Experimental validation of distributed algorithm

Martin Rokyta
Open Informatics

May 2023
Supervisor: doc. Ing. Jiří Vokřínek, Ph.D.
I. Personal and study details

Student's name: Rokyta Martin
Personal ID number: 492043
Faculty / Institute: Faculty of Electrical Engineering
Department / Institute: Department of Computer Science
Study program: Open Informatics
Specialisation: Software

II. Bachelor's thesis details

Bachelor's thesis title in English: Experimental validation of distributed algorithm
Bachelor's thesis title in Czech: Experimentální validace škálování distribuovaných algoritmů
Guidelines:
- Analyze the features of the distributed computational platform for Raspberry-Pi cluster [1].
- Propose and implement an extension of the platform for validation of different algorithms deployment (single-CPU, multi-CPU, distributed experiments).
- Implement selected distributed coordination multi-agent algorithms [i.e. 2].
- Propose an evaluation metric to compare various features of implemented algorithms according to deployment strategy.
- Conduct extensive experimental evaluation of the algorithms on personal computer and Raspberry-Pi cluster.

Bibliography / sources:
[1] https://dspace.cvut.cz/handle/10467/100974

Name and workplace of bachelor's thesis supervisor:

doc. Ing. Jiří Vokřinek, Ph.D. Department of Computer Science FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: 23.02.2023 Deadline for bachelor thesis submission: 26.05.2023
Assignment valid until: 16.02.2025

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt Student's signature
I would like to thank Mr. Jiří Vokřínek for his patience, the chance to work on this topic, and the supervision provided. His insight, advice, and critique had proven to be essential both to my understanding of the problem, and while working on the project. I would also like to thank Mr. Roman Janků for his help and assistance with his Picocluster platform, and Mr. Michal Čap for providing the source code to COBRA and ADPP algorithms.

I declare that I have written this work only by myself and I have cited all sources I have used in the bibliography in accordance with the methodical instructions for observing the ethical principles in the preparation of a university thesis.

Prague, May 26, 2023
Abstrakt / Abstract

Distribuované systémy se staly neod- dělitelnou součástí mnoha moderních oborů. Jejich implementace se však výrazně liší, přičemž ty teoretické často využívají simulovaného distribuovaného systému běžícího na jednom počítači. Využívání takových simulací však nemusí vykazovat stejné výsledky jako práce na reálném distribuované prostředí.

Tato bakalářská práce má za cíl přepracování dvou distribuovaných algoritmů, které využívají simulace distribuovaného systému na jednom stroji tak, aby byly schopné běhu na skutečně distribuovaném systému, a experimentálně porovnat a zhodnotit jejich výkon na obou těchto prostředcích.

Algoritmy vybrané pro tento úkol jsou Continuous Best-Response Approach algorithm (COBRA) a Asynchronous Decentralized Priority Planning (ADPP), které se zabývají koordinačním problémem pro více agentů.

Tyto algoritmy byly integrovány do platformy pro distribuované výpočty na klastru Raspberry Pi, která byla zároveň rozšířena o možnost běhu na jakémkoliv prostředí.

Tato práce přináší důkladné zhodnocení škálování daných algoritmů na různých prostředcích a poukazuje na výhody a nevýhody různých metod testování distribuovaných algoritmů.

Klíčová slova: distribuované systémy, distribuované algoritmy, simulování distribuovaných systémů, vyhledávání cesty více agentů, škálování algoritmů

Překlad titulu: Experimentální validace škálování distribuovaných algoritmů

Distributed algorithms have become an essential feature of many modern problems and fields. Their implementations can vary significantly, with the theoretical ones often relying on a simulated distributed system running on a single machine. Evaluating such simulations may not accurately reflect the algorithm’s performance in real-world settings.

This bachelor thesis refactors two distributed algorithms, implemented as a simulation for one machine, to be able to run on a truly distributed system and to evaluate and compare their performance on both environments experimentally.

The algorithms selected for this task are the Continuous Best-Response Approach algorithm (COBRA) and the Asynchronous Decentralized Priority Planning (ADPP), examples of multi-agent distributed coordination problem solvers.

The algorithms were integrated into a platform for distributed computing on Raspberry Pi cluster, which was extended to run on any environment.

The thesis provides an extensive evaluation of the scaling of these algorithms on various deployment environments and discusses the advantages and disadvantages of different methods of distributed algorithm testing.

Keywords: distributed systems, distributed algorithms, simulation of distributed algorithms, multi-agent pathfinding, algorithm scaling
## Contents

1 Introduction  
1.1 Assignment  

2 Theoretical background  
2.1 Picocluster platform for distributed algorithms  
2.1.1 Raspberry Pi platform  
2.2 Algorithms  
2.2.1 ADPP algorithm  
2.2.2 COBRA algorithm  

3 Algorithm integrations and platform enhancements  
3.1 Proposal of solution  
3.2 Enhancements of the Picocluster platform  
3.2.1 Modifiable environment setting  
3.2.2 UDP communication  
3.3 Integrating the algorithms  
3.3.1 Shared features  
3.3.2 Integration of ADPP  
3.3.3 Integration of COBRA  

4 The experiment  
4.1 Preparing the experiment  
4.2 Experiment measuring  
4.2.1 Preparations  
4.2.2 Environments  
4.2.3 Experiment instances  

5 Results and Analysis  
5.1 ADPP results  
5.1.1 Time to convergence  
5.1.2 Trajectory planning  
5.1.3 Planning time  
5.1.4 Runtime without planning  
5.1.5 Normalized Distributed method  
5.2 COBRA results  
5.2.1 Runtime  
5.2.2 Planning  
5.2.3 Waiting for token  
5.2.4 Base task duration and prolongation  
5.3 Analysis and summary  

6 Conclusion  

References  

A Project structure and usage  
A.1 Version manager  
A.2 Maven structure  
A.3 Running the algorithms and experiment  
A.3.1 Starting the project and algorithms  
A.3.2 Starting the experiment  

B Attachments
Tables / Figures

3.1 ADPP parameters.................. 15
3.2 COBRA parameters.............. 17
4.1 Comparison of deployment environment hardware .......... 24
4.2 Experiment instances .......... 24

2.1 Showase of a GUI of the server module of Picocluster platform ................................ 3
2.2 Blinkt! module ...................... 3
2.3 48-node Pico cluster connect-ed to PC ......................... 4
2.4 A scheme of network connection of Picocluster ................. 4
2.5 Well-formed infrastructure ...... 5
2.6 Show .................................. 6
2.7 Showcase of ADPP runtime visualization ....................... 6
2.8 Showcase of COBRA runtime visualization .................... 7
3.1 Showcase of available param-eters for COBRA in the GUI .. 11
3.2 Failed ADPP algorithm caused by insufficient nodes ... 12
3.3 Showcase of GUI logs of ADPP runtime ....................... 13
3.4 Showcase of ADPP’s visual-ization over Blinkt! .............. 16
3.5 COBRA token passing in GUI logs ......................... 18
3.6 Showcase of COBRA Blinkt! visualization .................... 20
4.1 ADPP experiment GUI ............... 21
4.2 Map of Empty hall environ-ment with docks.................. 23
4.3 Map of Warehouse environ-ment with docks................. 23
4.4 Map of Ubremen environ-ment with docks.................... 23
This project emerged from a discussion about simulating distributed algorithms in a non-distributed environment. Such simulation may achieve excellent results regarding solution correctness, but it can only partially substitute an actual distributed environment. Even though additional simulation settings may compensate for the lack of communication issues and networking delays, the hardware restrictions in limited thread count and thread management will never let the individual distributed subproblems be genuinely independent. The result difference can be enormous, and even algorithms that behave deterministically could end up unexpectedly.

This bachelor’s thesis aims to evaluate specific distributed algorithms experimentally and discuss their results from various simulated environments and truly distributed ones.

To achieve this, I will introduce two selected distributed algorithms and integrate them into a platform created for running and managing tasks across a distributed environment. Extensive research and modifications will also be conducted in order to allow the platform to operate on a single CPU apart from the distributed network to collect data from simulated conditions.

Lastly, I will present various metrics to test the scaling of the algorithms in all deployment settings and experimentally evaluate and discuss the results collected.

1.1 Assignment

1. Analyze the features of the distributed computational platform for Raspberry-Pi cluster
2. Propose and implement an extension of the platform for validation of different algorithms deployment (single-CPU, multi-CPU, distributed experiments)
3. Implement selected distributed coordination multi-agent algorithms
4. Propose an evaluation metric to compare various features of implemented algorithms according to deployment strategy.
5. Conduct extensive experimental evaluation of the algorithms on personal computer and Raspberry-Pi cluster.
Chapter 2

Theoretical background

Distributed computing has seen a significant increase in utilization over the last few years. Multi-agent path planning, one of the possible usages of such algorithms, focuses on choosing optimal paths for multiple agents in complex environments while avoiding mutual collisions. The coordination problem was chosen as a typical example of a more advanced distributed algorithm with significant potential for practical usage and many possible solution approaches which can produce compelling results for the scaling evaluation.

