A COMPUTATIONAL PERFORMANCE INVESTIGATION OF JAVA CONCURRENCY USING MULTI-THREADING ON MULTI-CORED PROCESSORS

Ken A. Hawick
Computer Science, University of Hull, Cottingham Road, Hull HU6 7RX, UK.
Email: k.a.hawick@hull.ac.uk
Tel: +44 01482 346311 Fax: +44 01482 466666

ABSTRACT
Java continues to be an attractive language and development platform for portable simulations. Changes to Java thread management facilities make it possible to utilise Java environments for making use of the multiple cores now widely available on many CPUs. We investigate the performance of multi-threaded Java simulations that make use of 2, 4, 6 and 8-core Intel CPUs and 16 core AMD CPUs running shared resources including memory and buses with one and two processors per board. We report on the effect of the Intel hyper-threading capability. We describe Java multi-threaded approaches to managing software and custom codes for computationally intensive complex systems simulations and discuss the performance implications for future developments and use of Java 7 for simulations software development.

KEY WORDS
Java; concurrency; threading; multi-core; dual CPUs; scalability.

1 Introduction
The Java [10] programming language [41] and its associated runtime environment [11] fill an interesting position in the tradeoff space between portable easy-to-develop software and fast but less elegant concurrent applications software [3, 31]. The relatively recent changes to Java’s multi-threading capabilities and the built in thread futures management libraries appear to have changed Java’s position in this space. In this article we explore some compact simulation-oriented benchmark programs in Java that can use the new Java concurrency features. In particular we experiment with various multi-cored processors that can support the Java concurrency and yield genuine parallel speedups. Figure 1 shows some such parallel speedup measured experimentally on a dual 16-core AMD Bulldozer CPU system.

![Figure 1. Java benchmark performance on dual 16-Core AMD Bulldozer CPU system.](image)

There have been many investigations and attempts to assess Java as a high performance computing platform. The portability of the Java Virtual machine (JVM) remains very attractive and some inefficiencies have often been deemed acceptable given the convenience and ability to soak up cycles on so many platforms.

The new concurrency features in Java 7 appear to make it even easier to use thread level parallelism for managing separate jobs in a user program. This ability to exploit concurrency [32] in user programs [4, 42] and not just at the operating system level [24, 25] is increasingly important as most users will typically have at least two cores in their desktop CPU at the time of writing. Four cores per CPU is common and as the data reported in this present paper will show, six cores in for example on Intel Xeon processor appears to be the current optimal point. We have also experimented with eight core CPUs from Intel and also the 16-cored Bulldozer processors from AMD. Intel [35] and AMD [2] have different approaches to servicing their processors as well as managing potential concurrency and we also consider dual processor systems that share memory and bus structure.

Java has supported multi threading to some extent since its inception [30], although prior to Java 7 the thread management capabilities have attracted some degree of criticism in terms of performance [14, 37] and useability [36]. The JavaGrande community has developed and reported upon a number of Java benchmarks [23] and uses of Java for numerically intensive applications [6] and algorithms [28]. There are other approaches to incorporat-

DOI: 10.2316/P.2014.811-013
Table 1. CPU and relevant properties for the systems used for benchmarking.

<table>
<thead>
<tr>
<th>Model Platform</th>
<th>CPUs</th>
<th>Cores</th>
<th>L2 Cache (MBytes)</th>
<th>CPU Clock (GHz)</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac Mini Intel Xeon 1x2 Core</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2.30</td>
<td>OSX</td>
</tr>
<tr>
<td>Mac Pro Intel Xeon 1x4 Core</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>2.66</td>
<td>OSX</td>
</tr>
<tr>
<td>SMD Blade Intel Xeon 1x8 Core</td>
<td>1</td>
<td>8</td>
<td>20</td>
<td>3.10</td>
<td>Linux</td>
</tr>
<tr>
<td>Mac Pro Intel Xeon 2x6 Core</td>
<td>2</td>
<td>12</td>
<td>12</td>
<td>2.66</td>
<td>OSX</td>
</tr>
<tr>
<td>SMD Blade AMD Bulldozer 1x16 Core</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>2.20</td>
<td>Linux</td>
</tr>
<tr>
<td>SMD Blade AMD Bulldozer 2x16 Core</td>
<td>2</td>
<td>32</td>
<td>2</td>
<td>2.20</td>
<td>Linux</td>
</tr>
</tbody>
</table>

