Utilisation de Q# pour la programmation quantique

Pierre Engelstein

1 Introduction

Plusieurs constructeurs mettent à disposition des outils pour tester les algorithmes quantiques dévelopés. Deux techniques existent afin de pouvoir programmer un algorithme quantique. On peut tout d'abord utiliser un ordinateur quantique réel. Plusieurs entreprises fournissent des portails en ligne permettant l'accès à ce genre de solution. L'autre solution est consiste à émuler un processeur quantique sur un processeur classique. Dans le premier cas on est certes sur un vrai processeur quantique, on tire donc pleinement partit des phénomènes quantiques. On est néanmoins limités par la quantité de qubits présents dans le processeur. En revanche, la solution d'émulation n'est pas limitée à la quantité de qubits disponibles et est plus simple à mettre en oeuvre. néanmoins, les phénomènes aléatoires sont ici seulement pseudo-aléatoires (par exemple la mesure d'un qubit qui ne va pas vraiment être aléatoire suivant les poids).

Nous allons voir la solution que propose Microsoft. Dans le cadre de son environnement DotNet, Microsoft fourni un nouveau langage, Q#. Montré lors d'une conférence en 2017, il permet l'écriture de programmes dans un langage haut niveau, compatible avec son langage principal C#. Sa force est de fournir un seul langage à la fois sur un ordinateur classique donc en émulation, et à la fois sur des processeurs quantiques grâce à sa plateforme Azure.

2 Exemple : Génération d'un nombre aléatoire

Pour illustrer les capacités de Q#, nous allons voir la génération d'un nombre aléatoire en utilisant les techniques quantiques. Classiquement, la

génération de nombre aléatoire se base sur des calculs effectués à partir d'une graine, et afin de simuler l'aléatoire on peut se baser sur le timestamp actuel.

Dans le cadre quantique, on peut à la place utiliser une porte de Hadamard appliquée à un qubit initialisé à $|0\rangle$. On obtient : $|u_1\rangle = \frac{1}{2} \times |0\rangle + \frac{1}{2} \times |1\rangle$. Lors de la mesure, on va obtenir soit 0, soit 1 équiprobablement. On peut donc générer un vrai nombre aléatoire de la façon suivante, en utilisant Q# :

```
operation GenerateRandomBit(): Result {
    using(q= Qubit()) {
        H(q);
        return M(q);
    }
}
```

Cette première opération vient créer un qubit, appliquer la porte de Hadamard avec l'opérateur `H(q)`, et enfin effectuer la mesure. L'opérateur `M` vient effectuer la mesure que l'on peut retourner.

Ce langage présente au développeur une liste d'opérations intrinsèques, correspondant aux portes quantiques de base (hadamard, mesure, rotation, etc). Le type `Qubit` décrit un bit quantique, et n'a comme opération disponible que la comparaison. On peut néanmoins utiliser les opérations intrinsèques afin d'effectuer des actions sur ce qubit. Ce mode de fonctionnement correspond bien à la théorie, avec notament l'interdiction de copier l'état d'un qubit vers un autre.

On peut aller plus loin sur la partie de la mesure. Classiquement, lorsqu'une zone de mémoire n'est plus utilisée, on demande à libérer la mémoire de façon à ce qu'elle soit réutilisable plus tard. D'une façon similaire, un qubit après utilisation doit être libéré, et on peut le faire en forçant son état à $|0\rangle$. A la place de l'opérateur `M`, il existe l'opérateur `MResetZ` qui effectue la mesure et remet le qubit dans l'état souhaité.

Ceci représente la seule partie quantique du programme, qu'on peut venir combiner avec de la logique classique pour générer un nombre aléatoire :

```
operation RndInRange(): Int {
    mutable output = 0;
    mutable bits = new Result[0];
    for(idxBit in 1..64){
        set bits += [GenerateRandomBit()];
    }
    set output = ResultArrayAsInt(bits);
    return output;
}
```

Un point d'entrée peut être inclus avec la procédure suivante :

```
@EntryPoint()
operation RandomNumber(): Int{
    return RndInRange();
}
```

Pour exécuter le programme, plusieurs options s'offrent à nous. On peut tout d'abord exécuter le programme Q# en standalone, mais on peut aussi utiliser un programme hôte comme python ou C# dans lesquels on vient appeler les fonctions exposées.

3 Références

1. Microsoft, Microsoft Quantum Documentation, `https://docs.microsoft.com/en-us/quantum/?view=qsharp-preview`
2. Microsoft, QuantumKatas, `https://github.com/Microsoft/QuantumKatas`