Nevertheless, the implementation of such algorithms can vary significantly. Many modern algorithms are described only theoretically, and the potential implementations are often accomplished using a simulated distributed environment. Even though the difficulties of acquiring a truly distributed and well-behaving\textsuperscript{1} platform are an acceptable cause of these implementations, a question arises on how this simulation might influence the algorithm’s runtime and results.

The main difference between the simulations and the actual distributed system is resource sharing between the client processes. Not only do they share RAM, but the demands and possible racing for CPU threads might result in a process having to wait for resources taken by another client. Furthermore, truly distributed systems introduce a delay in communication.

2.1 Picocluster platform for distributed algorithms

The chosen platform for controlling distributed algorithms is the Application for distributed algorithms on Picocluster\textsuperscript{[1]}, developed by Roman Janků. This application is specifically designed to manage tasks and assignments within a distributed system using a Picocluster\textsuperscript{[2]} architecture. The project comprises three modules: the Server, the Node, and the Library. Each module can be independently packaged as a .jar executable, with the server and node jars fully executable.

- The Library module represents a shared codebase for the other two modules, containing essential code for communication, utility, and algorithms.
- The Node module provides the necessary code to run an executable application on individual clients within the distributed system, managing algorithm execution, LED signals, and communication between nodes.
- The Server module serves as the user interface for the application, managing the distribution and control of tasks. Additionally, this module manages the passing of task arguments, as well as the collection and export of resulting data.

\textsuperscript{1} e.g. easy to operate, troubleshoot, and resource undemanding
2.1 Picocluster platform for distributed algorithms

Figure 2.1. Showase of a GUI of the server module of Picocluster platform

The platform provides the base for communication between the server and the clients using the TCP protocol and peer-to-peer communication for the clients over UDP. Furthermore, it introduces a framework that allows the creation of various tasks that can be run across the clients and manage their runtime. The management includes assigning specific tasks to several clients, starting and stopping the task, and suspending or banishing clients. The framework also allows passing formatted parameters to each task and exporting the results.

The original implementation uses static ports for TCP and UDP communication, which requires starting each node on a unique IP address so the nodes can distinguish themselves and communicate.

2.1.1 Raspberry Pi platform

Figure 2.2. A Blinkt! module. Image taken from [1]

Although the Picocluster[1] platform can run tasks on any system consisting of separate machines connected to the same network, it has been specifically designed to run the nodes on a Raspberry Pi cluster. This cluster comprises individual Raspberry
Pi machines, each equipped with a Blinkt!\textsuperscript{2} module connected to its GPIO header, allowing the programmer to visualize the algorithm’s state via LED signals.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_3.png}
\caption{48-node Pico cluster connected to personal computer}
\end{figure}

Two of these platforms were available for this thesis — a Pico 5 cluster comprising five and a big cluster with 48 individual Raspberry Pi 4B+ boards. These single-board computers contain a 64-bit ARM v7 processor with four cores and 8 GB RAM\textsuperscript{1}. Full specifications of these clusters and Raspberry Pi boards can be found in Chapter 2 of Roman Janků’s bachelor thesis\textsuperscript{1}.

The individual boards’ ethernet ports are connected to an internal switch, which is connected to the side of the cube and allows the connection to all included boards. The boards have a predefined static IP address and operate on network 10.1.10.0/24\textsuperscript{1}. The scheme of connection can be seen in Figure 2.4, and the step-by-step manual on connecting to the cluster can be found in Appendix A of the above-mentioned bachelor thesis of Mr. Janků.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_4.png}
\caption{A scheme of network connection of the Raspberry Pi clusters. Image taken from \cite{1}}
\end{figure}

\textsuperscript{2} https://shop.pimoroni.com/products/blinkt?variant=22408658695

\textbf{RaspberryPis}
2.2 Algorithms

The multi-agent coordination problem was chosen as a typical example of a more advanced distributed algorithm with practical usage and many possible solution approaches, which can produce exciting results for the scaling evaluation. The selected algorithms are ADPP (Asynchronous Decentralized Priority Planning) and COBRA (Complete Best Response Approach). These algorithms were selected because they are implemented using the same libraries and were proposed, and both were developed by teams led by Dr. Michal Čáp. These facts allow easy integration of both algorithms. Moreover, they solve the problem using different approaches, resulting in more complex data.

An essential feature of these algorithms is that they guarantee a solution in a well-formed environment. This term stands for environments that guarantee an existing path between any free docks (e.g., start and goal positions), even if other agents occupy all other docks or there are planned trajectories between these nodes. The precise definition of this term can be found in section 4.3 of Michal Čáp’s doctoral thesis.

![Well-formed and not well-formed infrastructure](image)

Figure 2.5. Examples of well-formed and not well-formed infrastructure. Image taken from [5]

2.2.1 ADPP algorithm

The Asynchronous Decentralized Priority Planning algorithm was proposed by Michal Čáp, Peter Novák, Alexander Kleiner and Martin Selecký, and also solves multi-agent path-planning problems.

The ADPP algorithm comprises several vital steps. Firstly, the agents are initialized with their initial positions and goal locations. Then, priorities are assigned to the agents, which are determined by their names. The names assigned are unique and dependent on a numeric id, and the priorities are set alphabetically.

Subsequently, agents simultaneously plan their paths. Upon the planning completion, the agent sends a new message to all other agents, informing them that it has generated a new trajectory. Receiving this message from an agent with higher priority triggers collision detection, which may cause new trajectory planning. Each agent stores the trajectories of higher-priority agents and uses them as dynamic obstacles when planning new trajectory.
When the highest priority agent finishes planning, it also sends a message informing other agents that it has finished. When an agent has a non-conflicting trajectory and receives this message from an agent that is precisely one priority above, it also finishes. After the lowest priority agent finishes in this manner, it recognizes that it is the last one to do so and informs all other agents about a successful convergence, which ends the algorithm.

This approach more effectively exploits the advantages of a distributed system, as it allows multiple agents to plan simultaneously. The best-case scenario runtime is therefore based on the longest time of any of the agents’ planning. In contrast, the worst-case scenario includes the lowest-priority agent to plan once for each other agent. A comparison of the runtime of these two scenarios can be seen in Figure 2.6.

The Figure 2.6 also represents a rough comparison of the algorithm’s runtime results in a distributed environment and on a limited count of single-CPU threads, which will always produce results resembling the worst-case scenario. The effect responsible for this resemblance is that only one agent can plan at a time.

Figure 2.6. Comparison of the worst-case (on the left) and the best-case (on the right) scenarios of ADPP. The image was edited from a base from [3]
2.2.2 COBRA algorithm

Continuous Best-Response Approach (COBRA) [4] is a multi-agent path-planning algorithm proposed by Dr. Michal Čáp, doc. Jiří Vokřínek and doc. Alexander Kleiner in 2015. The algorithm introduces a planning token, which includes information about all currently planned trajectories, and whose possession is a necessity for an agent in order to plan his trajectory. Such an approach guarantees a solution in well-formed[5] infrastructure. The original algorithm is implemented in Java, with a handful of testing instances and a visualization tool.

Figure 2.8. Showcase of COBRA runtime visualization

Agents participating in this algorithm can create any number of tasks sequentially but will always have to obtain the token to plan. This is the cause of the algorithm’s solution guarantee. There will be no conflicts between the trajectories as only one agent can plan at a time and possesses all currently planned trajectories. However, this also introduces a prolongation caused by the race for token possession and sequentializes the algorithm’s runtime.

While talking about the algorithm’s runtime, it is essential to note that it is implemented as a theoretical simulation. Each agent is run on a separate thread, with the token being implemented as a shared Singleton object, whose acquisition is controlled by a critical section. Furthermore, the time spent on a trajectory is simulated in real-time. This causes the algorithms’ CPU requirements to be minimal during the runtime, as, for the most part, there is no planning, and the agents are only traveling.
Chapter 3
Algorithm integrations and platform enhancements

This chapter summarizes the work that had to be done on all the subprojects. At first, an extensive examination of all the projects was conducted, from which the steps that will have to be undertaken emerged. These steps are summarized in the section 3.1, with a more detailed description of the actual platform enhancement and argument integration in the later sections.

3.1 Proposal of solution

After carefully examining all the base projects, several tasks must be taken care of to measure the algorithms’ scaling in different environments.

- Firstly, enhancements to the Picocluster platform must be made. These enhancements include reworking the inter-client communication system, allowing multiple clients to run on a single machine.

  In order to achieve this, the UDP port assignment and messaging must be enhanced so that the clients can find a free UDP port if the default one is taken. Furthermore, they must report the UDP port they are using to the server so that it can then pass all used ports mapped to clients to all co-workers participating in a task.

  Additionally, the UDP messages will have to include an identifier of the sender client so that the clients know who sent them a message, as the original messages only contain content serialized to String. The sender was determined based on the IP address.

  A method that will allow all the nodes to run the same cores and threads of the CPU will also have to be found.

- Secondly, both algorithms must be reworked as Tasks for the platform. This will include adding a communication layer using the platform’s interfaces. The algorithms will be reworked to allow a correct runtime using this changed communication and produce correct results in an identical format as the original implementations.

