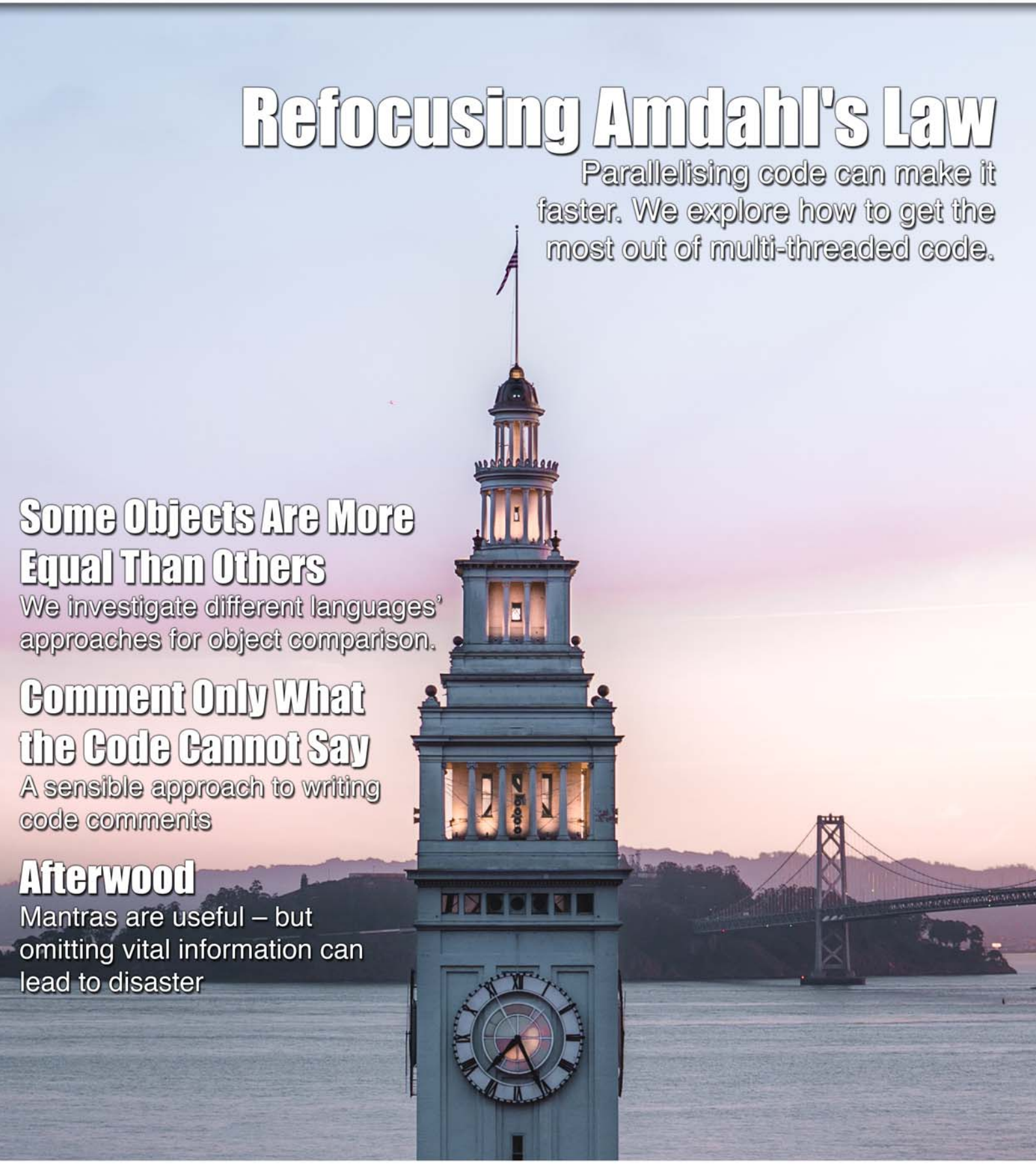
overload 157

JUNE 2020 £4.50



Refocusing Amdahl's Law

Parallelising code can make it faster. We explore how to get the most out of multi-threaded code.

Some Objects Are More Equal Than Others

We investigate different languages' approaches for object comparison.

Comment Only What the Code Cannot Say

A sensible approach to writing code comments

Afterwood

Mantras are useful – but omitting vital information can lead to disaster

A magazine of ACCU

ISSN: 1354-3172

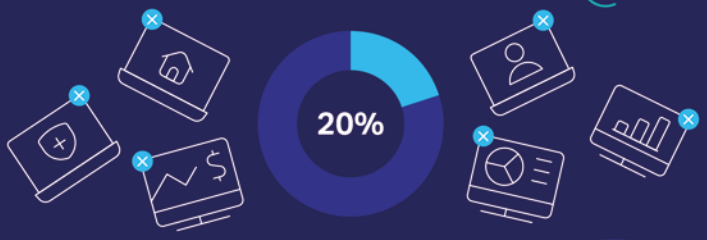
Remote access software:

opentext™

Ensuring productivity by eliminating single points of failure

Unresponsive programs

Knowledge workers on average see productivity drop by as much as **20%** because of unresponsive applications.[1]



Increase productivity and eliminate downtime

A remote access platform maximizes end user productivity with:

- Extremely fast responsiveness resulting from highly efficient data compression protocol.
- Access to Windows®, UNIX® and Linux® applications and desktops.
- Auto-resume for network interruptions.
- Remote collaboration and built-in screen sharing.
- Stable sessions.
- Highly available architecture out of the box.

How it works

Example:
Ensure a highly available, load balanced remote desktop infrastructure for users.

1. Establish load balancing rules based on the number of current sessions and CPU or memory utilization of available hosts.
2. Assign users to the least loaded servers in their host pool with the load balancer.
3. Install connection nodes to offload CPU-intensive tasks and provide faster performance of application servers.
4. Ensure a user's work is never lost with auto-reconnect, which resumes a session when the device is back online.
5. Configure servers in a high availability (HA) cluster to distribute website load and eliminate single points of failure.

The result

Provide users with reliable and fast access to work desktops and server applications from any platform and location, avoiding costly downtime, interruptions and loss of work.

OpenText™ Exceed™ TurboX empowers a global workforce with a high-performance remote access solution that ensures accessibility to graphically demanding applications and desktops through a web browser.

Key partners include:

Agisoft · ASPOSE · ATLASSIAN · axure · BrowserStack · DevExpress · embarcadero · Hex-Rays · intel Software · Microsoft Silver Partner · qbs PUBLISHING · Sketch · SMARTBEAR · SPARX SYSTEMS · Telerik Develop experiences · think-cell · Visual Paradigm

For your latest software needs, contact our team on:

020 8733 7101 **sales@qbs.co.uk** **www@qbs.co.uk**

QBS PUBLISHING

OVERLOAD 157

June 2020

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 158 should be submitted by 1st July 2020 and those for Overload 159 by 1st September 2020.

Overload is a publication of the ACCU
For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Copyrights and Trade Marks

Rainclouds and Olive Branches

Sometimes warning signs get missed or ignored. Frances Buontempo considers signs of trouble and seeds of hope.

By now, you might have realized I am unlikely to write an editorial, and you're right. The warnings signs have been there for years. Before I make my excuses this time, let me apologise for the typos in the previous edition of *Overload*. When I read through the final draft my mind was on others things; telcos, meetings, deadlines, global pandemics. I suspected I wasn't paying 100% attention. At that point, I should have stopped and asked for help, or at least stepped away from the keyboard for a while. Many thanks to Daniel and Bob for catching the mistakes and correcting the online version.

Do you ever have a nagging sense that you're missing something? The pressure of a deadline or needing to get something out of the way might encourage you to ignore the sense of doubt, and press on anyway. Sometimes that's the right thing to do – a distraction can derail you. However, sometimes you need to listen to your gut feelings. This discernment takes practice. If you're trying to hunt the cause of a bug, experience can drive you to look for one kind of cause. This may or may not be pertinent to the problem at hand. A rigorous approach, adding tests round various possibilities and eliminating them one at a time might be the best way. Or, it might be worth trying your first instinct to initiate investigations. If you catch yourself with a sense of déjà vu some hours later, with a dim recollection of having tried that already, you do need to step away from the keyboard. Listen out for warning signs, then decide a course of action. Going with an impending sense of doom or possible looming disaster and forming action plans every time isn't wise. Being aware that something might not work, and forming an exit strategy or plan B, or at least being prepared to, might be more sensible.

Some warnings are hidden in plain sight, for example compiler warnings. Do you ignore these or `#pragma disable` specific warnings? I deliberately turned off all warnings related to doxygen comments on a codebase a while ago, thereby reducing the number from a couple of thousand to about 40. Some may argue that turning off warnings is a crime – the warning is often telling you something important. However, in this case, the remaining problems were genuine issues, or rather ones that could affect the outputs of the system, which we managed to fix. Since we were used to the flood of warnings, we had ignored them, allowing new problems to hide in the noise. Linters and other code analysis tools can cause similar problems – a flood of issues makes it hard to know where to start. It's OK to hide some problems for a while, shielded by an umbrella of some sort, as it were, and concentrate on what's left.

