Computing, Then and Now

(July 8, 2020)

Computers are inherently parallel.

Early on, hardware designers found ways to deal with the inherent parallelism (e.g. TTL, state machines, clocking, etc.).

At an early stage, von Neumann suggested using a single-threaded approach in order to tame parallelism in software (N.B. note the similarity to consciousness - a single thread, coordinating the parallel processes of the human body)

At that time, the assumptions were:

- memory was a scarce resource and needed to be conserved and reused
- CPUs were very expensive and needed to be time-shared across multiple users
- mathematical notation (as it existed) could solve all problems
- programming languages were hard to create.

Today, these basic assumptions have been overturned

- we have nearly infinite memory
- microprocessors are cheap and abundant
- written mathematical forms do not use the full capabilities of what can be displayed with a computer
- programming languages (e.g. DSLs) are much easier to create now, e.g. using PEG and backtracking parsers.

The computing environment has changed drastically, but software programming languages have not kept up with the new reality.

We continue to use languages whose design was based on these early prejudices. This has led us into many accidental complexities that continue to vex us. The most glaring of such accidental complexities are time-sharing-based multi-tasking and memory sharing (I count some 40+ accidental complexities).

We have developed a language, phrases and words to describe the problematic aspects of these choices, instead of dealing with the new reality and problems based on the early assumptions. We talk of thread-safety, priority inversion, mmap, race conditions (some race conditions are inherent in parallel systems, but many of the race conditions we deal with are due to accidental complexity), garbage collection, parameter lists, return values, the vague notions of *complexity,* Agile, call-return, etc., all the while ignoring realities that TTL hardware dealt with – throughput time, asynchronous design, 1-page documentation, encapsulation, buses, etc.

We use programming editors that can edit only text (which are graphical glyphs made up of mini-bitmaps) instead of programming with graphical symbols, of which text is only *one* of the choices.

We have created variable font editors for business (e.g. Word), we have created diagram editors for business (e.g. Visio) and we have invented new UIs for business (Excel, iPad), but the basic model of editors for *programming* has not kept pace. Programmers use all ten fingers for typing, but programmers are forced to take their hands off of keyboards to use much more limited mouse pointing devices. For example, a *box* glyph could be bound to a single key on the keyboard[1] and two *boxes* could be selected (e.g. using *point* and *mark*) with another keystroke creating a line/connection between them. Snippets of text could be typed anywhere on the screen and not be relegated to the 24x80 line-oriented mindset of early predecessors. Technologies, once thought forbidden, like PROLOG backtracking, Early parsing, miniKanren (core.logic in clojure-speak) are now entirely viable and finish processing in the blink of an eye. We *can* waste computing resources to make programming easier, but instead we waste resources on tool-tips and mountains of APIs based on outdated notions of function libraries.

Most programming languages (and even tools like spreadsheets) use the outdated notion of *absolute addressing* – where functions are named and called directly instead of using indirection (which is much more flexible for architecting new solutions). Hardware microprocessors underwent a transition from absolute addressing to relative addressing, but this has mostly not happened in the design of popular programming languages. (See David Ackley's MFM for someone who is thinking along these lines (albeit mostly for hardware, AFAICT) https://www.cs.unm.edu/~ackley/papers/hotos-11.pdf).

We spend time improving code tools instead of building new tools and languages for harder problems, like Architecture (which I call D.I. - Design Intent). We relegate D.I. to whiteboards instead of concrete notations which can be compiled and executed. We build languages that only mathematicians can only love, instead of building tools to concretely communicate between CEOs, CTOs, Architects, Engineers and Programmers.

We have essentially *forgotten* that synchronous code is a *trick* used as *but one way* to tame parallelism.

We expect all languages to provide call/return, parameters, return values, exceptions, etc. and we try to force-fit these ideas onto every problem we solve ("*when all you've got is a hammer, then everything looks like a nail"*). The result is that we know how to solve only one problem (building unreliable websites). The rest of the problems in the real world (gaming, machine control, etc.) are left to C and the new kid on the block Rust (which is just a re-hash of old ideas).

The new reality consists of applications consisting of multiple microprocessors (I refuse to call them C.P.U.s - Central Processing Units – and call them P.U.s instead (Processing Units)) each with their own private memories, distributed across space. One thread per P.U. Communicating over wires (not memory).

Current programming languages, stack-based, can describe a program *on* a P.U. but cannot (easily, without accidental complexity) describe programs *across* distributed P.U.s.

We need languages that *do less* so that Architects and Engineers can free their minds of unnecessary details (I discuss my ideas about how to structure a Software Organization elsewhere).

---

1 Or, a single gesture consisting of multiple keys, or multiple gestures, or, just about anything that is repeatable and stays the same (modeless).

We have explored all sorts of tools and technologies and paradigms. Now, we need to pick through the pile and select the best tools for every task and throw away the rest (see https://alarmingdevelopment.org/?p=766).