WIKIPEDIA
The Free Encyclopedia

Concurrent computing

Concurrent computing is a form of computing in which several computations are executed *concurrently*—during overlapping time periods—instead of *sequentially*—with one completing before the next starts.

This is a property of a system—whether a program, computer, or a network—where there is a separate execution point or "thread of control" for each process. A *concurrent system* is one where a computation can advance without waiting for all other computations to complete.[1]

Concurrent computing is a form of modular programming. In its paradigm an overall computation is factored into subcomputations that may be executed concurrently. Pioneers in the field of concurrent computing include Edsger Dijkstra, Per Brinch Hansen, and C.A.R. Hoare.[2]

Introduction

The concept of concurrent computing is frequently confused with the related but distinct concept of parallel computing,[3][4] although both can be described as "multiple processes executing *during the same period of time*". In parallel computing, execution occurs at the same physical instant: for example, on separate processors of a multi-processor machine, with the goal of speeding up computations—parallel computing is impossible on a (one-core) single processor, as only one computation can occur at any instant (during any single clock cycle).[a] By contrast, concurrent computing consists of process *lifetimes* overlapping, but execution need not happen at the same instant. The goal here is to model processes in the outside world that happen concurrently, such as multiple clients accessing a server at the same time. Structuring software systems as composed of multiple concurrent, communicating parts can be useful for tackling complexity, regardless of whether the parts can be executed in parallel.[5]:1

For example, concurrent processes can be executed on one core by interleaving the execution steps of each process via time-sharing slices: only one process runs at a time, and if it does not complete during its time slice, it is *paused*, another process begins or resumes, and then later the original process is resumed. In this way, multiple processes are part-way through execution at a single instant, but only one process is being executed at that instant.

Concurrent computations *may* be executed in parallel,[3][6] for example, by assigning each process to a separate processor or processor core, or distributing a computation across a network.

The exact timing of when tasks in a concurrent system are executed depends on the scheduling, and tasks need not always be executed concurrently. For example, given two tasks, T1 and T2:

- T1 may be executed and finished before T2 or *vice versa* (serial *and* sequential)
- T1 and T2 may be executed alternately (serial *and* concurrent)
- T1 and T2 may be executed simultaneously at the same instant of time (parallel *and* concurrent)

The word "sequential" is used as an antonym for both "concurrent" and "parallel"; when these are explicitly distinguished, *concurrent/sequential* and *parallel/serial* are used as opposing pairs.[7] A schedule in which tasks execute one at a time (serially, no parallelism), without interleaving (sequentially, no concurrency: no task begins until the prior task ends) is called a *serial schedule*. A set of tasks that can be scheduled serially is *serializable*, which simplifies concurrency control.

Coordinating access to shared resources

The main challenge in designing concurrent programs is concurrency control: ensuring the correct sequencing of the interactions or communications between different computational executions, and coordinating access to resources that are shared among executions.[6] Potential problems include race conditions, deadlocks, and resource starvation. For example, consider the following algorithm to make withdrawals from a checking account represented by the shared resource `balance`:

```
1  bool withdraw(int withdrawal)
2  {
3      if (balance >= withdrawal)
4      {
5          balance -= withdrawal;
6          return true;
7      }
8      return false;
9  }
```

Suppose `balance = 500`, and two concurrent *threads* make the calls `withdraw(300)` and `withdraw(350)`. If line 3 in both operations executes before line 5 both operations will find that `balance >= withdrawal` evaluates to `true`, and execution will proceed to subtracting the withdrawal amount. However, since both processes perform their withdrawals, the total amount withdrawn will end up being more than the original balance. These sorts of problems with shared resources benefit from the use of concurrency control, or non-blocking algorithms.

Advantages

The advantages of concurrent computing include:

- Increased program throughput—parallel execution of a concurrent program allows the number of tasks completed in a given time to increase proportionally to the number of processors according to Gustafson's law
- High responsiveness for input/output—input/output-intensive programs mostly wait for input or output operations to complete. Concurrent programming allows the time that would be spent waiting to be used for another task.
- More appropriate program structure—some problems and problem domains are well-suited to representation as concurrent tasks or processes.

Models

Introduced in 1962, Petri nets were an early attempt to codify the rules of concurrent execution. Dataflow theory later built upon these, and Dataflow architectures were created to physically implement the ideas of dataflow theory. Beginning in the late 1970s, process calculi such as Calculus of Communicating Systems (CCS) and Communicating Sequential Processes (CSP) were developed to permit algebraic reasoning about systems composed of interacting components. The π-calculus added the capability for reasoning about dynamic topologies.

Input/output automata were introduced in 1987.

