Using JetBench to Evaluate the Efficiency of Multiprocessor Support for Parallel Processing

HaiTao Mei
University of York, UK
hm857@york.ac.uk

Andy Wellings
University of York, UK
Andy.Wellings@york.ac.uk

ABSTRACT

JetBench is an Open Source OpenMP-based multicore benchmark application that was created to be used to analyse the real time performance of a multicore platform. The application is designed to be platform independent by avoiding target specific libraries and hardware counters and timers. JetBench uses jet engine parameters and thermodynamic equations presented in the NASA's EngineSim program, and emulates the calculation of a jet engine's performance. As a benchmark, it is, therefore, an *application benchmark* rather than a *synthetic benchmark*. This paper reports on the use of JetBench to compare the efficiency of several concurrency models when implemented on a shared memory multicore platform. We compare Ada, C (used in conjunction with Open Multi-Processing (OMP)), Java 8 using the Thread class, Java 8 using Thread Pools, Java using OpenMP, the Jamaica implementation of the Real-Time Specification for Java (RTSJ) with Real-time Threads (compiled), and C#. Our results show that Ada and C with OMP general perform slightly better than compiled RTSJ, and that all three outperform the Java-based languages and C#.

1. INTRODUCTION

Benchmarks are computer programs that are designed to simulate a particular type of workload on a component or system. Their goal is to evaluate the performance of the component (or system). The component (or system) may be hardware based (for example, an instruction set of a processor) or software based (for example, a compiler or an operating system). Benchmarks are usually classified into two types: application benchmarks and synthetic benchmarks. Application benchmarks are "real-world" programs whereas synthetic benchmarks are contrived program designed to generate a particular workload or pattern of usage. Arguably application benchmarks give a much better indication of real-world performance on a given system than synthetic benchmarks. For real-time benchmarks, however, it is often difficult to acquire good application benchmarks that test real-time performance.

JTRES'14, October 13–14 2014, Niagara Falls, NY, USA

http://dx.doi.org/10.1145/2661020.2661029.

The JetBench [16] benchmark is an application benchmark written in C (and used in conjunction with OpenMP [9]) that contains a set of real-time jet engine thermodynamic calculations. Its goal is to evaluate the performance of shared memory multiprocessor architectures. Here we use JetBench in a slightly different context. Our goal is to provide implementations of JetBench in various languages in order to compare how efficiently each language's concurrency model can be implemented. We consider the following languages, where necessary used in conjunction with middleware:

- Ada,
- C used in conjunction with OMP,
- Java 8 using the Thread class,
- Java 8 using Thread Pools,
- Java using Open MP,
- Jamaica RTSJ with Real-time Threads (compiled) and
- C#.

The languages have been chosen as they all support the shared memory model of computation. We use a mixture of languages that have been used in the real-time embedded systems domain and those that have not.

The paper is structured as follows. In Section 2 we describe the details of JetBench. Although it is an application benchmark, the benchmark is contrived for multiprocessor execution. Essentially, the set of calculations are independent from one and other and executed in parallel to obtain an overall speedup. We show that the internal structure of the benchmark is not coherent as it contains many needless accesses to shared variables. We propose a restructuring to make the code more easy to understand and more coherent from a multi-threaded viewpoint. In Section 3, we present the code for the RTSJ version of JetBench as an example implementation. Then, in Sections 4 and 5, we show the results of our language comparisons. In Section 6 we briefly review related work. Conclusions are given in Section 7.

2. JETBENCH

JetBench [16] is an application benchmark written in C that is used in conjunction with OpenMP [9] for real time jet engine thermodynamic calculations. The thermodynamic calculations are inspired by a sequential application named NASA EngineSim [14]. JetBench proposes a parallel way to perform the calculations based on different input data. JetBench employs OpenMP to carry out the parallel computation instead of using target specific libraries or hardware

counters. This approach allows JetBench to be portable across various architectures and operating systems.

2.1 Goals and structure

JetBench has two main goals. The first is to propose a benchmark that can execute in parallel on multiprocessor platforms. The second goal is to provide a tool to analyze real time performance of a real-time operating system, including thread scheduling, execution efficiency and memory management capabilities etc.

JetBench's execution can be divided into 3 steps:

1. initialization,
2. create threads with the help of OpenMP and carry out the calculation in parallel, and
3. print out results.

In the first step, JetBench initializes parameters and opens a file that contains all the input data needed to be processed in the second step. The input data consists of the values of three sensors: altitude, air speed, and throttle. A fourth input value gives a contrived deadline that represents a time constraint on the calculation of the engine's performance figures. In our experiments, the deadline has been fixed at a value of 0.05 seconds. This has been chosen to be approximately 2-3 times the value of the execution time of the raw C code calculations. Hence, the required utilization is less than 50%.

In the second step, JetBench creates multiple threads (one for each processing core in the system) and starts these threads. All the threads perform the same operations. Each thread first calculates π using a loop, measuring the time taken. After this, each thread reads a set of input data from the file opened in the first step. Then, each thread processes its input data, carries out thermodynamic, geometry and engine performance calculations, and prints out the performance data. The times taken to perform the calculations are recorded. When all the data in the file is processed, the threads terminate. After all the threads have terminated, the benchmark enters into the last step of its execution. During the last step, the results are collected from the second step and are printed. An overview of the execution structure is shown in Figure 1.

