量子程序并行化执行

> 撰写：叶之帆
>
> 讨论与审核：孙晨寅，吴浩洋
>
> 全体组员：叶之帆（18级计算机学院，组长），孙晨寅（17级物理学院），吴浩洋（18级信息学院）

摘要

以往的量子程序优化主要考虑针对单个量子程序的优化，这种优化建立在这样一种模型上：量子机器逐个执行被提交的每个量子程序。

我们考虑的模型是：多个量子程序可以在同一个量子机器上并行执行，以量子比特作为量子机器的基本资源单位，一个量子机器具有的全部量子比特在同一时刻可能被划分给不同的量子程序。当一个量子比特的所有权从一个量子程序转移到另一个量子程序时，采用reset操作重置该量子比特。

本组考虑的优化手段主要从 量子比特的映射 和 量子程序间执行顺序的重排 两方面着手，试图提高量子机器的并行度。

我们基于全连通假设和非全连通假设提出了两种简单的贪婪算法，并分别进行了测试。

背景

动态的量子比特reset操作

量子比特的初始化是实现量子计算机的一个关键问题，传统的初始化方法是等待一段很长的时间，使量子比特自然回到初始态：

> Preparation of a qubit into a well-defined initial state is one of the key requirements for any quantum computational algorithm. The conventional passive initialization protocol relies on the relaxation of the qubit to a thermal state determined by the residual coupling to the environment. This protocol is inherently slow because the relaxation rate has to be minimized to decrease the probability of errors in a coherent quantum computation.
>
> *引自：Natrure: Efficient protocol for qubit initialization with a tunable environment*

另一种方法是主动使一个量子比特被 reset 到初始状态|0>，一些文章讨论了如何设计一个 reset 协议：
Demonstrating a Driven Reset Protocol for a Superconducting Qubit
这篇文章讨论的 reset 方法可以在 3 μs中 reset 一个量子比特，并且达到至少 99.5 %的可信度。

其他关于动态reset操作的研究：

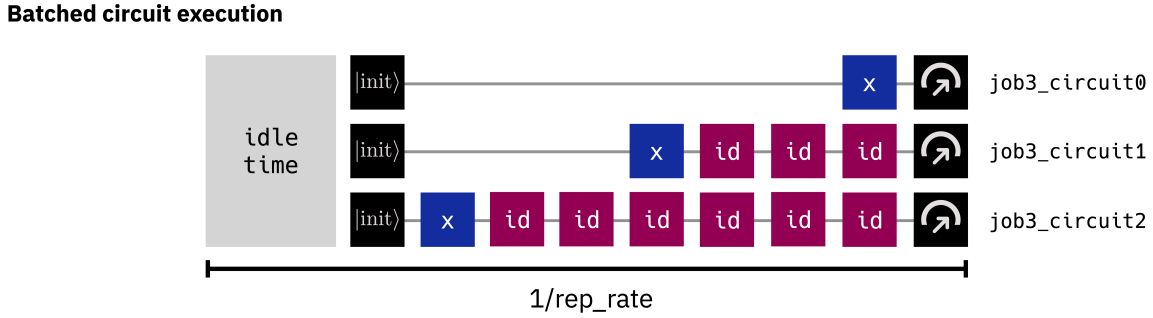
> Digital Feedback in Superconducting Quantum Circuits
>
> Fast and Unconditional All-Microwave Reset of a Superconducting Qubit

在 Qiskit 中，已经加入了对单个量子比特的 reset 操作的支持：
Conditional Reset on IBM Quantum Systems
IBM 的动态reset方法是：测量 + 基于测量结果的初始化，即一个有条件的NOT门

> The `reset` instruction is now supported in Qiskit on IBM Quantum systems. This allows users to reset qubits into the ground state (`|0>`) with high-fidelity. These instructions are comprised of a not-gate conditioned on the measurement outcome of the qubit.

