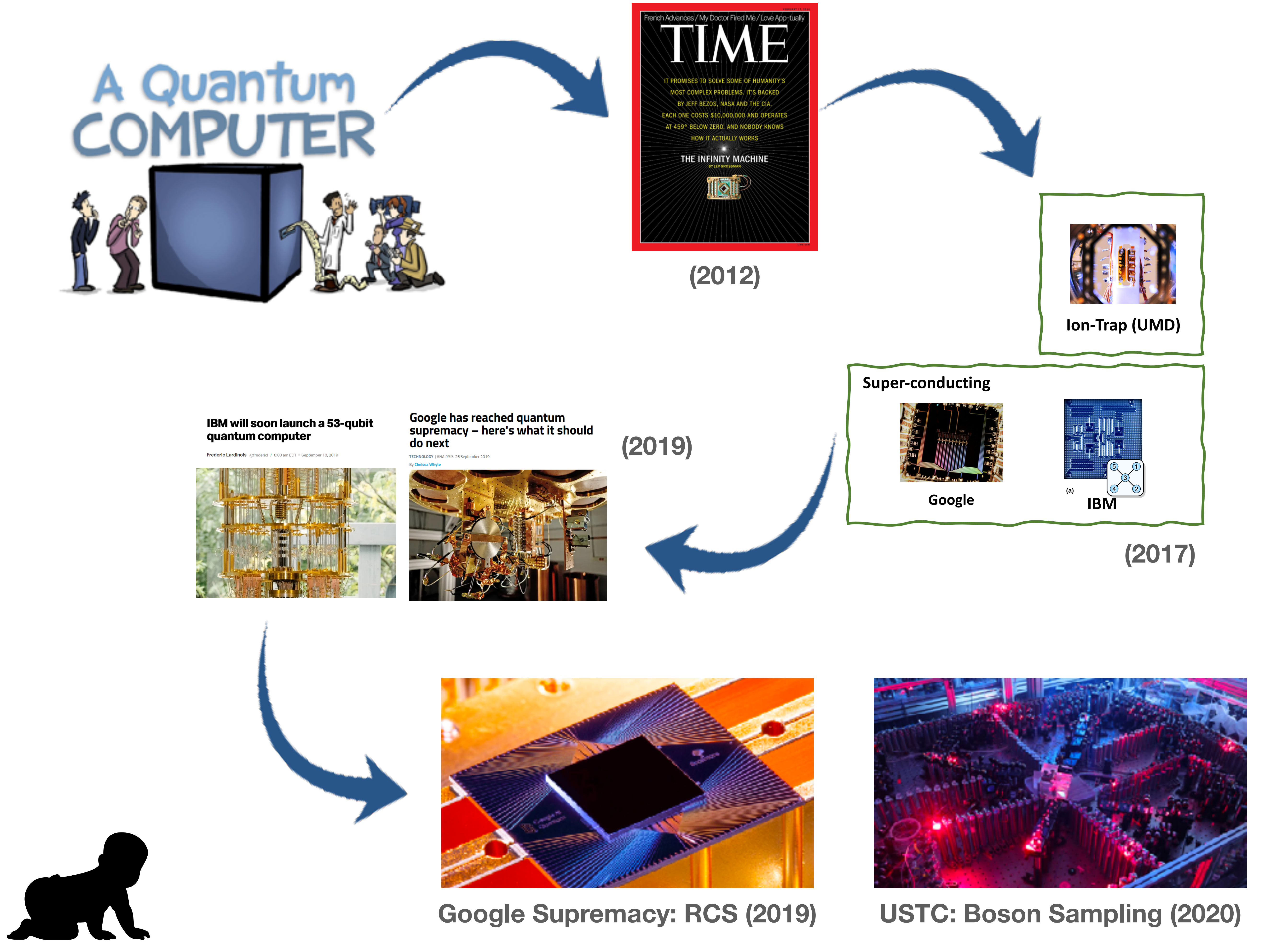
Computational Thinking toward End-to-End Quantum Applications

Xiaodi Wu
QuICS & UMD

UNIVERSITY OF MARYLAND

JOINT CENTER FOR QUANTUM INFORMATION AND COMPUTER SCIENCE


A Quantum COMPUTER
French Advances / My Doctor Fired Me / Love App-tually
TIME
IT PROMISES TO SOLVE SOME OF HUMANITY'S MOST COMPLEX PROBLEMS. IT'S BACKED BY JEFF BEZOS, NASA AND THE CIA. EACH ONE COSTS $10,000,000 AND OPERATES AT 459° BELOW ZERO. AND NOBODY KNOWS HOW IT ACTUALLY WORKS
THE INFINITY MACHINE
(2012)
Ion-Trap (UMD)
Super-conducting
Google
IBM
(2017)
IBM will soon launch a 53-qubit quantum computer
Google has reached quantum supremacy – here's what it should do next
(2019)
Google Supremacy: RCS (2019)
USTC: Boson Sampling (2020)

Computational Thinking

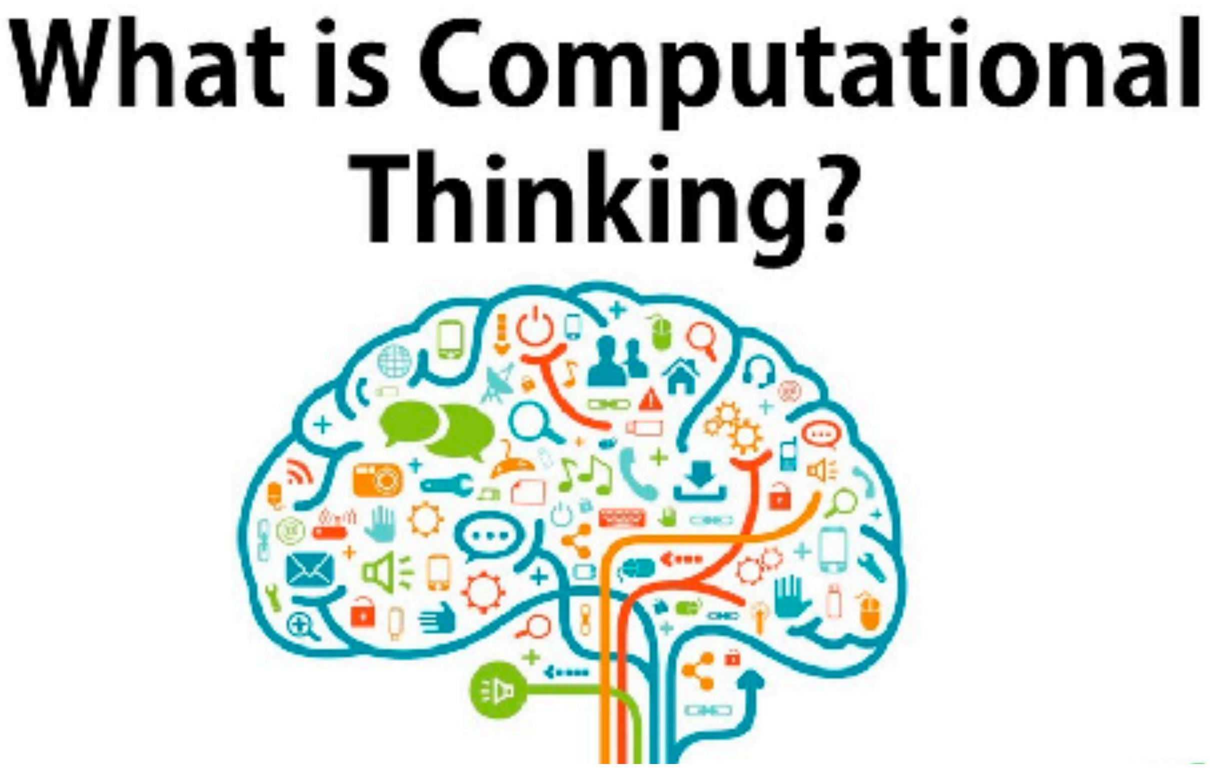
Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.

- Jeannette M. Wing, “Computational Thinking”, CACM Viewpoint, March 2006

Computational Thinking

Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.

- Jeannette M. Wing, "Computational Thinking", CACM Viewpoint, March 2006



First **A - Abstractions:** "metal" tools

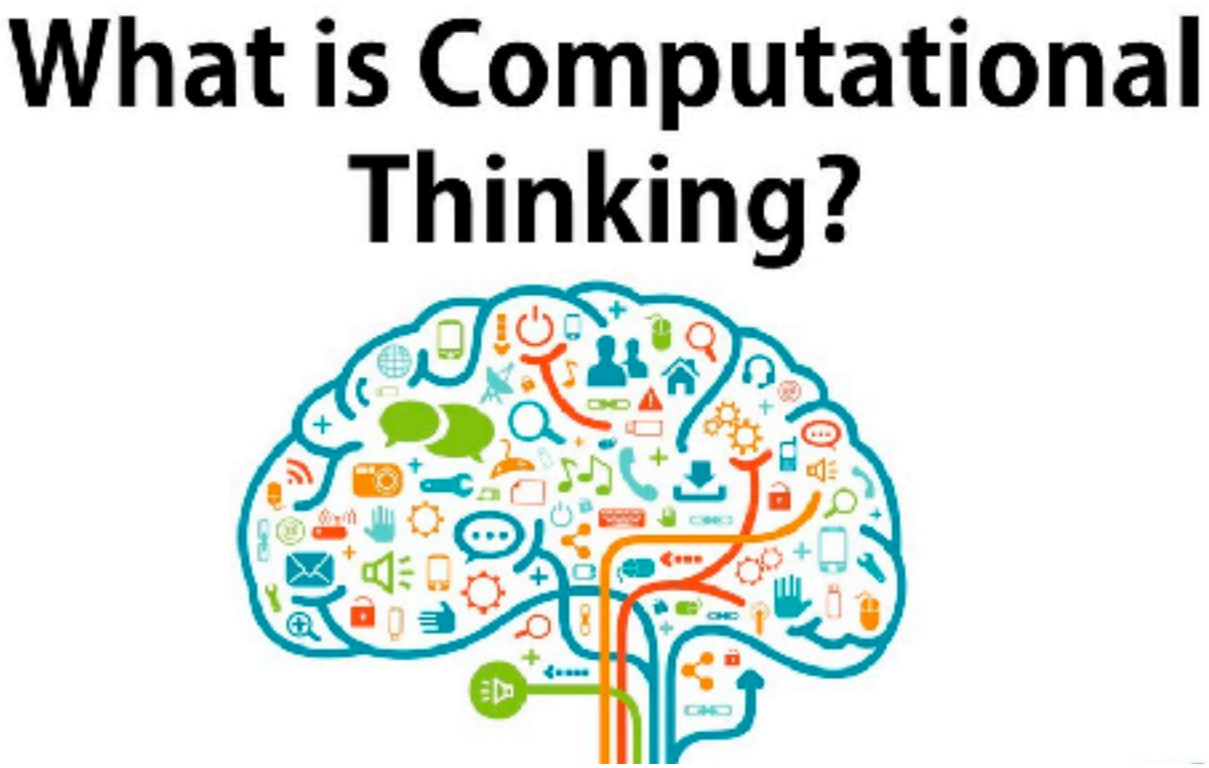
Second **A - Automation:** mechanizing abstractions and their relationships

Two **A**'s for Computational Thinking

Computational Thinking

Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.

- Jeannette M. Wing, "Computational Thinking", CACM Viewpoint, March 2006



First **A - Abstractions:** "metal" tools

Second **A - Automation:** mechanizing abstractions and their relationships

Computing: **Automation** of **Abstractions**

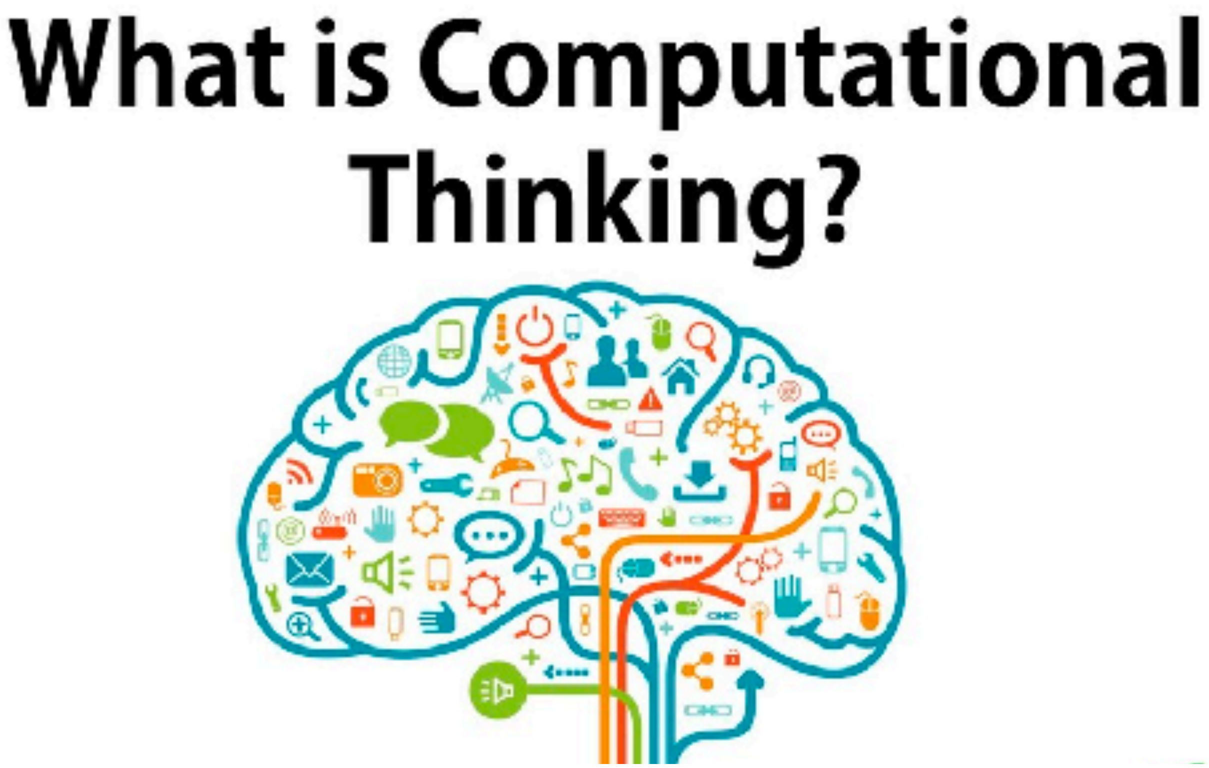
- They give us the audacity and ability to scale

Two **A**'s for Computational Thinking

Computational Thinking

Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.

- Jeannette M. Wing, “Computational Thinking”, CACM Viewpoint, March 2006



Two **A**’s for Computational Thinking

First **A - Abstractions:** “metal” tools

Second **A - Automation:** mechanizing abstractions and their relationships

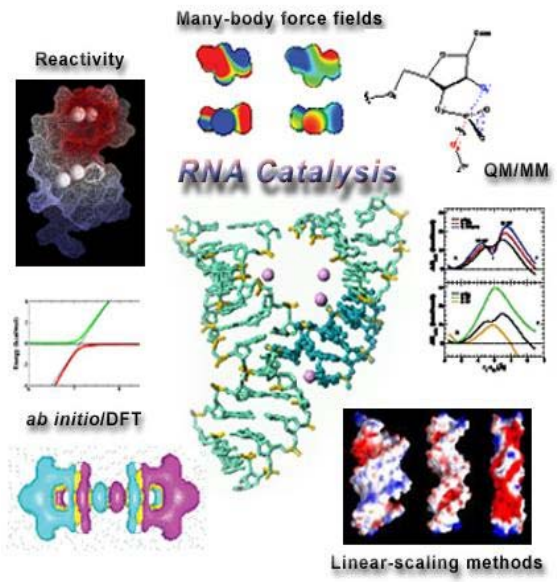
Computing: **Automation** of **Abstractions**

- They give us the audacity and ability to scale

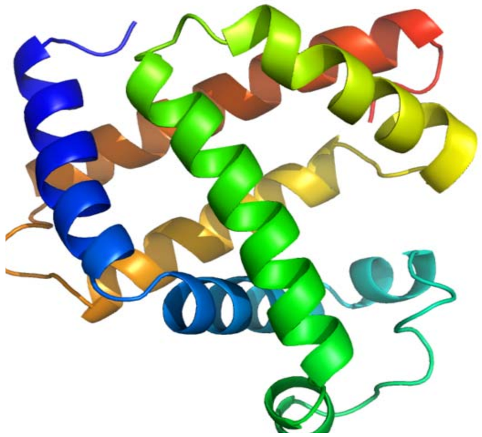
Computational Thinking

- choosing the right abstractions
- choosing the right automation or “computer”

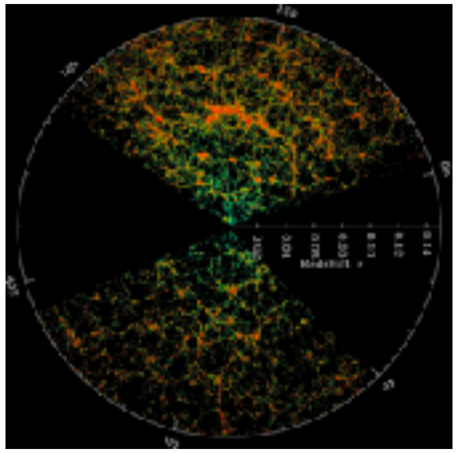
Computational Thinking: everywhere!



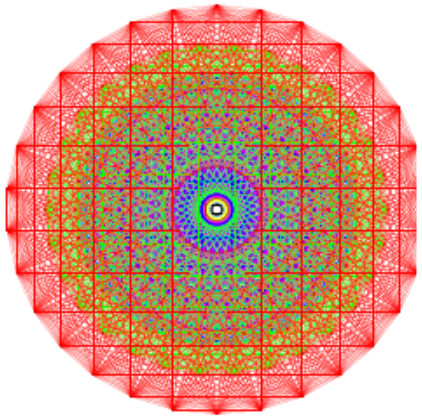
Chemistry: atomistic calculation, optimization over reaction conditions …



Biology: DNA sequencing
Protein structures, …



Astronomy: Sloan Digital Sky Server, …



Mathematics: E8 Lie group, four-color theorem proof

Economics: Automated mechanism design

MANY MORE :

- social science
- medicine
- art
- law
- entertainment
- sports
- …

- Jeannette M. Wing, "Computational Thinking", CACM Viewpoint, March 2006

Computational Thinking in Quantum Computing

How difficult is the problem and how best can I solve it with quantum computers?

Computational Thinking in Quantum Computing

Quantum
Application

Algorithm & Complexity

How difficult is the problem and how best can I solve it with quantum computers?

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

How difficult is the problem and how best can I solve it with quantum computers?

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

How to effectively express quantum applications and do trouble shooting?

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

Programming Languages

How to effectively express quantum applications and do trouble shooting?

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

Programming Languages

How to effectively translate high-level descriptions of quantum applications to quantum machine instructions?

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

Programming Languages

System

How to effectively translate high-level descriptions of quantum applications to quantum machine instructions?

Archite
cture

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

Programming Languages

System

Archite
cture

How to optimize the manipulation of quantum machines in terms of metrics like noise, power, …?

Computational Thinking in Quantum Computing

Quantum Application

Algorithm & Complexity

Variational Methods

Programming Languages

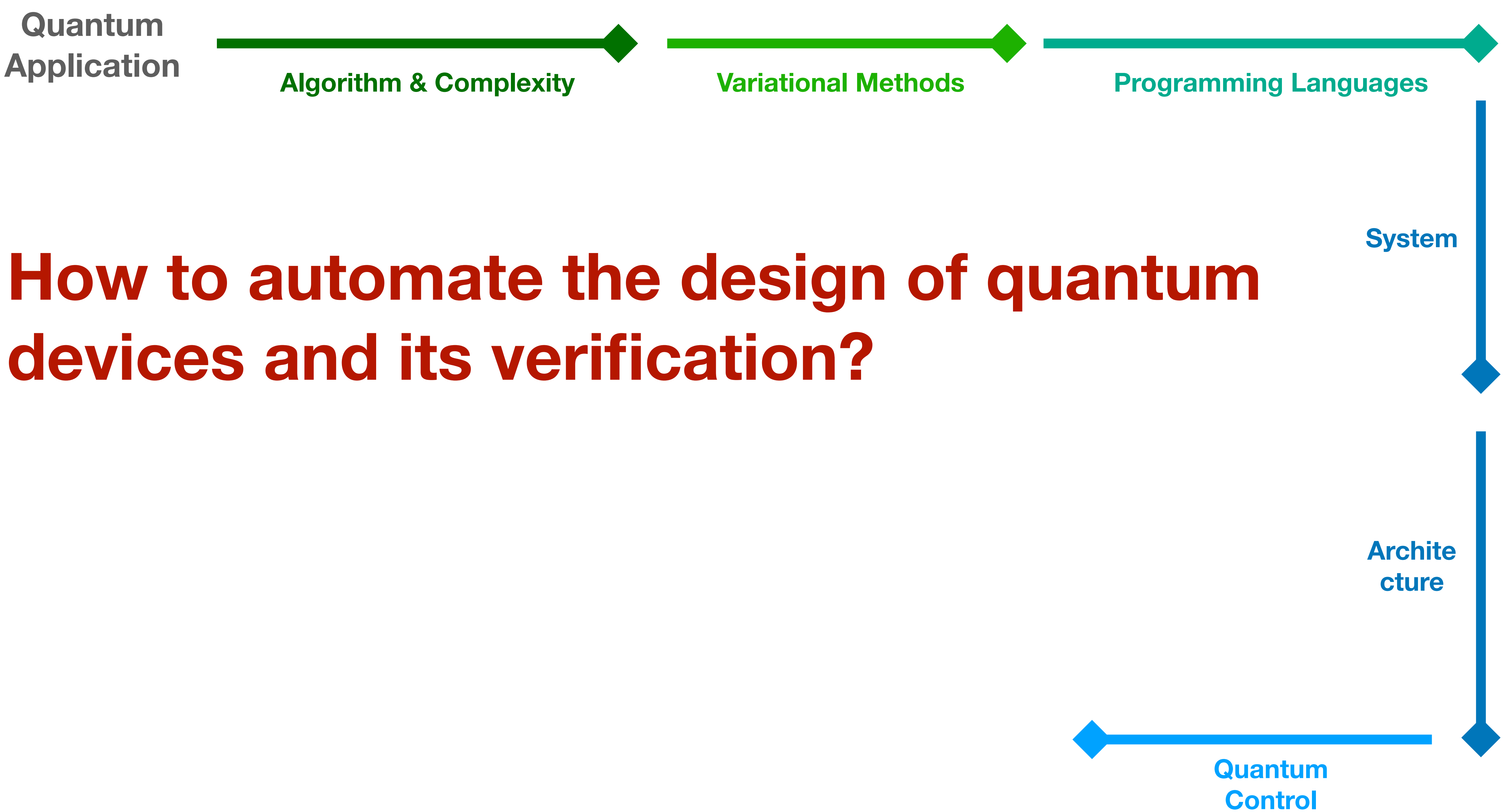
System

Architecture

Quantum Control

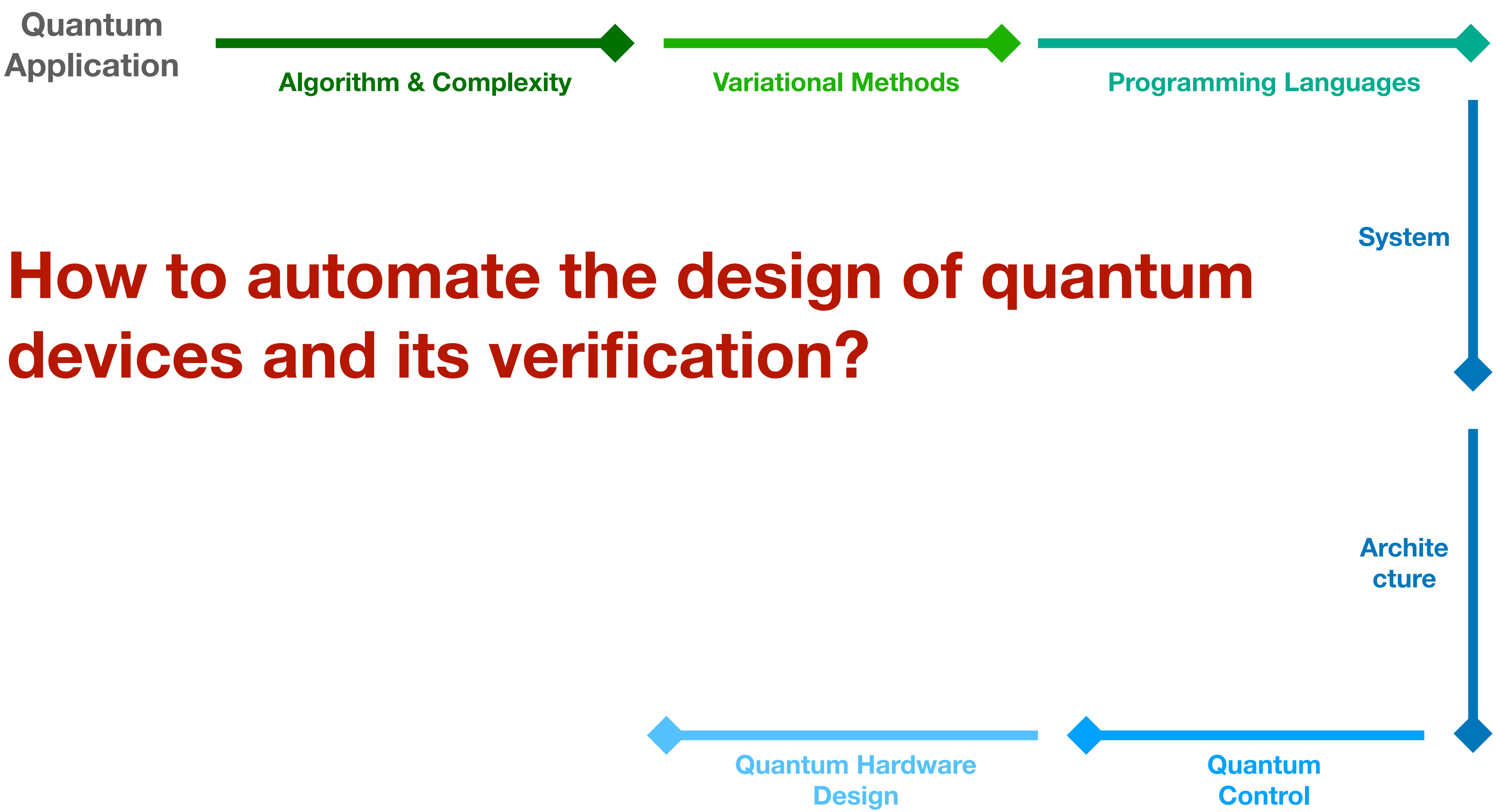
How to optimize the manipulation of quantum machines in terms of metrics like noise, power, …?

