INSTRUCTION SCHEDULING FOR BLIND QUANTUM COMPUTING

Bob Dorland
Student ID: 4653327

Thesis advisor: Stephanie Wehner
Daily supervisor: Emir Demirović
Daily co-supervisor: Bart van der Vecht

Master Thesis
Computer Science programme, Software Technology track
Faculty of Electrical Engineering, Mathematics and Computer Science
Technical University of Delft

Defended on 2 February 2023

Contents

Abstract

Quantum networks offer more capabilities than classical networks. For example, quantum networks can solve certain problems faster than classical networks and they can even solve problems which cannot be solved with classical networks. One well-known quantum network application is blind quantum computing (BQC). In a BQC application a client node sends a computation to a server node in the network. The server node performs this computation without knowing the details about the actual input or computation being performed and returns the result of the computation to the client node. Quantum applications are often repeated many times, due to randomness that is involved in the result of executing a quantum application. When a server node performs BQC repeatedly with multiple client nodes, there follows an interesting problem how all corresponding instructions can be scheduled for the server optimally, given that certain types of instructions referred to as entanglement instructions can only be scheduled at given moments by a so called network schedule. One classical metric used to assess the quality of a schedule is makespan, which is the time that it takes to complete all instructions. A quantum oriented metric involved is success probability, which indicates what fraction of executions yield a desired result when repeatedly executing an application. In this thesis, an experimental demonstration is given of the use of constraint programming (CP) for the scheduling of instructions of quantum network applications. This demonstration focuses on scheduling instructions that the server node should execute when performing BQC repeatedly with a small number of client nodes at the same time. Different CP models are presented that assign start times to tasks that the server node should execute. By comparing the CP models to a baseline scheduler that schedules tasks as soon as possible, it was found that CP can be used to reduce the makespan when BQC is performed by the server node with multiple clients at the same time, while preserving a similar success probability. A main drawback that followed from the CP approach is scalability. Future work is to be done on studying how CP can be used for larger quantum networks.

Acknowledgements

First, I would like to express special thanks to the supervisors that were involved during this project. Stephanie Wehner offered me the opportunity to do my thesis project at QuTech, allowing me to learn more about quantum networks. She also assisted me with feedback during the project. The second supervisor that I would like to thank is Emir Demirović. He also assisted me with feedback during the process and also helped me with structuring my thesis. The third supervisor is Bart van der Vecht. The weekly meetings that I had with him helped me during the course of the project. Furthermore, he assisted me in preparing the Qoala simulator for performing my experiments.

I would also like to thank Menno Veldhorst, for being part of the thesis committee next to the three supervisors, as an external expert.

Furthermore I would like to thank the people at the Algorithmics research group who provided me with feedback after the presentation that I gave there to present my early progress.

Finally, I would like to thank the rest of the Wehner group at QuTech. During group meetings I was able to have conversations with fellow group members to learn about projects that they are working on.

Abbreviations

- **BQC:** blind quantum computing
- **CC:** classical communication
- **CL:** classical local
- **CP:** constraint programming
- **EDF:** earliest deadline first
- **IF:** interleaving free
- **IS:** interleaving strict
- **QC:** quantum communication
- **QL:** quantum local
- **QoS:** quality-of-service
- **RCPSP:** resource-constrained project scheduling problem
- **SF:** sequential free
- **SS:** sequential strict

1 INTRODUCTION

Quantum networks offer serious advantages over classical networks. For example, they can offer great speed-ups or they can provide solutions for problems which cannot be solved with classical networks [1] [2] [3]. Given the potential that quantum networks have to offer, these networks provide an interesting research field.

Different applications can be executed in quantum networks, of which one well-known application is blind quantum computing (BQC) [4] [5]. When executing a BQC application, a client node in the quantum network sends a computation to a server node which the server node blindly performs (not knowing about the input or the actual computation being performed), after which it returns the result to the client. It is the case that there can be many client nodes performing BQC applications with the same server node and these BQC applications can also be repeated many times, as it is usual to execute quantum applications repeatedly, due to uncertain outcomes. From this, there follows an interesting problem how all instructions that the server node needs to perform can be scheduled, while meeting a given set of constraints and optimising both classical and quantum oriented performance metrics. A classical metric that is considered is makespan, which is the total amount of time that it takes to finish all instructions. An important quantum oriented metric is success probability, which is defined as the fraction of application executions that yield a desired outcome, when repeatedly executing a quantum application.

Experimental realisations of quantum networks exist at QuTech [6]. These quantum networks are able to compile network applications and execute instructions sequentially. These applications can consist of both local operations on qubits in the nodes and network operations between nodes in the network. A special type of network operation is entanglement. Entanglement is performed between two nodes in the network, which should trigger this operation at the same time. A network schedule is used to indicate when which pair of nodes performs entanglement. In the current realisations, there is idle time when the next instruction to execute is entanglement for a pair of nodes for which it is not the turn yet in the network schedule to do so.

Since instructions are always scheduled sequentially in the realised quantum networks, there is no flexibility in re-ordering instructions. The lack of flexibility for instruction scheduling can have negative effects. For example, when a pair of nodes needs to wait to perform entanglement, no other instructions can be done in the meantime. This leads to a higher makespan. Especially when a server node performs BQC applications repeatedly with multiple clients, for which entanglement is needed, the makespan becomes even higher.

In this thesis, an experimental demonstration is given of the use of constraint programming (CP) for the scheduling of quantum network applications. This is done by showing how CP can be used specifically for scheduling instructions for the server node that performs BQC applications repeatedly with a small number of clients at the same time. The types of instructions in BQC are general types that can be part of any quantum network application, which means that the concepts of the case study in this thesis could be further used for other kinds of quantum network application scheduling problems.

Four different CP models are presented that assign start times to instructions that a server node in a quantum network needs to execute when performing BQC applications repeatedly with multiple clients in the network. Schedules created by using these CP models satisfy a set of constraints and directly minimise the makespan, as minimising the makespan is the objective function in the models. The CP models do not directly have a notion of the application success probability, and hence do not directly maximise this probability. Instead, the CP models have ’execution deadlines‘ as one of their constraints. The idea is that a smart application compiler provides suitable deadlines such that the application success probability is maximised. Our scheduler, which respects these deadlines using our CP models, hence indirectly enables success probability maximisation. The CP models are evaluated by comparing them to a baseline scheduler which iterates over all tasks that the server should perform and schedules them as soon as possible. As tasks are scheduled as soon as possible for the baseline scheduler, this provides an upper bound for the success probability, given that the success probability increases as the execution time decreases, given qubit degradation over time. The CP models are evaluated based on two criteria. First, the makespan should be lower for the CP models than for the baseline scheduler, to verify that CP can indeed help to generate schedules which complete all tasks faster. Secondly, the CP models should yield a success probability that is similar to the upper bound provided by the baseline scheduler, in order to investigate that the makespan can be improved while still meeting the best possible success probability provided by the baseline scheduler.

By enabling the opportunity to re-order instructions that a server node has to execute when performing BQC applications with different clients, these applications can be executed more efficiently. Waiting times to perform entanglement with a specific client node can be used to execute other instructions, such as performing entanglement with other clients, following the network schedule. Having the opportunity to re-order instructions allows the server to interleave instructions that are done for different clients.

For two out of the four implemented CP models, the SF and IF models, it was found that the makespan of the server node can be significantly decreased in the case of multiple clients invoking BQC compared to the baseline scheduler, while preserving a similar success probability. For the remaining two models, the SS and IS models, it was found that they can find solutions quicker than the SF and IF models and improve the makespan compared to the baseline when there are enough clients in the network. However, there is a loss in success probability for these models compared to the baseline and the SF and IF models. One main drawback found with CP is scalability. As the size of the problem input increases, it becomes harder to solve the problem in a feasible amount of time.

Main contributions. The main contributions in this thesis are as follows: four CP models are presented that assign start times to tasks that a server node in a quantum network should execute when repeatedly performing BQC applications with multiple client nodes in the network. These CP models are evaluated by comparing them to a baseline scheduler, based on the performance metrics *makespan* (objective value that the CP models minimise) and *success probability*.

This thesis is further structured as follows: first, preliminary information is given in section 2. This is followed by a definition of the scheduling problem under study and the introduction of the implemented CP models in section 3. Next, related work is discussed in section 4. Section 5 presents the experimental results. Finally, section 6 concludes this research.

2
Preliminaries

In this section, different concepts are explained which are further used in this thesis.

2.1. Qubit

In quantum computing, a qubit is a basic unit of quantum information. Whereas classical bits only have a single state of 0 or 1, qubits combine these two states, meaning that qubits are represented by a linear combination of 0 and 1. Qubits degrade over time and when performing operations on them. The process of qubits degrading is called decoherence. Decoherence of qubits may have a lower application success probability as a consequence, since it can affect the result of executing an application negatively.

2.2. Quantum network

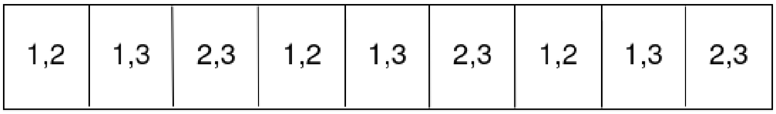
A quantum network consists of nodes, which are connected by channels. Nodes in a quantum network contain a quantum memory, consisting of qubits. There are four types of operations that nodes in such a network can perform. First, there are classical local (CL) operations. These are operations that can be performed on classical operation systems (for example Linux or Windows) such as simple additions. There are also quantum local (QL) operations. These operations can be quantum gates such as rotations, and measuring the value of a qubit. Next, there are classical communication (CC) operations. This involves the sending of classical messages between nodes in the network. Finally, there are quantum communication (QC) operations, representing entanglement operations between nodes in the network. Nodes in the quantum network can perform all kinds of applications containing these types of instructions.

2.3. Entanglement

Entanglement can be performed between two nodes in a quantum network, resulting in two entangled qubits, where one qubit is in one node and the other qubit is in the other. When two qubits are in an entangled state, they cannot be described independently. This means for example that a measurement of a qubit in one of the two nodes automatically influences the qubit in the other node. Entanglement is timed according to a network schedule.

2.4. NETWORK SCHEDULE

Entanglement generation happens according to a network schedule, which we assume is global (all nodes see and follow the same schedule). An example network schedule is illustrated in Figure 2.1. For simplicity it is assumed that only one pair can generate entanglement at a time. A network schedule is needed because both nodes in a pair that performs entanglement need to do an entanglement attempt at the same time, so a network schedule can provide information about when this should happen. Also, it might be the case that two nodes which generate entanglement are not directly connected via channels in the quantum network. In that case, the entanglement generation between these two nodes should go via the nodes in the network that are on the path of channels between these two nodes. These nodes which connect the two nodes performing entanglement should be ready to do so. Thus, the network schedule informs the whole quantum network when it is the turn of which pair of nodes to perform entanglement.



1,2	1,3	2,3	1,2	1,3	2,3	1,2	1,3	2,3

Figure 2.1: The network schedule consists of timeslots which contain a pair of nodes indicating that during that timeslot it is the turn of that pair of nodes to perform entanglement. In this example, there are three nodes in the quantum network. The network schedule repeats a sequence of node pairs as long as needed. In this illustration, that sequence is that nodes 1 and 2 are first scheduled. This is followed by nodes 1 and 3. Last in the sequence are nodes 2 and 3. In this example, the sequence is then repeated twice. It is assumed in this thesis that the timeslots in the network schedule are of constant length.

2.5. BLIND QUANTUM COMPUTING

Blind quantum computing (BQC) [4] [5] is an application performed between two nodes in a quantum network. When executing a BQC application, one node in the network acts as a server node and another node acts as a client node. The idea is that a client node can submit a computation to the server node, which the server node would then compute blindly. This means that the server node does not know anything about the input or the actual computation being performed, meaning that the computation can be securely performed. BQC starts with the server node and the client node both triggering entanglement synchronously one or multiple times. Then, for every entanglement link that is generated between the client and server nodes, both the server and client nodes perform their own instructions. First, local operations are performed on the client node, resulting in a value for a rotation angle. This rotation angle is sent to the server node via classical messaging by the client node. After receiving the angle value, the server node performs local operations with this value and performs a measurement with the transformed value. Finally, the server node sends the measurement outcome to the client node via classical messaging.

2.6. BQC-CLIENT AND BQC-SERVER APPLICATIONS

During the execution of a BQC application, the server and client nodes both perform their own list of instructions. These are referred to as a server performing a BQC-server application and a client performing a BQC-client application. For both of these applications the instructions are listed in Figure 2.2. In this thesis, the problem involves the scheduling of BQC-server applications. It is assumed that client nodes perform their instructions when needed.

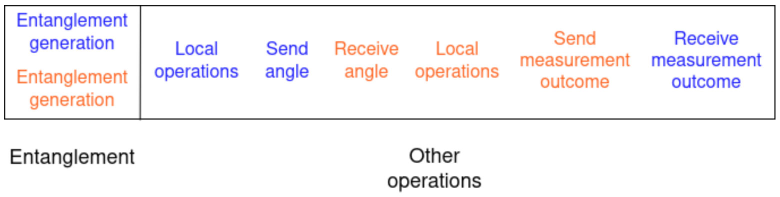


Figure 2.2: Timeline of operations in a BQC application, with left the first operation and right the last operation. The operations in blue are performed by the client node (part of the BQC-client application) and the operations in orange are performed by the server node (part of the BQC-server application). It can be seen that the BQC application is split into two parts. The first part involves both nodes performing entanglement. They do so by synchronously doing an entanglement attempt, for that reason the client operation is placed above the server operation in the timeline. Entanglement generation is done one or multiple times. The second part consists of the other operations that are performed individually for every generated entanglement link. First, the client performs local operations in order to calculate a rotation angle and this value is then sent to the server. The server does local operations with this angle and then performs a measurement of which the outcome is sent back to the client.