量子机器后端实现：针对批量提交的任务

IBM 在System Circuits Execution提到量子机器对批量提交的量子程序的执行方式是：



量子机器具有固定的执行速率 rep_rate，一个 1/rep_rate 的时间内执行一次量子程序，该量子程序在这些时间内独占量子机器。这种逐个执行的方式阻止了批量提交的量子程序并行执行的可能。

改进

我们希望打破这种顺序执行的运行模型，我们建立的模型是：量子程序不必限定在 1/rep_rate时间内执行，也不必顺序执行批量提交的量子程序。这要求量子机器支持动态初始化和动态测量操作：我们的测量和初始化操作不限定固定周期，而且支持只对部分量子比特测量和初始化。reset 和 measure 正是 IBM-Q 机器将会支持的操作。

时间开销的抽象

由于需要计算量子比特被占用的时间长短，我们抽象每一个量子比特操作的执行时间（也包括 reset 操作的时间）

一个简单的抽象是：

> *参考：CODAR: A Contextual Duration-Aware Qubit Mapping for Various NISQ Devices*
> 定义量子时钟周期(quantum clock cycle)为 τ，所有门操作的时间近似为 τ 的整数倍，相同的门操作所用的时间是一个固定值 nτ，其中 n 取决于具体的底层实现

建模

基于孙晨寅同序的调研结果，我们归纳以下时间开销（超导体系）：

单量子比特门(Rx, Ry, Rz)：20ns

双量子比特门(CNOT)：40ns

测量时间（Measure操作）：~20us

对于reset操作，我们假定的实现方法是有条件的NOT门，如上一节所述。

所以时间开销为： Measure耗时 + 控制电路耗时 + NOT门耗时

所以认为这个时间开销约等于 20us

所以单量子比特门，双量子比特门，Measure操作，Reset操作的耗时抽象为：τ, 2τ, 1000τ, 1000τ

全连通假设：重排算法

输入

批量提交的量子程序，

每个量子程序包含它的OpenQasm代码和执行次数两个属性

特征提取

运行Q-CODAR程序，提取每个量子程序在全连通的量子机器上的执行时间，并且记录量子程序占用的量子比特数

贪婪算法

约束：每个时刻并行执行的量子程序占用的量子比特数之和不得多于整个量子机器的全部量子比特数

目标：在每个时刻，尽可能利用空闲的量子比特

手段：在一个特定时刻，选择一个占用量子比特数最多并且满足约束的量子程序执行（当剩余的空闲量子比特数还可以执行其他量子程序时，再用同样的手段选择一个其他量子程序开始执行，以此类推）

代码：

```
//Greedily Schedule
    int time=0;
    int occupied=0;
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, cmp>
        timeline;
        //timeline represents in progress programs, 最小优先队列

    while(true){
        //Release occupied bits
        if(!timeline.empty()){
            auto next = timeline.top();
            timeline.pop();
            time = next.first;
            occupied -= next.second;
        }
        while(!timeline.empty() && timeline.top().first == time){
            auto next = timeline.top();
            timeline.pop();
            occupied -= next.second;
        }

        //找到满足约束并且占用量子比特数最多的程序进行执行，重复多次，直到不能再找到满足约束的程序
        for(auto& p: programs){
            while(p.bit_useage <= TOTAL_BITS-occupied && p.repeat_times > 0){
                p.start_points.push_back(time);
                p.end_points.push_back(time+p.time_cost);
                p.repeat_times--;

                occupied += p.bit_useage;
                timeline.push(std::make_pair(p.end_points.back(), p.bit_useage));
            }
        }

        //仅当没有在执行的程序时，推出循环
        if(timeline.empty()){
            break;
        }
    }
```

非全连通假设

问题

非全连通模型下，必须考虑并行执行的量子程序之间相互干扰的可能性：

当量子程序A的两个不相邻的量子比特之间需要使用一个CNOT门时，必须将这两个量子比特交换到相邻位置，这需要一连串的SWAP操作，但是SWAP操作过程中可能途径并行执行的量子程序B的所占用的量子比特

这些干扰可能导致映射算法的复杂性大大增加

解决方法：分区

1. 将量子机器的实际物理比特划分成一系列区块，每一个区块中包含一系列相邻的量子比特；
2. 使用映射算法将待执行的量子程序映射到一个或多个相邻的分区中执行，量子程序的所有操作不会超出它所占用的分区；
3. 占用不同分区的量子程序之间必然不会相互干扰

分区方法的优势：

1. 将问题分解为单个量子程序的映射和多个量子程序的执行时机的排序，便于解决
2. 对每个量子程序，只需要执行少数几次映射，就可以在整个执行过程中多次使用映射的结果
3. 便于扩展到不同的量子机器上

算法

单个程序的映射

使用Q-CODAR等算法将每个输入的量子程序映射到一个或多个分区，可以保留将量子程序映射到不同分区的多个映射方案

贪婪算法

约束：每个时刻并行执行的量子程序不得占用重叠的分区

目标：在每个时刻，尽可能利用空闲的分区

手段：在一个特定时刻，选择一个占用分区数最多并且满足约束的量子程序执行（当剩余的空闲分区还可以执行其他量子程序时，再用同样的手段选择一个其他量子程序开始执行，以此类推）