Logics such as Lamport's TLA+, and mathematical models such as traces and Actor event diagrams, have also been developed to describe the behavior of concurrent systems.

Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Consistency models

Concurrent programming languages and multiprocessor programs must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

One of the first consistency models was Leslie Lamport's sequential consistency model. Sequential consistency is the property of a program that its execution produces the same results as a sequential program. Specifically, a program is sequentially consistent if "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".[8]

Implementation

A number of different methods can be used to implement concurrent programs, such as implementing each computational execution as an operating system process, or implementing the computational processes as a set of threads within a single operating system process.

Interaction and communication

In some concurrent computing systems, communication between the concurrent components is hidden from the programmer (e.g., by using futures), while in others it must be handled explicitly. Explicit communication can be divided into two classes:

Shared memory communication

Concurrent components communicate by altering the contents of shared memory locations (exemplified by Java and C#). This style of concurrent programming usually needs the use of some form of locking (e.g., mutexes, semaphores, or monitors) to coordinate between threads. A program that properly implements any of these is said to be thread-safe.

Message passing communication

Concurrent components communicate by exchanging messages (exemplified by MPI, Go, Scala, Erlang and occam). The exchange of messages may be carried out asynchronously, or may use a synchronous "rendezvous" style in which the sender blocks until the message is received. Asynchronous message passing may be reliable or unreliable (sometimes referred to as "send and pray"). Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming. A wide variety of mathematical theories to understand and analyze message-passing systems are available, including the actor model, and various process calculi. Message passing can be efficiently implemented via symmetric multiprocessing, with or without shared memory cache coherence.

Shared memory and message passing concurrency have different performance characteristics. Typically (although not always), the per-process memory overhead and task switching overhead is lower in a message passing system, but the overhead of message passing is greater than for a procedure call. These differences are often overwhelmed by other performance factors.

History

Concurrent computing developed out of earlier work on railroads and telegraphy, from the 19th and early 20th century, and some terms date to this period, such as semaphores. These arose to address the question of how to handle multiple trains on the same railroad system (avoiding collisions and maximizing efficiency) and how to handle multiple transmissions over a given set of wires (improving efficiency), such as via time-division multiplexing (1870s).

The academic study of concurrent algorithms started in the 1960s, with Dijkstra (1965) credited with being the first paper in this field, identifying and solving mutual exclusion.[9]

Prevalence

Concurrency is pervasive in computing, occurring from low-level hardware on a single chip to worldwide networks. Examples follow.

At the programming language level:

- Channel
- Coroutine
- Futures and promises

At the operating system level:

- Computer multitasking, including both cooperative multitasking and preemptive multitasking
  - Time-sharing, which replaced sequential batch processing of jobs with concurrent use of a system
- Process
- Thread

At the network level, networked systems are generally concurrent by their nature, as they consist of separate devices.

Languages supporting concurrent programming

Concurrent programming languages are programming languages that use language constructs for concurrency. These constructs may involve multi-threading, support for distributed computing, message passing, shared resources (including shared memory) or futures and promises. Such languages are sometimes described as *concurrency-oriented languages* or *concurrency-oriented programming languages* (COPL).[10]

Today, the most commonly used programming languages that have specific constructs for concurrency are Java and C#. Both of these languages fundamentally use a shared-memory concurrency model, with locking provided by monitors (although message-passing models can and have been implemented on top of the underlying shared-memory model). Of the languages that use a message-passing concurrency model, Erlang is probably the most widely used in industry at present.

Many concurrent programming languages have been developed more as research languages (e.g. Pict) rather than as languages for production use. However, languages such as Erlang, Limbo, and occam have seen industrial use at various times in the last 20 years. A non-exhaustive list of languages which use or provide concurrent programming facilities:

- Ada—general purpose, with native support for message passing and monitor based concurrency
- Alef—concurrent, with threads and message passing, for system programming in early versions of Plan 9 from Bell Labs
- Alice—extension to Standard ML, adds support for concurrency via futures
- Ateji PX—extension to Java with parallel primitives inspired from π-calculus
- Axum—domain specific, concurrent, based on actor model and .NET Common Language Runtime using a C-like syntax
- BMDFM—Binary Modular DataFlow Machine
- C++—std::thread
- Cω (C omega)—for research, extends C#, uses asynchronous communication
- C#—supports concurrent computing using lock, yield, also since version 5.0 async and await keywords introduced
- Clojure—modern, functional dialect of Lisp on the Java platform
- Concurrent Clean—functional programming, similar to Haskell
- Concurrent Collections (CnC)—Achieves implicit parallelism independent of memory model by explicitly defining flow of data and control
- Concurrent Haskell—lazy, pure functional language operating concurrent processes on shared memory
- Concurrent ML—concurrent extension of Standard ML
- Concurrent Pascal—by Per Brinch Hansen
- Curry
- D—multi-paradigm system programming language with explicit support for concurrent programming (actor model)
- E—uses promises to preclude deadlocks
- ECMAScript—uses promises for asynchronous operations
- Eiffel—through its SCOOP mechanism based on the concepts of Design by Contract
- Elixir—dynamic and functional meta-programming aware language running on the Erlang VM.
- Erlang—uses synchronous or asynchronous message passing with no shared memory
- FAUST—real-time functional, for signal processing, compiler provides automatic parallelization via OpenMP or a specific work-stealing scheduler
- Fortran—coarrays and *do concurrent* are part of Fortran 2008 standard
- Go—for system programming, with a concurrent programming model based on CSP
- Haskell—concurrent, and parallel functional programming language[11]

- Hume—functional, concurrent, for bounded space and time environments where automata processes are described by synchronous channels patterns and message passing
- Io—actor-based concurrency
- Janus—features distinct *askers* and *tellers* to logical variables, bag channels; is purely declarative
- Java—thread class or Runnable interface
- Julia—"concurrent programming primitives: Tasks, async-wait, Channels."[12]
- JavaScript—via web workers, in a browser environment, promises, and callbacks.
- JoCaml—concurrent and distributed channel based, extension of OCaml, implements the join-calculus of processes
- Join Java—concurrent, based on Java language
- Joule—dataflow-based, communicates by message passing
- Joyce—concurrent, teaching, built on Concurrent Pascal with features from CSP by Per Brinch Hansen
- LabVIEW—graphical, dataflow, functions are nodes in a graph, data is wires between the nodes; includes object-oriented language
- Limbo—relative of Alef, for system programming in Inferno (operating system)
- Locomotive BASIC—Amstrad variant of BASIC contains EVERY and AFTER commands for concurrent subroutines
- MultiLisp—Scheme variant extended to support parallelism
- Modula-2—for system programming, by N. Wirth as a successor to Pascal with native support for coroutines
- Modula-3—modern member of Algol family with extensive support for threads, mutexes, condition variables
- Newsqueak—for research, with channels as first-class values; predecessor of Alef
- occam—influenced heavily by communicating sequential processes (CSP)
  - occam-π—a modern variant of occam, which incorporates ideas from Milner's π-calculus
- Orc—heavily concurrent, nondeterministic, based on Kleene algebra
- Oz-Mozart—multiparadigm, supports shared-state and message-passing concurrency, and futures
- ParaSail—object-oriented, parallel, free of pointers, race conditions
- Pict—essentially an executable implementation of Milner's π-calculus
- Raku includes classes for threads, promises and channels by default[13]
- Python — uses thread-based parallelism and process-based parallelism [14]
- Reia—uses asynchronous message passing between shared-nothing objects
- Red/System—for system programming, based on Rebol
- Rust—for system programming, using message-passing with move semantics, shared immutable memory, and shared mutable memory.[15]
- Scala—general purpose, designed to express common programming patterns in a concise, elegant, and type-safe way
- SequenceL—general purpose functional, main design objectives are ease of programming, code clarity-readability, and automatic parallelization for performance on multicore hardware, and provably free of race conditions
- SR—for research
- SuperPascal—concurrent, for teaching, built on Concurrent Pascal and Joyce by Per Brinch Hansen
- Swift—built-in support for writing asynchronous and parallel code in a structured way[16]

- Unicon—for research
- TNSDL—for developing telecommunication exchanges, uses asynchronous message passing
- VHSIC Hardware Description Language (VHDL)—IEEE STD-1076
- XC—concurrency-extended subset of C language developed by XMOS, based on communicating sequential processes, built-in constructs for programmable I/O

Many other languages provide support for concurrency in the form of libraries, at levels roughly comparable with the above list.

See also

- Asynchronous I/O
- Chu space
- Flow-based programming
- Java ConcurrentMap
- Ptolemy Project
- Race condition § Computing
- Structured concurrency
- Transaction processing

Notes

a. This is discounting parallelism internal to a processor core, such as pipelining or vectorized instructions. A one-core, one-processor *machine* may be capable of some parallelism, such as with a coprocessor, but the processor alone is not.

