Predicting Multi-Core Performance:A Case Study Using Solaris Containers

Akbar Siami Namin
Advanced Empirical Software
Testing and Analysis(AVESTA)
Computer Science
Texas Tech University
akbar.namin@ttu.edu

Mohan Sridharan
Computer Science
Texas Tech University
mohan.sridharan@ttu.edu

Pulkit Tomar
Advanced Empirical Software
Testing and Analysis(AVESTA)
Computer Science
Texas Tech University
pulkit.tomar@ttu.edu

ABSTRACT

Multi-core technology is an emerging hardware trend that provides significant capabilities for computationally expensive applications. However, it also demands a paradigm shift in the software industry, Software developers need to think about the best distribution of software components across the available CPUs, and trade-off the computational efficiency against the cost of re-structuring the standard sequential execution of software. The relationship between the measured performance and the corresponding parameters such as the number of threads and CPUs remains an interesting open problem, especially since it is a challenge to conduct controlled experiments.

This paper reports a case study on the use of Solaris containers to control the assignment of threads to the available CPUs in a set of applications. We model the performance as a function of the number of threads, the number of CPUs and the type of program. We use two different modeling strategies: linear regression and Neural Networks, which are applied to the well-established Java Grande benchmark. We observe that there is a nonlinear relationship between these parameters and the associated performance. In addition, neural network models are observed to be consistently better at estimating the performance over a range of parameter values. The results reported in this paper can therefore be used to suitably re-structure software programs to fully utilize the available resources.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Measurement

Keywords

Multi-Core Performance, Solaris Containers, Regression, Neural Networks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWMSE '10, May 1 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-964-0/10/05 ...$10.00.

1. INTRODUCTION

In recent times, there has been a radical shift in the computer hardware industry towards the use of multi-core chips. In addition to enhancing the computing performance dramatically, multi-core processors are increasingly having a significant impact on our daily life as a result of their deployment in several critical applications. There is a widely held belief that the full potential of the multi-core systems can only be exploited through the development of new languages and computing practices. Though multi-core architectures are capable of providing a much-needed performance boost to applications involving computationally intensive operations, they also require a major change in the established ways of writing code. The standard sequential execution of code will need a complete makeover in order to fully utilize the potential of the multi-core chips.

Parallel programming has traditionally been under the exclusive purview of the extreme programmers, especially of the associated daunting challenges that are not faced by the programmers who write sequential code. Parallel programming requires that the programmers effectively identify, expose and express parallelisms without introducing any logical (or other) errors in the underlying system. Race conditions, deadlocks and starvation are some of the possible errors that are specific to parallel programs.

A key requirement for the widespread utilization of the the multi-core technology is the development of proper techniques for creating an optimum number of threads and allocating these threads to an optimal number of CPUs. Researchers have explored this problem space extensively by designing suitable static and dynamic models. However, the allocation of threads to CPUs is typically handled by the resource manager of the underlying operating system. It is a challenge to conduct controlled experiments where a specific number of threads are executed on a specific number of CPUs. A recent related development has been that of Solaris containers, which is an implementation of an operating system-level virtualization technology that provides system resource controls. Each Solaris container can act as a totally isolated virtual server that controls the allocation of specific threads to specific CPUs.

This paper uses Solaris containers to investigate the effect of the two key parameters that influence the performance of a multi-core architecture: the number of threads that need to be executed; and the number of CPUs available for executing these threads. We model the measured performance as a function of these underlying parameters using linear re-

gression and neural networks [1]. The chief contributions of this paper are hence as follows:

1. We conduct controlled experiments on effect of the associated parameters on the performance of multi-core architectures, using Solaris containers.
2. We model the performance as a function of the number of threads and CPUs, thereby estimating the optimal allocation of threads to CPUs.

The proposed models were evaluated on two machines with different multi-core architectures, using the Java Grande benchmark [11]. The results show a nonlinear relationship between the performance and the associated parameters. In addition, the neural network model performs consistently better than the linear regression models. Furthermore, we identify points where the addition of a single thread can result in severe degradation of performance.

The remainder of the paper is organized as follows. Section 2 briefly summarizes related work on multi-core technology and benchmarks. Section 3 describes the experimental setup of the containers and the machines. Section 4 analyzes the measured data to build linear regression and neural network models. Section 5 discuses the ramifications of the results obtained, followed by the threats to validity (Section 6) and the conclusions (Section 7).

2. RELATED WORK

In this section, we first review some representative Java benchmarks for performance evaluation. Next, we discuss some related work on performance measurements in parallel applications.

2.1 Java Benchmarks

Bull et al. [3, 4, 5] introduced a set of benchmarks to assess the performance of execution environment while running sequential programs. The suite, known as Java Grande Benchmark, is divided into three sections comprising low level operations, kernels computation, and large scale applications. Each section contains various inputs representing the size of the input data. The suite is a framework for testing various performance aspects of Java execution environments while running the Java Grande programs.

