Do Language Constructs for Concurrent Execution Have Impact on Energy Efficiency?

Gustavo Pinto
Informatics Center, Federal University of Pernambuco
Recife, PE, Brazil
ghlp@cin.ufpe.br

Abstract

This study analyzed the performance and energy consumption of multicore applications, using three techniques to manage concurrent execution in a set of benchmarks. We conclude that these constructs can heavily impact on energy consumption. Nonetheless, the trade-off between performance and energy consumption in multicore applications is not so obvious.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming, Parallel Programming

Keywords Refactoring, Concurrent/Parallel Programming, Energy Consumption

1. The research problem and motivation

Measuring the energy consumption of an application and understanding where the energy usage lies provides new opportunities for energy savings. In order to understand the complexities of this approach, we specifically look at multi-threaded applications. The performance of the existing constructs for concurrent execution is reasonably well-understood [4, 7]. Furthermore, since parallel programming enables programmers to run their applications faster, it makes sense to assume that an application that finish earlier will also consume less energy [3]. This idea is known as the "Race to idle" principle [1]. In summary: faster programs will theoretically consume less energy because they will have the machine idle fast, and modern processors consume very little energy when idle. Nevertheless, parallelism and the overheads inherent to concurrent and parallel programming constructs might impact energy consumption in ways that are hard to predict.

This paper presents an empirical study consisting of the evaluation of the performance and energy consumption of four applications that use three programming concurrent constructs plus a sequential implementation with the goal of demonstrating that it is hard to establish a trade-offs between energy-efficiency and performance. This study is relevant because it is known that concurrent construct is often used in high-level applications [5]. Moreover, the performance of the existing constructs for concurrent execution is reasonably well-understood [7], but little is know about its energy efficiency.

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514880

2. Related work

To the best of our knowledge, only two studies have dealt with the topic of understanding the impact of concurrent constructs on the energy efficiency of applications [2, 6]. Gautham et al. [2] analyzed synchronization primitives and explore the following synchronization techniques to find an ideal solution for synchronization-intensive workloads: i) spin lock ii) mutexes iii) software transactional memory. They show that Software Transactional Memory (STM) systems can perform better than locks for workloads where a significant portion of the execution time is spent in the critical sections. Trefethen [6] studies the behavior of the NAS Benchmark suite for its energy and runtime performance. The benchmark suite considered includes I/O and compute-intensive applications. The authors concluded that there is a clear interaction between execution time and energy but this is not a simple relationship and can be affected by the computer environment and algorithmic approach used in the application. Nonetheless, none of these papers compare the energy-efficiency between techniques to manage concurrent execution.

3. Approach and uniqueness

To achieve this study's goal, we ran a set of benchmark applications from different domains in a number of different configurations while varying a group of attributes. We can divide these attributes in two groups: internal (programming language construct, number of threads in use and resource usage - CPU and/or IO) and external (the clock frequency and the JVM implementation). We then provided a set of variant implementations for each benchmark using three concurrent programming constructs, plus a sequential implementation[1].

In this experiment, we used commodity hardware: an Intel(R) Xeon(R), 2.13 GHz, 4 cores/8 threads and with 16Gb of memory, running Linux 64-bit, kernel 3.0.0.-31-server. These experiments were run using three JVMs: i) OpenJDK version 1.7.0_09, ii) HotSpot JDK version 1.7, and iii) JRockit version 1.6. When the experiments were performed using JRockit, we provided an external jar file containing the `ForkJoin` implementation.

Our experiments consisted of running these four applications in each of the three JVM implementations, scaling the CPU frequency, and limiting the number of threads in use. We ran each experiment twelve times for each workload, while measuring the system using the *powertop* utility[2]. We discarded the executions

[1] The implementation details are available at http://bit.ly/parallel-construct

[2] https://01.org/powertop/

	Energy Consumption (J)							
	N-Queens		LargestImage		Mandelbrot		Knucleotide	
	Median	SD	Median	SD	Median	SD	Median	SD
S	**732**	1.3	679	1.2	1978	1.2	5395	3.3
T	1023	2.2	766	3.8	1290	3.9	4808	4.7
E	984	2.1	**633**	3.7	**1287**	3.4	3281	3.6
FJ	753	1.8	749	4.5	1292	3.6	**2993**	3.9

	Time (s)							
	N-Queens		LargestImage		Mandelbrot		Knucleotide	
	Median	SD	Median	SD	Median	SD	Median	SD
S	85	0.7	78	1.2	71	1.2	106	0.6
T	44	1.2	42	1.9	41	2.1	68	1.3
E	38	1.1	**31**	1.8	41	1.9	62	1.1
FJ	**27**	1.1	54	3.8	**35**	1.3	**55**	1.4

Table 1. The comparison between the language constructs in terms of energy consumption and time. The obtained values for energy consumption and time are the medians of 10 executions.

with the lowest and the highest to reduce bias caused by outliers. We have two main metrics that evaluate the experiments: the energy consumed (in Joules) and the execution time (in seconds).