Computational Thinking in Quantum Computing



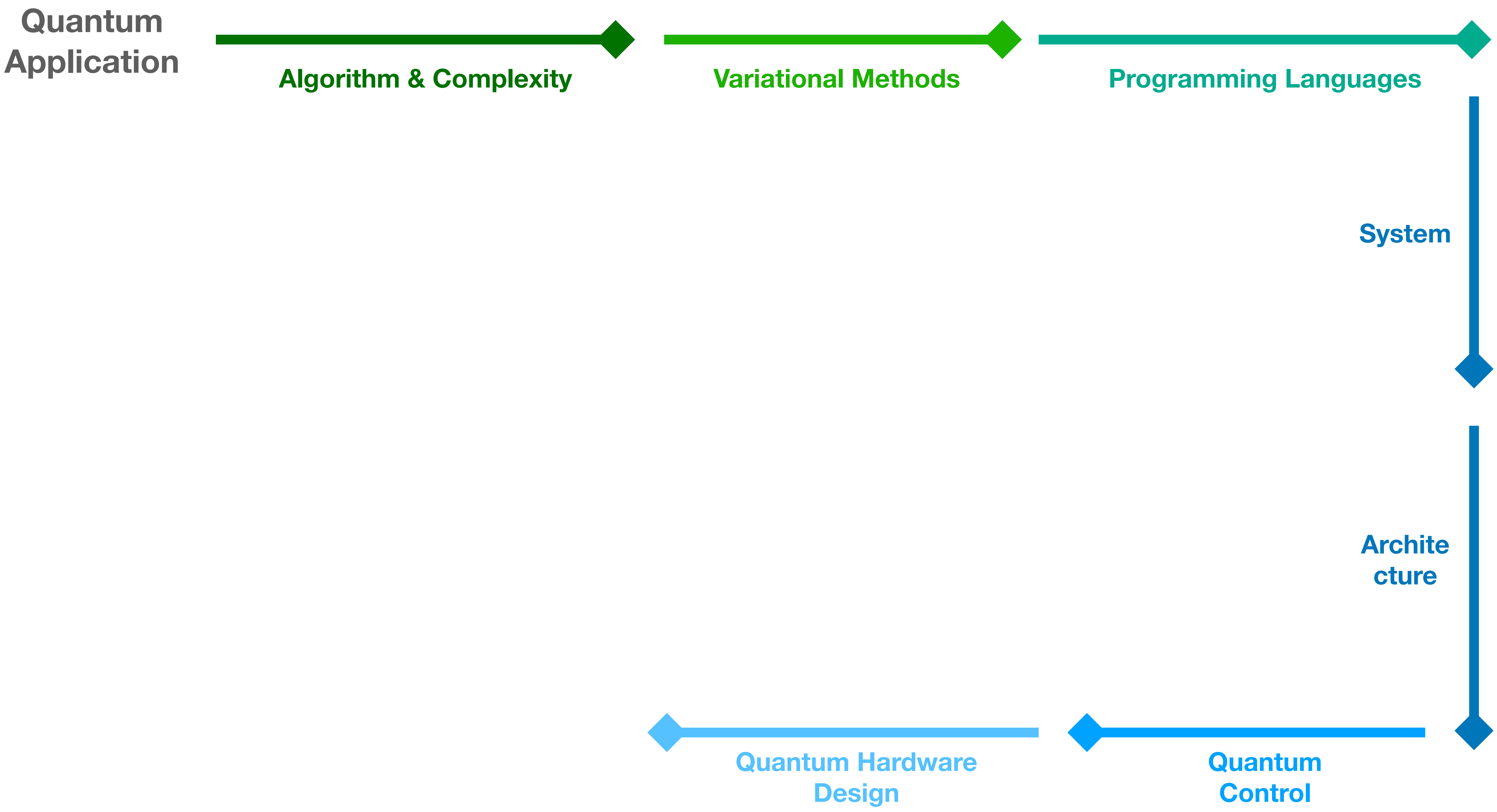
How to automate the design of quantum devices and its verification?

Computational Thinking in Quantum Computing

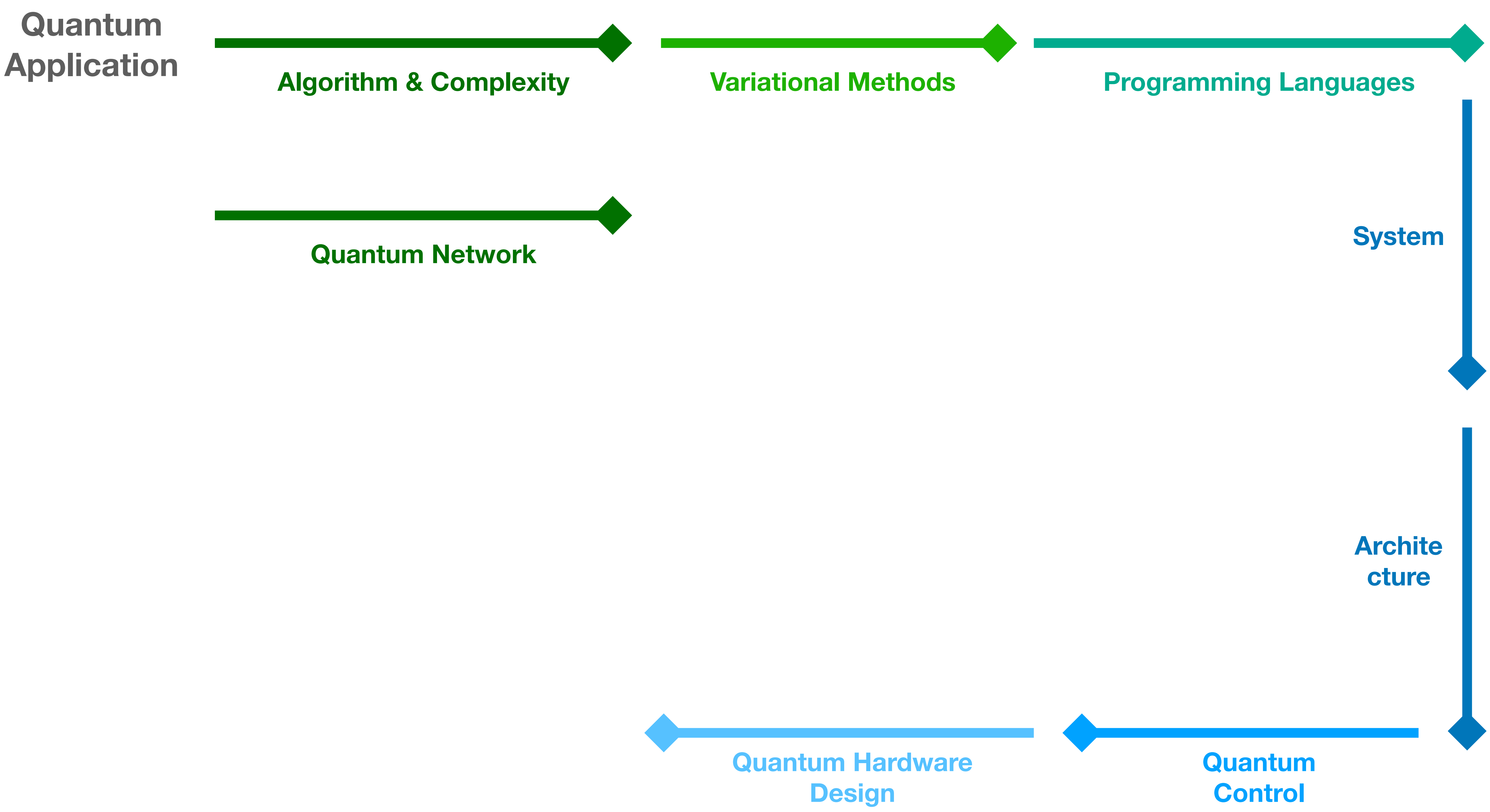


How to automate the design of quantum devices and its verification?

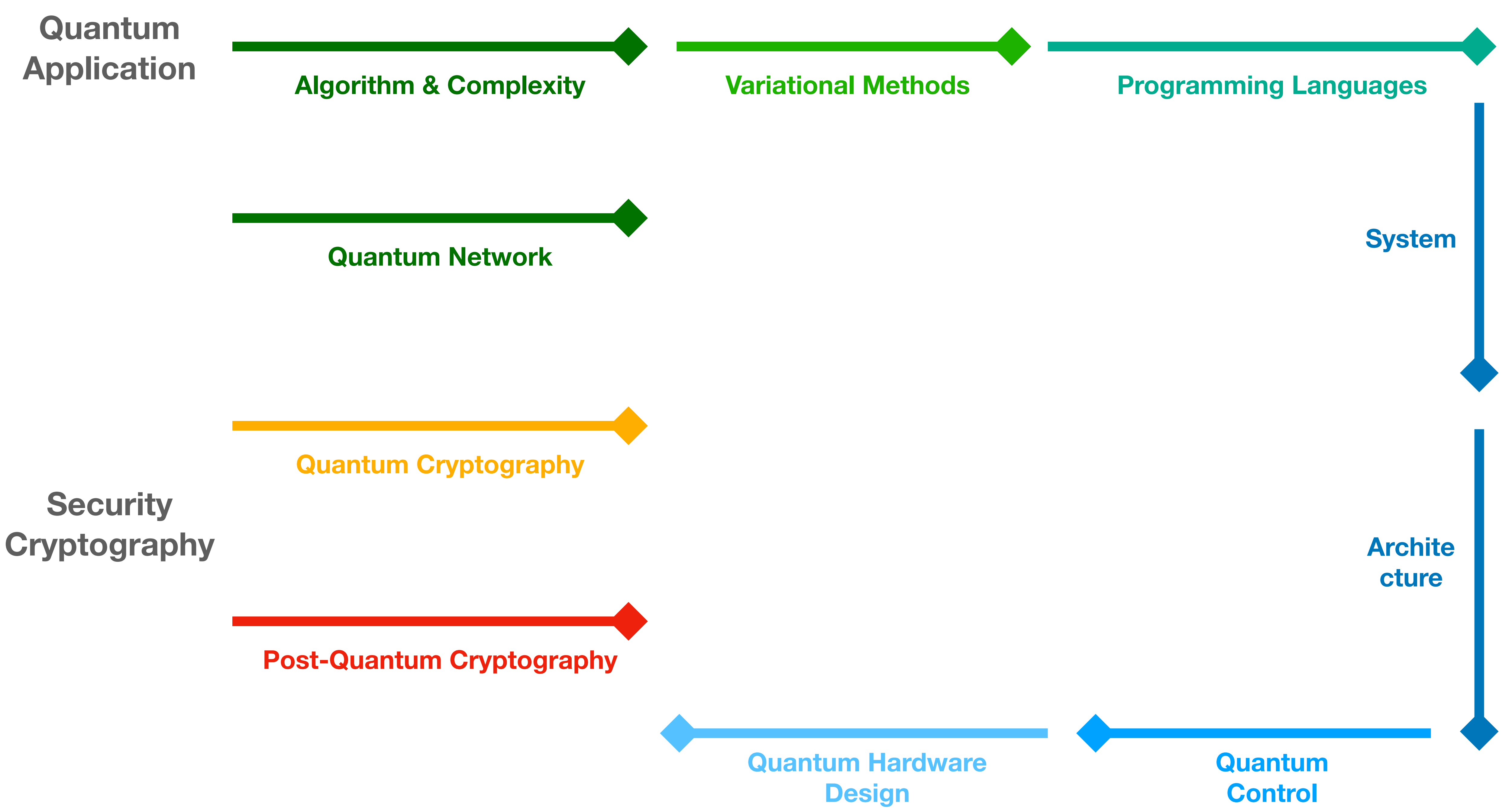
Computational Thinking in Quantum Computing



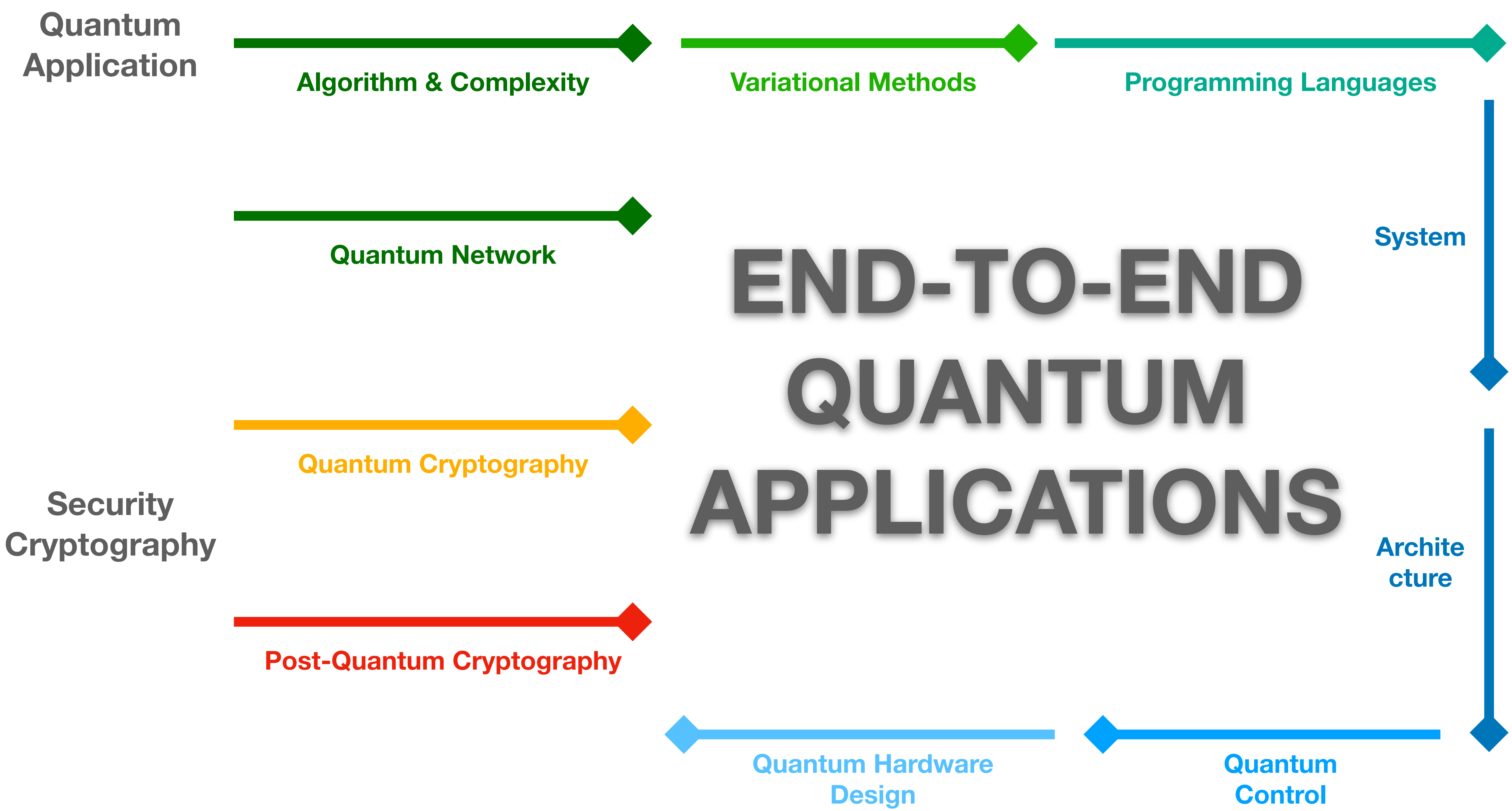
Computational Thinking in Quantum Computing



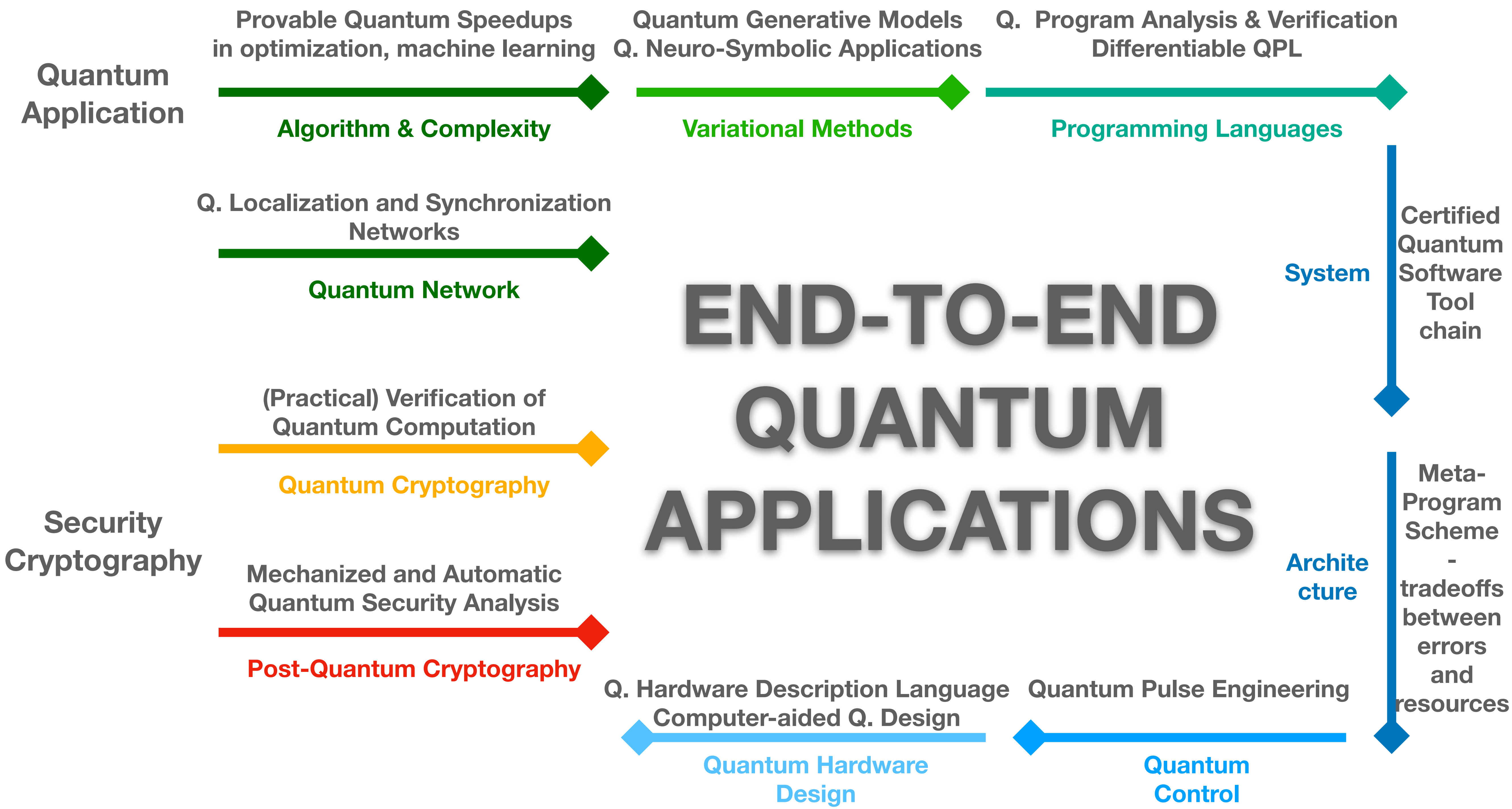
Computational Thinking in Quantum Computing



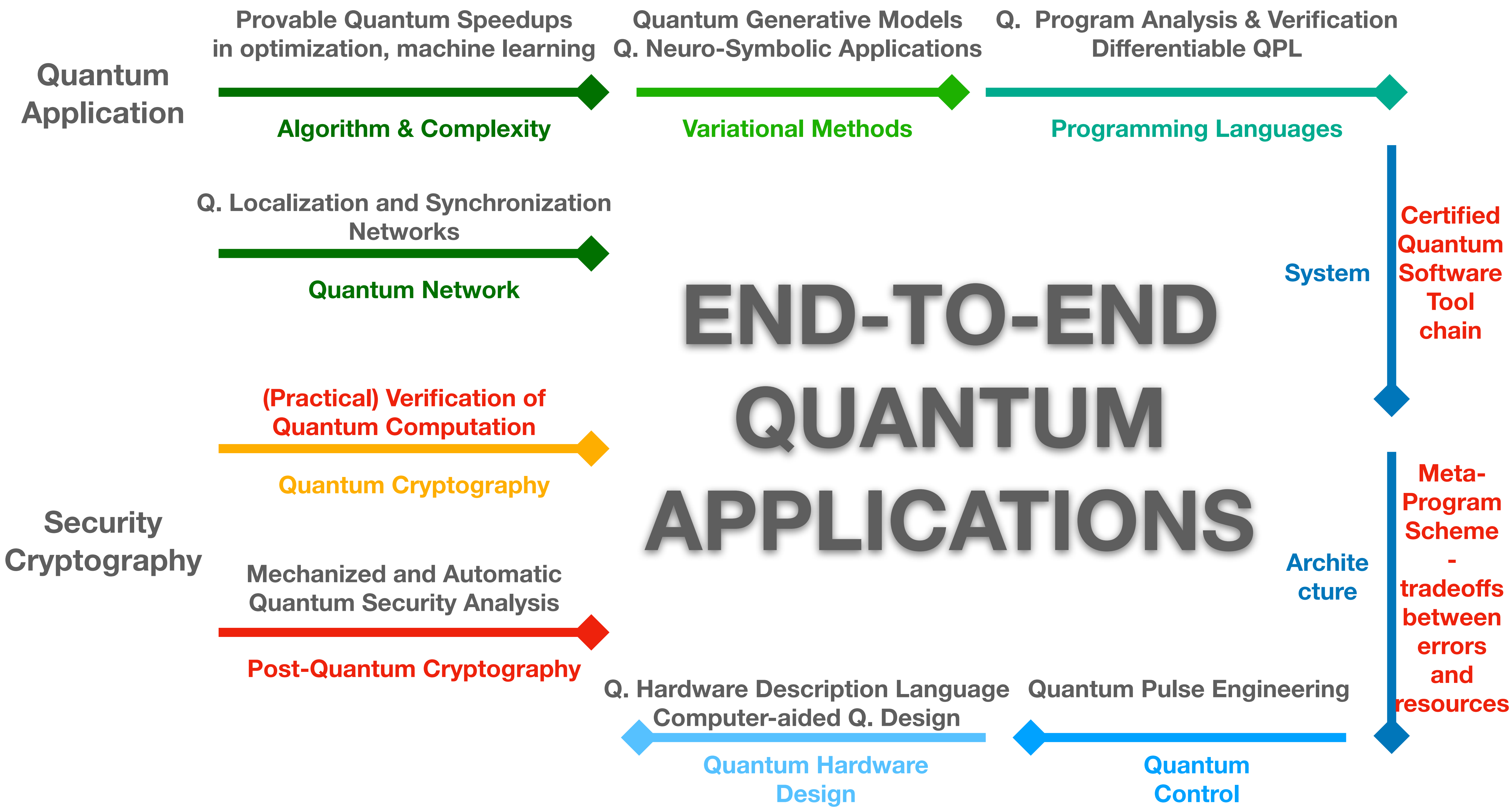
Computational Thinking in Quantum Computing



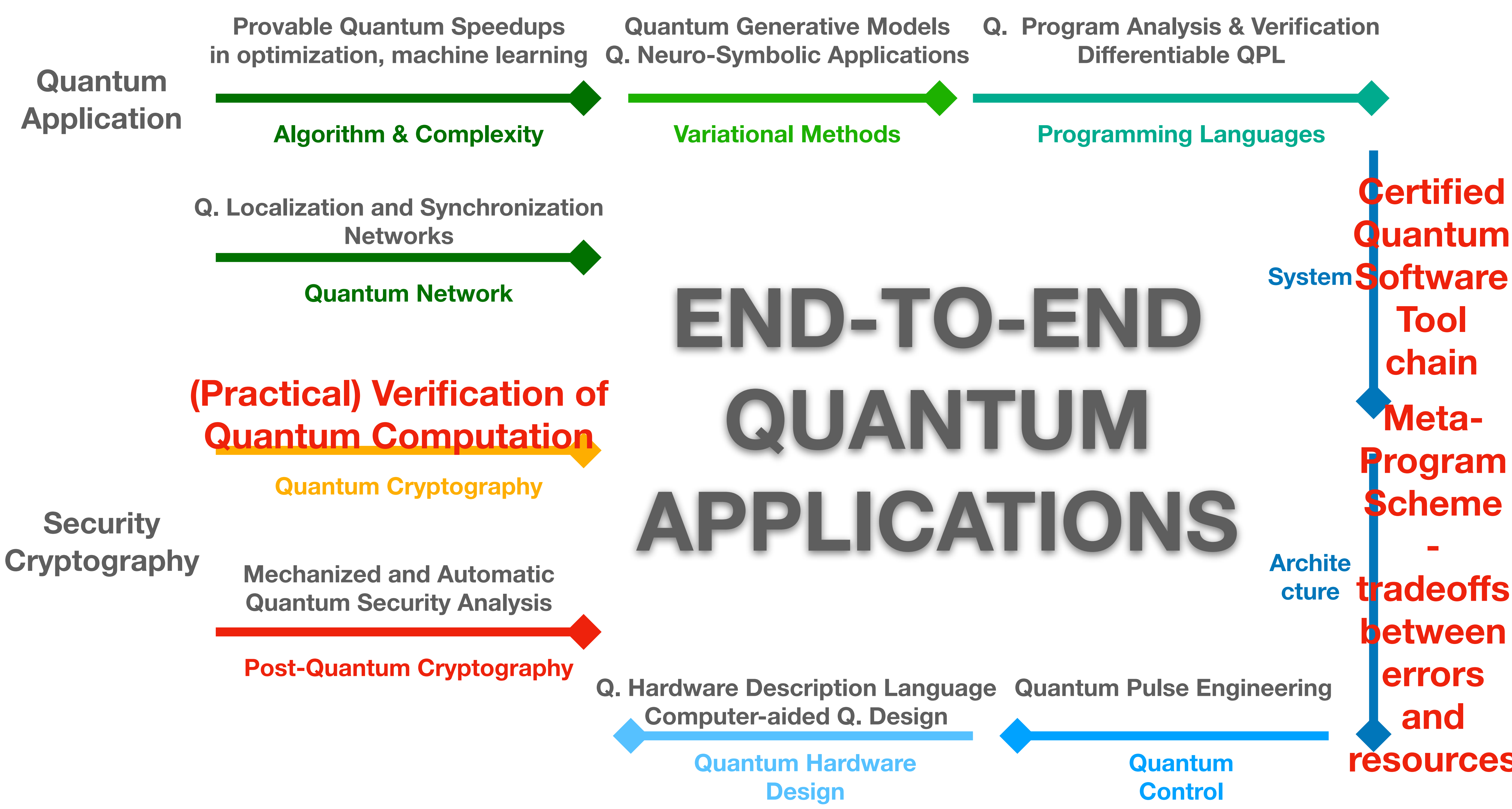
Computational Thinking in Quantum Computing