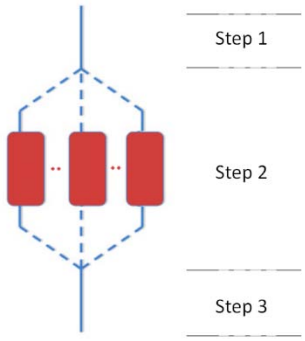

Figure 1: The overview of benchmark structure

Although the benchmark works and produces results, it is hard to understand its current structure. This is because the data is not accessed in a coherent manner for the following reasons

- There are needless accesses to shared variables and never-used variables. The set of input data use in the calculations is independent of each other. However, operations in all the running threads share the same set of variables. For example, all threads use shared variables to store input data which is read from file. As the threads run in parallel, these variables are overwritten and used in a random order during the thermodynamic, geometry, and performance calculations. Massive accessing to shared variables (which should not be shared) in OMP's #pragma parallel directives makes the code hard to read and understand. The functional results are, consequently, incorrect. Although this is not important for the benchmark to produce useful performance results, it nevertheless adds to the confusion.

- Race conditions – JetBench reads input data from file in parallel without any mutual exclusion controls. Furthermore, the measurement of execution times in each thread also use shared variables and are not protected from race conditions.

- There is misleading output information. Firstly the response time of each thread is presented as execution time. This is only the case when the threads are not preempted by any operating system activity on the processor. Secondly, in order to investigate overall speedup obtained in parallel execution, the total response time should be the benchmark's response time rather than the sum of threads' response times, as this includes the overheads of thread creation and termination.

2.2 The revised structure of JetBench

The aims of restructuring is to make the code easier to understand and more coherent in its multi-threading. In the new structure, unnecessary shared variables are moved to local variables, and race conditions are eliminated through mutually exclusive methods. Additionally, the main code of the benchmark is restructured to be more readable. Finally, more meaningful results are output.

In the first step of restructuring, all unused variables are removed along with duplicate operations. The parameters that impact the benchmark's behavior (for example, the number of threads) are moved into a configuration class. All the needlessly shared variables are moved to thread local variables, and the truly shared variables are accessed via critical sections.

In the second step, the input data is read into memory before calculation starts and stored in an input array. During calculation, each thread reads input data at specific positions of the input array. After calculation, the results will be written into an output array. This ensures that input and output times are not included in the response times

In the last step, the benchmark's response time, which is measured by the new method, will be printed out, along with an indication of the number of deadlines that have been missed.

The overall revised structure is illustrated in Figure 2 and is thus

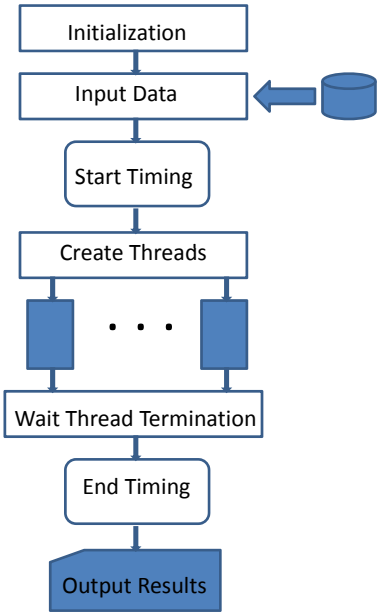

Figure 2: The revised benchmark structure

- Initialize the parameters according to the configuration data and initialize the input data array,
- create threads to carry out parallel processing of the in-memory input data, and
- print out the result after all threads finish their jobs.

3. JETBENCH IN THE REAL-TIME SPECIFICATION FOR JAVA

This section shows how the JetBench program has been instantiated for the Real-Time Specification of Java.

In order to remove any possible garbage collection delays, the main program creates a real-time thread that then creates the worker threads as no-heap real-time threads of default priority. The overall response time for the benchmark is calculated within this first thread to exclude file input and output but to include worker thread creation, processing and termination. This is illustrated with the abridged code given below.

```
1  class starRTThread extends RealtimeThread {
2    private static double BenchmarkStartTime,
3                          BenchmarEndTime;
4    //arrays, store input and output data
5    public static double[][] inputArray;
6    public static double[][] outputArray;
7    public static int LineCount=0;
8
9    public void run() {
10     final NoHeapRealtimeThread threads[]=new
11        NoHeapRealtimeThread[NUM_THREADS];
12     ImmortalMemory.instance().enter(
13      new Runnable() {
14       public void run() {
15         InitializeArray(); // read in data
16         BenchmarkStartTime=System.nanoTime();
17         /**   create threads */
18         for(int i=0;i<NUM_THREADS;i++){
19           threads[i]=new WorkingRTThread(i);
20           threads[i].start();
21         }
22         //join thread and print final result
23         for(int i=0;i<NUM_THREADS;i++){
24           try {
25             threads[i].join();
26           } catch (InterruptedException e) {
27             e.printStackTrace(); }
28         }
29         BenchmarEndTime=System.nanoTime();
30         printResult();
31       }
32     });
33   }
34
35   public static void InitializeArray(){...}
36   public static void printResult(){...}
37 }
```

