Benchmarking MASS JAVA

By: Jonathan Acoltzi

Advisor: Dr. Fukuda

Table of Contents

Purpose: ***1***

Quarter Summary ***2***

JCilk Benchmarks **2**

MASS GraphPlaces **2**

MASS Annotations **2**

Results ***3***

Programmability **3**

JCilk 3

MASS GraphPlaces 3

MASS Annotations 4

Performance **5**

JCilk 5

MASS GraphPlaces 6

MASS Annotated 6

Further Analysis **7**

Conclusion ***7***

Future work **7**

Citations ***8***

Purpose:

The purpose of my research project is to further benchmark MASS and compare against competitors.

My tasks included

- Developing a closest pair of points(CPP) and triangle counting algorithm using JCilk
- Rerun and get new results for KD-Tree range search program on MASS Java
- Work on Shortest path program and rewrite using MASS Annotation feature
- Gather data of all developed programs

My overall goal this quarter was to create benchmark programs for both MASS and competitors that can then be used to compare performance of MASS and programmability.

Quarter Summary

This quarter I worked developing benchmark programs using MASS and JCilk.

JCilk Benchmarks

JCilk is a multi-threaded programing language that is based on Java. Unlike java, JCilk can fork and join threads with no need for a runnable class or a start() method.

For tringle counting program I first created a graph. Each vertex is then accessed in parallel. At each vertex, the neighbors are hoped to if the ID is less than the current vertex ID. This step is done twice. After this step is done twice, the original vertex is searched for in the current vertex's neighbors.

The next program implemented in JCilk was CPP program. In CPP points in a 2d plane are given and then search for the closest points. The algorithm used is divide and conquer algorithm. In this algorithm the points are first sorted by the x-coordinate, the points are then divided by two halfs recursively. This is done until 3 points are left. At this point the closest pair between the three are found. The closest pair is returned along with an array with the points sorted by the y-coordinate. Once the answer is returned the closest pair from the right and left are compared and the closest is kept. The next step is to merge the sorted list of points that were returned by the left and right halves. After this point the strip in the middle where the distance from the middle is equal or less than the smallest distance of the pair of points.

Source code for JCilk programs can be found at:
https://bitbucket.org/mass_application_developers/mass_java_appl/src/f5b8d1dcf7ce20715cfadb6abb240c83d11fb9fe/Benchmarks/jcilk_release/benchmarks/?at=jacoltzi%2Fbenchmarks_jcilk

MASS GraphPlaces

Apart from JCilk programs I also re-evaluated the performance of MASS GraphPlaces' implementation of KD-Tree Range Search. Our data showed a significant increase in performance than what was originally recorded when submitting paper to IEEE BigData Conference.

Source code for KD-Tree can be found at:
https://bitbucket.org/AC_rojas/mass_appl_jonathan/src/master/

MASS Annotations

Nearing the end of the quarter I was able to write a shortest path benchmark program using MASS Annotations. Annotations are used in MASS to write programs which are event-oriented. Events such as an agent or place being created, an agent leaving a place and agent arriving to a place can trigger a method to be called on either the agent or place that is a part of the event. Decorating a method on the place or agent with @OnArrival, @OnDeparture or @OnCreating, will trigger that method for the corresponding event.

For shortest path three other programs had already been written. One was an agent discrete migration implementation, the next two included asynchronous migration. The difference came from the third implementation being done using doAll method instead of the common callAll and manageAll. The doAll call specified two different methods to be called on the agents. For annotation the doWhile method was called instead of a using an implicit while loop or callAll and manageAll.the doWhile loop ended when there were no more agents left.

Source code for shortest path can be found at:
https://bitbucket.org/mass_application_developers/mass_java_appl/src/bac9514cf86d469f7892a86879f039afea9b0493/?at=jacoltzi%2Fannotations_benchmarks

Results

From the programs mention above all of them have been measured for both performance and programmability.

Programmability

The programs that I created are measure by programmability by the amount of lines of code(LOC), classes and/or boiler plate code. Boiler Plate code refers to the code that is needed to paralyze the program but has little to do with the actual logic of the program.

JCilk

JCilk in terms of programmability is great. JCilk is not a framework and does not use too much boiler plate code. JCilk uses keywords, such as spawn and sync to fork and join threads. Figure Table 1.1 shows JCilks LOC and boiler plate for both triangle counting and CPP. In both boiler plate code is fairly low compared to other parallel computing frameworks.

Table 1.1

Programs	LOC	Boiler Plate
Triangle counting	~140	3
CPP	~200	3

MASS GraphPlaces

For MASS' KD-tree range search program has already been evaluated for programmability Table 2.1 includes results.

Table 2.1

Frameworks	LOC	Boiler plate	# of Classes
MASS	~ 490	4	6
Spark	~ 350	6	3
MapReduce	~ 450	25	6

MASS Annotations

Table 3.1 shows how an annotated approach to shortest path can lead to less code and less boiler plate than any other approach within MASS. The shortest Path code can be seen in Figure 1.1 for MASS discrete-event and Figure 1.2 for MASS annotated. The two code snippets include only the shortest path search. In these small code snippets, the difference in code can be seen.