2.7. SUCCESS PROBABILITY

In quantum computing, applications are often executed many times. This is the case, because quantum applications have uncertain outcomes, for example because of decoherence. The success probability is defined as the fraction of application executions that yield a desired value out of all executions of a particular application. The success probability is used as a quantum performance metric in this thesis. When qubits degrade much for an application execution, the chance of a desired outcome decreases.

2.8. MAKESPAN

Makespan is a classical metric that is used for scheduling problems. The makespan indicates how long it takes to finish a set of tasks that have to be completed. Makespan is used as a classical performance metric in this thesis.

2.9. RCPSP

The resource-constrained project scheduling problem (RCPSP) is a well-known scheduling problem. The idea is that there is a set of activities that needs to be scheduled.

Each activity has a duration and cannot be interrupted. There are also precedence relations between pairs of activities, meaning that the second activity in the pair can only be started after the first activity was finished. Furthermore, there is a set of renewable resources. Each resource has a maximal amount that can be used of it at a time. Each activity has a demand for every resource, which can also be zero.

The goal is to schedule activities in such a way that the makespan is minimised, while meeting precedence relations and never exceeding the resource capacities.

2.10. HEURISTICS

Heuristic algorithms are used to find solutions to a problem faster when other solving methods such as performing brute force search (simply iterating over all possible solutions) or constraint programming do not find solutions in a feasible amount of time. Heuristic algorithms focus on performing approximations rather than finding exact solutions, in order to find a possible solution in a shorter amount of time.

2.11. CONSTRAINT PROGRAMMING

Constraint programming (CP) is a technique used to find solutions to a problem, where a valid solution would meet a set of conditions (constraints). With the use of constraint programming, one can make a CP model that contains the following elements:

- Input parameters: this is a set of variables which form the input to the model. These variables are assigned a value for the particular instance of the problem to solve.
- Decision variables: this is a set of variables to which values are assigned during the solving process. A solution to a constraint problem would be a found assignment of values to the decision variables. In the context of this thesis, the decision variables would be the start times for the tasks to schedule for the server node performing BQC applications in a quantum network.
- Constraints: this is a set of clauses which should hold for potential value assignments to decision variables. A solution is valid if all these clauses hold for a given assignment.
- Objective function: this is a function which maps the values assigned to decision variable to some other value, which is the objective value. The objective function should be minimised or maximised, depending on the problem. Thus, an objective function is used to find optimal solutions, in case of an optimisation problem. For satisfaction problems, where the goal is to only find a solution that meets all constraints, an objective function is not needed. In the context of this thesis, the objective function would be to minimise the makespan.

One well-known constraint programming language is that of MiniZinc [7], which is also used for this research project. In MiniZinc, one can implement a constraint model and run it with a given solver back-end. Different solver back-ends use different methods in order to find solutions to a given problem.

2.12. GLOBAL CONSTRAINT

MiniZinc offers a range of global constraints to the programmer. These are constraints that are already built-in and that can be called when making a constraint model. In this way, the programmer does not need to implement the constraint on their own. When there exists a global constraint that implements a constraint that is needed for a model, it is good practice to use it. The use of global constraints makes the model easier to read and understand and it can improve the solver performance.

2.13. Symmetry-breaking constraint

It can occur that there is symmetry a constraint model. This means that there can be multiple solutions, yielding a same objective value, while being permutations of each other. As an example, think of the n-queens problem, where n queens should be placed on a n x n size chess board. The n queens should be placed in such a way that no two queens share a row, column or diagonal on the board. Here, there would be symmetry, as one could rotate the board and find essentially the same solution. Symmetry-breaking constraints break this symmetry by enforcing some kind of order, depending on the problem under study. This means that there would be less possible solutions to explore, making problems easier to solve. MiniZinc has support for implementing symmetry-breaking constraints.

2.14. Redundant constraint

Just as global constraints and symmetry-breaking constraints, redundant constraints can also be used for optimising a CP model. Redundant constraints are constraints that logically follow from other constraints, thus they do not constraint the model any further. Redundant constraints can be used to give the solver more information early on during the solving process, which may improve performance. MiniZinc has support for implementing redundant constraints.

2.15. Propagator

In constraint programming, a propagator is used to reduce the search space of the decision variable domains. During the solving process, various values are assigned to the decision variables to see if that combination of assignments yields a valid solution. With the use of a propagator, value assignments to decision variables that cannot produce valid solutions are removed from the search space, possibly giving faster solving times.

2.16. Search annotation

In MiniZinc, it is possible to include a search annotation in the constraint model. This search annotation determines the search strategy of the solver. The search annotation consists of two parts. First, there is a variable selection strategy. This determines which variables to try to assign values to first. For example, one can choose to search variables with the smallest domain first. There is also the value selection strategy, which determines which value to assign to the given variable. An example value selection strategy is to assign the smallest possible value first. The use of search annotations can make it easier to find solutions for the solver, as the programmer is able to give the solver hints on how possible solutions in the solution space should be searched.

3 Problem definition

In this section, the underlying scheduling problem is defined. First, a brief introduction is given of the problem. This is followed by a formal description. Next, four different variants to the problem are discussed. Furthermore, the constraints are introduced that solutions to the problem should adhere to for the different problem variants. Finally, the four constraint programming models are discussed which were implemented for the problem variants.

3.1. Introduction

When a client node in a quantum network invokes a server node in the network to perform a BQC application, the client and server nodes perform their own applications, referred to as the client performing a BQC-client application and the server performing a BQC-server application. This is visualised in Figure 3.1. The focus of this project is on scheduling BQC-server applications. A BQC-server application contains various instructions which include performing entanglement with the client to establish a connection, receiving input from the client, performing a computation with this input and sending the computation result back to the client. A server and client node can execute the same BQC application multiple times, as executing quantum applications involves random results and thus applications are often repeated to get a desired result. The server repeats executing its corresponding BQC-server application when a BQC application is repeated with a client. As there can be different client nodes that perform BQC applications with the server node repeatedly at the same time, there can be many BQC-server instructions that need to be scheduled. The problem under study involves assigning start times to all tasks that the server has to perform such that a list of constraints is satisfied, where a task is defined as a single execution of an application instruction. In general, a low makespan and a high success probability are desired. The problem is modelled as a constraint model, where the makespan is directly minimised as the objective function. The success probability is not directly part of the constraint model as this is complex to model, but this is linked to a constraint which involves application deadlines. Tighter deadlines mean that qubits are in memory for a shorter amount of time, giving qubits less time to degrade. This helps to increase the success probability, which is experimentally analysed in simulation. The main challenge involves finding a schedule for BQC-server applications that the server node should execute given that entanglement (QC) operations can only be performed with the respective client during given timeslots as indicated by the network schedule, while meeting all other constraints and minimising the makespan. Note that the duration of a QC task can be shorter than the length of a timeslot in the network schedule, meaning that other instructions might also be scheduled during these timeslots, as long as the QC tasks are scheduled during the assigned timeslots. It is assumed that the client nodes perform all their own instructions when the server node requires this, so the instructions of BQC-client applications themselves are not considered in the problem description. Four different problem variants are considered, which depend on the type of network schedule used and the execution order for tasks that belong to a same BQC-server application.

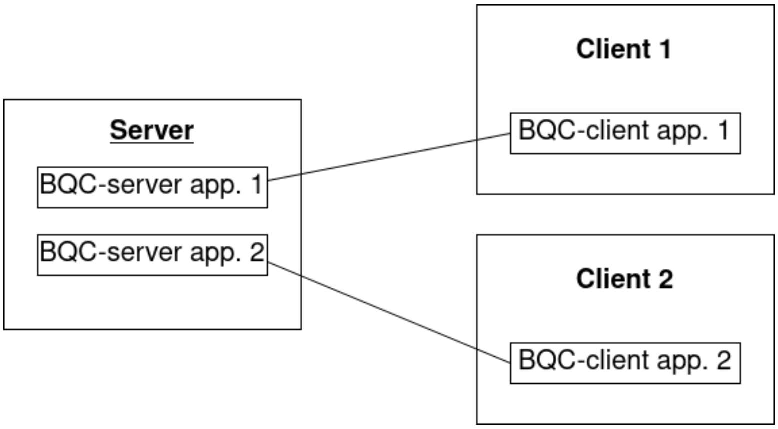
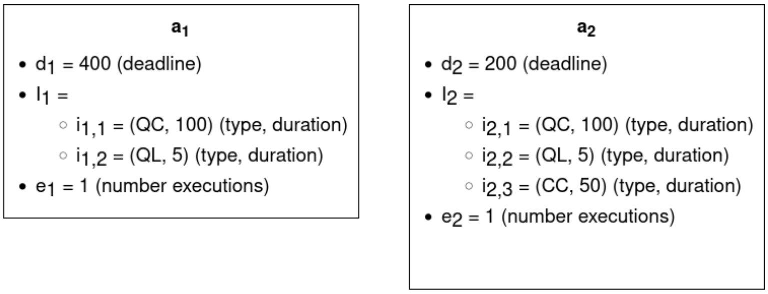


Figure 3.1: Visualisation of the server node performing a BQC application with two clients: clients 1 and 2. It can be seen that every client performs their own BQC-client application. The server node performs a BQC-server application for every client node. The corresponding BQC-server and BQC-client applications are connected. This project focuses on the scheduling of the BQC-server applications for the server node, hence the title of the server node is underlined.

3.2. FORMAL DESCRIPTION

The set $A := \{a_1, .., a_n\}$ represents BQC-server applications that need to be scheduled. Each application a_j (where $j \in \{1, .., n\}$) is a 3-tuple, which is denoted as (D_j, I_j, e_j) where D_j is the maximal run time of application a_j in μs represented as an integer. The ordered list I_j represents the instructions that should be scheduled for a single execution of application a_j. The integer e_j is the amount of times all instructions in the instruction list I_j should be executed. The list I_j is denoted as $I_j := (i_{(j,1)}, .., i_{(j,m_j)})$, where the integer m_j represents the number of instructions of application a_j. Each instruction $i_{j,k}$ (where $j \in \{1, .., n\}$ and $k \in \{1, .., m_j\}$) is represented by a 2-tuple. This tuple is denoted as $(r_{(j,k)}, d_{(j,k)})$. The string $r_{j,k} \in \{CL, CC, QL, QC\}$ represents the type of instruction $i_{j,k}$ and the integer

$d_{j,k}$ is the maximal time instruction $i_{j,k}$ takes to complete in μs. Figure 3.2 visualises an example of two applications with a small list of instructions.



a_1

- d_1 = 400 (deadline)
- I_1 =
  - $i_{1,1}$ = (QC, 100) (type, duration)
  - $i_{1,2}$ = (QL, 5) (type, duration)
- e_1 = 1 (number executions)

a_2

- d_2 = 200 (deadline)
- I_2 =
  - $i_{2,1}$ = (QC, 100) (type, duration)
  - $i_{2,2}$ = (QL, 5) (type, duration)
  - $i_{2,3}$ = (CC, 50) (type, duration)
- e_2 = 1 (number executions)

Figure 3.2: Two BQC-server applications are given that should be executed, applications a_1 and a_2 respectively. Both applications consist of a list of instructions, lists I_1 and I_2 respectively. List I_1 contains two instructions of types QC and QL. The durations are 100 and 5 μs respectively. List I_2 consists of three instructions of types QC, QL and CC. The durations are 100, 5 and 50 μs respectively. Every execution of application a_1 should be finished within 400 μs after it starts and every execution of application a_2 should finish within 200 μs after it starts. For this example, both applications are only executed once.

Also, an ordered list $G := (g_1, .., g_\infty)$ is given, which represents the network schedule. Each integer $g_b \in \{1, .., n\}$ (where $b \in \{1, .., \infty\}$) is the index of the BQC-server application for which entanglement (QC) tasks can be scheduled during timeslot b, so when it is the turn of the server node and the corresponding client node BQC is performed with. Network schedule G is formed by infinitely repeating a sequence of application indices of length P. It is assumed that in this sequence of application indices that is repeated, every application index occurs only once. Thus, after it was the turn to perform QC operations for a certain application a_j, this application has to wait $P-1$ timeslots in G before it can perform another QC operation. All timeslots have the same duration in μs, defined by the integer L. In Figure 3.3, an example is shown what a network schedule could look like, given the two example applications in Figure 3.2.

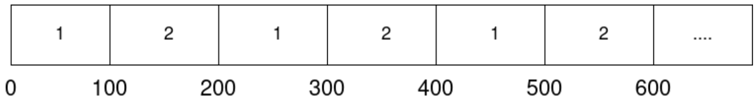


Figure 3.3: An example network schedule which assigns timeslots to the two example applications a_1 and a_2 for the executions of QC tasks. The timeslots contain the application indices. For example, between 0 and 100 it is the turn of application a_1 to perform QC tasks and between 100 and 200 it is the turn of application a_2. When it is the turn of a BQC-server application in the network schedule, only the client corresponding to the BQC application can perform entanglement with the server node. The sequence that is repeated in the network schedule here is that application a_1 can first perform QC operations, followed by application a_2. Thus, the sequence length $P = 2$ in this case. The timeslot length L is defined as 100 μs.

The integer $s_b := L \cdot (b-1)$ is the start time of timeslot b. The integer w_j is the index of the first timeslot in the network schedule where it is the turn of application a_j to perform QC tasks.

