Exploring the Energy Consumption of Highly Parallel Software on Windows

Mads Hjuler Kusk*, Jeppe Jon Holt*
and Jamie Baldwin Pedersen*
Department of Computer Science, Aalborg University, Denmark
*{mkusk18, jholt18, jjbp18}@student.aau.dk

March 6, 2023

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

1 Introduction

Research questions:

- What is the best measuring instrument for Windows?
- How does parallelism affect the energy consumption
- How does P-cores and E-cores affect the execution of parallelism in a process, versus only P-Cores?

2 Related Work

2.1 Previous Work

This paper builds upon the knowledge gathered in our previous work *"A Comparison Study of Measuring Instruments"*[1] where different measuring instruments were compared to explore whether a viable software-based measuring instrument was available for Windows. It was found that Intel Power Gadget (IPG) and Libre Hardware Monitor (LHM) on Windows have similar correlation to hardware-based measuring instruments as Intel's Running Average Power Limit (RAPL) has on Linux. This chapter builds upon the related work chapter of the previous work and as such will not be repeated, however, it will be expanded upon.

2.2 Parallel Software

In [2], the energy consumption for sequential and parallel genetic algorithms was explored, where one research question aims to explore the impact on energy consumption when using different numbers of cores. It found that a larger number of cores in the execution pool results in a lower running time and energy consumption, and conclude that parallelism can help reduce energy consumption. Parallelism's ability to reduce energy consumption was argued to be due to the large number of cores working to solve the problem simultaneously, where the combination of more cores, and more parallel operations per time unit will require less energy. When considering parallel software, it also found asynchronous implementations to use less energy, because there are no idle cores waiting for data in asynchronous implementations, while in synchronous implementations cores can be blocked during runtime, while waiting for responses from other cores.

In [3], three different thread management constructs from Java are explored and analyzed regarding energy consumption. It found that as the number of threads increased, the energy consumption would do the same. This was however only to a certain point, where from this point the energy consumption would start to decrease as the number of threads started to approach the number of cores in the CPU. However, the peak of this energy consumption was application dependent.It also found that in eight out of nine benchmarks, there was a decrease in execution time going from sequential execution on one thread to using multiple threads. However, it should be noted that four of their benchmarks are embarrassingly parallel whereas only one was embarrassingly serial. It should also be

noted that decreased execution time does not necessarily mean a decreased energy consumption, because in six out of nine benchmarks the lowest energy consumption was found in the sequential version using one thread. Furthermore, it investigated the energy-performance trade-off using the Energy-Delay-Product (EDP), which was the product between energy consumption and execution time. Using EDP it generally found parallel execution to be more favorable however depending on the benchmark increasing the number of threads may not be aligned with an improvement of EDP.[3]

In [4], the behavior of parallel applications and the relationship between execution time and energy consumption are explored. It tests four different language constructs which can be used to implement parallelism in C#. Furthermore, it uses varying amounts of threads and a sample of micro- and macro-benchmarks. It found that workload size has a large influence on running time and energy efficiency and that a certain limit must be reached before improvements can be observed when changing a sequential program into a parallel one. Additionally, it was found that execution time and energy consumption of parallel benchmarks do not always correlate. Comparing micro- and macro-benchmarks the findings remain consistent, although the impact becomes low for the macrobenchmarks due to an overall larger energy consumption. Furthermore, it has included some recommendations, which should be considered:[4]

- Shield cores: Avoid unintended threads running on the cores used in the benchmarking
- PowerUp: Can be used to ensure that benchmark is not optimized away during compilation
- Static clock: Make the clock rate of the CPU as static as possible
- Interrupt request: Avoid interrupt requests being sent to cores used in the benchmarking
- Turn off CPU turbo boost
- Turn off hyperthreading