  For ADPP, this means adding a layer above its simulation messaging channels and binding them to the messaging layer of the platform.

  For COBRA, this will include reworking the Token passing completely, with a mutual access guarantee and request handling.

- A specialized task for the platform will also have to be created, which will allow starting of the experiments of each algorithm. This experiment will load multiple problem instances, parse the input parameters, and run the algorithm on these instances sequentially, saving the results in a correct format.

- Lastly, various deployment environments must be selected on which the scaling can produce engaging results. Such metrics should be chosen and measured, allowing the appropriate insight into the differences in these results.
3.2 Enhancements of the Picocluster platform

Even though the original Picocluster platform for running distributed tasks, created by Roman Janků, is a well-implemented solution that allows the user a sufficient degree of task handling and a custom algorithm implementation, several changes had to be made.

In this chapter, I will only discuss the significant changes done to the platform specifically to enhance the platform’s capabilities for this thesis. In contrast, other enhancements and changes, which were a part of implementing the algorithms, will be discussed in the chapter 3.3. All the changes can be seen in a forked repository on GitLab.

3.2.1 Modifiable environment setting

A new environment variable called ON_CLUSTER was added. This variable defaults to false and, as can be guessed, specifies whether the node is running on a cluster. This variable’s effects indicate whether the node should behave as originally implemented (option true) or use customized runtime settings to run on a non-Raspberry Pi machine (option false).

One of these effects tells the node whether to use the Blinkt! module, which is originally used by default, and would cause a runtime exception if the module was not connected to the machine. If the variable is set to false, the program will presume no such module is connected, and a blinkItSupported boolean in the BlinkT class will cause all functions working with this module to return before any effect occurs.

3.2.2 UDP communication

Another change the ON_CLUSTER environment variable enforces is the change in UDP port selection. As mentioned in section 2.1, the original implementation works with the node modules running on unique IP addresses, which would not allow us to run multiple nodes on a single machine, as proposed for this experiment.

The port used for the UDP messaging was statically derived from the server TCP port for each client, which would result in all the nodes trying to access the same port if they all would use the same IP address, for example, when running on localhost. In order to overcome this, the ON_CLUSTER variable changes how the UDP port is determined. Instead of deriving it from the server’s TCP port, it uses the node’s TCP port as a base and adds 1000. This approach does not consider more node processes on a single cluster machine.

The subsequent issue with the transition to the same IP address is caused by the fact that the original nodes determine the sender of a UDP message by the address from which it was sent. This was handled by adding the UDP port to the NodeDto class, which holds information about all connected nodes for the server and all known co-workers for the nodes. In order for the server, which sends this list of connected nodes to each node, to retrieve this information, the UDP port was added to the UUID request message a node sends to the server upon connection, formated as follows:

```
<uuid:String>/<udpPort:String>
```

Therefore, the NodeDto class contains a node’s UUID, address, and both TCP and UDP ports. This allows the nodes to find the appropriate UDP port for any node.
it wants to send a message to. In order to recognize which node sent an incoming UDP message, the node's UUID was added to any UDP message sent in the following format:

```
<uuid:string>:<messageContent:String>
```

Any receiving node then splits an incoming message, serialized as a String, by the first colon and uses the first part of the message to identify the sender.

A small addition has been added to the communication layer. Each message's content field is compressed by default using GZip library. This was conducted to limit the size of larger messages passed in the distributed network.

### 3.3 Integrating the algorithms

In order to run the algorithms on the Picocluster platform, both algorithms had to be integrated into the platform as executable Tasks. Creating the Tasks was relatively easy, as the well-prepared structure of the platform required only the creation of new algorithm classes that would extend the `AbstractAlgorithm` class and take care of creating the tasks from the implementation provided. Furthermore, the shared code base of both algorithms allowed me to work with a high level of abstraction while creating these classes, with many methods and functionalities shared between them. Nevertheless, both algorithms required a higher amount of custom integration changes and case-specific communication additions.

#### 3.3.1 Shared features

Many features could be implemented as a part of multiple abstract classes, providing basic functionality for both algorithms, including scenario creation, argument and result parsing, algorithm startup, and other management. An agent thread, which would simulate the remote agent on the node, could also be abstracted.

**Parameter parsing**

One of the first issues taken care of was that both algorithms accept parameters as command-line arguments, resulting in a String array of both argument names and values. Picocluster, however, utilizes an array of String, which contains only the argument values sorted in a way set by order of the parameters in GUI. To resolve this issue, the `AliteArgumentsParser` class was implemented, providing static methods which take care of the correct argument parsing in both ways. This allows the parameters passed into the Picocluster server GUI to be parsed so that the algorithm ScenarioCreator can create a valid problem.
3.3 Integrating the algorithms

Figure 3.1. Showcase of available parameters for COBRA algorithm in the GUI

Additionally, default parameters for both algorithms were prepared and are used if the user does not fill in any of the parameters in the GUI. All the parameters required by the algorithms are prepared in this manner, and all possible settings and instances can therefore be started on the platform. An argument called `agentsRemote` has been added, which cannot be set by the user, and is always `true` when launching the algorithm from the GUI. This selects the mode in which the algorithm runs, as will be discussed later.

As for the parameters themselves, their definition can be read in the README files of ADPP\(^3\) and COBRA\(^4\) in the original repositories of Dr. Čáp.

The `problemInstance` parameter requires the full name of the file with the serialized problem instance, specifying only the folder of the desired instance set and the problem file, e.g., `ubremen-r27-docks/2.xml`. The program will add an adequate path prefix if the program is launched correctly. The `showVis` parameter specifies whether visualization of the algorithm should be started, with `true` as a default value.

**Scenario Creation**

When the algorithm is launched, the parameters inputted by the user are parsed to arguments for the algorithm and are passed to the `ScenarioCreator` class via an `AlgorithmDirector`, which will be further discussed in section 3.3.1. The `ScenarioCreator` classes offer static methods for deserializing the problem file and further parsing the argument and are different for each algorithm.

These classes’ `createFromArgs(String[] args)` methods would originally start the original implementation of the algorithm. However, they have been refactored for this experiment, changing their return value from `void` to a `Parameters` class instance. The method now behaves according to the new `agentsRemote` argument and starts the original simulated implementation only if this argument is `false`. However, if the argument is set to `true`, the method will not start the simulation and instead returns the `Parameters` instance. Moreover, the method sets a static field containing the deserialized problem. For ADPP, this field contains an instance of the `EarliestArrivalProblem` class, while for COBRA, it returns an instance of the `RelocationTaskCoordinationProblem` class. These classes are specific for each of these algorithms and included in their dependencies.

\(^3\) [https://github.com/mcapino/adpp-journal/blob/master/admap-solver/README](https://github.com/mcapino/adpp-journal/blob/master/admap-solver/README)

It is important to note that the method mentioned above was further reworked to allow custom problem deserialization. A static field called `problemString` was added and can be set before calling the `createFromArgs(String[] args)` method. This results in deserializing the problem from the String provided rather than from the specified problem file. The goal of this approach is to lower the amount of resources needed to deploy to the nodes before the start of the experiment and allows the nodes to create the problem from a String passed from the server when a new task starts. The server, therefore, uses the serialization from the problem file and then sends the contents of this file to the node along with other arguments required to start the algorithm. The node then creates its copy of the problem being solved from this String.

As mentioned above, the problem and parameter objects represent the whole instance and are created by both the server and the node. Further information on solving the problem using this distributed approach is discussed in the following sections.

The distributed runtime structure and starting of algorithms

Both algorithms are integrated very similarly, as can be deducted from this chapter, and their startup and runtime basics are almost equivalent.

An instance of the solved problem is created on the server and all participating nodes. While the nodes create the problem and then start an instance of an agent designated to solve this problem — this will be thoroughly discussed in section 3.3.2 for ADPP and 3.3.3 for COBRA — the server will start instances for each agent participating. These instances serve as dummy objects and are not active during the algorithm’s runtime. Their purpose is to store each agent’s task and trajectory, allowing the server to visualize the problem if visualization is enabled, which will be discussed more later.

![Failed ADPP algorithm caused by insufficient number of nodes for problem](image)

In the following text, I will talk about `TaskDirector` and `AgentThread` which will be more deeply explained later in the thesis in sections. The algorithms startup can be summarized in a couple of steps:

- User inputs the parameters in the GUI and starts the task
- The server recognizes that the task is either ADPP or COBRA and parses the parameters to the appropriate algorithm arguments
- The server starts the appropriate `TaskDirector` and creates a local version of the problem based on input data
3.3 Integrating the algorithms

- If the parameters are malformed, the problem file is missing, or the number of nodes assigned to the task is lower than the number of agents required to solve this problem, the server marks the task as **Failed** and adds an appropriate error message.
- If the previous step is successful, the server sends a **NewTask** message to all assigned nodes, containing all arguments and content of the problem file.
- All nodes create the problem from the received data and send an **AGENT-INITIALIZED** message to the server after creating the problem.
- After the server receives the above-mentioned message from all nodes, assigns an ID to each node and sends a **StartAgent** message containing the ID and additional algorithm-specific variables to each node. The server also starts the visualization at this point if it is enabled.
- After a node receives the **StartAgent** message from the server, and the ID assigned to it is higher than the number of agents needed for the current problem, it ends with a partial result **Agent was not needed for this problem**. Otherwise, it creates an appropriate **AgentThread** with an appropriate Agent instance and proceeds to start the agent.