The main calculations are performed in the no-heap real-time worker threads. Their structure is given below.

```
1  public class WorkingRTThread
2        extends NoHeapRealtimeThread {
3
4    // various global constant variables, e.g.
5    public static int engine =
6           configuration_data.engine;
7    private static final double g0 = 32.2;
8    private static final double gama = 1.4;
9
10   //local variables for calculating, e.g.
11   double altd,u0d;
12   double throtl;
13   double[] trat = new double[20];
14   double[] tt = new double[20];
15   double[] prat = new double[20];
16   double[] pt = new double[20];
17   double[] eta = new double[20];
18   double[] gam = new double[20];
19   double[] cp = new double[20];
20
21   public WorkingRTThread(int id){
22     super(null, ImmortalMemory.instance());
23     this.id=id;
24     InitializeParam();
25   }
26
27   @Override
28   public void run() {
29
30     // variables for calculating time related,
31     // e.g.deadline etc for each thread
32     double StartTime = 0, EndTime = 0,
33            ExecTime = 0;
34     double used = 0;
35
36     // variables for pi calculation
37     final long num_steps = 1000000;
38     double step = 1.0 / (double) num_steps;
39     int i = 0;
40     double x, pi, sum;
41     // variables for input data
42     double a,b,c,d;
```

```
43
44      while (moreData) {
45        /** Pi calculation*/
46        sum=0;
47        StartPiTime= System.nanoTime();
48        for (i = 0; i < num_steps; i++) {
49          x = (i + 0.5) * step;
50          sum += 4.0 / (1.0 + x * x);
51        }
52        pi = sum * step;
53
54        // read input data
55        // Speed Altitude and Throttle
56        int index=CurrentPoint-1;
57        a=starRTThread.inputArray[index][0];
58        b=starRTThread.inputArray[index][1];
59        c=starRTThread.inputArray[index][2];
60        d=starRTThread.inputArray[index][3];
61
62        /** START CALCULATIONS **/
63        deduceInputs();
64        getThermo();
65        getGeo();
66        calcPerf();
67        EndTime = System.nanoTime();
68
69        ExecTime = (EndTime - StartTime)/
70                   1000000000;
71            //convert to seconds
72        usedTime = ExecTime - d ;
73          /** save RESULTS **/
74      }//end of while
75    }
76
77    private void InitializeParam() {..}
78    public void deduceInputs() {...}
79    public void getThermo() {...}
80    public void calcPerf() {...}
81    public void getGeo() { }
82  }
```

The total number of lines of code in the RTSJ version of the benchmark is 712, and the mean McCabe Cyclomatic Complexity of the class methods is 3.118 and the maximum is 13.

4. RESULTS

The restructured JetBench benchmark has been coded in each of the languages under test. The programs have been tested and each produce the same functional results. In this section we present the results of the experiments that determine the performance of the benchmark in each of the languages. We consider the following:

1. The mean total number of deadlines missed during an experiment in each language – Although the deadline is a contrived value, it is a useful measure to show the difference is both the quality of sequential code generated and the efficacy of the scheduling. Here the deadline is measured from the moment that a task/thread starts a calculation rather than some notional absolute start time (that is, we treat the tasks/threads as sporadic tasks that are released when work is allocated to them).

2. The response time of the benchmark – This is the time taken to complete all of the required calculation minus the time taken to start the program and read in the sensor data from the input file.

3. Speedup – This is the relative speedup of the benchmark as the number of allocated cores increases. Each language's value is normalized with its single processor value being set to 1. It shows the efficiency of the languages support for multiprocessor execution.

The experiments were performed on an Intel Core i7-2630QM Processor, 4 cores with hyper-threading 2 GHz processor (giving 8 processing cores) running Linux version 3.13.0-29-generic X86_64. The logical processors were selected using the Linux "taskset" shell command.

An experiment consists of running the benchmark in one of the languages 30 times. The number of cores is then varied and the experiment is run again. Hence the benchmark was executed 30 time in each language on 1, 2, 4, 6 and 8 cores. Hence in total there are 1015 data points. The following should be noted.

- For the one processing core experiments, processor 1 was used, i.e. "taskset -c 1 ".
- For the two processing core experiments, processors 1,3 were used, i.e. "taskset -c 1,3 ". This ensured that the two logical processors were on separate physical cores
- For the four processing core experiments, processors 1,3,5,7 were used, i.e. "taskset -c 1,3,5,7 ". This ensured that the four logical processors were on physical separate cores.
- For the six and eight processing core experiments, the other processors were utilized. As these will share hardware components with the other logical processors, this may cause interference and reduce the potential for further speedup.

For the languages we used the following implementations.