Smith et al. [10, 11] added parallel and multi-threaded versions of the sequential benchmarks in the Java Grande Benchmark to assess the execution in parallel environments. The parallel versions replace low level benchmarking with threads, `Barrier`, `fork join`, and `synchronization`. Furthermore, synchronization and scheduling based on JOMP (Java OpenMP library) were added. As a precursor of parallel versions, multi-threaded versions were developed by replacing `Barrier`, `forkjoin`, and `synchronization` while the rest of the benchmarks remained untouched. As reported in [10] the sequential versions outperformed the concurrent versions in most cases.

The DaCapo benchmark [2] is a Java benchmark containing a set of real world applications with extensive memory loads. Several benchmarks such as parsers, optimizers and XML transformers have been added to the DaCapo framework. The benchmarks are accompanied by three inputs (small, default, and large). There are several other Java benchmarks such as The Tak Benchmark, Java Generic Library (JGL), RMI Benchmark Suite and JavaWorld benchmark: see [3] for a complete description.

2.2 Auto-Tuning Performance

Several existing papers have discussed the automatic optimization of performance for specific programs, based on the dynamic allocation of threads and CPUs. In [15], the authors focus on identifying the near optimum configuration of tuning parameters from a search space. They evaluate the ability of different number of processors to maximize the performance in terms of criteria such as speed, efficiency, and cost-benefit ratio.

In [8, 9], the authors handle large search spaces of possible tuning parameters by reducing the search space using the characteristic information of parametrized parallel patterns. Schaefer [8] focuses on the problem of large search space for all possible configurations of tuning parameters: number of threads, load per worker, number of worker threads etc. Instead of testing all possible configurations of these parameters, this method uses the characteristics of parametrized parallel patterns, master-worker model and pipelines for limiting the search space of possible values of tuning parameters. Schaefer et al. [9] provide constructs to specify tunable variables in the source code and add meta-information.

Hall et al. [7] propose a dynamic approach of increasing and decreasing the number of threads depending on the performance. They use it to change the degree of parallelism in compiler parallelized code depending on the performance. The work also proposes a method for adaptive thread management that computes the number of threads to use. Sondag et al. [12, 13] discuss assignment of threads on heterogeneous multi core processors by clustering instruction blocks into similar types and mapping the cluster types to cores which can fulfill the requirements of these blocks.

A methodology for creating tools which dynamically monitor execution and address performance drawbacks through optimization techniques that adjust tuning parameters was proposed by Cesar et al. [6]. Dynamic monitoring and optimization is necessary since an application may behave differently in each execution, thus making a static approach inefficient. This method consists of measure points (parameters used for model evaluation), performance functions that identify drawbacks from various measure points, and a tuning tool that addresses these drawbacks through actions such as modification of tuning parameters.

In this paper, we report the result of a case study focusing on thread-to-CPUs assignment using Solaris containers. The experiment has been conducted on two machines capable of treating multi-threaded applications in different ways. Though, the results obtained show consistency between machines, it demonstrates their deficiencies when the number of threads exceeds the number of CPUs allocated for each Solaris container.

3. EXPERIMENTAL SETUP

In this section, we describe the Solaris containers, the multi-core machines and the experiments conducted.

3.1 Solaris Containers

Solaris container, first introduced in 2005 as part of Solaris 10, is a server virtualization implementation that provides isolation between applications. Applications can be managed independently and resources can be allocated to them dynamically. One of the major goals of introducing the Solaris containers was to manage workload resources and gain control over their execution. The resource management in

Solaris containers makes it possible to: restrict access to specific resources, isolate workloads, and define security mechanisms to control their execution. Workloads are defined using *projects*, which are used to create containers and allocate resources such as CPUs. The Solaris command `create` is used to create projects and containers.

During the creation of the containers, the maximum number of CPUs was set using `pset.max`, an argument to the `create pset` command. By default, the minimum number of CPUs is set to 1. However, since a Solaris container automatically changes the number of allocated CPUs until the maximum number of CPUs is reached, we set the minimum number of CPUs involved in each container (`pset.min`) to be equal to the maximum number of CPUs, i.e. $pset.max = pset.min$. This ensured the correctness of the exact number of CPUs allocated for each container and the validity of the experiments.

3.2 Machines Used

For our experiments, we used two Sun machines. The first was a Sun Fire T1000 with a UltraSPARC T1 processor, which is a multi-threaded, multi-core CPU. This machine supports 32 concurrent hardware threads. According to Sun, this machine is well-suited for tightly coupled multi-threaded applications [14]. It contains one 1.2 GHz UltraSPARC T1 processor with 32 GB memory. In addition, it uses CoolThreads technology, thereby offering eight cores with four threads per core. According to Sun, the goal of designing this processor was to run as many concurrent threads as possible [14].

The second machine was a Sun SPARC Enterprise M3000 system that supports eight concurrent hardware threads. According to Sun, this machine is ideal for single-threaded workloads. It is powered by a SPARC64 VII processor, with 64 GB memory and a 2.75 GHz quad-core processor with two threads per core [14].