The input can be mapped to a set of tasks, where for each instruction $i_{j,k}$, there exist e_j tasks, where e_j is the number of times application a_j is executed. Thus, tasks represent the instructions that the server node repeatedly executes. A task is denoted as $t_{j,k,h}$, which indicates that instruction $i_{j,k}$ is performed for the h-th time, where $h \in \{1,..,e_j\}$. The goal is to create a schedule which indicates when each task should be performed. A solution to the problem would be a set of start times for every task, S. An entry in the set S is represented as $s_{j,k,h}$, represented by an integer, being the start time of task $t_{j,k,h}$ in μs. The integer T represents the makespan in μs, which represents the end time of the task that finishes last in the resulting schedule S. T should be minimised as the objective function. The integer $f_{j,h}$ represents the time that it takes to complete the h-th execution of application a_j in μs in the resulting schedule S (difference between the end time of the last task and the start time of the first task of the application execution). The ordered list $Q_j := (q_{(j,1)},..,q_{(j,v_j)})$ represents the QC tasks that have to be executed for application a_j, where integer v_j represents the total number of QC tasks for application a_j and $q_{j,z}$ (where $z \in \{1,..,v_j\}$) represents QC task z to schedule for application a_j. A possible schedule for the two example applications is given in Figure 3.4. The symbols used in this section are also listed in Table 3.2.

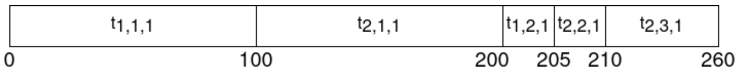


Figure 3.4: A possible schedule for the example applications a_1 and a_2 in Figure 3.2. In total, there are five tasks to schedule. This is, because for application a_1 there are two instructions that are performed once and for application a_2 there are three instructions that are performed once. Important to note are the positions of the QC tasks for both applications, tasks $t_{1,1,1}$ and $t_{2,1,1}$. These tasks are scheduled according to the timeslots in the example network schedule in Figure 3.3. As from 0 to 100 it was the turn of application a_1, its corresponding QC task is scheduled during that timeslot. For application a_2 this is the case from 100 to 200. The other three tasks can simply be scheduled afterwards, since all task types other than QC do not have specific timeslots during which they can only be executed. The makespan $T = 260\mu$s for this instance, since at that time all tasks are finished that the server node should execute.

3.3. VARIANTS

Four different variants to the given problem are studied. These problem variants differ in two aspects. Firstly, two variants of the network schedule are considered. The second aspect involves the order in which multiple executions of a same application are performed.

For the network schedule, there are the following two variants, which are also visualised in Figure 3.5:

- The free network schedule. Here, entanglement operations *can* always be triggered by the server *during any* timeslot where it is the turn of the server/client pair. It is assumed that the client repeatedly triggers during all timeslots during which it is the turn of the pair, so the client node is always ready to perform the operation and it is up to the server to choose.

- The strict network schedule. Here, entanglement operations *must* always be triggered by the server at the *start* of the *next* timeslot where it is the turn of the

Symbol	Definition
A	given BQC-server applications
a_j	application j in A
D_j	a_j maximal run time
I_j	a_j instructions
e_j	a_j number executions
m_j	a_j number instructions
$i_{j,k}$	a_j instruction k
$r_{j,k}$	$i_{j,k}$ type
$d_{j,k}$	$i_{j,k}$ maximal duration
G	network schedule
g_b	G timeslot b application index
P	G repeating sequence length
L	G timeslot length
s_b	G timeslot b start time
w_j	index first timeslot for a_j in G
$t_{j,k,h}$	task: $i_{j,k}$ execution h
S	solution list of task start times
$s_{j,k,h}$	$t_{j,k,h}$ start time in S
T	S resulting makespan
$f_{j,h}$	duration execution h of a_j in S
Q_j	QC tasks for a_j
v_j	a_j number QC tasks
$q_{j,z}$	a_j QC task z

Table 3.2: Symbols used in the problem definition. For each symbol, the definition is shown.

server/client pair. It is assumed that the client also only triggers at the start of these timeslots. So, here both the server and the client only trigger an entanglement attempt at moments that have been agreed on beforehand (the timeslot starts).

In general, the strict network schedule is preferred, when taking into account that entanglement generation attempts cause decoherence. In a free network schedule, it could happen that the client attempts entanglement generation while the server does not (since the server is free to choose to do it at another time). Therefore, these client attempts lead to decoherence while not generating any entanglement. This unnecessary decoherence does not happen with the strict network schedule. However, it is the case that for the BQC applications that are taken into account for this research the client node does not use its quantum memory, as the client measures qubits directly after generating entanglement with the server in a BQC application (no values are stored), meaning that decoherence is not a problem for the client node for this specific case and the server only triggers when the entanglement generation would actually happen, so for the server no unnecessary entanglement attempts are triggered. Hence, it is interesting to study

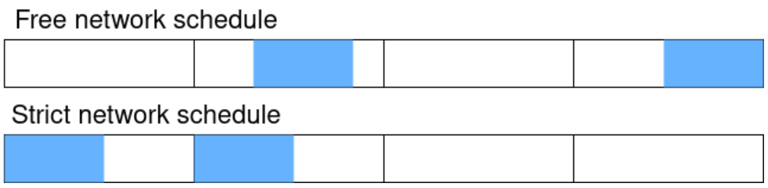


Figure 3.5: Visualisation of the two network schedule variants. On top, the free network schedule is shown with below the strict network schedule. In this illustration it is assumed that only the timeslots are shown for one specific application, the other timeslots are left out. For the application, two QC tasks have to be performed. It is assumed that these QC tasks have a constant duration for this example. The executions of the QC tasks happen during the periods marked in blue. It can be seen that for the free network schedule the first and the third timeslots are skipped, as skipping timeslots is allowed for the free network schedule. Also, the start time of the QC task can be at any moment during the timeslot for the free network schedule. Note that the QC tasks should be finished at the end of the timeslot that they start in; QC operations are not allowed to continue in the next timeslot. For the strict network schedule, it can be seen that QC tasks should always start at the start of the next timeslot where it is the turn of the application.

the free network schedule as well, to see if offering the server node more flexibility for triggering entanglement can yield better schedules in terms of makespan, since using a free network schedule would provide more options for scheduling tasks for the server node.

For the execution order, the following two variants are taken into account, which are also visualised in Figure 3.6:

- A sequential execution order. Here, an execution of an application should be fully completed before starting the next execution of the same application.
- An interleaving execution order. Here, instructions of the next execution of an application may already be executed before the current execution of this application is completed.

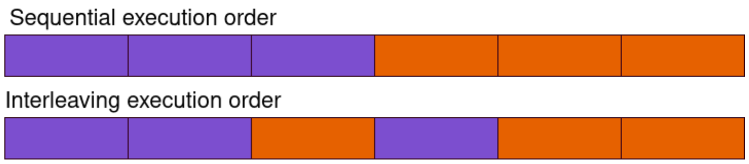


Figure 3.6: Visualisation of the two execution order variants. On top, the sequential execution order is shown with below the interleaving execution order. For this illustration, it is assumed that there is one application with three instructions, with all instructions being of constant duration. This application is executed two times. For the first execution the tasks are shown as purple blocks and for the second execution the tasks are shown as orange blocks. It can be seen that for the sequential execution order, the first execution must be fully finished before the second execution can start. For the interleaving execution order this is not the case; there it is possible to start the next execution earlier, as it can be seen that the last instruction of the first execution is executed after the first instruction of the second execution was finished. Note that these execution orders only apply for tasks belonging to a same application as in this example. Tasks of different applications can always be interleaved.

An interleaving execution order would be more flexible, which could possibly give a lower makespan. However, it is the case that with a sequential execution order the problem would be easier to solve and executing instructions on hardware would be easier, especially for processors with less computing power. Thus, it is interesting to study the differences between these two execution orders.

The four variants to the problem are the different possible combinations of the network schedule variants and the execution order variants. These problem variants are further referred to in this thesis as: **sequential free (SF), sequential strict (SS), interleaving free (IF) and interleaving strict (IS).** For each of these four names, the first part represents the execution order and the second part refers to the network schedule variant used.

3.4. CONSTRAINTS

For each of the four problem variants, a resulting schedule S should meet a list of constraints in order to be considered a valid solution. The constraints are now defined.

3.4.1. NO OVERLAP

This constraint is for all four problem variants. Only one task can be executed at a time. This means that it is required to check that there are no overlapping executions of tasks. Given two applications j and j_2, two instructions k and k_2 from these respective applications, two executions h and h_2 from the respective applications, task start times $s_{j,k,h}$ and s_{j_2,k_2,h_2} and maximal instruction durations $d_{j,k}$ and d_{j_2,k_2}, this constraint is formally defined as follows: $s_{j,k,h} + d_{j,k} \leq s_{j_2,k_2,h_2} \vee s_{j_2,k_2,h_2} + d_{j_2,k_2} \leq s_{j,k,h}$.

3.4.2. APPLICATION DEADLINES

This constraint is for all four problem variants and is mainly there for optimisation (it is added as an extra hard constraint). Every time that a BQC-server application is executed, all tasks corresponding to this specific application execution should finish within a maximal number of μs. This ensures that qubits are in memory as short as possible, to prevent significant decoherence over time, in order to achieve a success probability as high as possible. It is possible to run BQC applications without these deadlines, but this constraint allows for finding schedules which possibly yield better success probabilities. Given application j, execution h, the time that it takes to finish the application execution $f_{j,h}$ and application execution deadline D_j, this constraint is formally defined as follows: $f_{j,h} \leq D_j$.

3.4.3. INSTRUCTION ORDER

This constraint is for all four problem variants. During the execution of a BQC-server application, tasks performed for that execution depend on earlier tasks performed for that execution. For example, the server should have actually performed the computation before it can send the result to the corresponding client. Thus, the list of instructions of a BQC-server application should always be performed in order every time this application is executed. Given application j, instruction k, execution h and task start times $s_{j,k,h}$ and $s_{j,k+1,h}$, this constraint is formally defined as follows: $s_{j,k+1,h} > s_{j,k,h}$.

3.4.4. APPLICATION EXECUTION ORDER

For the SF and SS problem variants specifically (this constraint does not apply to the IF and IS variants), application executions should be performed sequentially for simplicity. This means that when a same application is performed multiple times, a next execution of this application can only start if the current execution of this application is finished, referring to the sequential execution order. Thus, tasks belonging to different executions of the same application cannot be interleaved. Given application j, execution h, total number of instructions m_j of application j and task start times $s_{j,1,h+1}$ and $s_{j,m_j,h}$, this constraint is formally defined as: $s_{j,1,h+1} > s_{j,m_j,h}$.

3.4.5. NETWORK SCHEDULE

The final constraint involves that the network schedule should be respected at all times. This constraint is different for the SF and IF variant pair and the SS and IS variant pair.

SF & IF

For the SF and IF problem variants there is more flexibility for the scheduling of QC tasks. Any timeslot where it is the turn of the application to which the task belongs, can be chosen. This means that timeslots can be skipped and that the QC operation can be performed during any timeslot where it is the turn of the application. Also, the QC task can start at any time during the timeslot, as long as it is finished before the next timeslot in the network schedule starts, as the timeslot duration can be longer than the duration of a QC task. Given an application j, an instruction k, type of instruction $r_{j,k}$, a network schedule timeslot b, task start time $s_{j,k,h}$, network schedule timeslot start times s_b and s_{b+1} and maximal instruction duration $d_{j,k}$, this constraint is formally defined as: $r_{j,k} = QC \Rightarrow \exists b(g_b = j \wedge s_{j,k,h} \geq s_b \wedge s_{j,k,h} + d_{j,k} \leq s_{b+1})$.

SS & IS

For the SS and IS variants, there is no flexibility for the scheduling of QC tasks. A QC task should be performed at the start of the next timeslot where it is the turn of the application to which the task belongs. Note that here not any moment during the timeslot can be chosen, the QC task should always start at the start of the specific timeslot, meaning that only one QC operation can be done per timeslot for these variants. Given application j, instruction k, execution h, corresponding QC task z, task start time $s_{j,k,h}$, network schedule timeslot length L, index of the first network schedule timeslot where it is the turn of the application w_j and the repeating sequence length of the network schedule P, this constraint is formally defined as: $s_{j,k,h} = L \cdot (w_j - 1 + P \cdot (z - 1))$.

3.5. CONSTRAINT PROGRAMMING MODELS

For each of the four problem variants a constraint programming (CP) model was implemented in MiniZinc that given the problem input generates a list of task start times that meets the constraints of the corresponding variant and minimises the makespan as the objective function. Thus, the makespan is used to directly assess the quality of a schedule. The application deadline constraint is used to decrease qubit degradation over time, to avoid significant loss in success probability while optimising the makespan. The

SF model can be found in appendix A. This is followed by the SS model in appendix B. Next, is the IF model in appendix C. Finally, there is the IS model in appendix D.

Optimisation is done for the four CP models. First, the use of global constraints was considered. The ‘disjunctive’ MiniZinc global constraint was used for the no overlap constraint. Next to making the models more readable, it also improved the solving time compared to checking each pair of tasks manually if they do not overlap.

Also a symmetry-breaking constraint was added for the IF and IS models, as here there is symmetry between multiple executions of a same application instruction (executions that are repeated have no set order). The symmetry-breaking constraint breaks the symmetry by enforcing that tasks with a same application and instruction index should be ordered by execution number of the application. This significantly improved the solving time for these two models, as an order was now enforced for the repeating executions. For the SF and SS models symmetry was not a problem, since in those models all tasks that belong to the same application are enforced an order already.

Also a redundant constraint was considered that all task start times should be different, as this logically follows from the constraint that two task executions cannot overlap, however this constraint had no significant effect on the solving time. Therefore, this redundant constraint is not part of the final CP models.

The final part of the optimisation included the comparison of different combinations of solver back-ends and search annotations, to see which search annotation would be useful in the CP models and which solver back-end would be the best to use in simulation for the final evaluation of the CP models. The results of this comparison and the final evaluation of the CP models are included in section 5.

4
Related work

In this section, related work is discussed in the field of scheduling problems. Two categories of related work are discussed. First, papers are discussed which also use a constraint programming approach to solve the problem under study, in order to see how this technique is used for solving other kinds of scheduling problems. Next, papers are discussed which use an alternative method to constraint programming, in order to see how other methods than constraint programming are used in the field of scheduling problems.