代码：

```
//Greedily Schedule
    int time=0;
    std::vector<bool> occupied(RegionNum, false);

    std::priority_queue<std::pair<int, std::vector<bool>>, std::vector<std::pair<int,
std::vector<bool>>>, cmp>
            timeline;
    //timeline represents in progress programs, 最小优先队列
    while(true){
        //Release occupied bits
        if(!timeline.empty()){
            auto next = timeline.top();
            timeline.pop();
            time = next.first;
            region_release(next.second, occupied);
        }
        //Release bits that should be released at the same time
```

```
        while(!timeline.empty() && timeline.top().first == time){
            auto next = timeline.top();
            timeline.pop();
            region_release(next.second, occupied);
        }

        //找到满足约束并且占用分区数最多的程序进行执行，重复多次，直到不能再找到满足约束的程序
        for(auto& p: programs){
            if(p.repeat_times > 0){
                for(int i=0; i < p.map_count; i++){
                    if(!region_converge(p.region_usage[i], occupied)){
                        p.start_points.push_back(time);
                        p.end_points.push_back(time+p.time_cost[i]);
                        p.repeat_times--;

                        region_occupy(p.region_usage[i], occupied);
                        timeline.push(std::make_pair(p.end_points.back(),
p.region_usage[i]));
                    }
                }
            }
        }

        if(timeline.empty()){
            break;
        }
    }
```

测试

全连通假设

输入

qft_n7算法和qft_n5算法（OpenQASM代码），执行次数之和为1000

Q-CODAR

运行Q-CODAR，提取算法特征（占用的量子比特数，执行耗时），结果如下：

```
qft_5:
5   //占用5个量子比特
2059    //执行耗时（包括reset和measure）为2059
```

```
qft_7:
7
2096
```

重排算法

针对20量子比特的全连通机器，

运行排序算法，针对 qft_n5 和 qft_n7 执行次数的不同组合，得到如下执行结果

执行结果

qft_n5占比	并行执行时间	串行执行时间	加速比
1.00	514750	2059000	4
0.90	543367	2062700	3.796145
0.80	569925	2066400	3.62574
0.70	596483	2070100	3.47051
0.60	623877	2073800	3.324053
0.50	651658	2077500	3.188022
0.40	678216	2081200	3.068639
0.30	733600	2084900	2.842012
0.20	838400	2088600	2.491174
0.10	943200	2093200	2.219254
0.00	1048000	2096000	2

这个执行结果是显然的，当全部输入都是 qft_n5 程序时，同一时间可以执行4个 qft_n5 程序，所以加速比为4，当 qft_n7站全部输入时，同理可知加速比为2

非全连通下的算法

输入

qft_n7算法和qft_n15算法（OpenQASM代码），执行次数之和为1000

分区

将IBM-TOKYO机器的20个量子比特划分为两个10量子比特的分区，原先的联通性如下：

```
// ibm tokyo: IBM TOKYO机器的连通性，QPAIR(i,j)表示标号为i和j的量子比特相互连通
m = 4;
n = 5;
QPAIR(0, 1);    QPAIR(0, 5);
QPAIR(1, 0);    QPAIR(1, 2);    QPAIR(1, 6);    QPAIR(1, 7);
QPAIR(2, 1);    QPAIR(2, 3);    QPAIR(2, 6);    QPAIR(2, 7);
QPAIR(3, 2);    QPAIR(3, 4);    QPAIR(3, 8);    QPAIR(3, 9);
QPAIR(4, 3);    QPAIR(4, 8);    QPAIR(4, 9);
QPAIR(5, 0);    QPAIR(5, 6);    QPAIR(5, 10);   QPAIR(5, 11);
QPAIR(6, 1);    QPAIR(6, 2);    QPAIR(6, 5);    QPAIR(6, 7);    QPAIR(6, 10);   QPAIR(6,
11);
QPAIR(7, 1);    QPAIR(7, 2);    QPAIR(7, 6);    QPAIR(7, 8);    QPAIR(7, 12);   QPAIR(7,
13);
QPAIR(8, 3);    QPAIR(8, 4);    QPAIR(8, 7);    QPAIR(8, 9);    QPAIR(8, 12);   QPAIR(8,
13);
QPAIR(9, 3);    QPAIR(9, 4);    QPAIR(9, 8);    QPAIR(9, 14);
QPAIR(10, 5);   QPAIR(10, 6);   QPAIR(10, 11);  QPAIR(10, 15);
QPAIR(11, 5);   QPAIR(11, 6);   QPAIR(11, 10);  QPAIR(11, 12);  QPAIR(11, 16);
QPAIR(11, 17);
```

```
QPAIR(12, 7);   QPAIR(12, 8);   QPAIR(12, 11);  QPAIR(12, 13);  QPAIR(12, 16);
QPAIR(12, 17);
QPAIR(13, 7);   QPAIR(13, 8);   QPAIR(13, 12);  QPAIR(13, 14);  QPAIR(13, 18);
QPAIR(13, 19);
QPAIR(14, 9);   QPAIR(14, 13);  QPAIR(14, 18);  QPAIR(14, 19);
QPAIR(15, 10);  QPAIR(15, 16);
QPAIR(16, 11);  QPAIR(16, 12);  QPAIR(16, 15);  QPAIR(16, 17);
QPAIR(17, 11);  QPAIR(17, 12);  QPAIR(17, 16);  QPAIR(17, 18);
QPAIR(18, 13);  QPAIR(18, 14);  QPAIR(18, 17);  QPAIR(18, 19);
QPAIR(19, 13);  QPAIR(19, 14);  QPAIR(19, 18);
```

