An OpenMP runtime profiler/configuration tool for dynamic optimization of the number of threads

Tamara Dancheva, Marjan Gusev, Vladimir Zdravevski, Sashko Ristov
Ss. Cyril and Methodius University, Faculty of Computer Science and Engineering
1000 Skopje, Macedonia
Email: tamaradanceva19933@gmail.com, {marjan.gushev, vladimir.zdraveski, sashko.ristov}@finki.ukim.mk

***Abstract*—This paper describes the implementation and the experimental results of a tool for dynamic configuration of the number of threads used in the OpenMP environment based on the current state of the runtime at the time of the call. For this purpose, we use a mix of profiling and machine learning techniques to determine the number of threads. The decision to set the number of threads is made at the time of the call.**

The proposed approach is designed to be cost effective in the scenario of a highly dynamic runtime primarily when running relatively long tasks consisting of a number of parallel constructs.

I. Introduction

The OpenMP API was designed with a goal to provide a simple, fast, compact and portable way of parallelizing programs [1]. Computer architectures have diversified and evolved a great deal since the first release requiring OpenMP to evolve as well. Today, considering the prevalence of SMPs and SMT on the market, the massive virtualization and ccNUMA architectures, the default schemes that have sufficed in the past are no longer producing the desired results with the same consistency as before.

Multi-cache architectures, hyper threading and multi-core techniques helped prevent the potential stagnation in performance due to the limits imposed by physics with a cost. The intensive exploitation of virtualization techniques especially lately (of memory, computing power, application, operating systems) only enhance the complexity introduced.

In order to get a good performance, programmers as well as scientists have had to start studying the hardware characteristics, cache associativity, swapping, virtualization and many other low-level hardware and software characteristics. The main problem is the diversity of resource sharing mechanisms in different architectures, most notably the SMT [2]. OpenMP has a default scheme that is implementation specific, meaning that OpenMP implementations use different default policies to assign and interpret the internal control variables (ICVs). Additionally, in virtual machines, the number of threads is affected by the way different hypervisor present and assign processors and virtual cores to the virtual machine. The default scheme is static and tends to use the number of physical cores as the default number of threads. This does not exploit the hyper threading automatically and whether it is a good or bad decision depends on the interconnection, location, resource sharing policies and mechanisms of the processors involved as well as how the program utilizes them.

Having a tool that can automatically learn the formula for the optimal number of threads without previously gathering explicit user feedback on the program to be ran can significantly automate the process of producing better performing algorithms.

Initially, a suite of benchmarks is ran concurrently using pThreads. Using profiling to capture the state of the runtime while running the suite, an output dataset is created. This dataset is used as an input for the creation of random trees [3] (the forest tree algorithm [4]). This is a machine learning algorithm that uses a supervised approach to train a number of decision trees that all vote on the number of threads i.e classifying the current runtime state, designed in such a way as to avoid overfitting (generate bad predictions because the tree is too specific to a subset of all the possible inputs or runtime states).

In this paper we aim at testing the validity of the following hypotheses:

- H1 There exists a subset or multiple subsets of dataset attributes for which the predictions generated by the random forest/tool does not degrade performance in any of the test cases relative to the performance achieved by assigning the number of cores to the num_threads OpenMP variable.
- H2 There exists a subset or multiple subsets of dataset attributes for which the tool boosts performance in comparison with the default OpenMP scheme either by increasing or decreasing the number of threads.

Further details are disclosed in the following sections. Implementation details of the proposed approach are given in Section II. The experiments are described in Section III and results in Section IV. Relevant discussion is given in Section V, related work in Section VI and conclusions in Section VII.

II. Implementation Details

This section describes the approach used to optimize the number of threads for an OpenMP program.

