix space partitioning.
- goes through all possible solutions that can't be pruned away.
- generates all possible paths through recursive enumeration.
- caches the sum of the partial path throughout the recursion.
- prunes iff the partial sum plus the MST lower bound is bigger than the known optimum.

Now to the communication. Let us assume that the communicator world size is n. We choose rank 0 as the communicator and rank $\{1, \dots, n-1\}$ as the workers.

Using prefix space partitioning[7], each worker knows the prefixes it has to process. In order to prune, every time a worker finishes a prefix, it sends its current lowest cost it ever encountered to the coordinator. This is done even if it was not improved during that prefix.

It is done because this message is also used as an update request for the newest global minimum known from the coordinator. After taking the worker's minimum into account, it returns the global minimum that all workers ever archived. Note that only the cost and not the full path is sent to minimize the amount of traffic between the communicator and the workers. After all assigned prefixes are computed, the worker waits at a barrier for the other workers to complete.

Since the workers prune, the coordinator can not know how many requests it can expect from each worker. Thus, each worker has to tell the coordinator when it is done, otherwise, it will deadlock waiting for yet another processed prefix.

This is done by sending another message with a negative cost. The coordinator tracks how many of those messages were received. Once the coordinator receives $n-1$ messages it stops listening. Receiving $n-1$ finish messages implies that all workers are already waiting at the barrier. Thus, the coordinator joins the barrier node; breaking the barrier and starting the wrapup.

After the barrier was broken, the coordinator broadcasts which rank won with which cost. Therefore, all workers know who won. The winner then finally broadcasts the winning tour to every other node. Thus, in the end, every worker knows the best cost from the coordinator and the best path from the winner. Again, note that the wrapup is a network efficiency optimization since it allowed us to not send the current best path to the coordinator each time it was improved by a worker.

2.3.5 Dynamically Partitioned Distributed Memory Parallelization

This algorithm is an optimization of the previous MPI-based algorithm. This is only done by changing the communication scheme. Thus, the algorithmic steps are the same as the statically partitioned algorithm.

[7]Variadic prefix length, defaults to 3 vertices.

In the static solver, the load was divided locally through the rank and prefix space partitioning. This is easy to compute since one can just divide with n-ary numbers.

But, especially since the problem is factorial, the problem should be pruned as much as possible. In the static version, this was done by telling the root the local minimum it currently knows, and as an answer receiving the global current minimum with that answer in mind. *Note that this requires one bidirectional communication per prefix.*

This approach has one disadvantage. Note that, while the pruning greatly optimizes the average-case scenario, it does not improve the worst-case scenario. The algorithm performs great, but its performance is very dependent on the graph and its pruneability. The same logic applies to all subgraphs with fixed prefixes.

The workers have (potentially) vastly different workloads, depending on how well they can prune. Since the coordinator has to wait for all workers to finish before it knows the global best solution, all finished workers have to idle, wasting precious compute. Even more important, global work coordination wouldn't even increase the amount of communications, since we do one bi-directional send and receive per prefix nonetheless.

Thus, instead of using prefix space partitioning to predivide it statically, the computation workflow is as follows:

1. The worker asks the coordinator for a new prefix to compute[8].
2. The coordinator answers with the next non-computed prefix as well as the current minimum.
   If all prefixes are computed, the worker receives a prefix with all zeroes, which means it waits at a barrier for the others to finish.
3. The worker computes the current prefix given to it.
   It uses the current global minimum given with the prefix to prune accordingly.
4. The worker returns its local minimum to the work node.
   This is an implicit ask for more work.
   The root node can use that node-specific minimum to update the global minimum if needed.

Although the root could already know which prefix resulted in the global minimum by keeping track of which ones it assigned to whom, it does not know the whole minimal path. Thus, it needs the same wrapup as the static solver.

After the barrier was broken, the coordinator broadcasts which rank won with which cost. Therefore, all workers know who won. The winner then finally broadcasts the winning tour to every other node. Thus, in the end, every worker knows the best cost from the coordinator and the best path from the winner. Again, note that the wrapup is a network efficiency optimization since it allowed us to not send the current best path to the coordinator each time it was improved by a worker.

2.4 Approximate Solving

Walky provides two different approximation algorithms for solving the TSP: The *Nearest Neighbour* algorithm, which provides a simple but not tight approximation, and the *Christofides* algorithm, which is more sophisticated but produces a tighter bound.

[8]For easier MPI communication, prefixes have a fixed length of 3 vertices.

2.4.1 Nearest Neighbour

The 1-nearest-neighbour algorithm[9] is a simple, straightforward greedy algorithm to compute an approximation for the TSP. It works as follows:

Algorithm:

- Choose a random starting node.
- From that node, greedily visit the nearest node which was not visited before.
- Repeat until all nodes are visited.
- Return to the starting node to close the tour.

In walky, the nearest neighbour algorithm computes the 1-nearest-neighbour for all starting nodes and returns the minimum. Since the 1-nearest-neighbour has a complexity of $\Theta(n^2)$, the nearest neighbour has a complexity of $\Theta(n^3)$.

Shared-Memory Parallelization: The parallelization was done trivially. Using the `rayon` create, all staring nodes were processed parallelly using a preallocated thread pool. The results were then reduced to get the global minimum.

Distributed-Memory Parallelization: The MPI-based implementation uses a simplified version of the aforementioned prefix space partitioning.

This algorithm requires no coordinator, i.e. all nodes are workers. The 1-nearest-neighbour computations will be done sequentially, the parallelization consists of the workload partitioning through the starting nodes.

Given an n-vertex graph and m workers, the i-th rank computes the 1-nearest-neighbour for the starting nodes $[n/m \cdot i, n/m \cdot (i+1))$. Since every worker knows its rank, this can be computed locally and independently without any communication.

Once the minimum of that local chunk has been found, the MPI communication starts. First, through an `ALL_REDUCE` on the cost every worker finds out the best cost and which worker won. After that, the best worker knows that it won. It then proceeds to broadcast the whole path to everyone. Analogously to the static exact solver, this is a network efficiency optimization by only sending one path through the network at the end.

Now every worker knows the global minimum and can successfully return it.

2.4.2 Christofides Algorithm

Next is an algorithm that requires more assumptions on the input, but also gives an approximation to the TSP that is guaranteed to have a weight $\leq 1.5 \cdot \omega$, with ω being the weight of the optimal solution [10]. The algorithm requires two preliminary definitions.

Definition 2.6 (Matching)**.** *See [11]. Let $G = (V, E)$ be a Graph. A set $M \subseteq E$ is called* matching *of G, if all edges in M are pairwise disjoint*

$$\forall x, y \in M : x \cap y = \emptyset$$

.

[9]Not to be confused with the nearest neighbour lower bound used for exact solving.

Definition 2.7 (Perfect Matching)**.** *See [11]. Let $G = (V, E)$ be a graph and $M \subseteq E$ be a matching of G. Then, M is called* perfect*, if M spans the whole graph*

$$\forall v \in V \, \exists e \in M : v \in e$$

.

Now, the Christofides Algorithm can be defined.

Definition 2.8 (Christofides Algorithm)**.** *Let $G = (V, E)$ be a metric Graph (i.e. the triangle inequality holds). The Christofides algorithm is a five-step procedure [10]:*

1. *Calculate the MST of G: $MST = (V_M, E_M)$.*
2. *Calculate an exact matching in $S := (E_S, V_S)$.*

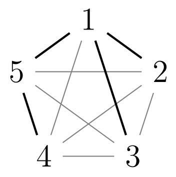
   Let $V_S := \{v \in V_M | \deg_M(v) \equiv 1 \mod 2\}$ the set of all vertices that have odd degree in the MST. Then let $E_S := \{e \in V_G | e = (e_1, e_2) \wedge e_1 \in V_S \wedge e_2 \in V_S\}$. Of all possible perfect matchings, choose one with minimal weight.
3. *Combine the MST and the matching into one multigraph.*
4. *Find an Eulerian cycle through the multigraph.*
5. *Make the Eulerian cycle Hamiltonian.*

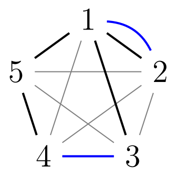
Remark. According to [10], in step 2 a perfect matching can always be found.

Some of the above steps use constructs that are not defined yet. Those steps will be elaborated on below. In contrast to that, for step 1 simply refer to section 2.1.

Explaining Step 2 Of Christofides Algorithm In definition 2.8, only a high-level description for step 2 was provided. To provide further explanation, a visualization of that step on the K_5 graph follows.

Let $G = K_5$. Edge weights are left out for the simplicity of visualization. In the left graph, an MST of G is highlighted with bold edges in black. The vertices of odd degree w.r.t the MST are: $1, 2, 3, 4$. The task is then to find a matching over these vertices. The resulting matching is visualized in the right graph with blue edges.





Finding A Matching Executing the Christofides Algorithm involves finding a minimum-cost perfect matching. In a sequential setting Edmonds's Blossom Algorithm[10] gives an exact solution to the problem. Instead of this algorithm, a naïve randomized approximate solution was implemented, to be able to utilize parallelization.

The randomized approximate solution follows this idea: randomly guess a matching and do some randomized improvements. Repeat this and take the matching with minimal cost. This solution is easy to implement and easy to parallelize.

[10]For a definition and live demonstration see `https://algorithms.discrete.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/index_en.html`.

Definition 2.9 (Algorithm: Finding an initial matching)**.** *Let $G = (V, E)$ be a complete graph with an even amount of vertices $|V| = 2n, n \in \mathbb{N}$. W.l.o.g. let $V = \{1, \ldots, 2n\}$. Select a permutation $\pi : V \to V$ uniformly at random. Now*

$$M = \{\{\pi(1), \pi(2)\}, \ldots, \{\pi(2n-1), \pi(2n)\}\}$$

is a perfect matching of G.

Now, focus on improving a matching consisting of only 2 edges.

Theorem 2.10 (Minimum-cost perfect matching on K_4)**.** *Given the Graph $K_4 = (V, E)$ w.o.l.g. $V = \{\{1, 2\}, \{3, 4\}\}$. Then exactly 3 perfect matchings exist on K_4:*

$$\begin{tikzcd} 1 \arrow[d, dash] & 3 \arrow[d, dash] \\ 2 & 4 \end{tikzcd}, \quad \begin{tikzcd} 1 \arrow[r, dash] & 3 \\ 2 \arrow[r, dash] & 4 \end{tikzcd}, \quad \begin{tikzcd} 1 \arrow[dr, dash] & 3 \arrow[dl, dash] \\ 2 & 4 \end{tikzcd}$$

A matching with the lowest cost is a minimum-cost perfect matching on K_4.

Proof. When constructing a matching for K_4, there are $\binom{4}{2} = 6$ options for the first edge $e_1 = \{a, b\}$. The choice for the second edge immediately follows as $e_2 = \{c, d\}$, by the property of a matching: $e_1 \cap e_2 = \emptyset$. Then $M = \{e_1, e_2\}$. Because the order of edges is unimportant for the matching, for every choice of (e_1, e_2), there is one choice (e'_1, e'_2) with $e_1 = e'_2 \land e_2 = e'_1$. Therefore, there are $\frac{6}{2} = 3$ perfect matchings on K_4. □

With this theorem, one can improve matchings of arbitrary size.