It's a shame you can't turn off compiler errors at will. A way to avoid a deluge of error messages is to compile the code frequently – small baby steps. I used to write code for days before attempting to build it when I was a lot younger. When I finally got round to trying to compile, usually on a Friday afternoon, I ended up staying late, giving up in despair after several attempts to Ctrl-z my way back to something that might work. First, undo and redo isn't a good way to do version control, and second, smaller changes means smaller, incremental builds. Furthermore, you can commit your code to a proper version control between steps, and get a proper weekend. I can't face working the way I used to ever again, partly because I can't hold that much state in my head any more, and partly because the smaller steps appear to make progress quicker. Regardless of whether you are working on a compiled or interpreted language, if you are walking in the shadow of the valley of a worrying cloud of errors and warnings, find a sheltered spot and fix one small thing at a time. Give yourself an olive branch of sorts – a glimmer of hope that the flood will stop soon.

I touched earlier on getting older changing my perspective; however, it can also cloud my judgement. If I see a problem that's similar to something I've encountered before, I may filter the evidence or discussion through the eyes of my experience and miss subtle, or glaring differences. Sometimes the newest or youngest person in a group asks the best questions. You can suffer team-blindness or experience-blindness, rather like Hans Christian Anderson's story of the Emperor's new clothes [Wikipedia-1]. Two weavers are paid a fortune to make the Emperor new clothes. They take the money and eventually hand over a magic invisible outfit, which he duly parades around town wearing. His subjects are too afraid to speak up but eventually a child cries out, "But he isn't wearing anything at all!" The older people are perhaps afraid of the consequences of speaking up, or can't understand why no one else is pointing out the obvious, so wait and wonder. It takes a child to point out the situation. In some kind of parallel, Greta Thunberg [Wikipedia-2] has spoken out on climate change so many times. Her speeches elicit various reactions. Some would claim she is too young to understand the economy and will 'grow out of it' when she has bills to pay. Others would say there is no economy if everything is dead. I know which side I'm on. In both cases, a brave young person calls out a situation.

Getting older means you have had more time to develop deeply ingrained habits, some may be good, some may not. I've been trying to do at least 10k steps a day for a while now. I do wonder where the magic number came from, but that's another story. Wandering off to have a cigarette outside might not do much for my health, but it does up my step count. Standing up during our daily stand up and pacing on the other hand might be a good thing. However, I need to remind myself that being a few hundred steps away from my target isn't a disaster – for any routine or habit, you need to understand why a specific approach is being taken or metric is being used. Cargo-cult style rumours or superstitious going on [Buontempo14] are prevalent. "Have you tried turning it off and on

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

again?" "You have to do this in the right order". "You mustn't write tests for this – it's just a script." "It is written." Such across-the-board statements should be questioned. Sometimes ideas are recommendations written by experts, perhaps giving best practice guidelines. They may not apply in all circumstances across the board, but it's worth listening to people who know what they are talking about. It's OK to have questions, for example examining the source code used to model epidemiology data and wondering if it's thread-safe. That doesn't make you a full-on Luddite.

If you are given general advice or guidelines, ask "Why? For what purpose? For what reason?" I've mentioned the five whys before [Wikipedia-3]. Do the same if you have a habit or ritual you stick to. Some things are helpful for a while, but end up no longer applying. Some things are hard to stick to, like exercise, but reminding yourself why can be motivational. Other things are total nonsense and need calling out. Sometimes, you really can't tell what's for the best, so need to sit tight and see what happens.

When you question approaches, you may find nobody knows what will happen if things change. In that case, you need a way to ask "What-if" type questions. When I worked in finance, I worked on countless risk systems, trying to measure how much money might be lost under various circumstances, and insure against possible outcomes, I also came across 'PnL attribution' engines [Wikipedia-4]. These attempt to find root causes of changes in value – the PnL, profit and loss of the name. Both types of systems allow you to ask "what if" questions. What if interest rates go negative? What if there's a recession? Some questions may not be covered in the model: negative interest rates caused problems, not just in real life, negative prices of commodities may not be catered for, everybody staying indoors for weeks and giving up shopping apart from buying bread flour may also be somewhat specific and not covered. This doesn't make the models useless. They can't deal with some situations, but they still allow you to explore what might happen under a change of circumstances.

However good or bad such system are, they require some form of mathematical model or rules (think finite state machine), which need encoding. Many systems are incredibly complicated so it's hard to know where to start. Nonetheless, starting somewhere, with a simple model or set of rules gives a starting point. A glimmer of hope. If you want to see what happens if you change the interest rate, *r*, slap the interest rate in your model and try what-if scenarios. See what happens. It may do flaky unlikely things, so improve your model, after you have tested it thoroughly and made sure you don't have lots of bugs. Don't panic. Currently, another *r*, the *r* number, is making the news. Again, don't panic. This *r*, the reproduction number, works rather like an interest rate – it gives exponential growth. If it's less than 1, then things don't grow/spread out of hand. However, let's ignore the pandemic elephant in the room for now. It's surprising to see how surprised people appear to be about exponential growth, if the number of times it has been explained on the news is anything to go by.

Without a numerical problem, you can still ask what-if questions. Try role-playing a situation with someone, or running a discussion in your head, like fixing a bug. I've mentioned rubber duck debugging before [Buontempo20]. Believe it or not, you can build computer simulations to explore possibilities rather than giving a repeatable output for a given set of inputs. One claimed form of AI, chatbots, may help you role play to decide how to proceed. Another, to my mind, canonical example, are cellular automata, 'Conway's Game of Life' being well known [Wikipedia-5]. I am told John Conway recently died of the pandemic [Guardian20]. As I saw tributes flow in, one reminded me of using the game of life to build a simulated computer, which could run the game Tetris [StackExchange13]. Over-engineered, but fun.

Let's face the elephant in the room. The pandemic had warning signs. I watched news of Wuhan going into lockdown and worried for people living there. It never occurred to me this would spread. I suspect I'm not alone. Back in 2015, Bill Gates gave a TED talk entitled 'The next outbreak? We're not ready' [Gates15]. It takes expertise and discernment to know how to respond warning signs. During the pandemic, I notice people misunderstanding information. When we are told over 60s with underlying health conditions are more at risk, some people hear, "If you're under sixty or don't have health problems, you won't catch the virus." As we all know, trying to convey technical information is hard. Does making code multi-threaded make it quicker? It depends. People want easy answers but that's not always possible. Andrew Peck wrote a short piece for us [Peck14] a while ago, exploring technical communication, describing us as prophets, scribes and high priests. Re-listening to Bill Gate's talk certainly makes him seem like some kind of prophet. But, as they say, hindsight is 20/20. Mind you, that seems to mean average eyesight [UrbanDictionary] – I'm not sure if that's a good thing or not.

I've talked mainly about rain clouds so far. There are some olive branches, or rays of hope on the way though. By looking at warnings, a few at a time, you can overcome problems. I hope we can look back on what's going on across the globe now, and form our own rainbow of promises, figuring out things that need to stop or change, and finding new ways of working and even living. These two thoughts are well expressed by Schopenhauer,

> All truth passes through three stages. First, it is ridiculed. Second, it is violently opposed. Third, it is accepted as being self-evident.
>
> Mostly it is loss which teaches us about the worth of things.

References

[Buontempo14] Frances Buontempo (2014) 'Peer Reviewed' (editorial), *Overload* 123, October 2014, available at: https://accu.org/index.php/journals/2017

[Buontempo20] Frances Buontempo (2020) 'R.E.S.P.E.C.T.' (editorial), *Overload* 156, April 2020, available at: https://accu.org/index.php/journals/2775

[Gates15] Bill Gates, March 2015, 'The next outbreak? We're not ready' available at: https://www.ted.com/talks/bill_gates_the_next_outbreak_we_re_not_ready?language=en

[Guardian20] 'John Horton Conway obituary', *The Guardian*, 23 April 2020, available at: https://www.theguardian.com/science/2020/apr/23/john-horton-conway-obituary

[Peck14] Andrew Peck (2014) 'People of the Doc', *Overload* 124, available at: https://accu.org/index.php/journals/2044

[StackExchange13] 'Build a working game of Tetris in Conway's Game of Life', available at: https://codegolf.stackexchange.com/questions/11880/build-a-working-game-of-tetris-in-conways-game-of-life

[UrbanDictionary] 'Hindsight is 20/20': https://www.urbandictionary.com/define.php?term=Hindsight%20is%2020%2F20

[Wikipedia-1] 'The Emperor's New Clothes', https://en.wikipedia.org/wiki/The_Emperor%27s_New_Clothes

[Wikipedia-2] Greta Thunberg: https://en.wikipedia.org/wiki/Greta_Thunberg

[Wikipedia-3] 'Five whys': https://en.wikipedia.org/wiki/Five_whys

[Wikipedia-4] 'PnL Explained: https://en.wikipedia.org/wiki/PnL_Explained

[Wikipedia-5] 'Conway's Game of Life': https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Comment Only What The Code Cannot Say

Comments can help or hinder. Kevlin Henney assesses when to avoid them.

As with any other form of writing, there is a skill to writing good comments in code. And, as with any other form of writing, much of the skill is in knowing what not to write.

It has been said that the difference between theory and practice is greater in practice than it is in theory. This observation certainly applies to comments. In theory, the general idea of commenting code sounds like a worthwhile one: offer the reader detail, an explanation of what's going on. What could be more helpful than being helpful? In practice, however, comments all too easily and all too often become a blight.