4.1. Constraint programming

[8] presents a study on the resource-constrained project scheduling problem (RCPSP) with general temporal and calendar constraints. Calendar constraints make some resources unavailable on certain days in the scheduling period. Various constraint programming models are given to tackle this problem, together with a specialised propagator for the cumulative resource constraints, taking the calendar constraints into account. One main difference between the problem description in [8] and in this thesis, is that in the first case resources are part of the environment which activities use and which can be used to a maximum capacity at a time. The use of resources is not part of this thesis, which makes the problem easier to solve.

In [9], another problem related to RCPSP is studied where constraint programming is used to solve the problem. This problem involves industrial test laboratories in which a large number of tests has to be performed by qualified personnel using specialised equipment, while respecting deadlines and other constraints. As [9] also solves a problem that is based on RCPSP, the use of resources is one of the main differences in the underlying problem description compared to this thesis. There is another difference in the way that the constraints in [9] are structured, namely that soft constraints are also part of the model. Compared to hard constraints which should always hold for solutions, soft constraints are there to make a solution more preferred (they are not strictly checked). [9] models the soft constraint as functions which should be minimised. One example is that for each project the total number of employees that are assigned to the project should be minimised. Soft constraints are not part of the problem under study in this research, since for this problem it is only needed to minimise the makespan, given the hard constraints that should hold for solutions.

4.2. ALTERNATIVE TECHNIQUES

[10] also focuses on scheduling for quantum networks. Specifically, a design is presented of a centralised quantum network with multiple users that orchestrates the delivery of entanglement which meets quality-of-service (QoS) requirements of applications. A different method is used than constraint programming, namely heuristics that focus on the problems of RCPSP and periodic task scheduling. One main difference between the scheduling problem in [10] and the problem under study in this thesis, is that the first work mainly focuses on network operations, as entanglement operations are mainly discussed. This thesis focuses on the scheduling of both local operations on individual nodes and network operations between nodes in a quantum network. One main benefit of the heuristic approach used in [10] is that solutions to the scheduling problem are found in a shorter amount of time, since it involves performing approximations rather than finding exact solutions.

[11] provides a study on the problem of multi-program scheduling on a single processor. Two scheduling techniques are discussed. One technique assigns fixed priorities to tasks that need to be scheduled on the processor. The other technique assigns priorities to tasks dynamically, based on their current deadlines. One main difference between the underlying problem description in [11] and the problem description in this thesis, is that for the first problem description priorities are assigned to tasks. This is not the case for the problem under study, since there are no explicit priorities assigned to tasks. However, given the case that entanglement operations can only be performed during timeslots as indicated by the network schedule, this implies that the constraint solver may give priority to these tasks during these timeslots indirectly, as they cannot be executed during periods outside of these timeslots. Another interesting aspect of the work in [11] has to do with the fact that their deadline driven scheduling algorithm assigns priorities to tasks dynamically, based on the deadlines of current requests. The request with the nearest deadline is assigned the highest priority. This is harder to do with a constraint programming approach, since there all tasks are assigned start times already before executing them. In other words, a static scheduling approach is used.

[12] studies dynamic real-time systems. These are systems in which tasks arrive at random times and which have associated time constraints. When a task arrives, a scheduling algorithm is in place that accepts the task when the time constraints can be satisfied or else the task is dropped. A main difference between the problem in [12] and the problem under study is the fact that in the problem description of the first work, task arrive at random times, whereas in this thesis all tasks that have to be scheduled are known beforehand. This makes the problem as discussed in [12] more complex to solve, as the algorithm has to adapt to incoming tasks, instead of being able to generate a schedule directly when all tasks would have been known.

[13] presents another scheduling approach that differs from constraint programming. The problem is studied of scheduling a set of periodic or sporadic tasks on a uniprocessor without preemption and without inserted idle time. A set of conditions is introduced such

that a set of periodic or sporadic tasks can be scheduled for arbitrary release times of the tasks. It is shown that any set of tasks fulfilling these conditions can be scheduled, using an earliest deadline first (EDF) scheduling algorithm. Tasks are modelled differently in [13] than in this thesis. Tasks are modelled as pairs. First, there is the computational cost, which is the time required to execute the task. Besides this being similar to the problem description under study (tasks in this thesis also have a maximal duration), the second element of the pair represents a minimal interval between invocations of a task, which slightly differs from the problem under study. For the SF and SS problem variants this is indirectly the case, as it is needed to finish the execution of an application, before being able to start the next execution of this application. However, for the IF and IS models there is not a minimal waiting time between executions of tasks that correspond to the same instruction, as tasks can be scheduled freely, as long as all constraints are satisfied for the corresponding schedule.

5

Experimental results

In this section, the experimental results are presented. First, the experimental setup is described which includes the system specifications and the versions of different software used. Next, the first experiment is discussed which was conducted to find a search annotation and solver back-end to use with the CP models for the final evaluation. Finally, the second experiment is discussed, which involved the final evaluation in simulation of the CP models by comparing them to a baseline scheduler.

5.1. Experimental setup

The experiments were conducted on a HP ZBook Studio G4 laptop, on a 64-bit Ubuntu 20.04.4 LTS operating system. This laptop has an Intel® Core™ i7-7700HQ CPU @ 2.80GHz processor with 4 cores and 8 logical processors. 8GB RAM is installed on this laptop.

MiniZinc was used for both the implementation and the use of the CP models in the experiments. Version 2.6.4 was used of MiniZinc.

Furthermore, the MiniZinc Python library was used to solve data instances with the developed CP models. Version 3.8.10 of Python was used.

Three solver back-ends were used for the MiniZinc solving process. First, version 6.3.0 of Gecode [14] was used. This is followed by version 0.10.4 of Chuffed [15]. Also version 9.4.1874 of OR-Tools [16] was used.

Finally, the Qoala simulator [17] was used to perform simulations with the generated schedules for the baseline scheduler and the CP models in the final evaluation. Version 0.1.7 of the Qoala simulator was used.

5.2. Search annotation & solver back-end comparison

In order to use the CP models in simulation, a solver back-end needs to be chosen which should be used to execute the MiniZinc constraint solver. Furthermore optional search annotations can be used in the model in order to help the solver to find solutions faster. A comparison was conducted on various solver back-ends and search annotations in order to choose a solver back-end and search annotation for the evaluation of the CP models. Thus, this experiment was mainly conducted to prepare for the second experiment, which is the actual final evaluation of the developed CP models.

5.2.1. DESCRIPTION

When solving a constraint problem in MiniZinc, it is necessary to specify a solver back-end. Three solver back-ends were selected for comparison. The first solver is the default solver for MiniZinc, Gecode. Next, Chuffed was used, as this solver can often be faster than traditional CP solvers as stated in the MiniZinc documentation [18]. The last solver used is OR-Tools, which is accessible to use due to its open source nature and it is tuned for solving problems in the field of constraint programming among other fields.

Search annotations are optional in a MiniZinc model, but they can help to improve the solver performance, by specifying how the solver should search for solutions. When specifying a search annotation, a variable selection annotation and a value selection annotation should be specified. The first annotation determines which decision variable to search first. In the context of this thesis, that would be which task to find a start time for first. The latter determines which value to assign to this variable. That would be the actual start time to assign to a task in this context. It is stated in the MiniZinc documentation that the variable selection strategy affects performance more than the value selection strategy [19]. In order to not make this experiment too complex, the value selection annotation was kept constant for that reason and various variable selection strategies were compared. The value selection strategy used is 'indomain', which assigns values in ascending order to the variables. This would be a simple strategy that would try the different possible start times in order. Three different variable selection strategies were compared for each solver back-end. First is 'occurrence', which chooses the variables with the largest number of attached constraints, which may be useful for QC tasks for example, as these tasks have additional network schedule related constraints. Next is 'smallest', which chooses the variable with the smallest value in its domain, which may be useful to determine which tasks should start first. Finally, 'first_fail' was considered, which chooses the variable with the smallest domain, which may be useful to schedule tasks early on for which the least amount of start times are possible.

Three data files were used to compare the different search annotation and solver back-end combinations. The first data file, which can be found in appendix E, focuses on a case where there are a few smaller applications which are executed many times. The second data file, which can be found in appendix F, focuses on a case where there are a few larger applications which are executed only a few times. Finally, there is a data file which takes many small applications into account and execute these a few times. This data file can be found in appendix G. These data files were made in order to find out how the number of applications, the number of instructions per application and the number of executions per application influence the solving time and objective value (makespan). The data files were made manually by looking for values for the stated input variables such that the data files were complex enough to see differences between the different search annotation/solver back-end combinations, but still feasible to solve in a reasonable amount of time.

For all four models all three data files were solved for each solver back-end and search annotation combination. This was done 10 times and the average value and standard deviation of both the solving time and objective values were taken. The solving process was repeated to get a more accurate solving time as this differs per solver execution and to verify that always the same objective value was returned, as the search annotations used in the comparison do not use any form of randomness. A time-out of 5 seconds was considered for the solving process. This is, because the data files could take a long time to solve for specific combinations and data files and therefore it would return the best objective value found within the time-out period.

5.2.2. Results

For each of the four models graphs were made to study how the different combinations of solver back-ends and search annotations influence the objective value and solving time. Objective values are represented by bars in the graphs and solving times are projected as black dots, or black triangles in case of a time-out (the best objective value found within that time would then be shown). When no solution was found before the time-out, a cross is shown instead of a bar. The variance in solving time was negligible. For the objective value, there was only a small variance for the objective value for the IF model, for the other models the variance was zero for the objective value for all combinations. The results for the models are now individually discussed.

SF

Figure 5.1 illustrates the results of the search strategy comparison for the SF model. Interesting here is that for OR-Tools only solutions are found for the second data file. Also remarkable is that for both Gecode and Chuffed, no solution is found for the ‘occurrence’ variable selection strategy in specific cases (second data file for Gecode and first and third data files for Chuffed). On overall, Chuffed seems to have better solving times than Gecode for the SF model, especially for the first and third data files for the ‘smallest’ and ‘first_fail’ variable selection strategies. In terms of objective value, the largest differences are in the second data file. It can be seen that the ‘smallest’ variable selection strategy provides the best objective values for the Gecode and Chuffed solver back-ends. For the last data file, Chuffed seems to find a better objective value than Gecode for the ‘smallest’ variable selection strategy.

SS

Figure 5.2 illustrates the results of the search strategy comparison for the SS model. Interesting here is that for each individual data file, the different combinations of solver back-ends and variable selection strategy seem to produce similar objective values. There is only one case in which no solution can be found, which is for the second data file for the Gecode solver back-end for the ‘occurrence’ variable selection strategy. Furthermore, all solving times seem to be low, excluding the fact that there were two time-outs for the Gecode solver back-end for the second data file.

IF

Figure 5.3 illustrates the results of the search strategy comparison for the IF model. Similarly to the SF model, OR-Tools only finds solutions for the second data file. Additionally, the ‘smallest’ variable selection strategy seems to only find solutions for both the Gecode and the Chuffed solver back-ends for all data files, as no solution is found for ‘occurrence’ for both the Gecode and the Chuffed solver back-ends in the second data file and also there is no solution for the ‘first_fail’ variable selection strategy for the Chuffed solver. In terms of solving time, both Gecode and Chuffed do well for the ‘smallest’ annotation in the second data file. Chuffed also finishes quickly for the ‘smallest’ search annotation in the last data file. Interesting here is that for the first data file, using the Chuffed solver back-end and the ‘smallest’ search annotation, a non-zero standard deviation of approximately 155 was found for the objective value. This could have to do with the variance in solving times. That for some runs a better objective value was found for this combination just before the time-out, whereas for other runs this objective value was not found yet and a higher objective value was found before the time-out. For all other bars the standard deviation of the objective value was 0.

IS

Figure 5.4 illustrates the results of the search strategy comparison for the IS model. Interesting is, that the results for this model look identical to those of the SS model. This means that like the SS model, the IS model is also able to find solutions quickly.

5.3. Evaluation of CP models

In order to evaluate the effectiveness of the CP models, BQC-server applications were scheduled and simulated for both a baseline scheduler and the CP models that the server node performs when performing BQC applications repeatedly with different numbers of clients. Each BQC-server application had the same instructions, listed in appendix H. These instructions were similar to the instructions listed for the server node in Figure 12 in [20]. This means that given a number of clients, for each client these instructions were executed the number of times the particular application was executed. The baseline scheduler would iterate over all BQC-server applications and their tasks and schedule them as soon as possible. This means that the baseline scheduler provides an upper bound for the success probability, taking decoherence over time into account. All tasks other than QC tasks are always scheduled directly and when a QC task is encountered the baseline scheduler waits until the next timeslot that it is the turn of the belonging application and continues from there, if the QC task cannot be scheduled directly because of the network schedule. The produced schedules by the CP models and the baseline scheduler were simulated in the Qoala simulator and the results were compared. The CP models and the baseline scheduler were compared based on makespan and success probability. The idea was to show that the CP models improve the makespan for the case when a server node performs BQC with multiple clients, while the success probability is similar to that of the baseline scheduler. As aforementioned, the baseline scheduler gives an upper bound for the success probability, since tasks are scheduled as soon as possible. Therefore it is necessary to show that a similar success probability can be preserved for the CP models, while improving the makespan to show that tasks can be finished faster without the success probability degrading.