- For Ada, we use the AdaCore GNAT GPL 2014 compiler version 4.6 and compiled with the -O2 optimization flag.
- For C used in conjunction with OMP, we use gcc 4.8.2 (compiled with the -O2 optimization flag) and OpenMP 3.1
- For Java 8 using the Thread class, we use Java version 1.8.0_05 (build 1.8.0_05-b13).
- For Java 8 using Thread Pools, we use Java version 1.8.0_05 (build 1.8.0_05-b13).
- For Java using Open MP, we use Java version 1.8.0_05 with jomp1.0b.
- For RTSJ with Real-time Threads (compiled with the -O2 optimization flag), we use the aicas Jamaica Builder version 6.2 Release 4 (build 8016).
- For C#, we use the Mono JIT compiler version 3.2.8.

Figure 3 shows the mean number of deadline misses that occurred for each experiment. As can be seen, there are a few deadlines missed with Java and many with Java JOMP.

Figure 4 shows the response times of the experiments. The error margins show the variation at the 95% level. Finally, Figure 5 shows how each language performs in terms of the benchmarks speed-up when more cores are added.

Mean Number of Deadline Misses per Experiment							
CORES	Ada	C+OpenMP	Java	Java ForkJoin	Java JOMP	RTSJ	C#
1 Core	0	0	0	0	1	0	0
2 Cores	0	0	0	0	1	0	0
4 Cores	0	0	0	0	6	0	0
6 Cores	0	0	1	0	9	0	0
8 Cores	0	0	2	1	13	0	0

Figure 3: Total number of deadline misses in each language

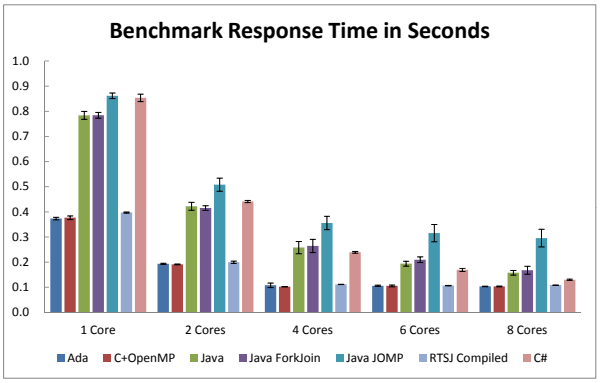

Figure 4: Benchmark response times

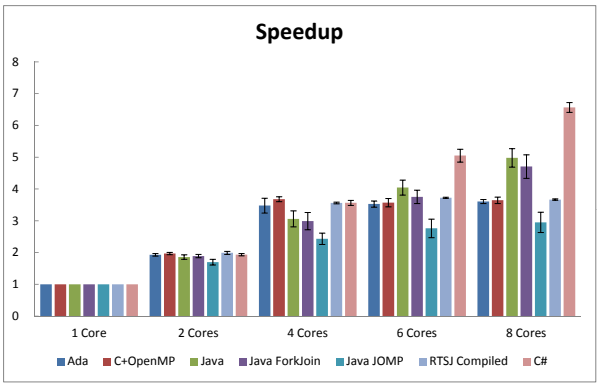

Figure 5: Benchmark speed-up

4.1 Detailed analysis of results

Analysis of variance (ANOVA) is a general statistical technique for separating the total variation in a set of measurements into the variation due to measurement noise and the variation due to real differences among the alternatives being compared [12]. The one-way ANOVA tests the null hypothesis that samples in two or more groups are drawn from populations with the same mean values.

The two-way analysis of variance is an extension of the one-way ANOVA that examines the influence of two different categorical independent variables on one dependent variable. It determines both the main effect of contributions of each independent variable and if there is an interaction effect between them [21]. The analysis computes an F value which describes this relationship. It is an appropriate technique for the analysis of the measurements of execution times [12].

The goals of applying this analysis to our results is to prove that both programming languages and the number of core has an impact on the benchmark's response times, and also that there is significant interaction between them, i.e. different programming languages have different efficiency impacts in multiprocessor parallel processing.

The null hypothesis is made that both factors (programming languages and number of cores) have no effect on the benchmark's response times, i.e. its efficiency. The number of times each experiment is run is 30 and the alpha value is 0.05 (a statistical significance level of 95%). After carrying out two way ANOVA analysis (using MATLAB), the results are presented in Table 1 – where SS is the Sum of Squares due to each source, df is the degrees of freedom associated with each source, MS is the Mean Squares (MS), which is the ratio SS/df, and F is the ratio of the variance calculated among the means to the variance within the samples. The p value is the computed probability that the null hypothesis holds. If any p value is near zero, this casts doubt on the associated null hypothesis.

Source	SS	df	MS	F	P-value
Cores	33.46	4	8.37	41650.41	< 0.01
Language	12.75	6	2.12	10576.99	< 0.01
Interaction	4.40	24	0.18	914.45	< 0.01
Total	50.82	1049			

Table 1: The two way ANOVA table

Matlab indicates that the probabilities are very close to 0. To give an indication of what the values of F would be for the 0.01 probability that the null hypothesis holds, the following F values are taken from the Tables of F Probability Distribution for given levels of statistically significance (e.g. [10]).