When code is ill-formed, compilers, interpreters, and other tools will be sure to object. If the code is in some way functionally incorrect, reviews, static analysis, tests, and day-to-day use in a production environment will flush out many bugs. A known bug is a call to action.

But what about comments? In *The Elements of Programming Style*, Kernighan and Plauger [Kernighan78] note that:

> A comment is of zero (or negative) value if it is wrong.

Incorrect comments, however, do not seem to raise the same sense of alarm as incorrect code. Such comments often litter and survive in a codebase in a way that coding errors never could. They provide a constant source of distraction and misinformation, a subtle but constant drag on a programmer's time and attention.

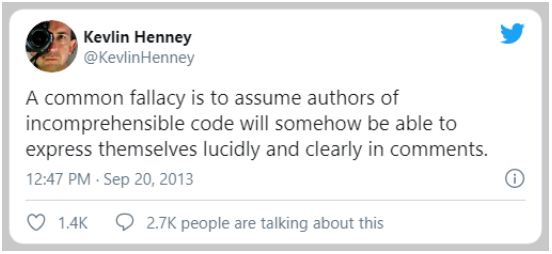
What of comments that are not technically wrong, but add no value to the code? Such comments are noise. Comments that parrot the code offer nothing extra to the reader – stating something once in code and again in natural language does not make it any truer or more real. Although often interpreted simply as a principle of avoiding code duplication, DRY (Don't Repeat Yourself) cautions more broadly against repeated expression of knowledge within a system.

If repeating the workings of code in comments is noisy, retaining code in comments is noisier. Commented-out code is not executable code, so it has no useful effect for either reader or runtime. A common justification for retaining commented-out code is that it might (in some indefinite future, for reasons unknown) become useful. Commented-out code, however, becomes stale very quickly. As time passes, it becomes increasingly less likely that uncommenting such code will be meaningful or even compilable – and be careful what you wish for: in the worst case, it will compile. Version-related comments and commented-out code try to address questions of versioning and history. These questions have already been answered (far more effectively) by version control tools.

Kevlin Henney is an independent consultant, speaker, writer and trainer. His development interests include programming languages, software architecture and programming practices, with a particular emphasis on unit testing and reasoning about practices at the team level. Kevlin loves to help and inspire others, share ideas and ask questions. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*. He is also editor of *97 Things Every Programmer Should Know* and co-editor of *97 Things Every Java Programmer Should Know*.

A prevalence of noisy comments and incorrect comments in a codebase can instil a habit in programmers: a habit to ignore all comments, either by skipping past them or by taking active measures to hide them. Programmers are resourceful and will route around anything perceived to be damage: folding comments up; switching syntax colouring so that the comments and the background are the same colour; scripting to filter out comments. To save a codebase from such misapplications of programmer ingenuity – and to reduce the risk of overlooking any comments of genuine value – comments should be treated as though they were code. Each comment should add some value for the reader, otherwise it is waste that should be removed or rewritten.

What then qualifies as value? Comments should say something code does not and cannot say. Ask what problem the presence of a comment addresses, and ask if there's another way to solve it. A comment explaining what a piece of code should already say is an invitation to change code structure or coding conventions so the code speaks for itself.



Instead of compensating for poor variable, method, class, or test names, rename them. Instead of commenting sections in long functions, extract smaller functions whose names capture the former sections' intent. Instead of writing apologies and apologia, follow Kernighan and Plauger's advice from the 1970s:

> Don't comment bad code – rewrite it.

Try to express as much as possible through code that means something to both the compiler and the programmer. Use language structure, identifiers, and idioms to communicate to the reader what you mean. Any shortfall between what you can express in code and what you would like to express in total becomes a plausible candidate for a useful comment. Comment what the code cannot say, not simply what it does not say. ■

Reference

[Henney13] Kevlin Henney's tweet: https://twitter.com/KevlinHenney/status/381021802941906944

[Kernighan78] B. Kernighan and P. J. Plauger (1978) *Elements of Programming Style* (2nd Edition), McGraw-Hill Education.

This article was previously published on Kevlin's blog: https://medium.com/@kevlinhenney/comment-only-what-the-code-cannot-say-dfdb7b8595ac

Refocusing Amdahl's Law

Parallelising code can make it faster. Lucian Radu Teodorescu explores how to get the most out of multi-threaded code.

Throwing 10 cores at a serial algorithm doesn't make it 10 times faster – that's a well-known fact. Oftentimes, the algorithm barely becomes 3 times faster. Such a waste of resources! And who is to blame? Of course, it must be Gene Amdahl, with his stupid law [Amdahl67] – after all, we spent several months fine-tuning our algorithm to perfection, so we are not to blame.

Now seriously, Amdahl's law seems to be a universal law that prevents us taking advantage of increasing parallelism in our hardware. It's similar to how the speed of light constant is limiting progress in microprocessor design.

But, similar to how the hardware industry learned to avoid the restrictions imposed by physics by focusing on other aspects (like multithreading, multicores, caching, etc), the software industry can improve parallelism by changing the focus.

This article explores how we can change our focus in concurrent applications to drastically reduce the negative effects of Amdahl's law. It mainly aims at moving the focus off lock-based programming.

Amdahl's law and related formulas

Amdahl's law [Amdahl67] gives an upper bound to the maximum amount of speedup one algorithm[1] can obtain by increasing the parallelism level:

$$S_p \leq \frac{1}{(1-f)+\frac{f}{p}} \qquad (1)$$

where f is the fraction of the code that is parallelizable and P is the parallelism level (number of cores). The formula assumes that the parallelizable part is completely parallelizable.

Figure 1 shows the maximum speedup of an algorithm for parallelism factors of 50%, 80%, 90%, 95%, 99%.

A parallelism factor of 90% – that is, 90% of the algorithm can be parallelizable – sounds good at first glance. However, if we plug this value into Amdahl's formula, we obtain a maximum speedup of 5.26 for 10 cores – roughly half of what we naively expect. If the parallelism factor is 80%, then the speedup for 10 cores would be 3.57 – this doesn't seem good at all.

If we had an infinite amount of cores, a 90% parallelizable code would have a maximum speedup of 10. Similarly, an 80% parallelizable code would have a maximum speedup of 5. The intuition behind this is simple:

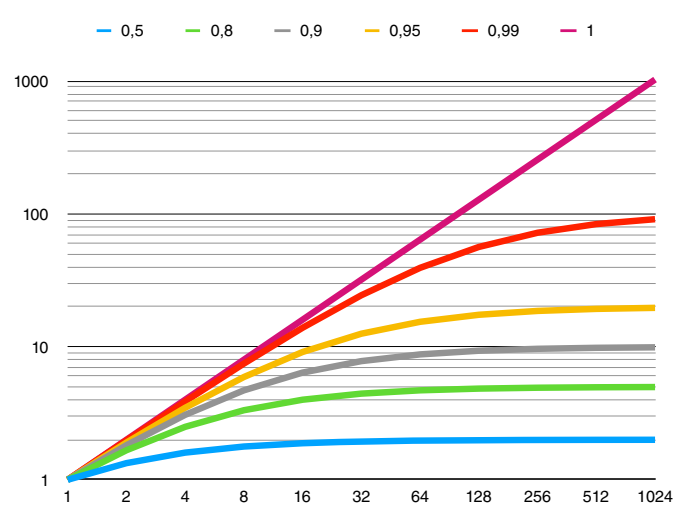


Figure 1

no matter how many cores we throw at the problem we can't improve the serial part; for the 90% parallelizable factor we can't improve 10% of the code no matter what.

If W_{ser} is the time needed to perform the work that cannot be parallelised at all, and W_{par} is the time needed for the parallelizable work, then the time taken if we have P cores is bounded by $T_P \geq W_{ser} + W_{par}/P$. If $P \to \infty$, then $T_P \approx W_{ser}$.

The time needed in the absence of parallelism, T_1, is called *work*. The time needed to run the algorithm if we assume infinite parallelism, T_∞, is called *span*. Similar to Amdahl's law, we can use *work* and *span* to calculate an upper bound to the speedup, somehow simpler to compute:

$$S_p = \frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} \qquad (2)$$

In Amdahl's law, W_{ser} is work that cannot be parallelised at all, while W_{par} is work that can be parallelised to a certain degree. Thus, we obtain an upper bound for speedup. Brent [Brent74] uses a slightly different division to arrive at a lower bound formula. He assumes that T_∞ is work that cannot be perfectly parallelizable and, thus, T_1-T_∞ is perfect parallelizable work. With this division he arrives at the following inequality:

$$T_p \leq \frac{T_1 - T_\infty}{P} + T_\infty \qquad (3)$$

1. I'm using the term *algorithm* here to denote the work that we consider parallelism for. This may not be the full program/application, but, by convention, we assume it's a significant part of it. I wanted to distinguish it from *application*, which contains one or multiple invocations of *the algorithm*. Beside the invocation of *the algorithm*, the application may have other parts that are not relevant for improving parallelism (e.g., startup, shutdown). Also, the use of the term *algorithm* should not be confused with implementations of well-known algorithms (like the ones found in standard libraries); we should use the general form (finite well-defined set of steps).

