Understanding computation time:
A critical discussion of time as a computational performance metric

David Harris-Birtill[1] and Rose Harris-Birtill[2,3]

[1] *School of Computer Science, University of St Andrews, UK*
[2] *School of English, University of St Andrews, UK*
[3] *Open Library of Humanities, Birkbeck, University of London, UK*

Abstract

Computation time is an important performance metric that scientists and software engineers use to determine whether an algorithm is capable of running within a reasonable time frame. We provide an accessible critical review of the factors that influence computation time, highlighting problems in its reporting in current research and the negative practical impact that this has on developers, recommending best practice for its measurement and reporting. Discussing how computers and coders measure time, a discrepancy is exposed between best practice in the primarily theoretical field of computational complexity, and the difficulty for non-specialists in applying such theoretical findings. We therefore recommend establishing a better reporting practice, highlighting future work needed to expose the effects of poor reporting. Freely shareable templates are provided to help developers and researchers report this information more accurately, helping others to build upon their work, and thereby reducing the needless global duplication of computational and human effort.

Keywords: *time; computation time; performance metrics; time template; reporting time; execution time; response time; computation time template; measuring time*

1 Introduction: why computation time matters

This paper seeks to provide a non-specialist introduction to computation time - the time that a computational process or program takes to run - and to review the ongoing problems in the way that computation time is reported in current research, demonstrating the negative impact that this can have on developers as they seek to optimise their code. After a review of current problems in the field and their practical implications, recommendations of best practice when measuring and reporting computation time - including freely shareable templates - are also provided in order to avoid perpetuating these critical issues, and make these recommendations easier to implement.

The time that it takes for a computer algorithm to finish its task (computation time) is a key performance metric that scientists and software engineers use to determine whether an algorithm is capable of completing its task in a reasonable time frame. However, hardware capability is rapidly increasing, and new techniques are being developed that combine software and hardware to quickly solve problems of increasing complexity. This means that in practice, the supposedly fixed amount of time that it takes one researcher's computer to complete a task will often be vastly different for another. Software developers are also frequently unable to build on the work of others. Poor reporting practices mean that there is currently a substantial duplication of effort required to evaluate if an algorithm will be suitable for an intended task. It is to this end that this paper offers a practical suggestion to help combat this needless duplication of effort, and its human, financial and environmental costs.

When writing or using any computer program, it is important to first understand how long a process or program would take to run; this time frame is known as the computation time of the task. This temporal aspect to computing is an inherent factor that is often taken for granted by the end user; even the fastest programs take time to calculate across several steps, running on the central processing unit (CPU) or graphical processing unit (GPU), along with the other hardware components, each of which has its own individual frequency at which it is able to update variables and calculate the steps needed to complete the process. For example, when a Sat-Nav, or satellite navigation mapping system, calculates the fastest route from one location to another, the software has to calculate multiple different routes, taking a number of computational steps. The time that the program takes to provide a final mapped route back to the user may be imperceptible (i.e., a few milliseconds), or a noticeable amount of time (i.e., several seconds), depending on the complexity of the route (see Delling et al. 2009 for an example of this in practice in route calculation). Critically, for a system to be useful to a user, not only must it perform the required task successfully, but the computation time from initiating the task to receiving the end result (also known as the response time) needs to be within a reasonable time frame for a busy software user.

This reasonable time frame can vary depending on the task, and the environment that task is being conducted in. For simplicity, we can arbitrarily split these types of tasks into three groups: “real-time”, “quick decision”, and “worth-the-wait”. For a real-time task the computation time needs to be faster than we can notice; for example, for a video feed this would be less than the frame rate of a camera or screen (approximately less than 1/30th second, as noted in Wu, Houben, and Marquardt 2017), and this is particularly important in gaming and real-time video analysis. A “quick decision” (approximately less than a few seconds) is where the user may be prepared to wait a short time to get an answer to a question, for example, in an initial GPS route calculation, when a short delay is not a problem for the user (see Delling et al. 2009 and Delling et al. 2015 for discussions on calculating route complexity). Finally, there is the “worth the wait” timescale, which can be a few minutes (as demonstrated in Kong et al. 2017; Huynh, Li, and Giger 2016), hours (as demonstrated by Wahab, Khan, and Lee 2017; Wang et al. 2014), days (as demonstrated by Melinsčak, Prentašić, and Lončarić 2015; Cheng et al. 2017; Mok and Chung 2018) or even months (as demonstrated by Vo, Jacobs, and Hays 2017; Luong and Manning 2016).[1] This is where the system may be solving a very complex problem, such as when training a machine learning algorithm on a very large dataset (for example, see Vo, Jacobs, and Hays 2017 and Luong and Manning 2016). Accurate knowledge of the computation time - both predicted and measured - and the ability for the system to adhere to this “reasonable time frame” are both important; if a system is unable to perform within what the user considers to be a reasonable amount of time, it will not be fit for purpose. For example, if a real-time video processing system takes several seconds to process each frame, the system will not be fit for purpose and a more optimal algorithm (or hardware) must be found to enable the system to work.

It is worth noting here that energy consumption has also been proposed as a method of monitoring computational resource usage (for a proposal combining time and energy see Martin 2001, and for examples of energy consumption as a performance metric see Wirtz and Ge 2011; Yang, Chen, and Sze 2017; and Jiao et al. 2010). Whilst this is out of the scope of this paper, such methods are likely to be particularly useful in scrutinising the huge energy cost (both financial and environmental) of some high performance computing demands. For example, Strubell, Ganesh, and McCallum 2019 highlight the power demands and hence the carbon dioxide output of modern natural language processing systems, while Schwartz et al. 2019 critiques this further. Other examples can be seen in the computational process of mining cryptocurrencies such as Bitcoin, where the energy demands can be colossal: O’Dwyer and Malone 2014 estimate that in 2014 the power consumption for mining Bitcoin was comparable to that of the power consumption of the entire country of Ireland, and Li et al. 2019 estimates that in 2018 Monero, an alternative cryptocurrency, had similar power consumption demands, releasing approximately 21,000 tons of carbon dioxide into the earth’s atmosphere through the mining of the currency in China alone. Therefore in order to determine the efficiency and environmental impact of a program, power consumption is an additional metric that scientists and developers should increasingly measure alongside their measurement of computation time.

2 How do computers measure time?

Before proceeding further, it is particularly useful to understand how computers actually *measure* time. Computers can use several different ways to measure the passing of time, each of which has different levels of precision. The most conventional way that computers measure time is using the in-built “real

1. To give a few examples from the field of cancer research that demonstrate the vast differences in computation time: see Kong et al. 2017 which uses convolutional neural networks (CNNs) and two dimensional Long-Short-Term-Memory (2D-LSTM) to detect breast cancer metastasis, a process which takes 1 minute and 45 seconds to detect metastasis in a whole slide image (a type of microscopy image with many pixels e.g. 100,000 x 200,000 pixels); see Wahab, Khan, and Lee 2017 which discusses training a deep learning system to detect breast cancer in histopathology images, a computational process which took approximately 15 hours; and see Mok and Chung 2018 which discusses the segmentation of brain tumours in medical images using generative adversarial networks, recording a time frame of approximately 4 days to train the Coarse-to-fine Boundary-aware Generative Adversarial Networks (CB-GANs). In other areas, computational tasks have be shown to take even longer. As an example, Luong and Manning 2016 explain that in machine translation (e.g. from English to Czech) it took 3 months to run a program to train a deep learning character-based model using a dataset of 15.8 million sentences and 1.269 billion character tokens.

time clock" (RTC) which essentially uses the same type of clock that most wristwatches use. With this method, the computer uses a quartz crystal electronic oscillator, which essentially 'vibrates' at 32.768 kHz, and is therefore able to provide a precision of 30.5 microseconds - the maximum temporal accuracy of this method, as noted by Horan 2013. This quartz crystal time chip sits on the computer's motherboard (the main circuit board) and provides the time for the operating system to use. This can also provide programmers with the ability to measure the computation time of their process, up to the accuracy of 30.5 thousandths of a second.