After each agent finishes, it returns its partial result. These results are collected by the Picocluster’s server and are combined using a method defined in the algorithms base class.

**Figure 3.3.** Showcase of ADPP runtime logs on GUI. The problem required 2 agents, but was assigned to 5

**AgentThreads**

Each node contains a thread designated only to run a remote agent for each algorithm. Both threads extend an **AbstractAgentThread** class because of the differences in the algorithms’ specifics but use the same principle.

This thread is started after receiving the **StartAgent** message from the server, as mentioned above, and immediately creates an instance of an appropriate Agent required to solve the respective algorithm. A loop is then started that can be stopped after a **StopAgent** message is received from the server or when the agent recognizes the end of the algorithm. The loop’s function differs for each agent and will be discussed later. This thread can also accept messages received from the server and...
3. Algorithm integrations and platform enhancements

other agents and either process them or pass them to the agent, depending on the context.

- **Algorithm directors**
  
  The server counterpart of AgentThread is a task director. These encapsulate all methods used to run, visualize, and control the integrated algorithms. Even though each director offers different methods for the respective algorithm, a shared base is implemented in the AbstractTaskDirector class, which both ADPP and COBRA extend.

  Both directors can create the problem instance from input data and start the algorithms as described in section 3.3.1. They also take care of all communication with the remote agents during the algorithm’s runtime and update the local instances of the agents with passed trajectories in order for them to be visualized.

  Additional specific functionalities of the directors will be discussed in upcoming chapters.

- **Visualizations**

  In the original implementation, visualization is started after creating the problem in the createFromArgs(String[] args) method. However, for this thesis’s usage, the visualization code was moved to a new public method in the ScenarioCreator class and launched from the TaskDirector. This approach allows the visualization to start after all agents are successfully initialized.

  The local agent instances of the director are passed to the visualization, which then visualizes their trajectories and can change them instantly when the director updates the trajectory after a message from a node is received.

  The visualization is run using the VisUtil tool, which is a part of cz.agents.alite:de-conflictiontools toolkit developed by Agent Technology Center at Department of Computer Science and Engineering of Czech Technical University in Prague\(^5\). The visualization is implemented as a singleton, with a hard-coded shutdown hook that closes the whole application when the visualization window is closed. This prevents us from closing the visualization window when opened; however, thanks to the singleton approach, we can use the same visualization for any subsequent algorithm runs while being free to change the algorithm used.

- **3.3.2 Integration of ADPP**

  The specific parts of the integration of ADPP will be discussed in this chapter. The algorithm was relatively easy compared to COBRA, which will be discussed later. It was relatively quick and straightforward thanks to the original implementation of ADPP, which already uses messages and inbox-based communication on simulated channels.

  - **Parameters**

    Only five arguments needed to be implemented and assigned default values. Their names, default values, and a brief description can be found in the table 3.1.

  - **Task director and Agent thread**

    The ADPPDirector and ADPPAgentThread classes do not differ much from their abstract parents. They primarily handle messages, which are for this algorithm very simple, as will be discussed lower.

\(^5\) [https://github.com/aicenter/alite](https://github.com/aicenter/alite)
3.3 Integrating the algorithms

<table>
<thead>
<tr>
<th>Argument</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>problemfile</td>
<td>ubremen-r27-docks/0.xml</td>
<td>A relative path to the problem file from the instances directory</td>
</tr>
<tr>
<td>timestep</td>
<td>27</td>
<td>A timestep used as a tick length</td>
</tr>
<tr>
<td>timeout</td>
<td>15000</td>
<td>An algorithm timeout in ms</td>
</tr>
<tr>
<td>maxtime</td>
<td>15000</td>
<td>Maximum time available for trajectory planning in ms</td>
</tr>
<tr>
<td>showvis</td>
<td>true</td>
<td>A flag enabling the visualization</td>
</tr>
</tbody>
</table>

Table 3.1. ADPP arguments with default values and description

The agent thread’s loop calls the `tick()` method of the ADPP agent, but it is essentially purposeless for this type of agent. The only significant use of the thread is to queue messages meant for the agent when the agent is not yet fully initialized. The causes of this will be discussed in section 3.3.2. Another major usage of the thread is to set the BlinkT LEDs accordingly, create a `Communicator` object, which serves as a message-accepting interface for the agent, as will be discussed later, and detect the finish of the agent’s computations.

### Agent

There were no actions needed in refactoring the ADPP agent. The original implementation is sufficient for this thesis, as it uses communication channels, which could be enhanced with a specifically implemented interface and will be discussed later.

### Communicator

The most crucial change for integrating ADPP into the Picocluster platform was the creation of a custom implementation of the `Communicator` interface. This interface provides essential functions for the ADPP’s messaging layer, allowing the agent to receive and send messages. A new `ExternalCommunicator` implementation was created with a helper `ADPPCommunicatorAdapter` class.

The communicator only registers a listener and is called by the agents upon sending a message, which is passed to the listener immediately. This approach was chosen to overcome the class scope issues caused by the multi-module structure of the project.

The `ADPPCommunicatorAdapter` is an adapter between Picocluster’s and ADPP’s messages. It encapsulates the inner messages of ADPP, which were left unchanged, and serializes them into the content of the `Message` class of Picocluster’s communication framework. The class comes with a handful of serialization and deserialization tools used to losslessly store the ADPP’s inner messages into the outer-scope messages.

There was no need to create any new Picocluster messages for this algorithm apart from the new `ADPP-INNER-MESSAGE` message type. This message type was used to properly recognize the messages meant to be parsed inside the agent’s communicator, are only sent between the nodes, and contain the original ADPP inner message in a serialized format.

For other parts of ADPP runtime, already existing message type for trajectory notification made for COBRA could be used along with `StargAgent` and `AGENT-INITIALIZED` messages, with other needs concerning the task handled by Picocluster’s original messages.

### Startup issues

An issue emerged when testing the integration of ADPP on the Raspberry Pi cluster concerning an edge-case scenario introduced by the integration. Some remote boards...
sometimes take much longer to initialize the agent thread, even after creating the problem and starting the thread when sending the `AgentInitialized` message to the server. This often resulted in the highest-priority agent finishing his task very early and sending the `AgentFinished` message to nodes that did not yet have the AgenThread running. These nodes would then forever wait for the higher-priority agent to finish, not knowing it has already finished.

This issue was solved by introducing a message queue on both the base ADPP Task instance and the communicator, where all incoming messages are stored if no thread is running. The queue has a priority and is checked when any new message arrives and when the agent starts, using a critical section to ensure that no message passes before an earlier-received one. This successfully intercepted all messages sent to the agent and made sure that all messages would not only be parsed by the agent but also that they would be parsed in the correct order.

**Blinkt! visualization**

The Blinkt! module, described in section 2.1.1, is used to visualize the agent’s state during the algorithm’s runtime. The four left-most LEDs light up in orange to signal that a node works as an ADPP agent. The right-most LED then signalizes the count of trajectory planning, cycling through blue, green, and red after every finished planning. When an agent finishes, it lights up all the LEDs in green, apart from the right LED, which still shows the last trajectory planning color. When the lowest-priority agent registers a convergence, it lights all its LEDs in blue.

![Figure 3.4](image-url). ADPP’s visualization over Blinkt! module. We can see that the two topmost nodes has finished (nodes 0 and 1), and that nodes 0, 1 and 3 had to only plan once up to this point, while nodes 2 and 4 had to replan at least once.
3.3.3 Integration of COBRA

Integrating COBRA into the platform took a lot of work compared to ADPP. The reason for this was the different approach in implementation, which did not simulate a distributed environment. It was implemented as a pool of Java Threads, one for each agent, with the token being represented as a static object in memory, enclosed within a synchronized block for ensuring mutual access. Therefore, a completely new messaging had to be implemented, ensuring mutual access to a shared variable.

Parameters

The parameters used for COBRA are very similar to those used by ADPP, with an additional number of tasks for each agent and a seed used for randomizing the tasks. The parameters, default values, and description can be found in table 3.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>problemfile</td>
<td>ubremen-r27/1.xml</td>
<td>A relative path to the problem file from the instances directory</td>
</tr>
<tr>
<td>ntasks</td>
<td>4</td>
<td>A number of tasks each agent must fulfill</td>
</tr>
<tr>
<td>timestep</td>
<td>650</td>
<td>A timestep used as a tick length</td>
</tr>
<tr>
<td>maxtime</td>
<td>6000000</td>
<td>Maximum time available for trajectory planning in ms</td>
</tr>
<tr>
<td>timeout</td>
<td>6000000</td>
<td>An algorithm timeout in ms</td>
</tr>
<tr>
<td>seed</td>
<td>8</td>
<td>Seed used for random number generation throughout the algorithm</td>
</tr>
<tr>
<td>showvis</td>
<td>true</td>
<td>A flag enabling the visualization</td>
</tr>
</tbody>
</table>

Table 3.2. ADPP arguments with default values and description

Remote Agent

Unlike the ADPP algorithm, a new agent class called RemoteCOBRAAgent had to be created. It extends COBRAAgent class as a parent and overwrites some methods to work on a distributed environment. This class’s main achievement is separating trajectory planning from agent’s tick into separate methods. Therefore, an instance of this class does not plan trajectory on tick but only sets wantsToStart flag to true if it needs to start planning. This flag is then handled by CobraAgentThread, which calls the agent’s method to plan a new trajectory after receiving the token and updating the local copy of planned trajectories.