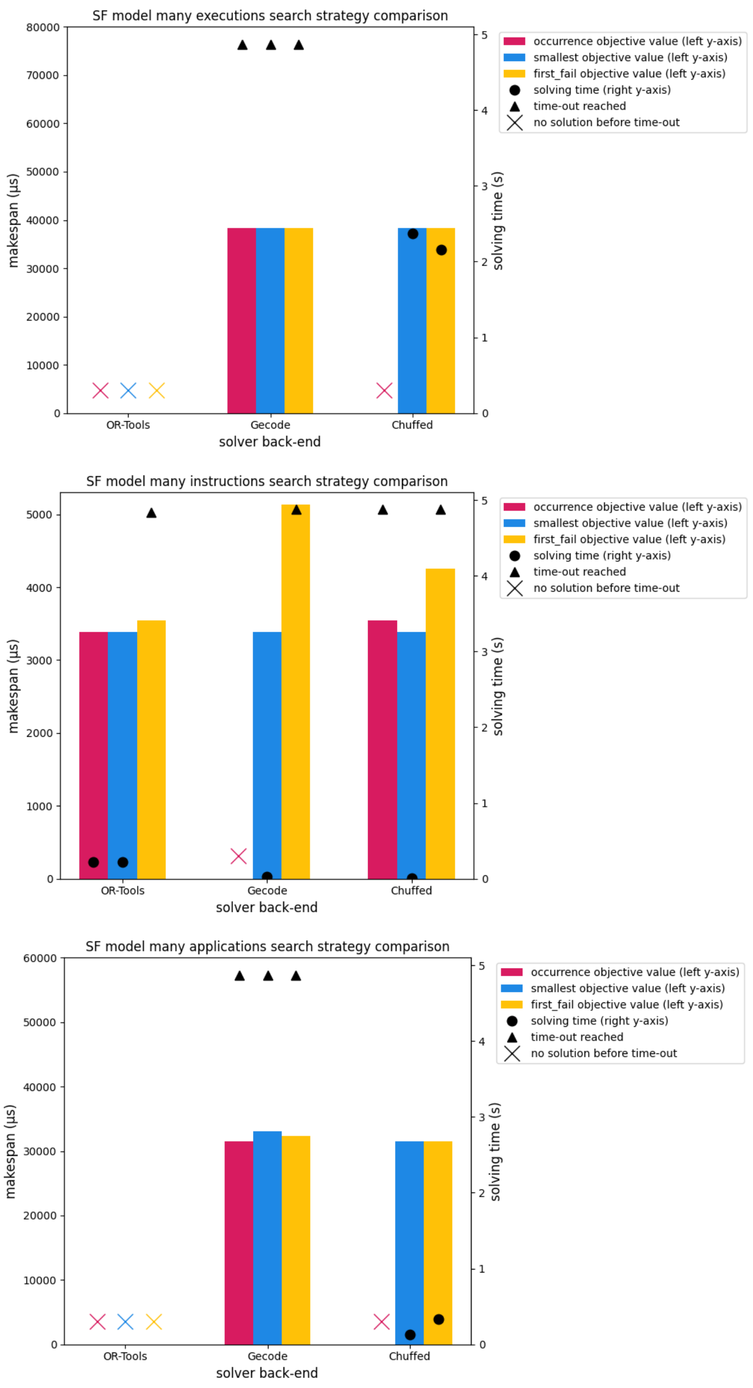
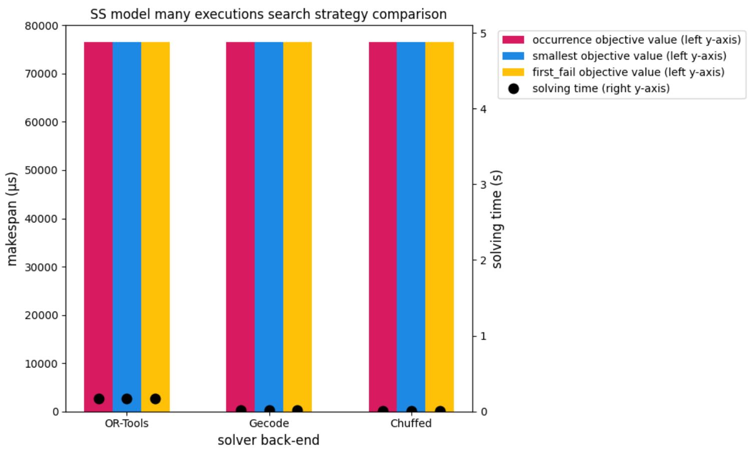
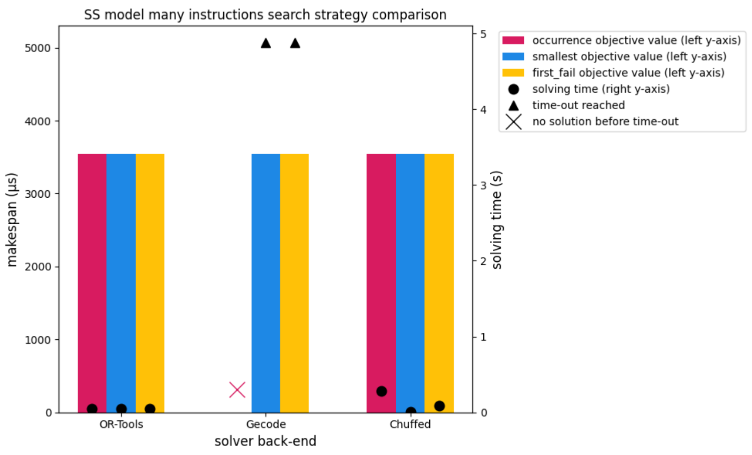


Figure 5.1: Results of comparing solver back-end/search annotation combinations for the SF model for the 3 different data files. The bars represent the objective values (makespan) found for every combination, or a cross is shown when there was no solution found within the time-out period. The black dots indicate the time that solving the data file took for the CP model, using the given combination. When a black triangle is shown instead of a black dot, that means that there was a time-out, which means that the best objective value found within that time is shown in the associated bar.





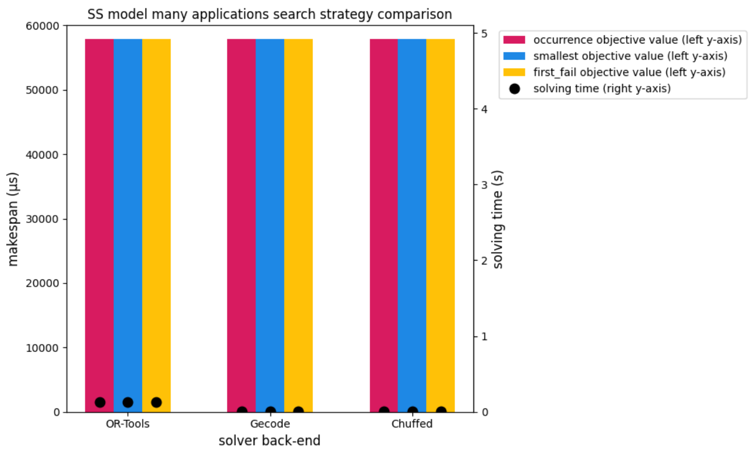


Figure 5.2: Results of comparing solver back-end/search annotation combinations for the SS model for the 3 different data files. The bars represent the objective values (makespan) found for every combination, or a cross is shown when there was no solution found within the time-out period. The black dots indicate the time that solving the data file took for the CP model, using the given combination. When a black triangle is shown instead of a black dot, that means that there was a time-out, which means that the best objective value found within that time is shown in the associated bar.

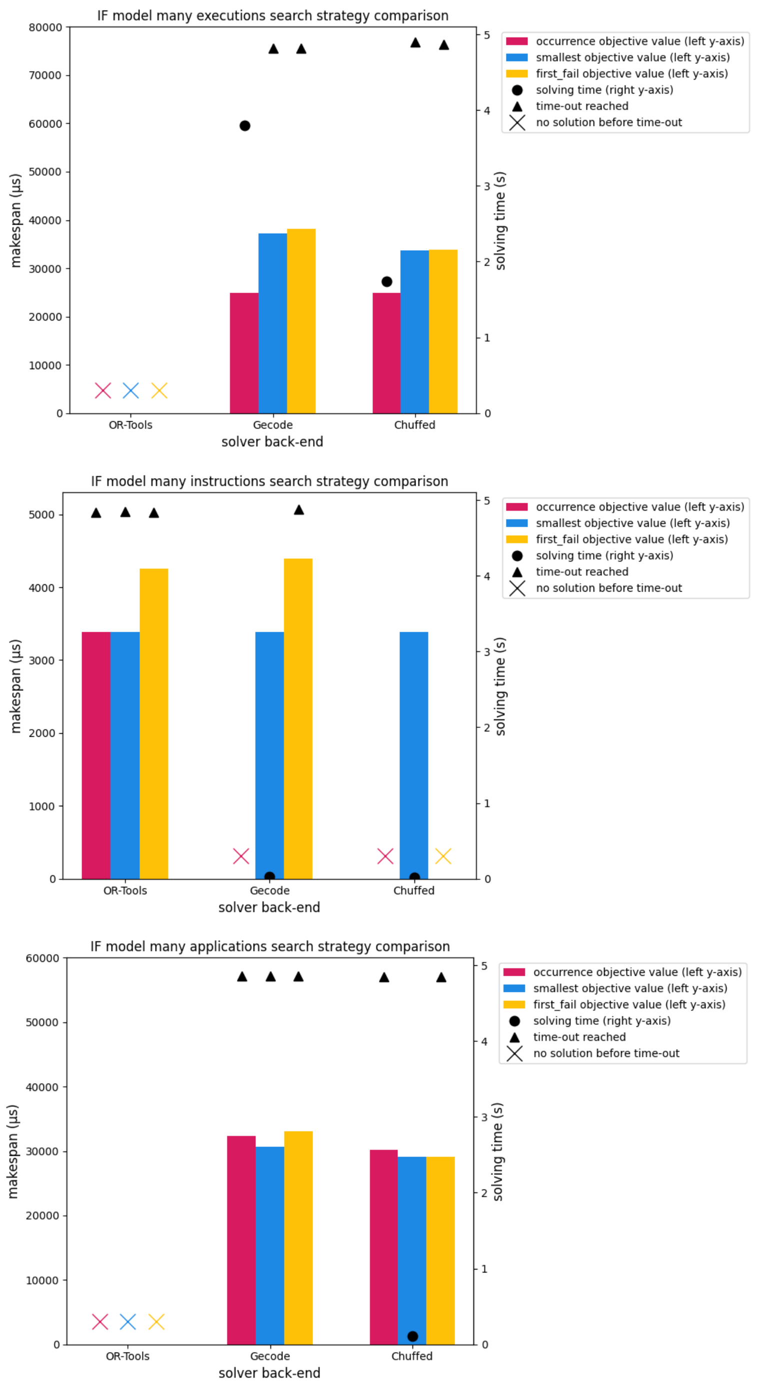


Figure 5.3: Results of comparing solver back-end/search annotation combinations for the IF model for the 3 different data files. The bars represent the objective values (makespan) found for every combination, or a cross is shown when there was no solution found within the time-out period. The black dots indicate the time that solving the data file took for the CP model, using the given combination. When a black triangle is shown instead of a black dot, that means that there was a time-out, which means that the best objective value found within that time is shown in the associated bar.

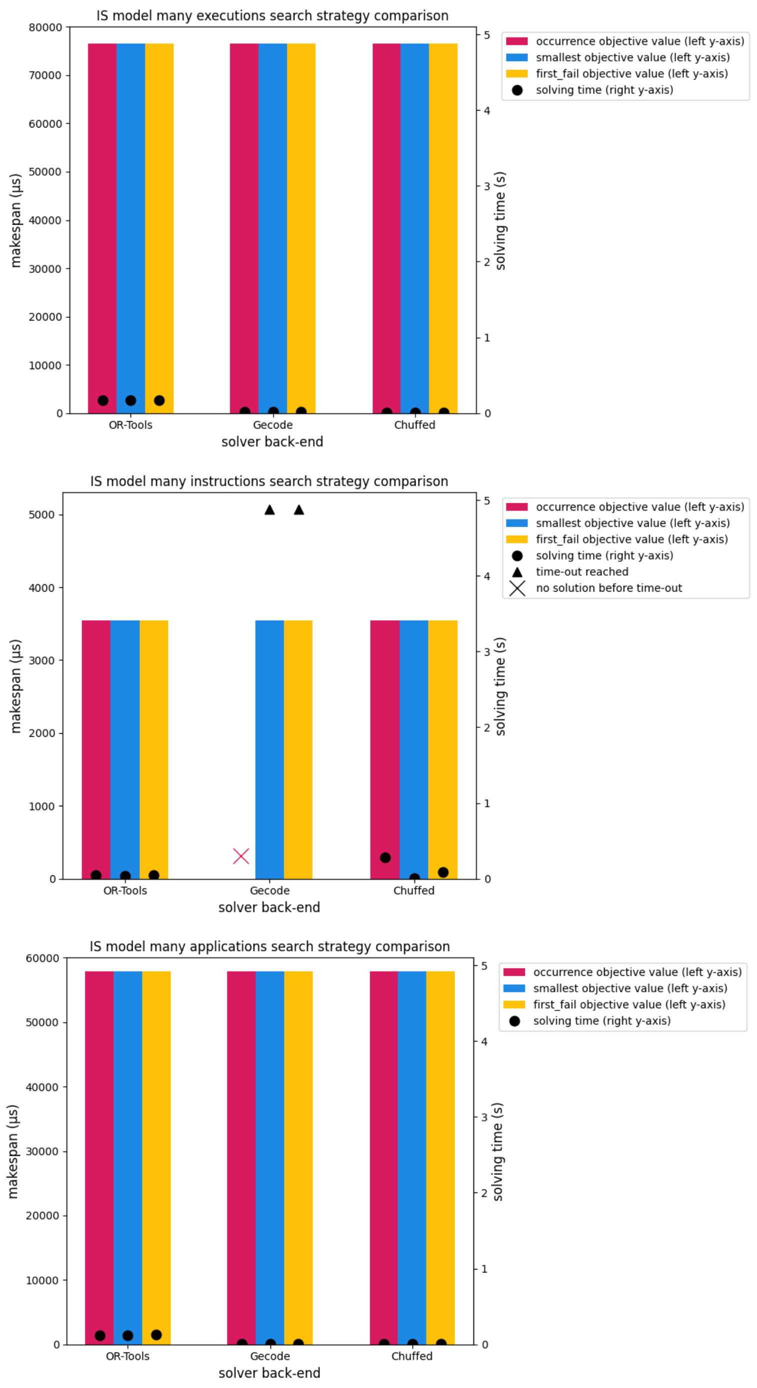


Figure 5.4: Results of comparing solver back-end/search annotation combinations for the IS model for the 3 different data files. The bars represent the objective values (makespan) found for every combination, or a cross is shown when there was no solution found within the time-out period. The black dots indicate the time that solving the data file took for the CP model, using the given combination. When a black triangle is shown instead of a black dot, that means that there was a time-out, which means that the best objective value found within that time is shown in the associated bar.