To provide higher precision, a programmable interval timer (PIT) can be used, which is another type of microchip that sits on the motherboard and has a clock cycle of 1.193182 MHz, enabling an 0.838 microsecond precision in which the computer can interrupt the CPU and start or stop processes, as noted by Crandall et al. 2006. To get greater precision than this, a time stamp counter (TSC) using the CPU's cycle frequency can be used (an example use of TSC is in Skopko and Orosz 2011, and an explanation of TSC by Intel can be found in Intel Corporation 2016). Previously, these CPU clocks ran on a constant clock speed. However, to reach higher speeds, these CPU frequencies are now variable within their use - i.e., the computer can effectively increase or decrease the speed of its own clock (within a defined range) for better performance - which effectively reduces the consistency of the temporal precision, and therefore the reliability of temporal measurement. Finally, there are also high precision event timers (HPET) which, again, are microchips that sit on the motherboard and are able to probe the time of a process when prompted. These operate at a minimum of 10MHz and typically at 18MHz, and therefore have a time precision of less than 0.1 microseconds, or ten thousandths of a second, as noted by VMware 2008.

The measurement of computation time is made even more complicated by the precision of the programmatic function calls - when a programmer requests the amount of time that has elapsed - that easily enables programmers to measure the time that a process has taken. The precision of these can depend on the timing function (or the method of requesting the time used within a program) and which operating system (OS) is being used. Taking an example from the popular programming language Python (Python Software Foundation 2018), this language also allows its users to inspect the commonly-used `time.time` function (Python Software Foundation 2019).[2] This provides a temporal 'resolution' of approximately 15 milliseconds for a Windows operating system, as tested on Windows 8.1 (see Hudek, Sherer, and Schonning 2017; and Stinner 2017 for more information), but only 1 microsecond for a Linux based operating system (tested on Fedora 26 (kernel 4.12) in Stinner 2017) - an in-practice temporal difference in which one system is effectively 15,000 times more accurate for such short computational processes. When using the nanosecond timer function in Python (`time.time_ns()`) this precision changes again, with a resolution of approximately 84 nanoseconds on Linux, and 318 microseconds on Windows - a temporal difference with a factor of 3,750. When documenting computation time using such short processes that will be run many times, it is therefore useful to document the programmatic timing function call that has been used (where possible), along with the operating system that it was used on, *and* the hardware that was used to record it.[3] However, this is not commonly documented in current practice, leading to a range of issues (see section 5 below).

When measuring and documenting computation time it is also important to know what type of computation time is being referred to. Computation time normally either refers to response time or execution time (see Figure 1). The *response* time is the total time elapsed from starting to completing a process, including the time that any other processes, such as operating system processes, might have taken whilst the program was running. By contrast, the *execution* time is is the total time spent computing the process of interest, not including any other processes (see Audsley et al. 1993; and Baruah and Burns 2006). Therefore when documenting how computation time is measured, it is important to note which of these is being referred to, so that the reader can understand if other processes are also being included and, if so, the types of other processes being run in the background should also be documented to provide a complete picture of the situation. A useful early example of a discussion of calculating the execution time of a real-time program by Puschner and Koza highlights some of the issues around calculating the maximum execution time that persist to this day; this paper also provides an interesting example of a real-time program conducting image analysis on a video camera feed in order to control a robotic arm (Puschner and Koza 1989).

2. Python is used by an estimated 8.1 million coders worldwide, according to a 2019 report by SlashData (SlashData 2019). A survey of 90,000 coders by StackOverflow also shows that 41.7% of code developers use Python (Stack Overflow 2019).

3. Note that the temporal resolution of the programmatic function calls are not normally variable on different hardware across most modern devices, but the hardware needs to be recorded as the time to process is greatly affected by the hardware used, as is discussed in more detail in section 4.

When writing algorithms it is possible, in controlled environments, to theoretically determine how many computational steps are required in order to complete the process. This does not immediately correspond to the time that it will take to run, but it does provide the best indication of when, given an input of X, then Y number of steps will need to be computed. The careful art (or sub-field of computer science) for determining this temporal scaling is also known as computational complexity.

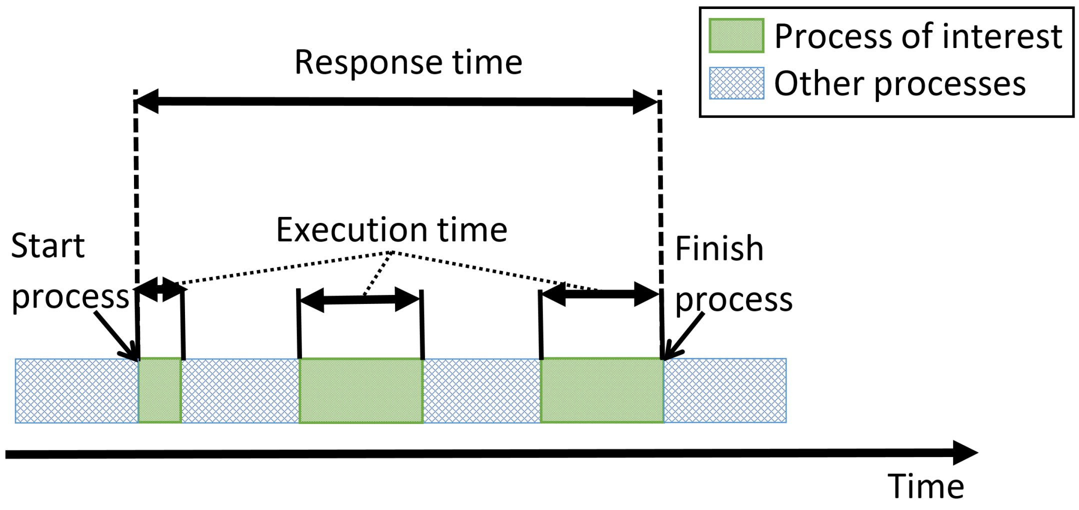

Figure 1: A block diagram illustrating the difference between response time (total time elapsed from starting and finishing a process) and execution time (total time spent actually computing that process, represented by the sum of the green blocks in the diagram). Other processes might be operating system processes or other (possibly high priority) processes which interrupt the process of interest.

3 Computational complexity: too complex in practice?

Computational complexity, a well-studied area within computer science, investigates how an algorithm scales with various inputs - or how mathematically optimised an algorithm is for its intended purpose. A useful text on computational complexity by Arora and Barak provides a thorough overview of the field (Arora and Barak 2009). The mathematical representation of how many steps a program takes to compute, and the time implication due to this complexity, is sometimes referred to as time complexity (for example, see Tomita, Tanaka, and Takahashi 2006; and Avoine, Dysli, and Oechslin 2005). However, while an understanding of the following mathematical concepts is required in order to grasp computational complexity, in practice, many developers do not have the mathematical training to enable them to optimise their code accordingly. Put simply, the gap between thesis and praxis is effectively too great at present for the careful research of computational complexity's talented theoreticians to be widely applied in a broad range of practice.

3.1 Computational complexity: a brief overview

To give a short general overview of the relevant mathematical concepts involved in computational complexity (lay readers may simply skip this section): normally, the exact time to compute is not of particular interest in this mainly theoretical field. What is of interest here is determining how a function (or algorithm) scales up with increasing the numbers of a variable n, mostly in order to calculate the worst case scenario (i.e., the most number of steps required to complete a process), and identifying if a method scales well with increasing this variable or not (for example, increasing the number of data samples to be processed). Scaling well in this context means asking whether increasing this variable also increases the complexity (the number of computations that need to be performed), for example, exponentially, in a linear way, or not at all.