3.3 Benchmarks Used

The Java Grande Multi-threaded Benchmark was used for the experiments reported in this paper. The benchmark consists of three sections. Section one focuses on low level operators such as forking and joining threads `ForkJoin`, barrier synchronization `Barrier`, and synchronization of blocks and methods `Sync`. Section two concentrates on kernel processes and contains several computationally expensive applications such as Fourier coefficient analysis, LU factorization, successive over-relaxation, IDEA encryption, and sparse matrix multiplication. Section three is a set of large scale application such as molecular dynamic simulation, Monte Carlo simulation, and 3D ray tracer. The benchmark is designed in such a way that the number of threads created and used can be specified as an argument. Further description of this benchmark can be found in [10].

3.4 Setup

We used Solaris containers to create various combinations of CPUs. For the T1000 machine, we created five projects (i.e. containers) with one, two, four, eight, and sixteen CPUs, which were called One-CPU, Two-CPU, Four-CPU, Eight-CPU, and Sixteen-CPU respectively. For the M3000 machine, we created three containers with one, two, and four CPUs. The `poolcfg` command was used to create processor sets, pools, and the required associations. Furthermore, the `projadd` command was used to create projects and their associations to the pools created. The allocation of the logical CPU units to each processor set was handled by the Solaris container automatically. We ran each benchmark for a set of threads ranging from 1 to 50 on each container on each machine. We used the `mpstat` UNIX command to monitor the utilization of CPUs and the correctness of assignments. For our experiments with the Java Grande benchmark, *performance* was measured as the the number of operations accomplished in a given time period. The measured data was used for statistical analysis as described below.

4. DATA ANALYSIS

In this section, we first provide some plots visualizing the measured performance data. Then, we describe the results of modeling this data using regression and neural networks.

4.1 Visualization

Figure 1 depicts the performance over a range of threads and CPUs. The x-axis shows the number of threads generated for the experiments reported in this paper, and the y-axis represents the performance on a logarithmic scale in order to provide a reasonable scaling mechanism for the measured performance.

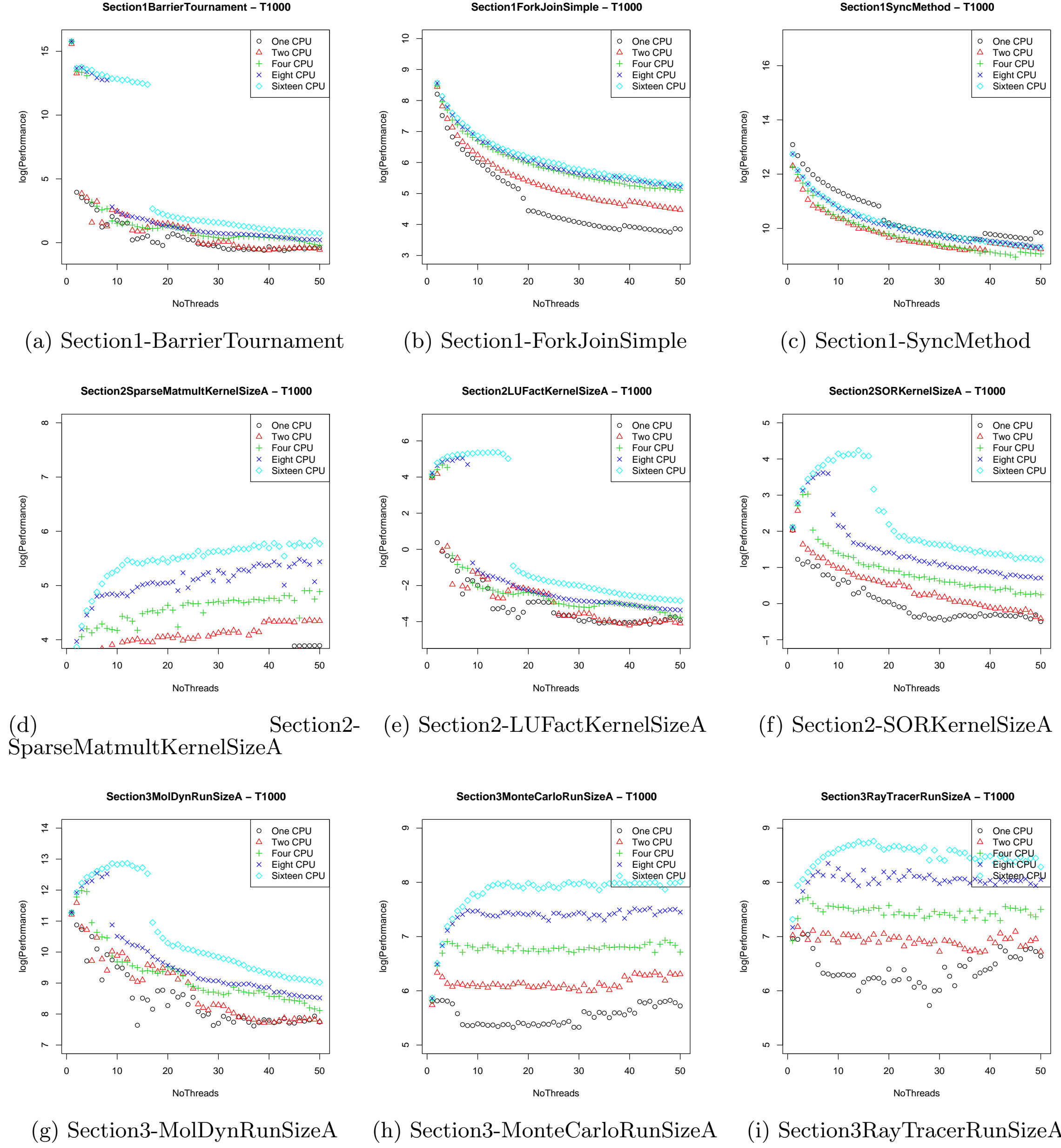
Figure 1(b) shows the performance of each Solaris container for a specific program: `Section1-ForkJoinSimple`. As the number of threads increases, the performance deteriorates. In addition, though there are improvements when additional CPUs are added to the container, the difference is not significant when four more CPUs are added to the container. The results may be interpreted as a suggestion that for applications similar to the fork-join program, the most effective option may be to employ up to four CPUs. On the other hand, Figure 1(c) depicts data for a synchronization program. The figure shows that the cost of synchronization causes a loss in performance when the number of CPUs used is more than one.