Table 3.1

Strategy	LOC	Boilerplate
MASS Discrete Migration	~107	9
MASS Async	~104	10
MASS DoAll	~102	7
MASS Annotated	~98	7

```
case 1:
// discrete-event migration
int nextEvent = -1;
for ( int time = 0; ; time = nextEvent ) {
Object currTime = ( Object )(new Integer( time ) );
gravity.callAll( Gravity.departure_, currTime );
gravity.manageAll( );
Object[] args4callAll = new Object[gravity.nAgents( )];
for ( int i = 0; i < args4callAll.length; i++ )
    args4callAll[i] = currTime;
Object[] allEvents =
( Object[] )gravity.callAll( Gravity.onArrival_, args4callAll );
if ( ( nextEvent = minInt( allEvents ) ) == -1 ) {
    System.out.println( "the shortest path = " + time );
    break; // Simulation finished
}
gravity.manageAll( );
}
break;
```

Figure 1.1: Discrete-event migration

```
case 4:
    // Annotated
    gravity.doWhile( () -> gravity.hasAgents() );
    Object[] args4doWhile = new Object[nNodes];
    for ( int i= 0; i < nNodes; i++ )
    args4doWhile[i] = new Integer( 1 ); // just a dummy
    Object[] allCosts_doWhile
    = ( Object[] )network.callAll( Node.cost_, args4doWhile );
    System.out.println( "the shortest path = " +
allCosts_doWhile[dst] );

}
```

Figure 1.2: Annotated

Performance

Performance is measured through running the benchmark programs and recording the execution time.

JCilk

For JCilk I Gathered data on the triangle counting program where the graph size was 3000 vertices. For a graph this big, the program would often break. I was only able to gather data for when all 4 processors were needed. Even when all 4 processors were used JCilk's performance was slow. Compared to MASS, Spark, and MapReduce as seen in figure 3.1, JCilk did worse than all other competitors.

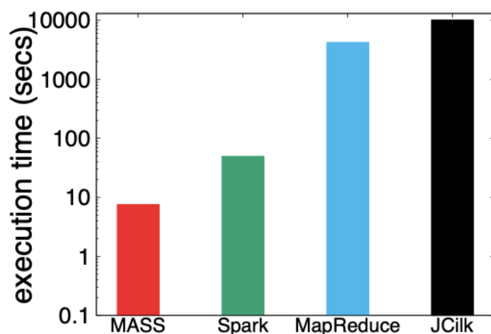

Figure 2.1 Triangle Counting 3000 vertices

Credit: OAC Core: RUI: Agent-Based Scientific Data Analysis(NSF2020)

For CPP program 32768 point were evaluated to find the closest pair of points. Figure 3.2 shows results of JCilk for 1-4 processors. Unlike triangle counting CPP execution times were low compared to those of other frameworks where execution time can range around 25 seconds.

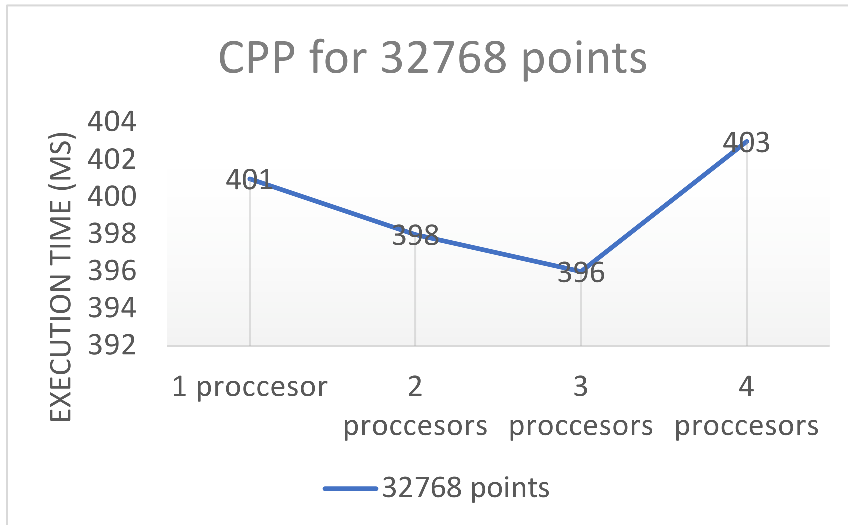

Figure 3.1

MASS GraphPlaces

Although KD-Tree range search benchmark program showed better performance than what was first recorded, this time the evaluation showed a decrease in performance when there were more computing nodes added. As seen in figure 4.1 the increase of computing nodes did not decrease the time that the program would take to execute. As more points were added to the KD-tree the more the performance decrease can be seen. The reason for this is speculated to be due to a GraphPlaces dependency on Hazelcast.