Retrieval of the runtime parameters of an OpenMP program is done using the SIGAR API [5]. The SIGAR API is an open source cross-platform library that offers helps in parsing and locating major system parameters including CPU usage, ram usage, virtual memory IO traffic (total and by process) as well as more detailed information about the processes, the hardware

TABLE I
LIST OF ATTRIBUTES USED FOR PROFILING

num_threads	number of threads for one run
cpu_usage	cpu usage for all processes
pid_cpu_usage	cpu usage for the current process
ram_usage	ram usage for all processes
pid_ram_usage	ram usage for the current process
pid_page_faults	page faults for the current process
vm_usage	virtual memory usage for all processes
pid_vm_usage	virtual memory usage for the current process
pid_vm_read_usage	virtual memory reads percentage for the current process
pid_vm_write_usage	virtual memory writes percentage for the current process
processes	# of processes and threads created
procs_running	# of running processes and threads
procs_blocked	# of blocked processes and threads

features and the operating system running on the machine. An indicator for its quality is that it is a framework suitable for use in cloud environments for measuring hypervisor performance [6]. It offers an extensive set of utilities to monitor the state of a highly dynamic runtime which makes it a suitable tool for the kind of measurements needed for this tool. A sample of all potential dataset attributes is given in Table I.

PThreads is a an implementation of thread-level parallelism [7]. It is used to concurrently run a suite of OpenMP benchmarks. The benchmarks are ran for a range of threads [1, max_num_threads] constituting one run. This is done 50 times per benchmark by default. The optimal number of threads from the specified range is found using the best performance time yielded during each consecutive run. The corresponding runtime state is then written to disc.

The OpenMP algorithm which utilizes the tool has to call the omp_set_num_threads function before each parallel block to invoke the wrapper function. The number of threads is predicted using the random forest and used as a parameter in the new handle which is returned.

III. DESCRIPTION OF EXPERIMENTS

The goal of running the experiments is to simulate as many states of the runtime as possible so that a representative decision tree can be constructed out of the provided dataset. This is closely related to the parameters that are used to represent the state and is essential to be carefully configured so that performance boost can be obtained. Random delays are enforced in between runs in order to accomplish this better.

The dataset is boosted by generating additional dataset records using the existing dataset records with repetition. Basically the experiment is repeated a number of times, and this helps to break the independency, or decrease its negative impact on the decision trees that are to be generated out of the dataset by adding a variance to the output that helps remove some of the initial bias. This works for small biases only. Additionally, boosting enables us to get p-values and provides a good basis for running statistical tests and deriving conclusions about our data. Therefore, the parameters have to be very carefully selected. Additionally, random forest does not require any normalization of the dataset, since it considers each attribute independently [4].

Each forest is created the first time when the call to set_num_threads is made.

After the initial configuration, the forest is used to predict the number of threads for the parallel region in the OpenMP algorithm. The overall results are 50-150 obtained values of elapsed times in both cases by using the random forest for prediction, and by utilizing OpenMP default schemes.

TABLE II
EXPERIMENTAL ENVIRONMENT

Benchmark programs	Dijkstra, Multitask, Poisson [8]
Hardware	HP Probook 650 i5-4210M RAM 4GB
OS	Debian, GNU Linux 6.0

TABLE III
LIST OF ATTRIBUTES INCLUDED IN THE DATASETS

Dataset 1	all attributes listed in Table I
Dataset 2	num_threads, cpu_usage ,pid_cpu_usage, lst_cpu_usage, ram_usage, pid_page_faults, procs_blocked, user, sys, idle, irq
Dataset 3	num_threads,pid_cpu_usage,lst_cpu_usage, nice, idle, wait, irq, soft_irq
Dataset 4	num_threads, pid_ram_usage, ram_usage, pid_cpu_usage, cpu_usage, procs_blocked nice, idle, user, sys, wait, irq, soft_irq, lst_cpu_usage

The experimental environment is presented in Table II.

The experiments are based on using 4 subsets of the parameters listed in Table I. The corresponding parameters are listed in Table III.

The overall performance is analyzed in two environments. *No stress environment* is the one where the experiments are executed with no additional workload. A *stress environment* is simulated using the stress program package for POSIX systems that imposes stress on the system (CPU, memory, I/O, disk stress) by starting various number of threads concurrently to the experiments.