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. As hobbies, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

the unfortunate reality is that using locks is still mainstream – and these probably have the biggest negative impact on Amdahl's law

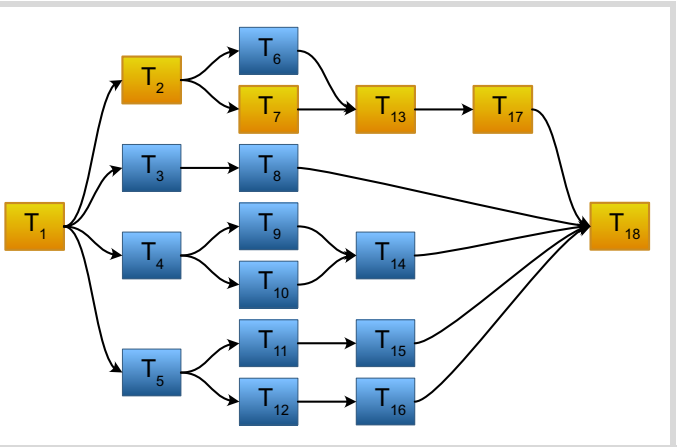


Figure 2

This puts a lower bound to the speedup we might have. All these formulas can be used to estimate the parallelism level of an algorithm. See [McCool12] for more theoretical details.

Let us take the example from Figure 2. It represents the work that an algorithm needs to make, with the explicit dependencies between different parts. The yellow (light) work units are on the critical path – they are the span of the algorithm. Assuming all the work units take 1 second to complete, $T_\infty = 6$, while $T_1 = 18$.

According to Amdahl's view, T_1 and T_{18} are not parallelizable, so it overestimates the possible speedup. The inequality with work and span gives an even tighter bound to speedup and Brent's lemma gives a lower bound to/for the speedup. They are all shown in Figure 3.

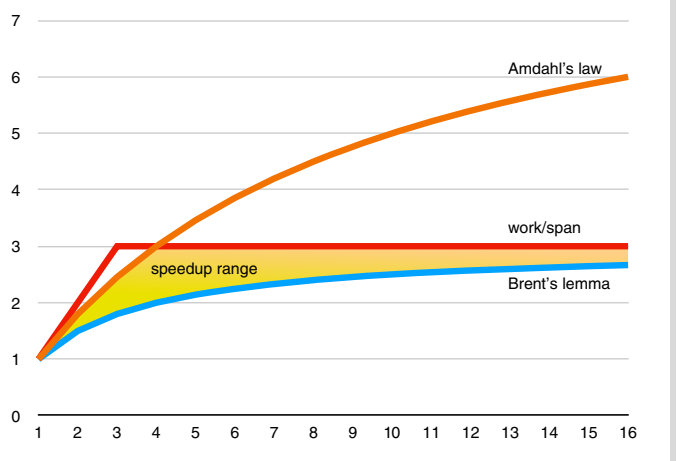


Figure 3

The classic multithreaded world

Let me be very clear on this one: in a multithreaded world, **mutexes provide exclusive access to evil**.

By design, mutexes block threads from executing certain regions of code; by design, they limit the throughput of the application; by design, they are bottlenecks. The reader should excuse me for putting it as bluntly as I can: after so much has been written on this topic, and after so many talks on this topic (see for example [Henney18], [Henney17], [Parent17], [Lee06]), if someone still believes that mutexes are not inefficiencies/bottlenecks then that person must be very confused. And the same goes with other lock-based primitives.

But the unfortunate reality is that using locks is still mainstream. And these probably have the biggest negative impact on Amdahl's law.

Whenever we distinguish between parallelizable and non-parallelizable code in an algorithm, we must accept that the non-parallelizable code comes from two main sources:

- Inherently serial code; i.e., sometimes we need to wait for the result of a computation before we can start another computation
- Code serialised through locks.

Of course, the ratio between the two depends on the application, but, in practice, we often find that locks are the main source of non-parallelizable code. In my experience working on an application that should have a sub-second reaction time, I've encountered (multiple times) cases in which threads are waiting to acquire locks for more than two minutes, and also cases in which lock chains have involved more than 10 locks (a lock waits on a lock, which waits on another lock, and so on). Typically, as soon as people add locks to a project, the performance quickly goes south.

We add locks to make our code *thread-safe*. This *thread-safe* terminology gives us a sense that everything is ok, including performance – and, of course, it's not. Instead, we should be saying that we add locks to become *thread-adverse*.

Let's assume that we have a unit of work that takes 1 second to execute. When there is no contention, the impact of adding the locks is minimal – and probably most people just measure this scenario. But, in the presence of contention, that unit of work can take 10%, 50%, or even 100% more time to execute. In Figure 4, we have an example of running a unit of work with locks in parallel; the example shows a performance degradation of 20% and 30%. And, what is worse, we may not be hitting all the non-parallelisable code in the example. Thus the factor in Amdahl's law can be even less than 70%. This can be a big impediment to scaling.

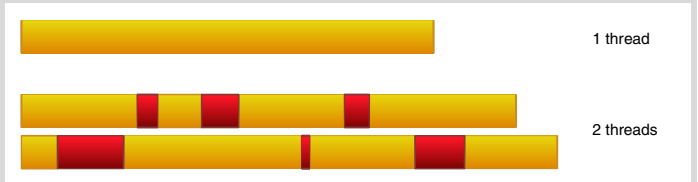


Figure 4

Two active tasks should never block each other. If two tasks are conflicting, they cannot be active at the same time

Another problem that we often encounter in the classical multithreaded world is the static allocation of work per thread. Applications, from build time, are configured with a fixed number of threads. Of course, if the machine has more cores than the number of threads, the application cannot properly scale. But also the opposite case can hurt performance: having too many threads on a limited number of cores can make the performance worse (i.e., because of thread switching and cache effects). Besides these two extremes, the static allocation of work may cause periods of times with not enough work.

So, in the classical multithreaded world, it's very easy to reach cases in which the ratio of parallelizable code is reduced, thus affecting the speedup of the application.

A change of perspective

To obtain significant speedups, we need to combat the negative effects of Amdahl's law (which provides an upper bound) and to take more advantage on Brent's lemma (which guarantees us a lower bound). That is, we need to reduce the amount of serial code, and increase the work/span ratio.

We would want to ensure that there is no contention between two units of work (tasks) that run in parallel, and moreover, at any given point in time, we have enough (better: the right amount) such tasks to execute. This is achievable by following a relatively simple set of rules:

- we break the algorithm into a set of tasks; a task is an **independent unit of work**
- we add constraints between tasks, modelling dependencies and possible race conditions
- we use a dynamic execution strategy for tasks; one worker thread per core
- we ensure that the algorithm is decomposed in enough tasks at every given point in time

The key here is the definition of tasks: the unit of work is *independent*. That is, two *active* tasks should never block each other. If two tasks are conflicting, they cannot be active at the same time. That is why we need to add constraints between tasks (second bullet). And, as the constraints between the tasks would have to be dynamically set (as we want our worker threads to always work on the next available task), we need to have a dynamic execution of our tasks. And, a simple greedy execution would work well in practice, as it would maximise the amount of work being executed.

And with that, we roughly solved the contention part between tasks. At this point I would ask the reader to accept that it is feasible, in practice, to implement such a system; I will not try to prove that we can always restructure the algorithms this way – let's leave this to a follow-up article.

The other part, ensuring that the work/span ratio is large enough, can be done through proper decomposition of the problem. Unfortunately, there is no general algorithm to solve this. There are a series of patterns that can be applied to parallel programming (a good catalogue of patterns can be found in [McCool12]), and experience so far tells us that this decomposition is doable in practice for most problems.

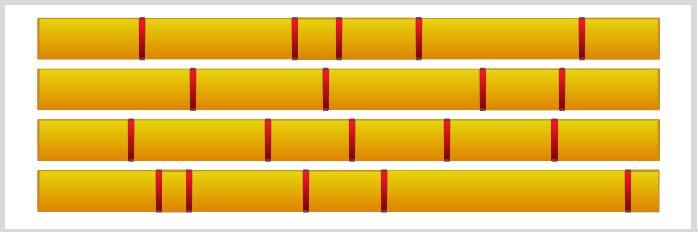


Figure 5

If both of these conditions are met, and for most of the problems can be met, we can count on a significant speedup. But let's work out the details.

Deriving the new formula

First, let us consider a simplified case in which, for the whole duration of the algorithm, we have running tasks, and we run them continually. This is depicted in Figure 5. We code with yellow the actual duration of the task and with red the time spent in the framework that executes tasks (overhead compared to static allocation of work).

The tasks are, by design, fully parallelizable, but in the real world they always have some indirect contention (i.e., caching effects, memory allocation, OS calls, etc.). But, let's assume for now that they can achieve perfect parallelism. That is, in our figure, at any point in which we have yellow bars we are achieving perfect parallelisation. After executing each task, we need to add some logic in the execution framework to choose the next task to execute and actually start executing it. Depending on the implementation of the framework code, this can be more or less parallelizable (a good implementation would have very low contention for average use cases). Let us denote this time as $W_{fw}=\alpha W_{tasks}$, assuming that it's a fraction of the time spent executing tasks.