Figure 1(a) demonstrates the performance of each Solaris container for: `Section1-BarrierTournament`. The figure indicates that for an application similar to `BarrierTournament`, the performance is considerably high when each thread is assigned a separate CPU. However, the performance drops drastically when even one additional thread is added to a Solaris container that has a one-to-one mapping between threads and CPUs. Ideally, we would have expected to see a smoother degradation in performance. Similar cases are depicted in Figures 1(e), 1(f) and 1(g).

Figures 1(d), 1(h) and 1(i) show more ideal cases where increasing threads actually improves the performance. These benchmarks are typical instances of parallel applications. For instance, matrix multiplication and Monte Carlo sampling are cases where we may split the process into independent sub-processes that are handled separately. We also observe a significant improvement when using more CPUs.

4.2 Linear Regression Models

In many application domains, the measured (target) variable is a function of one or more attributes. Researchers are often interested in modeling and understanding the possible inter-relationships between these attributes, and their effect on the measured quantity. Though these attributes may be correlated, it is often assumed (for ease of analysis) that they are independent of each other. In addition, it is often also of



(a) Section1-BarrierTournament (b) Section1-ForkJoinSimple (c) Section1-SyncMethod

(d) Section2-SparseMatmultKernelSizeA (e) Section2-LUFactKernelSizeA (f) Section2-SORKernelSizeA

(g) Section3-MolDynRunSizeA (h) Section3-MonteCarloRunSizeA (i) Section3RayTracerRunSizeA

Figure 1: Performance as a function of threads and CPUs for Solaris containers on Sun Fire T1000.

interest to measure the contribution of each such attribute to the target variable. Linear regression is a well-known tool to model and understand such relationships.

A linear regressions model fits a best possible line among observed data such that the sum of squares of the residuals is minimized. A linear model of data is a linear equation of the form $Y = C_0.X_1 + C_1.X_2 + \cdots + C_n.X_n + \epsilon$, in which the X_i variables correspond to attributes or parameters and Y corresponds to an outcome variable. Regression is the process of computing the values for the coefficients $C_i : i \in [0, n]$ that best fit the actual data.

There are several possible ways to fit a line to points in space of attributes, resulting in several different ways to construct the desired models. Model checking is therefore required to identify the best possible model. The *coefficient of determination* R^2 and *mean squared error* (MSE) are two popular measures of the precision of the constructed models. The R^2 measure can be used for interpreting the correlation between two variables i.e. it measures how the change in one variable affects the change in the other variable. A value of $R^2 = 1$ represents perfect correlation between two variables, while $R^2 = 0$ represents the lack of any correlation. The R^2 measure can also be used to represent the quality of fit of a model to the measured data.

Based on observed data in Figure 1, we hypothesized the following relationship between the measured performance and the associated attributes (number of threads and CPUs):

$$log(Performance) = C_0 + C_1 * log(nCPU) + C_2 * log(nThread) \quad (1)$$

where the target (i.e. response) variable Performance is a linear function of the number of threads ($nThread$) and the number of CPUs ($nCPU$) in the logarithmic (base 10) space. As mentioned before, performance represents the number of operations accomplished in a given time period.

We fit various linear regression models to predict the performance as a function of the number of threads and the number of CPUs. In order to estimate the individual contribution of each of the attributes ($nThread$, $nCPU$), we generated linear regression models that predicted Performance using: (a) $nThread$ alone; (b) $nCPU$ alone; and (c) both $nThread$ and $nCPU$. In addition, we also generated a linear regression model that does *not* operate in the logarithmic space. The assessment of these models indicated that the relationship hypothesized in Equation 1 results in the best fit to the measured data. The quality of fit with such a model was then measured using the R^2 and MSE measures. Table 1 summarizes the results of applying this model on the Java Grande benchmark dataset, on the two different machines described in Section 3.2. The table presents the regression coefficients that provide the best fit to the measured data, and the values of the R^2 and MSE measures.