Three experiments are conducted as OpenMP default schemes and four more experiments using the tool with different dataset. To evaluate the speedup of using the tool and the default configuration we compare the corresponding elapsed times.

Algorithm I conducts exhaustive search to find a combination of input gates to produce logical circuit with output 1 [8]. It has a repetitive structure without complex data dependencies and exploits a can high degree of parallelism. *Algorithm II* is an implementation of the Fast Fourier Transformation (FFT) algorithm with a lot of data-flow dependencies which do not allow a high level of parallelism.

IV. RESULTS

A. Algorithm I in a No Stress environment

Fig. 1 presents the results (elapsed times) obtained with the default OpenMP scheme setting the number of threads variable to 1, 2 and 4. The X-axis represents the number of conducted

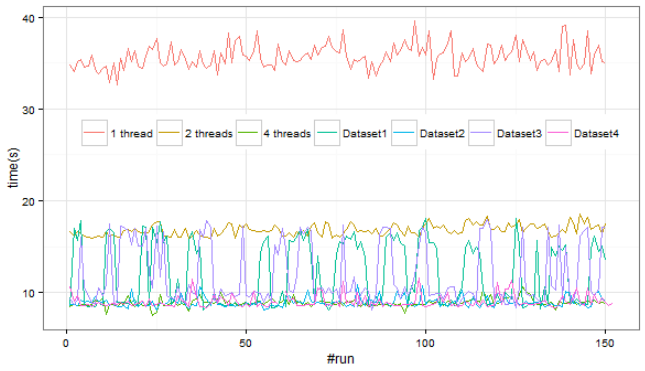

Fig. 1. Performance of executing the Algorithm I in a no stress environment

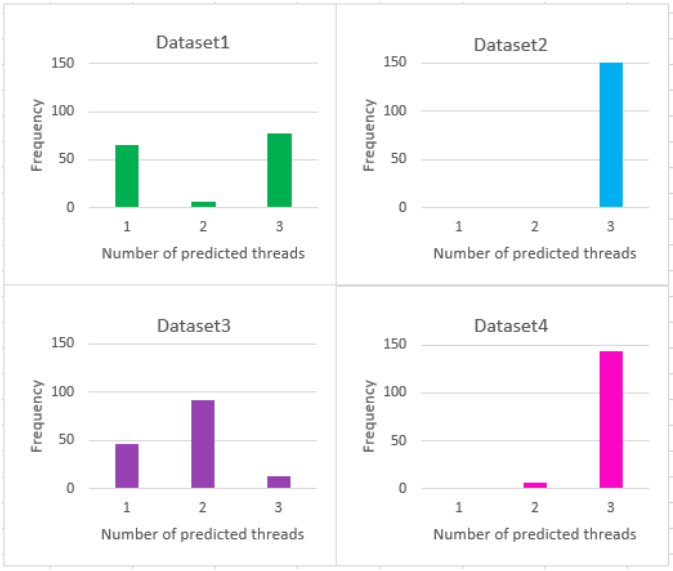

Fig. 2. Histograms for the 4 datasets for executing Algorithm I in a no stress environment.

test runs. Four separate measurements are conducted to evaluate the tool performance results. Each measurement uses a specific dataset defined in Table III to construct the forest. It is used to predict the number of threads, labeled in the graph with Dataset 1-4.

A more detailed information about the output number of predicted threads with the tool is presented in Fig. 2. A total of 150 test runs of Algorithm I is utilizing the same 4 forests used for the performance measurements in Fig. 1 labeled with Dataset 1-4 in the decision making.

The default OpenMP scheme yields best results with 4 threads (value of the OpenMP num_threads variable). Using this reference value for the optimal choice of the number of threads, Dataset 2 and Dataset 4 result in a forest that most accurately predicts the optimal number of threads. This is presented in Fig. 2 pointing out the 100% accuracy for Dataset 2 and 96% accuracy for Dataset 4.

