Chapter

4

Concurrency

When it comes to concurrency, we are living the proverbial interesting times more than ever before. Interesting times come in the form of a mix of good and bad news that contribute to a complex landscape of tradeoffs, forces, and trends.

The good news is that density of integration is still increasing by Moore's law; with what we know and what we can reasonably project right now, that trend will continue for at least one more decade after the time of this writing. Increased miniaturization begets increased computing power density because more transistors can be put to work together. Since components are closer together, connections are also shorter, which means faster local interconnectivity. It's an efficiency bonanza.

Unfortunately, there are a number of sentences starting with "unfortunately" that curb the enthusiasm around increased computational density. For once, connectivity is not only local—it forms a hierarchy [12]: closely connected components form units that must connect to other units, forming larger units. In turn, the larger units also connect to other larger units forming even larger functional blocks, etc. Connectivity-wise, such larger blocks remain "far away" from each other. Worse, increased complexity of each block reclaims increased complexity of connectivity between blocks, which is achieved by reducing the thickness of wires and the distance between them. That means an increase of resistance, capacity, and crosstalk. Resistance and capacity worsen propagation speed in the wire. Crosstalk is the propensity of the signal in one wire to propagate to a nearby wire by (in this case) electromagnetic field. At high frequencies, a wire is just an antenna and crosstalk becomes so unbearable that serial communication increasingly replaces parallel communication (a somewhat counterintuitive phenomenon visible at all scales—USB replaced the parallel port, SATA replaced PATA as the disk data connector, and serial buses are replacing parallel buses in memory subsystems, all because of crosstalk. Where are the days when parallel was fast and serial was slow?)

Also, the speed gap between processing elements and memory is also increasing. Whereas memory density has been increasing at predictably the same rate as general integration density, its access speed is increasingly lagging behind computation speed for a variety of physical, technological, and market-related reasons [14]. It is unclear at this time how the speed gap could be significantly reduced, and it is only growing. Hundreds of cycles may separate the processor from a word in memory; only a few years ago, you could buy "zero wait states" memory chips accessible in one clock cycle.

The existence of a spectrum of memory architectures that navigate different tradeoffs between density, price, and speed, has caused an increased sophistication of memory hierarchies; accessing one memory word has become a detective investigation that involves questioning several cache levels, starting with precious on-chip static RAM and going possibly all the way to mass storage. Conversely, a given datum could be found replicated in a number of places throughout the cache hierarchy, which in turn influences programming models. We can't afford anymore to think of memory as a big, monolithic chunk comfortably shared by all processors in a system: caches foster local memory traffic and make shared data an illusion that is increasingly difficult to maintain.

To make us sweat even more, heat issues also join the fray. Increased density means smaller transistors and consequently lower power draw per transistor, but it also means more transistors within the same space need to be fed with power, leading to an overall trend towards increased heat emission. The silicon substrate sustaining the circuitry needs to dissipate ever larger amounts of heat—another trend that scales the wrong way.

In related, late-breaking news, the speed of light has obstinately decided to stay constant (`immutable` if you wish) at about 300,000,000 meters per second. The speed of light in silicon oxide (relevant to signal propagation inside today's chips) is about two thirds that; and the speed we can achieve today for transmitting actual data is significantly below that theoretical limit. That spells more trouble for global interconnectivity at high frequencies. If we wanted to build a 10 GHz chip, under ideal conditions it would take two cycles just to transport a bit across a 4-centimeter wide chip while essentially performing no computation.

In brief, we are converging towards processors of very high density and huge computational power, that are however becoming increasingly isolated and difficult to reach and use due to limits dictated by interconnectivity, signal propagation speed, memory access speed, and heat dissipation. The computing industry is naturally flowing around these barriers. One phenomenon has been the implosion of the size and power required for a given computational power; today's addictive portable digital assistants could not have been fabricated at the same size and power with technology only five years old.

Today's trends, however, don't help traditional computers that want to achieve increased computational power at about the same size. For those, chip makers decided to give up the battle for faster clock rates, and instead decided to offer computing power packaged in already-known ways: several identical CPUs connected to each other and to memory via buses. Thus, within an incredibly short time, the responsibility for making computers faster has largely shifted from the hardware crowd to the software crowd. More CPUs may seem like an advantageous proposition, but for regular desktop computer workloads it becomes tenuous to gainfully employ more than around eight processors. To speed up one given program, a lot of hard programming work is needed to put those CPUs to good use. Since only a short time ago, taking a vacation is not an option for increasing the speed of your program.

Computing industry has always had moves and shakes caused by various technological and human factors, but this time around we seem to be at the end of the rope. Interesting times indeed.

4.1 Look Ma, No Default Sharing

One startling aspect of the undergoing shift happening in computing is the speed with which processing and concurrency models are changing, particularly in comparison and contrast with the pace of development of programming languages; it may take a decade for a language to become compelling, whereas concurrency matters underwent several dramatic changes in the same amount of time.

For example, our yesteryear understanding of threading was centered around time-slicing: one processor uses a timer interrupt in conjunction with a thread scheduler. Upon each timer interrupt, the scheduler decides which thread gets processor time for the next time quantum, thus giving the illusion that many threads are running simultaneously, when in fact they all use the same processor. In those days, sharing memory across threads was as straightforward as possible—one thread writes, another reads. Today, even run-of-the-mill machines have more than one processor, which changes everything. Furthermore, the notion of sharing becomes increasingly blurred by the existence of sophisticated cache synchronization protocols. We need to change a lot about our assumptions on how threads work. Consider the following piece of advice given in the first edition of the excellent book "Effective Java" [4, Item 51, p. 204]:

> When multiple threads are runnable, the thread scheduler determines which threads get to run and for how long. [...] The best way to write a robust, responsive, portable multithreaded application is to ensure that there are few runnable threads at any given time.

The book was written in 2001. One startling detail for today's observer is that single-processor, time-sliced threading is not only used, but assumed without stating. Naturally, the second edition[1] [5] changes the advice to "ensure that the average number of runnable threads is not significantly greater than the number of processors." Interestingly, even that advice, although it looks reasonable, makes a couple of unstated assumptions: one, that there may be high contention between threads; and two, that the

[1]Even the topic title was changed from "Threads" to Concurrency" to reflect the fact that threads are but one concurrency model.

number of processors does not vary dramatically across machines that may execute the program. As such, the advice is contrary to that given in the "Programming Erlang" book [3, Ch. 20, p. 363]:

> **Use Lots of Processes** This is important—we have to keep the CPUs busy. All the CPUs must be busy all the time. The easiest way to achieve this is to have lots of processes.[2] When I say lots of processes, I mean lots in relation to the number of CPUs. If we have lots of processes, then we won't need to worry about keeping the CPUs busy.

Which recommendation is correct? As usually, it all depends. The first recommendation works well on 2001-vintage hardware; the second works well in scenarios of intensive sharing and consequently high contention; and the third works best in low-contention, high-CPU-count scenarios. Such recommendation closely reflect fundamental choices made by the respective languages, and one topic of particular importance is the approach to memory sharing.

Today's mainstream imperative languages have been developed in the good old days of simple memory architectures and straightforward sharing. Naturally, they modeled the realities of that hardware by defining threads that by default "see" the same memory. After all, the very definition of multithreading entails that all threads share the same address space, unlike operating system processes.

As mentioned, today's deep memory hierarchies are making sharing of memory across processors an expensive illusion. To ensure that a piece of data written by one thread is visible by another, a handshake is needed that ensures the data was properly propagated. Such handshakes slow down both processors involved. Furthermore, since transfers across the levels of memory hierarchies occur in blocks, the order in which data is read might not correspond to the order in which data was written. These effects conspire together to make approaches to programming based on memory sharing extremely pedestrian and error-prone.

Functional languages deem all mutation as undesirable and strive for "purity" defined as the absence of side effects.

[2] Erlang processes are distinct from OS processes.

Bibliography

[1] Suad Alagić and Mark Royer. Genericity in Java: persistent and database systems implications. *The VLDB Journal*, 17(4):847–878, 2008. (cited on p. 244)

[2] Jonathan Amsterdam. Java's `new` considered harmful. *Dr. Dobb's Journal*, April 2002. `http://www.ddj.com/java/184405016`. (cited on p. 220)

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. (cited on p. 44)

[4] J. Bloch. *Effective Java programming language guide*. Sun Microsystems, Inc. Mountain View, CA, USA, 2001. (cited on pp. 331, 43)

[5] J. Bloch. Effective Java, Second Edition. 2008. (cited on p. 43)

[6] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008. (cited on p. 216)

[7] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Commun. ACM*, 9(5):366–371, 1966. `http://doi.acm.org/10.1145/355592.365646`. (cited on p. 6)

[8] Walter Bright. The D assembler. `http://digitalmars.com/d/1.0/iasm.html`. (cited on p. 98)

[9] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995. (cited on p. 209)