将IBM-TOKYO机器的量子比特分成两个分区：0～9号量子比特和10~19号量子比特

其中0～9号量子比特的连通性表述如下：

```
// ibm half tokyo
m = 2;
n = 5;
QPAIR(0, 1);    QPAIR(0, 5);
QPAIR(1, 0);    QPAIR(1, 2);    QPAIR(1, 6);    QPAIR(1, 7);
QPAIR(2, 1);    QPAIR(2, 3);    QPAIR(2, 6);    QPAIR(2, 7);
QPAIR(3, 2);    QPAIR(3, 4);    QPAIR(3, 8);    QPAIR(3, 9);
QPAIR(4, 3);    QPAIR(4, 8);    QPAIR(4, 9);
QPAIR(5, 0);    QPAIR(5, 6);
QPAIR(6, 1);    QPAIR(6, 2);    QPAIR(6, 5);    QPAIR(6, 7);
QPAIR(7, 1);    QPAIR(7, 2);    QPAIR(7, 6);    QPAIR(7, 8);
QPAIR(8, 3);    QPAIR(8, 4);    QPAIR(8, 7);    QPAIR(8, 9);
QPAIR(9, 3);    QPAIR(9, 4);    QPAIR(9, 8);
```

Q-CODAR

将qft_n6程序映射到两个分区中的任何一个，记录两种映射方案

qft_n15算法必须同时占用两个分区

用Q-CODAR算法分别再上述分区中生成两种程序的映射方案，记录每个映射方案的属性（占用了哪些分区，执行时长），结果如下：

```
qft_n6:

2   //共两种分区方案
1 0 2065    //只占用1号分区，耗时为2065
0 1 2065    //只占用2号分区，耗时为2065
```

```
qft_n15:

1
1 1 2286
```

排序算法

运行排序算法，针对 qft_n15 和 qft_n6 执行次数的不同组合，得到如下执行结果

qft_n15占比	并行执行时间	串行执行时间	加速比
1.00	2286000.00	2286000.00	1
0.90	2160650.00	2263900.00	1.047787
0.80	2035300.00	2241800.00	1.101459
0.70	1909950.00	2219700.00	1.162177
0.60	1784600.00	2197600.00	1.231424
0.50	1659250.00	2175500.00	1.311135
0.40	1533900.00	2153400.00	1.403872
0.30	1408550.00	2131300.00	1.513116
0.20	1283200.00	2109200.00	1.643703
0.10	1157850.00	2087100.00	1.802565
0.00	1035200.00	2065000.00	1.994784

当全部输入程序都是占用15量子比特的QFT程序时，每次执行都同时占用两个分区，所以不能并行执行；当全部输入程序是占用6个量子比特的QFT程序时，每次执行占用两个分区之一，所以可以同时执行两个程序，加速比为2

展望

- 改进我们的分区算法
  - 不同的分区之间允许交叠，交叠的分区不允许同时被使用
  - 在映射到不同分区的方案中自动选择一个合适的方案
  - 如果进行分区划分
- 进行更多测试
  - 探究哪些因素可能影响并行度
- 更细致的物理建模
  - 考虑可信度等等因素

分工

叶之帆：建模，提出算法，协助算法测试，参与讨论；

孙晨寅：相关物理知识的调研与建模，参与讨论；

物浩洋：算法测试，参与讨论