For machines with the same hardware configuration that by default use the number of physical cores (2 in this case it is

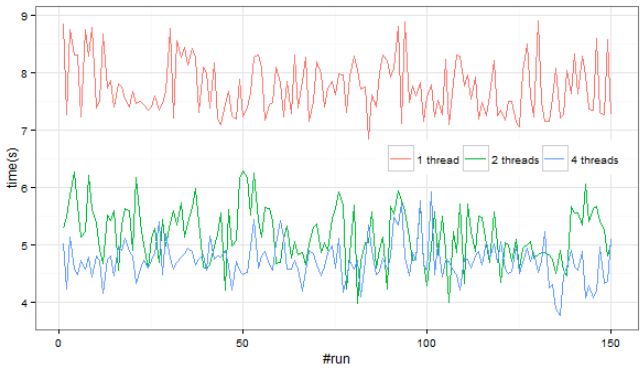

Fig. 3. Performance of executing the Algorithm II in a no stress environment with default OpenMP schemes

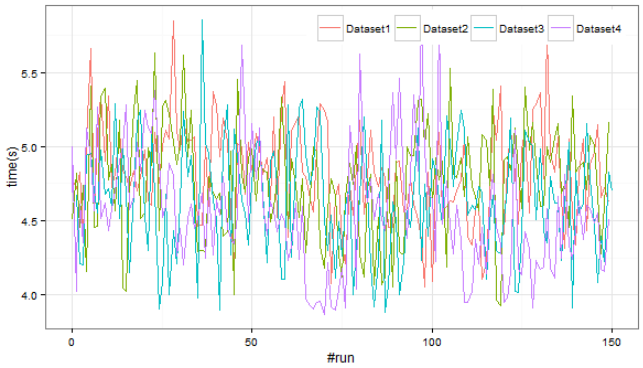

Fig. 4. Performance of executing the Algorithm II in a no stress environment using the tool with Datasets 1 - 4

2 Threads), the best speedup is $Sp(2T, D2)$ achieved using Dataset 2, with an average value of 1.876241, which confirms the Hypothesis H2.

Additionally, the predictions do not include a number of threads less than the physical number of cores nor for the best performing Dataset 2 nor for any of the other datasets, confirming the Hypothesis H1.

B. Algorithm II in a No Stress environment

The results obtained executing the Algorithm II using the OpenMP default scheme are presented in Fig. 3 and for executing the tool with Datasets 1 - 4 in Fig. 4

The histogram of predicted threads for executing the Algorithm II in a no stress environment is presented in Fig. 5.

The tool using Dataset 2 confirms the Hypothesis H2, boosting performance in comparison with the most common OpenMP default number of threads which corresponds to the number of cores, i.e 2, in the majority of the runs. It can be concluded that Dataset 4 results in slightly better performance according to the frequency of the Dataset 4 points that lie below Dataset2, which agrees with the minimum average out of these two datasets.

Dataset 3 and Dataset 4 out of the four datasets give the most accurate results according to the average elapsed

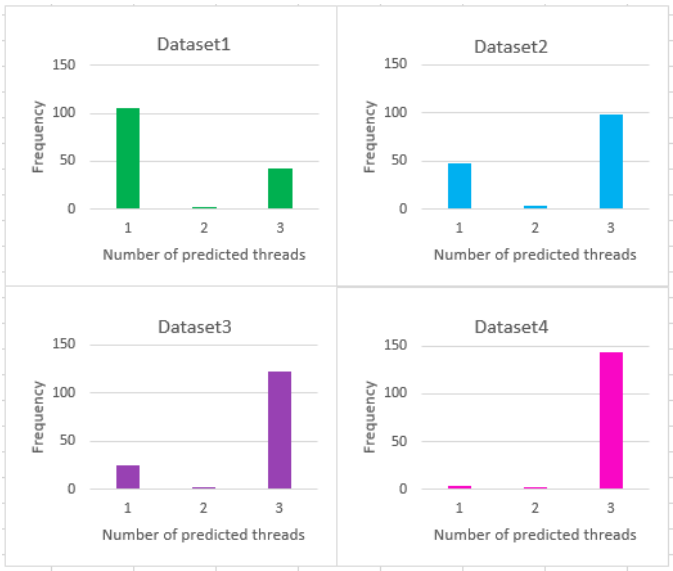

Fig. 5. Histograms of predicted threads for executing Algorithm II in a no stress environment.

