Concurrency Is Not Parallelism

Central Point

Concurrency is a programming *paradigm*.

Parallelism is an application problem, *not* a paradigm. Parallel programs must use the concurrent paradigm. Concurrent programs, though, are not necessarily parallel.

Rob Pike's Talk

https://vimeo.com/49718712

Conflating Concurrency and Parallelism

I used to conflate parallelism and concurrency until I watched Rob Pike's talk.

I used to try to convince people that they could write concurrent programs using something "better" than Processes (Threads). What I didn't realize was that I was trying to explain the difference between the paradigm called "concurrency" and the application problem called "parallelism".

A parallel program *must* run on multiple processors.

A program designed using the concurrent paradigm can run on a *single* Processor or on multiple processors.

The advantages of using the concurrent paradigm can be seen even on a single processor system.

One can write concurrent software that uses only a single Processor. One cannot write parallel software that uses a single Processor. One can *simulate* parallel software on a single processor, but this is not necessary (although it might help debugging, while hindering debugging).

Multi-tasking – as we know it today – is a *simulation* of parallelism. Multi-tasking, as we know it today, tries valiantly to solve too many problems at once – e.g. memory sharing, bandwidth optimization, time-sharing, etc. Multi-tasking as we know it today, is a simulation of parallelism that attempts to optimize concurrency using multiple stacks (a form of memory sharing optimization + processor optimization and sharing).

Implementing the Concurrent Paradigm

A concurrent system is composed of a number of *closures*. There is a single distinguished routine, called the Dispatcher(). The Distpacher() invokes closures, at random.

That's all there is.

It is simple – almost too simple – to implement concurrency in any language that supports *closures*. It is possible to implement the concurrent paradigm in C or assembler, too, one just needs to pay more attention to details.

As simple as this sounds, the concurrent paradigm forces one to program in a completely different manner. The program cannot rely on synchrony. The program – composed of many communicating closures – must be written in a way that acknowledges that *any* closure might be awakened at *any* time (by the Dispatcher()).

This seemingly simple requirement causes programmers to re-evaluate control-flow coupling.

Operating system Processes and threading libraries are heavy-handed approaches to creating closures and a Dispatcher(). In some, *but not most*, cases it is necessary to use hardware to protect one program from another (e.g. MMUs). In some, *but not most*, cases it is necessary to optimize memory usage using memory-sharing and stacks (stacks are just an optimized form of linked lists). In some, *but not most*, cases it is necessary to use time-sharing. When you strip these features out of threading libraries, you get multi-tasking that is easy. Closures have been around for a long time, and they don't require MMUs, memory sharing, stacks and time-sharing.

Closures are like GOTOs – they *can* be misused. Organizing closures in a hierarchical manner (see my other essays on this subject) is one way to tame their use. We have found that we need "languages" that encourage certain usage patterns – e.g. C vs. assembler programming (C encourages Structured control flow, through if-then-else statements, etc.) - e.g. OO encourages data encapsulation (but leaves control-flow encapsulation wide open, if not worse off).

GOTOs are assembler-level details. GOTOs must exist. GOTOs must be tamed. Likewise, Closures exist and must be tamed.

Message-passing is another GOTO-like atomic element. It must be tamed. Message-passing enables the concurrent paradigm. A component can send another component a message, but the receiving component is not constrained as to when it deals with the message. Message-passing defeats rendezvous. Message-passing allows asynchrony. The concurrent paradigm requires asynchrony.

Backus asked "Can Programming Be Liberated From The von Neumann Style?". The question is a good one. The answer that Backus proposed was insufficient, though. Mathematics, as we know it today, implies synchronous evaluation. We need to think of mathematics and synchrony being *isolated* on asynchronous *islands*.

Computers are parallel by default. Forcing synchrony onto every part of computing is contrary to the manner in which computers work. For example, multi-tasking is considered to be a "hard" problem mostly because multi-tasking is being solved using only one paradigm – a paradigm that is not the best choice for every problem in a particular solution. It is like trying to emulate C-like *format* statements in PROLOG. It can be done, but results in accidental complexity.

"Yes" to synchronous operations floating on asynchronous islands.

"No" to synchronous *everything*.

It is OK to use more than one paradigm at a time. For example, again, PROLOG leads the way towards thinking in relational terms, but casting *everything* as a relation brings unnecessary difficulty.

Paradigms are DSLs draped over full-featured assemblers.[1]

The Concurrent API

The API for the concurrent paradigm consists of one routine – Send().[2]

Examples of the Concurrent Paradigm

These are some examples of the concurrent paradigm:

- UNIX® shell pipelines
- closures using queues/mailboxes for inter-closure communication.

O/S threads, and threading libraries, are often conflated with the concurrent paradigm. O/S threads – and many threading libraries – solve much greater problems[3], and are poor examples of the concurrent paradigm.

Advantages to Using the Concurrent Paradigm

- Encourages the absence of dependencies
- Isolation.

1 Most HLL PLs started out life as DSLs built on top of full-featured substrate languages. Greenspun's 10th rule is usually taken as a joke, but taken in the light of HLLs-as-DSLs, it begins to ring true.

2 OK, OK, I am simplifying. There are some house-keeping functions that might be necessary. We definitely don't need RETURN and we don't need CATCH/THROW. Send() is sufficient.   RETURN? Use Send() instead. CATCH/THROW? Use Send() instead. LOOP? Get rid of it. Recursion? Get rid of it. [Also, we don't need dynamicism. Dynamic programming is just another way to say "self modifying code", which is even worse than using the word "GOTO"]. RETURN & CATCH/THROW & LOOP & Recursion make sense *only* on a synchronous island. They are not atomic elements of the concurrent paradigm.

3 Problems such as time-sharing, memory-sharing, protecting apps from one another, protecting against long-running apps, code-reuse, etc.. Most single apps don't need these features / problems. Note that mutual-multitasking is OK within a single app. A "bug" is just a "bug" regardless of whether the concurrent paradigm or the synchronous paradigm is used.