The notation for determining this is sometimes called "Big O" (or "Big Oh") notation, as the "Order" of the complexity function is abbreviated to "O" followed by, in brackets, how the method scales with an increase in the number of the variable "n" (for a full discussion of this, see Arora and Barak 2009). For example, we may want to calculate whether the upper bound (maximum) of the number of steps scales linearly with the number of the variable "n", referred to as $O(n)$ (also known as "Order n"), or does the maximum number of computational steps scale with the square of the number of the variable "n", referred to as $O(n^2)$ (also known as order n^2). To give a more detailed explanation, a function that takes a constant time to calculate would be an order of 1 ($O(1)$) as it does not change with the value of a variable (n) (for example, the number of data samples). A function which scales linearly with the number of samples (n) would be ($O(n)$), and a function which scales quadratically with the number of samples (n) would be ($O(n^2)$), while a function which scales exponentially with the number of samples (n) with constant c would be ($O(c^n)$). To give a real-world example of this in practice, to improve the efficiency of a specific machine learning neural network algorithm, the order of the method may be calculated. A lower order that more effectively scales with increasing numbers of neurons would greatly reduce the time taken to train and predict a result using this model; one example of where this is calculated is in Schmidhuber 1992, which explains an improved method which scales with $O(n^3)$) instead of the compared inferior method, which scales with $O(n^4)$), where n is the number of neurons used in the network. Decreasing this worst-case run time can dramatically improve the efficiency of the program, and can enable the system to be run in a "reasonable time frame".[4]

Calculating the time complexity, or "Big O", of an algorithm is reasonably simple for a small function, or a short amount of code, where the coder is able to understand all of the processes involved in the code (i.e., what the code does, and therefore how many mathematical steps that it will take). However, for larger code projects, these often include multiple functions and libraries, which are other code functions that the coder can draw on and use as a form of shorthand, but may not have been written by the coder, or may not even be inspectable by the coder (perhaps to obfuscate the code and maintain the original coder's intellectual property).

For larger real-world projects, it then becomes incredibly hard to calculate the theoretical time complexity of a program or process as there are simply too many unknowns, or even if all of the code is known, it could be far too big a task to inspect what may be many thousands of lines of code that make up the full program. Although there are some efforts to educate non-computer scientists in field of computational complexity, such as Mertens' paper on Computational Complexity for Physicists (Mertens 2002), another practical barrier is that the mathematical terminology described in the highly theoretical research papers common to the sub-discipline of computational complexity theory can be difficult for the vast majority of non-specialised programmers to understand (examples of such papers are: Martínez et al. 2007; He and Yao 2001; Ko and Friedman 1982; Chen et al. 2010; Cooper 1990; Ladner 1977; Shanbehzadeh and Ogunbona 1997; Cancela and Petingi 2004; and Peled and Ruiz 1980), making the existing body of knowledge unusable in practice for many coders.[5]

This all means that, in practice, calculating the theoretical time complexity is useful for understanding some important short functions that your code might make use of, in order to determine where your code might take a lot of time to run, or where your code might be improved to make it run faster. However, the nuances of computational complexity are, at present, too complex to give professional non-specialised coders the understanding needed to optimise many real-world programs in practice.

3.2 Timing-based attacks

It is worth briefly noting that sometimes it's advantageous for a computational task to take a long time. For example, this can be the case when protecting passwords and in encryption processes. Password and encryption protection works because performing a "brute-force" attack - trying all possible variations of a password or encryption key until the correct one is revealed - should take an extremely long time, deterring potential attackers. A timing-based attack on protected information essentially checks to see how long a request process takes to execute, and breaks the encryption in this way; for example the slower the result is returned the closer to the solution the user is, and so those trying to gain access to encrypted information can iterate through different options checking how long they take to return an answer, and then use this to infer what the correct result is. Timing-based attacks, a type of side-channel attack which takes advantage of security flaws in the implementation or hardware (rather than in the algorithms themselves), have enabled hackers to view private web-content (Felten and Schneider 2000) and to access protected memory on computers (Lipp et al. 2018). As such, computation time can

4. Discussion of what constitutes a "reasonable time frame" is given in section 1 above.

5. There is a marked lack of studies to evidence the effect or scale of this comprehensibility barrier; however, from many years of practical experience in research and teaching in the field, it is apparent that this is a significant barrier for professional programmers needing to optimise their code. A study which is able to quantitatively or qualitatively assess the extent of this issue would be very welcome to reveal the true scale of this problem.

be manipulated for a variety of legal and illegal purposes, demonstrating the real-world importance of computational complexity.

4 The evolution of hardware

The time that a computational process takes to run is also dependent on the hardware that it runs on; for computational processes, given the huge variety of rapidly-developing hardware available, computation time is very much in variance. This may sound obvious - but as this section will discuss, this is poses a far greater problem in practice than might initially be assumed. As hardware capabilities change rapidly over time, it is important to take into account that the time taken to run a program on one machine on a specific date is going to be very different for a similarly-priced machine just a few years later. Recent technological developments mean that the types of processors that are commonly produced today are quite different to those used before 2004, when most CPUs contained a processor with only one core (see Figure 2 for a plot of the evolution of CPU clock frequency and number of cores from 1971 to 2014). Previously, when all processes were sequential (i.e., could only run one at a time on a single core), if the programmer knew that an algorithm process took t seconds to execute and was running on a Y MHz processor, then, moving the same algorithm to a different CPU with a Z MHz processor, the programmer could reasonably expect the process to execute in approximately $(\frac{Y}{Z})t$ seconds.[6] This is no longer the case, as is described in more detail below, and it has now become quite difficult to predict how long an algorithm will take to run across different machines, particularly due to the non-sequential way in which many processes can now be executed. To understand these different hardware options, and how this affects the execution time, the evolution of hardware over time must be properly understood.

In just a few decades, hardware has evolved from single-core Central Processing Units (CPUs), to multi-core CPUs, to Graphical Processing Units (GPUs) (see Figure 3 below), and more recently, Tensor Processing Units (TPUs) (see Jouppi et al. 2017 and Jouppi et al. 2018 for more on TPUs). Over the last 50 years the number of transistors in a CPU has grown exponentially, adhering to Moore's law, which states that the number of transistors in a processor doubles every two years (see Moore 1965 and Moore 1975), and the clock speed of CPUs has also been increasing (see Figure 2). However, as the visible levelling off of the circles in the figure shows, in the last 15 years the increase in clock speed over time has slowed dramatically, and the preferred way to improve performance in practice has been to simply add more cores (as represented with "+" signs in the figure), effectively compensating for this lack of increase in clock speed.

4.1 Single core CPUs

A single core CPU can process one calculation at a time, for example, calculating $2\times3 = 6$. When there are many parts to a calculation then these are calculated sequentially (in series). CPUs are widely recognised to be appropriate for general purpose computing and are currently the quickest method available for sequential tasks.

4.2 Time sharing

Throughout the history of computing, there have been multiple attempts to improve the performance that can be squeezed out of a computer processor.[7] One of the earliest such attempts took place in 1959, when John McCarthy - the researcher who coined the name for the field of 'Artificial Intelligence' four years earlier - wrote a proposal for 'time sharing' on a computer to improve the performance of a computer system when used by many people at once (McCarthy 1959). A useful video, filmed in MIT in 1963, describes this time sharing project, and has been archived by the Computer History Museum; it is also viewable on YouTube (see Fitch 1963).[8] Time sharing was devised to enable many people to share

6. This is a necessary slight oversimplification of the situation as there are also other factors involved here, such as any latency time (i.e. any time delays: for example, it takes time for data to move across the computer circuit boards and initiate a process). However, the basic premise remains that there is in essence a linear scaling between the time to process and the the clock speed on a single core system.

7. For the purpose of this paper, performance is defined as the tasks or processes that are completed per second or per minute.

8. This video clearly describes the rationale behind this design and what such a design is capable of. As an aside, it is quite remarkable how many fundamental computer science concepts were known and discussed at such an early stage, even when computers did not yet have a computer screen interface (at the time the design was implemented, the computer output display was merely an automated typewriter, typing the output of the machine on to sheets or rolls of paper).