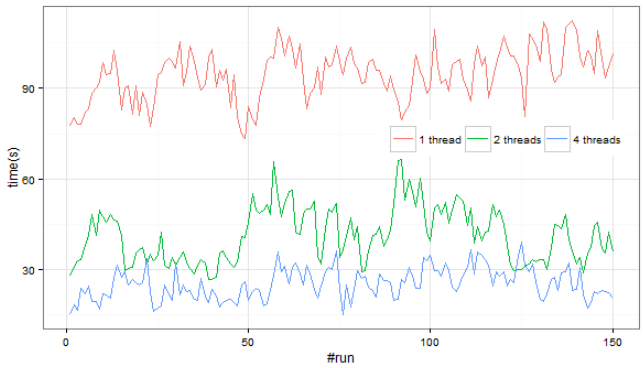

Fig. 6. Performance of executing the Algorithm I in a stress environment with OpenMP default schemes

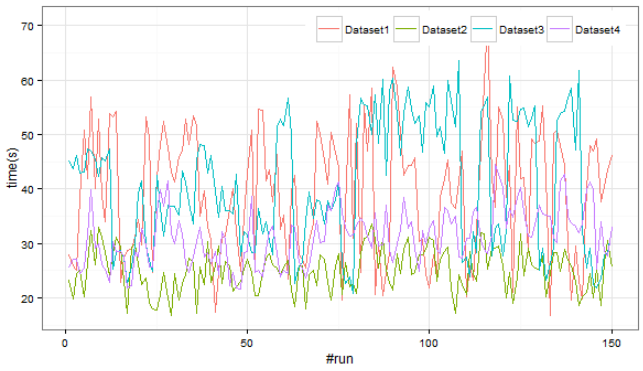

Fig. 7. Performance of executing the Algorithm I in a stress environment with Datasets 1 -4

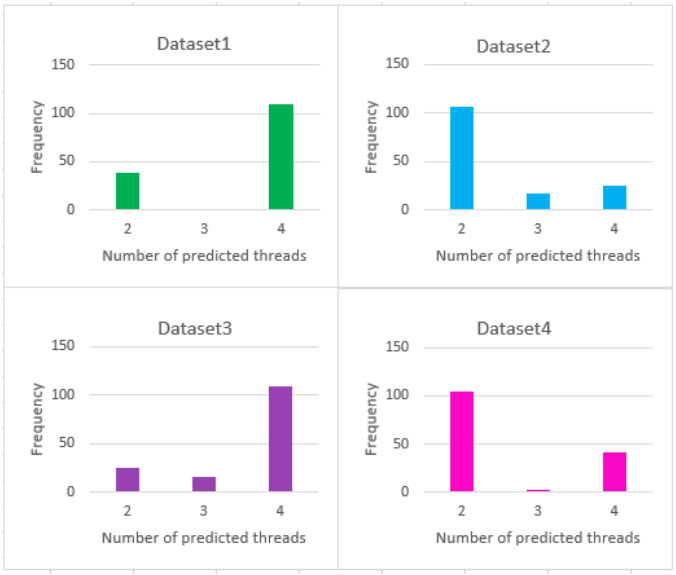

Fig. 8. Histogram of predicted threads for executing the Algorithm I in a stress environment

time. However, the average elapsed time for this algorithm is generally not a good indicator of the whether the predictions optimize the performance since the intervals [mean-sd, mean +sd] of the measurements listed in Table IV all overlap with each other except for 1 thread. Therefore the graphs along with the significant attributes should be analyzed in order to determine which dataset results in the best optimization of the algorithm.

Additionally, no dataset prediction degrades the performance, and results with a number of threads less than the number of physical cores Fig. 5.

C. Algorithm I in a Stress environment

The performances of executing the Algorithm I in stress environment are presented in Fig. 6 for default OpenMP schemes and in Fig. 7 for Datasets 1-4.

Fig. 8 presents the histogram of predicted threads for executing the Algorithm I in stress environment and contains information about the frequency associated with the predictions.

Algorithm I exploits the parallelism at a high degree and is accentuated enough so that in a stress environment using OpenMP default scheme, it is utilizing more cores. Eventually, this yields a better performance, except in very rare cases with some overlap. This overlap is located in the interval $[x - sd, x + sd]$ for a certain value of x as elapsed time. It is a case when the performance of executing the algorithm with a given dataset overlaps with the same interval for execution of an OpenMP scheme.

The execution of the Algorithm I with maximum number of threads in the OpenMP default scheme yields to the best performance, as presented in Table IV). The maximum number of threads is consequently used as a reference value when comparing the tool results with the OpenMP default scheme results.

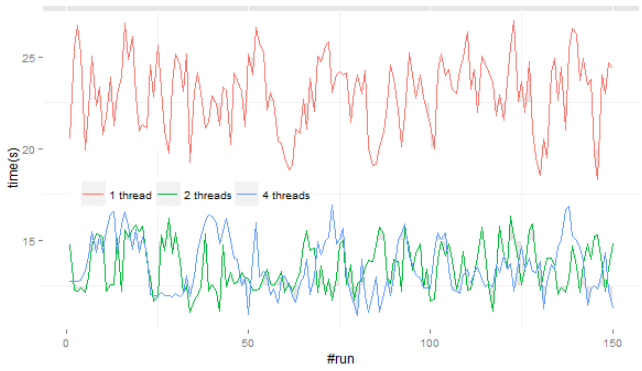

Fig. 9. Performance of executing the Algorithm II in a stress environment with default OpenMP schemes.

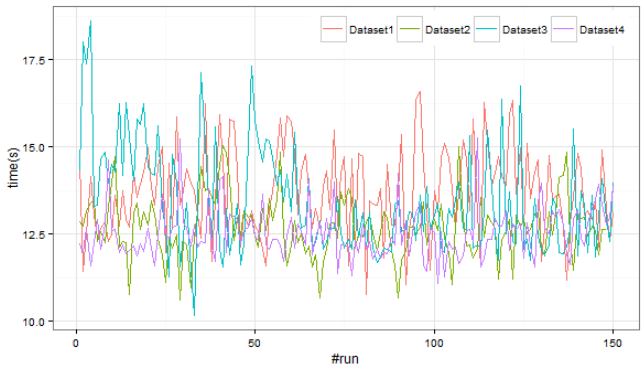

Fig. 10. Performance of executing the Algorithm II in a stress environment with Datasets 1-4.

D. Algorithm II in a Stress environment

Fig. 9 presents the performance of executing Algorithm II in a stress environment with OpenMP schemes and Fig. 10 with Datasets 1-4.

The best performance in the OpenMP schemes results in the configuration with 4 threads judging by the average. The majority of input states benefit more from 2 threads than 4 threads when comparing the overlaps shown in the performance graphs. This happens due to the variation. It is not a negligible percentage, so it should be considered when comparing the tool results with the OpenMP default scheme results. The reference optimal number of threads is therefore not uniquely determined, but is closely related to the runtime.

The results presented in Fig. 9 and Fig. 11 indicate that from the OpenMP default scheme, the optimal number of threads is leaning towards 4 threads. However the same as with Algorithm I in a stress environment, this preference will be reviewed more closely due to the higher variations of the results i.e the unpredictability caused by the the highly dynamic runtime state changes.

V. Discussion

Table IV presents the average values of elapsed times for executing the Algorithm I in a no stress environment (A1NS), Algorithm II in a no stress environment (A2NS), Algorithm I in a stress environment (A1S) and Algorithm II in a stress environment (A2S).

The obtained values of speedups for the experiments are displayed in Table V.