Table 1 lists the relevant core numbers, cache sizes and clock speeds of the conventional CPU devices we experimented with. Our lab operates a mix of such platforms as desktops and servers and as indicated above our aim is to abstractify and reuse parallel constructs and attempts to manage client-server solutions [12, 13] or cluster computing [5] and message passing parallelism [9] is a fairly well established use of Java for attaining computational performance. On the face of it Java’s relatively high-level object-oriented structure [18] and support for generics [1] and other modern language mechanisms [22] make it easier to abstractify and reuse parallel constructs and attempts have been made to incorporate parallel iteration and other high level parallel constructs into Java as a language. This has not met with widespread success however, probably due to the clash between high level programming elegance and attainment of unimpeded parallel performance.

It appears that the new Java 7 concurrency mechanisms do however ameliorate this tradeoff and we present some uses of Java for managing semi-interactive scientific simulations. The applications arena of interest to us is when we need to develop a numerical simulation, that may involve some quite complex algorithms, but which must be quite computationally efficient for cases where many independent experimental runs are necessary. This is often the case for problems that cannot be attacked using analytic methods and which require either exploration of a large parameter space numerically or where there is some stochastic sampling of non trivial trajectory space.

Some classic problems of this sort include simulating models in materials science [17], physics, chemistry, biology or even sociological systems [39] where there are critical phenomena and phase transitions [38] to be identified and localised. Such problems used to require development of optimised custom simulation codes ands the use of supercomputers. It was acceptable to undergo costly code development when the run times were expected to vastly outweigh code development times. More sophisticated applications and problems now often involve more complex and subtle algorithms and faster processing times have reversed this expectation. It is therefore important to have software tools and technologies that allow development of a sophisticated simulation that may well reuse code from an extensive framework, but which can run a set of numerical experiments in near interactive time. Run times that are comparable in magnitude with code development and experiment formulation times are now the expectation.

This present paper is part of an ongoing investigation into software approaches [15] and performance platforms [16, 21, 27] for complex systems simulations and numerical experiments of this nature. We formulate two compact and manageable benchmark problems - one doing simple arithmetic and another generating pseudo-random numbers [7, 34] and test them as Java tasks within a Java 7 thread managed environment and report on the performances attained on various multi-core processor combinations.

This article is organised as follows: We summarise the key properties of various multi cored processors and platforms used for our investigation in Section 2. In Section 3 we present a description and code fragments for the application benchmarks in Java that we study. We describe the details of our experimental timing approach in Section 4 and present some detailed performance data in Section 5. We offer a discussion of the implications for choice of simulation platform that use multi-threaded Java code in Section 6 and some conclusions and areas for further study in Section 7.

2 Multi-Core Processors

Our main interest in choosing particular platforms to study is to cover a broad range of different numbers of cores and also both dual and single CPUs on a particular motherboard that share memory and bus structure. It is expected that the Java 7 JVM should be able to interact with the Operating Systems facilities to directly exploit cores available under such systems without the need for the user to program in this information.

We have chosen a mix of Linux and Apple OSX platforms available to us - all running a Unix variant. These all run the Java OpenJDK 7 language environment and runtime support system [20].
to determine if complex systems simulations can readily be run in a multiple thread environment with Java threading concurrency on a whole range of such platforms.

We have included details of clock speeds as well as core and processor details to support meaningful direct comparisons of the speeds presented below. It is also useful to know any particular cache memory limitations which we also present, although we have chosen the application benchmarks to focus primarily of speed performance rather than memory and communications effects.

3 Application Benchmarks

It was useful to focus on two simple and explicit applications-oriented benchmark codes to study. We chose a simple summation of long (64 bit) integers and the generation and summation of 64 bit pseudo random numbers generated using the Mersenne-Twistor algorithm [29]. Experience shows that it is often important to make sure that a benchmark loop does something with each iterative value - such as sum them - and writes out the result to avoid over clever compilers optimising the loop contents out altogether. These choices focus on speed and processor operation performance rather than memory access considerations, which are an area for a separate study.