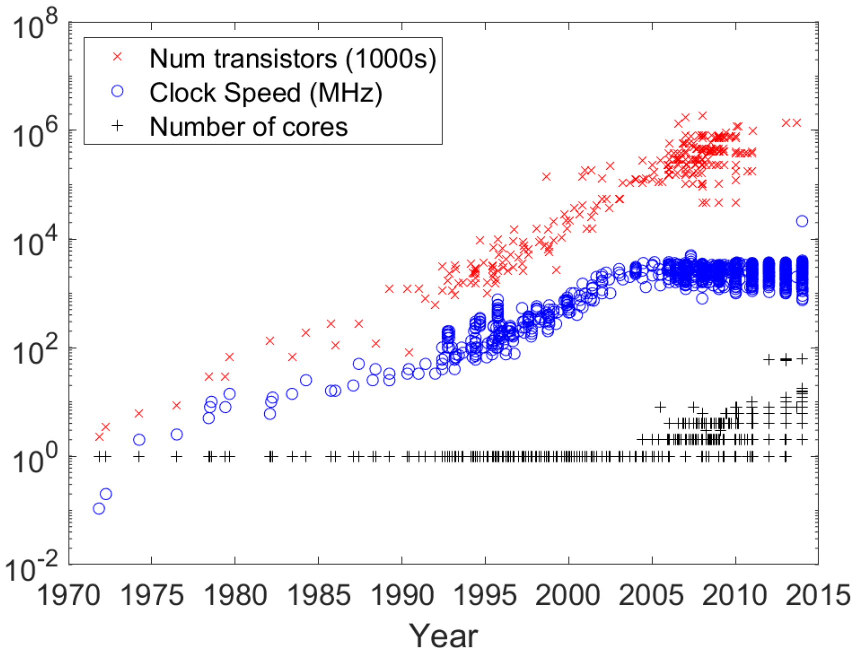

Figure 2: The evolution of CPUs. This plot shows, on a log scale, the number of transistors (crosses), the clock speed (circles), and the number of physical cores (plus signs) in different processors over time. The figure has been made using data from the Stanford CPU Database, using the details of 1,388 processors in which information of the release dates was published. The database was originally discussed in Danowitz et al. 2012 and the database has since been updated and can be downloaded from `http://cpudb.stanford.edu/download` (accessed 28 October 2019)). Note that sometimes not all three parts (clock speed, number of transistors, and number of cores) for some processors are in the database, and therefore only the known parts are plotted here. The data used and code written to produce this plot have been made freely available as open source at: `https://github.com/dcchb/ComputationTime`

the same operating system, enabling a more optimised use of a computer - a very expensive resource at the time. Using this method, the user's processes are effectively executed in a rolling way, which means that the response time for each user is faster than having to wait for each person to finish their program (waiting for a program to finish in its entirety is referred to as batch processing). On modern computers, a similar idea is used in multi- or hyper-threading; here, instead of having multiple users, there are multiple processes (normally from the same user) which can share a processor.

4.3 Cores and threads

If a user has to wait a long time for the computer to give the expected response, this may be due to the CPU not being fully utilised. When the CPU core is waiting for some other information in order to progress in its task, it can be put into an idle state. This idle state is effectively wasted time on the processor, and so multi- or hyper-threading is commonly used to reduce this potentially wasted time by queuing multiple calculations, enabling one calculation to be computed while another is waiting for further information required in order to complete its task (see Marr et al. 2002 for more on hyper-threading).

A CPU may also have a number of physical cores, which means that there are multiple processors built into the same microchip. This means that multiple processes can run at the same time (in parallel) and so the time taken to run many processes reduces. Programmers can therefore use a thread (or task) per core, which means that each core has a different job to do, enabling a program (or multiple programs) to run in a shorter time frame.

In addition to this, it's also possible to have more than one thread per core for a multi-core system, which makes it possible to have multiple jobs queued for each core. This means that while a process is waiting for completion, another thread can take its place. Having more physical cores enables genuine parallel processing (where multiple calculations are being computed at the same time). However, having more threads than the number of cores does not allow for any extra parallelisation - it only provides less

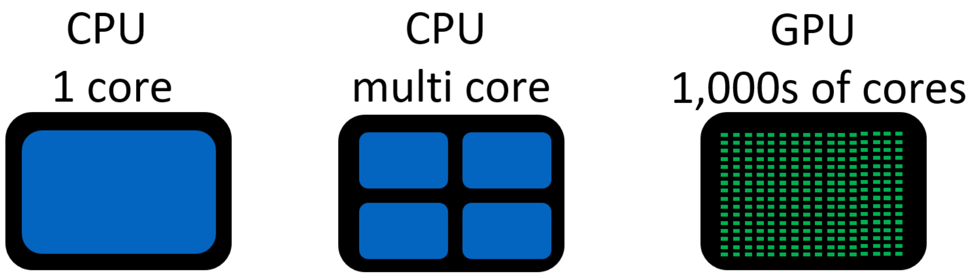

Figure 3: An illustration of the number of processing units on different processors.

down-time. This is particularly important, as the number of threads distributed across the physical cores is sometimes referred to as the number of logical or virtual cores, which in practice can lead to some confusion in the number of processes that can be calculated at the same time.

4.4 Multi-core scalability issues

The move to multi-core processors means that the time to process a calculation can be reduced when the code is made to work in parallel (i.e., using multiple cores at once). However, it is not always possible to do this, and so improvements in processing in series eventually limit this scalability. This is first described by Amdahl (see Amdahl 1967); although not in Amdahl's original paper, the equation which describes this scaling process is called Amdahl's law (Hill and Marty 2008). What this law states is that as long as there is some part of the code which does not run in parallel, the increase in computing speed that can be achieved by running code over many cores (or processors) is limited by the part of the code that runs in series (i.e., the part of the code that can't be split into parallel processes).

A plot of this mathematical law is shown in Figure 4, which illustrates the maximum processing 'speedup' factor that can be achieved when different percentages of the code are parallelised over different numbers of cores. This figure shows that the increase in speed achieved does not scale linearly with the number of cores, as one might expect; put another way, the speedup benefit does not keep increasing at the same rate when you continue to add more cores.

To take this to an extreme example, even when 95% of the execution time is spent using code that can run in parallel, the speedup factor from just running the code on 100 cores (or processors) compared with running it on just one core is only 16.8 times faster, and not anywhere near 100 times as fast, as might be expected. What is important here is that when describing an algorithm, it is not often made clear which parts of the code are possible to run in parallel and which parts of the code have to be run in series (sequentially), and therefore the potential speedup from parallelism is not at all clear in practice. To solve this issue, documenting the relative parts of code and the time that the computer spends on running them, and whether these should be run in parallel or series, would therefore form an incredibly useful collective knowledge base for the developer community who might use, or want to build on, that algorithm or code.

4.5 Ultra parallel processing with GPUs

Processing in parallel has become increasingly sophisticated through using Graphical Processing Units (GPUs). These were originally built to enable the processing of computer graphics, in order to allow graphics to be displayed on a computer screen. However, these are now widely used in game physics and other general purpose computing tasks (see Owens et al. 2008 for more on GPUs). These have grown in computation power in recent years, primarily due to rapid growth of the computer games industry. In GPUs, the base clock speed is still less than in modern CPUs, but GPUs have many more cores. While most CPUs in recent years have a handful of cores (in the range of 4 to 12 physical cores), GPUs take parallel processing to a whole new level. For example, the laptop on which this paper was written (a Surface Book 2) has an NVidia GTX 1050 graphics card which has 640 GPU cores, with a base clock frequency of 1.354GHz,[9] while the most high-end GPU "Deep Learning System" at the time of writing, the NVidia DGX-2, has 81,920 GPU cores.[10] GPUs are capable of super-fast processing of images and

9. For reference or comparison, the specification of the CPU on this Surface Book 2 is an Intel core i7-8650U CPU with a clock frequency of 1.9GHz, with 4 physical cores and 8 logical processors.