Table 1 indicates that the R^2 values for T1000 machine varies from 0.499 to 0.944, whereas, the values of $R2$ for M3000 machine change between 0.194 to 0.977. For most benchmarks, using Equation 1 resulted in good models. The model performed poorly (*Section3: RayTracerInitSizeA*) only when the underlying data was very noisy.

An investigation of p-values associated with the coefficients of the regression models (i.e. C_0, C_1, C_2) showed that the variables $nThread$ and $nCPU$ have significant contributions to the measured Performance i.e. p-values computed at 95% level of confidence were significantly less than 0.05. In addition, we noticed that for the benchmarks programs in Section-1, increasing number of CPUs increases the performance significantly—positive values for C_1. The synchronization programs (e.g. `Section1SyncMethod`) were the only exceptions—here C_1 is negative. This is expected because increasing the number of CPUs for these programs imposes an extra cost of synchronizing the CPUs.

The values computed for C_2 were mostly negative. This indicates that increasing number of threads degrades the performance. The only exceptions were *Section2: SeriesKernelSizeA*, *Section2: SpareMatmultKernelSizeA*, *Section3: MonteCarloRunSizeA*, *Section3: MonteCarloTotalSizeA*, and *Section3: RayTracerRunSizeA*. These programs correspond to applications that are suitable for parallelism. For instance, matrix multiplication and Monte Carlo methods are known to be suitable for parallel executions. The computed values for C_2 are hence reasonable. An important implication is that not every application is suitable for parallelism, and predictive models may be used to identify applications suitable for parallel execution.

Linear regression models are hence appropriate tools to investigate the influence and contributions of the predictors ($nThread$, $nCPU$) to the target variable (Performance). The magnitudes and signs of regression coefficients reflect the level of contribution of each variables used.

4.3 Neural Network

In addition to the linear regression model described above, we also implemented a neural network model. Neural networks are a well-established machine learning technique for classification and regression problems in several different application domains [1]. Nodes in the network represent individual variables (or hidden state abstractions), while the connection between the nodes represent the relationships between the variables. The basic neural network model is similar to the linear regression model, but suitable bias and transformation functions can be readily incorporated to model complex relationships between the input attributes and the target variable—see Figure 2.

In our trained neural network model, the target variable (i.e. Performance) for each program in the benchmark (5, 5, 7 in each of the three sections) is still modeled as a function of the number of CPUs and the number of threads in log-space, as described in Equation 1. We used a two-layer neural network with two inputs ($log(nThread)$, $log(nCPU)$), one output (log(Performance)) and one hidden layer with 15 nodes, as shown in Figure 2.

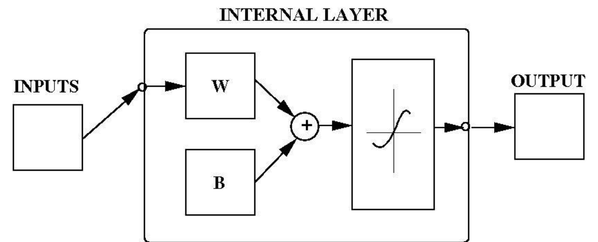


Figure 2: Block diagram of the Neural Network model used in the experiments.

In Figure 2, the terms W and B denote the matrices of weights and bias values that can be tuned to improve the quality of fit of the relationship between the inputs and the output. In order to enable the use of such a model to predict the best distribution of the threads across the available CPUs, we used the train-validate-test process that is typically used in the machine learning literature i.e. we created a $60 - 20 - 20\%$ split of the available data [1]. In other words, 60% of the data (for each program on each machine, over the range of CPUs and threads) was used to train the parameters of a neural network, which were tuned further (i.e. validated) using a separate 20% of the dataset. The network with the tuned parameters was then tested on the remaining 20% of the available data. This experiment was repeated ten times and the results in Table 2 present the average R^2 value and MSE during these experiments.

Table 2 shows that the neural network model consistently provides a good fit to the measured data—the quality of fit is better than the linear regression model. As mentioned above, the weights and bias components, along with the nodes in the hidden layer and the nonlinear transformation, enable the network to build a much more sophisticated model than the standard linear regression technique. As a result, the prediction capabilities are better than that of the linear regression model, even in cases where there is a lot of noise in the measured performance data. The neural network model's performance is not good only in cases where there is a large amount of noise in the measured performance over the set of threads and CPUs. However, the neural network model operates like a black-box—unlike the linear regression models, it is difficult to thoroughly analyze the effect of the individual parameters involved in the net-

Table 1: Summary of linear regression models of the form $log(Performance) = C_0 + C_1 * log(nCPU) + C_2 * log(nThread)$ fitted for two machines.