- F(cores)(4,1015) = 3.480
- F(languages)(6,1015) = 2.956
- F(interaction)(24,1015) = 1.791

As can been sen, F_{Core} and $F_{Language}$ are much larger than the maximum value in the F distribution tables indicates that hypotheses is rejected, i.e. not only programming languages, but also core/thread number have effect on efficiency. $F_{Interaction}$ is much larger than the maximum value in the F distribution table, indicating that there is interaction effect between them, i.e. programming languages have different efficiency in multiprocessor parallel processing.

4.2 Tukey HSD (Honest Significant Difference) Analysis

The ANOVA analysis allows us to have confidence that (with our implementations of the benchmark) the response times are significantly different depending on the language used. This may be because one language has a very poor implementation. In order to obtain a more detailed analysis we use a Tukey HSD [20]. This is used in conjunction with an ANOVA to find means where there are significant differences between the languages. Tukey's test compares the means of every language to the means of every other language and identifies any difference between two means that is greater than the expected standard error.

In the following, we use A >> B to indicate that A and B have means that significantly different, and A is larger than B. We use C > D to indicate that C and D don't have means that are significantly different, and C is larger than D. All the figures presented in this section come from HSD analysis performed by Matlab.

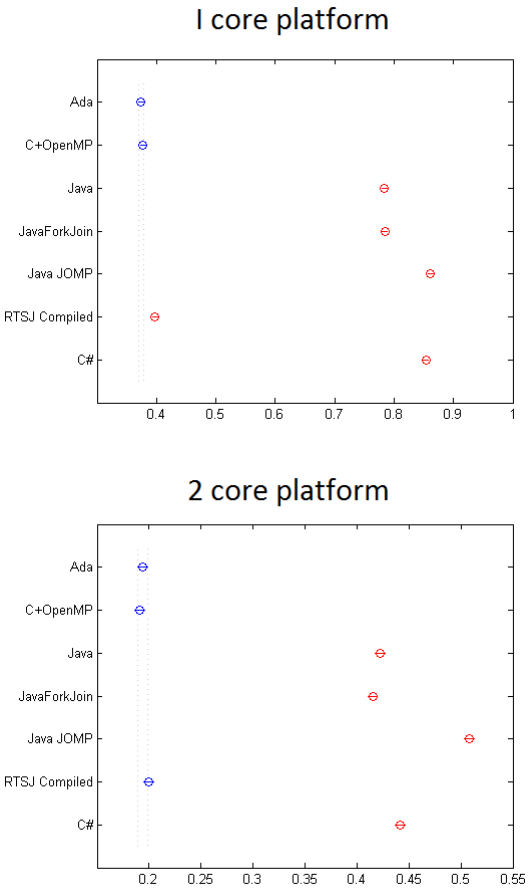

Figure 6: HSD analysis for response time means: 1 and 2 cores

Tukey's HSD Analysis of Response Time

Figure 6 shows the HSD analysis of the response times for 1 and 2 cores. It shows that, Ada and C+OpenMP do not have means that are significantly different from each other, but they are significantly different from the other languages.

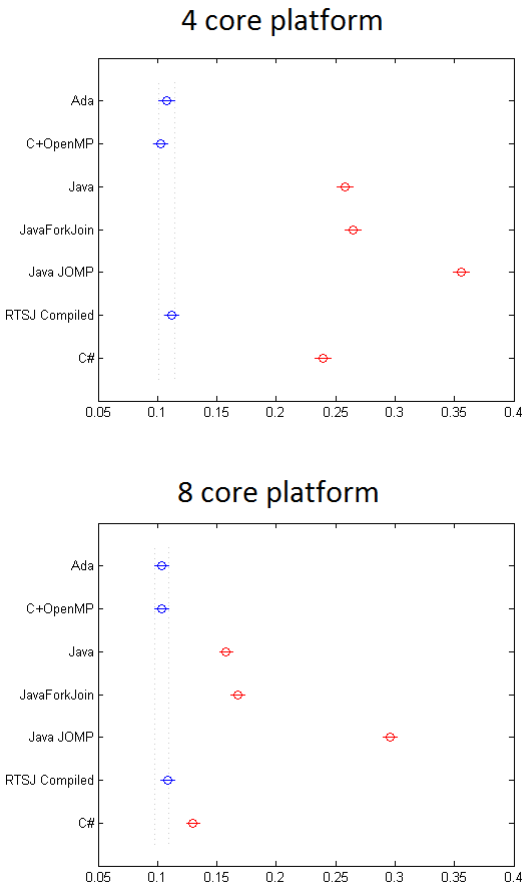

Figure 7: HSD analysis for response time means: 4 and 8 cores

The difference between Java and Java with ForkJoin is also not significant. Hence:

$$JavaJOMP >> C\# >> JavaForkJoin > Java >> RTSJ >> C + OpenMP > Ada$$

For two cores, the relationship is similar with some minor changes. In particular, RTSJ closes the gap on C+OpenMP and Ada.

$$JavaJOMP >> C\# >> Java > JavaForkJoin >> RTSJ > Ada > C + OpenMP$$

Figure 7 also shows similar diagrams for 4 and 8 cores. We see here that C# seems to improve its performance when compared to the Java-based languages.