10. Deep learning is a much-used term which benefits from some demystifying here: it actually refers to machine learning, a type of data analysis, where many layers of statistics are performed and mathematical models may be learnt automatically

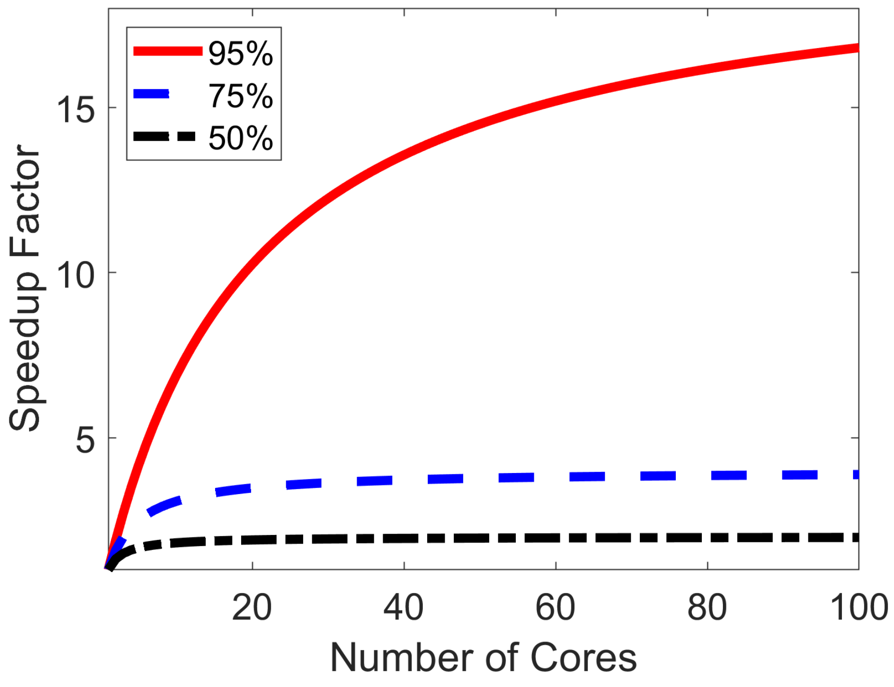

Figure 4: A plot of Amdahl's law, showing the increase in speed as a function of the number of cores in three different scenarios, where the execution time for the part of the code that can be run in parallel is 50%, 75%, and 95% of the total time to run. The code written for this paper to produce this plot has been made freely available as open source at: `https://github.com/dcchb/ComputationTime`

videos; as images and videos are essentially matrices (tables) of information, any dataset which can be composed in matrix form is suitable for GPU acceleration. In the last few years, processing using GPUs in machine learning research - a sub-field of Artificial Intelligence research - has become commonplace as the datasets to process are very large, and require a lot of processing which can be run in parallel. With this large number of processors it is possible to achieve an acceleration in performance when parts of the code are made to run in parallel. Nickolls and Dally found that, when taking Amdahl's law into account, a GPU-only system was fastest when most of the code was possible to run in parallel; however, when some of the code has to run in series, a CPU and GPU combination was found to be the best option (Nickolls and Dally 2010). It is therefore important to provide the details of the GPU hardware specification. When processing on GPUs, as these are so hyper-parallel when compared to CPUs, it becomes even more important to know which parts of the code can be run in parallel and which parts can't in order to find what can be "accelerated" with this parallel architecture.

5 The problem with current practice

In areas of computer science where computation time is a particular focus of the research, such as the sub-field of scheduling for example, computation time is generally very well reported, with a great level of detail frequently provided in such papers (examples of such papers where computation time is reported in enough detail to help others are: Jiang et al. 2010; Zhang and Chatha 2007; Camelo, Donoso, and Castro 2011; Radulescu and Van Gemund 1999; and Wang, Gong, and Kastner 2005). However, in other application domains in which computation time is important but perhaps not the central focus of the research, such as medical imaging and machine learning, important details about the computation time are often neglected in research papers. For example, if we consider the topic of using deep learning for the detection of cancer, where computation time is clearly an important factor in ensuring the result is clinically usable in practice (such calculations often use huge datasets that can take a very long time to process), there is no mention of the computation time taken in many papers. Such papers in which this

from the data provided in order to make either class predictions (e.g. cat or dog) or numerical value predictions (e.g. $10 or $1,000) information would be highly useful but is unfortunately missing include, but are certainly not limited to: Esteva et al. 2017; Rakhlin et al. 2018; Sun, Zheng, and Qian 2016; Abdel-Zaher and Eldeib 2016; Xu et al. 2016; Cruz-Roa et al. 2013; Fakoor et al. 2013; Tsehay et al. 2017; Dhungel, Carneiro, and Bradley 2017; Liu et al. 2017; and Bychkov et al. 2018). In practice, what such an omission may mean is that considerable resources - financial and human - may be dedicated to implementing such promising medical research without considering whether a cancer-detecting algorithm runs in a short enough time to be clinically usable - or if it takes so long to complete that it simply cannot be implemented in an already overburdened healthcare system. Furthermore, when time is a central focus of the research, the algorithms or the techniques used to evaluate them are often too niche to be usable by researchers in other domains. This leads to a problem of inconsistent reporting and inadequate information being provided to researchers, which hampers efforts to determine whether a particular algorithm or method is sufficient for the requirements of the system.

A major problem here is the lack of information provided in the literature - including in research papers and code documentation - about the many important factors that influence computation time, as mentioned above. To briefly recap, these include the type of computation time being measured or reported (e.g. the execution time or response time), details of the timing function or method used, details about which parts of the code were run in parallel and how many processors were used, details of the hardware, and so on. Without this information, this leads to the significant issue of developers having to undertake many iterations of trial and error experimentation in their coding practice, which greatly slows development time. When a researcher or developer doesn't know if an algorithm will perform as required (e.g. if a real-time video analysis algorithm will calculate the required task before the next frame in the video), they will often have to manually code and/or test it to see if it will work, rather than being able to build on the work of others who have already gone through this process.

To give a basic example of this in practice, let's say a developer has been asked to build a program that can solve a potentially lengthy computational task, such as using deep learning to detect cancer in medical images, within two weeks. Without knowing how long the algorithm might take to execute on a similar dataset on a given machine specification, it's very difficult to estimate if the algorithm will complete within that time without using trial and error. Without this information the developer could spend the entire two weeks simply working out that they do not have the resources to deliver the task using the initial algorithm and resources available. The task may have been allocated with the expectation that it would be solvable based on the work of others. However, current computational research is broadly driven by reporting results, rather than documenting processes, and so the fact that the task has been possible for another is of limited practical use if the hardware, software and computation time have not been adequately recorded. In this example, the developer would have effectively wasted two weeks to find that the task is not possible with their time frame or resources. However, had they had access to the right information, they would have been able to see how long the algorithms of others had taken to run, and their hardware and software resources. A quick look at these specifications could mean discovering early on that a similar process took two weeks to run on a much faster, newer machine - and so a different algorithm, or a longer time frame, would be needed.

A better global development process is required in order to reduce this sizable - and often needless - duplication of effort and waste of developer time. According to a 2019 global developer population and demographic study by software market research company Evans Data Corporation, there are 23.9 million developers worldwide (Evans Data Corporation 2019). This means that even if only a small percentage of developers are affected by this poor computation time reporting, leading to needless duplication of effort by others, then this is a significant worldwide issue which has potentially large-scale economic implications. There is therefore need for further research to determine the true economic costs of this issue, revealing the extent of the human, computational, and environmental resources wasted in the needless duplication of effort due to poor reporting.

5.1 The solution: computation time templates and automated reporting

Part of the reason why current practice remains largely inadequate here is that it simply takes time and effort to report computation time. Many developers and researchers may not know how to find and report this information, or will be reluctant to do so due to the presumed effort involved.

To reduce the human effort needed in reporting the required computation time details, below are two templates - a simple and an advanced version - for reporting computation time, which can be used in either code documentation or in research publications. To make this even easier to implement, we have also included an open source program to gather most of this information automatically, populating the template with the CPU processor type, clock speed, number of physical cores, number of logical cores, RAM and operating system. To do this, developers simply run the "Computation_time_reporter" program that is included as part of the code repository that goes with this paper at `https://github.com/dcchb/ComputationTime` which provides both a Windows executable program, and the corresponding Python script which can be run on any operating system. It is therefore hoped that developers across the world may benefit from these easy-to-use and freely accessible tools and templates to automatically report computation time for the benefit of all.