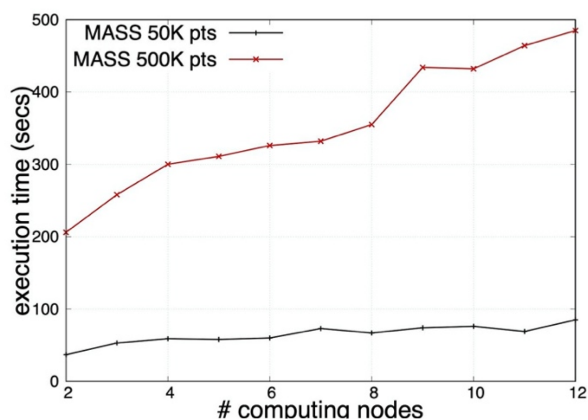

Figure 4.1: MASS KDTree results with different amount of point
Credit: Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory, BigGraphs 2020

MASS Annotated

In figure 5.1 a performance evaluation was taken between MASS Annotated program and MASS DoAll program. What is seen is that not only are annotations great to reduce code but also competes in performance against other versions of MASS.

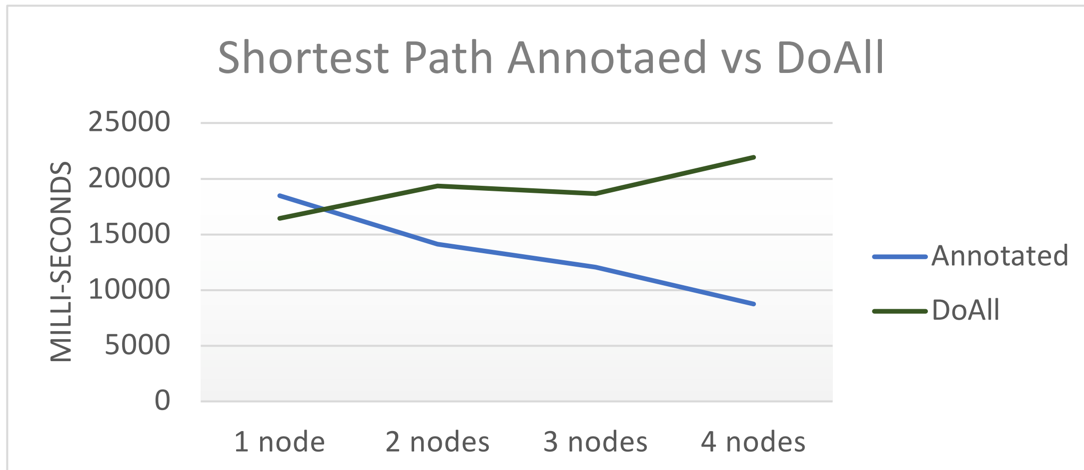

Figure 5.1

Further Analysis

JCilk can be concise if it is to write small programs. The amount of additional code needed to parallelize a program is not much. However, there is not much other than making multi-threading easier. Performance wise, JCilk is outperformed by all other parallel computing frame works as seen in Figure 2.1. JCilk also suffers from over-head when parallelizing a program. JCilk needs to create an object each time a spawn keyword is used on a method. This means that recursive solution create a lot of objects. This is most likely the reason for the poor performance. In the case of CPP the execution time can be credited to the algorithm, which is an O(NLog(N)) solution.

MASS can be both concise and can have execution performance that can compete with other frameworks. Although MASS GraphPlaces suffers from some overhead, this does not impact MASS as a whole. Only MASS GraphPlaces uses Hazelcast. MASS Annotated can lead to both concise code and completive execution performance. MASS annotations are also natural within a MASS program. Most MASS programs are guided by the events of the environment, using annotation makes creating the programs easy.

Conclusion

MASS' new components GraphPlaces and Annotations are a great addition to the MASS framework. GraphPlaces can improve by some modification on the Hazelcast dependency, but other than that GaphPlaces can be great for reducing code in MASS when dealing with graphs. MASS annotations are easy to use and can naturally be added to a MASS program due to MASS agent paradigm.

Future work

I have officially finished all work that was asked for me to do for MASS. I however am interested in creating the grasshopper optimization algorithm for finding K clusters. This work will be done using MASS Annotated. Other than this work I will just be available to answer questions on my work in order to help complete research papers that are being written using my work.

Citations

Fukuda, Munehiro, et al. “Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory.” *BigGraphs2020*, IEEE BogGraphs, docs.google.com/presentation/d/1WByOWHR3pTs8RAGtqpNqn_O1JINegv3U/edit#slide=id.p2.

Gilroy, J. (202Gilroy, J. (2020). *DYNAMIC GRAPH CONSTRUCTION AND MAINTENANCE* [Scholarly project]. In *Depts.washington*. Retrieved 2020, from http://depts.washington.edu/dslab/MASS/reports/JustinGilroy_whitepaper.pdf