Roman Kazarin
Michael Nekrasov
CS 290B – Final Project
Spring 2015

Abstract

Our work extends the forks/join Java Spaces implementation to accept less structured distributed applications like Ramsey Search. While some embarrassingly parallelizable problems such as the Traveling Salesman Problem and Fibonacci Sequence can be generically solved using a task decomposition method that can be agnostic to the computational capabilities it is running on, there is a wider class of distributed projects that benefit from controlled distribution of work. We provide an infrastructure that both provides a general fork/join task infrastructure and allows fine grained control of scheduling, work distribution, and application layer access to the capabilities of the underlying computational units. Our system gives a clean abstraction to the distributed computing architecture, while allowing deep customization when the application requires it. We test our system using the cases of the Traveling Salesman Problem and the Ramsey Search.

Introduction

Many distributed applications can fit into a Fork/Join model of computation. In this model tasks spawn subtasks which are then further distributed. Once a level of computation has been reached where the cost of parallelizing the task outways the runtime of the subtask the task performs its computation to produce a result. Tasks then join back up results fusing the computation into a single answer. The Traveling Salesman Problem (TSP) asks given a list of cities and a list of distances between them, what is the shortest walk that visits every city. The traveling salesman problem fits nicely into the fork join paradigm. The permutations of all possible walks can be subdivided, distributed out, compared, and joined back together producing the best possible walk. During the computation tasks are largely independent, similar sized, and can be scheduled and distributed in a fairly generic way.

Not every distributed application fits into a clean decomposition in this manner. Lets take the example of the finding a Ramsey number, a problem in combinatorics. The goal of a Ramsey Search is, given a condition R(m, p), is to find the smallest number of nodes that make a fully connected graph, such that there is a monochromatic subclique of at least one color. While an exact solution for the symmetric case R(5, 5) does not yet exists, what is known is that the solution lies within the range of [43, 49]. The counterexamples for Ramsey for higher graph sizes become rare and become valuable in themselves. Counterexamples for a Ramsey Search act like prime numbers and can be potentially used for a variety of functions like proof of work, cybercurency, and potentially encryption. Counterexamples can be used as seeds for further search.

In the case of the Ramsey Search, tasks need to be generated based on the overall state of the system, not solely based on the work of a particular task. As the search has a random element to it, task runtimes are unpredictable and varied. Complexity of the work increase exponentially with higher graph sizes, while counterexamples on lower graph sizes retain value as they can be used as seeds for further search. The search has no reachable end point as the complexity is $O(2^{n^2})$ so a full search of R(5,5) on a 43 node graph would take checking 2^{1849} graph permutations. Therefor the fork/join model was ill suited to this type of computation.

Instead of creating an application specific system each time we encountered a new computational model we instead proposed extending the fork/join Java spaces infrastructure to allow a higher degree of customization that could fit a variety of distributed applications that had a Space (frequently referred to as a Master) that directed work and Computers (frequently referred to as workers) connected to the space that executed the work. Our goal was to build an infrastructure that was malleable so that an application programmer could develop and deploy a Fork/Join application like TSP and Ramsey Search on the same infrastructure.