The applications scenario we are interested in is when we have a simulation model that must be run many times, either with separate (usually random) starting conditions to obtain a statistical sampling of model trajectory space, or when there is some parameter space to explore. A rule of thumb with such work is that the standard deviation of the independent samples. Often 10 or 100 such runs is sufficient to ascribe a standard deviation to a numerical measurement, than memory and communications effects.

We have included details of clock speeds as well as core and processor details to support meaningful direct comparisons of the speeds presented below. It is also useful to know any particular cache memory limitations which we also present, although we have chosen the application benchmarks to focus primarily of speed performance rather than memory and communications effects.

3 Application Benchmarks

It was useful to focus on two simple and explicit applications-oriented benchmark codes to study. We chose a simple summation of long (64 bit) integers and the generation and summation of 64 bit pseudo random numbers generated using the Mersenne-Twistor algorithm [29]. Experience shows that it is often important to make sure that a benchmark loop does something with each iterative value - such as sum them - and writes out the result to avoid over clever compilers optimising the loop contents out altogether. These choices focus on speed and processor operation performance rather than memory access considerations, which are an area for a separate study.

The applications scenario we are interested in is when we have a simulation model that must be run many times, either with separate (usually random) starting conditions to obtain a statistical sampling of model trajectory space, or when there is some parameter space to explore. A rule of thumb with such work is that the standard deviation of the experimental measurements often goes as \( \frac{1}{\sqrt{N}} \) for \( N \) independent samples. Often 10 or 100 such runs is sufficient to ascribe a standard deviation to a numerical measurement, so as to be able to interpret it properly.

Such jobs are entirely independent and need not exchange data during the inner loops of their runs. In other contexts such numerical problems can be managed using shell scripts running many processes a nd collating results to file afterwards. Our goal in this present work is to determine how easily such numerical experiments can be run within a single Java program handling separate threads for each independent task. We give our codes and components for managing the Java 7 callable futures explicitly below.

The code listed in Figure 2 shows how the Java ExecutorService is used to manage a pool of threads of a size determined by the user at runtime. Each is instantiated as shown with a small job object as payload, and they are added to the pool queue and invoked. We use the Future mechanism to collect return values from each and record these. This is a convenient mechanism to ensure the synthetic benchmark calculation is not optimised out by the compiler and that we do obtain representative wall clock times from the calculation as the number of job threads is scaled.

The code listed in Figure 3 shows our simple job object for summing 64 bit-long integers and indicates how the Java Callable<return-type> generic mechanism is used in practice.

Figure 4 shows a similar job object that encapsulates the Mersenne Twistor pseudo-random number generator. We have tuned the number of deviates generated down to \( 10^3 \) to attain run times that are long enough to avoid clock rounding, but not so long as to obviate averaging over many runs.
4 Timing and Speedup

While it is useful to consider absolute times to take account of the different speed/price advantages of individual systems, speedup measured in the classical manner is useful to compare the scalability of the multi-threaded software benchmarks on the mix of systems being compared.

Parallel speedup for \( n \) processors - or threads in this present work - is usually defined as:

\[
S_n = \frac{T_1}{T_n}
\]

where \( T_1 \) is the time taken for a single thread and \( T_n \) the time taken for \( n \) such threads. It is important to estimate the experimental uncertainty in this, and we obtain this from experimental standard deviations made in a set of times. Although a computer ought to be a completely deterministic device, there are a number of factors that can lead to a spread of measurements. Firstly if random number generation is involved then in principle the calculation may take subtly different lengths of time to compute depending on hardware and algorithm details. This is unlikely to be an important effect in our present reported benchmarks where we only sum long or random double values. Secondly, and more importantly, any modern computer system typically has a large number of independent processes running to service the operating system. While one can minimise the number of unusual interrupts or events by temporarily isolating computers from the network and limiting user access for duration of benchmarking, inevitably there are operating systems processes that wake up and consume resources briefly. Generally experience shows that these will contribute only a small uncertainty to benchmark times providing suitable averaging is done over for example 10 or more independent runs.