Benchmark Programs	T1000					M3000				
	C_0	C_1	C_2	R^2	MSE	C_0	C_1	C_2	R^2	MSE
Section1:BarrierSimple	11.460	0.149	-1.356	0.905	0.151	12.973	0.360	-1.366	0.900	0.164
Section1:BarrierTournament	11.457	1.554	-3.772	0.718	5.234	9.817	0.817	-3.071	0.651	4.024
Section1:ForkJoinSimple	9.958	0.519	-1.620	0.742	0.776	9.951	0.899	-1.212	0.977	0.026
Section1:SyncMethod	12.846	-0.036	-0.915	0.894	0.076	15.384	-0.489	-1.033	0.681	0.650
Section1:SyncObject	12.819	-0.040	-0.907	0.891	0.078	15.435	-0.450	-1.052	0.677	0.439
Section2:SeriesKernelSizeA	5.424	0.892	0.184	0.944	0.047	7.507	0.962	0.037	0.950	0.015
Section2:LUFactKernelSizeA	3.602	1.098	-2.331	0.753	1.763	3.454	0.595	-2.168	0.790	1.000
Section2:CryptKernelSizeA	7.483	0.787	-0.053	0.741	0.208	8.769	0.887	-0.023	0.713	0.101
Section2:SORKernelSizeA	2.179	0.799	-0.748	0.841	0.198	3.305	0.711	-1.062	0.866	0.160
Section2:SparseMatmultKernelSizeA	2.452	0.710	0.352	0.936	0.039	5.238	0.943	0.132	0.874	0.207
Section3:MolDynRunSizeA	11.758	0.735	-1.139	0.838	0.294	13.129	0.412	-1.451	0.907	0.172
Section3:MolDynTotalSizeA	-2.317	0.735	-1.133	0.842	0.283	-0.930	0.414	-1.448	0.909	0.168
Section3:MonteCarloRunSizeA	5.112	0.828	0.153	0.938	0.044	7.184	0.944	0.064	0.938	0.019
Section3:MonteCarloTotalSizeA	-4.08	0.742	0.131	0.936	0.036	-2.141	0.925	0.066	0.931	0.020
Section3:RayTracerInitSizeA	8.851	0.240	-0.809	0.499	0.561	9.865	0.187	-0.547	0.194	0.958
Section3:RayTracerRunSizeA	6.382	0.759	0.008	0.933	0.039	9.324	0.856	-0.445	0.847	0.070
Section3:RayTracerTotalSizeA	-3.568	0.741	-0.027	0.931	0.039	-0.601	0.790	-0.516	0.861	0.065

Table 2: Result of modeling the performance with Neural Networks.

Benchmark Programs	T1000		M3000	
	R^2	MSE	R^2	MSE
S1:BarrierSimple	0.991	0.051	0.908	0.003
S1:BarrierTournament	0.924	0.651	0.961	0.174
S1:ForkJoinSimple	0.996	0.034	0.995	0.004
S1:SyncMethod	0.992	0.002	0.963	0.043
S1:SyncObject	0.994	0.002	0.937	0.042
S2:SeriesKernelSizeA	0.931	0.101	0.851	0.087
S2:LUFactKernelSizeA	0.982	0.036	0.961	0.193
S2:CryptKernelSizeA	0.994	0.004	0.902	0.020
S2:SORKernelSizeA	0.984	0.002	0.963	0.011
S2:SparseMatmultKernel-	0.971	0.017	0.923	0.035
S3:MolDynRunSizeA	0.968	0.057	0.938	0.048
S3:MolDynTotalSizeA	0.967	0.052	0.935	0.034
S3:MonteCarloRunSizeA	0.990	0.003	0.978	0.013
S3:MonteCarloTotalSizeA	0.992	0.022	0.943	0.008
S3:RayTracerInitSizeA	0.612	0.385	0.595	0.496
S3:RayTracerRunSizeA	0.986	0.007	0.937	0.045
S3:RayTracerTotalSizeA	0.985	0.006	0.938	0.056

work. We therefore do not include the network parameter values in Table 2.

Finally, Figures 3(a)–3(b) show the regression results of the trained neural network on the test dataset corresponding to the execution of a range of threads for two specific programs on the available range of CPUs. Figure 3(a) corresponds to the execution on the T1000 machine, while Figure 3(b) corresponds to the execution on the M3000 machine. The figures show that the regression functions match the true performance very well. They can hence be used to predict the performance and decide on the best distribution of threads across the available CPUs.

5. DISCUSSION

In this section, we discuss the ramifications of the experimental results tabulated in the previous section.