Computational Thinking in Quantum Computing

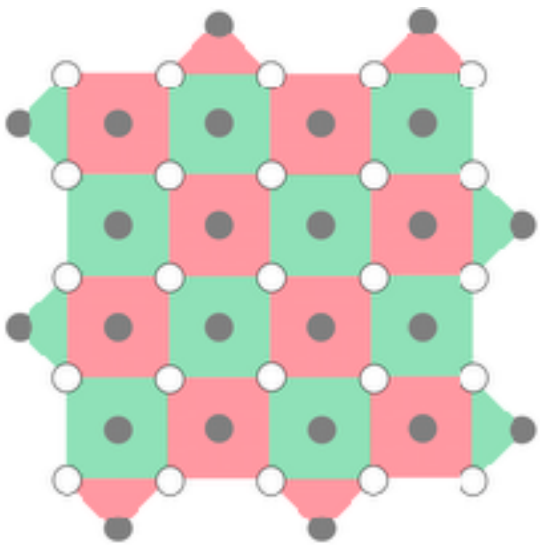


Computational Thinking in Quantum Computing

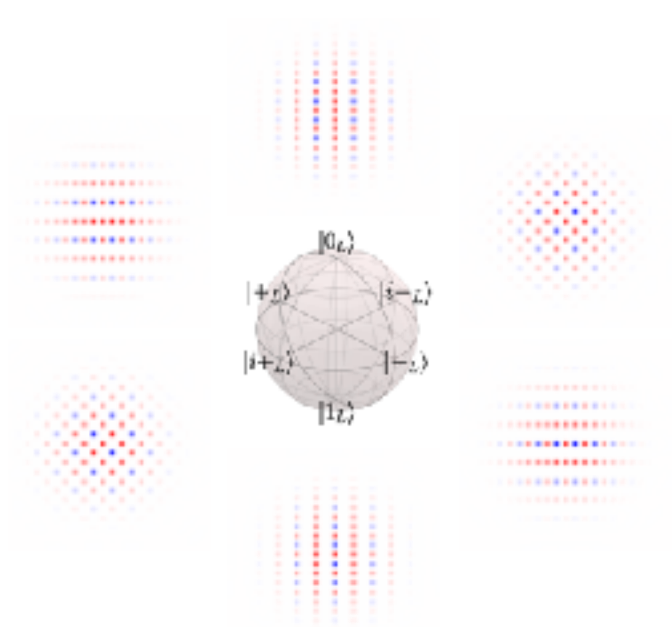


ERROR

Nature



Quantum Error Correction
Fight
Quantum Decoherence



ERROR

Features of NISQ Application Design

NISQ machines: very *restricted* hardware resources, where precisely controllable qubits are *expensive, error-prone, and scarce*.

Features of NISQ Application Design

NISQ machines: very *restricted* hardware resources, where precisely controllable qubits are *expensive, error-prone, and scarce*.

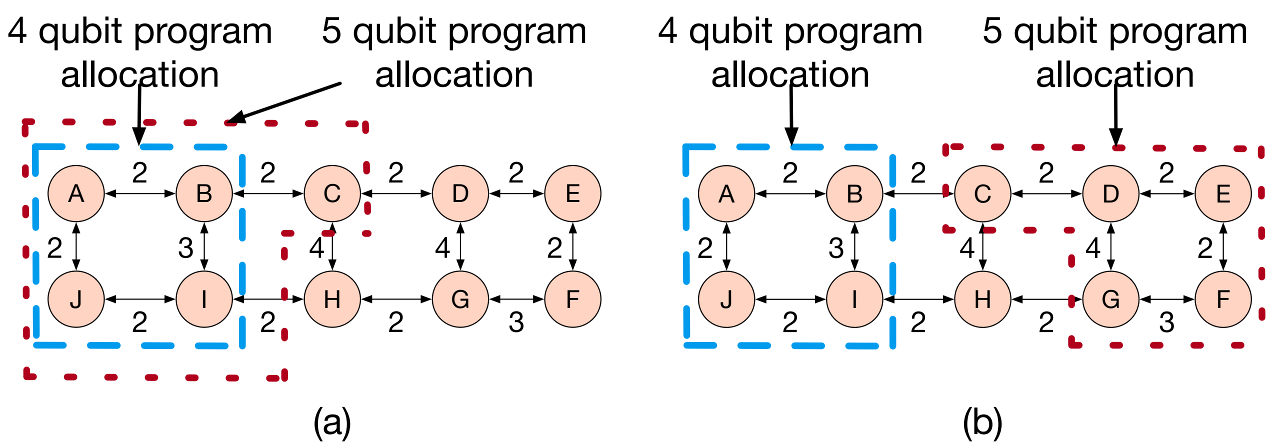
NISQ application design: investigate the best balance of **trade-offs** among a large number of (potentially heterogeneous) factors specific to the **targeted application** and **quantum hardware.**

Features of NISQ Application Design

NISQ machines: very *restricted* hardware resources, where precisely controllable qubits are *expensive, error-prone, and scarce*.

NISQ application design: investigate the best balance of **trade-offs** among a large number of (potentially heterogeneous) factors specific to the **targeted application** and **quantum hardware.**

Multi-Programming (MICRO 2019) **:**

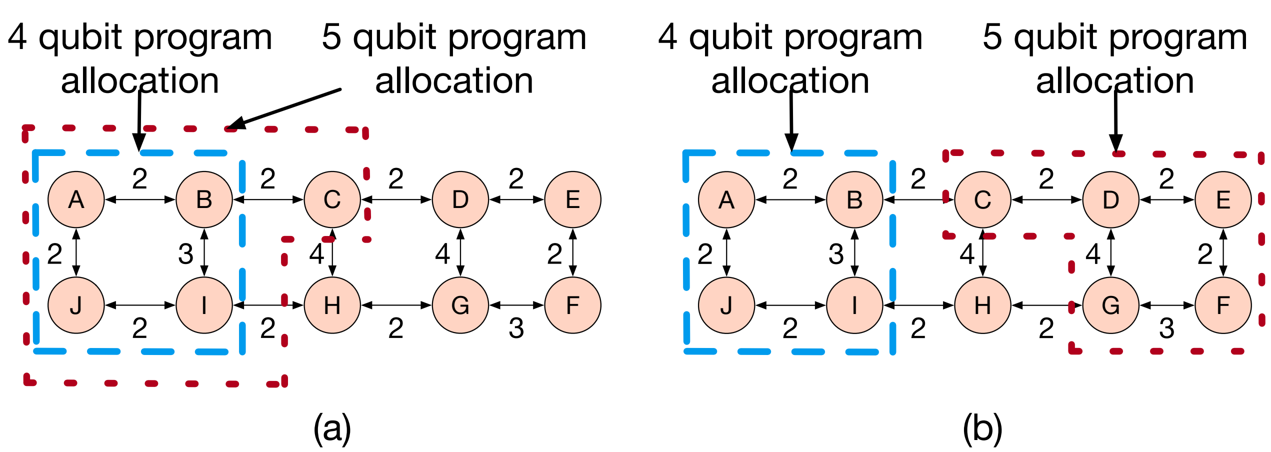


Features of NISQ Application Design

NISQ machines: very *restricted* hardware resources, where precisely controllable qubits are *expensive, error-prone, and scarce*.

NISQ application design: investigate the best balance of **trade-offs** among a large number of (potentially heterogeneous) factors specific to the **targeted application** and **quantum hardware.**

Multi-Programming (MICRO 2019) **:**



Competing Goals:

(1) Fully leverage qubits & Shorten the total execution
=> Multi-Programming

(2) High Reliability => Use the best qubits
=> Sequentially Allocate Programs

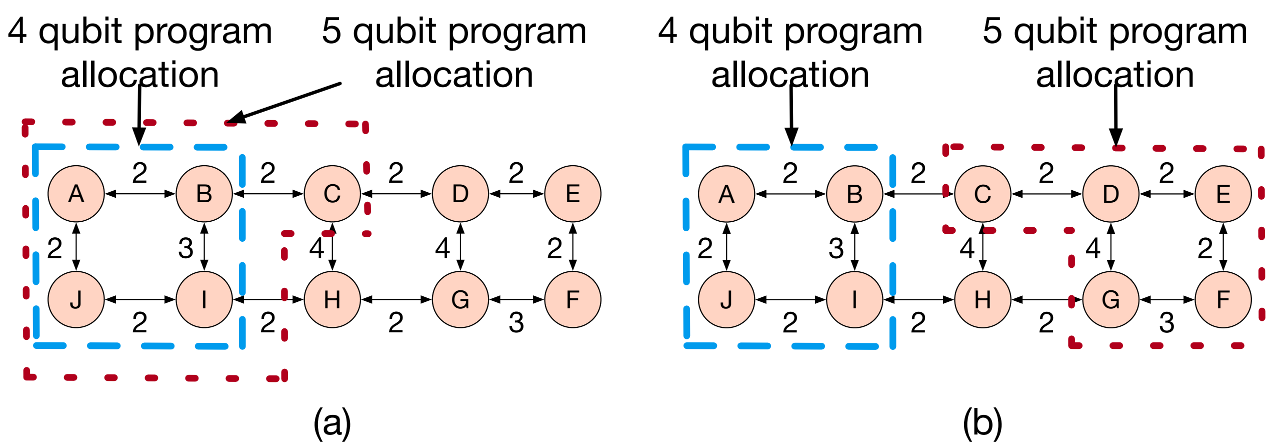
Solution: A run-time trade-off between these competing goals**.**

Features of NISQ Application Design

NISQ machines: very *restricted* hardware resources, where precisely controllable qubits are *expensive, error-prone, and scarce*.

NISQ application design: investigate the best balance of **trade-offs** among a large number of (potentially heterogeneous) factors specific to the **targeted application** and **quantum hardware.**

Multi-Programming (MICRO 2019) :



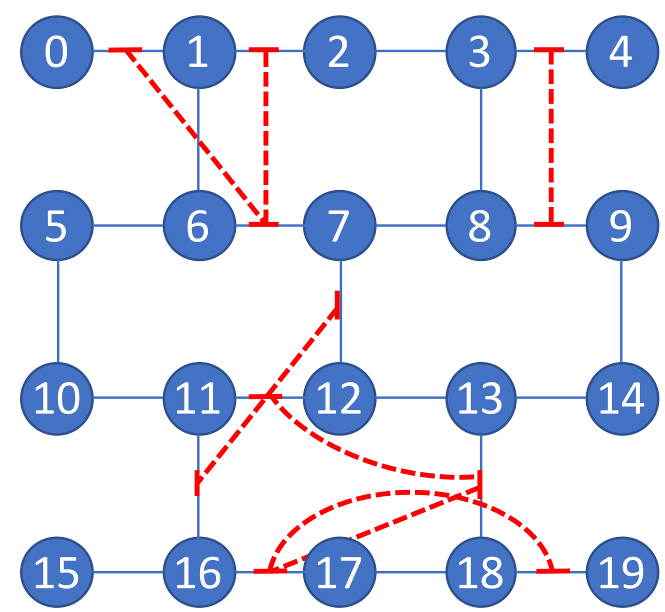
Competing Goals:

(1) Fully leverage qubits & Shorten the total execution
=> Multi-Programming

(2) High Reliability => Use the best qubits
=> Sequentially Allocate Programs

Solution: A run-time trade-off between these competing goals.

Cross-talk:



Cross-Talk: **Red Pairs** of gates when executed simultaneously will cause much larger errors.

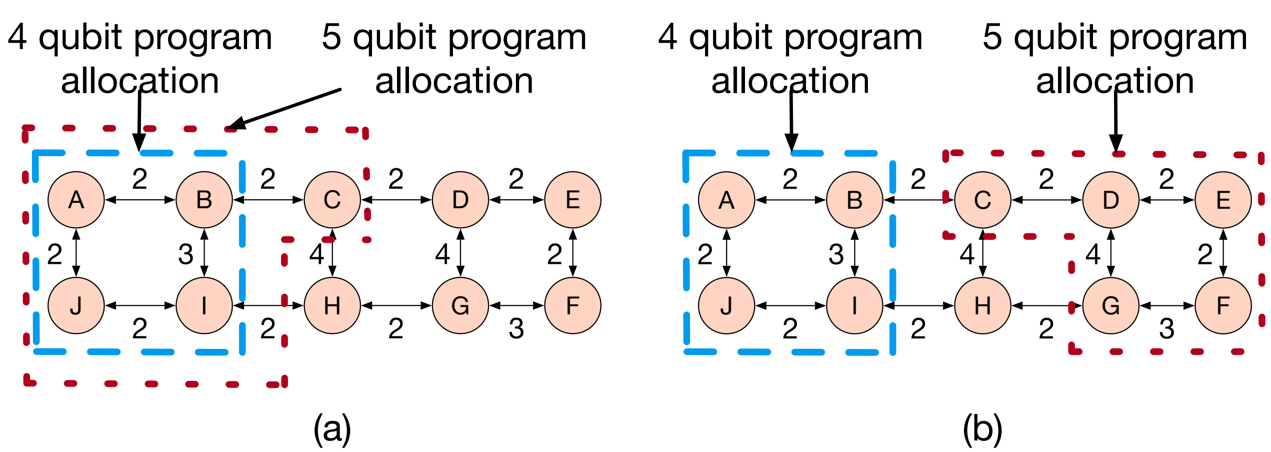
IBMQ Boeblingen

Features of NISQ Application Design

NISQ machines: very *restricted* hardware resources, where precisely controllable qubits are *expensive, error-prone, and scarce*.

NISQ application design: investigate the best balance of **trade-offs** among a large number of (potentially heterogeneous) factors specific to the **targeted application** and **quantum hardware.**

Multi-Programming (MICRO 2019) :



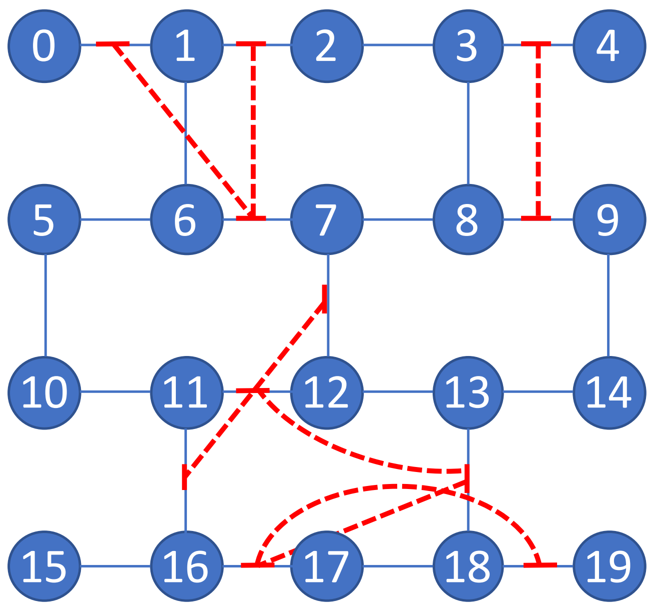
Competing Goals:

(1) Fully leverage qubits & Shorten the total execution => Multi-Programming

(2) High Reliability => Use the best qubits => Sequentially Allocate Programs

Solution: A run-time trade-off between these competing goals.

Cross-talk:



Cross-Talk: **Red Pairs** of gates when executed simultaneously will cause much larger errors.

IBMQ Boeblingen

Competing Goals:

Circuit Depth (decoherence) vs Cross-Talk

Software Solutions:

(1) Circuit Reschedule - Xtalk - (ASPLOS 2020)

(2) Frequency-Aware Compilation (MICRO 2020)

Automating NISQ Application Design

Current implementation of NISQ application design are CASE by CASE.

A **unified** and **automatic** framework for productivity?

Automating NISQ Application Design

Current implementation of NISQ application design are CASE by CASE.

A unified and automatic framework for productivity?

Desiderata:

Succinct Expression
of different design choices

Flexible Expression
of different optimization goals

Automation of Trade-offs
of competing optimization goals

High Reusability & Productivity
of balancing different trade-offs

Automating NISQ Application Design

Current implementation of NISQ application design are CASE by CASE.

A **unified** and **automatic** framework for productivity?

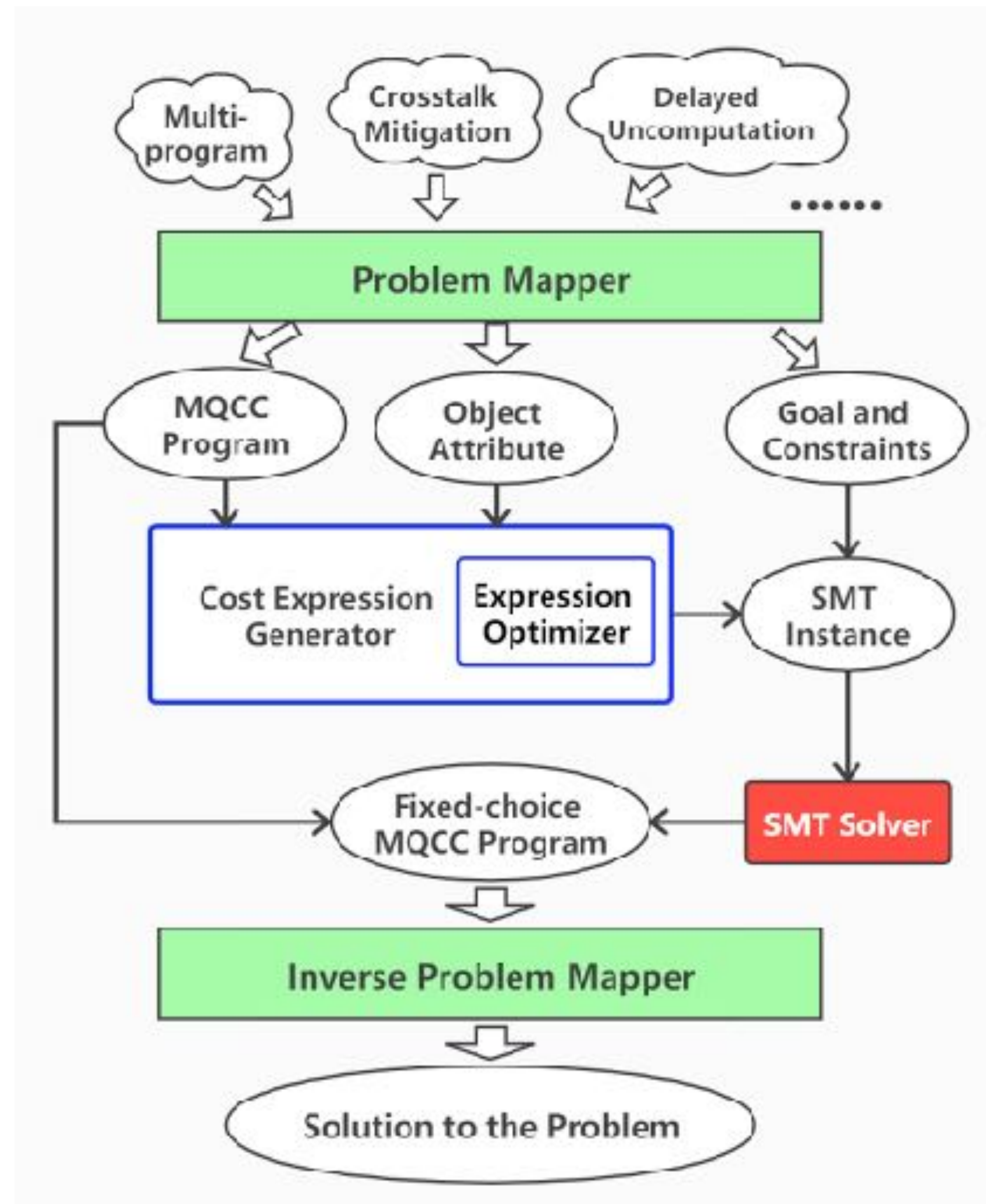
Desiderata:

Succinct Expression
of different design choices

Flexible Expression
of different optimization goals

Automation of Trade-offs
of competing optimization goals

High Reusability & Productivity
of balancing different trade-offs



Meta Quantum Circuits with Constraints (MQCC)

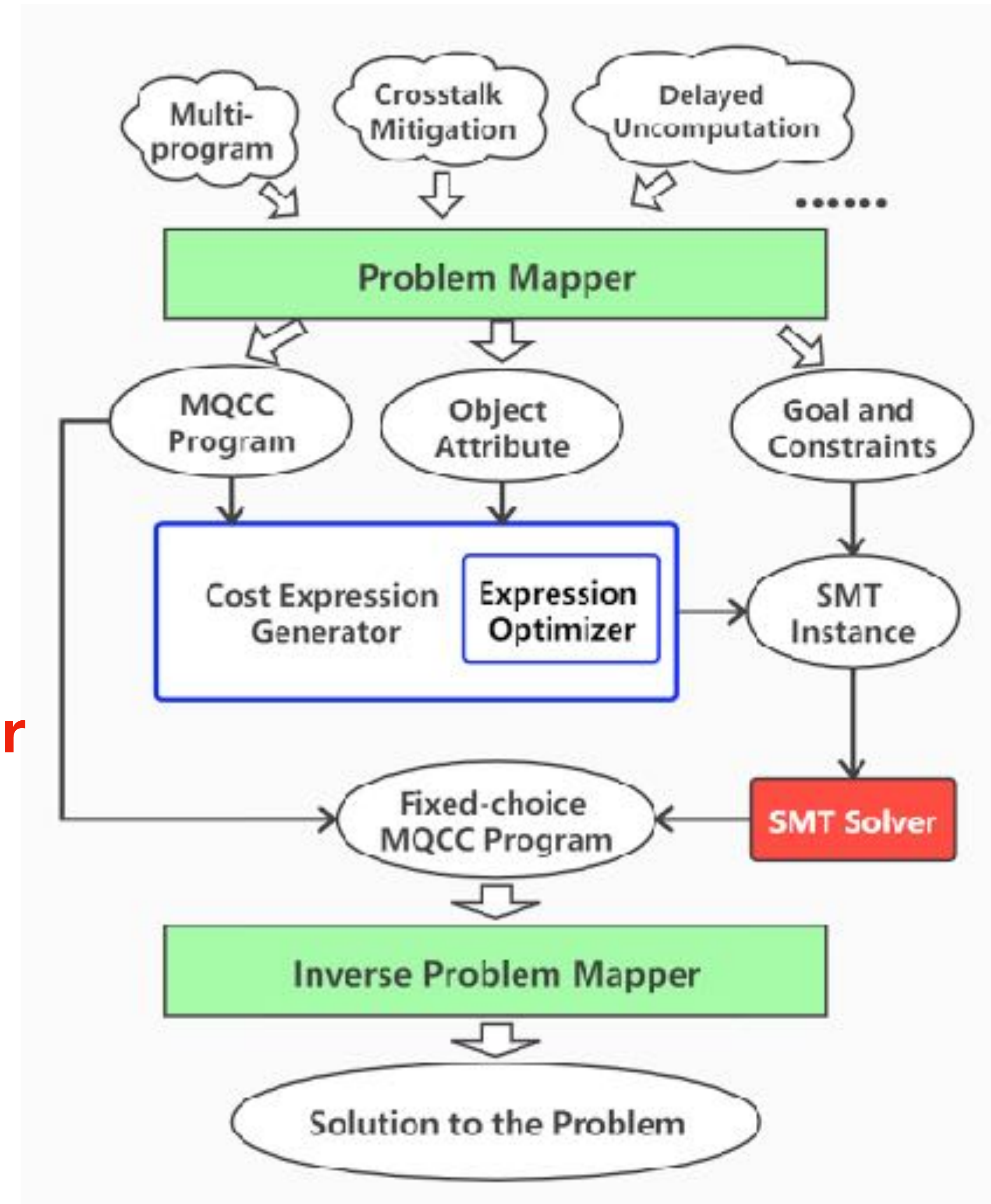
Desiderata:

Succinct Expression
of different design choices
MQCC with choice variables

Flexible Expression
of different optimization goals
Flexible Attributes Expression

Automation of Trade-offs
of competing optimization goals
Satisfiability Modulo Theories (SMT) Solver