5.3.1. Description

For different numbers of clients BQC-server applications were simulated. Every application had the same instructions. The types, individual durations and total duration of these instructions are in appendix H.

For every number of clients, a schedule was created using the CP models and the baseline scheduler that would schedule 35 executions of the BQC-server application for every client. This number was chosen as high as possible such that solutions could still be found for the maximal number of clients in a reasonable amount of time for the CP models, in order to find a more accurate success probability, as more executions would yield a more accurate success probability. The different schedules were simulated. For the makespan, the time would be taken that simulation of the schedule took. The success probability was calculated separately for every client, which is the fraction of executions that yield a desired result out of the total of 35 executions. The average was taken of all client success probabilities, which is the value that was used for the evaluation. As the success probability involves randomness, the simulation of the computed schedules was repeated 120 times. Out of all repetitions, the average was taken as the final success probability. The standard deviation was also calculated to get an idea of the variance in the success probability. The makespan would always be the same for every repetition, since the same schedule was repeated in simulation and only the success probability involves randomness, as that can vary per repetition of simulating the schedule.

For the CP models, a time-out was used of 5 minutes in order to give the solver extra time for finding solutions. The Chuffed solver back-end was used as it prove to give lower solving times in the previous experiment. In terms of variable selection annotations, ‘smallest’ was used, as for this annotation a solution was always found by the Chuffed solver back-end in the previous experiment. The ‘indomain’ value selection annotation was used that was already constantly used in the first experiment. The network schedule would repeat the clients in order. A timeslot length was used of 8e4 μs, which allowed to see the differences between the SF and IF models with the SS and IS models, since for the SF and IF models both QC instructions could be scheduled in the same timeslot, but for the SS and IS models the solver should wait until the next assigned timeslot to perform the second QC task. Furthermore, for every model different execution deadlines were used. For the SF model a tight deadline was taken of 7e4 μs, which is approximately the total duration of the BQC-server instructions. For the IF model this deadline was multiplied with the number of clients and doubled, in order to give extra time for interleaving executions. For the SS model a deadline was taken of 7e4 μs (again the total duration of all BQC instructions) with added the product of the number of clients and the timeslot length, to account for the fact that the solver has to wait for scheduling the second QC task in the next assigned timeslot. For the IS model, this deadline was doubled in order to assign extra time for interleaving instructions.

Different simulator specific parameters were used in order to model the decoherence, which affects the success probability. A link fidelity of 0.8 was used in order to specify the quality of entanglement links. The lower this quality is, the more decoherence occurs such that the success probabilities would be lower. Also parameters were set for the nodes themselves in the network. Probabilities were set for both single and two qubit gates that they cause decoherence, these are 0.05 and 0.1 respectively. Finally, the decoherence over time was modelled. A standard way of modelling the noise over time is with the use of two components: an amplitude damping component and a dephasing component [21]. These were set to 1e7 and 1e6 μs respectively.

In order to investigate if the differences between the means of the success probabilities of the CP models and the baseline were statistically significant, an independent (unpaired) t-test was done for each CP model with the baseline for every number of clients. The null hypothesis stated that there is no statistically significant difference between the means of the success probabilities of the baseline and a specific CP model for a specific number of clients. The alternative hypothesis would be that there is a statistically significant difference. A significance level (referred to as α) was taken of 0.05, which is the probability of rejecting the null hypothesis when it is actually true. This is a value commonly used for the significance level in t-tests. The sample size N is the number of times that the different schedules were simulated, 120. The t-test was performed with the use of a Python script that iterates over each CP model for each number of clients studied. For each number of clients the standard error SE was first calculated for every CP model and the baseline scheduler. Given a standard deviation S and a sample size N, the standard error of the sample is calculated as follows: $SE = \frac{S}{\sqrt{N}}$. Given standard errors SE_{model} and $SE_{baseline}$ for a CP model and the baseline respectively for a given number of clients, the standard error of the difference between the means SED was calculated as follows: $SED = \sqrt{SE^2_{model} + SE^2_{baseline}}$. Next, the value for t was calculated. Given the standard error of the difference between the means SED and means M_{model} and $M_{baseline}$ for a CP model and the baseline respectively for a specific number of clients, a value for t was calculated as follows: $t_{calc} = \frac{M_{model} - M_{baseline}}{SED}$. Next, the critical t-value, t_{crit} was calculated, with the use of a percent point function from the SciPy t-stats library [22], using the complement of α and the degrees of freedom df. Given sample sizes N_{model} and $N_{baseline}$ for a CP model and the baseline respectively for a specific number of clients (both the number of simulations, 120), the degrees of freedom are $df = N_{model} + N_{baseline} - 2$. Finally, the calculated t-value t_{calc} was compared to the critical value t_{crit}. If $|t_{calc}| <= t_{crit}$ would hold, it was shown that the null hypothesis could not be rejected, meaning that for a given number of clients there would be no significant difference between the found means of the success probabilities of a given CP model and the baseline scheduler, meaning that the desired success probability was indeed preserved.

5.3.2. Results

For both the makespan and the average success probability, graphs were made which show the values for the baseline and the four CP models for every number of clients. Both performance metrics are now individually discussed.

Makespan

Figure 5.5 shows the resulting makespans for every number of clients that executing the generated schedules by the baseline and CP models gave. First, it can be noticed that assigning higher deadlines to the IF and IS models than to the SF and SS models respectively that were used to give the opportunity to interleave executions did not change the makespan. It could be that a schedule without interleaving was the best schedule found before the time-out for the IF and IS models. Out of the four CP models, the SS and the IS models perform the worst, which can be explained by the fact that there is additional waiting time for performing the second QC task, since QC tasks only start at the start of the network schedule timeslots. For 1 client the makespan is equal for the baseline and the SF and IF models, which is logical, since in that case both the baseline and the SF and IF models would perform instructions as soon as possible, which yields the same results for 1 client. It can be seen that when adding extra clients, the makespan of the SF and the IF models are better than for the baseline. Also, adding even more clients increases the difference in makespan between the baseline and the SF and IF models. Another note is that from 3 clients on, the IS and SS models clearly perform better in terms of makespan than the baseline. So, it can be seen that the CP models are able to improve the makespan found by the baseline scheduler. An explanation for the lower makespan for the CP models is the fact that QC tasks interleave for clients: when one client has to wait before it can perform QC tasks again, the other clients for which it is their turn in the in-between timeslots, perform QC tasks. For the baseline scheduler this is not the case, there no other tasks are performed while it is not the turn yet of the application to perform QC tasks.

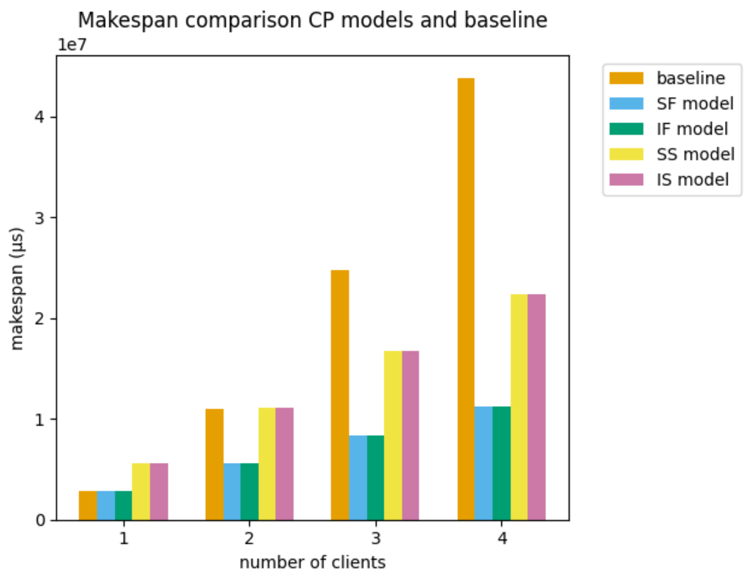


Figure 5.5: The results of the makespan comparison between the four CP models and the baseline. For every number of clients first the value for the baseline is shown. This is followed by the values for the CP models.

SUCCESS PROBABILITY

Figure 5.6 shows the resulting average success probabilities for every number of clients that executing the generated schedules by the baseline and CP models gave. Firstly, it is good to notice that despite the fact that simulations were repeated 120 times, there is still significant variance involved. This could have to do with the randomness that is involved in calculating the success probability. It seems however that this variance decreases when adding more clients. An explanation for this is that in the case of more clients, the average is taken of a bigger set of success probabilities (one for every client) for every repetition of the schedule simulation, meaning that the overall average success probability becomes more accurate. Out of the four CP models, the SS and IS models seem to give lower success probabilities than the SF and the IF models and for more clients this difference seems bigger. This can be explained by the fact that for the IS and the SS models there is a higher application deadline, because there executions of applications take longer to complete due to the waiting time between the QC operations. Decoherence that happens over time could have caused the fall in success probability. Furthermore, the difference between the baseline and the SF and IF models in success probability seems small, except for 1 client, where the average success probability for the SF model is noticeably higher. It is good to say that this is under a high variance. On overall, it can be seen that the SF and IF models are able to preserve the success probability of the baseline scheduler which was taken as the minimal desired value.

Table 5.1 shows the results of the t-tests performed for every CP model for every number of clients to see if the found mean of success probability for that combination differs significantly from the mean found for the baseline scheduler. The cases where the null hypothesis was rejected, so where there was a significant difference found, are highlighted. It can be seen that for the SF and IF models the null hypothesis was accepted for every case. This means that there was indeed no significant difference found between the means of the success probabilities for the SF and IF models with the baseline scheduler.

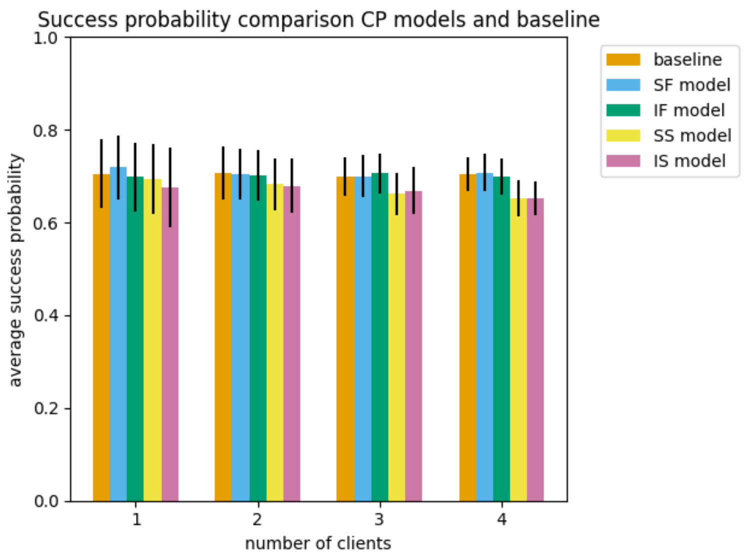


Figure 5.6: The results of the success probability comparison between the four CP models and the baseline. For every number of clients first the value for the baseline is shown. This is followed by the values for the CP models. The black vertical lines represent the standard deviation for a specific bar in the graph.

Model	Num. clients	t_{calc}	t_{crit}	Accept null hyp.
SF	1	-1.5656241938087199	1.6512811638136882	yes
	2	0.4228138029434098	1.6512811638136882	yes
	3	-0.12892801596526013	1.6512811638136882	yes
	4	-0.7246455191180255	1.6512811638136882	yes
IF	1	0.7107607773586572	1.6512811638136882	yes
	2	0.6836215792155761	1.6512811638136882	yes
	3	-1.2975485008118615	1.6512811638136882	yes
	4	0.7696178449843244	1.6512811638136882	yes
SS	1	1.1204839476536188	1.6512811638136882	yes
	2	3.373277196432736	1.6512811638136882	**no**
	3	6.692639011569419	1.6512811638136882	**no**
	4	10.452772967376136	1.6512811638136882	**no**
IS	1	2.923002593042345	1.6512811638136882	**no**
	2	3.8461601388686737	1.6512811638136882	**no**
	3	5.099945261565124	1.6512811638136882	**no**
	4	10.761468002801788	1.6512811638136882	**no**

Table 5.1: Table representing the results of the performed t-tests in order to study whether there is a significant difference between the means of the success probabilities for the baseline and the given CP models for every number of clients. For every CP model, there is a row for every number of clients. The calculated value t_{calc} is shown together with the critical value t_{crit}. The critical value t_{crit} is the same for every combination (this value only depends on the chosen value for significance level α and degrees of freedom). The last column indicates whether or not the null hypothesis should be accepted that there is no significant difference. The null hypothesis would be accepted if the absolute value for the calculated value t_{calc} was not higher than the critical value t_{crit}. The cases where the null hypothesis was rejected, so that there was a significant difference found between the means of the success probabilities for a given combination with the baseline, are highlighted. It can be seen that only significant differences in the means of the success probabilities were found for the SS and IS models. This means that there is no statistically significant difference found between the means of the success probabilities for the SF and IF models with that of the baseline for every number of clients.