Definition 2.11 (Randomly Improving A Matching)**.** *Let*

$$M = \{m_0, \ldots, m_{n-1}\}$$

be a perfect matching on K_n, $n \in \mathbb{N}$. Repeat the following procedure $k \in \mathbb{N}$ times:

Uniformly at random select a permutation $\pi : M \to M$. For every perfect matching

$$M'_i = \{\pi(m_{2i}), \pi(m_{2i+1})\}, \qquad i = 0, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$$

on the corresponding subgraph $K' \subseteq K_n$ with $K' \cong K_4$, use theorem 2.10 to compute a minimum-cost perfect matching $\widetilde{M'_i}$. Then set

$$M \leftarrow \left\{ \widetilde{M'_i} | i \in \left\{ i = 0, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \right\} \right\} \cup \begin{cases} \{\pi(m_{n-1})\} & , \text{ if } n \text{ is odd} \\ \emptyset & , \text{ else} \end{cases}$$

.

Remark. The application of theorem 2.10 on each M'_i can be parallelized, as the computation for each i is independent of all other computations.

Last Steps Of Christofides Algorithm For step 4 one needs to compute an Eulerian cycle. This is done with an implementation of Hierholzer's algorithm [12][13, Algorithm X.4].

Finally, step 5 is easily done using the metric property of the complete input graph: Start the Hamiltonian cycle at some vertex and keep track of which vertices have been visited while traversing the Eulerian cycle. If a vertex has not been visited, add it to the Hamiltonian cycle. If a vertex has already been visited, simply skip it and proceed with the next vertex. Such shortcuts exist because the graph is complete, and taking these shortcuts does not increase the weight of the solution, since the graph is metric whereby the triangle inequality holds.

Parallelization There is a parallel implementation using rayon, and one using MPI. Both only parallelize the improvement of randomized matchings, as in Def. 2.11.

The shared memory implementation distributes all the consecutive pairs of edges between the available threads, and computes the improved 2-matchings (see Def. 2.10) in parallel. Then, a new matching is constructed from all partial values of the threads. The details of synchronization and message passing are handled by the rayon `ParallelIterator` implementation.

The MPI-based parallelization uses a different approach: The randomized improvement of the matching is done independently at each process, only at the end the results are gathered at the root process and the matching with the least cost is chosen. For this, each process sends the local solution and the corresponding solution weight to the root process per MPI unicast. This is not optimal, as the optimal solution weight could be determined by a global reduction, such that only one full solution vector would need to be sent. Because the Rust MPI interface `rsmpi`[11] does not support the `MPI_MAXLOC`[12] operation at the moment, but the location of the maximum weight is needed, a simple reduction would not suffice.

2.5 Lower Bound

So far, there are multiple techniques for approximation a solution to the TSP, each result provides an upper bound on the exact solution. Now comes a brief look at computing lower bounds to the TSP.

2.5.1 MST Lower Bound

The MST of a graph is, in comparison to the TSP, fairly easy to compute. It turns out, that the MST provides a lower bound to the TSP in a very natural way.

Theorem 2.12 (MST lower bound)**.** *Let G be a weighted, complete graph. Let T_G be an exact solution to the TSP on G, with weight w_T. Let M be an MST of G, with weight w_M. Then*

$$w_M \leq w_T$$

holds.

Proof. T is the solution to the TSP on G. By definition, T then is an Eulerian cycle, that visits each vertex in G exactly once. Therefore, by removing any edge from T, one yields a spanning tree S of G, with weight w_S. By definition,

$$w_M \leq w_S \leq w_T$$

follows. □

2.5.2 1-tree Lower Bound

One can improve upon the MST-based approach, by using a variation of the MST, called minimum-weight 1-tree.

[11]This will be introduced in more detail in section 3.4.

[12]For a description of the operation see [14, section 5.9.4].

Definition 2.13 (1-tree)**.** *See [15, p. 1139]. Let $G = (V, E)$ be a graph. Let $\{v_1, \ldots, v_n\} = V$ be some enumeration of all vertices V. Then a 1-tree on G is defined to be*

1. *a tree, when restricted to the vertices $\{v_2, \ldots, v_n\}$,*
2. *and to have exactly one cycle. This cycle goes through vertex v_1.*

In the extreme case, such a 1-tree can be a cycle in a graph.

Theorem 2.14 (Cycle as a 1-tree)**.** *Let G be a Graph, and C be a circle in G. The C is a 1-tree.*

Proof. Let $G = (V, E)$, and let $V = \{v_1, \ldots, v_n\}$ be some enumeration. Then $C|_{\{v_2,\ldots,v_n\}}$ is a tree. By definition, C contains exactly one cycle, and it goes through v_1. Therefore, all properties of Def. 2.13 apply to C. □

The 1-trees for the lower bound must also be of minimal weight.

Definition 2.15 (minimum-weight 1-tree)**.** *See [15, p. 1139]. Let $G = (V, E)$ be a weighted graph. Let $\{v_1, \ldots, v_n\} = V$ be some enumeration of all vertices V. Then a minimum-weight 1-tree on G is a 1-tree, that also is*

1. *an MST when restricted to the vertices $\{v_2, \ldots, v_n\}$,*
2. *and the special vertex v_1 is connected to the rest of the 1-tree with the two distinct cheapest edges.*

Remark. For every choice of the special vertex v_1, there is a possibly distinct minimum-weight 1-tree. All of these 1-trees can be computed independently of each other, which is easy to parallelize. This point will be revisited later.

Combining this knowledge, one can now construct a tighter lower bound on the TSP.

Theorem 2.16 (1-tree lower bound)**.** *Let G be a weighted, complete graph. Let T_G be an exact solution to the TSP on G, with weight w_T. Let $\mathcal{M}$ be the set of all minimum-weight 1-trees of G, and let $W_{\mathcal{M}}$ be the set of the corresponding weights. Then*

$$\max_{w_M \in W_{\mathcal{M}}} w_M \leq w_T$$

holds.

Proof. Let $M \in \mathcal{M}$ with weight w_m, s.t.

$$w_M = \max_{w \in W_{\mathcal{M}}} w$$

. Then, let v_1 be the special vertex of M. T is an Eulerian cycle, thus by Theorem 2.14 it is a 1-tree on G, with special vertex v_1. Since M has minimal weight,

$$w_M \leq w_T$$

follows immediately. □

Remark. When removing one edge incident to the special vertex v_1 from a 1-tree, one yields a spanning tree. Thus, the 1-tree lower bound is tighter than the MST lower bound.

The sequential implementation is straightforward and follows directly from the above-mentioned definitions. It leverages the fact, that the used implementation of Prim's algorithm is capable of ignoring one vertex in a given graph, which is then used to construct the 1-trees.

There are two parallel implementations provided, one using rayon, and one using MPI.

For rayon, the parallelization is again handled by the `ParallelIterator` implementation. For every vertex in the graph there is a minimum-weight 1-tree (see Def. 2.15). In principle, these 1-trees can be computed in separate threads. At the end, the maximum of the tree weights is computed over all threads.

The MPI version follows the same idea. Each process knows their rank r and the size s of the MPI world. Then, the process computes a minimum-weight 1-tree for each vertex u, where $u \equiv r \mod s$. Afterwards, using an MPI reduction, the maximum over all processes is collected at the root process.

3 Implementation

3.1 Rust

Rust [16] is a systems programming language initially released by Mozilla Research in 2015. It was designed as a memory-safe alternative for C++ in Servo [17], which is the web rendering engine used in Firefox. Rust's main goal is to provide memory safety while having an on-par performance with other systems languages such as C or C++. Having memory safety is paramount; research suggests that in memory-unsafe languages, at least 65% of the security vulnerabilities are caused by memory unsafety. This was discovered simultaneously at Android [18] [19], iOS and macOS [20], Chrome [21], Microsoft [22], Firefox [23], and Ubuntu [24].

Specifically, Rust is great for HPC since one can think of it as a modern dialect of C++ enforced by the compiler. It uses RAII[13] internally to ensure memory safety, while references are roughly equivalent to `std::unique_ptr`. While Rust's ecosystem itself is still maturing, due to its clean Foreign Function Interface (FFI) and simple bidirectional C++ [25] and Python [26] interoperability allows for seamless integration into a typical HPC environment. Analogously, while the Rust compiler is comparatively new, it already supports most compiler optimizations by leveraging Low Level Virtual Machine (LLVM) as a compiler backend. It is natively compiled without a garbage collector.

Lastly, with its many functional patterns, it is a very loved language by the industry and developers alike. According to the yearly StackOverflow survey it was voted as the most loved language for the 7th year in a row [27]. It rapidly gets adopted by big tech firms such as AWS [28], Google [29], Meta [30], and Microsoft [31] and is even accepted as a language for the Linux kernel [32].

Walky requires an Minimum Supported Rust Version (MSRV) of 1.70.0[14]. Since MPI support is hidden behind a feature flag, no MPI is required for compilation.

3.2 Compiler Optimizations

To ensure the best possible performance, the following compiler optimizations were explicitly enabled:

- **Release Builds (`-O3`)**: If one does not use the release build[15] the code is not optimized. This enables several general optimizations as well as automatic vectorization.
- **LLVM Link Time Optimization (LTO)**: LTO enabled further, intermodular optimizations during the link stage. While this could improve code by optimizing beyond library bounds, it increases compile time, which is why it is disabled by default.
- **Compiling for Native Architecture:** When compiling for the native architecture[16] the compiler can use more specialized instructions that are not available on

[13]Resource Acquisition Is Initialization (RAII)

[14]This does not require that 1.70.0 or higher is available as a cluster module because Rust uses `rustup` [33] for easy userspace installation management.

[15]With `cargo build --release`.

[16]Using the `RUSTFLAGS` environment variable, i.e. `RUSTFLAGS="-C target-cpu=native" cargo build --release`.

every processor such as bigger vector registers for SIMD. Note that this may create binaries incompatible with other systems.

- **Using a single LLVM codegen unit:** Codegen units are analogous to translation units. This means that, when changing a single file, just the codegen unit in that file has to be recompiled. Therefore, optimizations can't be done beyond codegen unit bounds! Using a single codegen for the whole project allows the compiler to more aggressively optimize globally. Note that this effectively disables partial compilations.

3.3 Command Line Interface (CLI)

The `walky` binary implements a Command Line Interface (CLI) using the crate `clap`, structured with subcommands.