COBRA’s agent thread

The COBRA’s implementation of AgentThread called CobraAgentThread serves much more purpose than the ADPP one. Not only does it handle the message passing, but it also includes the logic of token requests and passing. Furthermore, its loop calls the tick() method on RemoteCOBRAAgent and subsequently checks whether the tick has resulted in the agent requiring a token to plan. If it has, the thread asks other nodes for the token if it does not possess it already. The thread’s loop also takes care of new trajectory planning if the agent possesses the token. Upon completing the agent’s runtime, the thread will not quit but wait until it exchanges AgentFinishedNotification with all other agents so that it can still participate in the token exchange.

Token and Current tasks

The token itself includes current trajectories for all the agents and is implemented as a singleton object. Each agent and server have their own instance of the token, which is updated by TokenDto messages.
The CurrentTasks class holds all possible docks of current problems, including the possible starting and ending points for individual tasks and agents. Each agent and server also have their own copy of this singleton, which is being updated after receiving the token and updating current trajectories.

### Token passing

The token is initially owned by the server, which fills it with the starting trajectories of all agents — trajectories that stay on the starting place of the agent for his assigned start delay period. When any agent decides it needs the token, it sends a message to the server, releasing the token and never holding it again. After receiving the first token, the node plans a new trajectory, saves it into the token, and the algorithm runs in a standardized manner from this point.

#### Figure 3.5. GUI logs showing the passing of the token during COBRA algorithm run

Whenever an agent asks for the token, it broadcasts the TokenRequest message to all co-workers over UDP. Upon receiving this message, the recipient saves it into his request database. This database is implemented as a HashMap, consisting of the agent’s id as a key and the request as a value. This guarantees that only the latest request is present for each node in each database.

When a node wants to send the token to a new recipient, it checks the request database. Thanks to each request’s timestamp specifying when the request was created on the sibling node, the longest-waiting node is always selected as the next recipient. If the current request database is empty, the node marks itself as the next recipient. The node then serializes the token, sets the nextRecipient field to the id of the next recipient, and sends the token to the server over TCP as a TokenDTO message.

This is done because the token can be enormous in more complex problems, and TCP communication between nodes still needs to be implemented. Therefore, the current UDP communication is unsuitable for sending these big messages.

The server updates its local Token and Tasks, which are used for visualization, and immediately sends the token to the next recipient.

Upon receiving the token, the recipient node deletes all records with the requestedAt timestamp older than the current time. This is done to prevent old requests from interfering with the process. The node then sends a TokenRequest message with the isNotification flag set to true to all co-workers to inform them about the acquisition. The nodes can now re-send the token request to this node if their interest in it still prevails.

### Task director

The CobraDirector class, which serves as an algorithm director for COBRA, includes several additional features apart from the ones specified in section 3.3.1.
3.3 Integrating the algorithms

The main addition to the class is token management. The director serves as the first holder of the token and fills it with pre-generated trajectories for all the agents. These trajectories describe the agents’ initial position on their first dock, with the trajectory remaining in this position for the amount of time determined by the agents’ initial waiting period. This allows the first nod to acquire the token to plan its trajectory with the other agents in mind. The director sends the token only to the first agent that sent a request, after which it disables the `tokenPassing` flag to `true` and will not send any other tokens during this instance. The second part of token management comes with token passing, as mentioned above.

The CobraDirector also serves as a keeper of CommonTime, a synchronization clock across the distributed network. This time synchronizes the nodes to start the task at a common timestamp sent to them when starting a new COBRA task. The need to synchronize the clock before each task run is since the visualization cannot be closed, as discussed in section 3.3.1, and subsequent simulations must therefore start at a later timestamp in order to be visualized by the visualization tool, which keeps incrementing the time.

Messages

Several new messages and message types were added for the COBRA’s integration. Along with this, a couple of new DTOs had to be implemented to effectively transfer data.

- **CobraAgentInitialized** — Message type for status update, sent by node to server after creating a local instance of the problem from passed arguments by regular start message.
- **StartRemoteAgent** — This message is sent to nodes by server, after receiving CobraAgentInitialized status update from all nodes that are to be working on the task.
  - id - an id assigned to the agent
  - tickLength - an uniform ticklength to be used on all agents
  - clockStartedAt - a time when the server’s clock was started. This is used by the node to sync
  - initializedAt - simulation starting time in milliseconds, to be a base for all agents
- **TokenRequest** — A message sent by node to node over UDP, requesting the token or notifying about the acquisition
  - requesterId - an id of the agent that sends the message
  - requestedAt - a time in ms, describing the time when the agent decided it needs a token
  - isNotification - flag, true if the message serves as a notification about acquisition of the token
- **AgentFinishedNotification** — Sent by node to all nodes over UDP, informing them that it had finished all its jobs. Each of the agents holds a list of all finished co-workers, and can fully end, and send result to the server, only after receiving this message from all coworkers.
  - agentId - id of the agent that sends this message
- **TokenDto** — TokenDto consists of all informations necessary to transfer and update token information between nodes.
  - nextHolderId - id of an agent to acquire the token next. This is determined by the sender, based on its request database
  - trajectories - a map with agent ids as key, and MovingCircleDto as a value, which includes all actual trajectories
MovingCircleDto – This is a DTO that is passed as a part of TokenDto message, serving as a transfer object for a trajectory
- trajectory - custom serialization of the trajectory, consisting of a byte array of subsequent elements of the trajectory’s coordinates
- radius - radius of the moving circle. This field is needed to recreate the trajectory representation in Token from MovingCircle
- samplingInterval - interval used for sampling. This field is needed to recreate the trajectory representation in Token from MovingCircle

Blinkt! visualization
This algorithm also uses the Blinkt! module described in section 2.1.1. To visualize this algorithm is running, the left-most 4 LEDs light up in blue. The right-most LED visualizes the status of the token. It lights up blue when the node has just asked for the token, changes to green when the agent has acquired it, and then switch to red when the token is passed further. The second right-most LED visualizes the trajectory change, switching to a random one after every new planning. The third and fourth LEDs from the right light up red when the agent fails to plan a trajectory within a designated time and green when the agent has finished all its tasks.

Figure 3.6. A showcase of COBRA’s visualization using the Blinkt! module. We can see that the top-most node currently holds the token and is planning, the lowest module has requested for the token and is waiting, and the middle nodes have planned their trajectories and are currently on the way
Chapter 4
The experiment

4.1 Preparing the experiment

With both algorithms integrated and the Picocluster enhanced, the last task left to overcome is automatizing multiple instance runs of the algorithms, e.g., creating an experiment that would subsequently start multiple instances from pre-generated inputs and collect the outputs.

After enhancing the argument parser mentioned in 3.3.1, the platform can flawlessly parse the arguments from one format to another. Therefore, original scripts for generation experiment instances could be used to prepare the input data.

An AbstractExperimentDirector class has been created, with two children classes, ADPPExperimentDirector and CobraExperimentDirector. Moreover, two dummy tasks have been created, called ADPPExperiment and COBRAExperiment. These tasks accept only one parameter, showvis, toggling the visualization for the whole experiment.

When a user starts any of these tasks, the server will recognize the attempt to start the experiment and will create an appropriate experiment director instead of starting the task. The director will then load all data from the prepared input files, create a task from each with parsed parameters, and enqueue the tasks into the platform’s task queue. After each task is finished, stopped, or failed, the director is notified and writes the task’s result into the output file.

![Figure 4.1](image-url)  
Figure 4.1. A showcase of the server’s GUI during the runtime of ADPP experiment
4. The experiment

4.2 Experiment measuring

The experiment was conducted in multiple environments and settings, with each algorithm collecting different data for further analysis. The experiment’s goal was to collect the results of the implemented algorithms for the same problem instances in all available environments to evaluate the proposed metrics for various instances correctly.

4.2.1 Preparations

As mentioned in section 4, the integrated algorithms already return the measured variables in the same format as the original implementation. Furthermore, they can process the input data in the same format. This approach allows the experiment to utilize the original scripts to generate problem instances and result processing. Various scripts have also been added to help prepare specific experiment instances, to run instance set generation, result processing, and plot generation automated for selected environments.