The paper [5] presents a program that solves Laplace equations and compares the observed speed up for computing the Laplace equations with one, two and four cores, with estimates given by Amdahl's law and Gustafson's law. Amdahl's law describes the maximum potential speedup that can be achieved through parallelization of an algorithm based on the proportion of the algorithm that can be parallelized and the number of cores used. Gustafson's law evaluates the speedup of a parallel program based on the size of the problem and the number of cores. Unlike Amdahl's law which assumes a fixed problem size and a fixed proportion of the program that can be parallelized, Gustafson's law take into account that larger problems can be solved when more core are available and that the parallelization of a program can scale with the problem size. Comparing the observed speed up and the estimates it is adamant that Gustafson's law is more optimistic than Amdahl's law, however both estimates smaller speed ups than the observed speed up on two and four cores. [5]

2.3 Compilers

In [6] the language C++ and different compilers are explored and compared to find the impact of using different coding styles and compilers, where the goal is to find a balance between performance and energy efficiency. The different coding styles introduced explore the impact of splitting CPU and IO operations and interrupting the CPU-intensive instructions with sleep statements. The C++ compilers used in [6] include MinGW GCC, Cygwin GCC, Borland C++, and Visual C++, and the energy measurements are performed using Windows Performance Analyses (WPA). All compilers are used with default settings, and no optimizations were chosen. This decision was made based on works like [7], where it was found how mainstream compilers will apply multiple optimizations to the final code, where these optimizations in the worst case will result in worse performance and increased energy consumption. The issue of optimizations being very machine dependent was also shown in [8], where analysis and optimizations were done on a Texas Instruments C6200 DSP CPU. In [8] it was found that a large portion of the energy is used by fetching instructions which was addressed by introducing a fetch packet mechanism, and also find loop-unrolling to reduce energy consumption. While these optimizations decrease the energy consumption for the Texas Instruments C6200 DSP CPU, they note that for other CPUs varying results are expected. A similar conclusion is also found in [6], where they find that when choosing a compiler and coding style the energy reduction depends on the nature of the target machine and application. Based on the test case used, this being an election sort algorithm, they find the best performance with the Borland compiler, and the lowest energy with the Visual C++ compiler. When considering the coding styles, they find that both separating IO and CPU operations and interrupting the CPU-intensive instructions with sleep statements also decrease the energy consumption.

3 Background

3.1 CPU States

This section provides an overview of CPU-states, which are a crucial aspect of energy management in

computer systems. The information presented in this section draws heavily from Intel [9] and the further explanation from HardwareSecrets [10]. The concept of CPU-states is concerned with how a system manages its energy consumption during different operational conditions.

The C-states are a crucial aspect of CPU-states, as they dictate the extent to which a system shuts down various components of the CPU to conserve energy. The C0 state represents the normal operation of a computer under load. As the system moves from C0 to C10, progressively more components of the CPU are shut down until, in C10, the CPU is almost entirely inactive. It is important to note that the number of C-states supported may vary depending on the CPU and motherboard in use.

In addition to C-states, there are other states such as CC-states (Core C-states), PC-states (Package C-states), Thread C-states, and Hyper-Thread C-states. However, information on these states is limited. Enhanced C-states (C1E), which can shut down more components of the CPU than C0 but not as much as the next C-state, are also present in some CPUs.

P-states are used only during C0 state and determine the frequency of the CPU under load, thereby managing its energy consumption. S-states (Sleep State), on the other hand, control how the system uses energy on a larger scale by determining whether the system is sleeping or not. All C-states occur within S0, with deeper states of sleep such as Sleep and Hibernation being defined by increments.

The G-states (Global-States) define the overall state of the system. G0 represents a working computer where C-states, P-states, and S0 states can occur, while G3 represents a completely shut-down system.

In the context of the present study, C-states can largely effect the results of the experiments.