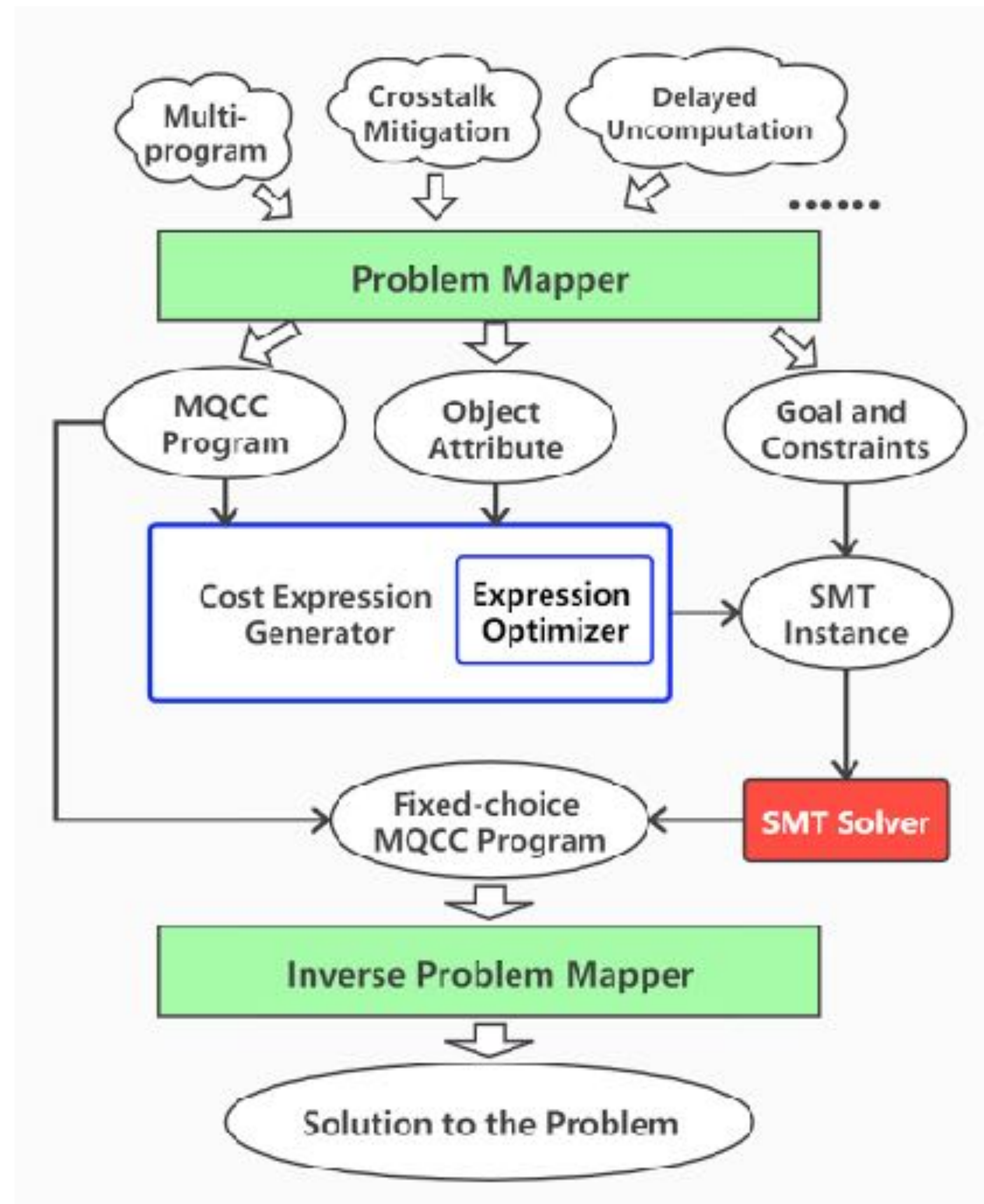
High Reusability & Productivity
of balancing different trade-offs
A Meta-Programming Framework



Meta Quantum Circuits with Constraints (MQCC)

```
\\Register and variable declarations
qreg q[10];
creg r[1];
fcho c1 = {0, 1};
fcho c2 = [0, 1];
\\lcho c = 1 - c1 * c2;

\\Module define
module Bell1(q1,q2){
    h(q1);
    cnot(q1, q2);
}

module Bell2(q1, q2){
    case (r[0]){
        1:  x(q1);
        0:  pass
    };
    h(q1);
    cnot(q1,q2);
}

\\Main part of the program
choice (c1){
    0:  Bell1(q[1], q[2]);
    1:  Bell1(q[7], q[8]);
};

h(q[0]);
measure(q[0],r[0]);
choice (c2){
    0:  Bell2(q[1], q[2]);
    default:  Bell2(q[7], q[8]);
};
```

A Sample Code of MQCC which shares many features with OpenQASM



Meta Quantum Circuits with Constraints (MQCC)

```
\\Register and variable declarations
qreg q[10];
creg r[1];
fcho c1 = {0, 1};
fcho c2 = [0, 1];
\\lcho c = 1 - c1 * c2;

\\Module define
module Bell1(q1,q2){
    h(q1);
    cnot(q1, q2);
}

module Bell2(q1, q2){
    case (r[0]){
        1:  x(q1);
        0:  pass
    };
    h(q1);
    cnot(q1,q2);
}

\\Main part of the program
choice (c1){
    0:  Bell1(q[1], q[2]);
    1:  Bell1(q[7], q[8]);
};

h(q[0]);
measure(q[0],r[0]);
choice (c2){
    0:  Bell2(q[1], q[2]);
    default:  Bell2(q[7], q[8]);
};
```

A Sample Code of MQCC which shares many features with OpenQASM

Define CHOICE variables

Free Choice (fcho) c1, c2 $\in \mathbb{Z}$, in certain ranges

Limited Choice (lcho) c=1-c1*c2 $\in \mathbb{Z}$

Meta Quantum Circuits with Constraints (MQCC)

```
\\Register and variable declarations
qreg q[10];
creg r[1];
fcho c1 = {0, 1};
fcho c2 = [0, 1];
\\lcho c = 1 - c1 * c2;

\\Module define
module Bell1(q1,q2){
    h(q1);
    cnot(q1, q2);
}

module Bell2(q1, q2){
    case (r[0]){
        1:  x(q1);
        0:  pass
    };
    h(q1);
    cnot(q1,q2);
}

\\Main part of the program
choice (c1){
    0:  Bell1(q[1], q[2]);
    1:  Bell1(q[7], q[8]);
};

h(q[0]);
measure(q[0],r[0]);
choice (c2){
    0:  Bell2(q[1], q[2]);
    default:  Bell2(q[7], q[8]);
};
```

A Sample Code of MQCC which shares many features with OpenQASM

Define CHOICE variables

Free Choice (fcho) c1, c2 $\in \mathbb{Z}$, in certain ranges

Limited Choice (lcho) c=1-c1*c2 $\in \mathbb{Z}$

Stitch Many Programs w/ choice variables

choice (c.v) $\{i : P_i\}$

Meta Quantum Circuits with Constraints (MQCC)

```
\\Register and variable declarations
qreg q[10];
creg r[1];
fcho c1 = {0, 1};
fcho c2 = [0, 1];
\\lcho c = 1 - c1 * c2;

\\Module define
module Bell1(q1,q2){
    h(q1);
    cnot(q1, q2);
}

module Bell2(q1, q2){
    case (r[0]){
        1:  x(q1);
        0:  pass
    };
    h(q1);
    cnot(q1,q2);
}

\\Main part of the program
choice (c1){
    0:  Bell1(q[1], q[2]);
    1:  Bell1(q[7], q[8]);
};

h(q[0]);
measure(q[0],r[0]);
choice (c2){
    0:  Bell2(q[1], q[2]);
    default:  Bell2(q[7], q[8]);
};
```

A Sample Code of MQCC which shares many features with OpenQASM

Define CHOICE variables

Free Choice (fcho) c1, c2 $\in \mathbb{Z}$, in certain ranges

Limited Choice (lcho) c=1-c1*c2 $\in \mathbb{Z}$

Stitch Many Programs w/ choice variables

choice (c.v) $\{i : P_i\}$

$n \in \mathbb{N} \quad i \in \mathbb{Z} \quad r \in \mathbb{R} \quad var \in Vars$

$qreg \in Quantum\ reg. \quad creg \in Classical\ reg.$

$$
\begin{aligned}
reg &::= qreg \mid creg \\
P \in Program &::= \overrightarrow{D}\ S \\
D \in Declaration &::= RegDecl \mid VarDecl \\
RegDecl &::= \textbf{qreg}\ qreg; \mid \textbf{creg}\ creg; \\
VarDecl &::= Free \mid Limit \\
Free &::= \textbf{fcho}\ var = \{\overrightarrow{i}\}; \mid \textbf{fcho}\ var = [i_1, i_2]; \\
Limit &::= \textbf{lcho}\ var = E; \\
E \in VarExp &::= i \mid var \mid E + E \mid E - E \\
&\quad \mid E * E \mid E/E \mid (E) \\
S \in Stmt &::= \epsilon \mid O \mid case \mid choice \mid S; S \\
O \in Operation &::= x(\overrightarrow{r}, \overrightarrow{reg}) \\
case &::= \textbf{case}(creg)\{\overline{i:\ S_i}\} \\
choice &::= \textbf{choice}(var)\{\overline{i:\ S_i}\}
\end{aligned}
$$

Expressing the Constraints on Costs/Attributes

Express **desired goals** as objects called **Attributes**. Thus, any MQCC program is a **transformer** on attributes.

Expressing the Constraints on Costs/Attributes

Express **desired goals** as objects called **Attributes**. Thus, any MQCC program is a **transformer** on attributes.

Precisely, any **attribute A** is defined by a tuple (*T, empty, op, case, value*) s.t.:

- T is a data type of the states. A state of type T consists of information needed in the computation of the cost.
- `empty` : T is the initial state at the beginning of the program.
- `op` : $T \times \text{string} \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg} \to T$ receives a state, an operation's name and its arguments, and generates a new state that merges the old state and the information of the operation.
- `case` : $T \times \overrightarrow{T} \to T$ receives an old state, a list of states corresponding to each case branch which has merged the corresponding sub-programs' information on the old state, and generates a new state merging the old state and the sub-programs' states.
- `value` : $T \to \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

Expressing the Constraints on Costs/Attributes

Express **desired goals** as objects called **Attributes**. Thus, any MQCC program is a **transformer** on attributes.

Precisely, any **attribute A** is defined by a tuple (*T, empty, op, case, value*) s.t.:

- T is a data type of the states. A state of type T consists of information needed in the computation of the cost.
- `empty` : T is the initial state at the beginning of the program.
- `op` : $T \times \text{string} \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg} \to T$ receives a state, an operation's name and its arguments, and generates a new state that merges the old state and the information of the operation.
- `case` : $T \times \overrightarrow{T} \to T$ receives an old state, a list of states corresponding to each case branch which has merged the corresponding sub-programs' information on the old state, and generates a new state merging the old state and the sub-programs' states.
- `value` : $T \to \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

choice vars

program S' attribute semantics $[\![S]\!]: \ (Vars \to \mathbb{Z}) \times T \to T$

transformer on T

Expressing the Constraints on Costs/Attributes

Express **desired goals** as objects called **Attributes**. Thus, any MQCC program is a **transformer** on attributes.

Precisely, any **attribute A** is defined by a tuple (*T, empty, op, case, value*) s.t.:

- T is a data type of the states. A state of type T consists of information needed in the computation of the cost.
- $\texttt{empty} : T$ is the initial state at the beginning of the program.
- $\texttt{op} : T \times \text{string} \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg} \to T$ receives a state, an operation's name and its arguments, and generates a new state that merges the old state and the information of the operation.
- $\texttt{case} : T \times \overrightarrow{T} \to T$ receives an old state, a list of states corresponding to each case branch which has merged the corresponding sub-programs' information on the old state, and generates a new state merging the old state and the sub-programs' states.
- $\texttt{value} : T \to \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

choice vars

program S' attribute semantics $\left[\left[S\right]\right] : \; (Vars \to \mathbb{Z}) \times T \to T$

transformer on T

$$\frac{S = opID(exps,\ regs)}{\llbracket S \rrbracket\,(\sigma, s) = \texttt{op}(s, opID,\ exps,\ regs)}$$

$$\frac{}{\llbracket S_1; S_2 \rrbracket\,(\sigma, s) = \llbracket S_2 \rrbracket\,(\sigma, \llbracket S_1 \rrbracket\,(\sigma, s))}$$

how transformers evolve over programs

$$\frac{S = \textbf{case}(creg)\{\overline{i:\ S_i}\}}{\llbracket S \rrbracket\,(\sigma, s) = \texttt{case}(s, [\llbracket S_i \rrbracket\,(\sigma, s)]_i)}$$

$$\frac{S = \textbf{choice}(var)\{\overline{i:\ S_i}\} \qquad k = \sigma[var]}{\llbracket S \rrbracket\,(\sigma, s) = \llbracket S_k \rrbracket\,(\sigma, s)}$$

Expressing the Constraints on Costs/Attributes

Express **desired goals** as objects called **Attributes**. Thus, any MQCC program is a **transformer** on attributes.

Precisely, any **attribute A** is defined by a tuple (*T, empty, op, case, value*) s.t.:

- T is a data type of the states. A state of type T consists of information needed in the computation of the cost.
- `empty` : T is the initial state at the beginning of the program.
- `op` : $T \times \text{string} \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg} \rightarrow T$ receives a state, an operation's name and its arguments, and generates a new state that merges the old state and the information of the operation.
- `case` : $T \times \overrightarrow{T} \rightarrow T$ receives an old state, a list of states corresponding to each case branch which has merged the corresponding sub-programs' information on the old state, and generates a new state merging the old state and the sub-programs' states.
- `value` : $T \rightarrow \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

choice vars

program S' attribute semantics $\left[\left[S\right]\right] : \; (Vars \rightarrow \mathbb{Z}) \times T \rightarrow T$

transformer on T

$$\frac{S = opID(exps,\ regs)}{\llbracket S \rrbracket\,(\sigma, s) = \mathtt{op}(s, opID,\ exps,\ regs)}$$

$$\frac{}{\llbracket S_1; S_2 \rrbracket\,(\sigma, s) = \llbracket S_2 \rrbracket\,(\sigma, \llbracket S_1 \rrbracket\,(\sigma, s))}$$

$$\frac{S = \textbf{case}(creg)\{\overline{i:\ S_i}\}}{\llbracket S \rrbracket\,(\sigma, s) = \mathtt{case}(s, [\llbracket S_i \rrbracket\,(\sigma, s)]_i)}$$

$$\frac{S = \textbf{choice}(var)\{\overline{i:\ S_i}\} \qquad k = \sigma[var]}{\llbracket S \rrbracket\,(\sigma, s) = \llbracket S_k \rrbracket\,(\sigma, s)}$$

how transformers evolve over programs

Express the constraints on the final T as SMT instances (details omitted)

Expressing the Constraints on Costs/Attributes

Express **desired goals** as objects called **Attributes**. Thus, any MQCC program is a **transformer** on attributes.

Precisely, any **attribute A** is defined by a tuple (*T, empty, op, case, value*) s.t.:

- T is a data type of the states. A state of type T consists of information needed in the computation of the cost.
- `empty` : T is the initial state at the beginning of the program.
- `op` : $T \times \text{string} \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg} \rightarrow T$ receives a state, an operation's name and its arguments, and generates a new state that merges the old state and the information of the operation.
- `case` : $T \times \overrightarrow{T} \rightarrow T$ receives an old state, a list of states corresponding to each case branch which has merged the corresponding sub-programs' information on the old state, and generates a new state merging the old state and the sub-programs' states.
- `value` : $T \rightarrow \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

program S' attribute semantics $[\![S]\!]: \ (Vars \rightarrow \mathbb{Z}) \times T \rightarrow T$

choice vars — *transformer on T*

$$\frac{S = opID(exps,\ regs)}{[\![S]\!]\,(\sigma, s) = \texttt{op}(s, opID,\ exps,\ regs)}$$

$$\frac{}{[\![S_1; S_2]\!]\,(\sigma, s) = [\![S_2]\!]\,(\sigma, [\![S_1]\!]\,(\sigma, s))}$$

$$\frac{S = \textbf{case}(creg)\{\overline{i:\ S_i}\}}{[\![S]\!]\,(\sigma, s) = \texttt{case}(s, [[\![S_i]\!]\,(\sigma, s)]_i)}$$

$$\frac{S = \textbf{choice}(var)\{\overline{i:\ S_i}\} \qquad k = \sigma[var]}{[\![S]\!]\,(\sigma, s) = [\![S_k]\!]\,(\sigma, s)}$$

how transformers evolve over programs

Express the constraints on the final T as SMT instances (details omitted)

Examples:

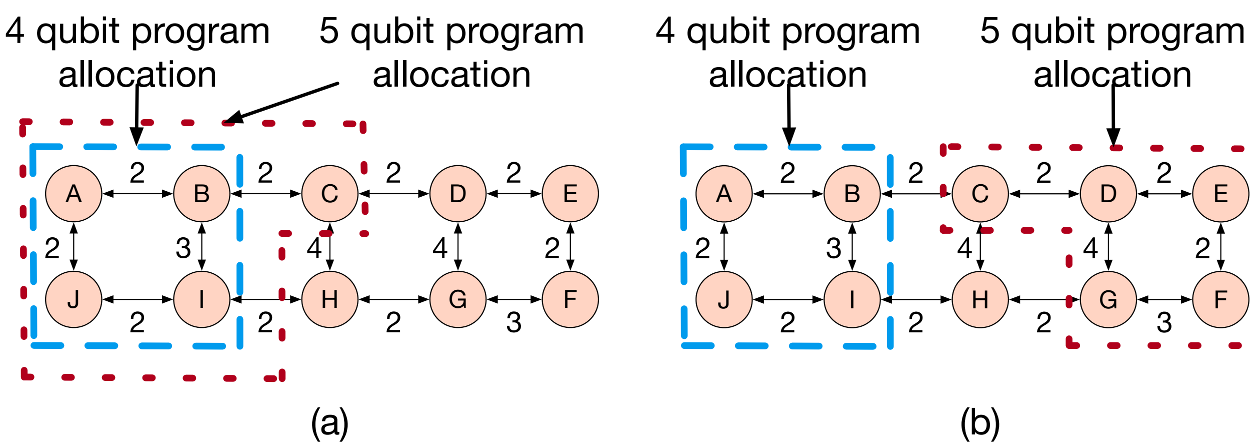
```
Attribute Noise:
  T:                noise : ℝ
  empty():=         init s : T, s.noise = 0
                    return s
  value(s : T):=    return s.noise
  op (s : T, OpID : str, exps : ℝ⃗, regs : Reg⃗):=
                s.noise += calNoise(OpId, exps, regs)
                    return s
  case (s : T, group : Vector of T):=
                    s.noise = max {n.noise| n ∈group}
                    return s
```

```
Attribute Depth:
  T:             dep : Map of Reg → ℕ
  empty():=      init s : T, s.dep = ∅
                 return s
  value(s : T):=  return (max s.dep.values)
  op (s : T, OpID : str, exps : ℝ⃗, regs : Reg⃗):=
                 share = s.dep.keys ∩ regs
                 next = max {s.dep[i]| i ∈share} + 1
                 for i ∈regs: s.dep.update(i, next)
                 return s
  case (s : T, group : Vector of T):=
                 all = ⋃_{n∈group} n.dep.keys
    s.dep = {(k, max {n.dep[k] | n∈group}) | k∈all}
                 return s
```

Case Study

All experiments performed on IBMQ machines

Multi-Programming (MICRO 2019) :



(a) (b)

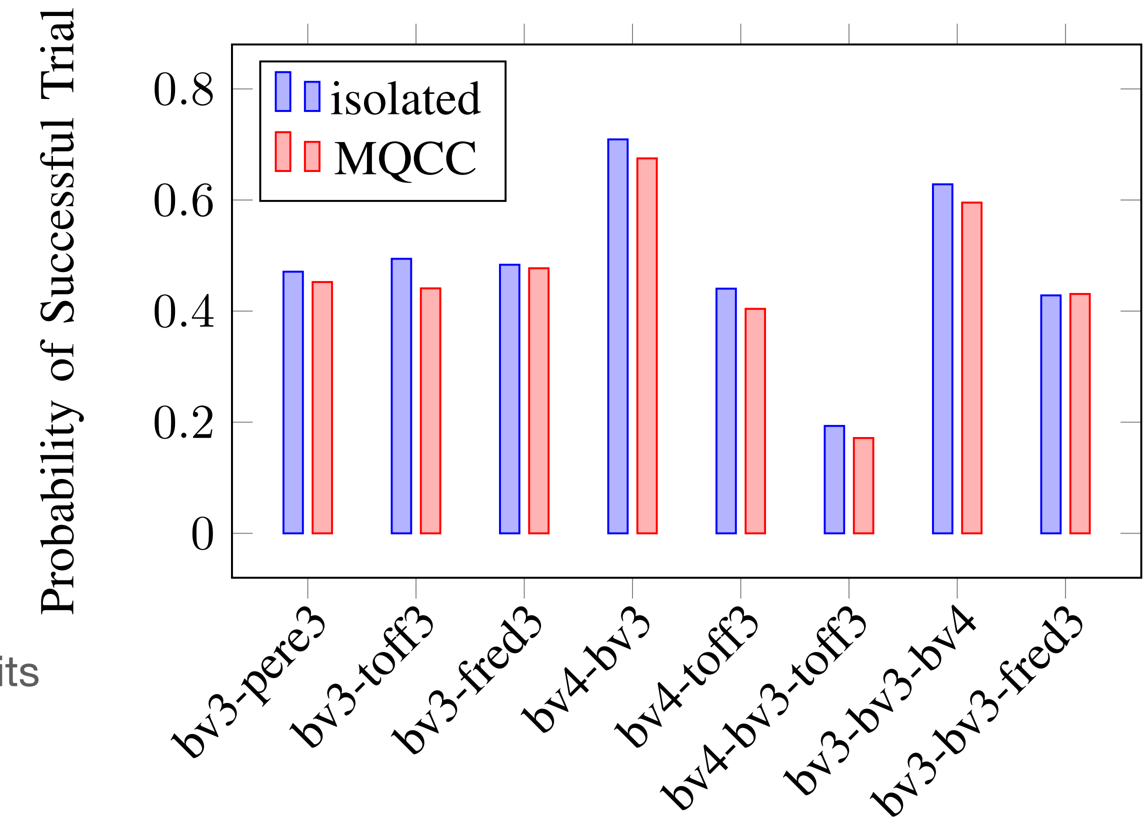
Competing Goals:
Depth vs High-quality Qubits

isolated

Sequential: always high-quality qubits

MQCC

Multi-Programming with MQCC

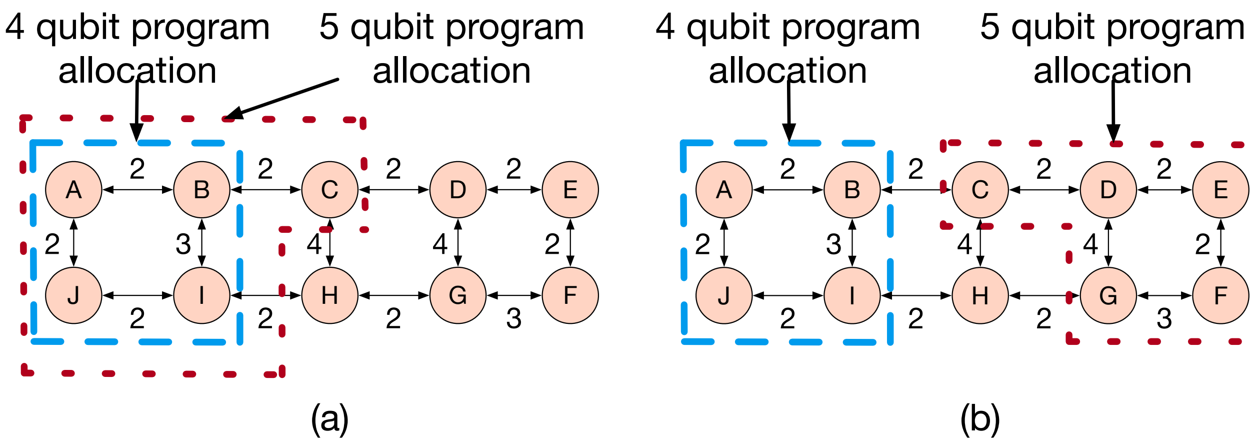


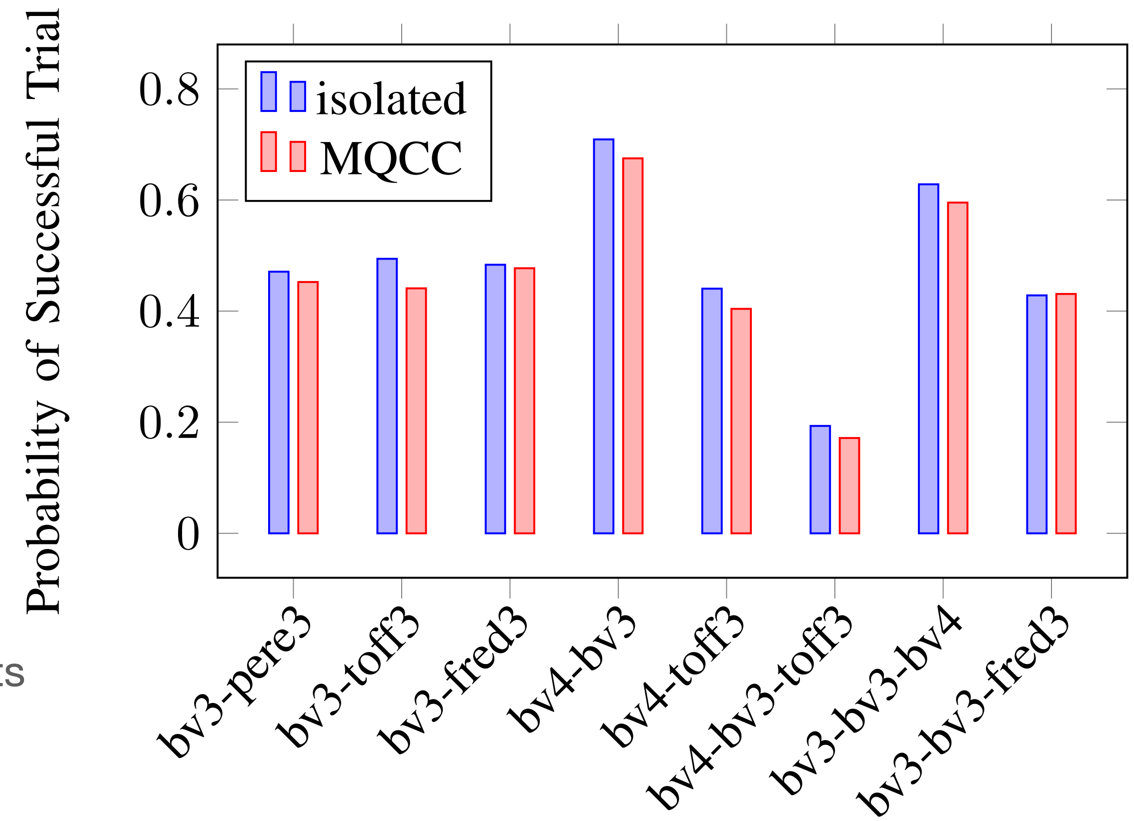
Multi-Tasks over Simple Quantum Algorithms