References

1. *Operating System Concepts* 9th edition, Abraham Silberschatz. "Chapter 4: Threads"
2. Hansen, Per Brinch, ed. (2002). *The Origin of Concurrent Programming* (https://link.springer.com/book/10.1007/978-1-4757-3472-0). doi:10.1007/978-1-4757-3472-0 (https://doi.org/10.1007%2F978-1-4757-3472-0). ISBN 978-1-4419-2986-0. S2CID 44909506 (https://api.semanticscholar.org/CorpusID:44909506).
3. Pike, Rob (2012-01-11). "Concurrency is not Parallelism". *Waza conference*, 11 January 2012. Retrieved from http://talks.golang.org/2012/waza.slide (slides) and http://vimeo.com/49718712 (video).
4. "Parallelism vs. Concurrency" (https://wiki.haskell.org/Parallelism_vs._Concurrency). *Haskell Wiki*.
5. Schneider, Fred B. (1997-05-06). *On Concurrent Programming* (https://archive.org/details/onconcurrentprog0000schn). Springer. ISBN 9780387949420.
6. Ben-Ari, Mordechai (2006). *Principles of Concurrent and Distributed Programming* (2nd ed.). Addison-Wesley. ISBN 978-0-321-31283-9.
7. Patterson & Hennessy 2013, p. 503.
8. Lamport, Leslie (1 September 1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". *IEEE Transactions on Computers*. **C-28** (9): 690–691. doi:10.1109/TC.1979.1675439 (https://doi.org/10.1109%2FTC.1979.1675439). S2CID 5679366 (https://api.semanticscholar.org/CorpusID:5679366).

9. "PODC Influential Paper Award: 2002" (http://www.podc.org/influential/2002.html), *ACM Symposium on Principles of Distributed Computing*, retrieved 2009-08-24
10. Armstrong, Joe (2003). "Making reliable distributed systems in the presence of software errors" (http://www.diva-portal.org/smash/get/diva2:9492/FULLTEXT01.pdf) (PDF).
11. Marlow, Simon (2013) Parallel and Concurrent Programming in Haskell : Techniques for Multicore and Multithreaded Programming ISBN 9781449335946
12. https://juliacon.talkfunnel.com/2015/21-concurrent-and-parallel-programming-in-julia Concurrent and Parallel programming in Julia
13. "Concurrency" (https://docs.perl6.org/language/concurrency). *docs.perl6.org*. Retrieved 2017-12-24.
14. Documentation » The Python Standard Library » Concurrent Execution (https://docs.python.org/3/library/concurrency.html)
15. Blum, Ben (2012). "Typesafe Shared Mutable State" (http://winninggraceconditions.blogspot.com/2012/09/rust-4-typesafe-shared-mutable-state.html). Retrieved 2012-11-14.
16. "Concurrency" (https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html). 2022. Retrieved 2022-12-15.

Sources

- Patterson, David A.; Hennessy, John L. (2013). *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design (5 ed.). Morgan Kaufmann. ISBN 978-0-12407886-4.

Further reading

- Dijkstra, E. W. (1965). "Solution of a problem in concurrent programming control" (https://doi.org/10.1145%2F365559.365617). *Communications of the ACM*. **8** (9): 569. doi:10.1145/365559.365617 (https://doi.org/10.1145%2F365559.365617). S2CID 19357737 (https://api.semanticscholar.org/CorpusID:19357737).
- Herlihy, Maurice (2008) [2008]. *The Art of Multiprocessor Programming*. Morgan Kaufmann. ISBN 978-0123705914.
- Downey, Allen B. (2005) [2005]. *The Little Book of Semaphores* (https://web.archive.org/web/20160304031330/http://www.greenteapress.com/semaphores/downey08semaphores.pdf) (PDF). Green Tea Press. ISBN 978-1-4414-1868-5. Archived from the original (http://www.greenteapress.com/semaphores/downey08semaphores.pdf) (PDF) on 2016-03-04. Retrieved 2009-11-21.
- Filman, Robert E.; Daniel P. Friedman (1984). *Coordinated Computing: Tools and Techniques for Distributed Software* (https://archive.org/details/coordinatedcompu0000film/page/370). New York: McGraw-Hill. p. 370 (https://archive.org/details/coordinatedcompu0000film/page/370). ISBN 978-0-07-022439-1.
- Leppäjärvi, Jouni (2008). *A pragmatic, historically oriented survey on the universality of synchronization primitives* (http://www.enseignement.polytechnique.fr/informatique/INF431/X09-2010-2011/AmphiTHC/SynchronizationPrimitives.pdf) (PDF). University of Oulu.
- Taubenfeld, Gadi (2006). *Synchronization Algorithms and Concurrent Programming* (http://www.faculty.idc.ac.il/gadi/book.htm). Pearson / Prentice Hall. p. 433. ISBN 978-0-13-197259-9.

External links

- Media related to Concurrent programming at Wikimedia Commons

- Concurrent Systems Virtual Library (https://web.archive.org/web/20060128114620/http://vl.fmnet.info/concurrent/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Concurrent_computing&oldid=1202038776"

-