One can conclude that no dataset produces a speedup factor of less than 1 with the number of physical cores equal to the number of threads in (V). The exception is the Dataset 1 in stress environment with a value of 0.986805. Considering the standard deviation of the results, this value can be considered acceptable due to the stress environment. This evidence sup-

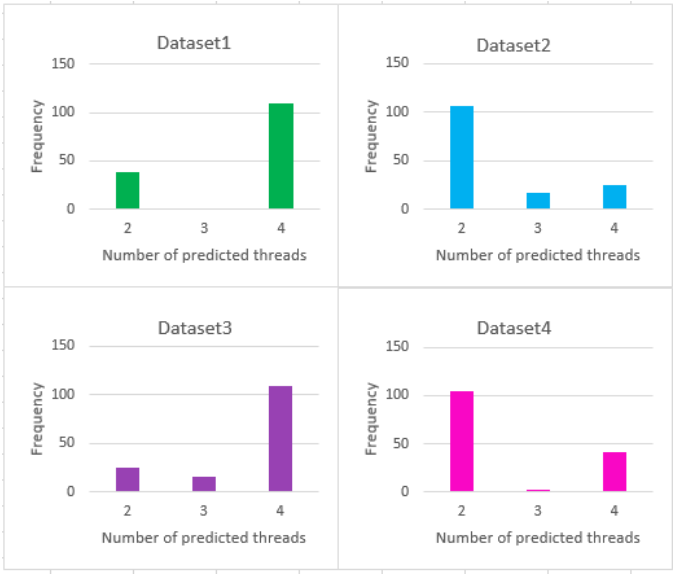

Fig. 11. Histogram of predicted threads for executing the Algorithm I in a stress environment

TABLE IV
AVERAGE ELAPSED TIMES FOR THE EXPERIMENTS

Test run	A1NS	A2NS	A1S	A2S
1 Thread	35.65783	7.752695	94.67928	22.99271
2 Threads	16.79859	5.192298	41.35630	13.52446
4 Threads	08.89994	4.724949	25.33419	13.58029
Dataset 1	11.95526	4.831201	39.43302	13.70530
Dataset 2	08.95332	4.770104	25.10371	12.70467
Dataset 3	12.16092	4.640983	40.83514	13.42115
Dataset 4	09.00934	4.556192	31.29540	12.54097

TABLE V
SPEEDUPS OBTAINED FOR THE EXPERIMENTS

	Test run	Dataset1	Dataset2	Dataset3	Dataset4
A1NS	1 Thread	2.982606	3.982636	2.932165	3.957874
	2 Thread	1.405121	1.876241	1.381358	1.864575
	4 Thread	0.744437	0.994037	0.731847	0.987857
A2NS	1 Thread	1.604713	1.625267	1.670486	1.701573
	2 Threads	1.074743	1.088508	1.118793	1.139614
	4 Threads	0.978007	0.990534	1.018092	1.037039
A1S	1 Thread	2.401015	3.771525	2.318574	3.025342
	2 Threads	1.048773	1.647418	1.012763	1.321481
	4 Threads	0.541023	0.849842	0.522447	0.681704
A2S	1 Thread	1.677651	1.809784	1.713170	1.833408
	2 Threads	0.986805	1.064527	1.007698	1.078422
	4 Threads	0.990874	1.068921	1.011857	1.082874

ports hypothesis H1 for all the datasets.

In summary, the performance evaluation yields the following conclusions. The speedups measured in the stress and no stress environment for both Algorithms I and II suggest that the predictions result in an optimized performance using either Dataset 2 or Dataset 4 for both Algorithms I and II. The indicators are the mean elapsed times combined with the conclusions made by analyzing the performance graphs that compare the results of executing the Datasets with the OpenMP default scheme results.