Case Study

All experiments performed on IBMQ machines

Multi-Programming (MICRO 2019) :





Competing Goals:
Depth vs High-quality Qubits

isolated

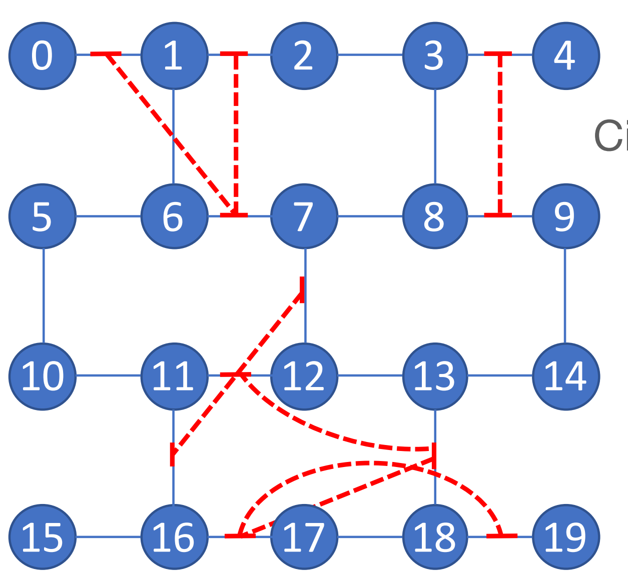
Sequential: always high-quality qubits

MQCC

Multi-Programming with MQCC

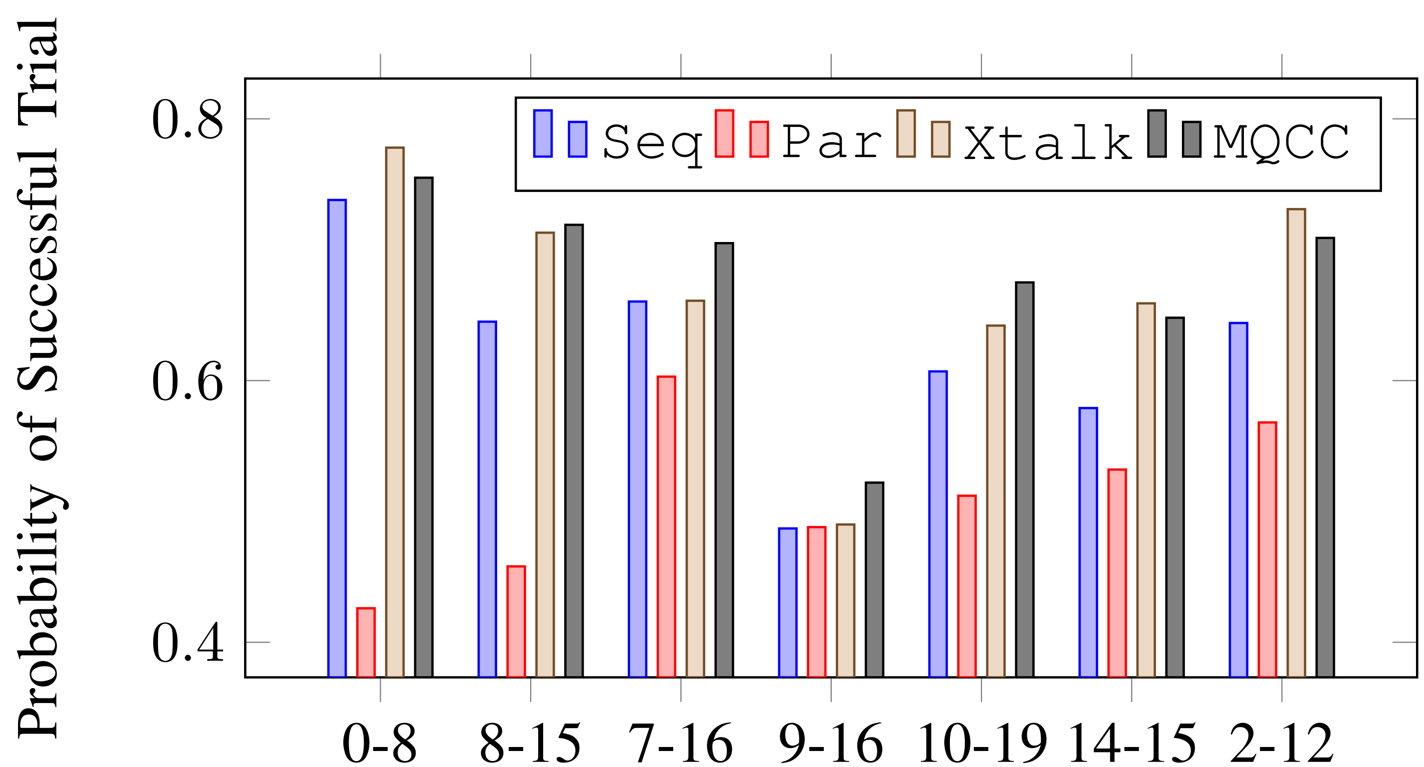
Multi-Tasks over Simple Quantum Algorithms

Cross-talk: (Xtalk - ASPLOS 2020)



Competing Goals:
Circuit Depth (decoherence) vs Cross-Talk

Benchmark over SWAP circuits connecting a-b on IBM Boeblingen



Case Study: Multi-Programming + Cross-Talk

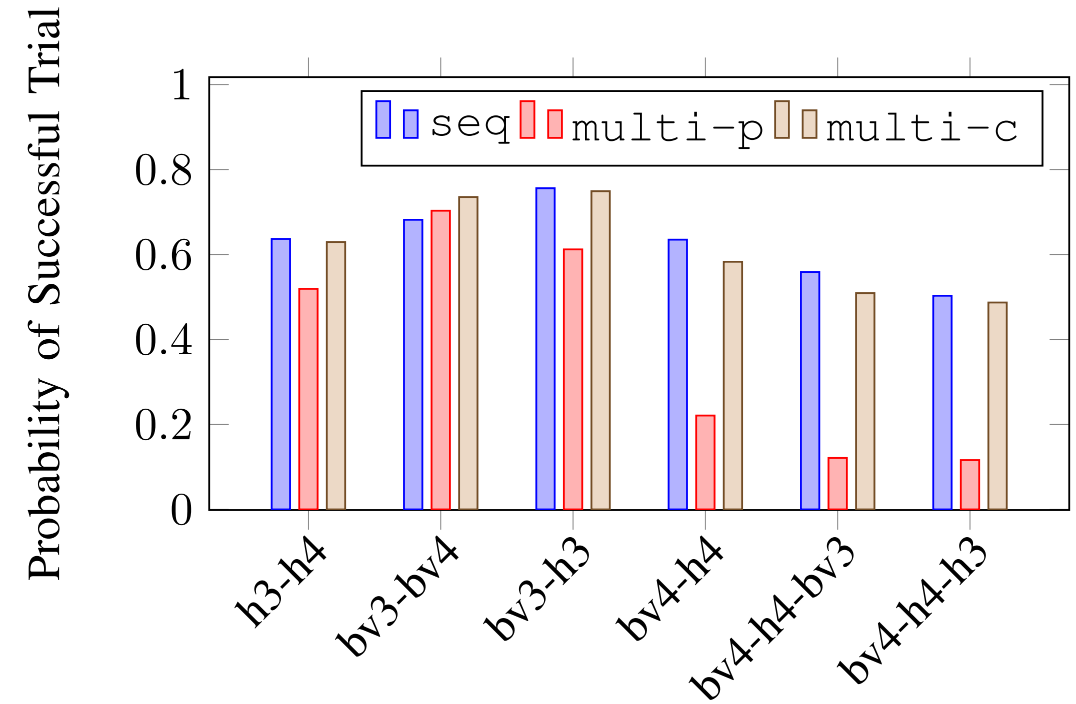
Optimizing Goal:
Noise + Decoherence + Crosstalk

EASY implementation in MQCC

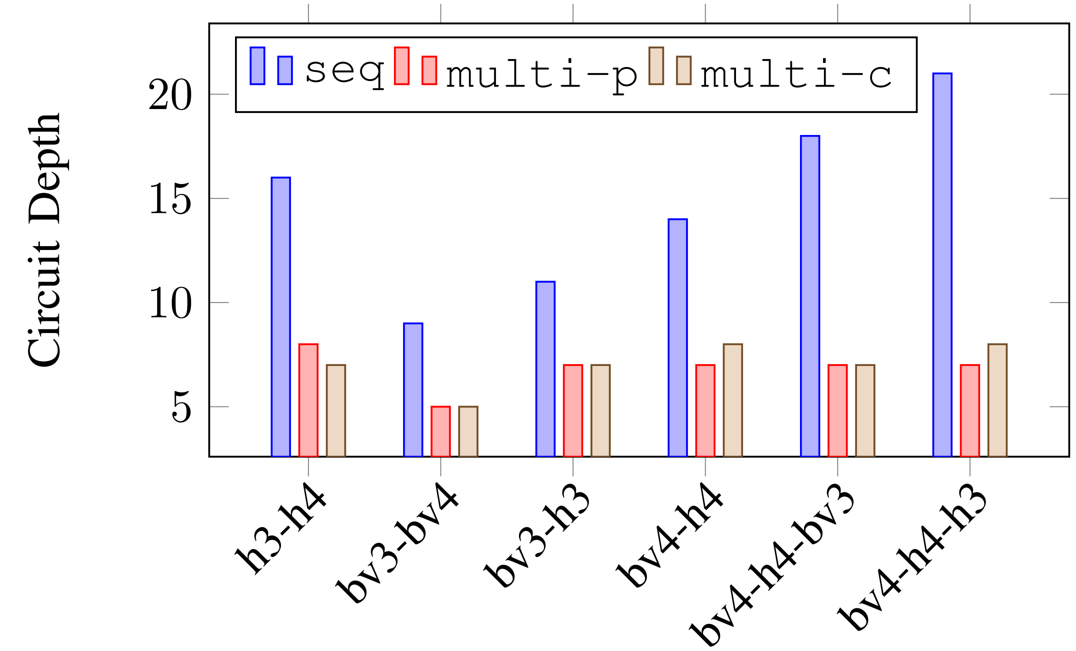
All experiments performed on IBMQ machines

- `seq` Sequential: always high-quality qubits. but larger depth (decoherence)
- `multi-p` Multi-programs without considering crosstalk short depth, but large crosstalk errors
- `multi-c` Multi-programs with crosstalk short depth and large successful probability

```
Attribute Crosstalk
  T:   dep : Map of Reg → ℕ
       rep : Map of ℕ → Set of (str × Reg⃗)
  empty ():=     init s:T, s.dep = ∅, s.rep = ∅
                 return s
  value (s : T):=   return calCross(rep)
  op (s : T, opID : str, exps : ℝ⃗, regs : Reg⃗):=
       if opId == "barrier" :
         cur = max s.dep.values
         for i∈regs: s.dep.update(i, cur)
       else:
         share = s.dep.keys ∩ regs
         next = max {s.dep[i] | i ∈ share} + 1
         s.rep[next].insert( (OpID, regs) )
         for i∈regs: s.dep.update(i, next)
       return s
  case (s : T, group : Vector of T):=
       all = ⋃ n.dep.keys, n∈group
       s.dep = {(k, max {n.dep[k]|n∈group})| k∈all}
  s.rep = {(k, ⋃_{n∈group} n.rep[k]) |∃u ∈ group, k∈ u}
       return s
```

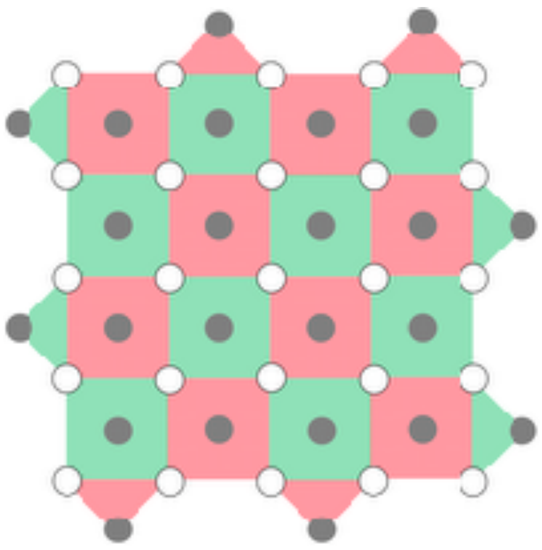


(a) Probability of Successful Trial. Here higher PST is better.

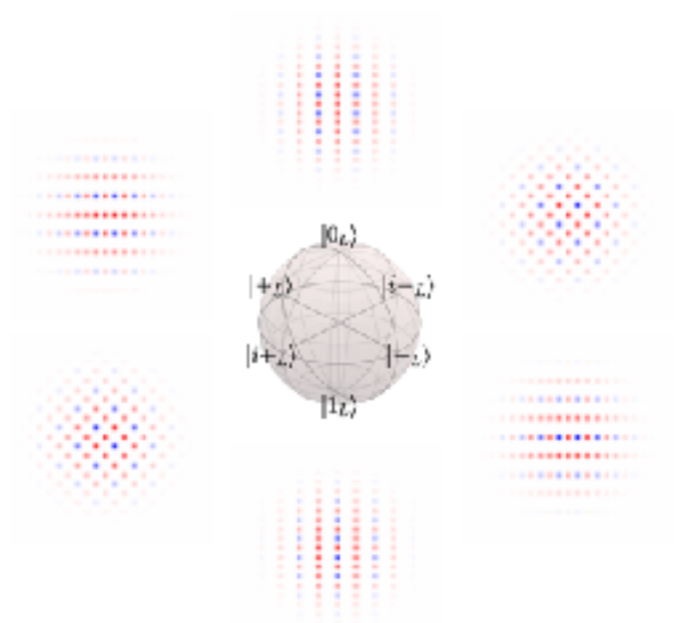


(b) Circuit Depth. Here lower circuit depth is better.

Nature

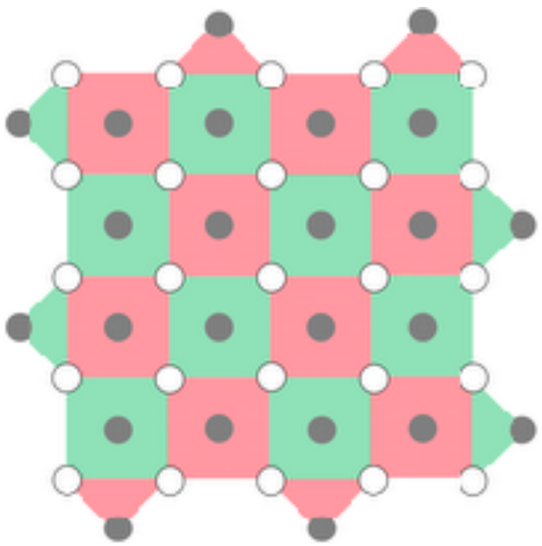


Quantum Error Correction
Fight
Quantum Decoherence

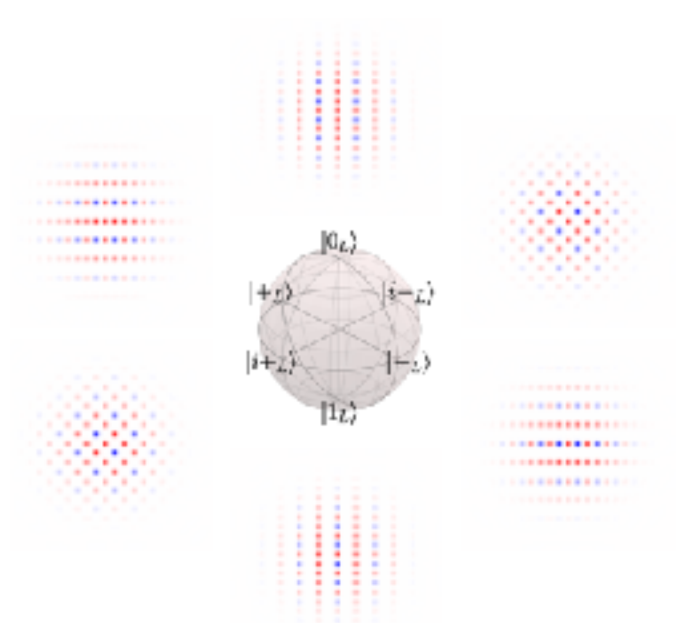


ERROR

Nature

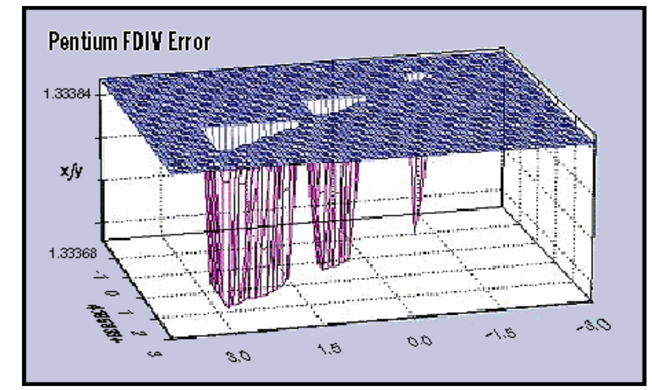


Quantum Error Correction
Fight
Quantum Decoherence



ERROR

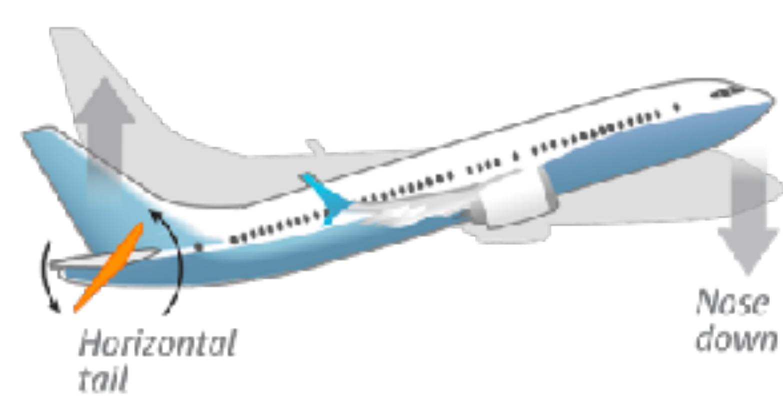
Human



Intel Pentium FPU error

Ariane 5

MCAS safety system engages



Human Errors in Quantum Software Engineering

Being careful cannot solve the human error problem in either classical or quantum.

Quantum case : Significantly More **CHALLENGING** than Classical

- standard software assurance techniques, e.g., black-box / unit test, expensive in q.
- quantum mechanics prohibits certain testing, e.g., assertions

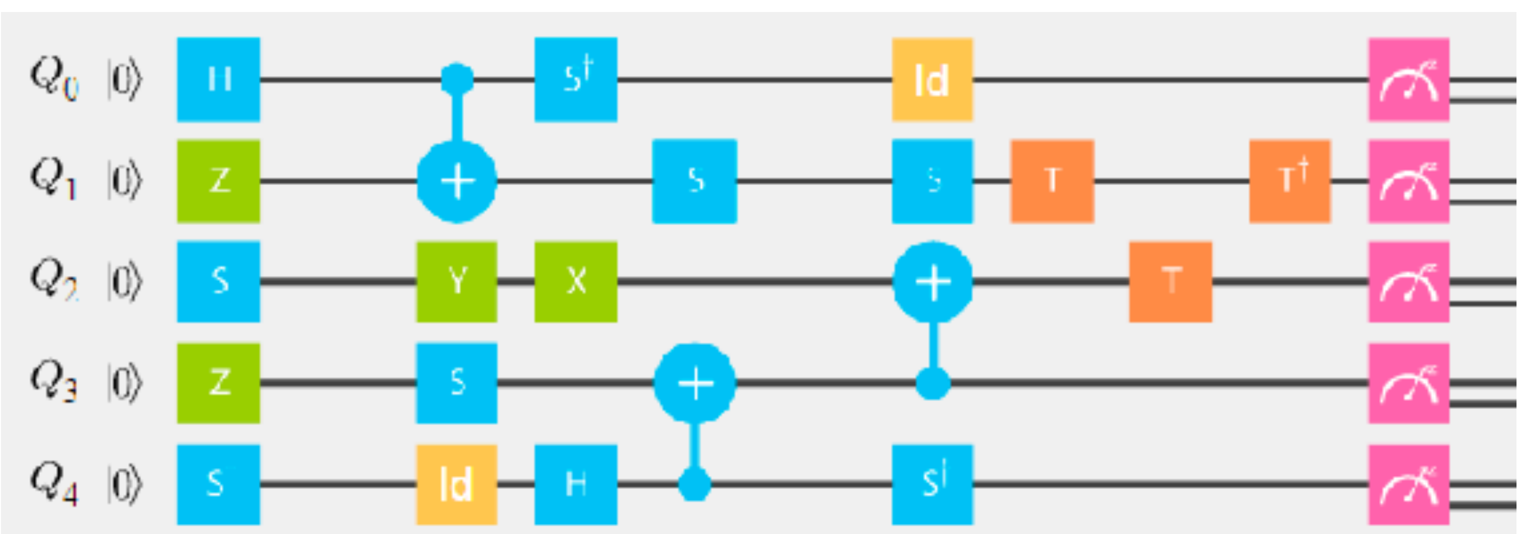
Human Errors in Quantum Software Engineering

Being careful cannot solve the human error problem in either classical or quantum.

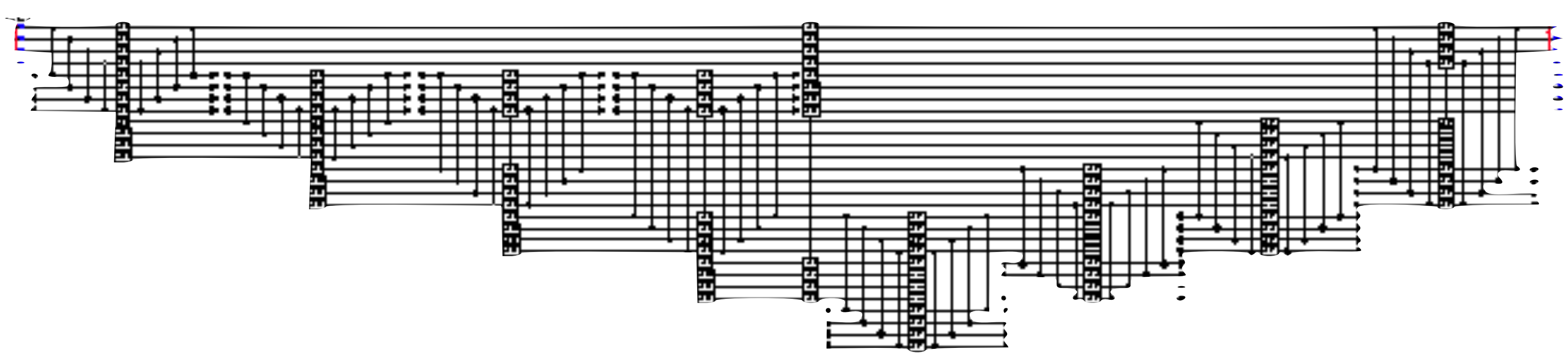
Quantum case : Significantly More **CHALLENGING** than Classical

- standard software assurance techniques, e.g., black-box / unit test, expensive in q.
- quantum mechanics prohibits certain testing, e.g., assertions

Reality: testing in quantum today



confirming the circuit by observation.... not scalable...



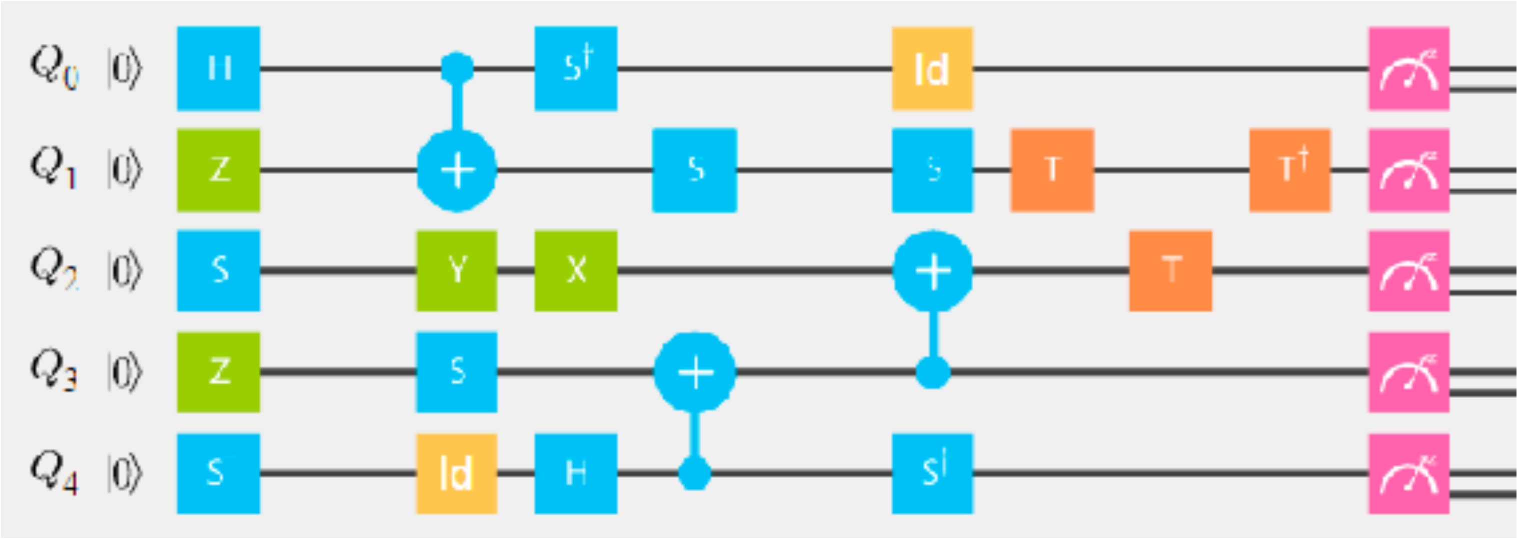
Human Errors in Quantum Software Engineering

Being careful cannot solve the human error problem in either classical or quantum.