6 Conclusion

Four different CP models were introduced that assign start times to tasks that a server node in a quantum network should perform for BQC. Schedules generated by these models satisfy constraints for performing entanglement according to a network schedule among other constraints. The makespan is directly minimised as the objective function and the extra execution deadlines allow for specifying how fast application executions should finish in order to prevent significant qubit decoherence over time and to preserve a sufficient success probability. The CP models were evaluated by comparing them on makespan and success probability with a baseline scheduler. As the baseline scheduler schedules tasks as soon as possible, this would give the best success probability that can be achieved, taking qubit decoherence over time into account. It was found that the SF and IF models can significantly decrease the makespan in the case of multiple clients and the IS and SS models as well when enough clients are in the network compared to the baseline. The SF and IF models yield a success probability that is similar to that of the baseline, which was also verified with a t-test. For the IS and SS models, the success probability is lower than that of the baseline, however these models can find schedules faster than the SF and IF models, because QC tasks are scheduled at set times, reducing the search space.

It has been shown with an experimental demonstration of the use of CP for scheduling instructions for a server node performing BQC applications with multiple client nodes repeatedly that CP is an approach that can be used to schedule instructions in such a way that a network schedule is respected for entanglement generation and the overall makespan is decreased. One main drawback of the CP approach is scalability. For larger inputs, it becomes hard to find solutions in a feasible amount of time. Therefore, a suggestion for future work would be to study how CP could be used to handle larger quantum network applications in larger quantum networks. Furthermore, other methods such as heuristics could be used which allow for finding solutions in a smaller amount of time.

Bibliography

[1] Stephanie Wehner, David Elkouss, and Ronald Hanson. “Quantum internet: A vision for the road ahead”. In: *Science* 362.6412 (2018), eaam9288.

[2] Axel Dahlberg et al. “A link layer protocol for quantum networks”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 159–173.

[3] Wojciech Kozlowski, Axel Dahlberg, and Stephanie Wehner. “Designing a quantum network protocol”. In: *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 2020, pp. 1–16.

[4] Pablo Arrighi and Louis Salvail. “Blind quantum computation”. In: *International Journal of Quantum Information* 4.05 (2006), pp. 883–898.

[5] Anne Broadbent, Joseph Fitzsimons, and Elham Kashefi. “Universal blind quantum computation”. In: *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE. 2009, pp. 517–526.

[6] Matteo Pompili et al. “Experimental demonstration of entanglement delivery using a quantum network stack”. In: *arXiv preprint arXiv:2111.11332* (2021).

[7] *MiniZinc*. https://www.minizinc.org/. Accessed: 16 January 2023.

[8] Stefan Kreter, Andreas Schutt, and Peter J Stuckey. “Using constraint programming for solving RCPSP/max-cal”. In: *Constraints* 22.3 (2017), pp. 432–462.

[9] Tobias Geibinger, Florian Mischek, and Nysret Musliu. “Investigating constraint programming for real world industrial test laboratory scheduling”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2019, pp. 304–319.

[10] Matthew Skrzypczyk and Stephanie Wehner. “An Architecture for Meeting Quality-of-Service Requirements in Multi-User Quantum Networks”. In: *arXiv preprint arXiv:2111.13124* (2021).

[11] Chung Laung Liu and James W Layland. “Scheduling algorithms for multiprogramming in a hard-real-time environment”. In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[12] Wei Zhao and John A Stankovic. “Performance analysis of FCFS and improved FCFS scheduling algorithms for dynamic real-time computer systems”. In: *1989 Real-Time Systems Symposium*. IEEE Computer Society. 1989, pp. 156–157.

[13] Kevin Jeffay, Donald F Stanat, and Charles U Martel. “On non-preemptive scheduling of periodic and sporadic tasks”. In: *IEEE real-time systems symposium*. US: IEEE. 1991, pp. 129–139.

[14] *Gecode*. https://www.gecode.org/. Accessed: 16 January 2023.

[15] *Chuffed*. https://github.com/chuffed/chuffed. Accessed: 16 January 2023.

[16] *OR-Tools*. https://developers.google.com/optimization. Accessed: 16 January 2023.

[17] *Qoala*. https://github.com/QuTech-Delft/qoala-sim. Accessed: 16 January 2023.

[18] *Chuffed solving technology*. https://www.minizinc.org/doc-2.3.0/en/solvers.html#chuffed. Accessed: 16 January 2023.

[19] *MiniZinc annotations*. https://www.minizinc.org/doc-2.5.5/en/mzn_search.html#annotations. Accessed: 16 January 2023.

[20] Axel Dahlberg et al. “NetQASM-A low-level instruction set architecture for hybrid quantum-classical programs in a quantum internet”. In: *Quantum Science and Technology* (2022).

[21] Guus Avis et al. “Requirements for a processing-node quantum repeater on a real-world fiber grid”. In: *arXiv preprint arXiv:2207.10579* (2022).

[22] *SciPy t-stats library*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.t.html. Accessed: 16 January 2023.

A

SF MODEL

```
% MiniZinc model that finds task starttimes for sequential executions with a free network
     schedule.

include "globals.mzn";

% Number of applications.
int: n;

% The max. durations for each application.
array[1..n] of int: deadlines;

% The number of instructions for each application.
array[1..n] of int: nr_instructions;

% Number of executions for each application.
array[1..n] of int: nr_executions;

% Max execution durations and instruction types for each instruction (-1 and "" respectively
     if less instructions)
int: max_nr_instructions = max(nr_instructions);
array[1..n, 1..max_nr_instructions] of int: instr_max_durations;
array[1..n, 1..max_nr_instructions] of string: instr_types;

% Length of a timeslot in the network schedule.
int: L;

% Network schedule which contains timeslots when which application can perform 'QC' operations
     .
array[int] of int: GS;

% The length of the network schedule.
int: g = length(GS);

% The total number of tasks to schedule (each instruction execution of each application).
int: total_nr_tasks = sum([app_nr_tasks(i) | i in 1..n]);

% Array that contains for each application which task index contains the first task of that
     application.
array[1..n] of int: app_start_indices = [sum([app_nr_tasks(j) | j in 1..i-1]) + 1 | i in 1..n
     ];

% The duration of each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_durations = [instr_max_durations[i,j] | i in 1..n,m in
     1..nr_executions[i], j in 1..nr_instructions[i]];

% The type of each task (ordered by application and instruction).
array[1..total_nr_tasks] of string: task_types = [instr_types[i,j] | i in 1..n,m in 1..nr_
     executions[i], j in 1..nr_instructions[i]];
```

```
% The application that belongs to each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_apps = [i | i in 1..n,m in 1..nr_executions[i], j in 1..
      nr_instructions[i]];

% Upper bound for task starttimes. This is the total duration + the maximal waiting time for
      QC tasks.
int: start_times_upper_bound = sum(task_durations) + L * g * count([task_types[i] == "QC" | i
      in 1..total_nr_tasks]);

% Decision variable: starttime for each task (ordered by application and instruction).
array[1..total_nr_tasks] of var 0..start_times_upper_bound: task_start_times;

% Constraint: For a same application, the previous task should be completed before starting a
      new task.
constraint forall(i in 2..total_nr_tasks where task_apps[i-1] == task_apps[i])(task_start_
      times[i] > task_start_times[i-1]);

% Constraint: no two tasks overlap.
constraint disjunctive(task_start_times, task_durations);

% Constraint: For all 'QC' tasks the task starts and ends in the same timeslot it is the turn
      of the application.
constraint forall(i in 1..total_nr_tasks where task_types[i] == "QC")((GS[((gs_timeslot(i) mod
       g) + 1)] == task_apps[i]) /\ (((L * (gs_timeslot(i) + 1)) - task_start_times[i]) >= task
      _durations[i]));

% Constraint: All applications executions are finished within the maximal duration (deadline).
constraint forall(i in 1..n, m in 1..nr_executions[i])((task_end_time(get_exec_task_start_
      index(i, m+1) - 1) - task_start_times[get_exec_task_start_index(i, m)]) <= deadlines[i]);

% Objective function: minimize the makespan. Search variables with smallest value in domain
      and values ascendingly.
solve :: int_search(task_start_times, smallest, indomain) minimize max([task_end_time(i) | i
      in 1..total_nr_tasks]);

% Given the number of a task calculates the time it is finished (start time + duration).
function var int: task_end_time(int: task_index) = task_start_times[task_index] + task_
      durations[task_index];

% Calculates the absolute timeslot in the network schedule when a task starts. First timeslot
      has index 0.
function var int: gs_timeslot(int: task_index) = task_start_times[task_index] div L;

% Finds the task index a given execution starts of a given application.
function int: get_exec_task_start_index(int: app_index, int: exec_nr) = app_start_indices[app_
      index] + (exec_nr - 1) * nr_instructions[app_index];

% Calculates the number of tasks for an application. For each instruction there is a task for
      each execution.
function int: app_nr_tasks(int: app_index) = nr_instructions[app_index] * nr_executions[app_
      index];

% Print the resulting starttimes.
output ["\(task_start_times)"];
```

B
SS MODEL

```
% MiniZinc model that finds task starttimes for sequential executions with a strict network
     schedule.

include "globals.mzn";

% Number of applications.
int: n;

% The max. durations for each application.
array[1..n] of int: deadlines;

% The number of instructions for each application.
array[1..n] of int: nr_instructions;

% Number of executions for each application.
array[1..n] of int: nr_executions;

% Max execution durations and instruction types for each instruction (-1 and "" respectively
     if less instructions)
int: max_nr_instructions = max(nr_instructions);
array[1..n, 1..max_nr_instructions] of int: instr_max_durations;
array[1..n, 1..max_nr_instructions] of string: instr_types;

% Length of a timeslot in the network schedule.
int: L;

% Network schedule which contains timeslots when which application can perform 'QC' operations
     .
array[int] of int: GS;

% The length of the network schedule.
int: g = length(GS);

% The total number of tasks to schedule (each instruction execution of each application).
int: total_nr_tasks = sum([app_nr_tasks(i) | i in 1..n]);

% Array that contains for each application which task index contains the first task of that
     application.
array[1..n] of int: app_start_indices = [sum([app_nr_tasks(j) | j in 1..i-1]) + 1 | i in 1..n
     ];

% The duration of each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_durations = [instr_max_durations[i,j] | i in 1..n,m in
     1..nr_executions[i], j in 1..nr_instructions[i]];

% The type of each task (ordered by application and instruction).
array[1..total_nr_tasks] of string: task_types = [instr_types[i,j] | i in 1..n,m in 1..nr_
     executions[i], j in 1..nr_instructions[i]];
```

```
% The application that belongs to each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_apps = [i | i in 1..n,m in 1..nr_executions[i], j in 1..
      nr_instructions[i]];

% Upper bound for task starttimes. This is the total duration + the maximal waiting time for
      QC tasks.
int: start_times_upper_bound = sum(task_durations) + L * g * count([task_types[i] == "QC" | i
      in 1..total_nr_tasks]);

% Decision variable: starttime for each task (ordered by application and instruction).
array[1..total_nr_tasks] of var 0..start_times_upper_bound: task_start_times;

% Constraint: For a same application, the previous task should be completed before starting a
      new task.
constraint forall(i in 2..total_nr_tasks where task_apps[i-1] == task_apps[i])(task_start_
      times[i] > task_start_times[i-1]);

% Constraint: no two tasks overlap.
constraint disjunctive(task_start_times, task_durations);

% Constraint: All 'QC' tasks should start in the first possible timeslots that it is the turn
      of the application.
constraint forall(i in 1..total_nr_tasks where task_types[i] == "QC")(task_start_times[i] ==
      qc_task_starttime(i));

% Constraint: All applications executions are finished within the maximal duration (deadline).
constraint forall(i in 1..n, m in 1..nr_executions[i])((task_end_time(get_exec_task_start_
      index(i, m+1) - 1) - task_start_times[get_exec_task_start_index(i, m)]) <= deadlines[i]);

% Objective function: minimize the makespan. Search variables with smallest value in domain
      and values ascendingly.
solve :: int_search(task_start_times, smallest, indomain) minimize max([task_end_time(i) | i
      in 1..total_nr_tasks]);

% Given the number of a task calculates the time it is finished (start time + duration).
function var int: task_end_time(int: task_index) = task_start_times[task_index] + task_
      durations[task_index];

% Finds the task index a given execution starts of a given application.
function int: get_exec_task_start_index(int: app_index, int: exec_nr) = app_start_indices[app_
      index] + (exec_nr - 1) * nr_instructions[app_index];

% Calculates the number of tasks for an application. For each instruction there is a task for
      each execution.
function int: app_nr_tasks(int: app_index) = nr_instructions[app_index] * nr_executions[app_
      index];

% Returns an array of all tasks indices which are the QC tasks for a particular app.
function array[int] of int: app_qc_tasks(int: app_index) = [i | i in 1..total_nr_tasks where
      task_apps[i] == app_index /\ task_types[i] == "QC"];

% Given the index of an application, returns the timeslot at which it is its turn for QC tasks
       in GS.
function int: app_gs_timeslot(int: app_index) = sum([if app_index == GS[i] then i else 0 endif
       | i in index_set(GS)]);

% Given the index of a QC task, returns the index it has in the QC tasks of the application.
function int: qc_app_task_nr(int: task_index) = sum([if task_index == app_qc_tasks(task_apps[
      task_index])[i] then i else 0 endif | i in index_set(app_qc_tasks(task_apps[task_index]))
      ]);

% Finds the time a QC task should start (the start of the next possible timeslot).
function int: qc_task_starttime(int: task_index) = L * ((app_gs_timeslot(task_apps[task_index
      ]) - 1) + g * (qc_app_task_nr(task_index) - 1));

% Print the resulting starttimes.
output ["\(task_start_times)"];
```