Finally, common sense indicates that clock accuracies are limited and any timing measurement that is comparable in size to CPU clock granularities will inevitably be contaminated by context switching times. While it is possible on some systems it is possible to obtain very accurate clock timings, it is advantageous for this sort of study to be able to use the portable and readily available systems clock. This is not necessarily more reliable than to a resolution of around milliseconds. The benchmarks have therefore been tuned so that the repeated number of operations performed lead to times of the order of tens of seconds or higher. These should therefore dominate any millisecond fluctuations or indeed any process or OS daemon context switching times of sub second scale. This means that the measured times should be representative - on average - of the quantities of actual interest and that conclusions can be safely drawn.

Using the usual first order calculus of derivatives to study the propagation of uncertainties in 1, we find that the uncertainty in speedup \( \delta S_n \) is obtained by adding the relative uncertainties in the times \( T_1 \) and \( T_n \).

\[
\delta S_n \approx \left( \frac{\delta T_1}{T_1} + \frac{\delta T_n}{T_n} \right) S_n
\]

For our purposes here we simply use the standard deviations in the distribution of measured values as the uncertainty values for \( \delta T_1 \) and \( \delta T_n \). This means that all our data points are heavily dependent upon a reliable and accurate measure of \( T_1 \) - the time for a single thread. In the results that follow, we therefore average over 10-20 individual timing measurements to minimize this uncertainties and which we plot as error bars on the diagrams.

5 Performance Results

We present data for the two Java benchmarks on the platforms described in Table 1. We consider the raw absolute timing data for each.

Figure 5 shows the raw timing data for the five platforms running the long summation benchmark. Not surprisingly the slow clock speed 2-core Xeon fares worse and the higher clock speed 8-core Xeon fares best. The 16 core Bulldozer has an intermediate clock speed and taking this into account fares reasonably well.

Figure 4. Callable Future class for encapsulating random number generator Benchmark method.
Figure 6. Java RNG benchmark times.

Figure 7. Java long summation benchmark speedups for Intel and AMD processors.

Figure 8. Java RNG benchmark speedups for Intel and AMD processors.

Figure 6 shows the raw absolute times for the five platforms running the random number generation benchmark. These exercise 64-bit floating point double computations and therefore exercise different core features. The ranking of processors is broadly similar however with some anomalous features starting to show up when particular platforms run out of actual hardware for the requested numbers of threads in the pool. Apart from relatively small fluctuational noise the curves are mostly monotonic. The systems improve with increased numbers of threads as might be expected and generally once they hit the limits of the number of actual cores present, the attainable performance plateaus. We have not attempted massively large numbers of threads for these benchmarks so we do not expect to hit a regime where adding threads actually lowers performance.

Figure 7 shows the parallel speedup evaluated for the long summation benchmark data. We see the details of the integer tasks affected by clever pipelining operations on the Intel processors. The 2-core Intel CPU saturates at a speedup of around 2.5 and the 4-core saturates at around a parallel speedup of 5 as a result of supplying the CPU with an abundance of work to keep its pipeline full.

Note also that the Intel dual 6-core Xeon system continues to increase in parallel speedup well beyond the expected 12 core point and in fact all the way up to 24 cores. This appears to be the effect of Intel’s hyper-threading management system that supports two virtual cores per real physical core. This is impressive and appears to squeeze the last drop of performance from the 12 real hardware cores in this system in a steady manner.

Figure 8 shows the parallel speedup evaluated for the random number generation benchmark. In this case the more sophisticated random number generation algorithm cannot be easily pipelined by the compiler as part of its optimisation and we see the 2-core saturate at a speedup of around 2, the 4-core around 4 and the 8-core around 8. Interestingly the dual 6-core does not manage to reach a steady speedup of 12, presumably due to bus and other memory resource contentions between to two CPUs. The 16-core Bulldozer likewise does not saturation values much different from the dual 6-core Xeon.

Again the Intel hyper threading system seems to allow the dual 6-core Xeon to scale up nearly linearly (albeit at a lower rate) to 24 - double the number of physical cores available. There are more fluctuations in the curve than for the long summation benchmark - presumably again because the random number generator algorithm is more limiting in terms of pipeline and other potential compiler level optimisations.