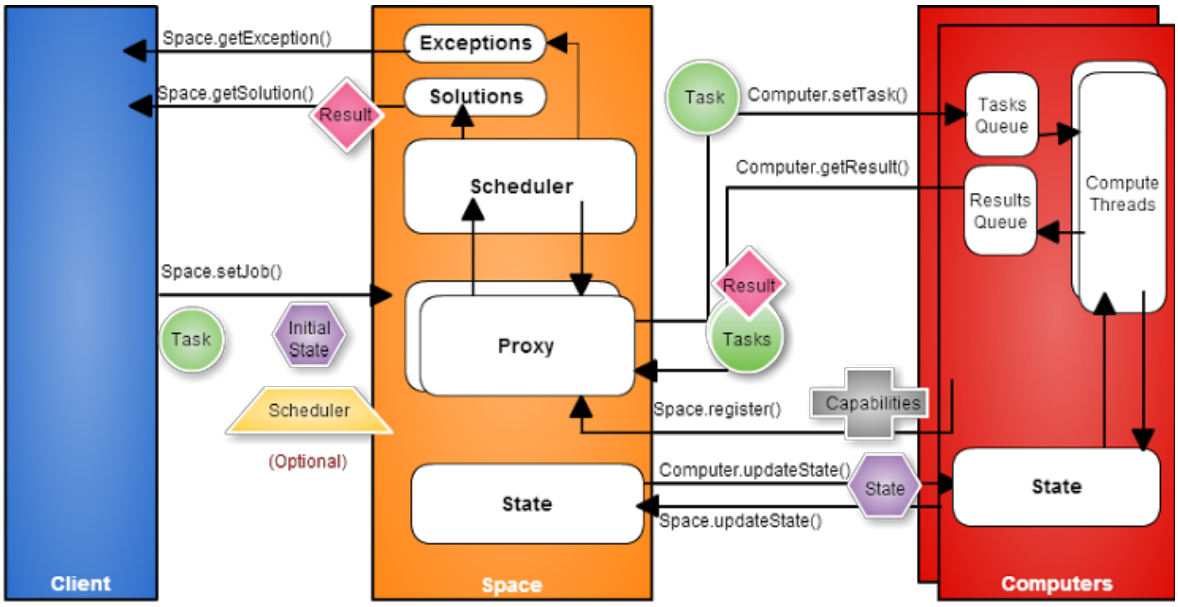
Functional Requirements

1. Infrastructure can still run TSP Fork/Join Model
    - Test: Run TSP and get an answer
2. Infrastructure can have modular schedulers
    - Test: After running TSP, without restarting the space Start Ramsey Search
3. Infrastructure still reliable
    - Test: For both TSP and Ramsey can kill computer without disrupting computation
4. Ramsey Search Application continuously computes Ramsey Counterexamples
    - Test: Can Inspect Counterexample output to verify Ramsey Counterexample, Sizes of counterexamples should increase with time
5. Ramsey Search Application stores a counterexample whenever it is found
    - Test: Does graph store show that the found counterexample has been saved?
6. Ramsey Search Application checkpoints computation
    - Test: Wait until counterexample of size n is found. Shut the system down. Start the system up again - the application should start the search at n + 1.
7. Infrastructure & Application supports machines with different capabilities and longevity.
    - Test: Register at least one self-identified ‘long running’ computer with and a self identified ‘short running computer with the space.
      Wait until a counterexample of size 31 has been found.
      The long running computer should continue to graph size 32. The short running should go back to 25.

Key Technical Issues

1. One architecture, for different problem types expands this problem beyond just Ramsey Search
2. Look for biggest possible counterexamples for R(5,5). a $O(2^{n^2})$ problem. We will **not** try to find the actual solution for R(5, 5).
3. Avoid getting stuck at local minimums
4. Leverage heterogeneous computer types.
    a. Computers with poor longevity can generate seeds, but are unlikely to be useful in finding higher order solutions
    b. Computers with longevity should start with best possible starting point

Architecture



Our architecture extends the fork/join model from this course. Computers handle execution of tasks. Each computer executed tasks on one or more ComputeThreads (usually one thread per core on the machine). Computers register with a Space and provide their capabilities (number of cores, buffer, predicted longevity, etc…). Computers maintain a buffer for taks and results to improve network performance.

Spaces connect to Computers via a Proxy class and assign tasks and collect results. Spaces run Schedulers that handle task assignment and process the results. When a particular task generates a new shared state that state is passed to the parent computer and if propagated to the space and in turn to the rest of the computers in a best effort manner.

Clients connect to the space, and are likely to represent the application programmer using the infrastructure. A client starts a job by providing an initial task (and its implementation) to the Space. In the simplest fork/join case this is all that is needed to run an application.

The application programer is provided with a high degree of control of the system. Our design philosophy was to provide default implementations that work in a general case so as not to overwhelm the application programmer, while allowing customizations of infrastructure components as the programmer familiarizes themselves with the system and needs to run more complex tasks that require application specific optimizations.

Distributed jobs are defined by a collection of tasks. Tasks override an execute method of a TaskClosure and produce a Result. The result must contain zero or one values (that can be passed to another task), zero or one exceptions (that are passed back to the client), zero, one or more new tasks (that are scheduled for execution on the space).

Tasks provide built in customization and optimization. When the application programmer defines a task they may specify certain flags allowing the space to optimize the execution. A task can flag that it should be run on the space directly (for example if it is a simple composition or decomposition that would be severely impacted by network transit costs). A task can also specify a priority for a task (for example in the TSP branch and bound scenario a priority on depth in the computation DAG can potentially produce a stronger bound and reduce overall computation time. Tasks can also produce partial results that are passed back to the Computer and Space using a callback method supplied when they are executed..

For performance optimizations a client can implement a shared state that is passed between the computers and the space. If an initial state is not provided by the application programmer an immutable state is automatically selected by the infrastructure containing no values. If the client does provide an implementation for the state they can pass it along with the initial task. For example in the TSP implementation a state is used to pass along the best upper bound found so far. This is used to shortcut computation using the branch and bound method.

For applications that do not fit well into the fork/join model and benefit from fined grained control of scheduling, an application programmer can implement a custom scheduler that is passed to the space. The scheduler has access to the capabilities of the computers, and can make scheduling decisions based on the state of the overall system and the computers connected to it at a given time.They are in full control of assigning tasks and processing results. Schedulers can even extend out beyond the infrastructure to implement custom features like application specific checkpointing. In the Ramsey Search example we implement a scheduler that accesses an RMI registry to pull up a graph store where it submits solutions and pulls out new ones which has automatic checkpointing and works independently of the Java Spaces infrastructure.