C
IF MODEL

```
% MiniZinc model that finds task starttimes for interleaving executions with a free network
     schedule.

include "globals.mzn";

% Number of applications.
int: n;

% The max. durations for each application.
array[1..n] of int: deadlines;

% The number of instructions for each application.
array[1..n] of int: nr_instructions;

% Number of executions for each application.
array[1..n] of int: nr_executions;

% Max execution durations and instruction types for each instruction (-1 and "" respectively
     if less instructions)
int: max_nr_instructions = max(nr_instructions);
array[1..n, 1..max_nr_instructions] of int: instr_max_durations;
array[1..n, 1..max_nr_instructions] of string: instr_types;

% Length of a timeslot in the network schedule.
int: L;

% Network schedule which contains timeslots when which application can perform 'QC' operations
     .
array[int] of int: GS;

% The length of the network schedule.
int: g = length(GS);

% The total number of tasks to schedule (each instruction execution of each application).
int: total_nr_tasks = sum([app_nr_tasks(i) | i in 1..n]);

% Array that contains for each application which task index contains the first task of that
     application.
array[1..n] of int: app_start_indices = [sum([app_nr_tasks(j) | j in 1..i-1]) + 1 | i in 1..n
     ];

% The duration of each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_durations = [instr_max_durations[i,j] | i in 1..n,m in
     1..nr_executions[i], j in 1..nr_instructions[i]];

% The type of each task (ordered by application and instruction).
array[1..total_nr_tasks] of string: task_types = [instr_types[i,j] | i in 1..n,m in 1..nr_
     executions[i], j in 1..nr_instructions[i]];
```

```
% The application that belongs to each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_apps = [i | i in 1..n,m in 1..nr_executions[i], j in 1..
     nr_instructions[i]];

% The execution number of the belonging application of each task.
array[1..total_nr_tasks] of int: task_ex_nrs = [m | i in 1..n,m in 1..nr_executions[i], j in
     1..nr_instructions[i]];

% Instruction index within the application of each task.
array[1..total_nr_tasks] of int: task_instr_nrs = [j | i in 1..n,m in 1..nr_executions[i], j
     in 1..nr_instructions[i]];

% Upper bound for task starttimes. This is the total duration + the maximal waiting time for
     QC tasks.
int: start_times_upper_bound = sum(task_durations) + L * g * count([task_types[i] == "QC" | i
     in 1..total_nr_tasks]);

% Decision variable: starttime for each task (ordered by application and instruction).
array[1..total_nr_tasks] of var 0..start_times_upper_bound: task_start_times;

% Constraint: Execute all tasks of the same execution of the same application in order.
constraint forall(i in 2..total_nr_tasks where task_apps[i-1] == task_apps[i] /\ task_ex_nrs[i
     -1] == task_ex_nrs[i])(task_start_times[i] > task_start_times[i-1]);

% Symmetry breaking constraint: For a same instruction of a same application, do the
     executions in order.
constraint symmetry_breaking_constraint(forall(i, j in 1..total_nr_tasks where i > j /\ task_
     apps[i] == task_apps[j] /\ task_instr_nrs[i] == task_instr_nrs[j])(task_start_times[i] >
     task_start_times[j]));

% Constraint: no two tasks overlap.
constraint disjunctive(task_start_times, task_durations);

% Constraint: For all 'QC' tasks the task starts and ends in the same timeslot it is the turn
     of the application.
constraint forall(i in 1..total_nr_tasks where task_types[i] == "QC")((GS[((gs_timeslot(i) mod
      g) + 1)] == task_apps[i]) /\ (((L * (gs_timeslot(i) + 1)) - task_start_times[i]) >= task
     _durations[i]));

% Constraint: All applications executions are finished within the maximal duration (deadline).
constraint forall(i in 1..n, m in 1..nr_executions[i])((task_end_time(get_exec_task_start_
     index(i, m+1) - 1) - task_start_times[get_exec_task_start_index(i, m)]) <= deadlines[i]);

% Objective function: minimize the makespan. Search variables with smallest value in domain
     and values ascendingly.
solve :: int_search(task_start_times, smallest, indomain) minimize max([task_end_time(i) | i
     in 1..total_nr_tasks]);

% Given the number of a task calculates the time it is finished (start time + duration).
function var int: task_end_time(int: task_index) = task_start_times[task_index] + task_
     durations[task_index];

% Calculates the absolute timeslot in the network schedule when a task starts. First timeslot
     has index 0.
function var int: gs_timeslot(int: task_index) = task_start_times[task_index] div L;

% Finds the task index a given execution starts of a given application.
function int: get_exec_task_start_index(int: app_index, int: exec_nr) = app_start_indices[app_
     index] + (exec_nr - 1) * nr_instructions[app_index];

% Calculates the number of tasks for an application. For each instruction there is a task for
     each execution.
function int: app_nr_tasks(int: app_index) = nr_instructions[app_index] * nr_executions[app_
     index];

% Print the resulting starttimes.
output ["\(task_start_times)"];
```

D
IS MODEL

```
% MiniZinc model that finds task starttimes for interleaving executions with a strict network
     schedule.

include "globals.mzn";

% Number of applications.
int: n;

% The max. durations for each application.
array[1..n] of int: deadlines;

% The number of instructions for each application.
array[1..n] of int: nr_instructions;

% Number of executions for each application.
array[1..n] of int: nr_executions;

% Max execution durations and instruction types for each instruction (-1 and "" respectively
     if less instructions)
int: max_nr_instructions = max(nr_instructions);
array[1..n, 1..max_nr_instructions] of int: instr_max_durations;
array[1..n, 1..max_nr_instructions] of string: instr_types;

% Length of a timeslot in the network schedule.
int: L;

% Network schedule which contains timeslots when which application can perform 'QC' operations
     .
array[int] of int: GS;

% The length of the network schedule.
int: g = length(GS);

% The total number of tasks to schedule (each instruction execution of each application).
int: total_nr_tasks = sum([app_nr_tasks(i) | i in 1..n]);

% Array that contains for each application which task index contains the first task of that
     application.
array[1..n] of int: app_start_indices = [sum([app_nr_tasks(j) | j in 1..i-1]) + 1 | i in 1..n
     ];

% The duration of each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_durations = [instr_max_durations[i,j] | i in 1..n,m in
     1..nr_executions[i], j in 1..nr_instructions[i]];

% The type of each task (ordered by application and instruction).
array[1..total_nr_tasks] of string: task_types = [instr_types[i,j] | i in 1..n,m in 1..nr_
     executions[i], j in 1..nr_instructions[i]];
```

```
% The application that belongs to each task (ordered by application and instruction).
array[1..total_nr_tasks] of int: task_apps = [i | i in 1..n,m in 1..nr_executions[i], j in 1..
      nr_instructions[i]];

% The execution number of the belonging application of each task.
array[1..total_nr_tasks] of int: task_ex_nrs = [m | i in 1..n,m in 1..nr_executions[i], j in
      1..nr_instructions[i]];

% Instruction index within the application of each task.
array[1..total_nr_tasks] of int: task_instr_nrs = [j | i in 1..n,m in 1..nr_executions[i], j
      in 1..nr_instructions[i]];

% Upper bound for task starttimes. This is the total duration + the maximal waiting time for
      QC tasks.
int: start_times_upper_bound = sum(task_durations) + L * g * count([task_types[i] == "QC" | i
      in 1..total_nr_tasks]);

% Decision variable: starttime for each task (ordered by application and instruction).
array[1..total_nr_tasks] of var 0..start_times_upper_bound: task_start_times;

% Constraint: Execute all tasks of the same execution of the same application in order.
constraint forall(i in 2..total_nr_tasks where task_apps[i-1] == task_apps[i] /\ task_ex_nrs[i
      -1] == task_ex_nrs[i])(task_start_times[i] > task_start_times[i-1]);

% Symmetry breaking constraint: For a same instruction of a same application, do the
      executions in order.
constraint symmetry_breaking_constraint(forall(i, j in 1..total_nr_tasks where i > j /\ task_
      apps[i] == task_apps[j] /\ task_instr_nrs[i] == task_instr_nrs[j])(task_start_times[i] >
      task_start_times[j]));

% Constraint: no two tasks overlap.
constraint disjunctive(task_start_times, task_durations);

% Constraint: All 'QC' tasks should start in the first possible timeslots that it is the turn
      of the application.
constraint forall(i in 1..total_nr_tasks where task_types[i] == "QC")(task_start_times[i] ==
      qc_task_starttime(i));

% Constraint: All applications executions are finished within the maximal duration (deadline).
constraint forall(i in 1..n, m in 1..nr_executions[i])((task_end_time(get_exec_task_start_
      index(i, m+1) - 1) - task_start_times[get_exec_task_start_index(i, m)]) <= deadlines[i]);

% Objective function: minimize the makespan. Search variables with smallest value in domain
      and values ascendingly.
solve :: int_search(task_start_times, smallest, indomain) minimize max([task_end_time(i) | i
      in 1..total_nr_tasks]);

% Given the number of a task calculates the time it is finished (start time + duration).
function var int: task_end_time(int: task_index) = task_start_times[task_index] + task_
      durations[task_index];

% Finds the task index a given execution starts of a given application.
function int: get_exec_task_start_index(int: app_index, int: exec_nr) = app_start_indices[app_
      index] + (exec_nr - 1) * nr_instructions[app_index];

% Calculates the number of tasks for an application. For each instruction there is a task for
      each execution.
function int: app_nr_tasks(int: app_index) = nr_instructions[app_index] * nr_executions[app_
      index];

% Returns an array of all tasks indices which are the QC tasks for a particular app.
function array[int] of int: app_qc_tasks(int: app_index) = [i | i in 1..total_nr_tasks where
      task_apps[i] == app_index /\ task_types[i] == "QC"];

% Given the index of an application, returns the timeslot at which it is its turn for QC tasks
       in GS.
function int: app_gs_timeslot(int: app_index) = sum([if app_index == GS[i] then i else 0 endif
       | i in index_set(GS)]);

% Given the index of a QC task, returns the index it has in the QC tasks of the application.
function int: qc_app_task_nr(int: task_index) = sum([if task_index == app_qc_tasks(task_apps[
      task_index])[i] then i else 0 endif | i in index_set(app_qc_tasks(task_apps[task_index]))
      ]);

% Finds the time a QC task should start (the start of the next possible timeslot).
function int: qc_task_starttime(int: task_index) = L * ((app_gs_timeslot(task_apps[task_index
      ]) - 1) + g * (qc_app_task_nr(task_index) - 1));
```

```
% Print the resulting starttimes.
output ["\(task_start_times)"];
```

E

MANY EXECUTIONS DATA FILE

```
%
% Instance where we have a few smaller applications, which are executed many times.
%


% Network schedule
GS = [1, 2];

% Length of a timeslot in the network schedule
L = 600;

% Number of appplications
n = 2;

% Deadline for each application (max duration)
deadlines = [ 700, 700];

% The number of times we execute each application
nr_executions = [ 19,64];

% Number of instructions for each application
nr_instructions = [ 2,2];

% The maximal duration of each instruction.
% Each application on a new row.
% -1 as dummies for applications with less instructions than the largest application.
instr_max_durations = [|100, 200
|100, 200|];

% The types of each instruction.
% Each application on a new row.
% "" as dummies for applications with less instructions than the largest application.
instr_types = [|"QC", "CC"
|"QC", "CC"|];
```

F
Many instructions data file

```
%
% Instance where we have a few bigger applications, which are executed only few times.
%


% Network schedule
GS = [1, 2];

% Length of a timeslot in the network schedule
L = 1000;

% Number of appplications
n = 2;

% Deadline for each application (max duration)
deadlines = [ 1600,1600];

% The number of times we execute each application
nr_executions = [ 2, 1];

% Number of instructions for each application
nr_instructions = [ 12,12];

% The maximal duration of each instruction.
% Each application on a new row.
% -1 as dummies for applications with less instructions than the largest application.
instr_max_durations = [|100, 200, 5, 5, 200, 200, 5, 200, 5,5,200,5
|100, 200, 5, 5, 200, 200, 5, 200, 5,5,200, 5|];

% The types of each instruction.
% Each application on a new row.
% "" as dummies for applications with less instructions than the largest application.
instr_types = [|"QC", "CC", "QL", "QL", "CC", "CC", "QL", "CC", "QL", "QL","QL", "QL"
|"QC", "CC", "QL", "QL", "CC", "CC", "QL", "CC", "QL", "QL","CC", "QL"|];
```

G
MANY APPLICATIONS DATA FILE

```
%
% Instance where we have many smaller applications, which are executed only few times.
%


% Network schedule
GS = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
20,21,22,23,24,25,26,27,28,29,30];

% Length of a timeslot in the network schedule
L = 800;

% Number of appplications
n = 30;

% Deadline for each application (max duration)
deadlines = [ 1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,
1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,1000,
1000,1000,1000,1000,1000 ];

% The number of times we execute each application
nr_executions = [ 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2 ];

% Number of instructions for each application
nr_instructions = [ 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2 ];

% The maximal duration of each instruction.
% Each application on a new row.
% -1 as dummies for applications with less instructions than the largest application.
instr_max_durations = [|100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200
                       |100, 200|];

% The types of each instruction.
% Each application on a new row.
% "" as dummies for applications with less instructions than the largest application.
instr_types = [|"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"
               |"QC", "CC"|];
```

H
Simulated BQC-server instructions

Instruction	Type	Duration (μs)
1*	CL	200
2*	QC	30000
3*	CL	200
4*	QC	30000
5	CL	200
6**	QL	100
7**	QL	100
8**	QL	100
9	CL	200
10	CC	2000
11	CL	200
12**	QL	100
13**	QL	1
14**	QL	1
15**	QL	10
16**	QL	100
17	CC	2000
18	CC	2000
19	CL	200
20**	QL	100
21**	QL	1
22**	QL	1
23**	QL	10

24**	QL	100
25	CC	2000
Total		69924

* Instructions 1 and 2 were made 1 single QC instruction, summing both durations. The same was done for instructions 3 and 4. This was done for both the baseline and CP models. This was done because the CL instructions are related to the QC tasks (used for set-up) and because the IS and SS model constraints require to start with a QC instruction.

** The subsequent QL instructions were made 1 single QL instruction, summing the individual durations. This was done specifically for the CP models, in order to reduce the total number of instructions, else the input would be too complex to find a schedule in a short amount of time for more clients and executions. The grouping of QL operations would not change anything for the schedule produced by the baseline scheduler, since it always performs QL tasks directly.