3.2 CPU Affinity

Affinity is a feature in operating systems(OSs) that enables processes to be bound to specific cores in a multi-core processor. In OSs, jobs and threads are constantly rescheduled for optimal system performance, which means that the same process can be assigned to different cores of the CPU. Processor affinity allows applications to bind or unbind a process to a specific core or range of cores/CPUs. When a process is pinned to a core, the OS ensures it only executes on the assigned core(s) or CPU(s) each time it is scheduled.[11]

```
void ExecuteWithAffinity(string path)
{
    var process = new Process();
    process.StartInfo.FileName = path
    process.Start();

    // Set affinity for the process
    process.ProcessorAffinity =
        new IntPtr(0b0000_0011)
}
```

Listing 1: An example of how to set affinity for a process in C#

Processor affinity is particularly useful for scaling performance on multi-core processor architectures that share the same global memory and have local caches referred to as the Uniform memory access architecture. Processor affinity is also useful for this study, as this allows the framework to assign a single or a set of cores and threads to a process.[11]

An example of how affinity can be set for a process in C# can be seen in Listing **1**. When setting the affinity for the processor, the input is a bitmask, where each bit represents a CPU core. In Listing **1**, the process is thus allowed to execute in core #0 and #1.

3.3 Scheduling Priority

When scheduling threads on Windows, it is done based on each thread's scheduling priority level and the priority class of the process. For the priority class of the process, the value can be either `IDLE`, `BELOW NORMAL NORMAL`, `ABOVE NORMAL`, `HIGH` or `REALTIME`, where the default is `NORMAL`. It is noted that `HIGH` priority should be used with care, as other threads in the system will not get any processor time while that process is running. If a process needs `HIGH` priority, it is recommended to raise the priority class temporarily. For the `REALTIME` priority class, this should only be used for applications that "talk" to hardware directly, as this class will interrupt threads managing mouse input, keyboard inputs, ect.[12]

For the priority level, the levels can be either `IDLE`, `LOWEST`, `BELOW NORMAL`, `NORMAL`, `ABOVE NORMAL`, `HIGHEST` and `TIME CRITICAL`, where the default is `NORMAL`. A typical strategy is to increase the level of the input threads for applications to ensure they are responsive, and to decrease the level for background processes, meaning they can be interrupted as needed.[12]

The scheduling priority is assigned to each thread as a value from zero to 31, where this value is called the base priority. The base priority is decided using both the thread priority level and the priority class and can be found in [12]. When assigning a base priority where both the priority class and thread priority are the default values, e.i.`NORMAL`, the base priority is 8.[12]

The idea of having different priorities is to treat threads with the same priority equally, by assigning time slices to each thread in a round-robin fashion, starting with the highest priority. In the case of none of the highest priority threads being ready to run, the lower priority threads will be assigned time slices. The

lower-priority threads will then execute until a higher-priority thread is available, in which case the system will assign a full time slice to the thread, and stop executing the lower-priority threads, without time to finish using its time slice.[12]

```
void ExecuteWithPriority(string path)
{
  var process = new Process();
  process.StartInfo.FileName = path
  process.Start();

  // Set priority class for process
  process.PriorityClass =
    ProcessPriorityClass.High;

   // Set priority level for threads
  foreach (var t in process.Threads)
  {
    thread.PriorityLevel =
      ThreadPriorityLevel.Highest;
  }
}
```

Listing 2: An example of how to set priorities for a process in C#

A thing to note is that when setting priority class and priority level for a process through C#, the priority class is supported for both Windows and Linux, while the priority level is only supported for Windows. An example of how both the priority class and priority level can be set for a process and its threads can be seen in Listing **2**

3.4 OpenMP

OpenMP (Open Multi-Processing) is a parallel programming API consisting of a set of compiler directives and runtime library routines, with support for multiple platforms like Linux, MacOs and Windows and and multiple compilers like GCC, LLVM/Clan and Intels OpenApi. OpenMP allows programmers to write parallel code for multi-core CPUs and GPUs.[13]

The directives provides a way to specify parallelism among multiple threads of execution within a single program, while the library provide mechanisms for managing threads and data synchronization. When using OpenMP programmers can write parallel codes and take advantage of multiple processors without having to deal with low level details.[13]

When executing using OpenMP, the parallel mode used is called the Fork-Join Execution Model. This model works by firstly executing the program with a single thread, called the master thread. This thread is executed serially until parallel regions are encountered, in which case a thread group is created, consisting of the master thread, and additional worker threads. This process is called fork. After splitting up, each thread will execute until an implicit barrier at the end of the parallel region. When all threads has reached this barrier, only the master thread continues.[13]

```
#pragma omp directive-name [
    clause[ [,] clause]...
    ]
```