6 Recommendations

There are several recommendations that we would like to provide for any researchers or software developers for whom the time taken for their programs to run is important (i.e., those working with large datasets, complex algorithms or real-time processing - or indeed any developers who are trying to ensure their code runs as quickly and efficiently as possible), to encourage what we consider to be best practice. As this 'best practice' does require effort, and can sometimes appear to get in the way of progressing with the task in hand, we have broken this down into two options: the 'Computation Time: Lite' and 'Computation Time: Pro'.

Documenting the 'Lite' version would mean providing the lowest level of basic documentation on the computation time as a form of shorthand, which would still be useful as a reference point for future developers. By contrast, the 'Pro' version builds on the information documented as part of the 'Lite' version, and adds some useful further documentation and recommended best practice. Of course, this added detail may not always be possible, depending on the situation. However, where available, adding these two forms of recorded documentation would enable the developed algorithm or software to be much more easily translatable into other future software or algorithms, and save the colossal duplication of effort which plagues the field today.

With all of the above information in mind we recommend that future researchers explain the following details when reporting on the computation time:

6.1 Computation Time: Lite (minimum details)

As a minimum, documentation of computation time should list as many of the following as possible:

1. Report the computation time and the type reported (e.g. execution time or response time).
2. Report the hardware used, including the number of cores and type of CPU/GPU.
3. Specify the operating system used.

6.2 Computation Time: Pro (additional details)

Further details that would be helpful are listed below, and can be documented in addition to the above to create a full 'Pro' version of the documentation:

4. Provide code for others to use (preferably open source).
5. Detail the programmatic timing function that has been used to record the computation time (e.g. Python's `time.time`).
6. Break down the processes and list the time each takes.
7. Show or explain which processes can run in parallel and which have to be run in series.

6.3 Computation Time: Templates

Documenting this information does not need to take up a significant portion of a research paper or code documentation; this could be easily inserted in just a few sentences. To make this even easier, templates are provided below and a program to auto-populate many of the fields is provided in the code that goes with the paper, which can be found here: `https://github.com/dcchb/ComputationTime` - we would encourage anyone to feel free to reuse and share these templates as needed, citing this paper.

Computation Time: Lite template

> This program was tested on [insert most important hardware specifications, such as CPU name and clock frequency, number of physical cores, and GPU used], with the Operating System [insert OS, such as Windows 10 Pro or Ubuntu 18.04.3 LTS]. With this hardware and software combination the [execution time (time CPU has taken to execute program only) or response time (time taken from start to end of process, including all background processes)] for this program took [insert time with units] to complete.

Computation Time: Pro template (add-on)

Including additional detail need not take up too much space either. A template example of the Pro version that could be added on to the above minimum details is:

> This program can be downloaded with a [insert licence type: preferably an easy to share open source licence[11]] license from [insert link to where code can be downloaded from, e.g. GitHub[12]]. The computation time was measured using function [insert name of timing function] with the programming language [insert programming language]. The majority of the computation time is spent in [insert name of the part of the code which is computationally expensive[13]], which on testing took [insert percentage of the computation time spent in that part of the code]% of the time to run. Methods [insert names of parallisable methods] can [completely/mostly/partially] be run using parallel processing, while methods [insert series methods] need to be run in series. The current implementation uses [CPU hyper-threading/GPU acceleration/other type of acceleration] for [insert methods]. It is anticipated that further optimisation could be achieved in [insert names of methods], possibly using [insert names of techniques/algorithms to further optimise]; however, this has not yet been investigated.

6.4 Recommendations for future research

In addition to the above recommendations for software developers, it is also worth highlighting some areas in which further scholarly research is needed, which would greatly benefit both the research and software development communities. The following future studies would greatly help in addressing these identified issues:

- A quantitative review of the current practice of computation time documentation within scholarly research, documenting the number of papers which do and don't fulfil each of the above minimum and additional details, and highlighting areas in which documentation is particularly poor.
- A quantitative study on the current practice of computation time documentation within software documentation, documenting the number of software packages which do and don't fulfil each of the above minimum and additional details.
- A thorough and detailed economic costing of the effects of global duplication of effort that occurs due to insufficient or non-existent documentation of computation time, with practical recommendations of how to improve this in practice.
- Detailed discussion of the global cost of computation time in light of computation power - including computation power templates, as used here for computation time - that raises awareness of ecological costs, and is specifically aimed at reducing computational power consumption where the direct relationship between computation time and power used is unclear (i.e., which types of processes consume more - rather than less - power when execution time is shortened by using more power-hungry hardware, for example). As large-scale computing becomes more prevalent in the twenty-first century (e.g., cloud computing, super-computing, and large-scale distributed computation such as cryptocurrency mining), such computational challenges must now attract greater scrutiny in light of the increasingly urgent problems of climate change.

11. A list of open source licenses can be found at: https://opensource.org/licenses, accessed 3rd December 2019. A detailed discussion of open source licences can be found in St. Laurent 2004.

12. GitHub - https://github.com/ - is one of the most widely used code repositories and hosts many open source programs, with over 44 million repositories contributed to between 1st October 2018 and 30th September 2019 (GitHub 2019).

13. Note that 'computationally expensive' just means that it requires a lot of computation power, which takes time to process.

7 Conclusion

As this paper has demonstrated, computation time is a vitally important measurement which can help us to determine whether a program is fit for purpose and completes within a reasonable time frame, acting as a useful signpost for where further optimisation may be needed.

However, without changes to current practice, the practical use value of this knowledge remains limited. To allow coders to more readily speed up the development and optimisation of computer programs in practice, then, it would be particularly useful to measure computation time using programmatic timing function calls, as detailed above, and document this information accordingly. Because the precision of these timing functions can vary by operating system, this also needs to be documented, along with the time to process and a detailed breakdown of the software and which parts of it can be run in parallel.

By being able to access these documented measures of computation time when undertaking other similar tasks, such data enables both the recording developer to track and review the limitations of their own implementation, and also provides other developers and researchers with the key information needed to determine the viability of such an approach for their own purposes. Documenting and sharing such information on what has already been achieved, alongside sharing the code itself, will help others to optimise their own code, allowing more rapid and cost-effective software development, and enabling coders, researchers and development communities to build on the efforts of those who have gone before. The above computation time templates and the corresponding scripts to automate the population of the required fields are an attempt to reduce the difficulty for researchers and developers in reporting computation time, providing the practical means to do this in a way which is easy to document, and easy to understand.

By helping to reveal the extent to which computation time is a crucial factor in optimising global software development, the suggested future research studies are likely to add a real incentive for academies, organisations and individuals to take action and put these recommendations into practice for the future benefit of us all. What is required now is a global effort to think beyond the immediate use of a single program, in an ethical move to help future coders who would greatly benefit from sharing these details. As Linus Torvold, the original creator of the Linux Operating System, notes of open source projects: ‘to really do something well, you have to get a lot of people involved’ (quoted in Diamond 2003). It is to this end that we hope that this paper helps to raise awareness about the importance of computation time documentation, and we hope that you will help to spread the message of how to do it well.

8 Acknowledgements

Thanks to the International Society for the Study of Time (ISST) for accepting an early version of this paper, which was presented at the 17th Triennial ISST Conference on Time’s Urgency, held at Loyola Marymount University in Los Angeles, California, in June 2019, and to its members for their warm feedback at the conference. Thanks also to Patrick Schrempf for directing us to the alternative metric of measuring power consumption as an indication of resource usage.

References

Abdel-Zaher, Ahmed M., and Ayman M. Eldeib. 2016. “Breast cancer classification using deep belief networks.” *Expert Systems with Applications* 46:139–144.

Amdahl, Gene M. 1967. “Validity of the single processor approach to achieving large scale computing capabilities.” In *Proceedings of the April 18-20, 1967, spring joint computer conference,* 483–485. ACM.