If f_1 is the parallelisation ratio of the work associated with the tasks (considering the indirect contention), and f_2 is the parallelisation ratio for the task framework code, then the fully parallelizable part would be $W_{par}=W_{tasks}\,(f_1 + \alpha f_2) / (1 + \alpha)$, while the non-parallelizable part would be $W_{ser}=W_{tasks}\,(1 - (f_1+ \alpha f_2) / (1 + \alpha))$. Plugging this into Amdahl's law, we obtain:

$$S_p \le \frac{1}{1-\frac{f_1+\alpha f_2}{1+\alpha}+\frac{f_1+\alpha f_2}{P(1+\alpha)}} \qquad (4)$$

To be able to make sense of this formula, let's consider for the moment that $f_1 = 1$. Yes, this is not true in practice – but for this discussion, it is a good convention. We can consider that this is an inherent limitation in software construction, one that largely cannot be avoided. So, to assess the limits of our task-based system, we should not consider factors that are not controllable. It's similar to ignoring speed of light for non-relativistic mechanics.

even though that algorithm might have enough tasks overall, there may be times in which the algorithm doesn't have enough tasks to execute

Moving forward, to give an example, we can consider $f_2 = 0.5$, and α to be 1/1000 (i.e., on my machine I have a task system that has an overhead of the order of microseconds, per task). That would give us a general factor in Amdahl's law of 0.9995005. That is, for 1000 cores we would obtain a maximum speedup of 667. For 10 cores, the speedup limit would be 9.955. That is a great result! (deliberately ignoring indirect parallelism)

Ok, now it's time to consider the second limiting factor: we don't always have tasks to run. When we start the algorithm, we typically have only one task running; that task then creates and spawns the tasks that can be executed in parallel. Also, there are problems that may not expose high degrees of parallelism, so at various points in time, the number of available tasks can be less than the number of cores (undersubscription).

We would introduce β to be the ratio between the time we have more tasks than cores and the total time of the algorithm. With this, the global Amdahl's factor becomes:

$$\beta = \frac{f_1 + \alpha f_2}{1+\alpha} \tag{5}$$

and the new formula:

$$s_p \leq \frac{1}{1-\beta\frac{f_1+\alpha f_2}{1+\alpha}+\beta\frac{f_1+\alpha f_2}{P(1+\alpha)}} \tag{6}$$

At this point, we can also argue that, for certain algorithms and applications, we can make β be practically 1. If we care about the total application throughput, then the fact that our algorithm does have enough tasks at a given point may be irrelevant if there are other parts of the application with tasks. So, if we succeed in scheduling enough tasks around the algorithm, we can practically consider that $\beta = 1$.

Another argument for discounting the effect of the β factor relates to an assumption in Amdahl's law: we always assume that the total amount of work is constant, thus we measure against a constant work. But, whenever we discuss scalability and throughput, we oftentimes want to increase the problem size. That is, we are not interested in how much faster we can make one single computation of 1 second in complete isolation. We often consider the throughput of making many such computations. This means that we could have enough tasks at any time to fill the available cores.

That is, if we measure the speedup of the algorithm inside an application that is designed to maximise throughput, we can theoretically obtain a speedup of 667 for 1000 cores and 9.995 for 10 cores.

This is the upper limit as given by Amdahl's law. If we can shoot for that it's very good… let's shoot for that. But to do so, let us turn our attention to the lower limit of speedup given by Brent's lemma.

To simplify our calculation, let's assume that our work is divided in N equal tasks: $W_{tasks} = N \cdot W_0$. The amount of task framework work that needs to be done is also proportional to N, so the total work is $T_1 = N(W_0+W_{fw0})$. If we need K tasks for our span, then $T_\infty = K(W_0+W_{fw0})$. This means that $T_1 = N/K \cdot T_\infty$, and thus $T_P \leq (W_0+W_{fw0})(1+(N/K - 1)/P)$. This would give us a speedup of:

$$S_p \geq \frac{N}{K+\frac{N-K}{P}} \tag{7}$$

If we are targeting high-throughput, and we can hide the latency of the computation among a multitude of other computations, we can practically consider $K = 1$, and thus the formula becomes:

$$S_p \geq \frac{N}{1+\frac{N-1}{P}} \tag{8}$$

Again, this formula ignores some the fact that, in practice, it is hard to find perfect parallelisation, so it must be used with care.

This formula is important, as it gives us a guaranteed speedup (under all the assumptions we considered). For example, if N =1000, we have a minimum speedup of 500.25 for 1000 cores, and a minimum speedup of 9.91 for 10 cores).

I believe that these numbers will make most readers want to drop lock-based multithreaded programming and embrace task-based programming. If that's the case, I am personally very happy.

But before we directly jump on task-based programming, a contextualisation is needed.

Discussion

Having enough tasks to run. A key assumption we've made when deriving the above formula is that we have enough tasks to run at any given time; to be more precise, more than the number of available cores. This is known in the literature as *overdecomposition* [McCool12] – decomposing the problem into more tasks than we need. In multithreaded contexts, we should always aim for overdecomposition, but this is not always possible.

A major problem is that, even though that algorithm might have enough tasks overall, there may be times in which the algorithm doesn't have enough tasks to execute. This is mainly induced by the constraints we need to have on our tasks. If, for example, an algorithm has a long task that would spawn other tasks just at the end, then the algorithm would have times with undersubscription, which would hurt scalability.

Too much decomposition. From the point of view of this article, overdecomposition is highly encouraged. However, in practice, there are also costs associated with it. One cannot generate a near-infinite number of tasks. Tasks may be bound to resources (i.e., they need memory or special initialisation code) and we cannot afford to create too many of these tasks. This heavily depends on the algorithm being solved. For example, for an h264 video decoding a task, decoding a frame might need a context to run into; creating such a context might not be very cheap. So, applications may also want to limit the amount of decomposition.

If the average size of the task is similar to the overhead of the task framework, then half of the running time of the algorithm would be just on the task overhead

The size of the tasks matters. Our formula shows that the more tasks we have, the better the speedup will be. One would be tempted just to break the application down into very small tasks. But this may degrade the overall performance. Smaller tasks mean a larger α factor, that increases the overhead associated with managing the tasks. If, for example, the average size of the task is similar to the overhead of the task framework, then half of the running time of the algorithm would be just on the task overhead. As a rule of thumb, for the types of applications I work in, I would try to keep my tasks in the order of milliseconds: no less than 1ms, but no greater than 1s.

High-throughput scenarios. The whole article focuses on maximising throughput. But, as we well know, maximising throughput can lead to increased latency of the application. So, the advice of this article may not apply to low-latency applications.

Fixed work vs scalable work. Most of the formulas in this article revolve around the total work that an algorithm needs to do, but they are not very explicit. For example, Amdahl's law assumes a fixed amount of work; running the algorithm on more cores should not change the amount of work done. But this is generally not true in practice. First, moving from a single-threaded application to a multi-threaded application implies more work. Secondly, the amount of work is typically determined at runtime based on the actual execution trace. For the sake of simplicity, I left this discussion outside the article.

Beware of indirect contention. If we remove locks from our multithreading programming, indirect contention might become visible. And the more parallelism we add to our programs, the more this will be a problem. There are two parts on indirect contention: one inside our software (I/O, memory allocator, OS calls, use of atomics[2] etc.), and one outside of our software (operating system, hardware, other equipment, etc.). For example, the hyperthreading feature on a processor can limit the processing power as two threads can be fighting on the resources of the same core.

Conclusions and next steps

No matter what we do, we can't completely eliminate the limitations imposed by Amdahl's law; there are always factors that would limit the speedup of our code. But, we can shift our focus. Instead of focusing on external limitations in our software, we can change the paradigm in which we are writing multithreaded software. Changing this focus, we can eliminate the heavy intrinsic limitations of Amdahl's law.

If we focus on eliminating the locks from our software and we replace them with tasks, then, provided that we have enough tasks to be run at any given time and that the overhead of the task execution framework is small, we can achieve very good speedups. We analyse the upper bound speedup (starting from Amdahl's law) and the lower bound of the speedup (starting from Brent's lemma). In both cases, we converge to speedups that are bigger to what we typically see in current practice.

The one big question that the article doesn't explore is whether we can move to tasks for any type of problems. It only previews the main principles that would make the system work, but doesn't provide solid arguments on how this can be done. I can only assume that the reader is not fully convinced that such a system would be feasible for most applications. And that just binds me to write another article on this topic.

Until that time, the reader should at least start to be suspicious of locks found in the code. As a good tip, follow Kevlin Henney's advice and call all the mutexes 'bottlenecks' – after all it's the truth. ■

Appendix. A small example

I cannot finish this article without giving a code sample, without putting the theory into practice. Let us use a task-based system to compute the Mandelbrot fractal. The concurrency of the algorithm is implemented with the help of the Concore library [concore]. See Listing 1 overleaf. (I started to write this library to help me understand how to better write concurrent programs).

The problem is relatively simple: we have a matrix that needs to be filled with the number of steps until the series $Z_{n+1} = Z_n^2 + c$ diverges (well, approximated) [Wikipedia]. The elements are completely independent of each other. That allows us a straight-forward parallelisation. We chose to (explicitly) create a task for each row of the matrix. The reader will pardon my use of atomic here; it's a source of indirect contention, but I just wanted to keep this simple.

Running a simple benchmark on my MacBook Pro 16″ 2019, 2.6 GHz 6-Core Intel Core i7, I get the results shown in Figure 6. Considering all the disclaimers we've made during this article (not a perfect throughput test, indirect contention on memory, indirect contention with other processes, hardware, etc.) the results are in line with our calculations for up to 6 cores – which is also the number of non-hyperthreading cores I have on my machine (this is a proof that hyperthreading is a big source of indirect contention).