Listing 3: The basic format of OpenMP directive in C/C++

When using OpenMP, the parallel regions are identified using a series of directives and clauses, where the basic format can be seen in Listing **3**. By default, the parallel regions are executed using the number of present threads in the system, but this can also be specified using `num_threads(x)`, where `x` represents the number of threads.[13]

4 Method

4.1 Measuring Instruments

This section present the different measuring instruments utilized in our work. The measuring instruments utilized in our previous work will only be briefly introduced, however more detail can be found in [1]. In this paper, four software-based measuring instruments and one hardware-based measuring instrument is used, where the hardware-based measuring instrument represents the ground truth.

Running Average Power Limit Intel's RAPL is a commonly used software-based measuring instrument seen in the literature.[1] It uses model-specific-registers (MSRs) and Hardware performance counters to calculate how much energy the processor uses. The MSRs RAPL uses include *MSR_PKG_ENERGY_STATUS*, *MSR_DRAM_ENERGY_STATUS*, *MSR_PP0_ENERGY_STATUS* and *MSR_PP1_ENERGY_STATUS*. Which corresponds to the power domains, PKG, DRAM, PP0, and PP1 which are explained in [1]. RAPL has previously only been directly accessible on Linux and Mac. In [1] we found that RAPL had a high correlation of 0.81 with our ground truth on Linux.[1]

Intel Power Gadget IPG is a software tool created by Intel, which can estimate the power of Intel processors. It contains a command line version called Powerlog which allows accessing the energy consumption using callable APIs. It uses the same hardware counters and MSRs as RAPL[14], therefore it is expected to observe similar measurements to that of RAPL. Which is also shown in [1] where we found that IPG had a high correlation of 0.78 with our ground truth on Windows. We also found that IPG had a high correlation of 0.83 with RAPL, although the measurements is on different operating systems.[1]

Libre Hardware Monitor LHM[15] is a fork of Open Hardware Monitor, where the difference is that LHM

does not have a UI. Both projects are open source. LHM can use the same hardware counters and MSRs as RAPL and IPG and as such can measure the power domains PKG, DRAM, PP0, and PP1. Since it uses the methods to read energy consumption, a similar measurement is expected between LHM and IPG. We found that LHM correlated 0.76 with our ground truth on Windows. LHM was also found to have a high correlation of 0.85 with IPG.[1]

AC Current Clamp Serving as our ground truth measurement is our hardware-based measuring setup which is comprised of an MN60 AC clamp that is connected to the phase of the wire that goes into the PSU. It is also connected to an Analog Discovery 2 which is used as an oscilloscope which in turn is then connected to a Raspberry Pi 4. This setup allows us to continuously measure and log our data. For more detail see [1].

Scaphandre One measuring instrument not used in our previous work is Scaphandre[16]. Scaphandre is described as a monitoring agent which can measure energy consumption and is made for Linux where it can use Powercap RAPL which is a Linux kernel subsystem where data can be read from RAPL. It also has the functionality of measuring the energy consumption of some virtual machines, specifically Qemu and KVM hypervisors. A driver also exists which allows for installing RAPL on Windows.[17] Doing so allows using Scaphandre on a Windows computer where the sensor is RAPL which is utilizing the MSRs to update its counters. The Windows version of Scaphandre has some limitations but is able to report the energy consumption of the power domain PKG, using the MSR *MSR_PKG_ENERGY_STATUS*. Furthermore, it can also give an estimation of the energy consumption for individual processes. It does so by storing CPU usage statistics alongside the values of the energy counters. Then it is able to calculate the ratio of the CPU time for each Process ID (PID). With the calculated ratio a new calculation is made to get the subset of the energy consumption which is estimated to belong to a specific PID. A Linux exclusive feature is that the monitoring system Prometheus can be used with Scaphandre to get the energy consumption of an application which consist of several PIDs.