Tukey's HSD Analysis of Speed-Up

Figure 8 shows the results of the speed-up analysis for two and four cores. It shows that the RTSJ implementations has the greatest speedup.

For two cores:

$$RTSJ > C + OpenMP > C\# > Ada > JavaForkJoin > Java >> JavaJOMP$$

For 4 cores

$$C + OpenMP > C\# > RTSJ > Ada >> Java > JavaForkJoin >> JavaJOMP$$

But note that C+OpenMP >> Ada

For 6 cores, we have

$$C\# >> Java >> JavaForkJoin > RTSJ >> C + OpenMP > Ada >> JavaJOMP$$

And for 8:

$$C\# >> Java >> JavaForkJoin >> RTSJ > C + OpenMP > Ada >> JavaJOMP$$

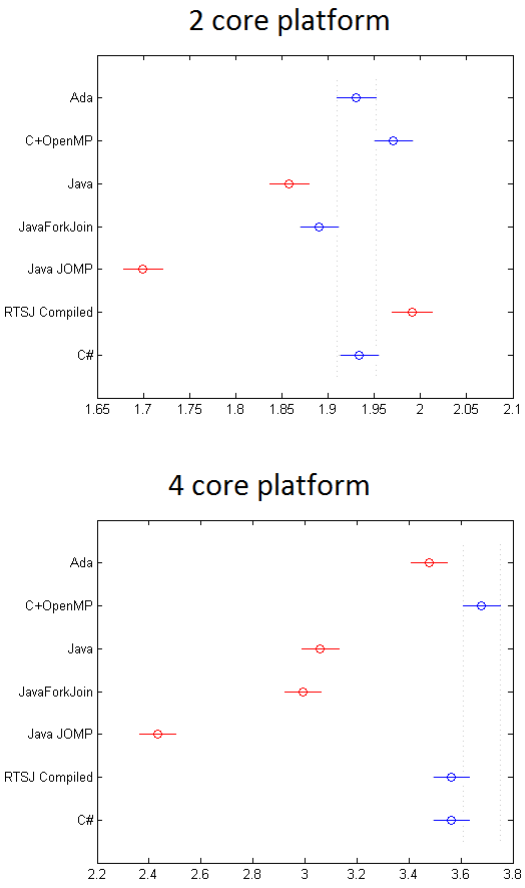

Figure 8: HSD analysis of speed-up means: 2 and 4 cores

Figure 9 shows similar diagrams for 4 and 8 cores. Here we see that C# and Java now show greater speedup.

4.3 Discussion

It is always difficult to draw general conclusions from a limited set of experiments on a single benchmark. That is why we have used well-known statistical analysis techniques to show the significance of our results. Of course, much depends on us not introducing any inefficiencies when coding the benchmarks in our languages. The core parts of the benchmarks (the calculations) are essentially the same for

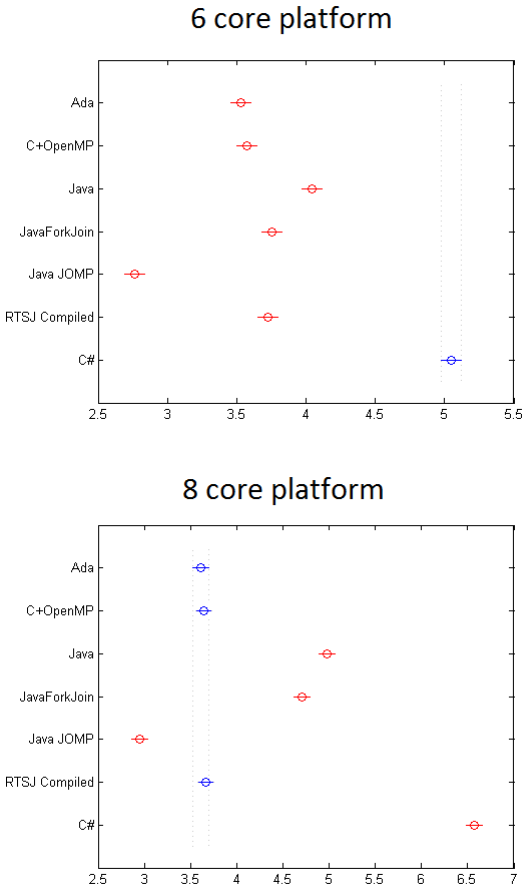

Figure 9: HSD analysis of speed-up means: 6 and 8 cores

the "C-based" languages as they all accept C syntax. Only the Ada syntax is different in this respect.

The implementation we have used for Java and C# are JIT-based, and as a consequence we would expect their overall performances to be less that the fully-compiled implementations of Ada, RTSJ and C. This is confirmed by the results of benchmark's response-times, which shows that C with OMP and Ada have the best performance followed by the RTSJ. The similarity between the Ada and C results is probably accounted for by the fact that they both use the gcc backends and optimizers. Indeed, repeating the experiments without the optimization degrades these languages' performances significantly. We also repeated the experiments with the JIT-based implementations allowing a warm-up phase before measurements were started. This resulted in some improvement in response times (and speed-ups) but they were still significantly poorer than the compiled languages.