[10] Brian Cabana, Suad Alagić, and Jeff Faulkner. Parametric polymorphism for Java: is there any hope in sight? *SIGPLAN Notices*, 39(12):22–31, 2004. (cited on p. 244)

[11] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997. (cited on p. 54)

[12] Richard Chang. *Near Speed-of-Light On-Chip Electrical Interconnects*. PhD thesis, Stanford University, 2003. (cited on p. 41)

[13] Tal Cohen. Java Q&A: How Do I Correctly Implement the `equals()` Method? *Dr. Dobb's Journal*, May 2002. `http://www.ddj.com/java/184405053`. (cited on p. 216)

[14] U. Drepper. What every programmer should know about memory. *Eklektix, Inc., Október*, 2007. (cited on p. 42)

[15] R.B. Findler and M. Felleisen. Contracts for higher-order functions. *ACM SIGPLAN notices*, 37(9):48–59, 2002. (cited on p. 14)

[16] J. Friedl. *Mastering regular expressions*. O'Reilly Media, Inc., 2006. (cited on p. 31)

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. (cited on pp. 164, 206, 220)

[18] Darryl Gove. *Solaris Application Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008. (cited on p. 326)

[19] Daniel M. Hoffman and David M. Weiss, editors. *Software fundamentals: collected papers by David L. Parnas*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (cited on p. 209)

[20] Donald E. Knuth. *The Art of Computer Programming Vol. 2*. Addison-Wesley, 1997. (cited on p. 176)

[21] J.K. Korpela. *Unicode Explained*. O'Reilly Media, Inc., 2006. (cited on p. 128)

[22] B. Liskov. Keynote address—data abstraction and hierarchy. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 17–34. ACM New York, NY, USA, 1987. (cited on p. 29)

[23] Digital Mars. dmd—freebsd d compiler. `http://digitalmars.com/d/2.0/dmd-freebsd.html`, 2009. (cited on p. 41)

[24] Digital Mars. dmd—linux d compiler. `http://digitalmars.com/d/2.0/dmd-linux.html`, 2009. (cited on p. 41)

[25] Digital Mars. dmd—osx d compiler. `http://digitalmars.com/d/2.0/dmd-osx.html`, 2009. (cited on p. 41)

[26] Digital Mars. dmd—windows d compiler. `http://digitalmars.com/d/2.0/dmd-windows.html`, 2009. (cited on p. 41)

[27] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, October 2002. (cited on p. 21)

[28] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. (cited on pp. 183, 220, 14)

[29] Scott Meyers. How non-member functions improve encapsulation. *C++ Users Journal*, 18(2):44–52, 2000. (cited on p. 211)

[30] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *CONFERENCE RECORD OF THE ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, volume 24, pages 146–159. Citeseer, 1997. (cited on p. 146)

[31] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. (cited on p. 209)

[32] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972. (cited on p. 14)

[33] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. (cited on pp. 183, 53)

[34] Rob Pike. Utf-8 history. `http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt`, 2003. (cited on p. 129)

[35] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge University Press New York, NY, USA, 2007. (cited on p. 180)

[36] Atanas Radenski, Jeff Furlong, and Vladimir Zanev. The Java 5 generics compromise orthogonality to keep compatibility. *J. Syst. Softw.*, 81(11):2069–2078, 2008. (cited on p. 244)

[37] International Standard. Programming languages—C. *ISO/IEC 9899:1999(E)*, 1999. (cited on p. 40)

[38] Alexander Stepanov. Interview for Edizioni Informedia srl. `http://stlport.org/resources/StepanovUSA.html`. (cited on p. 27)

[39] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical report, WG21 X3J16/94-0095, 1994. (cited on p. 164)

[40] Herb Sutter. Virtuality. *C/C++ Users Journal*, September 2001. `http://www.gotw.ca/publications/mill18.htm`. (cited on p. 222)

[41] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series).* Addison-Wesley Professional, 2004. (cited on p. 222)

[42] The Unicode Consortium. *The Unicode Standard, Version 5.0.* Addison-Wesley Professional, Reading, MA, USA, 2006. (cited on pp. 128, 38)

[43] J. Von Ronne, A. Gampe, D. Niedzielski, and K. Psarris. Safe bounds check annotations. *Concurrency and Computation: Practice and Experience*, 21(1), 2009. (cited on p. 106)

[44] P. Wadler. Proofs are programs: 19th century logic and 21st century computing. *Dr Dobbs Journal. Variant of New Languages, Old Logic in DDJ December 2000 see*, 2000. (cited on p. 13)

[45] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. *SIGPLAN Not.*, 44(1):41–52, 2009. (cited on p. 14)