For the following demonstration of the CLI, the `walky` binary has been compiled with

```
$ cargo build --release --features mpi
```

`walky` displays an overview of its subcommands:

```
$ walky --help
A TSP solver written in Rust

Usage: walky <COMMAND>

Commands:
  exact        Find the exact best solution to a given TSP instance
  approx       Find an approximate solution to a given TSP instance
  lower-bound  Compute a lower bound cost of a TSP instance
  help         Print this message or the help of the given subcommand(s)

Options:
  -h, --help     Print help
  -V, --version  Print version
```

Each subcommand follows the same basic syntax.

```
walky SUBCOMMAND [OPTIONS] <ALGORITHM> <INPUT_FILE>
```

Users need to specify a subcommand. Then, they can provide optional parameters, after which they need to specify a concrete algorithm to use and a file path to the input file. The full help for every subcommand is listed in section B.1.

3.4 Parallelism Libraries

For both shared- and distributed memory parallelism, specialized Rust crates were used:

Shared Memory Parallelism: For shared memory parallelism we used the rayon [34] crate. Rayon is a high-level parallelism library using dynamically sized thread pools. It guarantees *data-race freedom* by allowing only one thread to write at a time. Its main features are drop-in parallel iterators: By replacing `.iter()` with `.par_iter`, it is possible to use all functions provided for iterators, such as `.map()`, `.filter()`, `.reduce()` for typical functional patterns or `.join(|| a(), || b())` enabling the `fork-join` computation model. Thus, when initially written in an iterator-focused, functional way it allowed easy parallelization of the previously designed sequential algorithms.

Distributed Memory Parallelism: For distributed memory, multi-node parallelism the HPC native MPI environment was leveraged using the rsmpi crate [35]. rsmpi is a Rust-native MPI implementation[17] compliant with MPI-3.1. It is tested to be compatible with OpenMPI, MPICH, and MS-MPI for Windows. It supports most MPI features, such as blocking and non-blocking point-to-point communication, and most collective communications such as broadcasts or scatter/gather as well as aggregations such as reductions.

3.5 Correctness and Tests

To ensure the algorithm's correctness, both tests and runtime precondition checks were implemented:

Testing: Traditional testing was done using the `cargo-nextest` [36] for parallelized unit tests. The algorithms were tested using pre-computed examples[18], working as follows:

1. Generate a metric, fully connected, undirected graph by placing n 2D points onto a space and calculating their pairwise distance. This ensures the triangle inequality.
2. Solve the TSP for the graph using Python's battle-tested `python-tsp` [37].
3. Generate the input and output Rust code for the test cases using Python.

Preconditions: At runtime, the following preconditions are checked at runtime before any algorithm starts to ensure correctness:

- **Fully Connected:** The TSP is only defined for fully connected graphs, i.e. every node has a connection to any other node. This is always true for any real-world examples with metric spaces.
- **Undirectedness:** The TSP is also only defined for undirected graphs. This means that both directions of an edge should have the same cost, i.e. for any two vertices A, B the edge from A to B should have the same cost as the edge from B to A.
- **No Multiedges:** Lastly, we require that no multi-edges exist. This means that for any two edges A and B, there exists only one direct connection.

[17]With FFI bindings to other implementations through the beforementioned bindgen.

[18]Note that the Python tooling for the test case generation can still be found in `./utils` in the walky repository

3.6 CI pipeline

Furthermore, to keep the code quality high, a sophisticated CI pipeline was created, running on each commit on `main` as well as any pull request. It consists of the following steps running in parallel:

- **Build:** First and foremost, it is checked that the current version builds with release settings using `cargo build`. Note that, due to the limited Ubuntu CI runner, the MPI feature is disabled.
- **Tests:** Next, the automated unit tests are run using the aforementioned `cargo nextest`.
- **Formatter:** After that, the code formatting is verified using `cargo fmt`. The default Rust standard formatting is used.
- **Linter:** Also, the general linter `cargo clippy` is run to prevent common mistakes and ideomatize walky.
- **Documentation Linter:** Lastly, `cargo doc` is used to verify and lint our docstring documentation, on which our HTML-based documentation is based.

3.7 Documentation and Releases

Lastly, to improve the User Experience (UX) for walky, complete documentation and release management were set in place. Walky uses Semantic Versioning. When releasing a new version, the following artifacts become available:

- **Registry Upload:** The source code becomes available at `crates.io`, which is the default Rust crate registry. This results in being able to install walky using `cargo install walky`.
- **Hosted HTML-Documentation:** Whenever releasing a new version onto `crates.io`, an up-to-date, full test searchable HTML documentation becomes available at `docs.rs`[19].

[19]For walky: `https://docs.rs/walky/latest/walky/`

4 Performance Analysis / Evaluation

In this section, a performance analysis is done on all algorithms implemented in walky. Where applicable, problem size scaling, strong scaling, and MPI scaling analysis are done per algorithm. With problem size scaling, the problem size is increased with a fixed amount of parallelism while with strong scaling, the parallelism is increased with a fixed problem size. Due to the missing Score-P support for LLVM and especially cargo, the MPI analysis was done purely mathematically.

4.1 Cluster Setup

The benchmarks were done on the Scientific Compute Cluster (SCC) at GWDG, the joint data center of Max Planck Society for the Advancement of Science (MPG) and University of Göttingen.[20]. The SCC is a large HPC system, consisting of about 410 compute nodes with over 18000 CPU cores, 99TB RAM, and 5.2 PiB of storage, split into two filesystems. For our benchmarks, the so-called `amp` node type, of which 96 exist at the SCC, was used. An `amp` node has two Xeon Platinum 9242 with a total of 48 CPU cores running at a frequency of 3.8 GHz. Each node has 384 GB of RAM. The jobs were assigned using the internal SLURM workload manager.

4.2 Vampir-based Analysis of Rust MPI Code

It was initially planned to do the distributed analysis of the MPI code using Vampir [38]. Vampir internally uses Score-P [39] for the generation of trace log files.

Unfortunately, Score-P is not supported by the Rust compiler. While there is literature to show that Score-P can be integrated into the LLVM ecosystem [40], the source code was never released. Furthermore, what makes matters worse is that, even if the tool were published, it is still not trivial to include it into the cargo build process, which is required for building our third-party dependencies[21].

Note that this problem is not just contained to Vampir. Other common analysis tools such as Scalasca [41] or TAU [42] also rely on Score-P internally.

Therefore, the MPI analysis will be theoretically by mathematically scaling the number of messages and bytes sent as well as their temporal relationship.

4.3 Exact Solving Benchmarks

4.3.1 Problem Size Scaling

All single threaded iterations of the algorithm, together with the rayon-based multi-threaded version with 24 threads, were run for 24h on the cluster, sequentially computing a random graph from size 3 to size 50. Here are the results:

[20] `https://gwdg.de/en/hpc/systems/scc/`

[21] As many modern languages do, Rust does not specify an Application Binary Interface (ABI) and instead recompiles all subdependencies (for bounds, the C FFI convention is usually used). This means, that one can't just link against system-wide libraries as in C.

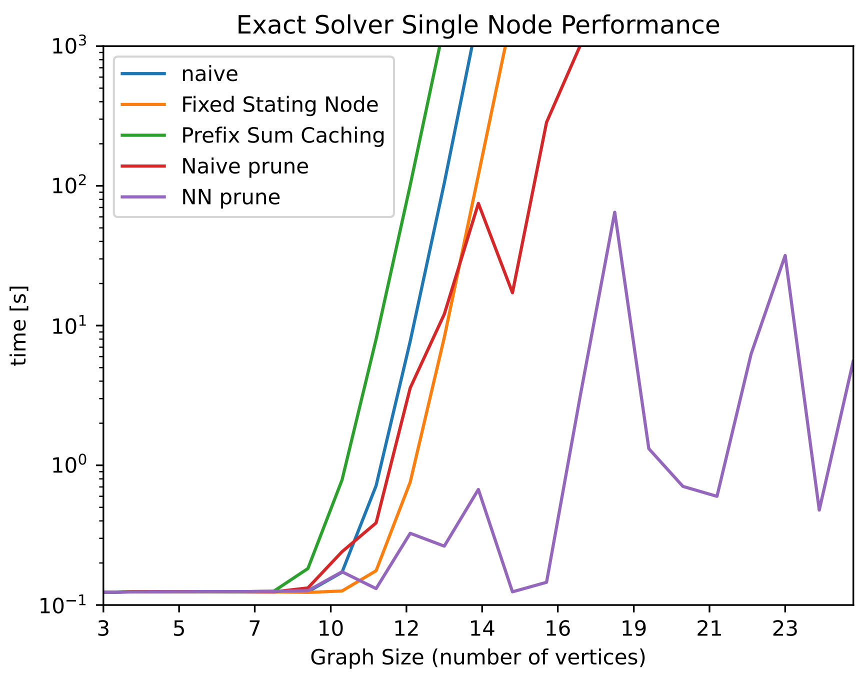


Figure 5

Figure 6: The results of the exact solver. shows all algorithms until the NN prune

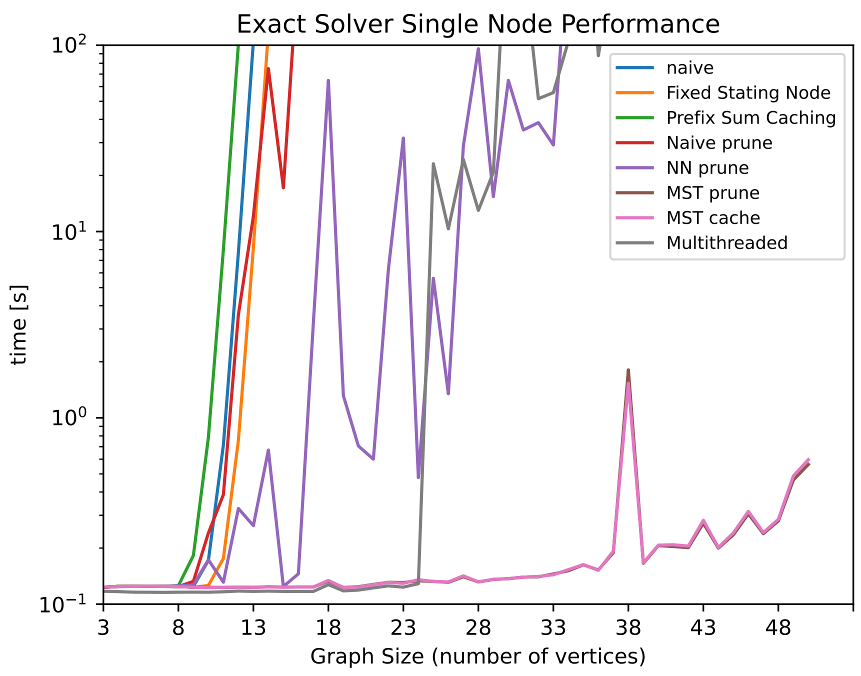


Figure 7

Figure 8: The results of the exact solver. shows all algorithms with a smaller y-axis

For the full results see Table **??** in the appendix.

The most important insight is that pruning, especially MST-based pruning, immensely improves the viability of exact solving. While $n = 14$ took over 20 minutes with the naïve version, it was computed in around 0.11 seconds using the optimized pruner. It was possible to compute $n = 50$ in around 0.3 seconds[22].

Other important insights include the following:

- The prefix sum caching was even slower than the naïve implementation! This was because we changed from the fast iterative permutation algorithm to recursive enumeration. There are several reasons why this is such slow, from function call overhead to less compiler optimization opportunities to not being tail call optimized. But by the time naïve pruning was used, it already consistently outperformed the initial $v0$ on the randomly generated graphs.
- While pruning generally improves performance, it does not improve performance deterministically, which is shown in the spikes on less-prunable graphs.
- The MST performance is insanely good. Remember that this is still an $\mathcal{O}(n!)$ worse case algorithm.
- The caching did not improve the previous algorithm enough to justify the added complexity. In fact, depending on the graph, it could worsen overall performance.

Last but not least, the multithreaded version performed way worse than the sequential one it was based upon. There are several possible reasons for this behaviour:

- Context-switching and threading overhead caused by the operating system.
- Blocking inter-thread locking on the current best solution, which was in a mutex that every thread got an atomically reference counted pointer for.
- Worse cache utilization. The more threads run on the system, the more context switching. Every time the thread is switched, the CPU caches are flushed by the previous program. Furthermore, hyperthreading always at least halves the cache if not destroys it completely by two threads greedily competing for it.

4.3.2 Strong Scaling

The strong scaling analysis of the MPI-based, distributed exact solver requires a more sophisticated analysis. Before looking at the plotted data, let us compare the statically allocated (V0) algorithm to the dynamically allocated (V1) algorithm:

[22]The benchmark was stopped at $n = 50$.

Type	Total Worker	V0 μ	V0 σ	V0 Efficiency	V1 μ	V1 σ	V1 Efficiency
1n2p	2	815.612	0.485	72.093	134.508	0.062	437.148
2n1p	2	815.850	0.453	72.072	134.787	0.091	436.245
1n4p	4	750.874	1.011	39.154	100.460	9.940	292.654
4n1p	4	746.932	0.678	39.361	63.301	0.077	464.451
8n1p	8	19.223	0.075	764.690	38.952	0.069	377.388
1n8p	8	19.401	0.013	757.705	39.311	0.683	373.945
1n16p	16	11.364	0.022	646.768	54.570	0.654	134.691
2n16p	32	50.456	0.534	72.836	70.250	0.566	52.313
4n16p	64	989.302	1.536	1.857	81.203	0.458	22.628

Table 1: The results of the exact MPI solver. Efficiency is computed as prefixes per worker per second. The type αnβp stands for α computing nodes with β workers per node.

As we can see, the worker topology of the MPI processes did not change performance significantly. Thus using the smallest mean for any total worker size, we get the following results:

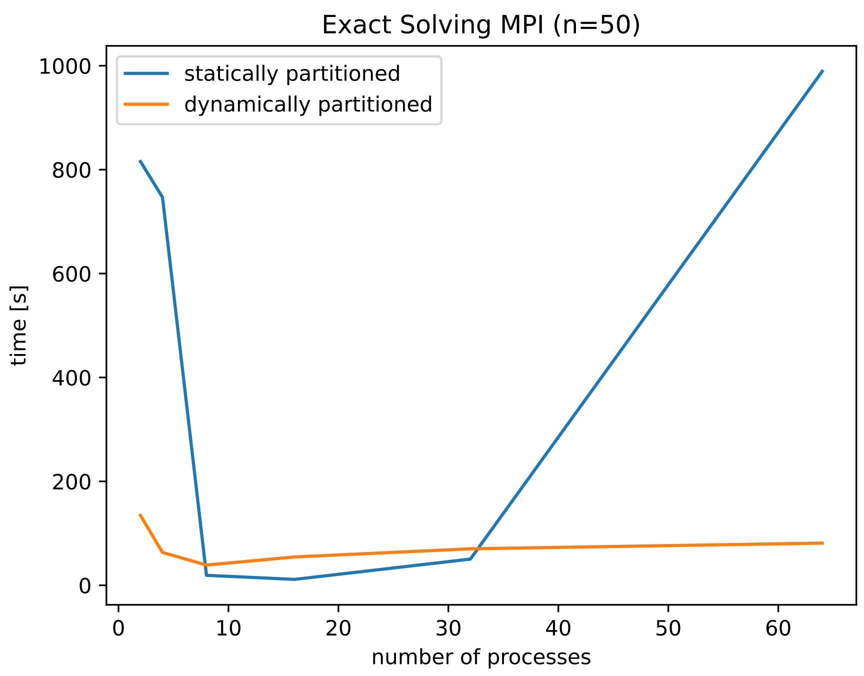


Figure 9: The comparison of the statically and dynamically allocated algorithm for different numbers of workers.

One can see that, averaged over all worker sizes, the dynamically partitioned algorithm performed better. This shows that, although more bits are sent, the more efficient allocation is worth the communication overhead. Furthermore, it is expected that the dynamically distributed algorithm performs comparatively better with more vertices, as more vertices increase the likelihood of an unfair static partitioning.

The optimal performance was recorded with the statically partitioned algorithm with 16 workers. Beyond 16 workers, the single coordinator becomes overwhelmed, resulting in longer waiting times for each worker to report their newly solved prefix.

Overall, it can be concluded that for smaller problems, the statically partitioned MPI algorithm should be used, while for large graphs and more workers, the dynamically partitioned algorithm is preferred.

4.3.3 MPI Analysis

The MPI analysis will be split into the statically partitioned and dynamically partitioned algorithm:

Statically partitioned Let m be the number of workers, n be the number of vertices in the graph, and k the length of the prefix. Thus, including prefixes that do not form a proper subtour, we have

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-k) = \prod_{i=n-k}^{n} i$$

prefixes. No communication is required for the prefix division. After each prefix, the worker sends 128 bits[23] to the coordinator and receives the global maximum back. Thus, excluding MPI overhead, a total of $256 \cdot \prod_{i=n-k}^{n} i$ bits of data are sent in the actual computation. In the wrap-up, two broadcasts to $m-1$ nodes are done. But since those do not scale with n, they can be ignored in the overall complexity.

Dynamically partitioned Let m be the number of workers and n be the number of vertices. The prefix length is fixed to 3. Thus, we have $n \cdot (n-1) \cdot (n-2) := n_p$ prefixes.

Each prefix gets assigned to a worker once, together with the current lowest minimum, resulting in $3 \cdot 64 + 64 = 256$ bits of data per message. For each prefix message to be sent, they have to be requested first. A request implicitly sends its rank and the current minimal path, thus $2 \cdot 64 = 128$ bits. Therefore, for the actual computation, a total of $n_p \cdot (128 + 256)$ bits are sent. Like with the statically partitioned solver, the wrap-up cost is independent of the graph size, and can therefore be ignored asymptotically.

4.4 Nearest Neighbour benchmarks

4.4.1 Problem Size Scaling

To compare the single-node sequential algorithm to the multithreaded version, starting at $n = 100$, the graph size was increased in steps of 100 up to 3000. The results are as follows:

[23]Assuming normal memory alignment.

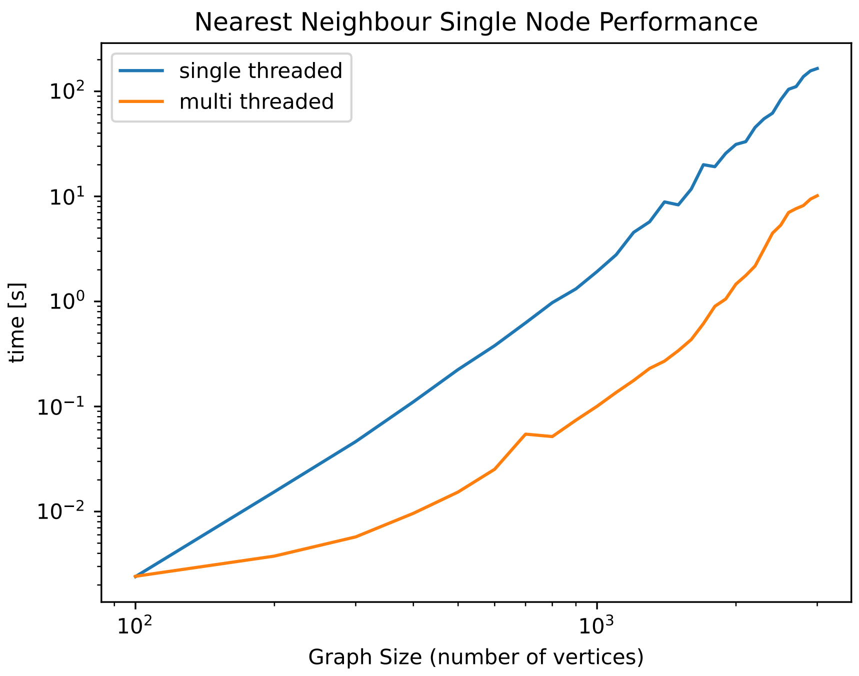


Figure 10: Problem Size scaling of the Nearest Neighbour algorithm

As one can see, in the beginning, the multithreading overhead keeps the problems performing more equally. Furthermore, while the multithreading is properly used, both algorithms keep the same asymptotic complexity. Overall, the nearest neighbour algorithm greatly benefits from multithreading. This was expected, as no inter-thread communication is required for computing the 1-nearest-neighbours. Furthermore, the minimum reduction at the end is single threaded in both versions, resulting in no overhead in the wrap-up.

4.4.2 Strong Scaling

For the MPI analysis, a strong scaling benchmark was used, with a fixed graph size of $n = 3000$.

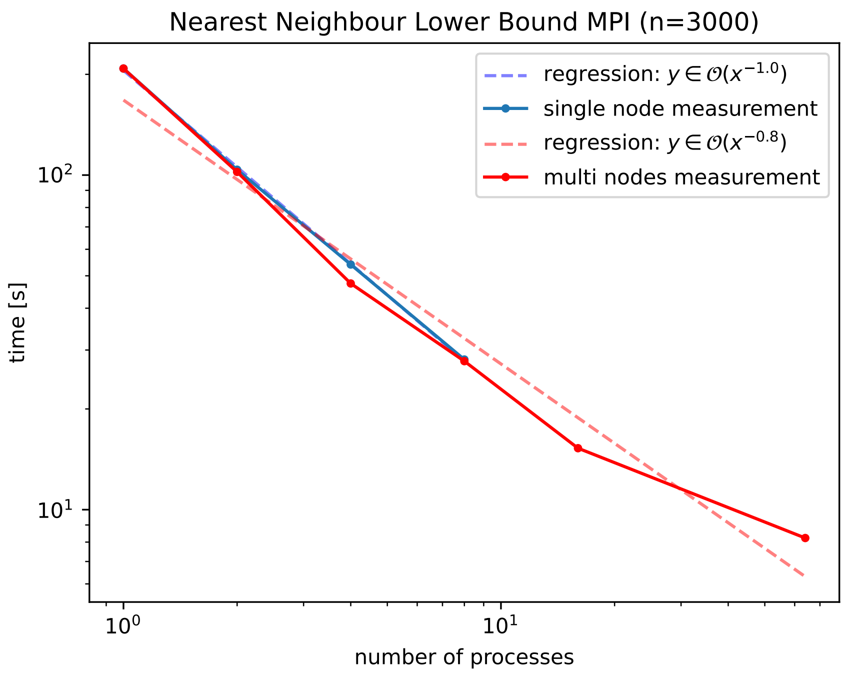


Figure 11: Strong scaling of the Nearest Neighbour algorithm

As one can see, the algorithm scales well with the amount of nodes. Analogously to the multithreading version, this was expected, as no inter-node communication is required for computing the single 1-nearest-neighbours. Furthermore, since the work partitioning is done locally on each node, no communication is needed for that either. Thus, with only two collective communications at all, MPI provides very little overhead.

4.4.3 MPI Analysis

Let m be the number of workers and n be the number of vertices.

Since the actual computation does not require any communication, only the wrap-up sends any data at all, by first `ALL_REDUCE` the best cost. After that, the best worker broadcasts the whole path. As the implementation of reductions is MPI-dependent, it can only be assumed that an `ALL_REDUCE` at most sends m^2 messages. Since it only sends the current best cost and its rank, it doesn't scale with the number of vertices and can therefore be ignored.

The broadcast at the end probably uses a tree structure internally, thus resulting in $n \cdot 64 + 64$ bits being sent in $\log(m)$ time steps. Therefore, the communication scales linearly in the number of graph vertices.

4.5 Christofides benchmarks

4.5.1 Problem Size Scaling

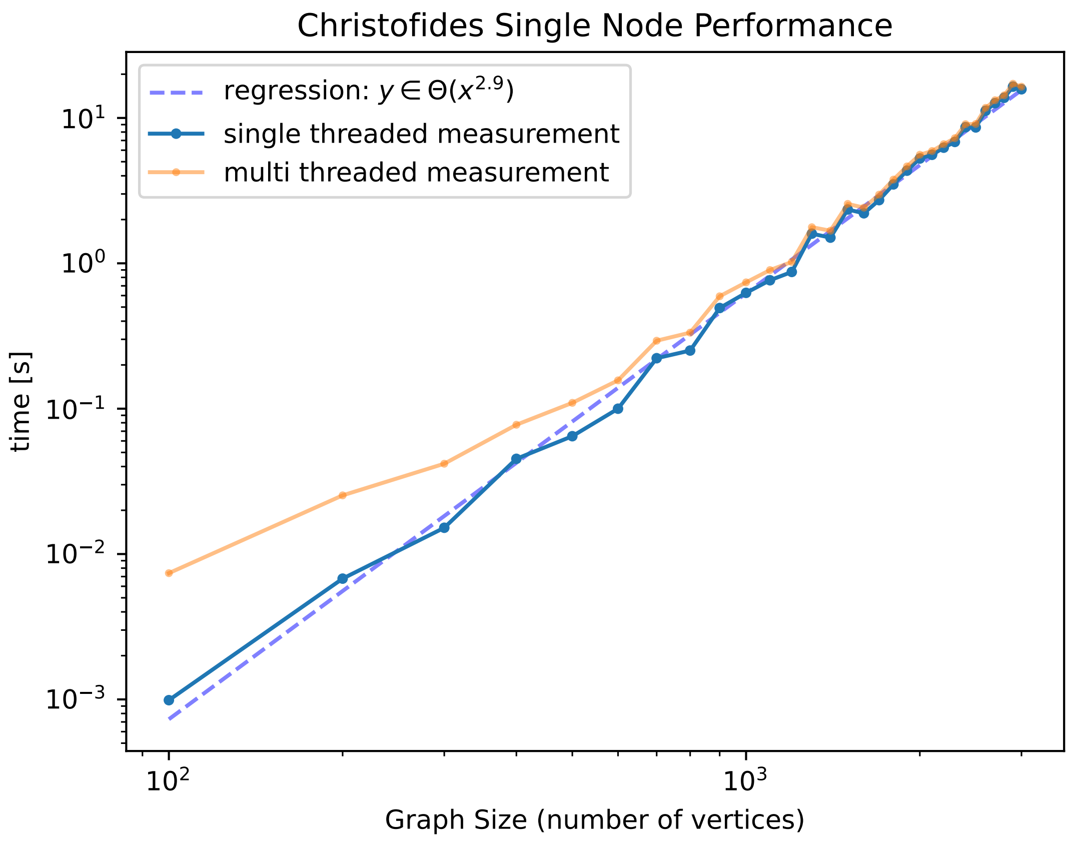


Figure 12: Problem Size scaling of the Christofides algorithm

To compare the performance between the single-threaded implementation, and the rayon-based implementation of the 1-tree lower bound, a problem size scaling analysis is used.

In Figure 12, one can see that the single-threaded implementation has roughly cubic time complexity, w.r.t. the number of nodes in the input graph.

For the tested graph sizes, the multithreaded implementation has no performance advantage, even the opposite is the case. The rayon-based implementation has a worse runtime behaviour on all tested graphs. Note, that the relative penalty diminishes for larger graphs. For the reasoning, see the analogous problem in 4.3.1.

4.5.2 Strong Scaling

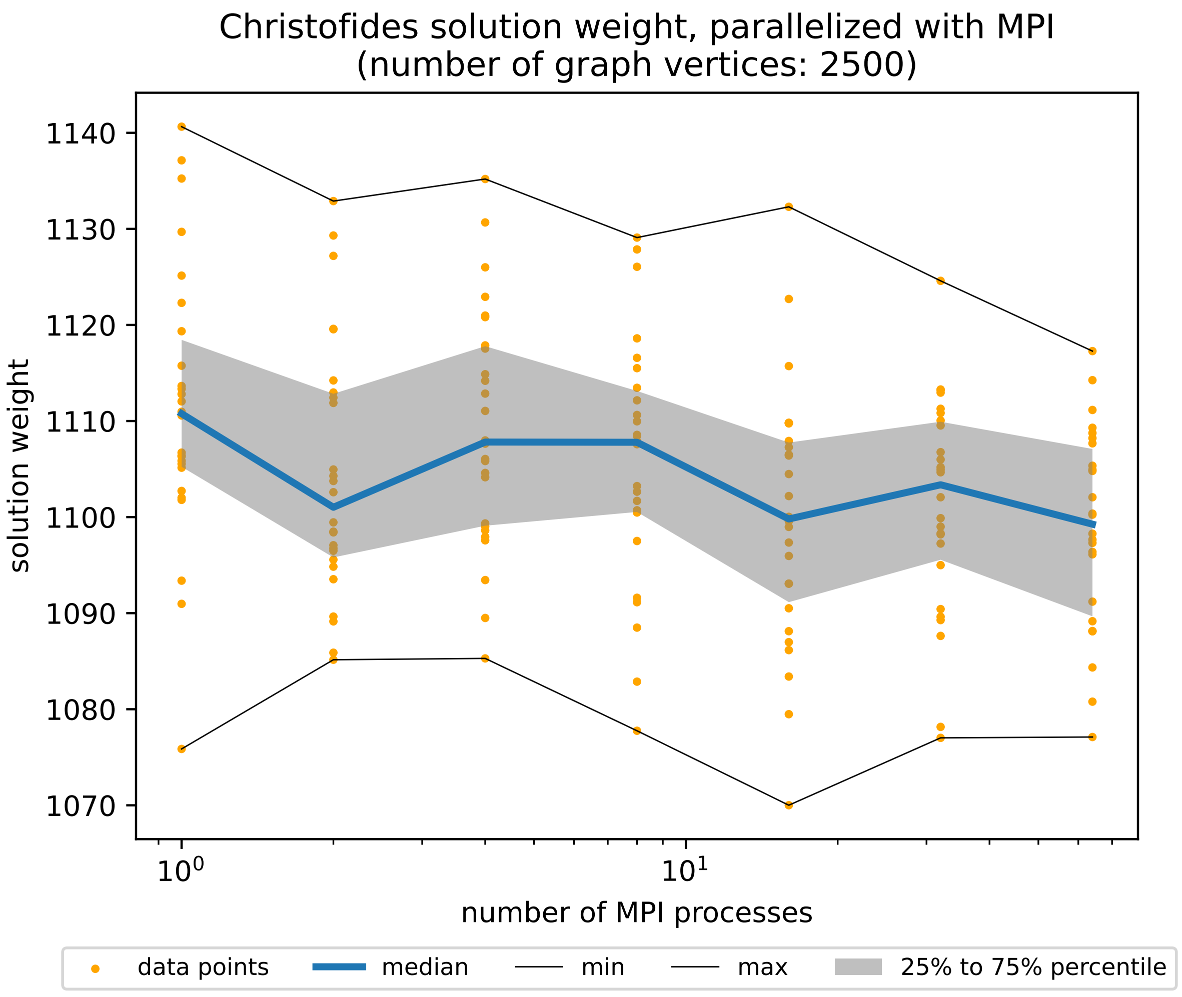


Figure 13: Strong scaling of the Christofides algorithm

To assess the performance of the MPI-based implementation of the Christofides algorithm, a strong scaling analysis has been done, see Figure 13.

Note, that the MPI implementation of the Christofides algorithm does not use the parallelism to compute the result quicker, but rather it uses the parallelism to compute a more precise result. Hence, the format of the analysis differs from all the other ones.

Especially noticeable is, that the upper outliers get significantly smaller when using more processes, whereas the outliers to the bottom do not experience any more improvement. This may be caused by worse initial solution guesses having greater potential for optimization, and with more parallelism it is more likely to find a path for optimization. In contrast, good initial guesses have less potential for optimization, whereby added parallelism cannot help in that case.

4.5.3 MPI Analysis

For the algorithmic description look at the parallelization paragraph in section 2.4.2.

Let $p \geq 2$ be the number of MPI processes. Let $G = (V, E)$ be the input graph. Then, during the execution of the MPI variant of the Christofides algorithm, $p-1$ tagged messages containing an `f64` value will be sent, as well as $p-1$ tagged messages containing a vector of $|V|$ many `usize` values. All of these messages will be received by the root process. This will take the root process $\mathcal{O}(p \cdot |V|)$ time, excluding time spent on synchronization, etc.

4.6 1-tree Lower Bound

4.6.1 Problem Size Scaling

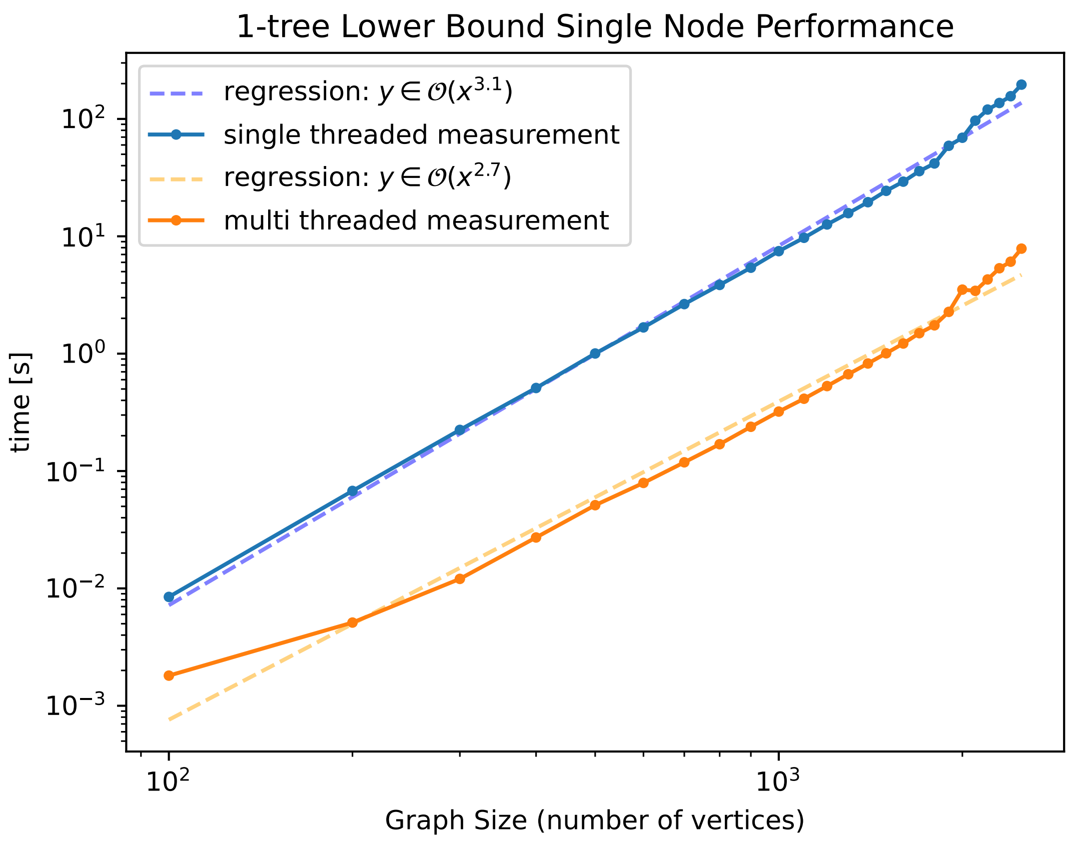


Figure 14: problem size scaling of the 1-tree lower bound

To compare the performance between the single-threaded implementation, and the rayon-based implementation of the 1-tree lower bound, a problem size scaling analysis comes in handy.

As seen in Fig. 14, both variants have roughly cubic time complexity, with the multi-threaded variant being faster by a constant factor. This is an expected result, as the 1-tree lower bound of a graph $G = (V, E)$ essentially involves computing $|V|$ many MSTs over $|V| - 1$ vertices. As can be seen in section 4.7, for this implementation, the sequential computation of an MST has quadratic complexity, thus the cubic complexity of the 1-tree lower bound follows easily. The parallel implementation having a similar complexity, but being faster by a constant factor is also to be expected since a constant number of MST computations is done in parallel, and other than that, nothing else is different from the sequential implementation.

4.6.2 Strong Scaling

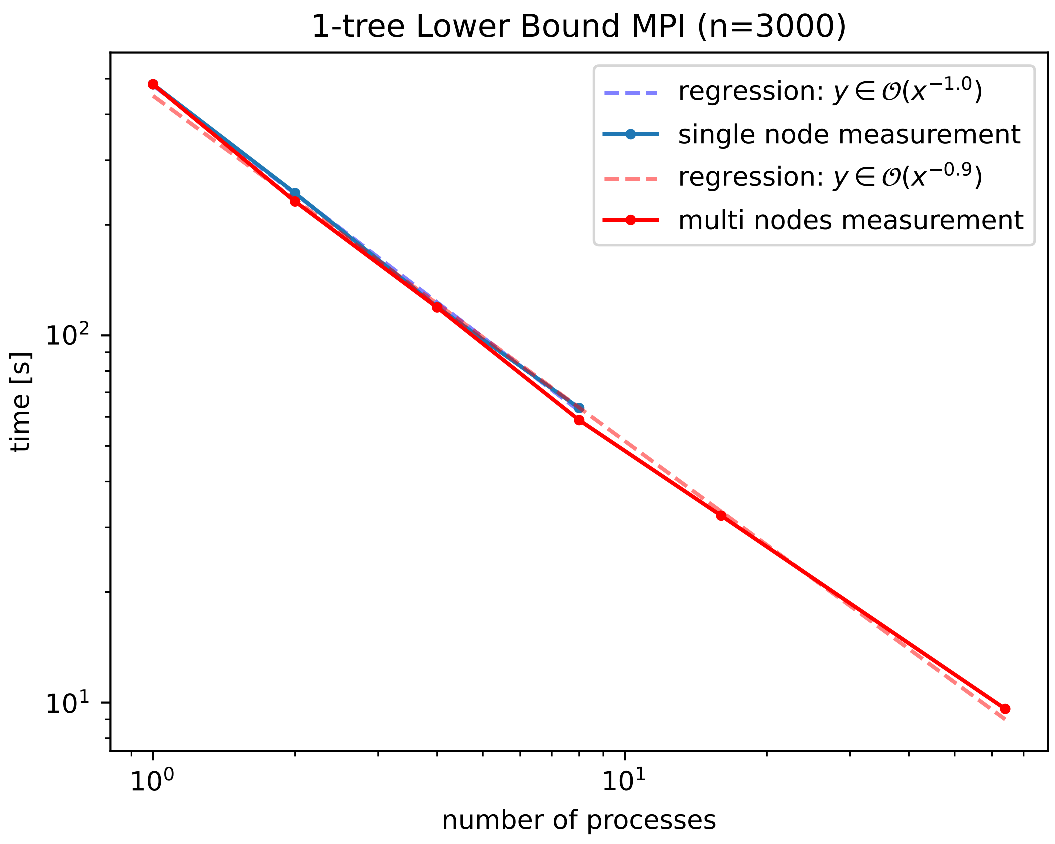


Figure 15: Strong scaling of the 1-tree lower bound

To analyze, how the MPI implementation of the 1-tree lower bound performs, a strong scaling analysis is applied.

As seen in Fig. 15, the MPI implementation's performance is roughly inversely proportional to the number of MPI processes, both when executed on only one host machine, and when executed on multiple machines in a cluster. This indicates, that the overhead of the MPI runtime is negligible.

4.6.3 MPI Analysis

For the algorithmic description look at section 2.5.2.

Let $p \geq 2$ be the number of MPI processes. Let $G = (V, E)$ be the input graph. During the execution of the MPI variant of the 1-tree lower bound, there is one MPI communication happening: After all processes have computed a lower bound, the maximum over all local lower bounds is collected at the root process using an MPI reduce operation. This operation theoretically only needs $\mathcal{O}(\log p)$ time, and $\mathcal{O}(p)$ many messages, by using a tree-like communication structure. Practically, the performance of MPI reduction operations is dependent on the concrete MPI implementation [14, section 5.9], but the fact that the operation has been the subject of research for many years [43] suggests, that the established MPI implementations optimize the reductions.

Note, that the cost of the MPI operations is independent of the graph size, it only depends on the number of available processes.

4.7 MST lower bound

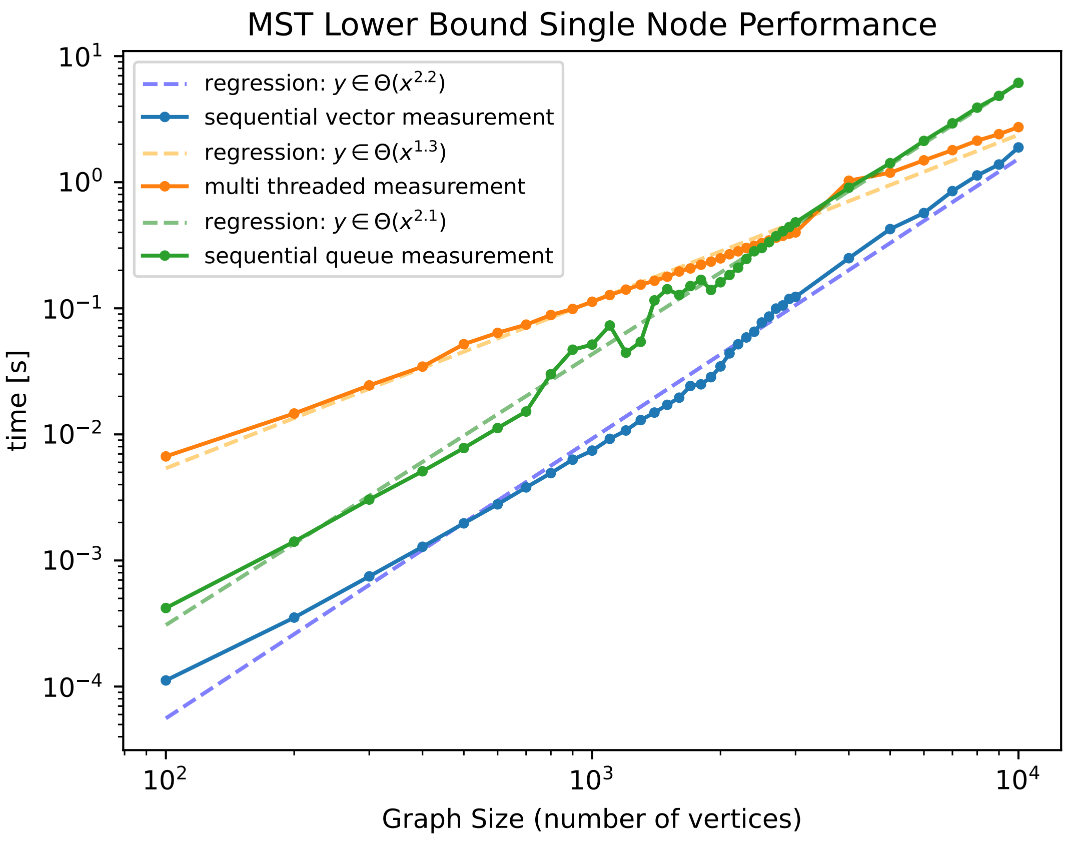


Figure 16: problem size scaling of the MST lower bound

The MST calculation has no MPI implementation, so only a problem size scaling analysis is performed.

There are three different implementations of Prim's algorithm provided: a sequential, and a multi-threaded variant using linear search on vectors (see also 2.1), and a sequential variant using a priority queue[24]. As the reader can see in Figure 16, the sequential vector-based algorithm performs best, for all tested inputs, and has roughly quadratic runtime behaviour, w.r.t. the number of nodes in the input graph.

Interestingly, the priority queue-based implementation is neither quicker nor does it show slower growth, than the sequential vector-based implementation. This may be explained with a sub-par third-party implementation of the priority queue data structure, or with insufficient input size. Note, that the largest tested graph (10,000 vertices) takes 4.6 GiB of disk space in TSPLIB-XML format.