Note that while the customization is available, for an application programmer to use our system they need only define a task. The rest of the customization options have natural defaults, that are only modified as necessary.

Experiments

The experiments we conducted revolved around testing our functional requirements and asserting that what we envisioned actually happened. Our experiment environment consist of one machine that ran the Space and RamseyStore. Three ComputerNodes were then ran on three separate machines - two were made “long-running” while one was made to be “short-running”. The space, at the end of this stage looked like the following:

```
runSpace:
     [java] Starting Space as 'Space' on port 8001
     [java] Registering Computer[0] '1 Threads' 'Buffer:1' 'On-Space'
     [java] Registering Computer[1] '4 Threads' 'Buffer:10' 'Long-Running'
     [java] Registering Computer[2] '4 Threads' 'Buffer:10' 'Long-Running'
     [java] Registering Computer[3] '4 Threads' 'Buffer:10'
```

We then ran a TSP Client of size 13. While the system was computing this job, we proceeded to kill one of the long running computers - the job finished successfully. This proved that our infrastructure is reliable and can handle a Computer Node going down.

After the TSP task finished with the correct result, we then restart the ComputerNode that we initially killed. The number of computers is now back to 3. We then proceed to start a RamseyClient **without** restarting the space. We observe that the application is running and continuously mining larger and larger counterexamples. We also see that whenever a ComputerNode finds a counterexample, after a few seconds the GraphStore updates and shows that this counterexample has been stored successfully.

We now test the checkpointing mechanism. We wait until the GraphStore has a counterexample of size 33 stored. We shut down the entire system. When we start up the GraphStore again we see that the GraphStore was indeed able to load the store that it saved after quitting:

```
runRamseyStore:
     [java] Starting Store as 'GraphStore' on port 8002
     [java]
     [java] Loading Store from: 'bank.save'
     [java] ---------------------------------- Size:1130 ----------------------------------
     [java] 0:  -      1:     -      2:     -      3:     -      4:     -
     [java] 5:  -      6:     -      7:     -      8:     -      9:     -
     [java] 10: -      11:    -      12:    -      13:    -      14:    -
     [java] 15: -      16:    -      17:    -      18:    -      19:    -
     [java] 20: 95     21:    95     22:    95     23:    95     24:    95
     [java] 25: 95     26:    95     27:    95     28:    95     29:    93
     [java] 30: 80     31:    61     32:    39     33:    2      34:    -
     [java] 35: -      36:    -      37:    -      38:    -      39:    -
     [java] 40: -      41:    -      42:    -      43:    -      44:    -
     [java] 45: -      46:    -      47:    -      48:    -      49:    -
     [java] --------------------------------------------------------------------------------
     [java]
```

We then test whether long-running computers will start at the highest available checkpoint and whether short-running computers will start at a graph below their allowed limit of 32. We first start up all of the original computers and then start the RamseyClient. We then observe our ComputerNodes: both long-running computers indeed start at graph size of 33. The short-running computer starts at a graph size of 26, showing that our application supports computers with different capabilities and longevity.

We expected our application to reach some graph size rather quickly and then slow down indefinitely. Our Ramsey application gets to a graph size of 35 in less than 5 minutes. A graph size of 36 is reached within another 20 minutes. However after running the application for over 7 hours we are still stuck at a graph size of 36. We suspect that if we let our application run for an extended period of time we can solve a graph size of 36. Also, because the point of this project was to use JavaSpace to distribute this algorithm, the actually Ramsey algorithm may not be the fastest/most optimal.

Conclusions

In conclusion we present a modular infrastructure on Java Spaces that is able to run both a fork/join model of computation like TSP and application specific computation like Ramsey search with virtually no overhead. We are happy with our API and believe that it is simple to learn, and allows the application programmer configurability of the system as they need it.

While the Ramsey Counterexample Search algorithm we present is not an optimal one (as that could be a massive project in itself), it does show how we can use Java Spaces to efficiently distribute the work across multiple machines, in a clean manner. We leave the optimization of the Ramsey Search itself to future work.

We liked our approach to computers with different capabilities, and future work could utilize features about computers such as memory, cpu, average uptime, to generate even more optimized scheduling patterns.

The next logical step is handling the early termination of a job that is currently executing. While the TSP problem has a finite solution, the Ramsey Search will never finish. Ideally it should be possible for a client to terminate an inprogress job. One way to do this is to ask the application programmer to have their tasks check for an interruption and gracefully terminate. In a real scenario however, that may not be sufficient as the application programmer could make a mistake and endlessly loop the code. Starting tasks in their own JVMs may lead to a more robust solution, but we also leave that to future work.