The speed-up experiments remove the advantages of full compilation by normalizing the times to the single processor times. Our, perhaps naive, expectation is that if the language maps its implementation of tasks/threads to Linux threads then most of the speed-up should be similar, as Linux is handling the scheduling. Furthermore, given that our test machine supports hyperthreading, we expect the speedup to decrease when we move from four to six and

eight cores. What we see is that the languages that are more efficiently implemented have less speed-up when going to six and eight cores than those languages that have less efficient implementations. Perhaps this is the impact of hyperthreading. The efficiently implemented languages make fuller use of the architecture components shared between the physical cores (pipeline etc) so that hyperthreading delivers less significant performance gains.

The exception is Java with JOMP, which performs badly overall.

5. EXPERIMENTS WITH SIMICS

Simics (see www.windriver.com/simics) is a full-system simulator. It allows virtual platforms to be created that run the same binary software that would run on the actual physical hardware. We have replicated our experiments for 1 to 4 cores on the simulator and have achieved very similar results. The result profiles are the same although the detailed measurements are slightly different.

We have used Simics to simulate a 128 core (the maximum supported by Simics) Linux system. The simulated hardware supported by Simics is a multicore system based on the Pentium 4e without hyperthreading (our version of Simics does not support the most recent Intel processors). We also increased the number of sensor readings in the data input file to reflect the increased number of processors. Here we report on an experiment involving just Ada, C (with OpenMP) and compiled RTSJ. In order to focus on the support for parallel processing, we have extracted out the main computational components and written them in C. Both the Ada and the C (with OpenMP) now use this common code. The RTSJ version uses this code as well – as it is also valid Java[1].

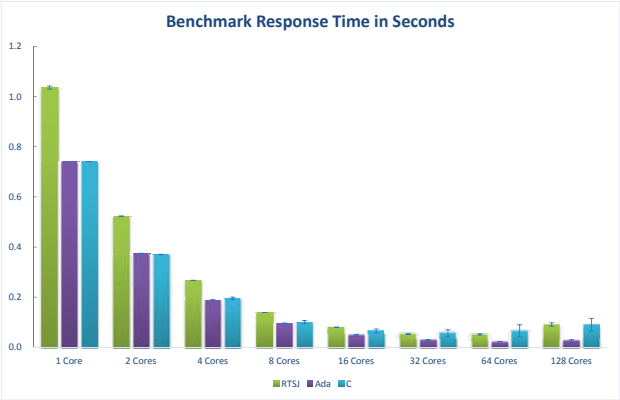

Figure 10: Simics benchmark response times

The results show that Ada and C have similar performance up to 8 cores, beyond 32 cores, the C version performance begins to degrade. For all languages, the performance with 128 cores is disappointing. The reason for this is that 128 tasks (threads) are created and this introduces added overheads. More importantly, as the size of the input is only 128 lines, each thread only processes one set of sensor readings. When we increased the number of sensor reading the performance continued to improve at 128 cores.

[1]We did try to use JNI to allow access to the common C code but the cost of the JNI calls significantly degraded the overall performance.

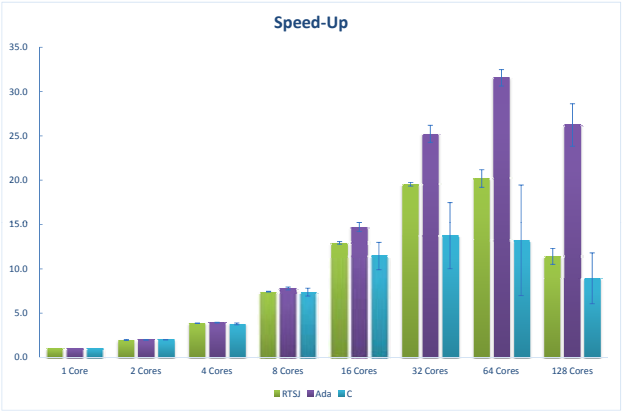

Figure 11: Simics benchmark speed-up

6. RELATED WORK

In their most general sense, benchmarks are designed to simulate a particular type of workload on a component or system. Their goal is to evaluate the performance of the component (or system). Benchmarks are broadly classified into two types: application benchmarks and synthetic benchmarks, although other groupings can be specified – see http://en.wikipedia.org/wiki/Benchmark_(computing).

Traditionally, benchmarking is one of the most important methods for evaluating the performance of processor designs. For example, Whetstone [19] and Dhrystone [18] are both synthetic benchmarks whose goal is to evaluate the performance of a CPU's internal workings. More recently, their use has been extended into multiprocessor system design. For example, the NAS Parallel Benchmarks are a widely used set of programs that were designed to evaluate the performance of supercomputers [1]. These benchmarks consist of five "parallel kernel" benchmarks and three "simulated application" benchmarks. A kernel benchmark being a core piece of code normally abstracted from an actual program, for example the Livermore loops [13]. Whilst the focus of these benchmarks is on performance in terms of speed, other benchmarks address energy efficiency, e.g., JouleSort [17].

PARSEC [5, 6, 4] is an application benchmark suite for chip-multiprocessors that focuses on emerging applications, including financial analysis, computer vision, data storage, animation, and data mining. While the SPLASH-2 [22, 3] benchmark suite is mainly aimed at the High-Performance Computing domain.