Speaking of indirect contention that cannot be controlled from software: during the run of the tests the operating frequency of my CPU drops as the number of worker threads go up. We just hit the limitations outside of software. And that just shows that focusing on obtaining the perfect Amdahl formula is a losing strategy.

References

[Amdahl67] Gene M. Amdahl (1967) *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings (30): 483–485 (Proceedings of the 18–20 April 1967 joint computer conference.)

[Brent74] R.P. Brent, 'The parallel evaluation of general arithmetic expressions', *Journal of the Association for Computing Machinery*, 21(2), 1974.

2. At the moment, I still use atomics in my day-to-day code while heavily avoiding locks. But atomics have essentially the same downsides as the locks do, just at a much finer scale. My hope is that after we are good at removing locks from our code, the next step would be to find systematic ways of removing the need for using atomics in high-level code.

No matter what we do, we can't completely eliminate the limitations imposed by Amdahl's law

```
int mandel_core(std::complex<double> c,
  int depth)
{
  int count = 0;
  std::complex<double> z = 0;

  for (int i = 0; i < depth; i++) {
    if (abs(z) >= 2.0)
      break;
    z = z * z + c;
    count++;
  }

  return count;
}

void serial_mandel(int* vals, int max_x,
    int max_y, int depth) {
  for (int y = 0; y < max_y; y++) {
    for (int x = 0; x < max_x; x++) {
      vals[y * max_x + x] =
      mandel_core(transform(x, y), depth);
    }
  }
}

void parallel_mandel(int* vals, int max_x,
    int max_y, int depth, concore::task& done) {
  std::atomic<int> remain{max_y};

  for (int y = 0; y < max_y; y++) {
    concore::global_executor([=, &remain,
        &done](){

      for (int x = 0; x < max_x; x++) {
        vals[y * max_x + x] =
        mandel_core(transform(x, y), depth);
      }

      if (remain-- == 1)
        concore::global_executor(done);
    });
  }
}
```

Listing 1

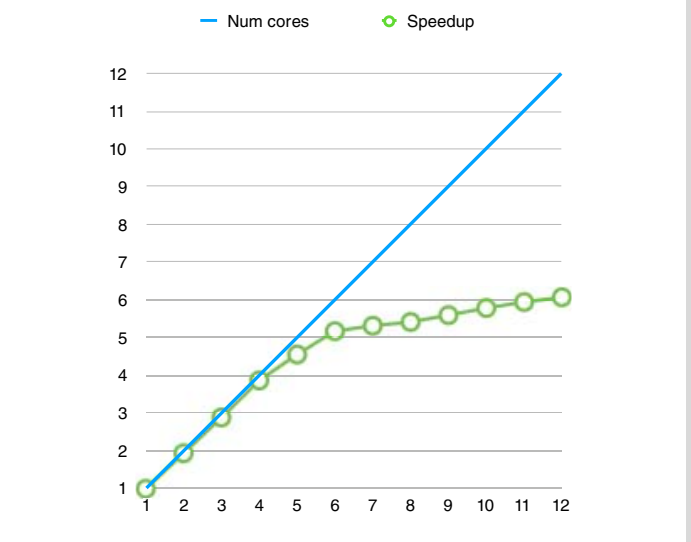


Figure 6

[concore] Lucian Radu Teodorescu, Concore library: https://github.com/lucteo/concore

[Henney17] Kevlin Henney, 'Thinking Outside the Synchronisation Quadrant', *ACCU 2017*, available at: https://www.youtube.com/watch?v=UJrmee7o68A

[Henney18] Kevlin Henney, 'Concurrency Versus Locking' *2 Minute Tech Tip*, available at: https://www.youtube.com/watch?v=mEtoXwB9HFk

[Lee06] Edward A. Lee (2006) *The Problem with Threads*, Technical Report UCB/EECS-2006-1, available at: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

[McCool12] Michael McCool, Arch D. Robison, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012

[Parent17] Sean Parent, *Better Code: Concurrency*, NDC London 2017, available at: https://www.youtube.com/watch?v=zULU6Hhp42w

[Wikipedia] Mandelbrot set: https://en.wikipedia.org/wiki/Mandelbrot_set

Some Objects Are More Equal Than Others

Comparing objects is a fundamental operation. Steve Love and Roger Orr consider different language approaches.

Testing for equality is an important concern in a lot of programming tasks and is often used for control flow: equality is one of the commonest expressions used in **if**, **for** and **while** statements. However despite being something that is covered in almost any introduction to a programming language the concept and implementation of equality can be quite complicated.

Possible meanings of 'equality'

There are a wide variety of meanings to the use of 'equality' in a programming language. The list of possible meanings includes:

1. Refer to the same memory location
2. Have the same value
3. Behave the same way

This article explores some of the details and pitfalls with equality in terms of just the first two items on this list. We found it was a harder task than it appears at first glance to get it right (for some definition of right), even ignoring the third item on our list or looking further afield for other meanings.

The first item in the list is often described as 'identity comparison' and the second one as 'value comparison', and we make use of these terms below. Note that value comparison usually refers to the *perceived* value for users of the object and fields that don't affect this (for example internally cached values) are usually not included in the comparison code.

We are further restricting the subject to focus primarily on only three languages: C++, C# and Java. Despite their common heritage and obvious similarities there are many differences in the sort of problems equality raises in each language: even at the basic level of language syntax we see:

- In Java: **a == b** always does something for all variables **a** and **b** of the same type (and compiles in some cases when they are of different types) and you cannot change what it does.

```
public class IntegerEquals
{
  public static void main(String[] args)
  {
    test(10);
    test(1000);
  }
  public static void test(int value)
  {
    System.out.println("Testing " + value);
    Object obj = value;
    Object obj2 = value;
    if (obj.equals(obj2))
       System.out.println("Equals");
    if (obj == obj2)
      System.out.println ("==");
  }
}
```

Listing 1

- In Java & C#: **anobject.[eE]quals(another)** always does something (we write **[eE]quals** because the method is spelled **equals** in Java but **Equals** in C#)
- In C++ & C# you can overload the meaning of **==** and in C# & Java you can override **[eE]quals** to customize behaviour.

Let's start with the language construct form of equality '**==**' on the grounds that this must be a pretty fundamental definition to have been enshrined in the syntax of the programming language, What does each language provide for this operator 'out of the box'?

In C++ '**==**' is predefined (as a value comparison) for all built-ins and the subset of the library types for which equality makes sense (e.g. **std::string**), but is not automatically provided for custom types defined in a program. However you can provide your own definitions of **operator==** as long as at least one argument is a custom type: and you can also specify your own return type for the operator (although returning anything but **bool** is usually a bad decision.)

In Java '**==**' is predefined for primitive built-ins and does a straightforward value comparison. For object types '**==**' performs identity comparison between the two objects supplied. You cannot change this behaviour.

In C# '**==**' is predefined, or overridden, for all built-ins and library types (whether these types are *reference* [**class**] or *value* [**struct**] types). It is not automatically provided for custom value types and performs identity comparison for custom reference types. C# lets you define '**==**' for any custom type, but you must additionally provide an implementation of '**!=**'.

Object comparisons

For Java and C# the presence of a single root class for all object types allows for a sensible definition of an equality method in this base class which takes an argument of the base class. In both languages the default implementation of this method, on custom types, performs identity comparison.

Java overrides **equals()** for some of the predefined types, such as **Integer**. However there is some confusing behaviour as Listing 1 demonstrates.

If you compile and run this simple program you might be surprised:

```
Testing 10
Equals
==
Testing 1000
Equals
```

Steve Love is a programmer who gets frustrated at having to do things twice. He can be contacted at steve@arventech.com

Roger Orr has been programming for far too long but still enjoys it far too much. Some of it is paid and some of it isn't. He can be contacted at rogero@howzatt.co.uk

unit tests ... will pass most tests whether you use the equals method or the == operator; but in actual use with runtime generated strings, the behaviour is different

```
public class Intern
{
  private static final String s1 = "Something";
  private static final String s2 = "Some";
  private static final String s3 = "thing";
  public static void main( String [ ] args )
  {
    if (s1 == s2 + s3)
       System.out.println("match!");
  }
}
```

Listing 2

The two objects `obj` and `obj2` compare the same using the `equals` method (as the overridden method in Integer compares values not identity) when executed with `value` set to 10 or 1000 as expected. However, on most implementations of Java, `obj` compares the ***same*** as `obj2` using `==` when value is set to 10 but they compare ***different*** when the value is set to 1000. What is happening here?

This is a consequence of an optimisation in the Java code that boxes primitive data into Integer objects. The compiler implements `obj == value` by calling `Integer.valueOf(value)` and this method caches 'commonly used values' such as 10[1]. Hence in the first case the compiler is performing identity comparison on two references to the same, cached, `Integer` with value 10 and in the second case the compiler is performing identity comparison on references to two *separate* temporary `Integer` objects with value 1000.

There is a similar problem with intern'ed strings (strings held in a shared pool of unique strings normally accessed using the `String.intern()` method) as demonstrated in Listing 2.

This program prints `match!` when executed as the compiler ensures that strings with the same compile time value generate references to a single object. This is perfectly safe since strings in Java are immutable, but can cause some confusion. In general checking strings for equality in Java with `==` is unsafe and some tools provide a warning for attempts to do so. The danger is that compile time strings, which are interned, are treated differently from any runtime strings (which typically aren't).