The multithreaded variant does not outperform the sequential vector-based implementation, though for graphs a little larger than 10,000 vertices it probably would have. It

[24]The priority queue is provided by the `priority-queue` crate [44]

did however scale nearly linearly w.r.t. number of vertices, even though asymptotically, its runtime behaviour is equivalent to the sequential vector-based implementation. This may be, because the smaller the graphs are, the more the overhead of multithreading dominates over the benefit of it.

5 Challenges and Future Work

5.1 Challenges

Although all goals were met and all algorithms successfully implemented and parallelized, a few problems arose in development.

Inperformant initial data structures: In walky, graphs are implemented using adjacency matrices instead of adjacency lists. Initially, this was implemented using a nested `Vec<>`. This turned out to be very inefficient for multiple reasons:

- **Capacity management**: Since vectors are dynamically sized, iteratively inserting elements can result in multiple reallocations with larger capacity.
- **Runtime bounds checking**: Since their size is not known at compile time, vectors require a lot of bounds checks, which creates more branching and overall instructions. See the bounds check cookbook [45] for more information on how to best avoid bounds checking.
- **Memory locality**: Since vectors are heap allocated, the inner vectors in a `Vec<Vec<T>>` struct can be located at significantly different memory distances from one another. This results in worse data cache utilization and memory prefetching.

Beyond that, it was an overall very naïve implementation. In the current version, we use the nalgebra crate. It is highly optimized and uses a lot of advanced Rust performance techniques such as leveraging procedural macros for more compile time utilization, a custom allocator, and SIMD instructions as well as leveraging the state-of-the-art literature.

Insufficient MPI Language Support: Rust already has great first-class MPI support using `rsmpi` [35]. Unfortunately, as already described above, this support does not extend to MPI benchmarking, as the Rust ecosystem is not yet integrated.

5.2 Future Work

While the goals for this practical were archived, a lot of possible future work is still to be done. Here are a few of the possible next steps:

- While the exact solving algorithm is already highly optimized from a performance engineering perspective, it internally still uses the most naïve algorithm resulting in a theoretical $O(n!)$ worst-case performance. This could be improved by using other algorithms common in literature such as the classic $\Theta(2^n n^2)$ Held-Karp algorithm [6].
- Similarly, it is desirable to implement a better algorithm to calculate minimum-weight maximal matchings. The implemented algorithm is randomized and very naïve, even though the problem is known to be solvable in polynomial time, e.g. by the blossom algorithm [46], of which implementations exist [47].
- Analogously, many other useful approximation techniques could be implemented. Some of them include simulated annealing [48], the Lin-Kernighan heuristic [49], or an ant colony optimization approach [50].

6 Conclusion

The Travelling Salesman Problem (TSP) is one of the most well-studied problems in computer science with many real-world applications. In order to solve these problems, walky, a new Rust-based TSP solver, was created. Walky supports exact solving using highly optimized sequential and multi-threading algorithms as well as distributed, MPI-based parallelization. Furthermore, it supports two different approximation algorithms; the simple, easy-to-implement Nearest Neighbour algorithm as well as the sophisticated Christofides algorithm producing a tight upper bound. Additionally, it supports a sequential, multi-threaded, and distributed lower-bound calculation using two different lower-bound algorithms. Note that, instead of just being a prototype, walky is fully documented, well-tested, and officially published as a Rust crate, allowing real-world usage for any TSPLIB-XML formatted problem.

The benchmarks showed, that pruning vastly increased the performance and thus viability of exact solving. The usage of dynamic space partitioning improved the scaling of the distributed memory algorithm. The nearest neighbour approximation, due to its minimal communication, scaled nearly optimal. They also showed, that the 1-tree lower-bound greatly benefits from parallelism. Christofides algorithm in its randomized implementation is a very quick approximation to the TSP, the randomized approximation can be made more reliable by utilizing parallelism. For the MST computation, the graphs tested in this setting were too small to benefit from parallelism, though the benchmarks indicated that for larger graphs a parallel implementation of Prim's algorithm would outperform its sequential counterpart.

Lastly, walky also proved that Rust is viable for distributed, MPI-based computation and implementing optimized, efficient algorithms and data structures. It shows that Rust is a sufficient programming language for developing HPC applications.

References

[1] Yin Song et al. *Solving the Traveling Salesperson Problem with deep reinforcement learning on Amazon SageMaker | AWS Open Source Blog*. en. Sept. 2021. URL: https://aws.amazon.com/de/blogs/opensource/solving-the-traveling-salesperson-problem-with-deep-reinforcement-learning-on-amazon-sagemaker/ (visited on 05/15/2023).

[2] Reducible. *The Traveling Salesman Problem: When Good Enough Beats Perfect*. en. July 2022. URL: https://www.youtube.com/watch?v=GiDsjIBOVoA (visited on 10/02/2023).

[3] *TSP Applications*. en. 2007. URL: https://www.math.uwaterloo.ca/tsp/apps/index.html (visited on 10/05/2023).

[4] Thomas H. Cormen et al. "Introduction to algorithms". eng. In: Fourth edition. Type: Band. Cambridge, Massachusetts: The MIT Press, 2022. ISBN: 9780262046305 | 978-0-262-04630-5 | 026204630X | 0-262-04630-X. URL: http://www.gbv.de/dms/weimar/toc/1767218192_toc.pdf.

[5] Gerhard Reinelt. *TSPLIB*. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (visited on 10/10/2023).

[6] Michael Held and Richard M. Karp. "A Dynamic Programming Approach to Sequencing Problems". In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), pp. 196–210. ISSN: 0368-4245. URL: https://www.jstor.org/stable/2098806 (visited on 10/06/2023).

[7] Jon Bentley. *Lecture 21: Tuning a TSP Algorithm \textbar Performance Engineering of Software Systems*. 2018. URL: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/resources/lecture-21-tuning-a-tsp-algorithm/ (visited on 10/06/2023).

[8] Nayuki. *Next lexicographical permutation algorithm*. June 2018. URL: https://www.nayuki.io/page/next-lexicographical-permutation-algorithm (visited on 06/28/2023).

[9] *rustc-hash*. Oct. 2023. URL: https://github.com/rust-lang/rustc-hash (visited on 10/06/2023).

[10] Nicos Christofides. "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem". en. In: *Operations Research Forum* 3.1 (Feb. 1976), p. 20. ISSN: 2662-2556. DOI: 10.1007/s43069-021-00101-z. URL: https://link.springer.com/10.1007/s43069-021-00101-z (visited on 06/26/2023).

[11] Eric W. Weisstein. *Matching*. en. Text. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/Matching.html (visited on 10/10/2023).

[12] Carl Hierholzer. "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren." de. In: Mathematische Analen.VI. Band. 1. Heft (1873), pp. 30–32. URL: https://www.digizeitschriften.de/id/235181684_0006%7Clog12?tify=%7B%22pages%22%3A%5B36%5D%2C%22pan%22%3A%7B%22x%22%3A0.53%2C%22y%22%3A0.786%7D%2C%22view%22%3A%22export%22%2C%22zoom%22%3A0.37%7D (visited on 10/02/2023).

[13] "Algorithms for Eulerian Trails and Cycle Decompositions, Maze Search Algorithms". In: *Eulerian Graphs and Related Topics*. Annals of Discrete Mathematics 50 (1991). Ed. by Herbert Fleischner. ISSN: 0167-5060, pp. X.1–X.34. ISSN: 0167-5060. DOI: https://doi.org/10.1016/S0167-5060(08)70158-4. URL: https://www.sciencedirect.com/science/article/pii/S0167506008701584.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. en. June 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[15] Michael Held and Richard M. Karp. "The Traveling-Salesman Problem and Minimum Spanning Trees". en. In: *Operations Research* 18.6 (Dec. 1970), pp. 1138–1162. ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.18.6.1138. URL: https://pubsonline.informs.org/doi/10.1287/opre.18.6.1138 (visited on 05/09/2023).

[16] *Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 08/15/2023).

[17] The Servo Project Developers. *Servo, the parallel browser engine*. URL: https://servo.org/ (visited on 08/15/2023).

[18] Evgenii Stepanov. *Detecting Memory Corruption Bugs With HWASan*. Feb. 2020. URL: https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html (visited on 08/15/2023).

[19] Stoep. *Queue the Hardening Enhancements*. May 2019. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (visited on 08/15/2023).

[20] Paul Kehrer. *Memory Unsafety in Apple's Operating Systems*. July 2019. URL: https://langui.sh/2019/07/23/apple-memory-safety/ (visited on 08/15/2023).

[21] *Memory safety*. URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 08/15/2023).

[22] Gavin Thomas. *A proactive approach to more secure code \textbar MSRC Blog \textbar Microsoft Security Response Center*. July 2019. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on 08/15/2023).

[23] Diane Hosfelt. *Implications of Rewriting a Browser Component in Rust – Mozilla Hacks - the Web developer blog*. Feb. 2019. URL: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust (visited on 08/15/2023).

[24] Geoffrey Thomas [@geofft]. *Some unofficial @LazyFishBarrel stats from @alex_gaynor and myself: 65% of CVEs behind the last six months of Ubuntu security updates to the Linux kernel have been memory unsafety*. May 2019. URL: https://twitter.com/geofft/status/1132739184060489729 (visited on 08/15/2023).

[25] Jyun-Yan You. *bindgen*. Aug. 2023. URL: https://github.com/rust-lang/rust-bindgen (visited on 08/15/2023).

[26] PyO3 Project and Contributors. *PyO3*. Aug. 2023. URL: https://github.com/PyO3/pyo3 (visited on 08/15/2023).

[27] *Stack Overflow Developer Survey 2023*. 2023. URL: https://survey.stackoverflow.co/2023 (visited on 08/28/2023).

[28] Matt Asay. *Why AWS loves Rust, and how we'd like to help \textbar AWS Open Source Blog.* Nov. 2020. URL: `https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/` (visited on 08/15/2023).

[29] *Welcome to Comprehensive Rust - Comprehensive Rust.* URL: `https://google.github.io/comprehensive-rust/` (visited on 08/15/2023).

[30] Garcia Garcia. *Programming languages endorsed for server-side use at Meta.* July 2022. URL: `https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/` (visited on 08/15/2023).

[31] jirehl. *Microsoft Azure CTO Wants to Replace C and C++ With Rust \textbar The Software Report.* Oct. 2022. URL: `https://www.thesoftwarereport.com/microsoft-azure-cto-wants-to-replace-c-and-c-with-rust/` (visited on 08/15/2023).

[32] Thomas Claburn. *Linus Torvalds says Rust is coming to the Linux kernel.* June 2022. URL: `https://www.theregister.com/2022/06/23/linus_torvalds_rust_linux_kernel/` (visited on 08/15/2023).

[33] *rustup.rs - The Rust toolchain installer.* URL: `https://rustup.rs/` (visited on 10/06/2023).

[34] *Rayon.* Oct. 2023. URL: `https://github.com/rayon-rs/rayon` (visited on 10/06/2023).

[35] *MPI bindings for Rust.* Oct. 2023. URL: `https://github.com/rsmpi/rsmpi` (visited on 10/06/2023).

