Java Assignment

Jan van der Lugt

1 Introduction

The report discusses my implementation of a distributed Rubik's cube solver based on the Ibis framework.

Section 2 will provide a design overview of my implementation, section 3 will go through the problems that were identified during the conversation from sequential to distributed code and how they were solved and section 4 will discuss performance results.

2 Design overview

My design revolves around a single double-ended queue (deque), which is maintained by the master. The master works on the front of the deque, where it removes a cube and places its children back on the front of the deque. This will cause the cubes with the most work to always be at the end of the deque. In the absence of worker, the master performs the work quite similar to the sequential implementation, with two small differences:

- The master doesn't use recursion, but use a work deque, which incurs a little more overhead, since adding and removing from a deque is more expensive than a function call. In the case of recursion, however, it is very difficult to steal work from the master.

- The children are processed in reverse order, since cubes added to the deque are processed in LIFO order. In micro-benchmarks this did not make any difference, which is to be expected because all cubes are explored up to a certain bound anyway.

I chose to let the workers steal cubes instead of have the master push them to the workers, since this scheme has the least bookkeeping. Workers always steal from the end of the deque, since these cubes have the most work left to do.

The workers still use the recursive solver, since it is faster and other workers will only steal from the master. In the case of real job-stealing, where every worker can steal from every other worker, they would also have to use a deque, in a similar fashion to the master.

Before the workers can steal work, the master fills the deque with two generations of cubes (in the case of cube of size 3, this amounts to 144 cubes). Future work could include setting this number dynamically based on the number of workers. In this filling phase, the master also works from the end of the deque, so the deque will have cubes with equal and relatively large amounts of work.

3 Problems during implementation

3.1 Concurrency and synchronization

In the presence of n workers, there are multiple threads active in the master: one main thread and up to n threads of workers requesting work. In order to make sure threads only steal work when they are supposed to and wait when necessary, several mechanisms are used to make sure everything works according to plan:

- A *status* enum which denotes the current phase of the computation. This can be initialization, filling deque, processing deque, deque empty, waiting for workers or done.

- 'synchronized' statements are used to make sure two threads are not accessing the deque at the same time. They also make sure that the status enum can be changed atomically based on an observation of the deque (for example when it is empty).

- notify(), notifyAll() and wait() calls are made to sleep when this is necessary (for example when workers need to wait until work is available).

- AtomicIntegers are used to store the number of solutions and the number of workers the master is still waiting for.

3.2 Handling sendports on the master

TODO

3.3 Termination

Because I was having some trouble with Ibis termination mechanism, which is probably caused by not having an example available explaining how to use it, I created a very simple termination mechanism:

Every cube that is sent from the master to the worker, is preceded by a boolean. Normally this boolean is false, but when the master decides it is time to terminate, it sends a true, signaling the workers to quit.

Everyone then closes their sendports and then their receiveports (since closing a receiveport blocks if there is still a sendport connected to it).