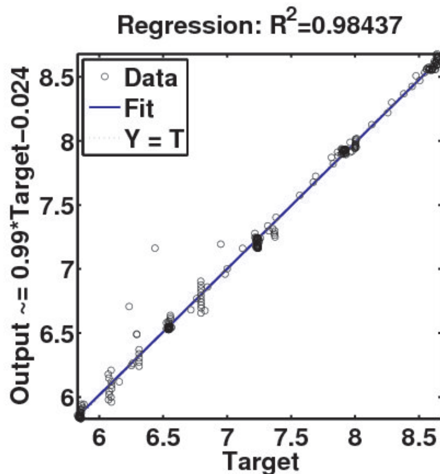
5.1 A Comparison with Related Work

Smith et al. [11, 10] reported the results of experiments conducted on JDK 1.2.1_04, JDK 1.3.1, JDK 1.3.0 running on HPC 18 400 MHz UltraSparc II processors with Solaris 2.7, and SGI JDK 1.3.0 running on 128 400 Mhz MIPS R12000 processors with IRIX 6.5. The results reported in [10, 11] indicate that for benchmarks related to threads, `BarrierTournament` showed best performance followed by `BarrierSimple`, while `ForkJoin` had the worst performance. Our experiments show different results: `BarrierTournamen` performs worse than `ForkJoin`. Furthermore, we obtained different results regarding the `LUFactKernel`, `MonteCarlo`, and `RayTracerRun`. Though [11] reports increase of speed for `LUFactKernel`, we observed that the performance increases as long as the number of threads and CPUs are balanced. Upon adding more threads, the performance drops sharply—see Figure 1(e). We obtained results for `MonteCarlo` and `RayTracerRun` that match with those reported in [11]. The reason for performance improvement while adding more threads to the execution of these programs is their inherent nature that supports parallelism. In addition, the results obtained for `ForkJoin` are consistent with those reported in [11]. It is a well-known fact that performance obtained using multi-core processors is a function of several factors such as the chip architecture, memory bandwidth and input programs. The goal in this paper is to analyze the contribution of some key factors (threads, CPUs), and advocate the use of strong statistical tools to model and hence predict the performance for any given application.

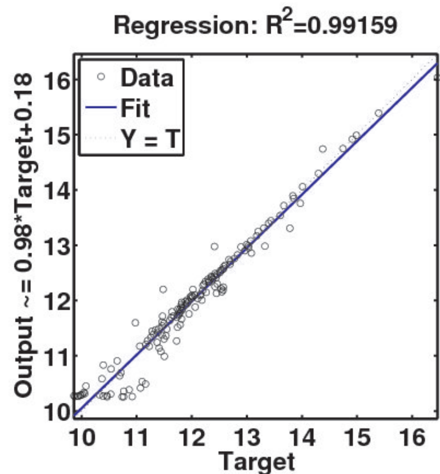
5.2 M3000 vs. T1000

Though there was no intention initially to compare the performance of M3000 (Java version 1.5) with T1000 (Java version 1.6), we can report that the performance of M3000 was considerably better than T1000. There could be several reasons for this observation. M3000 is basically a stronger machine than T1000. However, according to technical specification of both machines, T1000 is supposed to work better for multi-threaded application. Our results do not support this statement. Java compiler versions may have had some influence on the result.

From architectural point of view all cores in the UltraSPARC T1 machine share a single floating point unit, while each core in the SPARC64 machine has one floating point unit. As a matter of fact, the chip architecture has signifi-



(a) Program *Section2:SORKernelSizeA* on T1000.



(b) Program *Section1:SyncObject* on M3000.

Figure 3: Plots showing the regression performance for two specific programs, over all combinations of CPUs and threads using neural networks.

cant impact on performance. In addition, for highly parallel applications where there is little or no sharing or synchronization between threads, the performance is expected to be better on a 2.75GHz than on a 1.2Ghz machine. Furthermore, other factors (in addition to number of CPUs and threads) may affect the measured performance.

Figure 4 compares the performance of two machines for one of the benchmarks.

6. THREATS TO VALIDITY

The subject programs used in the study were middle-size Java programs. The models learned for these benchmarks may or may not perform well on other programs. Hence, we cannot draw any general conclusion that the results can be applicable to other programs written in other languages. Moreover, the architecture of other machines may influence the precision of models developed. However, the proposed schemes can be easily used to generate suitable models based on the data obtained for other specific applications.

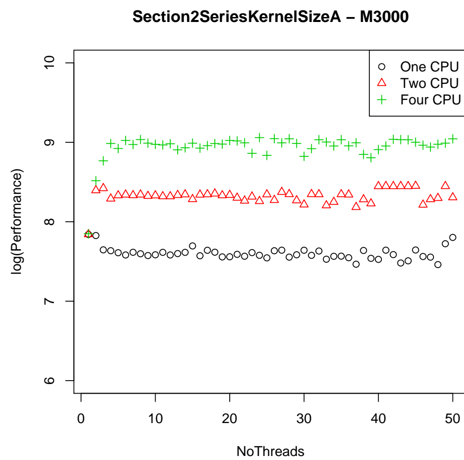
The Solaris projects were defined according to the Solaris containers created. However, the executions of benchmarks for all Solaris projects were performed with significant overlap. Specifically, we ran each benchmark 50 times, which reflects the number of threads created each time. These tests were conducted simultaneously for the five projects defined on T1000 i.e. One-CPU, Two-CPU, Four-CPU, Eight-CPU, and Sixteen-CPU. Similarly, we ran the benchmarks for the three projects defined on M3000, i.e. One-CPU, Two-CPU, and Four-CPU, simultaneously. Since the exact number of physical processors in each machine is one, there is a cost overhead associated with creating Solaris containers and running the applications allocated to them simultaneously. This simultaneous execution of benchmarks assigned to each container may introduce some construct threats. However, the performance degradation due to simultaneous execution is proportional to the number of CPUs allocated to each Solaris container and has little or no effect on the results.

We had Java 1.5 and 1.6 installed on T1000 and M3000 respectively. There has been some research work emphasizing the role of Java compiler on performance. For instance [4] concludes that difference in best and worst overall performance being less than a factor of two is due to the improvements in Java compiler technology. Furthermore, it has been reported in [5] that the difference in effectiveness of optimization in different compilers leads to major differences in the performance, as in case of Sun JDK 1.2.1. Though the Java compiler version might affect the result while comparing two machines, it does not affect the result when the experiments are run independently on the machines. We constructed some models and observed the true behavior of each program on each machine independently, and confirmed that the constructed models hold true for both machines.

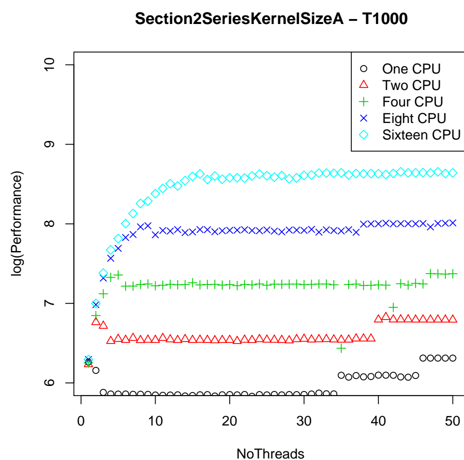
7. CONCLUSION AND FUTURE WORK

In this paper, we have reported the result of a case study conducted on two machines. We used Solaris containers in order to have a valid assignment of CPUs to each multi-threaded application in the Java Grande benchmark. Using regression models, we showed that the measured performance is a linear function of the number of CPUs and the number of threads in the logarithmic space. Results show that both parameters (number of CPUs and threads) make significant contributions to the measured performance. We also observed that a more sophisticated neural network model provides better prediction capabilities than the standard linear regression models. Overall, we observed that the performance increases as long as there is a on-to-one mapping between the number of threads and the number of CPUs in each Solaris container. However, the performance deteriorated drastically when the number of threads exceeded the number of CPUs allocated to a container.

This work is part of a research project concerning auto-tuning of parallel applications. The project intends to provide an online mechanism for allocating threads to CPUs while running applications. Though Solaris containers allow resource allocation, they lack automated mechanisms for dynamic assignment of threads to CPUs. In addition, we would like to compare the performance of sequential programs to



(a) M3000



(b) T1000

Figure 4: A comparison of performance for M3000 and T1000.

their multi-threaded versions, in order to understand when parallelism really improves the performance. In the long-term, we aim to investigate the adaptive testing of multi-threaded applications running on multi-core systems.

Acknowledgment

Many thanks to Larry Pyeatt and David Hensley in the computer science department at Texas Tech University for their help in setting up the Solaris Containers used in the experiments reported in this paper.

8. REFERENCES

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2008.

[2] S. Blackburn, R. Garner, C. Hoffman, A. Khan, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VavDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: The 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169 – 190, New York, USA, October 2006.

[3] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking java grande applications. In *Proceedings of ACM 1999 Java Grande Conference*, pages 81–88, June 1999.

[4] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande applications. In *The International Conference on The Practical Applications of Java*, pages 63–73, 2000.

[5] J. Bull, M. Westhead, D. Henty, and R. Davey. A benchmark suite for high performance java. *Concurrency, Practice and Experience*, 12:375–388, 2000.

[6] E. Cesar, A. Moreno, J. Sorribes, and E. Luque. Modeling master/worker applications for automatic performance tuning. *Journal of Parallel Computing*, 32:568 – 589, 2006.

[7] M. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proceedings of the 2nd SUIF Compiler Workshop*, Stanford University, August 1997. USC Information Sciences Institute.

[8] C. Schaefer. Reducing search space of auto-tuners using parallel patterns. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 17 – 24, May 2009.

[9] C. Schaefer, V. Pankratius, and W. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. Technical report, University of Karlsruhe, 2009.

[10] L. Smith and J. Bull. A multithreaded java grande benchmark suite. In *Proceedings of the Third Workshop on Java for High Performance Computing*, Sorrento, Italy, June 2001.

[11] L. A. Smith, J. Bull, and J. Obdrzalek. A parallel java grande benchmark suite. In *Proceedings of SC2001*, Denver, Colorado, November 2001.

[12] T. Sondag, V. Krishnamurthy, and H. Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *ACM SIGOPS 4th Workshop on Programming Languages and Operating Systems*, Skamania Lodge, Stevenson, Washington, USA, October 2007.

[13] T. Sondag and H. Rajan. Phase-guided thread-to-core assignment for improved utilization of performance asymmetric multi-core processors. In *The International Workshop on Multicore Software Engineering*, Vancouver, Canada, May 2009.

[14] Sun. Oracle Sun SPARC Enterprise Servers, 2010. `http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/index.html`.

[15] O. Werner-Kytl and W. Tichy. Self-tuning parallelism. In *The International Conference on High-Performance Computing and Networking*, pages 300–312, 2000.