In the real-time community, there are two notable application benchmark. The PapaBench benchmark [15] is written in C and provides a free implementation of an autopilot for unmanned aerial vehicles. A Java version (jPapaBench) is also available for plain Java, the Real-Time Specification for Java, and Safety-Critical Java (see https://code.google.com/p/jpapabench/). The CDx (Collision Detector) benchmark suite [11] is a Java open source application benchmark suite that targets different hard and soft real-time virtual machines.

In contrast to the above work, the focus of this paper is on evaluating the efficiency of a language's concurrency model. There has been some work on evaluating languages performance [8, 2, 7]. Notably, Berlin et al [2] have evaluated the impact of programming language features on the performance of parallel applications on cluster architectures. They consider Pthreads, Open MP, MPI, UPC and Global Arrays and evaluate portability and programmability along with performance. In contrast to our work, their focus is on comparing the shared memory, message passing and a hybrid approach. Their conclusion is that threads-based paradigms provide best performance on SMPs while paradigms with explicit communication achieve best performance in clusters.

7. CONCLUSIONS

It is now taken for granted that real-time and embedded platforms will be multi-core and that programs will need to be multithreaded if they wish to exploit the extra performance available to them. However, there are a plethora of programming language that can be used, all of which now allow programs to be executed in parallel. In this paper we have considered a few languages, some of which are important to the embedded and real-time community. We have set out to determine how well each language supports parallel processing. We have chosen to do this with an application benchmark rather than a synthetic benchmark as we believe that application benchmarks give a more realistic guide to the performance improvement that practitioners can expect. Unfortunately, application benchmarks that are appropriate for multiprocessor real-time systems are few and far between. We selected JetBench primarily because it is open source and claims to be for real-time.

On examining the original JetBench program, we found that its structure was very confusing and data was shared even when there was no need to do so. We, therefore decided to restructure the benchmark to make its structure more coherent. The benchmark was then rewritten in the languages under test. The final version of these implementations are freely available so that others can repeat our experiments.

We are wary of making wide-sweeping conclusions based on the limited experiments we have done. Our statistical analysis gives us confidence that our results are statistically significant. Overall we expect that the quality of the gcc backends is the dominating factor that explains why C with OMP and Ada have the best overall performance. By extracting out the main sequential calculations into common C code, we are better able to focus on the multiprocessor support. This showed that the overheads of task/thread creation became more significant as the number of cores increased, and that to continue to increase speedup it is imperative to ensure that each task/thread processes more than one set of sensor readings.

Source Code

The source code is available at https://sourceforge.net/projects/mpbenchmark.

Acknowledgement

We acknowledge Muhammad Yasir Qadri, the original creator of JetBench, for helping us understand the code. We would like to thank Paul Cairns for his help with using ANOVA and Tukey's HSD to analyse the statistical significance of our results. We are also indebted to Wind River Systems for providing us with access to their Simics Full System Simulator.

8. REFERENCES

[1] David. H. Bailey and et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[2] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2004.

[3] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56, Sept 2008.

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. Princeton University, Princeton, NJ, USA, 2011. AAI3445564.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[6] Christian Bienia and Kai Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[7] Bryan Carpenter and et al. Applications of HPJava. *Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, 2958:147–161, 2004.

[8] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 177–193. Springer Berlin Heidelberg, 2004.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[10] I Denov. Statistics online computational resource (socr). http://www.socr.ucla.edu/applets.dir/-f_table.html, accessed June 2014.

[11] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CDx: A family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages

41–50, New York, NY, USA, 2009. ACM.

[12] David J Lilja. *Measuring Computer Performance: A practitioner's guide*. Cambridge University Press, 2000.

[13] Frank H McMahon. The Livermore Fortran Kernels test of the numerical performance range. *Performance Evaluation of Supercomputers*, 4:143–186, 1988.

[14] NASA. Enginesim and rangegames download. http://www.grc.nasa.gov/WWW/k-12/Enginesim, accessed June 2014.

[15] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *Proceedings of the 6th Intl. Workshop on Worst-Case Execution Time Analysis (WCET'06)*, 2006.

[16] MuhammadYasir Qadri, Dorian Matichard, and KlausD. McDonald Maier. JetBench: An open source real-time multiprocessor benchmark. In Christian Müller-Schloer, Wolfgang Karl, and Sami Yehia, editors, *Architecture of Computing Systems - ARCS 2010*, volume 5974 of *Lecture Notes in Computer Science*, pages 211–221. Springer Berlin Heidelberg, 2010.

[17] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: A balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 365–376, New York, NY, USA, 2007. ACM.

[18] Reinhold P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[19] Reinhold P. Weicker. An overview of common benchmarks. *Computer*, 23(12):65–75, December 1990.

[20] Wikipedia. Tukey's range test. http://en.wikipedia.org/wiki/Tukey's_range_test, accessed June 2014.

[21] Wikipedia. Two-way analysis of variance. http://en.wikipedia.org/wiki/Two-way_analysis_of_variance, accessed June 2014.

[22] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.