The instance set generation scripts have been edited to add a new parameter, specifying the algorithm’s suffix. This suffix distinguishes between various algorithm runs based on the environment they run in, e.g., -Distributed, -singleProcess, or -4Core. It is printed out in the algorithm column in the results file and serves as a distinguisher for the plotting script. The plotting scripts have also been edited to generate plots for each measured variable and additional plots with the distributed environment’s results being normalized.

4.2.2 Environments

When talking about experiment environments, I refer to the combination of a spatial environment where the algorithm’s agents perform their computation and the deployment environment on which the algorithm is running.

Spatial environments

The spatial environments used in this experiment came as a part of ADPP and COBRA subprojects, each having three predefined environments. These environments are the same for both projects, and the algorithms could thus perform in the same settings. Each of the algorithms contains a script to generate various instances with various seeds and a number of agents. The ADPP instances have two alternatives - with and without docks. The docks symbolize predefined starting and ending points for each task, which are generated so the environment is well-defined and will always have a result, as mentioned in 2.2.

For this experiment, the environment variant with the docks has been selected, as it always guarantees a solution, and the objective of this experiment is to evaluate the performance and differences of the algorithms above, not to test their success rates. Moreover, the COBRA algorithm requires predefined docks to select available tasks, which is the only alternative for this algorithm.

Used environments:

- Empty hall — A basic environment consisting only of a large rectangular area with minimal obstacles. The docks are positioned around the outer walls.
4.2 Experiment measuring

- **Warehouse** — A more complex environment consisting of multiple rows of walls. The docks are positioned next to each row, and in small niches around the circumference.

- **Ubremen (Office corridors)** — Being the most complex environment of the three, it consists of various rooms, connected by passages where only one agent can fit and other bottlenecks and obstacles.
4. The experiment

### Deployment environments

As mentioned above, a raspberry pi cluster has been used to test the algorithm results in a truly distributed environment. Furthermore, to compare these results with a simulated environment, the results of the original single-process implementations of the algorithms were measured, and the multi-process enhancement of the algorithms was also run on a personal computer.

The last mentioned measurements were conducted on various numbers of CPU threads of the computer to test the algorithm performance based on hardware possibilities, concretely on 1, 4, and 7 threads. Starting the experiment on all eight threads of the available CPU yielded invalid results, as the process flooded the processor and the whole operating system froze in a struggle for resources. Starting the experiment on threads 1-7, with thread 0 left out for the OS, has been enough to prevent the problems mentioned above.

All the separate node processes had to be started on a specific set of threads to run the experiment with the thread limitation. To achieve this, the node executable was built as a Docker image and was started using the following command:

```
docker run --cpuset-cpus="1-4" picocluster-node
```

<table>
<thead>
<tr>
<th>CPU</th>
<th>CLK speed</th>
<th>cores/threads</th>
<th>RAM</th>
<th>CPU rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASUS VivoBook S15</td>
<td>Intel Core i7-8565U</td>
<td>1.80GHz</td>
<td>4/8</td>
<td>16GB</td>
</tr>
<tr>
<td>Raspberry Pi 4B+</td>
<td>ARM Cortex-A72</td>
<td>1.80GHz</td>
<td>4/4</td>
<td>8GB</td>
</tr>
</tbody>
</table>

Table 4.1. Comparison of hardware used for measurements

The CPU rating in Table 4.1 is based on the Single thread rating from cpubenchmark.net\(^1\), and is later used to determine the normalization factor for computational results.

#### 4.2.3 Experiment instances

Several measurements had to be undertaken in all combinations of the environments to measure the scaling of the algorithms in accordance with the proposed method. For every number of agents tested, multiple instances were created with different seeds, and all these instances were started on each of the three spatial environments. Furthermore, there are Tasks that specify a trajectory from point A to point B, with each algorithm having a different number of tasks per instance. The actual number of generated instances and tasks within them can be seen in Table 4.2.

<table>
<thead>
<tr>
<th>numbers of agents</th>
<th>seeds</th>
<th>tasks per instance</th>
<th>total instances</th>
<th>total tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPP</td>
<td>1;2;3;5;7;9;10;12;15</td>
<td>20</td>
<td>1</td>
<td>540</td>
</tr>
<tr>
<td>COBRA</td>
<td>1;2;3;5;7;9;10;12;15</td>
<td>5</td>
<td>3</td>
<td>135</td>
</tr>
</tbody>
</table>

Table 4.2. Number of experiment instances and tasks for each algorithm

### Determining the number of agents

After careful consideration and experimental testing, it was decided that the experiment would work with a maximum of 15 agents. This 15-agent limit was due to

\(^1\) [https://www.cpubenchmark.net/](https://www.cpubenchmark.net/)
the limitations in personal computer RAM, which could not handle more. The cause of this problem was that each agent runs on its own JVM, holding the trajectory of each of his co-workers, which caused the memory requirements to increase by a square. Additionally, the starting messages of the algorithms, which contain the serialized problem for the agents, caused the switch on the Picocluster platform to overload, resulting in a temporary shortage of power and subsequent restart of all RapsberryPis, when the messages were sent to more than 18 agents at once.

**ADPP**

Due to the nature of the original implementation of ADPP, each agent only plans one task in each instance. Thanks to this approach, each instance was finished within a few seconds, allowing me to generate 20 instances for each number of agents, with 20 different seeds. This resulted in 180 instances for each spatial environment, totaling 540 instances per experiment for each deployment environment.

**COBRA**

The COBRA algorithm allows each agent to start up to n tasks, which causes the algorithm’s runtime to increase, as mentioned in section 2.2.2. It was decided to use three tasks per instance for each agent, which averages the runtime in about three minutes but also ensures the data collected by the algorithm reflects the waiting and prolongation times more accurately, compared with only one task experiment, which the initial timeout can influence. Because of the longer runtime, only five instances were generated for every number of agents. Therefore, this algorithm’s total count of instances is 45 per spatial environment and 135 experiment instances for each method, with a total of 405 tasks.
Chapter 5
Results and Analysis

The average runtime of one COBRA experiment was around eight hours, while the ADPP experiment took around 80 minutes. These experiments have been run on each deployment environment, e.g., the distributed Picocluster platform and personal PC. As mentioned, 1, 4, and 7 threads were used to measure the experiment’s scaling on a personal PC, resulting in four deployment environments for each experiment. The generated instances were also run on the original implementation of the algorithms, totaling up to 45 hours of runtime.

The results were collected in output files, one for each spatial environment, with a CSV format and backed up after each run to merge them for final result processing.

All the results presented in relation to a number of agents are an average of all the runs for this agent in this environment with different seeds.

Note that the core in the plots denotes the CPU’s logical cores, e.g., thread and that the `singleProcess` method represents the original implementation of the algorithms.

5.1 ADPP results

The ADPP algorithm’s measured variables primarily focus on the time required to achieve convergence and the number of required replannings, as these are the primary metrics used to evaluate this algorithm’s scaling.

![Figure 5.1. Time to convergence for Empty hall environment](image)

5.1.1 Time to convergence

In figures 5.1, 5.2, and 5.3, we can see that the time to convergence varies widely, with the `singleProcess` method, representing the original implementation, having by far
5.1 ADPP results

The distributed method has produced the most significant delay, as was expected, but dependent on the environment, other environments also significantly prolong. The leading cause is the lower computational power of Raspberry Pi’s processors for the distributed environment, as will be discussed later.

However, we can see that some of the results for experiments run on threads of a single CPU also prolonged significantly compared to the original implementation, especially in more advanced environments. Apart from the Empty hall environment, which produces fewer conflicts, the environments create conflicts among the trajectories that must be resolved. The resolution of these conflicts increases the traffic on the UDP channels between the nodes and forces the nodes to assert all newly planned trajectories against their own, eventually replanning their trajectories if an inconsistency is detected, which further increases the resource demands by each node on both the processor and the communication channels.

We can see that the 1-Core method has had worse results than the other methods, caused by the struggle for resources on one thread. The 7-Core method has also produced a significant prolongation. This can be explained by the increasing memory

---

**Figure 5.2.** Time to convergence for Warehouse environment

**Figure 5.3.** Time to convergence for Ubremen environment
management when more cores simultaneously create new trajectories and deserialize trajectories of other agents. Such behavior increases the memory and CPU requirements and causes the core caches to overflow. Moreover, the fact that more cores are running worsens the percentage of lower-level cache hits, causing the prolongation.

5.1.2 Trajectory planning

The total number of planning per agent depends on the number of conflicts detected, as mentioned in section 2.2.1. As the figures 5.4, 5.5, and 5.6 show, each method’s average number of planning per agent is relatively similar, with slight fluctuations caused by the non-deterministic order in which the New Trajectory notifications can arrive. We can also observe more trajectory planning for more complicated environments. These plots also show that more agents planning simultaneously does not significantly influence the number of collisions created.
5.1 ADPP results

Figure 5.6. Average number of planning per agent for the Ubremen environment

5.1.3 Planning time

The average time spent planning all trajectories in the experiment instances can be seen in the figures 5.7, 5.8, and 5.9, including the time spent planning both the first trajectory and all subsequent replannings. A plot denoting the average planning time of one trajectory is not shown here because the results are similar to the summary planning time due to minimal differences in the average number of plannings. These plots can, however, be found in the attachments, as specified in appendix B.