4. Results

Table 1 shows the overall view of our experimental results. The column named "Energy Consumption (J)" organizes the results of the 10 executions. The value contained in each cell represents the median of the 10 executions, where each sample represents the total of energy consumed in this given execution. We use **boldface** to highlight the best result for the given benchmark application. The column "Time (s)" works similarly.

As Table 1 shows, we can notice that the results of the concurrent constructs can have significant differences. For instance, the `Thread` results are almost always more inefficient in terms of both consumption and performance, when compared to `Executors`. Thus, since the usage of `Thread` and the `Executors` is very similar, with a little effort, a programmer could use `Executors`. Table 1 also shows that the same technique can have different impact on the energy consumed. The `ForkJoin` variant exhibited the best performance for all the benchmarks with the exception of LargestImage, which is strongly IO-bound. It also exhibits the lowest consumption for Knucleotide and reasonably low consumption for N-Queens and Mandelbrot. On the other hand, considering the LargestImage application, we notice that the `ForkJoin` variant consumed more energy than every other variant except for the one using threads. Nonetheless, LargestImage is a benchmark application that makes heavy use of IO operations, and it is well know that `ForkJoin` is not adequate for this kind of computation. This fact is the probable reason for the poor energy consumption.

Moreover, it is interesting to note that the `Sequential` variant of the N-Queens benchmark application presented the best energy consumption result, in spite of presenting the worst performance. This benchmark's result can vary according to the input data (in case, the NxN size of the matrix). In this experiments, we realized that, for a small matrix, the overhead caused by the thread creation led to an increase in energy consumption. But, for example, if we doubled the matrix size, the `ForkJoin` variant becomes the most energy efficient as well.

Furthermore, we analyzed how the benchmark applications scale with respect to the number of threads. Both the Mandelbrot and Knucleotide benchmark applications scale well. This means that the more cores available, the faster the applications run, and more energy is saved. Nonetheless, for the other ones, it is not true. For instance, in the LargestImage benchmark, the more cores we have, more inefficient the `ForkJoin` variant is, in terms of both performance and energy. In summary, we collected a total of 128 samples (4 benchmarks x 4 variants x 4 nr. of threads x 2 clock frequencies), and for 36 of those, the variation which achieve the best performance were not the same that consume less energy.

We then repeated the experiments varying the CPU frequency from 1.2 GHz to 2.13 GHz. We notice that even after reducing the clock frequencies, the results seems to be very similar to the ones with the higher frequency, in terms of the better technique remaining the better. However, for the LargestImage benchmark, we found out that the lowest frequency consumes the same amount of energy as a middle clock frequency. It is interesting and acceptable, since it is an application that does not use a huge amount of CPU. This is something that requires further investigation.

Finally, we have also analyzed whether different JVMs had different impacts on performance and energy consumption. We observed that, in general, results are very similar, specially for OpenJDK and HotSpot. It does not surprise us, since the HotSpot is the primary reference implementation of JVM, and OpenJDK is heavily inspired by them. On the other side, the JRockit JVM presents the worst case scenario, for all variants. For example, taking into consideration the `Thread` variant, the results increased in more than 10%. For the other benchmark applications, JRockit also exhibited the worst results among the JVMs. Although the different JVMs did affect execution time and energy consumption, similarly to different clock frequencies, they did not significantly change the behavior of the variants, e.g. the fastest and slowest variants in one JVM were the fastest and lowest variants for all of them.

4.1 Contributions

This paper presented the following contributions:

- Different techniques for concurrent/parallel programming within the same language (Java, in this case) can impact both performance and energy consumption in very different ways for applications with different characteristics.
- For concurrent software, the Race to Idle principle is often not true. In fact, we found at least a few examples that go in the opposite direction.
- Some factors, such as clock frequency and different VMs, do not significantly affect the relationship between performance and energy consumption.

References

[1] S. Albers and A. Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1266–1285. SIAM, 2012.

[2] A. Gautham et al. The implications of shared data synchronization techniques on multi-core energy efficiency. HotPower'12, Berkeley, CA, USA, 2012.

[3] J. Jelschen et al. Towards applying reengineering services to energy-efficient applications. In *CSMR*, pages 353–358, 2012.

[4] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 8–8, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X.

[5] W. Torres et al. Are java programmers transitioning to multicore?: a large scale study of java floss. SPLASH '11 Workshops, pages 123–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0.

[6] A. Trefethen and J. Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 1(0):–, 2013. ISSN 1877-7503. .

[7] W. Zhu, J. del Cuvillo, and G. R. Gao. Performance characteristics of openmp language constructs on a many-core-on-a-chip architecture. In *IWOMP*, pages 230–241, 2006.