4.2 Statistics

Sample size: We need to know if we have enough measurements to be within our desired margin of error and margin of error. In [1] we used Cochran's formula to calculate how many measurement were needed. However after doing the Shapiro-Wilk test we discovered that some of our test case measurement do not follow a normal distribution. Therefore Cochran's formula should not be used. We are therefore going to take another approach in this work. There is still going to be an initial experiment, were 30 measurements will be collected which will then be used to decide whether more measurements are needed. Then we need to decide how to determine if 30 measurements is good enough.

Margin of error: As the sample size increases, the Central limit theorem (CLT) states that the distribution of sample means approaches a normal distribution, irrespective of the underlying distribution of the population. Therefore we can calculate the margin of error using the z-score to determine whether we are within our desired margin of error. If we are not, then further samples should be acquired.

Bootstraping:

5 Experiments

6 Results

7 Discussion

8 Conclusion

Acknowledgements

9 Future Works

References

1. Holt, J., Kusk, M. H. & Pedersen, J. B. *A Comparison Study of Measuring Instruments* (Aalborg University Department of Computer Science, 2023).
2. Abdelhafez, A., Alba, E. & Luque, G. A component-based study of energy consumption for sequential and parallel genetic algorithms. *The Journal of Supercomputing* **75,** 1–26 (Oct. 2019).
3. Pinto, G., Castor, F. & Liu, Y. D. Understanding Energy Behaviors of Thread Management Constructs. *SIGPLAN Not.* **49,** 345–360. ISSN: 0362-1340. https://doi.org/10.1145/2714064.2660235 (Oct. 2014).
4. Lindholt, R. S., Jepsen, K. & Nielsen, A. Ø. *Analyzing C# Energy Efficiency of Concurrency and Language Construct Combinations* (Aalborg University Department of Computer Science, 2022).
5. Prinslow, G. Overview of performance measurement and analytical modeling techniques for multi-core processors. *UR L: http://www. cs. wustl. edu/˜ jain/cse567-11/ftp/multcore* (2011).
6. Hassan, H., Moussa, A. & Farag, I. Performance vs. Power and Energy Consumption: Impact of Coding Style and Compiler. *International Journal of Advanced Computer Science and Applications* **8** (Dec. 2017).
7. Lima, E., Xavier, T., Faustino, A. & Ruiz, L. *Compiling for performance and power efficiency* in (Sept. 2013), 142–149.
8. Cooper, K. & Waterman, T. Understanding Energy Consumption on the C62x (Jan. 2004).
9. Intel. *https://www.intel.com/content/dam/develop/external/us/en/doc hill-sw-20-185393.pdf* https://www.intel.com/content/dam/develop/external/us/en/documents/green-hill-sw-20-185393.pdf. 2011.
10. hardwaresecrets. *Everything You Need to Know About the CPU Power Management* https://hardwaresecrets.com/everything-you-need-to-know-about-the-cpu-c-states-power-saving-modes/. 2023.
11. *1.3.1. processor affinity or CPU pinning* https://www.intel.com/content/www/us/en/docs/programmable/683013/current/processor-affinity-or-cpu-pinning.html. 03/03/2023.
12. Karl-Bridge-Microsoft. *Scheduling priorities - win32 apps* https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities. 03/03/2023.
13. Michael Womack, A. W. *Parallel Programming and Performance Optimization With OpenMP* https://passlab.github.io/OpenMPProgrammingBook/cover.html. 03/03/2023.
14. Mozilla. *tools/power/rapl* https://firefox-source-docs.mozilla.org/performance/tools_power_rapl.html. 24/02/2023.
15. source, O. *Libre Hardware Monitor* https://github.com/LibreHardwareMonitor/LibreHardwareMonitor. 03/03/2023.
16. Hubblo. *Scaphandre* https://github.com/hubblo-org/scaphandre. 23/02/2023.
17. Hubblo. *windows-rapl-driver* https://github.com/hubblo-org/windows-rapl-driver. 23/02/2023.