Figure 5.7. Average planning time per agent for the Empty hall environment

These plots are another example of the effect of Raspberry Pi’s low computational power, as the distributed method produces higher planning times. However, the plots also demonstrate the problems caused by multi-thread methods. It can be clearly seen that the 1-Core method also introduces a delay in planning caused by the context switches in the middle of planning and nodes waiting for the processor time. Moreover, the 4-Core and 7-Core methods also produce a delay, which is
5. Results and Analysis

![Graph 1](image1.png)

**Figure 5.8.** Average planning time per agent for the Warehouse environment

![Graph 2](image2.png)

**Figure 5.9.** Average planning time per agent for the Ubremen environment

also caused by the context switches, and in this context, also by the need to switch processes when parsing incoming messages.

These metrics clearly show a distributed environment’s advantages, as the agents can use one thread for planning while other threads are free to handle external events. In contrast, on a single processor solution, the agents compete not only for threads for their planning but also for these side problems.

### 5.1.4 Runtime without planning

The last metric for ADPP is the time to convergence without the planning time. Such results shed light on the prolongation caused by message handling, trajectory serialization and deserialization, and other algorithm management runtime.
5.1 ADPP results

Figure 5.10. Average runtime without planning per agent for the Empty hall environment

Figure 5.11. Average runtime without planning per agent for the Warehouse environment

Figure 5.12. Average runtime without planning agent for the Ubremen environment
As can be observed, the plots 5.10, 5.11, and 5.12 closely resemble the runtime plots and planning, which points out the fact that neither planning nor the message handling/algorithm management is responsible for the overall prolongation but that both are victims of insufficient CPU resources, both for the CPU power-related prolongation of Distributed method and for the context-switching-related prolongation of the single processor environments.

### 5.1.5 Normalized Distributed method

In order to be able to compare the methods truly, the Distributed method’s results have also been normalized. The normalization factor has been computed according to the comparison of the used CPUs, as seen in Table 4.1. This comparison gives us a factor of 4.04, with which all the Distributed method results were divided. The resulting plots serve as a more precise evaluation of the algorithm scaling based on the deployment environment. In order to preserve some space in this section, only the results for Warehouse instances are shown, as the results demonstrate the effect of the normalization. However, all the other normalized plots for other environments can be seen in the attachments of Appendix B.

![Figure 5.15. Average runtime without planning per agent for the Warehouse environment with normalized Distributed method](image)

The normalization is imperfect, as it is only roughly derived from the proportion of the one-thread performance and applies the normalization to the total result. This, for example, results in the time spent waiting for convergence confirmation being normalized as a part of the total runtime duration. Even though this time depends on the environment’s performance, as it includes the planning time for other agents, this metric could be more precise. However, it is a good approximation of the possible experiment result in a distributed environment with the same CPU power as the other environments.

It can be noted that the normalization has helped the Distributed method tremendously, as its results now approximately correspond to the result of the original implementation. Even though the results suggest a speedup compared with the original implementation, this cannot be taken as an outcome of the experiment because of the above facts. Nevertheless, the most probable outcome is that the Distributed method would generate better overall results than any environments that use one processor and surpass the original implementation in instances with more agents.
Due to COBRA algorithm implementation, as discussed in section 2.2.2, the majority of the runtime of each instance is caused by the agent’s actual travel. This shifts the main focus of the measurements, apart from the planning time, to the prolongation between starting and completing a new task. This prolongation is caused by other agents’ trajectory interference and the time spent waiting for the token.

5.2.1 Runtime

The runtime can vary because the COBRA algorithm will not produce the exact solution for each instance, as the agents can ask for the token in a different order based on communication delays and will therefore start different tasks. This divergence can be seen in figures 5.16, 5.17, and 5.18. However, the approximate runtime remains the same and scales accordingly with the increasing number of agents for each instance.
5. Results and Analysis

This algorithm’s different CPU power and thread management problems are less apparent because the agents’ travel time diminishes the prolongation of planning and waiting for tokens. For this reason, COBRA results will not be normalized, as the results would change minimally.
5.2 COBRA results

![Graph showing Avg. time to solution vs number of agents for different COBRA methods.](image)

**Figure 5.18.** Average runtime of COBRA in Ubremen

### 5.2.2 Planning

The planning time for COBRA is relatively uniform for all the methods, with the exceptions of the Distributed environment, as is to be expected. The lower computational power of the Raspberry Pis influences the planning duration, but the overall results are similar for the rest of the methods. It can be noted how more challenging environments cause the result to spike with the increasing number of agents. The plots denoting this effect can be seen in figures 5.19, 5.20, and 5.21.

![Graph showing Avg. planning time vs number of agents for different COBRA methods.](image)

**Figure 5.19.** Average planning time in Empty Hall
5. Results and Analysis

5.2.3 Waiting for token

The figures 5.22, 5.23, and 5.24 show the agents’ average time spent waiting for the token. There are similarities between these plots and plots 5.19, 5.20, and 5.21, showing the average planning time. This is caused by the fact that other agents’ planning is the main reason an agent might have to wait for the token. Regardless, the number of agents does not significantly influence the average waiting time. The explanation for this behavior is the fact that racing for the token is a rare condition, as the agents spend most of their runtime traveling. This significantly reduces the possibility of all the agents asking for the token simultaneously. This condition does not occur even at the start of the algorithm, as each agent has a randomly generated waiting period before embracing a new task.
5.2 COBRA results

Figure 5.22. Average time waiting for the token in Empty Hall

Figure 5.23. Average time waiting for the token in Warehouse

Figure 5.24. Average time waiting for the token in Utrecht
5. Results and Analysis

5.2.4 Base task duration and prolongation

In this subsection, the results of planned trajectories are presented. These metrics are insignificant for this evaluation, as these results may vary due to COBRA’s non-deterministic behavior. As far as this thesis is concerned, the fact that none of these metrics spike significantly is satisfactory.

![Figure 5.25. Average base duration of a task in Empty Hall](image)

![Figure 5.26. Average base duration of a task in Warehouse](image)
5.2 COBRA results

These results are divided into three categories:

- **Base** — the base duration of the planned task computed as an absolute shortest path. These durations can be seen in figures 5.25, 5.26, and 5.27.

- **ProlongT** — the prolongation gained while traveling, both by waiting for other agents to pass a bottleneck or selecting a longer trajectory. This metric also accounts for the prolongation caused by the agent waiting during his planning window and the initial wait for the first task. These prolongations can be seen in figures 5.28, 5.29, and 5.30.

- **ProlongR** — the prolongation gained after arriving at the goal location. This can be caused by another trajectory planned earlier, which crosses the goal’s location. Said agent then has to wait on his goal and move aside for the passing colleague before settling on his goal. These prolongations can be seen in figures 5.31, 5.32, and 5.33.

**Figure 5.27.** Average base duration of a task in Ubremen

**Figure 5.28.** Average travelling prolongation of a task in Empty Hall

The base durations vary, which can be caused by the agents choosing different targets, and are essential for this thesis as long as every agent plans a trajectory successfully.
The travel prolongation results might be surprising, as even instances with a single agent seem to be prolonged. However, as mentioned above, this can be caused by the initial wait each agent is assigned. The travel prolongation never spikes significantly, which satisfied the condition that the result does not differ considerably.

![Figure 5.29. Average travelling prolongation of a task in Warehouse](image)

![Figure 5.30. Average travelling prolongation of a task in Utrecht](image)

The resting prolongation figures show that some agents had to deal with trajectories planned over their goal. The frequency and prolongation caused by this are reasonably exact across the instances, signifying the correctly working COBRA.
5.2 COBRA results

**Figure 5.31.** Average resting prolongation of a task in Empty Hall

**Figure 5.32.** Average resting prolongation of a task in Warehouse

**Figure 5.33.** Average resting prolongation of a task in Ubremen
5. Results and Analysis

5.3 Analysis and summary

The experiment has produced different results for each algorithm. The measurements were limited by the threshold of 15 agents, even though this was enough to show some interesting behavior, with the likelihood of diverging more results for higher agent instances.

The COBRA algorithm has shown that introducing a planning Token de facto serializes the algorithm runtime, allowing only one agent to plan. This approach’s results are very similar across all tested environments, with even the CPU performance differences being annulated by the wait duration caused by the agents traveling. Regardless, the non-deterministic behavior induced by the communication layer causes the algorithm to differ from the original implementation and causes a communication delay in token passing compared to the original token shared in a critical section. This behavior can explain the spikes and abnormalities in COBRA’s results, as the tasks are not pre-generated for each agent but randomly assigned to the agents. Even though this randomized assignment is synchronized across the network, the fact that any agent can ask for a new task slightly earlier than it would be using another method can result in another task being assigned.