The classic case where this causes problems is that unit tests, which typically use compile time strings, will pass most tests successfully whether you use the `equals` method or the `==` operator; but in actual use with runtime generated strings (such as those read from a file) the behaviour is different.

C# implicitly provides some implementation assistance with the `Equals` method for value types, but it's more complicated than it might appear at first sight. (Listing 3)

Both these structures will have an `Equals` method synthesised by the compiler. The first class (`Easy`) only contains basic scalar members and

1. See http://download.oracle.com/javase/6/docs/api/java/lang/Integer.html#valueOf%28int%29

```
struct Easy
{
  int X;
  int Y[ 100 ] ;
}

struct Hard
{
  int X;
  MyType Y;
}
```

Listing 3

the `Equals` method will perform a bit-wise check on the two values (using the total size of the object), which is often exactly the desired behaviour (and is fast). In the case of the second class (`Hard`) the presence of the custom type `MyType` means that the synthesised `Equals` method performs reflection on the class at run time to identify the fields and then does a member-wise comparison of all the members (including the basic scalar `int` member `X`). While this produces the correct answer the performance is likely to be significantly worse than an explicit implementation of equality.

Finally in both C# and Java thought needs to be given to ensure the primary object reference is non-null. The simple example in Listing 4 demonstrates the problem and also a way (in C# only) to avoid it.

This program fails with a `NullReferenceException` as `a` is null in the first call to `Equals` and you cannot call a method on a `null` object. The second call, using the static method taking two arguments. does not throw such an exception when supplied with null references (and returns `false` if either `a` or `b` is `null` and `true` if they both are).

C++ does not have a single object root and so it doesn't really make sense to have an equals method, but it ***does*** have templates and to help with programming the STL there is `std::equal_to`, which by default performs `==`. You can specialise it for your own type to pass your own

```
public class NullEquals
{
  public static void Main()
  {
    object a = null;
    object b = new object();
    if (a.Equals( b ))
      Console.WriteLine("Now there's a thing");
    if (object.Equals(a, b))
      Console.WriteLine(
      "This should be safe enough");
  }
}
```

Listing 4

The trouble is that the overloaded method is called based on the **compile time type** of both **the primary object and the argument**

```
#include <functional>
#include <iostream>
int main( )
{
  std::cout << "std::equal_to<int>()(10,10): "
    << std::equal_to<int>()(10,10) << std::endl;
}
```

Listing 5

types to methods and classes implemented in terms of `equal_to` such as `std::unordered_map` (see Listing 5 for an example).

The full story for C# is even more complex as there is a long list of equality measures, which have been added to as various new versions of the .Net framework have been released. The list includes:

- `object.Equals` (we've already seen both flavours of this one)
- `object.ReferenceEquals`
- `IEquatable<T>`
- `IEqualityComparer`
- `IEqualityComparer<T>`
- `EqualityComparer<T>`
- `IStructuralEquatable`
- `StringComparer`

...and others we've probably missed...

The second element of this list, the `ReferenceEquals` method, is used to perform the identity check: that two references refer to the same object. The method is needed because `==`, which performs this check by default, can be overridden. (Since Java does not allow operator overloading it has no need for such a method.)

However, when used in conjunction with object boxing, `object.ReferenceEquals` has some interesting behaviour (see Listing 6).

This program prints `False` because the two temporary boxed integer objects created to pass into the `ReferenceEquals` method are distinct, and hence different, objects. This is a related problem to the one shown above using the Java `Integer` class.

```
public class RefEqual
{
  public static void Main()
  {
    int ten = 10;
    System.Console.WriteLine(
        object.ReferenceEquals(ten, ten));
  }
}
```

Listing 6

Overloading equality

Both Java and C# allow the programmer the freedom to overload the `[eE]quals` method to take an argument of a different type. Listing 7 is an example in Java that shows the problems of a naive implementation.

This program prints:

```
oe1.equals(oe2): true
oe1.equals(obj2): false
obj1.equals(oe2): false
obj1.equals(obj2): false
```

even though the **same** objects are being compared in each case. The trouble is that the overloaded method is called based on the **compile** time type of both the primary object and the argument. What you probably want in this case is logic based on the `runtime` type.

```
public class OverloadingEquals
{
  private int value;

  public OverloadingEquals(int initValue)
  {
    value = initValue;
  }

  public boolean equals(OverloadingEquals oe)
  {
    return oe != null && oe.value == value;
  }

  public static void main(String[] args)
  {
    OverloadingEquals oe1
       = new OverloadingEquals(10);
    OverloadingEquals oe2
       = new OverloadingEquals(10);
    Object obj1 = oe1;
    Object obj2 = oe2;
    System.out.println("oe1.equals(oe2): "
       + oe1.equals(oe2));
    System.out.println("oe1.equals(obj2): "
       + oe1.equals(obj2));
    System.out.println("obj1.equals(oe2): "
       + obj1.equals(oe2));
    System.out.println("obj1.equals(obj2): "
       + obj1.equals(obj2));
  }
}
```

Listing 7

There are some principles from the mathematics of 'equivalence relations'that, if adhered to, result in a consistent use of the concept of equality. They are that equality is...

- **Reflexive**

  a==a is always true

- **Commutative**

  if a==b then b==a

- **Transitive**

  if a==b and b==c then a==c

- **Reliable**

  Never throws. (This means checking for null!)

These rules are listed out in fuller detail in the language references for both C# [C# Equals] and Java [Java equals]. The wording from the C++ standard is short enough to quote in full: '(5.10p4) Each of the operators shall yield true if the specified relationship is true and false if it is false.' There you have it: succinct at any rate!

Now let us try and apply these rules when considering polymorphic equality. Consider a two-dimensional coordinate class in C# (Listing 8).

We might extend this class to support a three-dimensional coordinate system (Listing 9).

How does this polymorphic equality fare when checked against our four relationships for equality?

```
var p1 = new Coordinate { X = 2.3, Y = 5.6 };
var p2 = new Coordinate3d { X = 2.3, Y = 5.6,
                            Z = 10.11 };

p1.Equals( p2 ) is True
p2.Equals( p1 ) is False
```

Oops. The equality relationship fails the ***commutative*** requirement. We can improve our conformance to this requirement in C# by implementing `IEquatable<T>` – which enforces implementation of an override of `Equals` taking `T` – for both classes. This provides the symmetry for `p1` and `p2` but is still not a complete solution to the problem as this code fragment shows:

```
object o1 = p1;
Console.WriteLine(
   "p1.Equals(o2) {0}, o2.Equals(p1) {1}",
   p1.Equals(o2), o2.Equals(p1));
```

However, even if we fix the commutative relation by making our equality test more complex we *still* have a problem. Let's add this variable:

```
var p3 = new Coordinate3d {
   X = 2.3, Y = 5.6, Z = 1.22 };
```

Now `p1` will be equal to `p3` (for the same reason it is equal to `p2`), but `p2` and `p3` will *not* compare equal. We have broken the ***transitivity*** requirement. How can we resolve this? Should we even try?

Let's consider *why* we have the problems we see. Our problems are mostly caused by attempting to define equality in a class hierarchy. What sense is there to try and compare a two-dimensional and three-dimensional object? They are not the same class. The first solution is to change our design so that two and three dimensional classes are not related: we might use composition in preference to inheritance if we do wish to use some of the implementation of `Coordinate2d` in the implementation of `Coordinate3d`.

When inheritance is needed a good solution to the problematic elements of value equality is to allow comparison to succeed only if the actual run-time class types are the same, which can be implemented simply enough in C# by comparing the results of calling `GetType()` on each object.

Incidental and intentional equality

Avoid defining equality just so it can be used in conjunction with something that requires it, e.g. hashed containers. While it may make the initial implementation simpler to define 'just enough' equality to be able to use the type in this way, such partial implementations of equality have a nasty habit of causing more serious problems later on as the code evolves.

Suppose for example that you have a C# class and wish to create a `HashSet` of objects from this class. It can be tempting to define an `equals()` method on the class that fulfils just the checks necessary for this usage. However the equality used for a comparison in this context might be very different from one used elsewhere: perhaps only certain key fields are relevant. In this case an alternative way of solving the problem exists as the C# `HashSet` can use a pluggable equality comparer (`IEqualityComparer<T>`) instead of using `equals()`. This also provides a clearer way of stating the intent than implementing the equality operator just for using in the hash set. In C++ the `unordered_set` can be given its own equality comparer; however in the standard Java collection classes `HashSet` can only use `object.equals()`, so you're stuck with it.

Within a single application, *both* meanings of equality might be required: for example in an application for playing card games do you need **the** Ace of Spaces or **an** Ace of Spades? In Java and C#, override `[Ee]quals` for a value-check and leave `==` well alone to perform its default action of an identity check. In C++, which allows access to the address of an object, you can explicitly compare addresses (for identity) or contents (for value). Unless of course someone has defined `operator&` for one of the types...