Quantum case : Significantly More **CHALLENGING** than Classical

- standard software assurance techniques, e.g., black-box / unit test, expensive in q.
- quantum mechanics prohibits certain testing, e.g., assertions

Reality: testing in quantum today

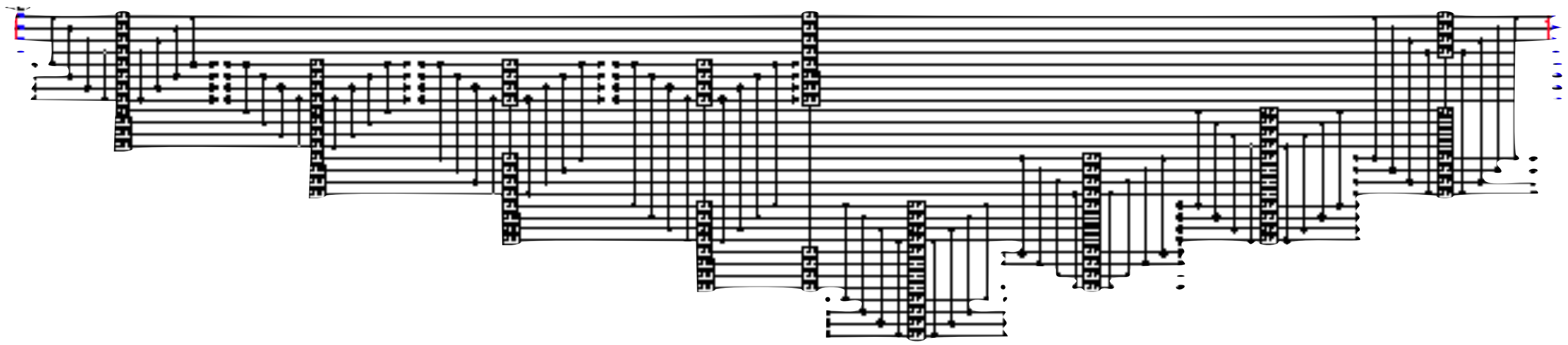


QISKIT Compiler **ERRORs**

Much **HARDER** to detect!

Serious Consequences!

confirming the circuit by observation…. not scalable…



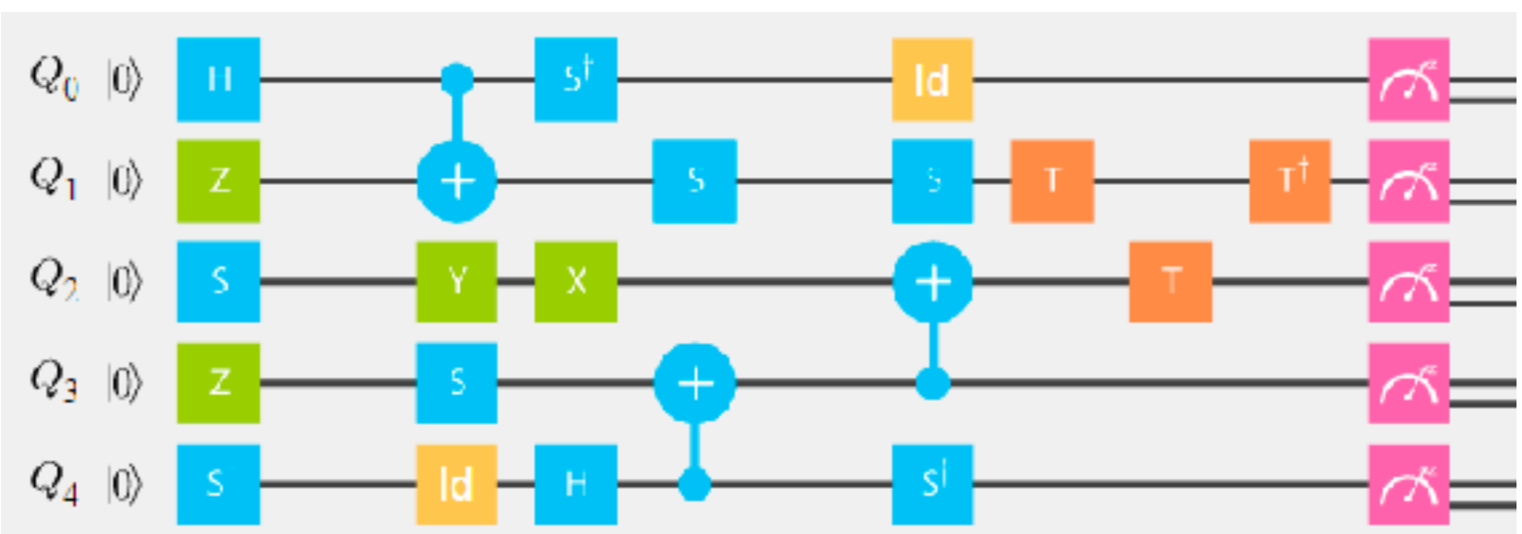
Human Errors in Quantum Software Engineering

Being careful cannot solve the human error problem in either classical or quantum.

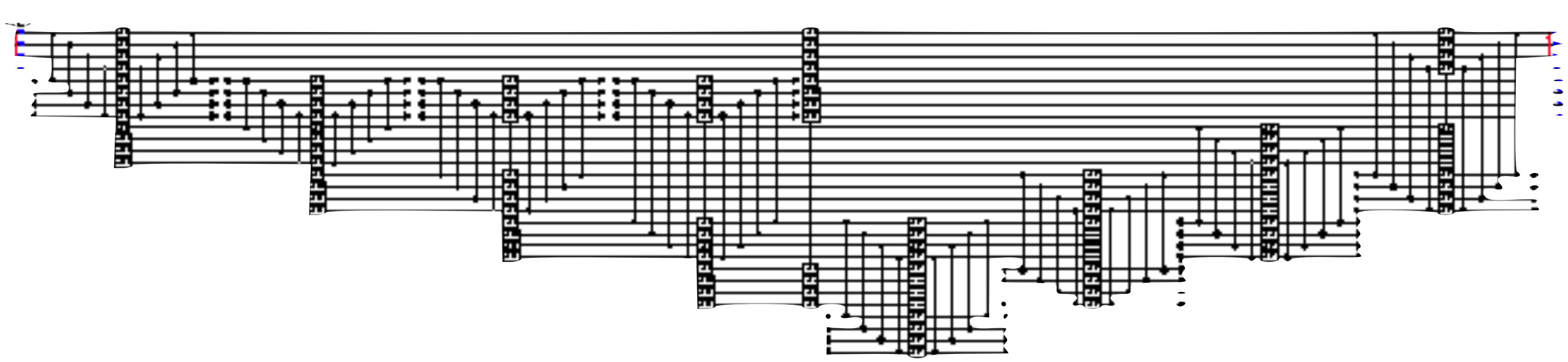
Quantum case : Significantly More **CHALLENGING** than Classical

- standard software assurance techniques, e.g., black-box / unit test, expensive in q.
- quantum mechanics prohibits certain testing, e.g., assertions

Reality: testing in quantum today



confirming the circuit by observation…. not scalable…



QISKIT Compiler **ERRORs**

Much **HARDER** to detect!

Serious Consequences!

Similar Concerns in classical !

More **SERIOUS** in quantum !

Certified software: a solution to validation of q. software

The Verifying Compiler: A Grand Challenge for Computing Research

TONY HOARE

Microsoft Research Ltd., Cambridge, UK

Journal of the ACM, Vol 50, 2003

Certified software: a solution to validation of q. software

The Verifying Compiler: A Grand Challenge for Computing Research

TONY HOARE

Microsoft Research Ltd., Cambridge, UK

Journal of the ACM, Vol 50, 2003

GCC : many bugs in software testing

CompCert: a certified "GCC", bug-free

Certified software: a solution to validation of q. software

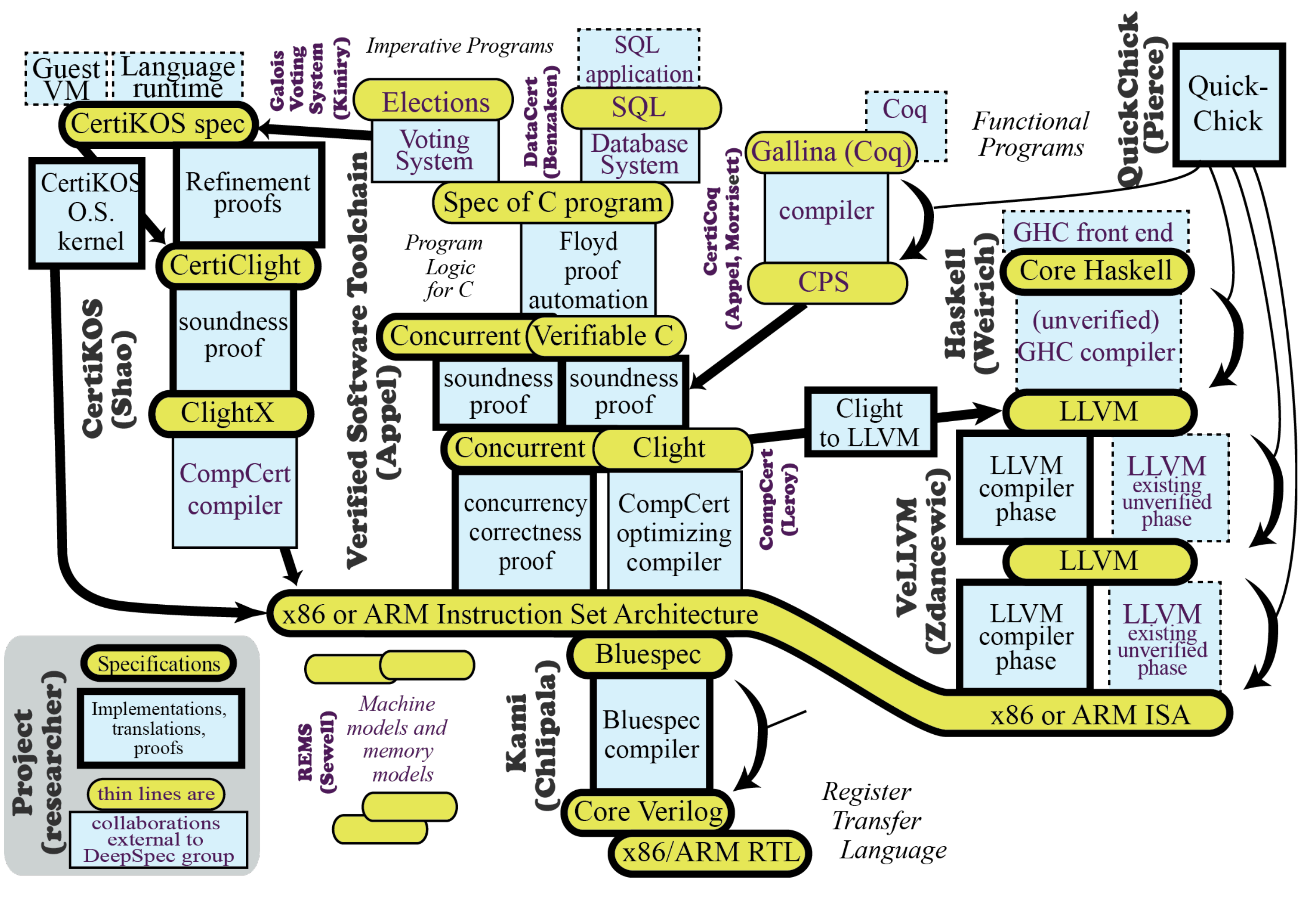
The Verifying Compiler: A Grand Challenge for Computing Research

TONY HOARE

Microsoft Research Ltd., Cambridge, UK

Journal of the ACM, Vol 50, 2003

GCC : many bugs in software testing
CompCert: a certified "GCC", bug-free



Verified Software Toolchain

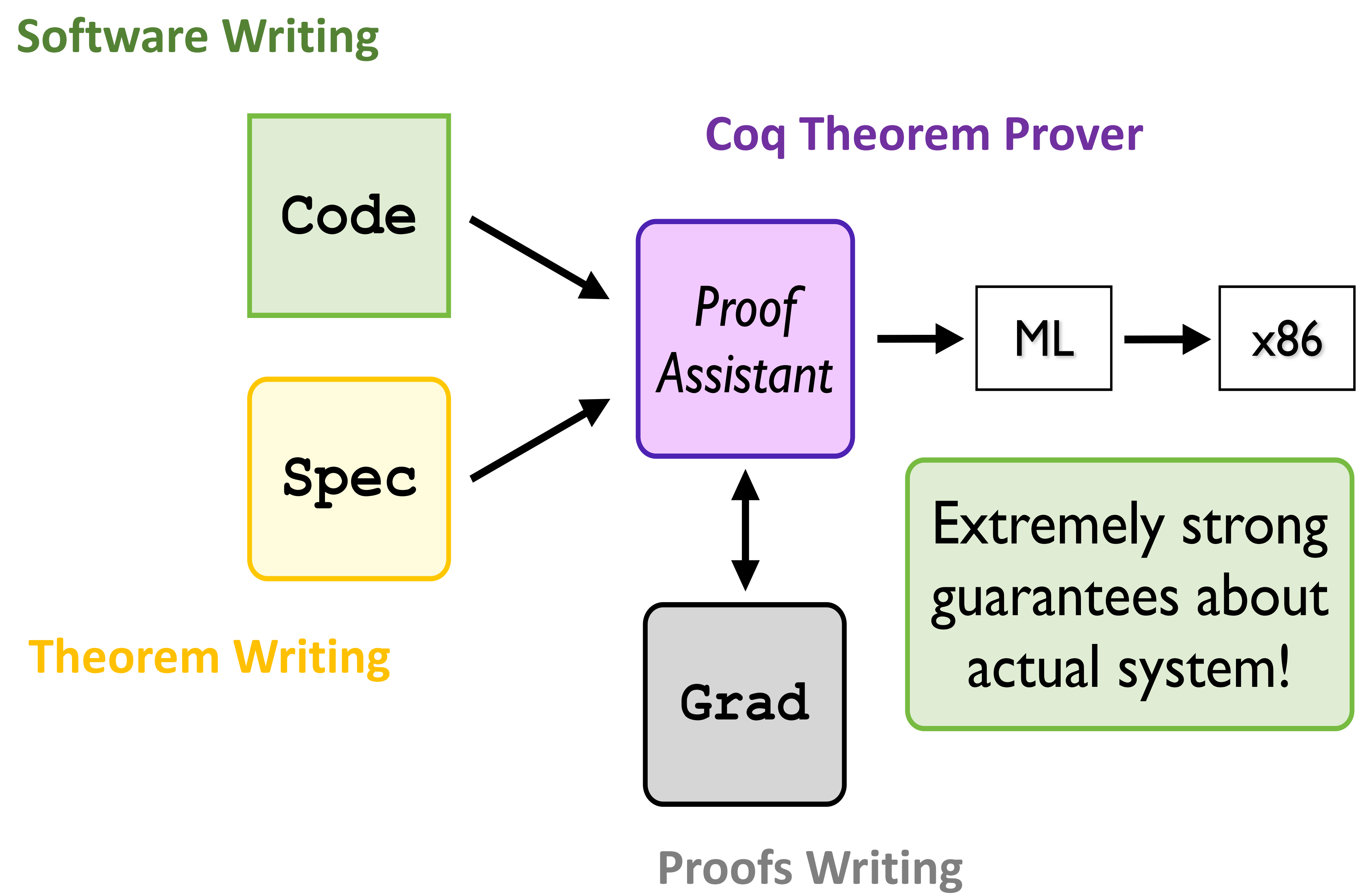
CERTIKOS

Vellvm verified LLVM

Certi Coq

Kami

Certified software: a solution to validation of q. software



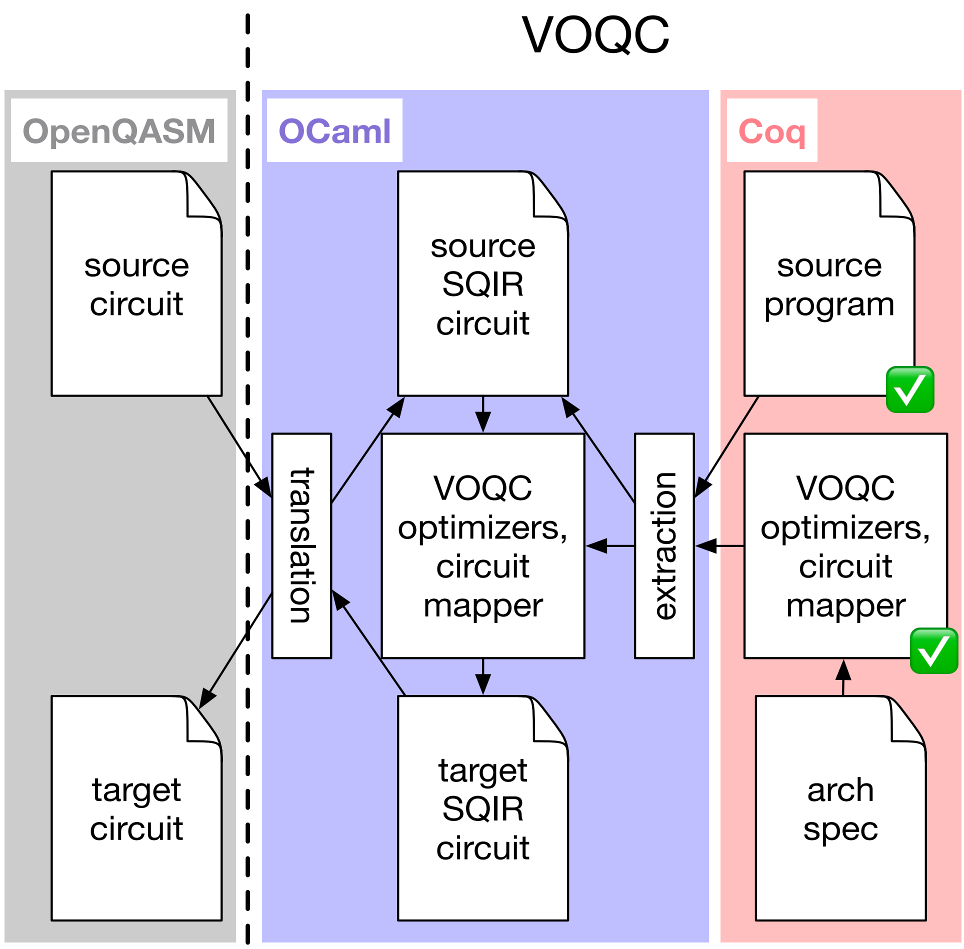
(1) Ensure correctness of code by construction.

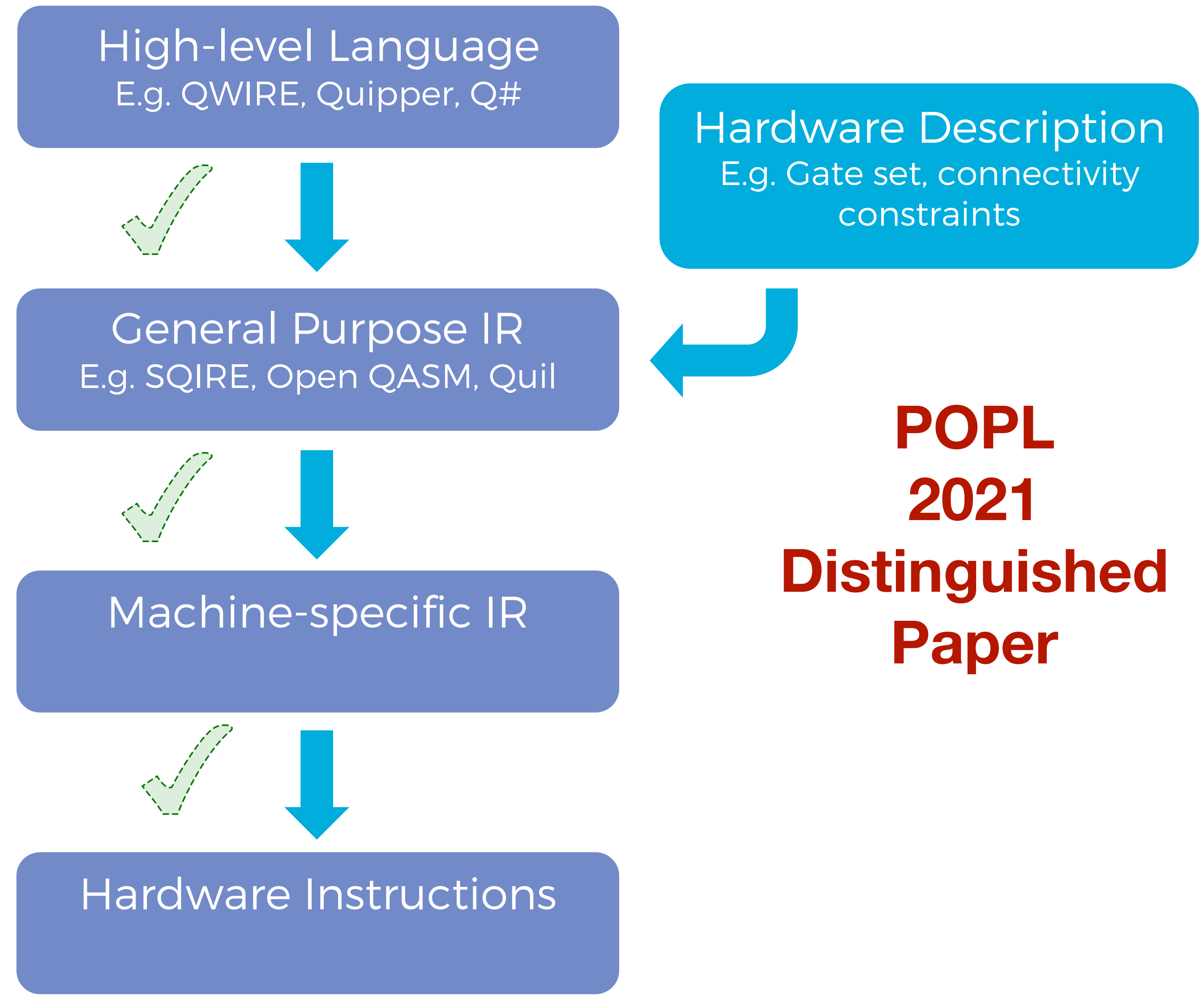
(2) Scalability for quantum based on symbolic proofs.

Joint Center for Quantum Information and Computer Science

PLUM

(Verified Optimizer for Quantum Circuits)





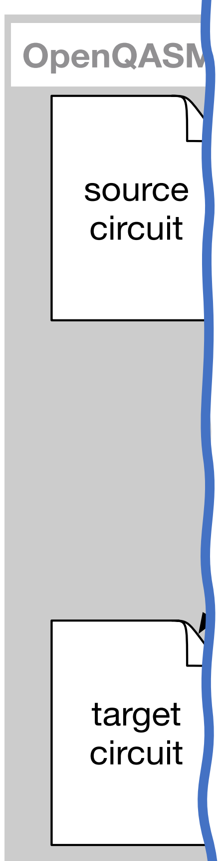
POPL 2021 Distinguished Paper

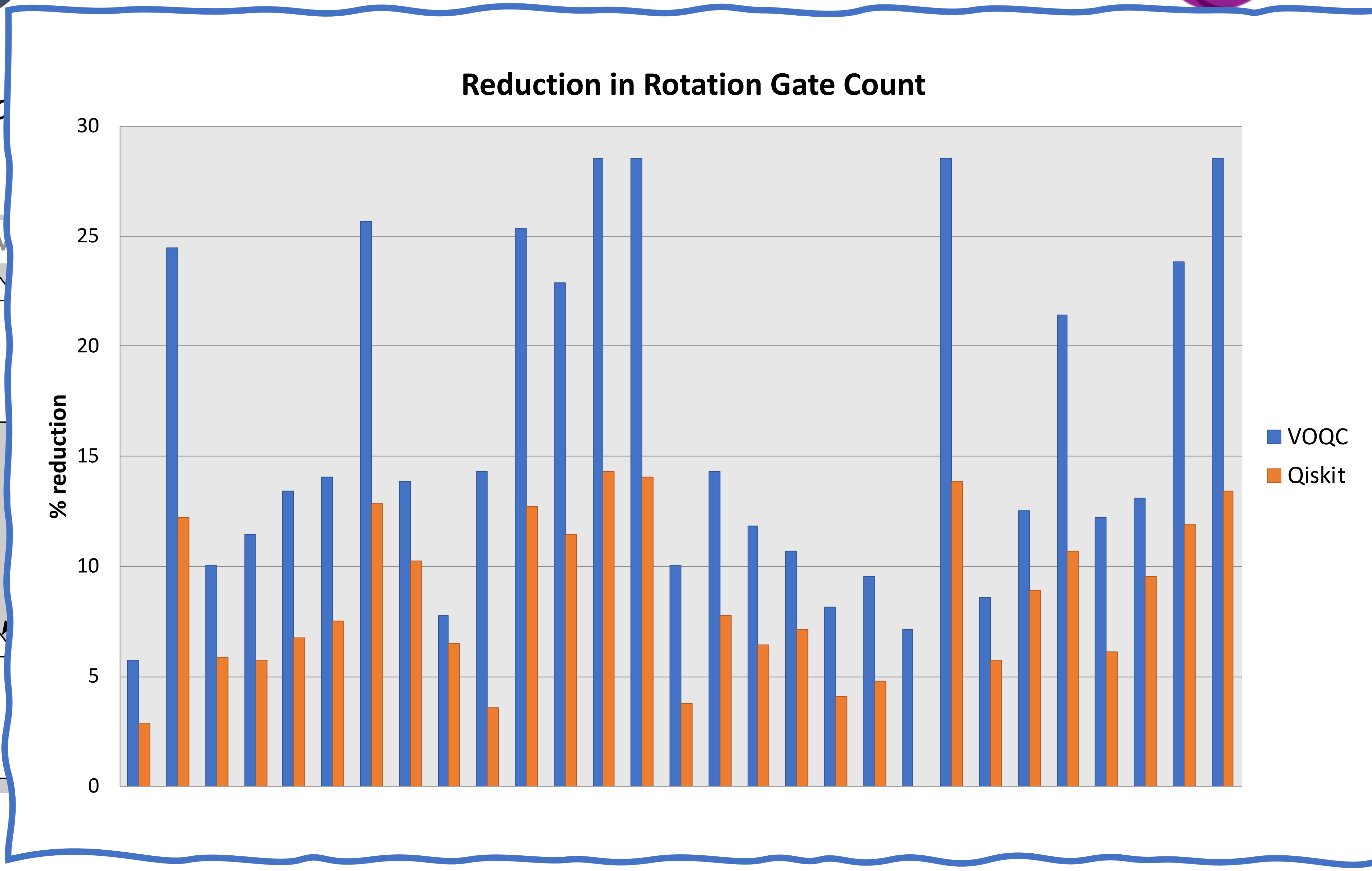
VOQC: a first step towards a fully certified quantum compiler.

SQIRE: a simple quantum intermediate-representation embedded in Coq.

JOINT CENTER FOR QUANTUM INFORMATION AND COMPUTER SCIENCE

(Verified





Description
, connectivity
traints

POPL
2021
nguished
'aper

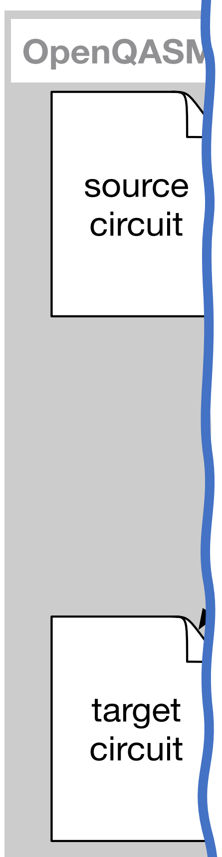
VOQC: a first step towards a fully certified quantum compiler.

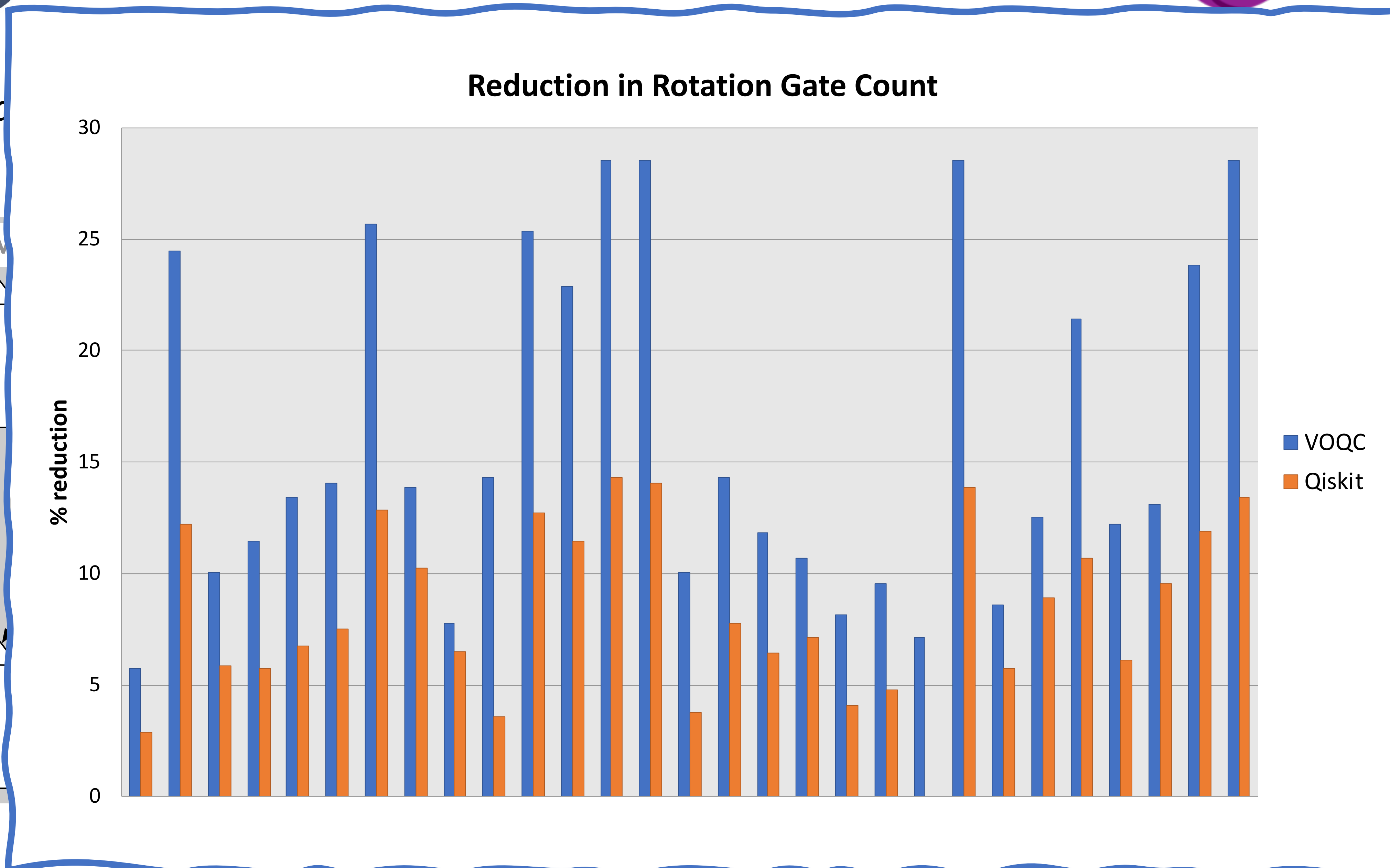
SQIRE: a simple quantum intermediate-representation embedded in Coq.

JOINT CENTER FOR QUANTUM INFORMATION AND COMPUTER SCIENCE

PLUM

(Verified





Description
, connectivity
traints

POPL 2021 nguished aper

VOQC: a first step towards a fully certified quantum compiler.

SQIRE: a simple quantum intermediate-representation embedded in Coq.

Our infrastructure powerful enough:

an end-to-end implementation of Shor's algorithm & its correctness proof.

Example: simple local gate rewrites

- Rz b ; H b ; CNOT a b ; H b ≡ H b ; CNOT a b ; H b ; Rz b
- Rz b ; CNOT a b ; Rz' b ; CNOT a b ≡ CNOT a b ; Rz' b ; CNOT a b ; Rz b
- Rz a ; CNOT a b ≡ CNOT a b ; Rz a
- X b ; CNOT a b ≡ CNOT a b ; X b
- CNOT a c ; CNOT b c ≡ CNOT b c ; CNOT a c
- CNOT a c ; CNOT a b ≡ CNOT a b ; CNOT a c
- CNOT a b; H b; CNOT b c; H b ≡ H b; CNOT b c; H b; CNOT a b

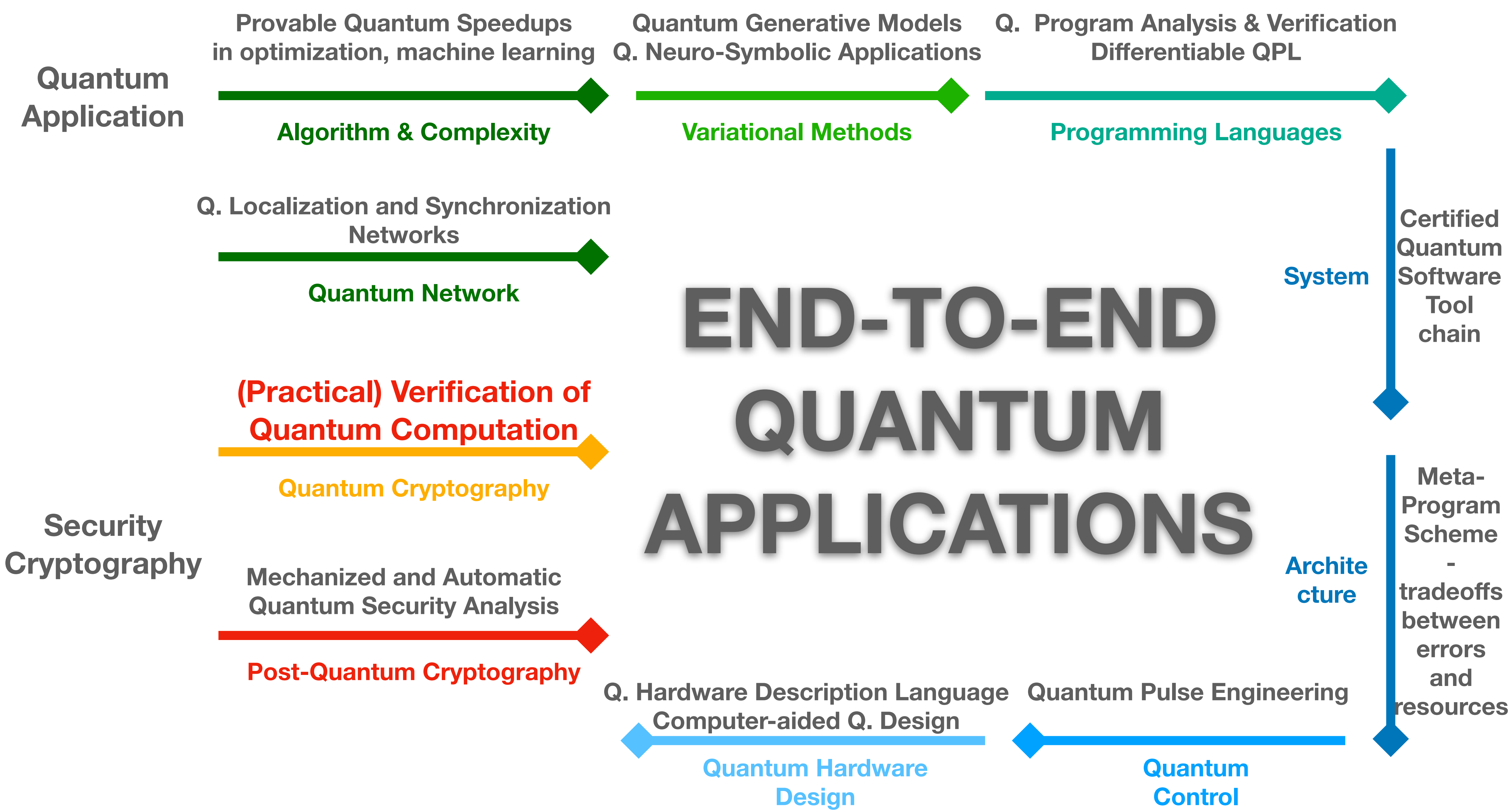
Implementation (~200 lines)

```
Definition Rz_commute_rule1 {dim} q (l : PI4_ucom_l dim) :=
  match (next_single_qubit_gate l q) with
  | Some (l1, UPI4_H, l2) =>
      match (next_two_qubit_gate l2 q) with
      | Some (l3, UPI4_CNOT, q1, q2, l4) =>
          if q =? q2
          then match (next_single_qubit_gate l4 q) with
               | Some (l5, UPI4_H, l6) => Some (l1 ++ [H q] ++ l3 ++ [CNOT q1
               | _ => None
               end
          else None
      | _ => None
      end
  | _ => None
  end.

Definition Rz_commute_rule2 {dim} q (l : PI4_ucom_l dim) :=
  match (next_two_qubit_gate l q) with
  | Some (l1, UPI4_CNOT, q1, q2, l2) =>
```

Spec + Proofs (~700 lines)

```
Lemma PI4_PI4_combine : forall {dim} q k k',
  @App1 _ dim (UPI4_PI4 k) q :: App1 (UPI4_PI4 k') q :: [] =l= App1 (UPI4_PI4 (k+k')) q :: [].
Proof.
  intros.
  unfold uc_equiv_l; simpl.
  repeat rewrite SKIP_id_r.
  unfold uc_equiv; simpl.
  autorewrite with eval_db.
  repeat rewrite phase_shift_rotation.
  gridify.
  rewrite phase_mul.
  repeat rewrite <- Rmult_div_assoc.
  rewrite <- Rmult_plus_distr_r.
  rewrite plus_IZR.
  rewrite Rplus_comm.
  reflexivity.
Qed.

Lemma PI4_PI4_m8_combine : forall {dim} q k k',
  @App1 _ dim (UPI4_PI4 k) q :: App1 (UPI4_PI4 k') q :: [] =l= App1 (UPI4_PI4 (k+k'-8)) q :: [].
Proof.
  intros.
  unfold uc_equiv_l; simpl.
  repeat rewrite SKIP_id_r.
  unfold uc_equiv; simpl.
  autorewrite with eval_db.
  repeat rewrite phase_shift_rotation.
```

Computational Thinking in Quantum Computing



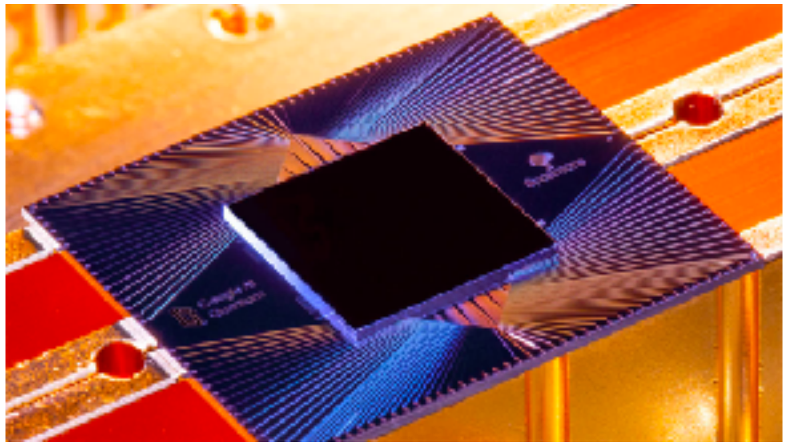
Quantum Supremacy

Preskill (2012): (1) What quantum tasks are feasible? in the near term?
(2) What quantum tasks are hard to simulate classically?

Quantum Supremacy

Preskill (2012): (1) What quantum tasks are feasible? in the near term?
(2) What quantum tasks are hard to simulate classically?

Many proposals: Boson Sampling, Random Circuit Sampling (RCS),
Instantaneous Quantum Computation, ….



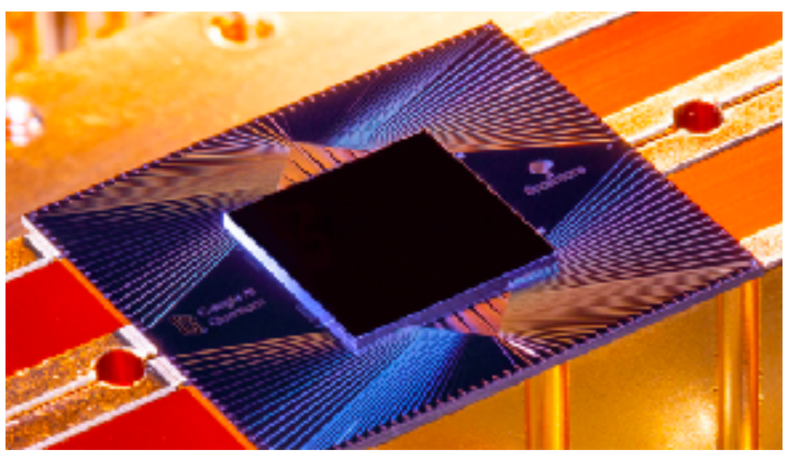
Google Supremacy: RCS (2019)

USTC: Boson Sampling (2020)

Quantum Supremacy

Preskill (2012): (1) What quantum tasks are feasible? in the near term?
(2) What quantum tasks are hard to simulate classically?

Many proposals: Boson Sampling, Random Circuit Sampling (RCS), Instantaneous Quantum Computation, ….



Google Supremacy: RCS (2019)

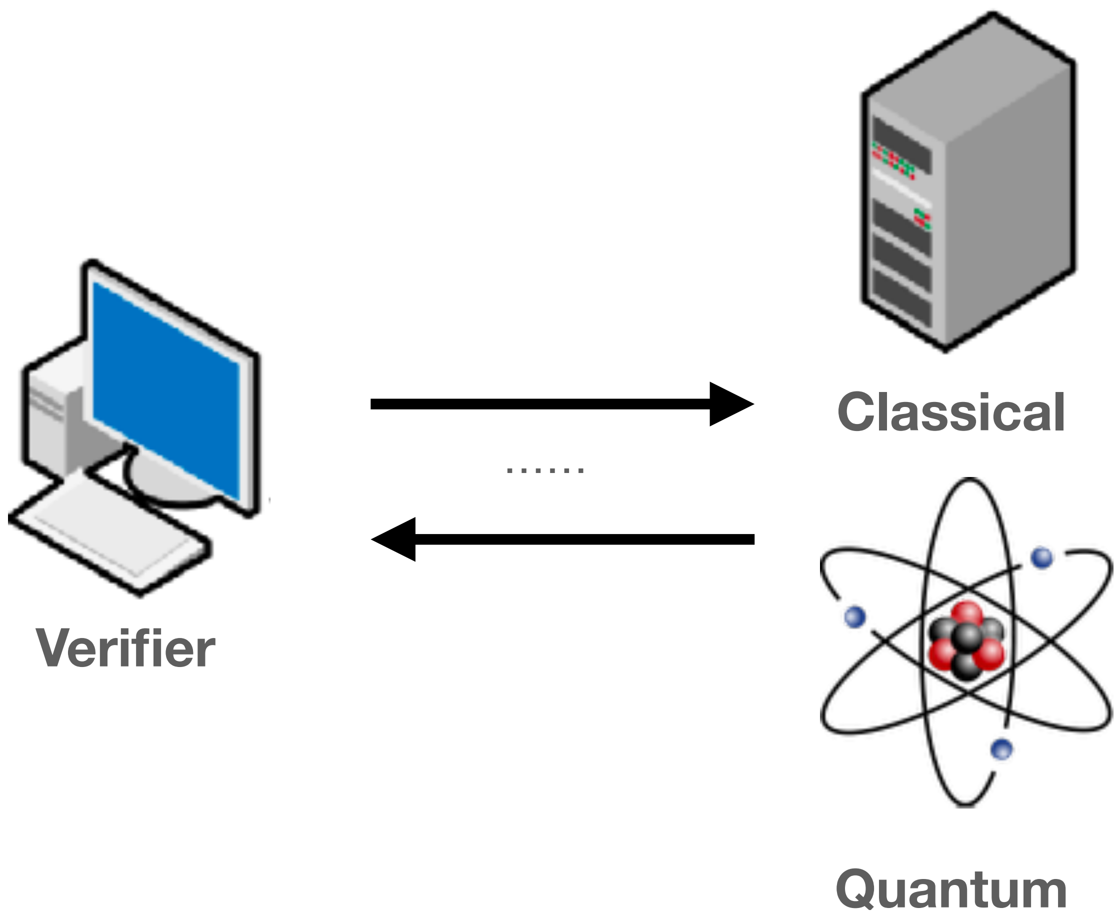
USTC: Boson Sampling (2020)

Theoretical Justification Hardness of classical simulation of the output distribution of quantum supremacy tasks under complexity-theoretical assumptions

References: Aaronson & Arkihipov 11, Bremner & Jozsa & Shepherd 11, Aaronson & Chen 17, Boixo et al 18, Bouland & Fefferman & Nirkhe & Vazirani 19, and so on

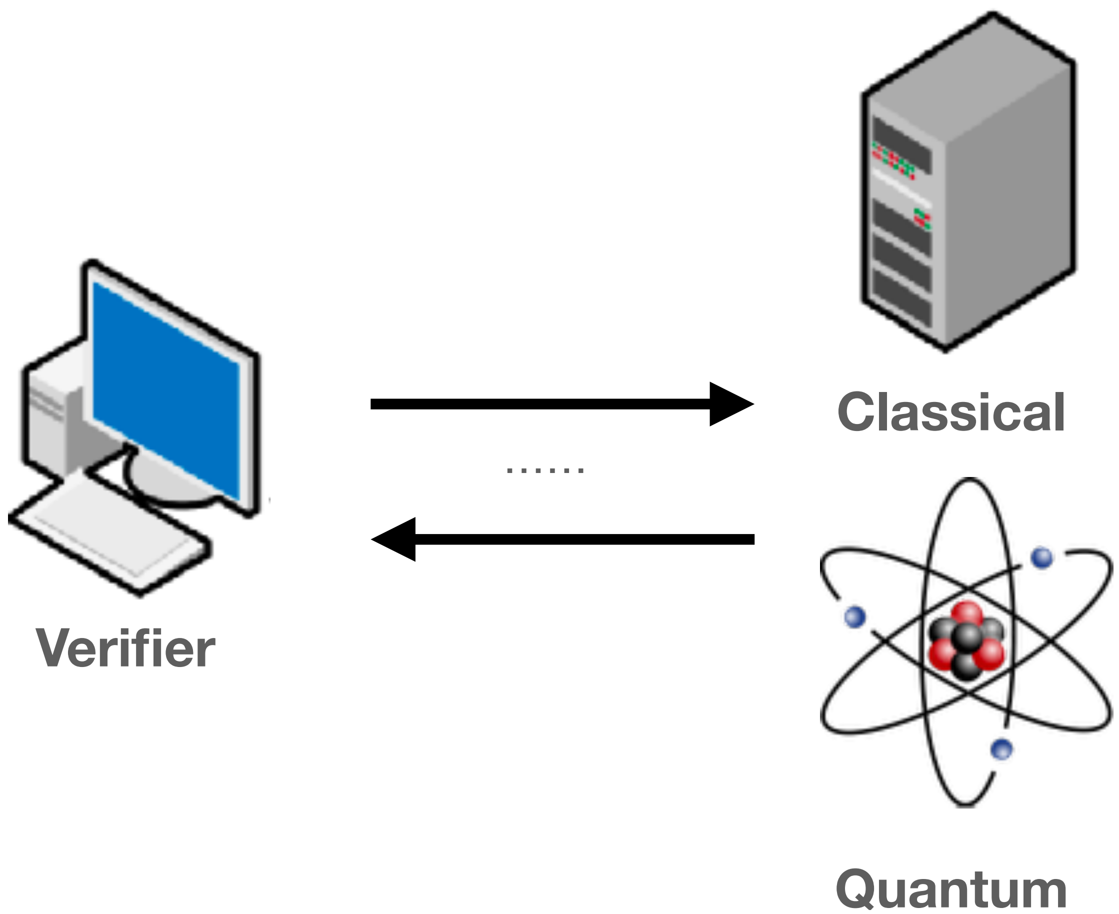
Verifiable Quantum Supremacy

GAP in the implementation: want **verifiability** in the real experiment!



Verifiable Quantum Supremacy

GAP in the implementation: want **verifiability** in the real experiment!

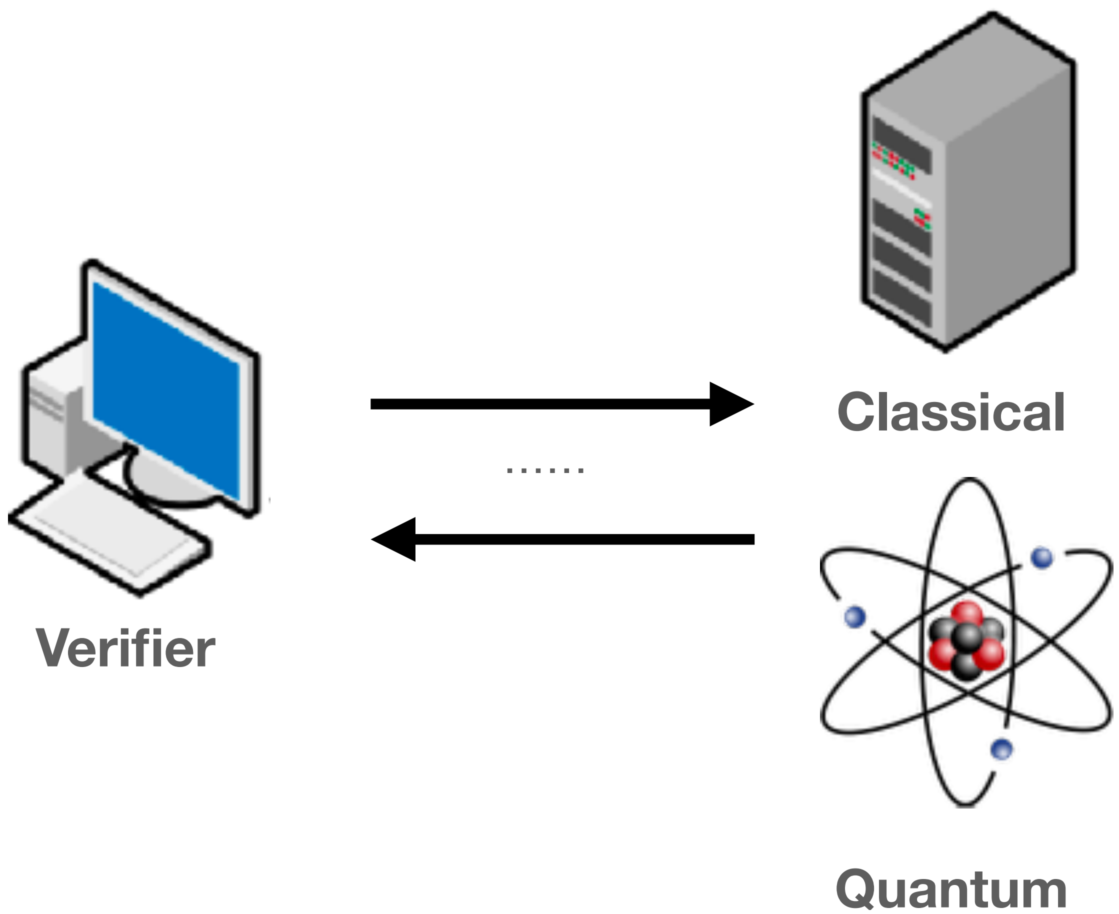


Infeasible Solutions:

- Factoring (in NP)
- Mahadev's delegation

Verifiable Quantum Supremacy

GAP in the implementation: want **verifiability** in the real experiment!



Infeasible Solutions:

- Factoring (in NP)
- Mahadev's delegation

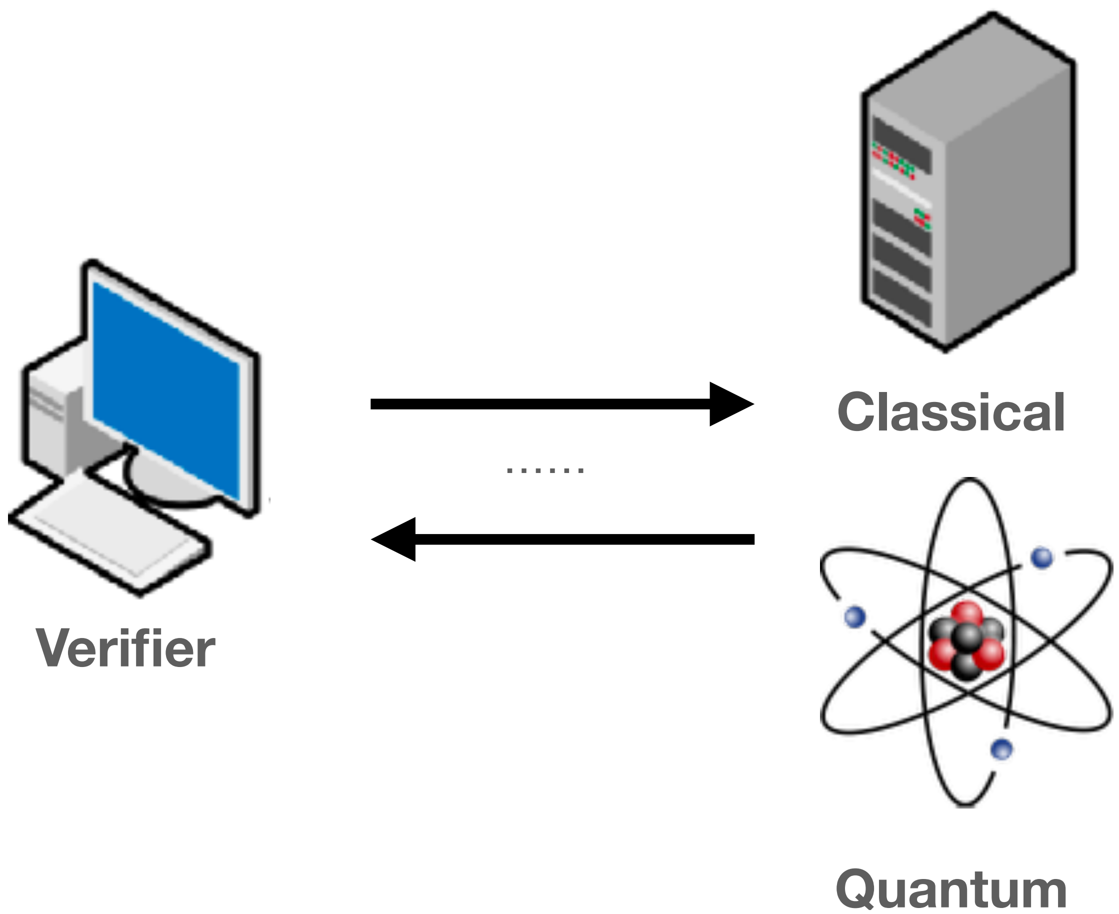
Hardness of Simulation

$\neq$ **Hardness of Spoofing**

needs additional assumption against spoofing (Aaronson & Gunn 20)

Verifiable Quantum Supremacy

GAP in the implementation: want **verifiability** in the real experiment!



Infeasible Solutions:

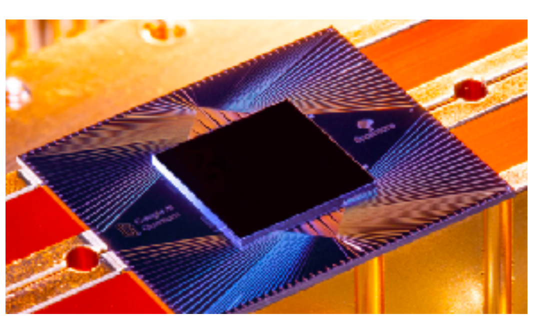
- Factoring (in NP)
- Mahadev's delegation

Hardness of Simulation

$\neq$ Hardness of Spoofing

needs additional assumption against spoofing (Aaronson & Gunn 20)

Expensive Verification Procedure


Google Supremacy: RCS

use supercomputers to calculate the outcome distribution of a given circuit for the verification


USTC: Boson Sampling

simulate Boson Sampling of small instances and then extrapolate

Verifiable Quantum Supremacy: Break the Symmetry

Why it is HARD?

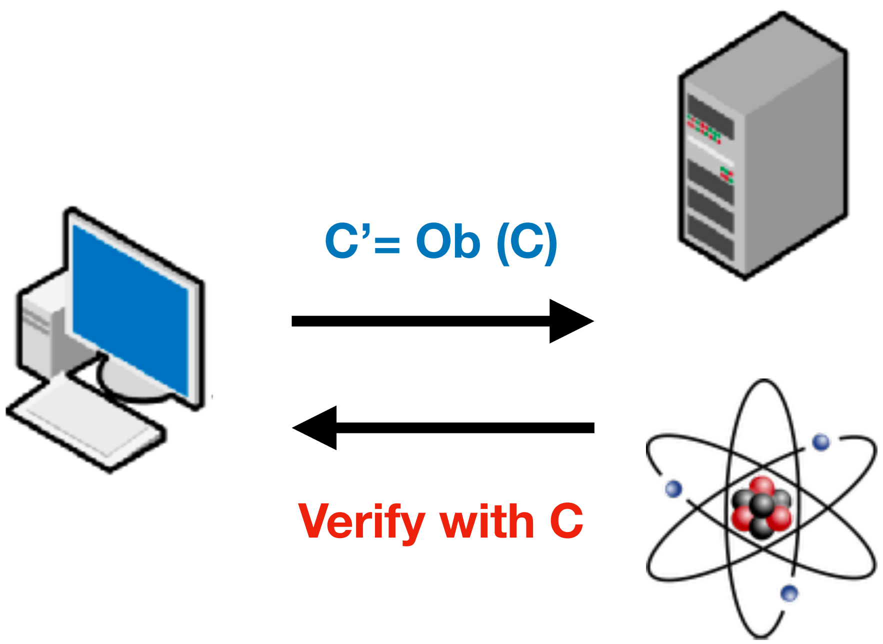
"If n is small enough for verification, it is also small enough for spoofing."
- Scott Aaronson

Verifiable Quantum Supremacy: Break the Symmetry

Why it is HARD?

"If n is small enough for verification, it is also small enough for spoofing."
- Scott Aaronson

Break the symmetry



size-growing circuit obfuscation

C'= Ob (C) : C ≡ C', but C' operates on larger machines, #qbts, #gates

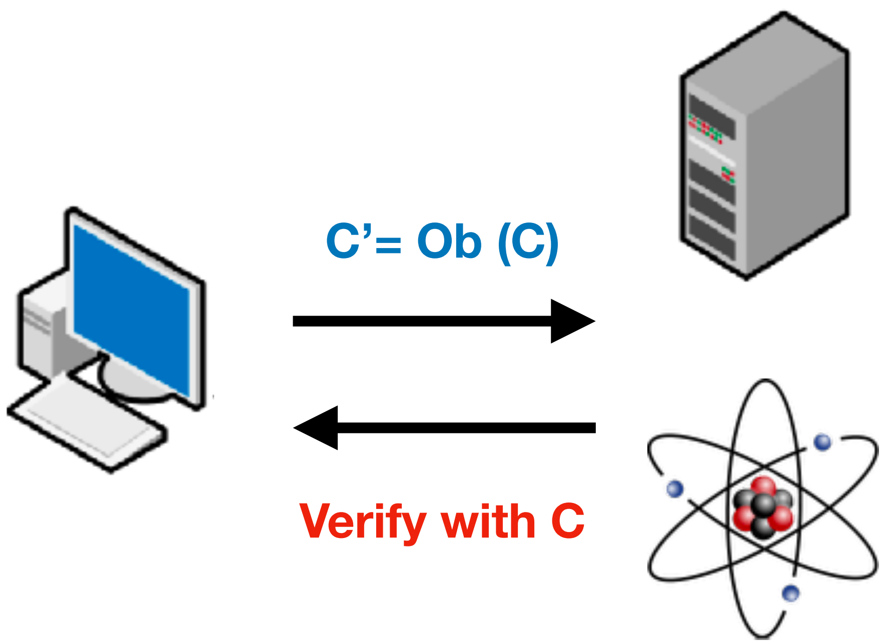
Verify with C: do whatever original verification at the cost of the original C

Verifiable Quantum Supremacy: Break the Symmetry

Why it is HARD?

"If n is small enough for verification, it is also small enough for spoofing."
- Scott Aaronson

Break the symmetry



size-growing circuit obfuscation

C'= Ob (C) : C ≡ C', but C' operates on larger machines, #qbts, #gates

Verify with C: do whatever original verification at the cost of the original C

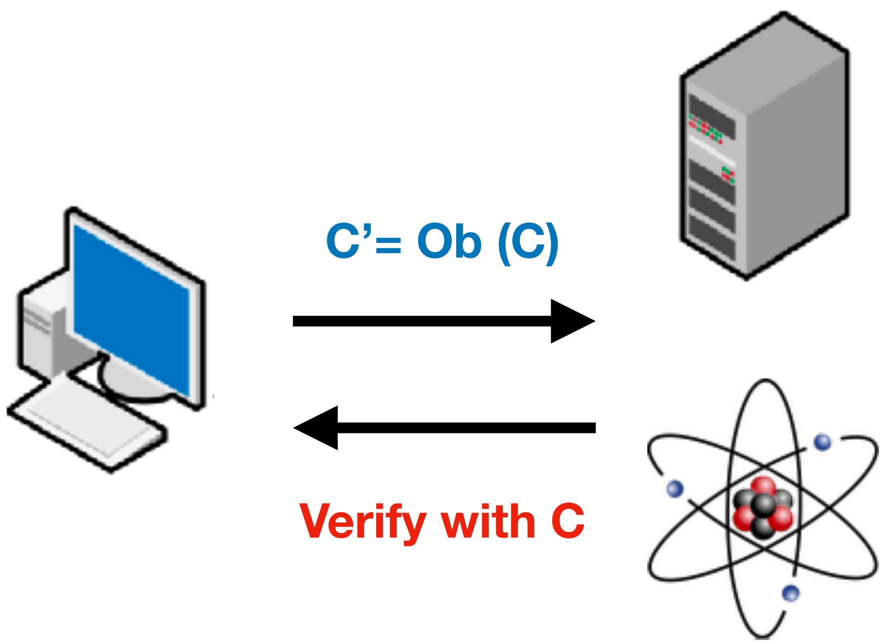
Completeness: quantum machines can run C'=ob(C) and return the answer which will pass the original test

Verifiable Quantum Supremacy: Break the Symmetry

Why it is HARD?

"If n is small enough for verification, it is also small enough for spoofing."
- Scott Aaronson

Break the symmetry



size-growing circuit obfuscation

C'= Ob (C) : C ≡ C', but C' operates on larger machines, #qbts, #gates

Verify with C: do whatever original verification at the cost of the original C

Completeness: quantum machines can run C'=ob(C) and return the answer which will pass the original test

Soundness: intuitively, hard to find C from C', backed by the hardness of Quantum MECP. Need additional assumptions like others.

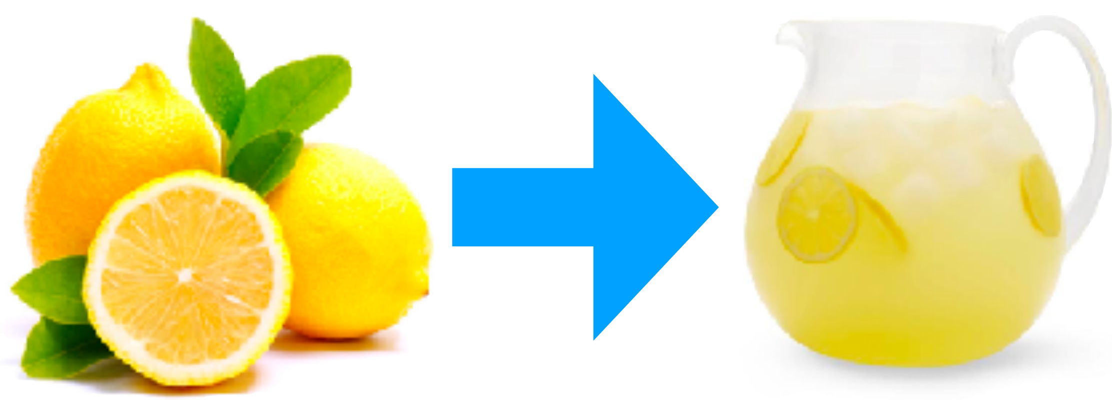
Construction of the Obfuscator

Why? * Need a feasible construction to run!

* Complexity arguments usually asymptotic! Care about empirical performance for a certain parameter range!

Construction of the Obfuscator

Why? * Need a feasible construction to run!
* Complexity arguments usually asymptotic! Care about empirical performance for a certain parameter range!

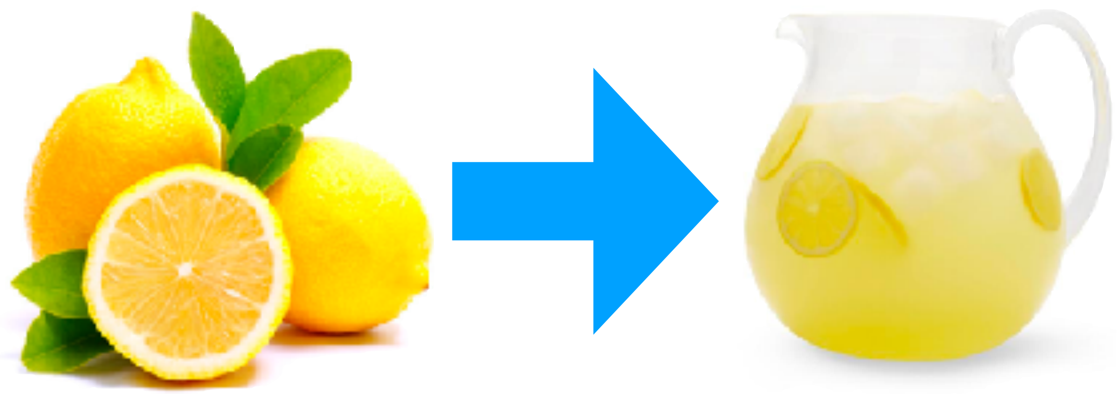


Need to Identify A Problem Where Reducing Circuit-Size is HARD!

Circuit Optimization **as we just see**

Construction of the Obfuscator

Why? * Need a feasible construction to run!
* Complexity arguments usually asymptotic! Care about empirical performance for a certain parameter range!



Need to Identify A Problem Where Reducing Circuit-Size is HARD!

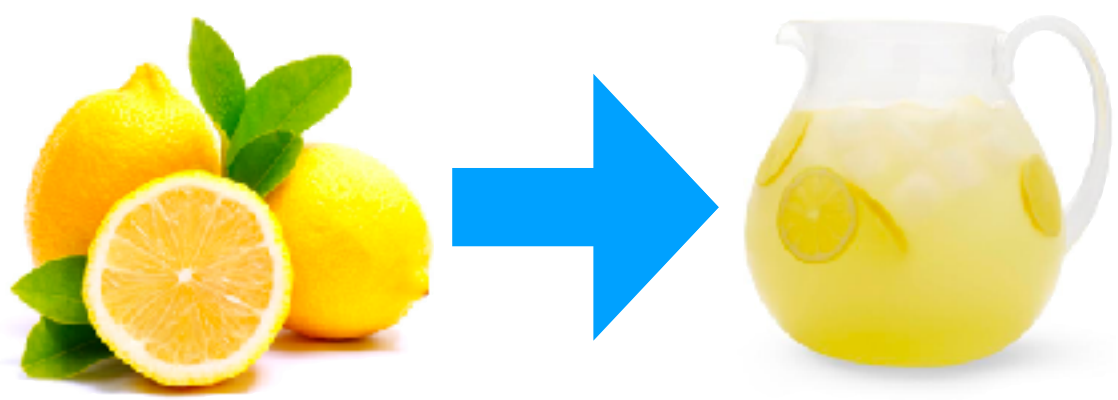
Circuit Optimization **as we just see**

Reverse the construction of Circuit Optimizers

- Reverse the local rewrites used for reducing the circuit size.
- Apply these local rewrites in a **random** order. Identify the order for reducing the size is hard. Identify this random order is harder.
- Also include teleportation + random cancelling pairs to grow the circuit size.

Construction of the Obfuscator

Why? * Need a feasible construction to run!
* Complexity arguments usually asymptotic! Care about empirical performance for a certain parameter range!



Need to Identify A Problem Where Reducing Circuit-Size is HARD!

Circuit Optimization as we just see

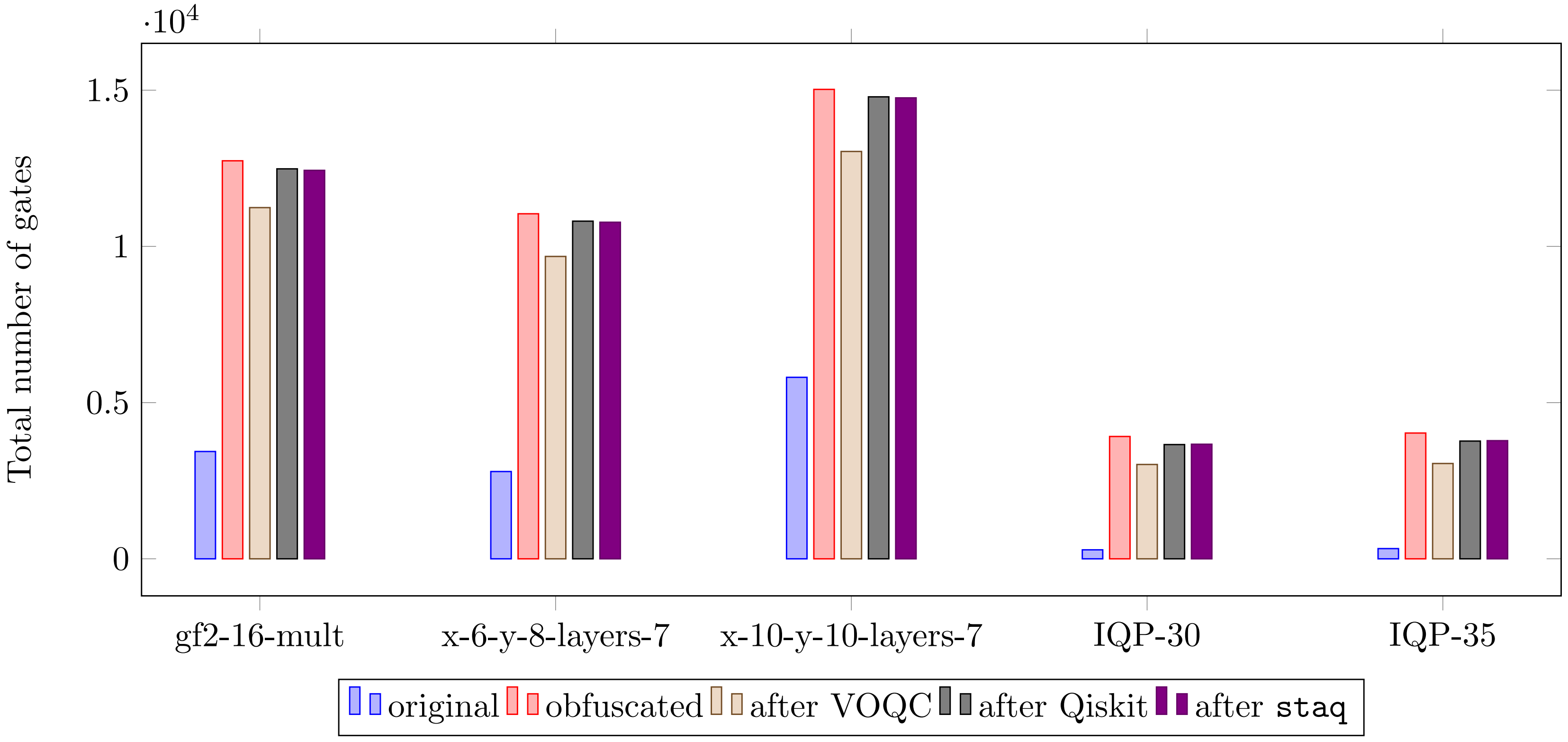
Reverse the construction of Circuit Optimizers

- Reverse the local rewrites used for reducing the circuit size.
- Apply these local rewrites in a **random** order. Identify the order for reducing the size is hard. Identify this random order is harder.
- Also include teleportation + random cancelling pairs to grow the circuit size.

Implementation in Coq with the SQIR infrastructure!

Additional **Benefits**: the correctness of the obfuscation is guaranteed by construction!

Evaluation and Conclusion



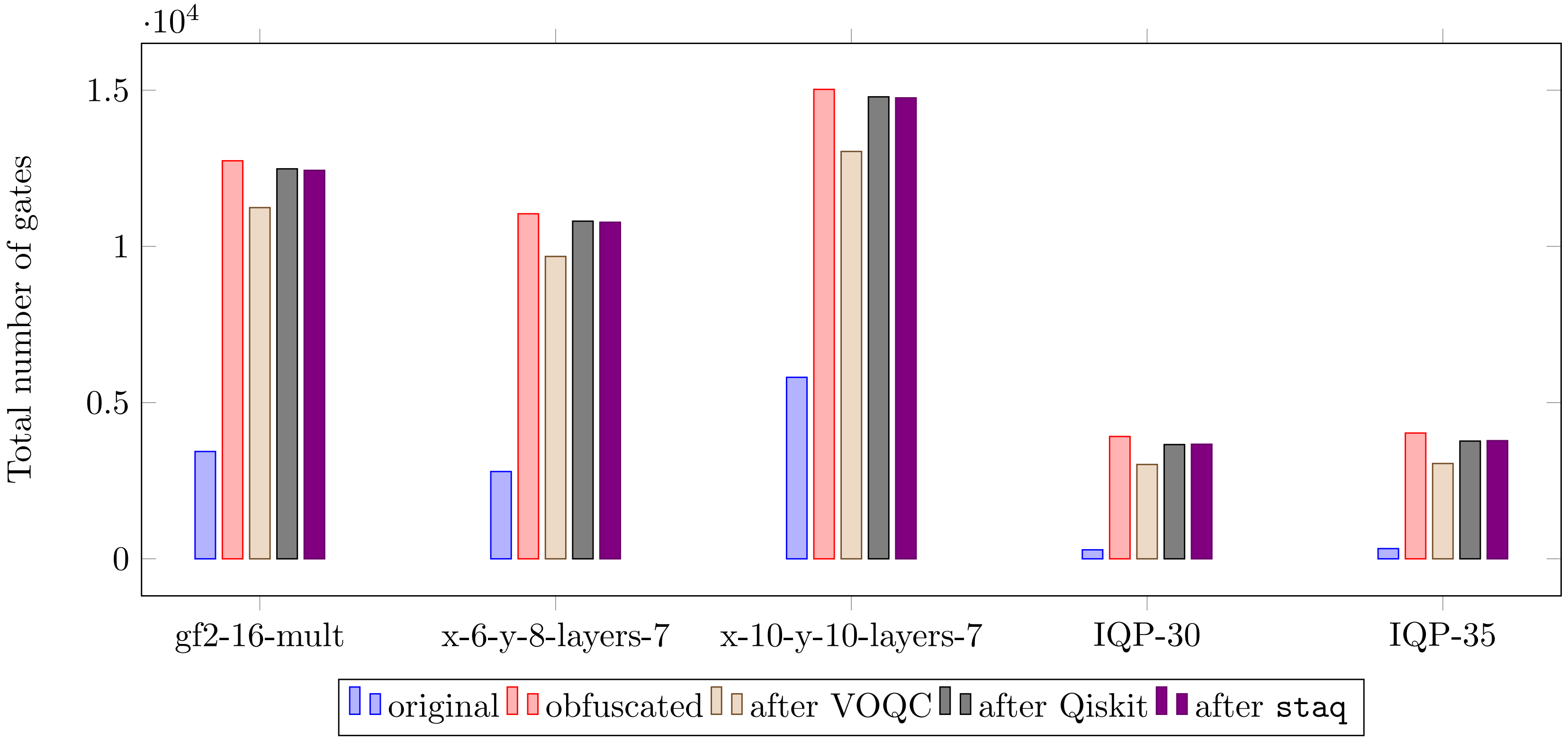
Reducing the Obfuscation w/

- VOQC
- Qiskit
- STAQ
- …

Obfuscated circuits maintain
- all qubits will be entangled during execution
- average depth = # gates/ # qbits at least the one of the original

to avoid simple attacks.

Evaluation and Conclusion



Reducing the Obfuscation w/

- VOQC
- Qiskit
- STAQ
- …

Obfuscated circuits maintain
- all qubits will be entangled during execution
- average depth = # gates/ # qbits at least the one of the original

to avoid simple attacks.

Highly Extensible Framework

- Demonstrate a framework with **theoretical evidence** and **empirical study**.
- This framework is **feasible** for NISQ machines and passes sanity check for its empirical performance.
- The construction of the obfuscation is highly **extensible**. One can easily adjust the framework for different supremacy tasks and experimental platforms.

END-TO-END QUANTUM APPLICATIONS

Quantum Application

- **Algorithm & Complexity**: Provable Quantum Speedups in optimization, machine learning
- **Variational Methods**: Quantum Generative Models; Q. Neuro-Symbolic Applications
- **Programming Languages**: Q. Program Analysis & Verification; Differentiable QPL
- **Quantum Network**: Q. Localization and Synchronization Networks

System: Certified Quantum Software Tool chain

Architecture: Meta-Program Scheme - tradeoffs between errors and resources

- **Quantum Control**: Quantum Pulse Engineering
- **Quantum Hardware Design**: Q. Hardware Description Language; Computer-aided Q. Design

Security Cryptography

- **Quantum Cryptography**: (Practical) Verification of Quantum Computation
- **Post-Quantum Cryptography**: Mechanized and Automatic Quantum Security Analysis

Thank You!

VOQC:
- *github/InQWIRE/SQIR*

Q. Obfuscator:
- *github/shouvanikc/Quantum-Obfuscator*

MQCC:
- *github/sqrta/MQCC*