Arora, Sanjeev, and Boaz Barak. 2009. *Computational complexity: a modern approach.* Cambridge University Press.

Audsley, Neil, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. 1993. “Applying new scheduling theory to static priority pre-emptive scheduling.” *Software Engineering Journal* 8 (5): 284–292.

Avoine, Gildas, Etienne Dysli, and Philippe Oechslin. 2005. “Reducing time complexity in RFID systems.” In *International workshop on selected areas in cryptography,* 291–306. Springer.

Baruah, Sanjoy, and Alan Burns. 2006. “Sustainable scheduling analysis.” In *2006 27th IEEE International Real-Time Systems Symposium (RTSS’06),* 159–168. IEEE.

Bychkov, Dmitrii, Nina Linder, Riku Turkki, Stig Nordling, Panu E. Kovanen, Clare Verrill, Margarita Walliander, Mikael Lundin, Caj Haglund, and Johan Lundin. 2018. “Deep learning based tissue analysis predicts outcome in colorectal cancer.” *Scientific reports* 8 (1): 3395.

Camelo, Miguel, Yezid Donoso, and Harold Castro. 2011. “MAGS—an approach using multi-objective evolutionary algorithms for grid task scheduling.” *International Journal of Applied Mathematics and Informatics* 5 (2).

Cancela, Héctor, and Louis Petingi. 2004. “Reliability of communication networks with delay constraints: computational complexity and complete topologies.” *International Journal of Mathematics and Mathematical Sciences* 2004 (29): 1551–1562.

Chen, Tianshi, Ke Tang, Guoliang Chen, and Xin Yao. 2010. “Analysis of computational time of simple estimation of distribution algorithms.” *IEEE Transactions on Evolutionary Computation* 14 (1): 1–22.

Cheng, Ruida, Holger R. Roth, Nathan S Lay, Le Lu, Baris Turkbey, William Gandler, Evan S McCreedy, Thomas J. Pohida, Peter A. Pinto, Peter L. Choyke, et al. 2017. “Automatic magnetic resonance prostate segmentation by deep learning with holistically nested networks.” *Journal of Medical Imaging* 4 (4): 041302.

Cooper, Gregory F. 1990. “The computational complexity of probabilistic inference using Bayesian belief networks.” *Artificial intelligence* 42 (2-3): 393–405.

Crandall, Jedidiah R., Gary Wassermann, Daniela A.S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. 2006. “Temporal search: Detecting hidden malware timebombs with virtual machines.” In *ACM SIGARCH Computer Architecture News,* 34:25–36. 5. ACM.

Cruz-Roa, Angel Alfonso, John Edison Arevalo Ovalle, Anant Madabhushi, and Fabio Augusto González Osorio. 2013. “A deep learning architecture for image representation, visual interpretability and automated basal-cell carcinoma cancer detection.” In *International Conference on Medical Image Computing and Computer-Assisted Intervention,* 403–410. Springer.

Danowitz, Andrew, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. 2012. “CPU DB: recording microprocessor history.” *Communications of the ACM* 55 (4): 55–63.

Delling, Daniel, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2015. “Customizable route planning in road networks.” *Transportation Science* 51 (2): 566–591.

Delling, Daniel, Peter Sanders, Dominik Schultes, and Dorothea Wagner. 2009. “Engineering route planning algorithms.” In *Algorithmics of large and complex networks,* 117–139. Springer.

Dhungel, Neeraj, Gustavo Carneiro, and Andrew P. Bradley. 2017. “Fully automated classification of mammograms using deep residual neural networks.” In *2017 IEEE 14th International Symposium on Biomedical Imaging (ISBI 2017),* 310–314. IEEE.

Diamond, David. 2003. “THE WAY WE LIVE NOW: 9-28-03: Questions For Linus Torvolds; The Sharer.” *The New York Times Magazine* (September 28): 23.

Esteva, Andre, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. 2017. “Dermatologist-level classification of skin cancer with deep neural networks.” *Nature* 542 (7639): 115.

Evans Data Corporation. 2019. “Global Developer Population and Demographic Study 2019 Vol. 1.” Accessed September 20, 2019. `https://evansdata.com/reports/viewRelease.php?reportID=9`.

Fakoor, Rasool, Faisal Ladhak, Azade Nazi, and Manfred Huber. 2013. “Using deep learning to enhance cancer diagnosis and classification.” In *Proceedings of the international conference on machine learning,* vol. 28. ACM New York, USA.

Felten, Edward W, and Michael A Schneider. 2000. “Timing attacks on web privacy.” In *Proceedings of the 7th ACM conference on Computer and communications security,* 25–32.

Fitch, John. 1963. "1963 Timesharing: A Solution to Computer Bottlenecks." YouTube. Accessed November 4, 2019. https://youtu.be/Q07PhW5sCEk.

GitHub. 2019. "The State of the Octoverse - 2019." Accessed December 3, 2019. https://octoverse.github.com/.

He, Jun, and Xin Yao. 2001. "Drift analysis and average time complexity of evolutionary algorithms." *Artificial intelligence* 127 (1): 57–85.

Hill, Mark D., and Michael R. Marty. 2008. "Amdahl's law in the multicore era." *Computer* 41 (7): 33–38.

Horan, Brendan. 2013. "Adding an RTC." In *Practical Raspberry Pi,* 145–162. Springer.

Hudek, Ted, Tim Sherer, and Nick Schonning. 2017. "Windows Driver Docs: High-Resolution Timers." Accessed November 28, 2019. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/high-resolution-timers.

Huynh, Benjamin Q., Hui Li, and Maryellen L. Giger. 2016. "Digital mammographic tumor classification using transfer learning from deep convolutional neural networks." *Journal of Medical Imaging* 3 (3): 034501.

Intel Corporation. 2016. *Intel® 64 and ia-32 architectures software developer's manual: Volume 3B: System programming Guide, Part 2, section 17.15 time-stamp counter.* Technical report.

Jiang, Yunlian, Kai Tian, Xipeng Shen, Jinghe Zhang, Jie Chen, and Rahul Tripathi. 2010. "The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions." *IEEE Transactions on Parallel and Distributed Systems* 22 (7): 1192–1205.

Jiao, Yang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. 2010. "Power and performance characterization of computational kernels on the gpu." In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing,* 221–228. IEEE Computer Society.

Jouppi, Norman P., Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. "In-datacenter performance analysis of a tensor processing unit." In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA),* 1–12. IEEE.

Jouppi, Norman, Cliff Young, Nishant Patil, and David Patterson. 2018. "Motivation for and evaluation of the first tensor processing unit." *IEEE Micro* 38 (3): 10–19.

Ko, Ker-I, and Harvey Friedman. 1982. "Computational complexity of real functions." *Theoretical Computer Science* 20 (3): 323–352.

Kong, Bin, Xin Wang, Zhongyu Li, Qi Song, and Shaoting Zhang. 2017. "Cancer metastasis detection via spatially structured deep network." In *International Conference on Information Processing in Medical Imaging,* 236–248. Springer.

Ladner, Richard E. 1977. "The computational complexity of provability in systems of modal propositional logic." *SIAM journal on computing* 6 (3): 467–480.

Li, Jingming, Nianping Li, Jinqing Peng, Haijiao Cui, and Zhibin Wu. 2019. "Energy consumption of cryptocurrency mining: A study of electricity consumption in mining cryptocurrencies." *Energy* 168:160–168.

Lipp, Moritz, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. "Meltdown: Reading kernel memory from user space." In *27th {USENIX} Security Symposium ({USENIX} Security 18),* 973–990.

Liu, Saifeng, Huaixiu Zheng, Yesu Feng, and Wei Li. 2017. "Prostate cancer diagnosis using deep learning with 3D multiparametric MRI." In *Medical Imaging 2017: Computer-Aided Diagnosis,* 10134:1013428. International Society for Optics and Photonics.

Luong, Minh-Thang, and Christopher D. Manning. 2016. "Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models." In *Association for Computational Linguistics (ACL).* Berlin, Germany. https://nlp.stanford.edu/pubs/luong2016acl_hybrid.pdf.