The maximum number of threads results in best performance for both the no stress and stress environments. Since Algorithm I is scaled vertically more successfully than Algorithm II stress testing does not result in a different optimal number of threads. The tool predictions that yield the best performance using the maximum number of threads are Dataset 2 and Dataset 4 following closely behind with slightly lower speedups. The optimization/performance boost is done by taking advantage of hyperthreading. The successful optimization using Dataset 2 and Dataset 4 confirms Hypothesis H2 for Algorithm I.

The optimal number of threads for Algorithm II in both the no stress and stress environment is not unique as in the previous case with Algorithm I. Therefore it requires more thorough graph analysis of the resulting performances. This analysis shows evidence that the tool succeeds in finding an optimal number of threads that varies from 2 to 4. Despite the high index of variance inevitably caused by the intensive stress testing, close examination of the graph/plots and the statistics of the datasets and the predictions, yields the same conclusion that Dataset 2 and Dataset 4 result again in a better performance than for the other datasets. This evidence confirms hypothesis H2 for Algorithm II.

VI. Related work

Among the research work done in area of adaptive or dynamic runtime OpenMP configuration, one of the most recent works, is based on creating a Multilayer Perception neural network with one hidden layer with back propagation to determine the number of threads for a parallel region based on the external workload [9].

Even though different machine algorithm, benchmarks and profiling tools are used, there are similarities between the goals set and methodologies used between this paper and [9]. The benefits of the approaches to optimization presented in both papers are mainly demonstrated with increase in the workload. A machine learning technique is used to predict the number of threads and is tested in an unknown setting.

Other similar work is done building up on [9] using other approaches other than neural networks in dynamic workload settings, such as reinforced learning and Markov decision processes for unsupervised learning [10]. Random forests are mentioned as a suggestion for another machine learning algorithm that can solve the problem.

Substantial amount of work is done in creating loop scheduling policies using runtime information. Other research work demonstrate promising results by using an adaptive loop scheduling strategy targeting SMT machine [11].

VII. Conclusion

The conducted experiments and performance analysis provide evidence that supports the correctness of hypotheses H1 and H2.

The main achievement is that the analysis tool provides a way to map the dynamic runtime state to the optimal number of threads in a real-time manner being able to respond to the changes in the runtime. The automation of this task becomes greater and greater advantage with increasing the number of processors and hyperthreading scale on a machine.

Therefore the significance of this paper is to explore a way of automating optimization in OpenMP and set up a groundwork for developing a truly generic cross-platform optimization tool that requires minimum configuration to produce good predictions on the optimal number of threads.

The results so far are encouraging, and the tool is able to correct the default scheme and find the optimal number of threads in cases of a highly dynamic runtime when the variation of the parameters is significant enough to provide indication of it.

References

[1] B. Barney. (2015, Aug) OpenMP. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/

[2] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos, "An evaluation of openmp on current and emerging multithreaded/multicore processors," in *OpenMP Shared Memory Parallel Programming*. Springer, 2008, pp. 133–144.

[3] OpenCV dev team. (2014, Nov) OpenCV documentation, Random Trees. [Online]. Available: http://docs.opencv.org/3.0-beta/modules/ml/doc/random_trees.html

[4] L. Breiman and A. Cutler. (2004) Random Forests. [Online]. Available: http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[5] R. Morgan and D. MacEachem. (2010, Dec) Sigar - system information gatherer and reporter. [Online]. Available: https://support.hyperic.com/display/SIGAR/Home

[6] P. V. V. Reddy and L. Rajamani, "Evaluation of different hypervisors performance in the private cloud with sigar framework," *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 2, 2014.

[7] B. Barney. (2015, Aug) POSIX Threads Programming. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[8] John Burkardt. (2011, May) C++ Examples of Parallel Programming with OpenMP. [Online]. Available: http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[9] M. K. Emani, Z. Wang, and M. F. O'Boyle, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 2013, pp. 1–10.

[10] M. K. Emani, "Adaptive parallelism mapping in dynamic environments using machine learning," Ph.D. dissertation, The University of Edinburgh, 2015.

[11] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss, "An adaptive OpenMP loop scheduler for hyperthreaded SMPs," in *ISCA PDCS*, 2004, pp. 256–263.