[36] *Nextest.* Oct. 2023. URL: `https://github.com/nextest-rs/nextest` (visited on 10/06/2023).

[37] Fillipe Goulart. *Python TSP Solver.* Oct. 2023. URL: `https://github.com/fillipe-gsm/python-tsp` (visited on 10/06/2023).

[38] *Vampir 10.3.* URL: `https://vampir.eu/` (visited on 10/10/2023).

[39] Andreas Knüpfer et al. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". en. In: *Tools for High Performance Computing 2011.* Ed. by Holger Brunst et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31475-9 978-3-642-31476-6. DOI: `10.1007/978-3-642-31476-6_7`. URL: `http://link.springer.com/10.1007/978-3-642-31476-6_7` (visited on 10/10/2023).

[40] Ronny Tschüter et al. *An LLVM Instrumentation Plug-in for Score-P.* en. Dec. 2017. DOI: `10.1145/3148173.3148187`. URL: `https://arxiv.org/abs/1712.01718v1` (visited on 10/10/2023).

[41] *Scalasca.* URL: `https://www.scalasca.org/` (visited on 10/10/2023).

[42] *TAU - Tuning and Analysis Utilities -.* URL: `https://www.cs.uoregon.edu/research/tau/home.php` (visited on 10/10/2023).

[43] Khalid Hasanov and Alexey Lastovetsky. “Hierarchical Optimization of MPI Reduce Algorithms”. en. In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Vol. 9251. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 21–34. ISBN: 978-3-319-21908-0 978-3-319-21909-7. DOI: `10.1007/978-3-319-21909-7_3`. URL: `https://hcl.ucd.ie/system/files/pact2015reduce.pdf` (visited on 10/10/2023).

[44] garro95. *PriorityQueue*. Feb. 2023. URL: `https://github.com/garro95/priority-queue`.

[45] Sergey "Shnatsel" Davidoff. *Recipes for avoiding bounds checks in Rust*. original-date: 2022-12-11T19:15:16Z. Oct. 2023. URL: `https://github.com/Shnatsel/bounds-check-cookbook` (visited on 10/10/2023).

[46] Vladimir Kolmogorov. “Blossom V: a new implementation of a minimum cost perfect matching algorithm”. en. In: *Math. Prog. Comp.* 1.1 (July 2009), pp. 43–67. ISSN: 1867-2949, 1867-2957. DOI: `10.1007/s12532-009-0002-8`. URL: `http://link.springer.com/10.1007/s12532-009-0002-8` (visited on 10/10/2023).

[47] Vladimir Kolmogorov. *Blossom V implementation*. URL: `https://pub.ista.ac.at/~vnk/software.html` (visited on 10/10/2023).

[48] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (May 1983). Publisher: American Association for the Advancement of Science, pp. 671–680. DOI: `10.1126/science.220.4598.671`. URL: `https://www.science.org/doi/10.1126/science.220.4598.671` (visited on 10/10/2023).

[49] S. Lin and B. W. Kernighan. “An Effective Heuristic Algorithm for the Traveling-Salesman Problem”. In: *Operations Research* 21.2 (Apr. 1973). Publisher: INFORMS, pp. 498–516. ISSN: 0030-364X. DOI: `10.1287/opre.21.2.498`. URL: `https://pubsonline.informs.org/doi/10.1287/opre.21.2.498` (visited on 10/10/2023).

[50] Shu-Chuan Chu, John F. Roddick, and Jeng-Shyang Pan. “Ant colony system with communication strategies”. In: *Information Sciences* 167.1 (Dec. 2004), pp. 63–76. ISSN: 0020-0255. DOI: `10.1016/j.ins.2003.10.013`. URL: `https://www.sciencedirect.com/science/article/pii/S0020025503004110` (visited on 10/10/2023).

A Work sharing

This report and the `walky` software were written by Lars Quentin and Johann Carl Meyer. The work was distributed as follows.

A.1 Lars Quentin

The following algorithms were researched, implemented, documented, and analysed by Lars Quentin:

- all iterations and versions of the exact solvers
- the nearest neighbour approximation

Furthermore, Lars Quentin worked on

- designing and implementing the `walky` CLI,
- maintaining good code quality by integrating a CI pipeline and doing manual work on the `walky` repository.
- Writing benchmarking scripts and test graph generation tools.

The following chapters were written by Lars Quentin:

- 2.2 Exact and Approximate Solving
- 2.3 Exact Solving
- 2.4.1 Nearest Neighbour
- 3 Implementation (excluding 3.3 Command Line Interfaces (CLI)
- 4.1 Cluster Setup
- 4.2 Vampir-based Analysis of Rust MPI Code
- 4.3 Exact Solving Benchmarks
- 4.4 Nearest Neighbour Benchmarks
- 5 Challenges and Future Work
- 6 Conclusion

All chapters were thoroughly reviewed by Johann Carl Meyer.

A.2 Johann Carl Meyer

The following algorithms were researched, implemented, documented, and analysed by Johann Carl Meyer:

- all MST implementations
- all Christofides implementations
- all 1-tree lower bound implementations
- the TSPLIB-XML parser

The following chapters were written by Johann Carl Meyer:

- 1. Introduction
- 2.1 Minimum Spanning Tree
- 2.4.2 Christofides Algorithm
- 2.5 Lower Bound
- 3.3 Command Line Interface (CLI)
- 4.5 Christofides Benchmarks
- 4.6 1-tree Lower Bound
- 4.7 MST lower bound

All chapters were thoroughly reviewed by Lars Quentin.

B Code samples

B.1 walky subcommands

The `walky exact` subcommand is used to call an exact solver.

```
$ walky exact --help
Find the exact best solution to a given TSP instance

Usage: walky exact [OPTIONS] <ALGORITHM> <INPUT_FILE>

Arguments:
  <ALGORITHM>
          The Algorithm to use

          Possible values:
          - v0: Testing each possible (n!) solutions
          - v1: Fixating the first Element, so testing ((n-1)!) solutions
          - v2: Recursive Enumeration; Keep the partial sums cached
          - v3: Stop if partial sum is worse than previous best
          - v4: Stop if partial sum + greedy nearest neighbour graph is bigger than
          ↪  current optimum
          - v5: As V5, but use an MST instead of NN-graph as a tighter bound
          - v6: Cache MST distance once computed

  <INPUT_FILE>
          Path to the TSPLIB-XML file

Options:
  -p, --parallelism <PARALLELISM>
          Whether to solve it sequential or parallel

          [default: single-threaded]

          Possible values:
          - single-threaded: Run in a single threaded
          - multi-threaded:  Run in multiple threads on a single node
          - mpi:             Run on multiple nodes. Requires MPI

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

The `walky approx` subcommand is used to call an approximate solver.

```
$ walky approx --help
Find an approximate solution to a given TSP instance

Usage: walky approx [OPTIONS] <ALGORITHM> <INPUT_FILE>

Arguments:
  <ALGORITHM>
          The Algorithm to use

          Possible values:
          - nearest-neighbour: Starting at each vertex, always visiting the lowest
          ↪  possible next vertex
          - christofides:      The Christofides(-Serdyukov) algorithm

  <INPUT_FILE>
          Path to the TSPLIB-XML file

Options:
  -p, --parallelism <PARALLELISM>
          Whether to solve it sequential or parallel

          [default: single-threaded]

          Possible values:
          - single-threaded: Run in a single threaded
          - multi-threaded:  Run in multiple threads on a single node
          - mpi:             Run on multiple nodes. Requires MPI

  -l, --lower-bound <LOWER_BOUND>
          Whether to also compute a lower_bound. Optional

          Possible values:
```

```
          - one-tree:  The one tree lower bound
          - mst:       The MST lower bound
          - mst-queue: The MST lower bound, computed with prims algorithm using a
          ↪  priority queue

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

The `walky lower-bound` subcommand is used to compute a lower bound of the input graph.

```
$ walky lower-bound --help
Compute a lower bound cost of a TSP instance

Usage: walky lower-bound [OPTIONS] <ALGORITHM> <INPUT_FILE>

Arguments:
  <ALGORITHM>
          The Algorithm to use

          Possible values:
          - one-tree:  The one tree lower bound
          - mst:       The MST lower bound
          - mst-queue: The MST lower bound, computed with prims algorithm using a
          ↪  priority queue

  <INPUT_FILE>
          Path to the TSPLIB-XML file

Options:
  -p, --parallelism <PARALLELISM>
          Whether to solve it sequential or parallel

          [default: single-threaded]

          Possible values:
          - single-threaded: Run in a single threaded
          - multi-threaded:  Run in multiple threads on a single node
          - mpi:             Run on multiple nodes. Requires MPI

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

C Tabular Results Exact Solving

n	v_0	v_1	v_2	v_3	v_4	v_5	v_6	MT
3	0.123	0.123	0.123	0.123	0.123	0.123	0.123	0.117
4	0.125	0.125	0.125	0.125	0.124	0.124	0.125	0.117
5	0.125	0.125	0.125	0.125	0.125	0.125	0.124	0.116
6	0.125	0.125	0.125	0.125	0.125	0.125	0.125	0.116
7	0.124	0.125	0.124	0.125	0.125	0.125	0.124	0.116
8	0.124	0.124	0.126	0.124	0.125	0.124	0.124	0.116
9	0.125	0.123	0.182	0.133	0.127	0.123	0.123	0.116
10	0.172	0.126	0.784	0.241	0.173	0.123	0.123	0.116
11	0.714	0.176	7.995	0.387	0.131	0.123	0.123	0.117
12	7.709	0.758	101.020	3.577	0.326	0.123	0.124	0.118
13	104.572	8.218	1422.181	12.063	0.264	0.123	0.123	0.117
14	1674.600	119.378		74.933	0.672	0.124	0.124	0.117
15		1754.863		17.168	0.124	0.124	0.123	0.117
16				284.757	0.145	0.124	0.124	0.117
17				1026.435	3.249	0.124	0.124	0.117
18					64.765	0.133	0.134	0.127
19					1.315	0.123	0.123	0.118
20					0.706	0.124	0.124	0.119
21					0.600	0.128	0.127	0.122
22					6.252	0.131	0.131	0.125
23					31.762	0.131	0.129	0.123
24					0.478	0.133	0.135	0.129
25					5.612	0.132	0.132	23.120
26					1.346	0.131	0.132	10.335
27					28.855	0.140	0.142	24.383
28					95.776	0.132	0.132	12.981
29					15.390	0.136	0.136	20.802
30					64.813	0.137	0.137	671.546
31					35.070	0.140	0.140	283.184
32					38.425	0.140	0.141	51.635
33					29.138	0.145	0.144	55.689
34					357.331	0.151	0.153	105.007
35					912.095	0.163	0.163	1054.830
36						0.153	0.152	87.845
37						0.189	0.193	277.350
38						1.810	1.535	
39						0.166	0.167	
40						0.206	0.207	
41						0.204	0.208	
42						0.201	0.205	
43						0.272	0.282	
44						0.200	0.202	
45						0.236	0.241	
46						0.305	0.315	