The ADPP algorithm, on the other hand, has produced significantly different results from the original single-process implementation, with significant delays measured. This difference can be observed in an implementation that correspondingly uses one processor but uses TCP as a communication device, creating an inter-process distributed environment instead of a simulated one. The instances run on a truly distributed system environment generated results comparable to the original in terms of speed when normalized. Furthermore, the experiment shows that these instances might surpass the original simulation, which is limited by agents competing for resources, which will most likely be even more apparent for a higher number of agents.

Overall, the results suggest that the best option for testing distributed algorithms is a distributed simulation running on a distributed environment or a full simulation with messages being passed using shared memory. This suggests that the delay the distributed environment gains on the communication is balanced by the possibility of multiple CPUs cooperating on the computations. However, running a distributed algorithm, implemented on a distributed network, on a single CPU, with messages over a real network, produces a significant prolongation and is not an ideal implementation possibility.
Chapter 6
Conclusion

The assignment of this work has been successfully fulfilled, and all assignment points have been accomplished.

Firstly a thorough analysis of the Picocluster platform for distributed computing on a Raspberry Pi cluster has been conducted, and an extension to this platform was proposed and implemented, which allowed running the platform on different deployment settings, such as the original cluster or a personal computer while being allowed to select the number of CPUs.

Additionally, two distributed coordination multi-agent algorithms were integrated as tasks into the platform.

An experiment measuring various metrics for these algorithms was then proposed and implemented, and an extensive experimental evaluation of the scaling of these algorithms on all available deployment settings was conducted.

The experiment’s results have shown the disadvantages of distributed environments regarding messaging prolongation and the enormous advantages of distributed computing on multiple processors. Furthermore, it roughly points the direction for selecting the optimal environment for testing distributed algorithms. It clearly shows that there are better methods than starting standalone nodes of a distributed network, which is otherwise capable of running on a truly distributed environment.
References


[6] Intel Core i7-8565U @ 1.80GHz vs ARM Cortex-A72 4 Core 1800 MHz [cpubenchmark.net] by PassMark Software. https://www.cpubenchmark.net/compare/3308vs4078/Intel-i7-8565U-vs-ARM-Cortex-A72-4-Core-1800-MHz.
Appendix A

Project structure and usage

The project is a mixture of three different subprojects, which are bundled inside the larger root project of this project. The three subprojects are located in the project directory of the root project:

- **adpp-journal** — a custom fork of the original repository of Dr. Čáp, containing the sources for the ADPP algorithm along with many others.
- **cobra-icaps2015** — a custom fork of the original repository of Dr. Čáp containing sources for the COBRA algorithm and an implementation of the ORCA algorithm.
- **picocluster** — a custom fork of a subtree of Roman Janků’s repository containing his bachelor thesis.

A.1 Version manager

The whole project uses Git as a version manager. The root repository contains sources for both the thesis, located in the thesis directory, and the project itself, located in the source directory. The project folder can, however, seem empty, as this directory’s only content is in the form of submodules. This approach allows the user to bundle three repository forks for each subproject listed above, which enables proper history management and content separation.

The following command allows cloning the whole project, including the submodules.

```
git clone --recurse-submodules git@gitlab.fel.cvut.cz:rokytmar/bachelor-thesis.git
```

Additionally, the following commands can update the content of the submodules at any time.

```
git submodule init
git submodule update
```

Both the ADPP and COBRA repositories are standard git Forks. However, the Picocluster’s repository forking has proven to be quite challenging, as the repository also contains all of the sources for Roman Janků’s bachelor thesis. I created a subtree of this repository, which only contains the folder with the desired application. The resulting forked repository, therefore, contains only desired folders, and the history is filtered to contain only commits that affect this subtree. The forked subtree repository is still linked to the upstream original, and merges can occur to exchange commits and git flow between the upstream and the fork. Nevertheless, the user can work with all three submodules as with standard git repositories, with the only addition being the need to commit the submodule changes to the root project.
A.2 Maven structure

Another common feature of all three subprojects is the usage of Maven as a dependency manager and packaging tool. This fact was used to bundle the projects further, creating a root Maven project. This project serves only as a parent wrapping all the other modules, allowing the whole project to be packaged from the source folder by running the `mvn package` command, which will create .jar files for all submodules.

The code snippet below provides a preview of the parent module’s pom.xml

```xml
<groupId>cz.cvut.fel.rokytemar.picocluster-mapp</groupId>
<artifactId>root</artifactId>
<packaging>pom</packaging>
<version>0.1</version>
<modules>
  <module>project/picocluster</module>
  <module>project/cobra-icaps2015</module>
  <module>project/adpp-journal/admap-solver</module>
</modules>
```

The Picocluster module further divides into three modules, one for each the node, the server, and the library.

Additionally, the ADPP and COBRA modules were added as dependencies to all the modules of the Picocluster in order to allow the platform to access all respective packages needed to integrate the algorithms. This also allows the `node-0.9-jar-with-dependencies.jar` executable to carry all dependencies needed for any of the algorithms’ run, thus reducing the number of files needed for a deploy to only one.

A.3 Running the algorithms and experiment

In order to run either any of the algorithms or the experiment, we must first package the project using `mvn package` command and then start the server and the desired amount of nodes.

A.3.1 Starting the project and algorithms

To launch the server, we can simply run the `start_server.sh` script located in the root of the project’s source directory. The node has multiple possibilities for launching. We can either start the node on a cluster environment, which requires a machine with a Blinkt! module connected to its GPIO header, or on any available machine.

Either way, there are three possible methods how to start the node. The first includes launching the `start_node.sh` script in the source root. This is by far the easiest method, and all required environment variables can be set in the script. Ensure that the `PICOCLUSTER_SERVER` variable is set to correct the IP and port the server is running on.

Another option is to use Docker. There are two available Dockerfiles, called `Dockerfile` and `Dockerfile-noAnsible`. The Dockerfile is, however, tailored specifically for the use on the cluster, using `rpi-base-image` as a base for the docker image and creating the image in an environment prepared by
A.3 Running the algorithms and experiment

_ansible_ deployment pipeline created by Roman Janků. Therefore, the usage of _Dockerfile-noAnsible_ is suggested.

The environment variables are specified in the docker file, and the node can be created by this command:

```
sudo docker build . --file Dockerfile-noAnsible -t picocluster-node
```

Subsequently, this command can be used to run the node:

```
sudo docker run picocluster-node
```

Furthermore, the docker allows us to specify threads on which the node will be allowed to run. To achieve this, use a command similar to the following one, with the custom specification of the _cpuset_:

```
sudo docker run --rm --cpuset-cpus="1-4" picocluster-node
```

Lastly, the already mentioned method using Ansible can be used. To launch the node with this setup, ensure your Raspberry Pi’s IP address is listed in any group of the _hosts_ file located in the source root. Ansible can be used to create various pipeline tasks on all specified machines. These tasks are located in the source root and are named _full-deploy.yml_, _deploy.yml_, _restart-docker.yml_ and _redeploy-docker.yml_. The effect of the tasks is apparent from their name.

To start any of these tasks, use the following command:

```
ansible-playbook -i ./hosts -kK --ssh-extra-args='-o "PubkeyAuthentication=no"' -u picocluster deploy.yml
```

These scripts can build the docker image from scratch, or just redeploy it, and will always start the node right at the end of the task.

Starting any of the algorithms is then relatively easy, as it only requires a sufficient amount of nodes to be connected and the arguments inputted correctly. However, the user can also provide no input, and the server will use default arguments.

### A.3.2 Starting the experiment

Starting the experiment is even easier than starting one node, as there is only one parameter to be inputted. There are however some other problems, a user must know of.

The experiment runs all the desired algorithms on all instances it can find. If there are no instances generated, start the _prepare-picoagents-experiment.sh_ script in the _adpp-journal_ folder for ADPP, or in the _cobra-icaps2015_ folder for COBRA.

We must only be aware, to start the experiment task using all possible agents connected, to prevent multiple instances trying to run at once. Furthermore, starting the visualization slows down the experiment, and the visualization cannot be shut down, as it would shut down the entire _Server_.

The results of the experiment will be stored in _data.out_ file in each instance environment folder. Subsequent usage of _prepare-picoagents-results.sh_ will result in _data.out.head_ file being produced, containing the result data with a header, and usage of _prepare-picoagents-plots.sh_ will generate plots from these data in _plots_ folder.
Beware that running the experiment will always rewrite the whole output file, so in order to join the results of more experiment runs, we create backups and combine the files later using an editor or terminal.

A thorough manual for launching the project on the cluster can be found in Appendix A of Roman Janků’s bachelor thesis.
All additional attachments have been uploaded to the KOS submission system and are also available in the project’s GitLab repository\(^1\). There are the following attachments:

- all the plots generated using the experiment results for both algorithms in all environments. These can be found in the `plots` folder of the uploaded attachments or at
  ```
  ```

- the experiment results in a raw format. The data can be easily imported as a CSV, and are located in the `results` folder of the uploaded attachments or at
  ```
  ```

- the project’s complete source code can be found in the `source` folder of the uploaded attachments or at
  ```
  ```

---

\(^1\) [https://gitlab.fel.cvut.cz/rokytmar/bachelor-thesis](https://gitlab.fel.cvut.cz/rokytmar/bachelor-thesis)