Marr, Deborah T., Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. 2002. "Hyper-Threading Technology Architecture and Microarchitecture." *Intel Technology Journal* 6 (1).

Martin, Alain J. 2001. "Towards an energy complexity of computation." *Information Processing Letters* 77 (2-4): 181–187.

Martínez, Sonia, Francesco Bullo, Jorge Cortés, and Emilio Frazzoli. 2007. "On synchronous robotic networks—Part II: Time complexity of rendezvous and deployment algorithms." *IEEE Transactions on Automatic Control* 52 (12): 2214–2226.

McCarthy, John. 1959. *Memorandum to P.M. Morse proposing time sharing.*

Melinšćak, Martina, Pavle Prentašić, and Sven Lončarić. 2015. "Retinal vessel segmentation using deep neural networks." In *10th International Conference on Computer Vision Theory and Applications (VISAPP 2015).*

Mertens, Stephan. 2002. "Computational complexity for physicists." *Computing in Science & Engineering* 4 (3): 31.

Mok, Tony C.W., and Albert C.S. Chung. 2018. "Learning data augmentation for brain tumor segmentation with coarse-to-fine generative adversarial networks." In *International MICCAI Brainlesion Workshop,* 70–80. Springer.

Moore, Gordon E. 1965. "Cramming more components onto integrated circuits." *Electronics* 38 (8): 114.

———. 1975. "Progress in digital integrated electronics." In *Electron Devices Meeting,* 21:11–13.

Nickolls, John, and William J. Dally. 2010. "The GPU computing era." *IEEE micro* 30 (2): 56–69.

O'Dwyer, K. J., and D. Malone. 2014. "Bitcoin mining and its energy footprint." In *25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014),* 280–285.

Owens, John D., Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. 2008. "GPU Computing." *Proceedings of the IEEE* 96 (5): 879–899. doi:10.1109/JPROC.2008.9177571.

Peled, Abraham, and Antonio Ruiz. 1980. "Frequency domain data transmission using reduced computational complexity algorithms." In *ICASSP'80. IEEE International Conference on Acoustics, Speech, and Signal Processing,* 5:964–967. IEEE.

Puschner, Peter, and Ch. Koza. 1989. "Calculating the maximum execution time of real-time programs." *Real-time systems* 1 (2): 159–176.

Python Software Foundation. 2018. *Python.* Version 3.7, June 27. Accessed November 28, 2019. https://www.python.org/.

———. 2019. "time — Time access and conversions." Accessed October 8, 2019. https://docs.python.org/3/library/time.html.

Radulescu, Andrei, and Arjan J.C. Van Gemund. 1999. "On the complexity of list scheduling algorithms for distributed-memory systems." In *International Conference on Supercomputing,* 68–75. Citeseer.

Rakhlin, Alexander, Alexey Shvets, Vladimir Iglovikov, and Alexandr A. Kalinin. 2018. "Deep convolutional neural networks for breast cancer histology image analysis." In *International Conference Image Analysis and Recognition,* 737–744. Springer.

Schmidhuber, Jürgen. 1992. "A fixed size storage O (n 3) time complexity learning algorithm for fully recurrent continually running networks." *Neural Computation* 4 (2): 243–248.

Schwartz, Roy, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2019. *Green AI.* arXiv: 1907.10597.

Shanbehzadeh, Jamshid, and Philip O. Ogunbona. 1997. "On the computational complexity of the LBG and PNN algorithms." *IEEE Transactions on Image Processing* 6 (4): 614–616.

Skopko, Tamas, and Peter Orosz. 2011. "Investigating the Precision of the TSC-based Packet Timestamping." *Carpathian Journal of Electronic & Computer Engineering* 4 (1).

SlashData. 2019. "Global Developer Population Report - Community Edition." Accessed November 29, 2019. https://slashdata-website-cms.s3.amazonaws.com/sample_reports/EiWEyM5bfZe1Kug_.pdf.

St. Laurent, Andrew M. 2004. *Understanding open source and free software licensing: guide to navigating licensing issues in existing & new software.* O'Reilly Media, Inc.

Stack Overflow. 2019. "Stack Overflow's Developer Survey Results - 2019." Accessed November 29, 2019. https://insights.stackoverflow.com/survey/2019#most-popular-technologies.

Stinner, Victor. 2017. "Python Developer's Guide: PEP 564 – Add new time functions with nanosecond resolution: Annex: Clocks Resolution in Python." Accessed November 28, 2019. https://www.python.org/dev/peps/pep-0564/#annex-clocks-resolution-in-python.

Strubell, Emma, Ananya Ganesh, and Andrew McCallum. 2019. "Energy and Policy Considerations for Deep Learning in NLP." In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics,* 3645–3650. Florence, Italy: Association for Computational Linguistics, July. doi:10.18653/v1/P19-1355.

Sun, Wenqing, Bin Zheng, and Wei Qian. 2016. "Computer aided lung cancer diagnosis with deep learning algorithms." In *Medical imaging 2016: computer-aided diagnosis,* vol. 9785, 97850Z. International Society for Optics and Photonics.

Tomita, Etsuji, Akira Tanaka, and Haruhisa Takahashi. 2006. "The worst-case time complexity for generating all maximal cliques and computational experiments." *Theoretical Computer Science* 363 (1): 28–42.

Tsehay, Yohannes K., Nathan S. Lay, Holger R. Roth, Xiaosong Wang, Jin Tae Kwak, Baris I. Turkbey, Peter A. Pinto, Brad J. Wood, and Ronald M. Summers. 2017. "Convolutional neural network based deep-learning architecture for prostate cancer detection on multiparametric magnetic resonance images." In *Medical Imaging 2017: Computer-Aided Diagnosis,* 10134:1013405. International Society for Optics and Photonics.

VMware. 2008. "Timekeeping in VMware Virtual Machines." Accessed October 8, 2019. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf.

Vo, Nam, Nathan Jacobs, and James Hays. 2017. "Revisiting im2gps in the deep learning era." In *Proceedings of the IEEE International Conference on Computer Vision,* 2621–2630.

Wahab, Noorul, Asifullah Khan, and Yeon Soo Lee. 2017. "Two-phase deep convolutional neural network for reducing class skewness in histopathological images based breast cancer detection." *Computers in biology and medicine* 85:86–97.

Wang, Gang, Wenrui Gong, and Ryan Kastner. 2005. "Instruction scheduling using MAX-MIN ant system optimization." In *Proceedings of the 15th ACM Great Lakes symposium on VLSI,* 44–49. ACM.

Wang, Haibo, Angel Cruz Roa, Ajay N. Basavanhally, Hannah L. Gilmore, Natalie Shih, Mike Feldman, John Tomaszewski, Fabio Gonzalez, and Anant Madabhushi. 2014. "Mitosis detection in breast cancer pathology images by combining handcrafted and convolutional neural network features." *Journal of Medical Imaging* 1 (3): 034003.

Wirtz, Thomas, and Rong Ge. 2011. "Improving mapreduce energy efficiency for computation intensive workloads." In *2011 International Green Computing Conference and Workshops,* 1–8. IEEE.

Wu, Chi-Jui, Steven Houben, and Nicolai Marquardt. 2017. "Eaglesense: Tracking people and devices in interactive spaces using real-time top-view depth-sensing." In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems,* 3929–3942. ACM.

Xu, Tao, Han Zhang, Xiaolei Huang, Shaoting Zhang, and Dimitris N. Metaxas. 2016. "Multimodal deep learning for cervical dysplasia diagnosis." In *International Conference on Medical Image Computing and Computer-Assisted Intervention,* 115–123. Springer.

Yang, Tien-Ju, Yu-Hsin Chen, and Vivienne Sze. 2017. "Designing energy-efficient convolutional neural networks using energy-aware pruning." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* 5687–5695.

Zhang, Sushu, and Karam S Chatha. 2007. “Approximation algorithm for the temperature-aware scheduling problem.” In *2007 IEEE/ACM International Conference on Computer-Aided Design,* 281–288. IEEE.