It is interesting to note the region of straight-line monotonic speedup. A straight-line least squares fit indicates a near perfect linear speedup - for each platform - until there are insufficient cores to service the required numbers of threads. The Intel processors deliver more or less over this expected regime. The AMD Bulldozer delivers up to around 8 threads and afterwards tends to tail
off, although on average the speedup is still monotonically increasing until it hits 16 cores. This is disappointing but appears to be fairly universal. It is not entirely clear what is missing from the AMD architecture or support system to keep the cores fed for longer.

We investigated the Bulldozer performance with a larger number of threads and hence more available work on dual socket blade platforms. We were particularly interested in how the computations and multi thread management systems scaled towards a high number of threads on the (relatively) cheap AMD Bulldozer processors with their 16 real physical cores each and on our dual socket systems.

Figure 9 shows that the dual Bulldozer on a well configured blade configuration fares quite well. The initial linear speedup region continues to around the expected 16 core point, then a kink appears and a new near linear region continues to around the expected 32 core point. This might be expected as the two Bulldozers will experience bus and memory contentions. The ringing overshoot phenomena is less easy to explain. We hypothesis that it represents a partial pipeline optimisation success, the benefit of which tails off above 36 cores.

Figure 10 shows a similar parallel speedup for the Bulldozers on the random number generator benchmark. In this case the speedups saturate without overshoot, probably because the optimising compiler was unable to pipeline the random number generator algorithm.

These data give us some quantitative indications of which platforms give best performance for semi-interactive computational simulation experiments.

6 Discussion

It is always difficult to report on price performance for computer systems in the scientific literature since price is generally very volatile, usually falls rapidly and can vary considerably in different parts of the world. Nevertheless it does appear that the interesting competition is between dual 6-core Xeon processors and the dual 16 core Bulldozer processors. Taking into account the costs of the blade chassis and support environment for the dual Bulldozer system compared to a those of a desktop Mac Pro containing the dual 6-core Xeons, the latter would seem a better economic proposition at least for a University research group. This statement should be taken with many caveats however, as there might be a number of other economic total cost of ownership and maintenance considerations that would rule in favour of a rack-mounted blade for other groups.

Generally we have seen that the Java 7 JVM and its associated runtime performs rather well for all the platforms considered. The Java concurrency software appears to be considerably easier to use and a lot better in exploiting performance than prior Java thread management software.

These benchmarks although simple in nature have shown that there a number of subtle compiler level optimisation effects that are present and which are beyond user control. It is nevertheless impressive how well the JVM performance has scaled in all cases. Intel’s hyper-threading mechanism is particularly noteworthy and does appear to let the lower levels of the operating system cope with making the best use of available pipelining performance for codes such as these.

It is attractive to be able to integrate the different runs of a numerical experiment in a multi threaded simulation code of this nature without recourse to managing separate operating systems processes, jobs and accumulating results in separate files for example. The Java 7 concurrency mechanism do seem to meet the need for a software framework to manage this sort of work.

We have only considered independent job management in this present work. There is scope to study inter-thread data communications and its affect on performance. That would however be more relevant to use of Java threading for data-parallelism or some other form of problem decomposition, rather than the job management scenario we
have considered here.

7 Conclusion

We have described the problem of managing complex numerical experiments with many independent jobs either with different random seeds or with different model parameter spaces to explore. We have described use of the Java 7 thread concurrency features for managing such a scenario and have reported on the detailed performance of various combinations of multi-core CPUs.

In summary, we believe the Java 7 thread concurrency management facilities are quite suitable for work of this nature and that it is very encouraging how well our example codes scaled on all the platforms tested. There is scope for testing more sophisticated data communications mechanisms such as tuple space management or some explicit messaging framework to support non-independent computational tasks under a single Java Virtual Machine.

Confidence in the Java 7 thread management is likely to support continued development of more sophisticated complex systems simulations software that is both reusable, platform-portable, but still able to exploit a good fraction of the available performance on multi-core and multi-cpu desktops and blade systems.

References