Hashcodes

There is a close relationship between equality and hashing. For example the C# documentation states that 'classes [..] must [...] guarantee that two objects considered equal have the same hash code'. Java imposes a similar rule for `Object.hashCode()`.

```
class Coordinate
{
  public double X { get; set; }
  public double Y { get; set; }
  public override int GetHashCode()
  {
    // ...
  }
  public override bool Equals(object other)
  {
    var right = other as Coordinate;
    if (right != null)
      return X == right.X && Y == right.Y;
    return false;
  }
}
```

Listing 8

```
class Coordinate3d : Coordinate
{
  public double Z { get; set; }
  public override int GetHashCode()
  {
    // ...
  }
  public override bool Equals(object other)
  {
    var right = other as Coordinate3d;
    if (right != null)
      return base.Equals( other ) &&
        Z == right.Z;
    return false ;
  }
}
```

Listing 9

The reason is simple: when hashing functions are used with collections of objects the hash code is used first as a coarse filter to partition objects into buckets with the same, or related, hash codes. If you implement a hash code function that means two objects comparing equal have a different hash code then the two objects may end up in different buckets and the code won't ever get to the point of testing for equality.

Hash codes for objects that can mutate are another problem. See Listing 10 for an example.

Consider what happens if `Value` changes after inserting into a hashed container... if the object's hash code changes after being added to a hashed container, subsequent attempts to look for the object in the container will be accessing the wrong bucket.

The default implementation of `GetHashCode()` in C# for a value type is the hash code of the first field – this is rarely the best implementation for most value types. While we were investigating hash code behaviour in C# we found an interesting 'feature' of the Microsoft C# runtime: the hash code for a `boolean` value is constant! The program in Listing 11 demonstrates both these behaviours by printing `True` both times when compiled and run using Microsoft's implementation.

Using Visual Studio this program prints:

```
True
True
```

Collections

Another set of issues is raised by considering equality on container types. When are two collections of things equal? Is it enough that the two containers have the same items or do they need to be in the same order? (As a side note, we can add to the C# list of equality checks with `SequenceEqual`, which insists on the same items, in the same order).

This is a question that has performance implications too: comparing two sets are equal when permutations are allowed has a higher complexity measure than the case when the ordering must match.

A further question that may need addressing with containers is whether you want a value or reference comparison: do two containers match if they contain the identical objects or if they contain objects with identical values?

Note that this is a case where polymorphic equality makes a lot of sense: two collections are equal when they contain the same objects. You are not usually interested in whether they are from the same class (or even whether the internal states are the same); the important thing for equality is the objects they contain.

Conclusion

Equality is hard to define simply even for a single language. It is easy to implement if you stick to a small set of common sense rules; more complicated implementations are possible but not in general recommended.

One key distinction is between values and references. You should know the difference between (polymorphic) reference types and value types in all languages and avoiding treating the two the same way! Equality for references is a check for identity but equality for value types is a check for equal values of all (significant) fields.

Making use of immutability for value types has many benefits, far beyond equality. In the case of equality though it allows for the possibility of caching of objects and/or values and it also removes the class of problems exemplified by the example of modifying an object while it is held in a collection.

Using value equality in a class hierarchy rarely makes sense and should be avoided. It is often better to avoid inheritance in the sort of cases where equality might make sense and use composition instead. Classes can also be made `final` (or `sealed`) to prevent unwanted inheritance but this can be an annoyance when a user of the class has a valid reason for wanting to extend your class. ■

Further reading

C# in a Nutshell has a deep exploration of equality in C#. For more about equality in Java see http://www.javapractices.com, and follow links through Overriding Object methods to implementing equals.

Angelika Langer and Klaus Kreft wrote a pair of articles on the subject [Langer]. While the target of their article is Java many of the points apply to C# as well.

References

[C# Equals] http://msdn.microsoft.com/en-us/library/bsc2ak47%28v=VS.100%29.aspx

[Java equals] http://download.oracle.com/javase/6/docs/api/java/lang/Object.html#equals%28java.lang.Object%29

[Langer] http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html

```
public static class Bogus
{
  public String Value;
  @Override public int hashCode()
  {
    return Value.hashCode();
  }
  @Override public boolean equals( Object other )
  {
    return ((Bogus)other).Value.equals( Value );
  }
}
```

Listing 10

```
using System;
static class Program
{
  struct HashTest
  {
    public bool Enabled;
    public string Value;
  }

  public static void Main()
  {
    var h1 = new HashTest{
       Enabled = true, Value = "Great!"};
    var h2 = new HashTest{
       Enabled = false , Value = "Great!"};
    Console.WriteLine(
       h1.GetHashCode() == h2.GetHashCode());
    h1.Value = "Rubbish!";
    Console.WriteLine(
       h1.GetHashCode() == h2.GetHashCode());
  }
}
```

Listing 11

This article is a reprint. It was first published in *Overload* 103 (June 2011). Some topics are always relevant, even as languages and practices change.

Afterwood

Mission statements are all the rage. Chris Oldwood considers whether there are any benefits to omission statements.

I'm a big fan of quotes. Not the funny little marks we use to enclose strings in source code and speech in prose, or use inappropriately when distinguishing between plurals and possession, but those tiny utterances which are either written or spoken and yet manage to convey much more. One man who has provided so many of the latter and yet equally suffered from the malaise of the complex rules of the apostrophe is Fred Brooks. (I constructed that sentence very carefully to avoid that particular trap and therefore save valuable time for the editor and reviewers.) From his vast catalogue of memorable writing, I can easily single out one quote which sums up a career in programming without passing judgement on those who find themselves on the wrong side of the tracks:

> Good judgement comes from experience, and experience comes from bad judgement.

Relayed in its entirety, it stands as a shining example of how even failure can be seen in a good light – a continuous source of learning – so long as the culture surrounding us provides us with the safety we need to experiment.

Sadly, not all quotes have the structure that allows them to remain intact or at least to withstand the continual need to compress and distil a concept into as few words as possible. This reductionism works pretty well in the Design Patterns community, where we have managed to reduce many common solutions to repeating problems in software design down to just a few words. In the case of the Gang of Four, they reached the holy grail of just a single word for a number of fundamental concepts and gave us the power to communicate volumes with a ubiquitous language. In time, they even gave us the power to evoke a complex emotional response simply by uttering the word 'singleton'.

In one of her many (self-confessed) failed attempts to write an editorial, the *Overload* editor reminded us of those immortal words from Strunk & White, which no doubt play a significant role in the crusade to eliminate the superfluous – 'omit needless words'. Given the prevalence of soot in the printing process (as an ingredient of ink) you could make an argument that following this advice would lead you to a reduction in your carbon footprint too. Fran amusingly points out that with undue care you can take this too far and quickly lose the essence, creating a monster in its wake. Needless to say that naively paraphrasing Strunk & White as 'omit words' is not a zero sum game.

I'm sure there are people who feel that that famous six-word story from Ernest Hemingway about unworn baby shoes on eBay can be improved upon, although that wasn't it. I've definitely been in project meetings where Kent Beck's succinct expression of how late you can leave a decision, namely the 'last responsible moment', has undoubtedly been irresponsibly re-interpreted as the 'last moment', at which point a mad scramble takes place to continue to hit the (no doubt artificial) deadline. From the vitriol that appears regularly on the Internet, you could be mistaken for believing that Beck was also responsible for suggesting that we merely 'do the simplest thing' and forgo any thought or planning (the consequences be damned). After all, what possible extra value does '...that could possibly work' add to such a proposition – who would be stupid enough to trade-off 'simplicity' over 'working'? Maybe the very same people that would prefer to just 'Keep It Simple' and kiss goodbye to correctness.

Sir Tony Hoare is definitely no stranger to this phenomenon. I wonder if our industry's billion pound mistake is an inability to write quotes out in full. The poster child for this whole affair is almost certainly the continued debate around whether or not someone is 'prematurely optimising'. Delivered by itself, Hoare may as well have written: "we should forget about full quotations, say about 97% of the time". I guess some quotes need to be made so simple they have no obvious deficiencies.

Sometimes it feels like there is a Newton's Third Law of Truism – for every quotation there is an equal and opposite (misguided) interpretation. This makes some conversations seem like a game of Quote Trumps™ where you try to make a succinct point and then your colleague feels the need to respond with a 'superior' quote in the hope of scoring more points. For example, 'the customer is always right' is almost always 'trumped' by that old classic (which is commonly attributed to Henry Ford) about the customer really wanting 'faster horses'. This quote is almost always used to justify why developers know better than their customers and therefore give themselves the right to explore some shiny new technology rather than spend more time understanding what problem the customer is really trying to solve. Sometimes the problem is to discover what the problem is.

I don't believe it's limited to our industry but maybe we're just in more of a rush than anyone else and are therefore more content to skip the research and critical thinking phases and move straight on to the executive summary. Mind you, that expression is now considered overly verbose too and has been undercut by the new kid on the block, 'TL;DR'. How long before we capture this sentiment within a single emoji and continue to prematurely optimise reading time over comprehension?

Quotations are written for people to read and only incidentally for inclusion in motivational books (or for comedic effect). The moment we start to paraphrase, we run the risk of subtly changing the meaning in such a way that our audience will miss the point, with the catastrophic effect that they may even take away the opposite view to what was intended by the original author; in the wrong hands quotes could be considered harmful.

For those less familiar with Strunk & White, they need look no further than Hoare himself, as he was the one who suggested that inside every large program is a small one struggling to get out. Reductionism is therefore an essential process but how do we ensure that we make things as simple as possible, but no simpler? I fear there is no silver bullet... ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe