# Contents

## Articles



## References

# Article Licenses

# Parallel computing

IBM's Blue Gene/P massively parallel supercomputer

**Parallel computing** is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling.[1] As power consumption (and consequently heat generation) by computers has become a concern in recent years,[2] parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.[3]

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

The maximum possible speed-up of a single program as a result of parallelization is known as Amdahl's law.

## Background

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next is executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

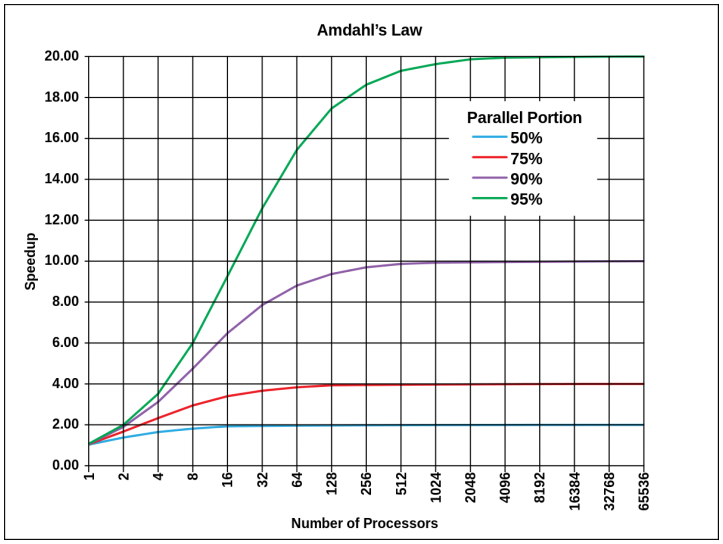
Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all compute-bound programs.

However, power consumption by a chip is given by the equation $P = C \times V^2 \times F$, where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and F is the processor frequency (cycles per second). Increases in frequency increase the amount of power used in a processor. Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

Moore's Law is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. Despite power consumption issues, and repeated predictions of its end, Moore's law is still in effect. With the end of frequency scaling, these additional transistors (which are no longer used for frequency scaling) can be used to add extra hardware for parallel computing.

## Amdahl's law and Gustafson's law

Optimally, the speed-up from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.




A graphical representation of Amdahl's law. The speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used.

The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If $\alpha$ is the fraction of running time a program spends on non-parallelizable parts, then:

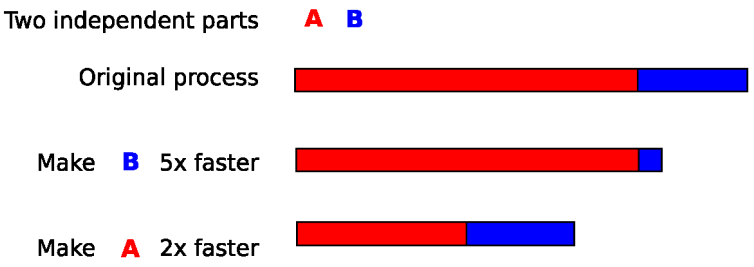
$$\lim_{P \to \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}$$

is the maximum speed-up with parallelization of the program. If the sequential portion of a program accounts for 10% of the runtime ( $\alpha = 0.1$), we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."

Gustafson's law is another law in computing, closely related to Amdahl's law. It states that the speedup with $P$ processors is

$$S(P) = P - \alpha(P-1) \qquad = \alpha + P(1-\alpha).$$

Both Amdahl's law and Gustafson's law assume that the running time of the sequential portion of the program is independent of the number of processors. Amdahl's law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also *independent of the number of processors*, whereas Gustafson's law assumes that the total amount of work to be done in parallel *varies linearly with the number of processors*.




Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. With effort, a programmer may be able to make this part five times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A twice as fast. This will make the computation much faster than by optimizing part B, even though B got a greater speed-up (5× versus 2×).

## Dependencies

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Let $P_i$ and $P_j$ be two program segments. Bernstein's conditions describe when the two are independent and can be executed in parallel. For $P_i$, let $I_i$ be all of the input variables and $O_i$ the output variables, and likewise for $P_j$. $P_i$ and $P_j$ are independent if they satisfy

- $I_j \cap O_i = \varnothing,$
- $I_i \cap O_j = \varnothing,$
- $O_i \cap O_j = \varnothing.$

Violation of the first condition introduces a flow dependency, corresponding to the first segment producing a result used by the second segment. The second condition represents an anti-dependency, when the second segment ($P_j$) produces a variable needed by the first segment ($P_i$). The third and final condition represents an output dependency: When two segments write to the same location, the result comes from the logically last executed segment.

Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep(a, b)
2: c := a·b
3: d := 3·c
4: end function
```

Operation 3 in Dep(a, b) cannot be executed before (or even in parallel with) operation 2, because operation 3 uses a result from operation 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2: c := a·b
3: d := 3·b
4: e := a+b
5: end function
```

In this example, there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.

## Race conditions, mutual exclusion, synchronization, and parallel slowdown

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, "threads" is generally accepted as a generic term for subtasks. Threads will often need to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. For example, consider the following program:

| Thread A | Thread B |
|---|---|
| 1A: Read variable V | 1B: Read variable V |
| 2A: Add 1 to variable V | 2B: Add 1 to variable V |
| 3A: Write back to variable V | 3B: Write back to variable V |

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished. Therefore, to guarantee correct program execution, the above program can be rewritten to use locks:

| Thread A | Thread B |
|---|---|
| 1A: Lock variable V | 1B: Lock variable V |
| 2A: Read variable V | 2B: Read variable V |
| 3A: Add 1 to variable V | 3B: Add 1 to variable V |
| 4A: Write back to variable V | 4B: Write back to variable V |
| 5A: Unlock variable V | 5B: Unlock variable V |

One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow a program.

Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock locks multiple variables all at once. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is

known as parallel slowdown.

## Fine-grained, coarse-grained, and embarrassing parallelism

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

## Consistency models

Main article: Consistency model

Parallel programming languages and parallel computers 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 parallel program that its parallel 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".

Software transactional memory is a common type of consistency model. Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Mathematically, these models can be represented in several ways. Petri nets, which were introduced in Carl Adam Petri's 1962 doctoral thesis, were an early attempt to codify the rules of consistency models. 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 and Communicating Sequential Processes were developed to permit algebraic reasoning about systems composed of interacting components. More recent additions to the process calculus family, such as the $\pi$-calculus, have added the capability for reasoning about dynamic topologies. 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.

## Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, and whether or not those instructions were using a single set or multiple sets of data.

### Flynn's taxonomy

| | Single instruction | Multiple instruction |
|---|---|---|
| **Single data** | SISD | MISD |
| **Multiple data** | SIMD | MIMD |

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

According to David A. Patterson and John L. Hennessy, "Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme."[4]

## Types of parallelism

### Bit-level parallelism

Main article: Bit-level parallelism

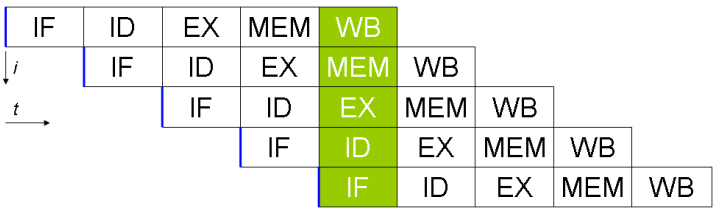
From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.

Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until recently (c. 2003–2004), with the advent of x86-64 architectures, have 64-bit processors become commonplace.

### Instruction-level parallelism

Main article: Instruction level parallelism

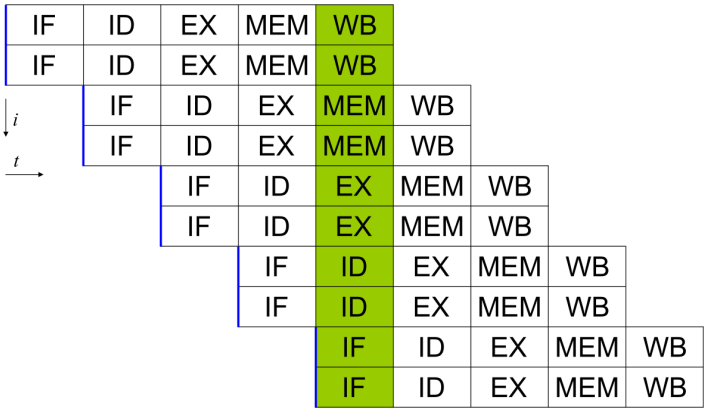
A computer program, is in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.[5]




A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch, decode, execute, memory access, and write back. The Pentium 4 processor had a 35-stage pipeline.[6]

In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm (which is similar to scoreboarding but makes use of register renaming) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism.




A five-stage pipelined superscalar processor, capable of issuing two instructions per cycle. It can have two instructions in each stage of the pipeline, for a total of up to 10 instructions (shown in green) being simultaneously executed.

### Task parallelism

Main article: Task parallelism

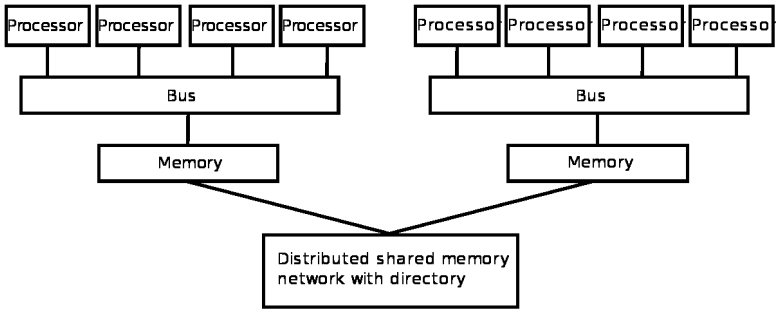
Task parallelism is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data".[7] This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism does not usually scale with the size of a problem.[8]

## Hardware

### Memory and communication

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space).[9] Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory.

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a Non-Uniform




A logical view of a Non-Uniform Memory Access (NUMA) architecture. Processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory.

Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access.

Computer systems make use of caches—small, fast memories located close to the processor which store temporary copies of memory values (nearby in both the physical and logical sense). Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Bus snooping is one of the most common methods for keeping track of which values are being accessed (and thus should be purged). Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared-memory computer architectures do not scale as well as distributed memory systems do.

Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a hypercube with more than one processor at a node), or n-dimensional mesh.

Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.

## Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

### Multicore computing

Main article: Multi-core (computing)

A multicore processor is a processor that includes multiple execution units ("cores") on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); in contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.

Each core in a multicore processor can potentially be superscalar as well—that is, on every cycle, each core can issue multiple instructions from one instruction stream. Simultaneous multithreading (of which Intel's HyperThreading is the best known) was an early form of pseudo-multicoreism. A processor capable of simultaneous multithreading has only one execution unit ("core"), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread.

### Symmetric multiprocessing

Main article: Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus.[10] Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors.[11] "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."

### Distributed computing

Main article: Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

### Cluster computing

Main article: Computer cluster

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer.[12] Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network.[13] Beowulf technology was originally developed by Thomas Sterling and Donald Becker. The vast majority of the TOP500 supercomputers are clusters.[14]

A Beowulf cluster

### Massive parallel processing

Main article: Massively parallel (computing)

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having "far more" than 100 processors.[15] In a MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect."[16]

Blue Gene/L, the fifth fastest supercomputer in the world according to the June 2009 TOP500 ranking, is a MPP.

A cabinet from Blue Gene/L, ranked as the fourth fastest supercomputer in the world according to the 11/2008 TOP500 rankings. Blue Gene/L is a massively parallel processor.

### Grid computing

Main article: Distributed computing

Distributed computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems. Many distributed computing applications have been created, of which SETI@home and Folding@home are the best-known examples.

Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common distributed computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, distributed computing software makes use of "spare cycles", performing computations at times when a computer is idling.

### Specialized parallel computers

Within parallel computing, there are specialized parallel devices that remain niche areas of interest. While not domain-specific, they tend to be applicable to only a few classes of parallel problems.

### Reconfigurable computing with field-programmable gate arrays

Reconfigurable computing is the use of a field-programmable gate array (FPGA) as a co-processor to a general-purpose computer. An FPGA is, in essence, a computer chip that can rewire itself for a given task.

FPGAs can be programmed with hardware description languages such as VHDL or Verilog. However, programming in these languages can be tedious. Several vendors have created C to HDL languages that attempt to emulate the syntax and semantics of the C programming language, with which most programmers are familiar. The best known C to HDL languages are Mitrion-C, Impulse C, DIME-C, and Handel-C. Specific subsets of SystemC based on C++ can also be used for this purpose.

AMD's decision to open its HyperTransport technology to third-party vendors has become the enabling technology for high-performance reconfigurable computing.[17] According to Michael R. D'Amour, Chief Operating Officer of DRC Computer Corporation, "when we first walked into AMD, they called us 'the socket stealers.' Now they call us their partners."

### General-purpose computing on graphics processing units (GPGPU)

Main article: GPGPU

Nvidia's Tesla GPGPU card

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing.[18] Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.

In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively. Other GPU programming languages include BrookGPU, PeakStream, and RapidMind. Nvidia has also released specific products for computation in their Tesla series. The technology consortium Khronos Group has released the OpenCL specification, which is a framework for writing programs that execute across platforms consisting of CPUs and GPUs. AMD, Apple, Intel, Nvidia and others are supporting OpenCL.

### Application-specific integrated circuits

Main article: Application-specific integrated circuit

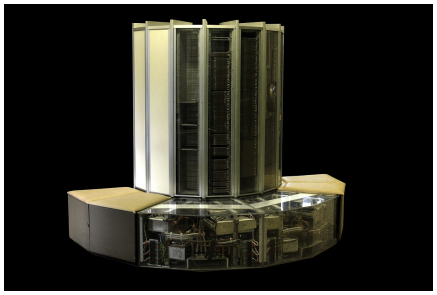
Several application-specific integrated circuit (ASIC) approaches have been devised for dealing with parallel applications.[19]

Because an ASIC is (by definition) specific to a given application, it can be fully optimized for that application. As a result, for a given application, an ASIC tends to outperform a general-purpose computer. However, ASICs are created by X-ray lithography. This process requires a mask, which can be extremely expensive. A single mask can cost over a million US dollars.[20] (The smaller the transistors required for the chip, the more expensive the mask will be.) Meanwhile, performance increases in general-purpose computing over time (as described by Moore's Law) tend to wipe out these gains in only one or two chip generations. High initial cost, and the tendency to be overtaken by Moore's-law-driven general-purpose computing, has rendered ASICs unfeasible for most parallel computing applications. However, some have been built. One example is the peta-flop RIKEN MDGRAPE-3 machine which uses custom ASICs for molecular dynamics simulation.

### Vector processors

Main article: Vector processor

A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. "Vector processors have high-level operations that work on linear arrays of numbers or vectors. An example vector operation is $A = B \times C$, where $A$, $B$, and $C$ are each 64-element vectors of 64-bit floating-point numbers."[21] They are closely related to Flynn's SIMD classification.



The Cray-1 is the most famous vector processor.

Cray computers became famous for their vector-processing computers in the 1970s and 1980s. However, vector processors—both as CPUs and as full computer systems—have generally disappeared. Modern processor instruction sets do include some vector processing instructions, such as with AltiVec and Streaming SIMD Extensions (SSE).

## Software

### Parallel programming languages

Main article: List of concurrent and parallel programming languages

Concurrent programming languages, libraries, APIs, and parallel programming models (such as Algorithmic Skeletons) have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture—shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads and OpenMP are two of most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API.[22] One concept used in programming parallel programs is the future concept, where one part of a program promises to deliver a required datum to another part of a program at some future time.

CAPS entreprise and Pathscale are also coordinating their effort to make HMPP (Hybrid Multicore Parallel Programming) directives an Open Standard called OpenHMPP. The OpenHMPP directive-based programming model offers a syntax to efficiently offload computations on hardware accelerators and to optimize data movement to/from the hardware memory. OpenHMPP directives describe remote procedure call (RPC) on an accelerator device (e.g. GPU) or more generally a set of cores. The directives annotate C or Fortran codes to describe two sets of functionalities: the offloading of procedures (denoted codelets) onto a remote device and the optimization of data transfers between the CPU main memory and the accelerator memory.

### Automatic parallelization

Main article: Automatic parallelization

Automatic parallelization of a sequential program by a compiler is the holy grail of parallel computing. Despite decades of work by compiler researchers, automatic parallelization has had only limited success.

Mainstream parallel programming languages remain either explicitly parallel or (at best) partially implicit, in which a programmer gives the compiler directives for parallelization. A few fully implicit parallel programming languages exist—SISAL, Parallel Haskell, System C (for FPGAs), Mitrion-C, VHDL, and Verilog.

### Application checkpointing

Main article: Application checkpointing

As a computer system grows in complexity, the mean time between failures usually decreases. Application checkpointing is a technique whereby the computer system takes a "snapshot" of the application — a record of all current resource allocations and variable states, akin to a core dump; this information can be used to restore the program if the computer should fail. Application checkpointing means that the program has to restart from only its last checkpoint rather than the beginning. While checkpointing provides benefits in a variety of situations, it is especially useful in highly parallel systems with a large number of processors used in high performance computing.[23]

## Algorithmic methods

As parallel computers become larger and faster, it becomes feasible to solve problems that previously took too long to run. Parallel computing is used in a wide range of fields, from bioinformatics (protein folding and sequence analysis) to economics (mathematical finance). Common types of problems found in parallel computing applications are:[24]

- Dense linear algebra
- Sparse linear algebra
- Spectral methods (such as Cooley–Tukey fast Fourier transform)
- *n*-body problems (such as Barnes–Hut simulation)
- Structured grid problems (such as Lattice Boltzmann methods)
- Unstructured grid problems (such as found in finite element analysis)
- Monte Carlo simulation
- Combinational logic (such as brute-force cryptographic techniques)
- Graph traversal (such as sorting algorithms)
- Dynamic programming
- Branch and bound methods
- Graphical models (such as detecting hidden Markov models and constructing Bayesian networks)
- Finite-state machine simulation

## Fault-tolerance

Further information: Fault-tolerant computer system

Parallel computing can also be applied to the design of fault-tolerant computer systems, particularly via lockstep systems performing the same operation in parallel. This provides redundancy in case one component should fail, and also allows automatic error detection and error correction if the results differ.

## History

Main article: History of computing

The origins of true (MIMD) parallelism go back to Federico Luigi, Conte Menabrea and his "Sketch of the Analytic Engine Invented by Charles Babbage".[25][26] IBM introduced the 704 in 1954, through a project in which Gene Amdahl was one of the principal architects. It became the first commercially available computer to use fully automatic floating point arithmetic commands.

ILLIAC IV, "perhaps the most infamous of Supercomputers"

In April 1958, S. Gill (Ferranti) discussed parallel programming and the need for branching and waiting.[27] Also in 1958, IBM researchers John Cocke and Daniel Slotnick discussed the use of parallelism in numerical calculations for the first time. Burroughs Corporation introduced the D825 in 1962, a four-processor computer that accessed up to 16 memory modules through a crossbar switch. In 1967, Amdahl and Slotnick published a debate about the feasibility of parallel processing at American Federation of Information Processing Societies Conference. It was during this debate that Amdahl's Law was coined to define the limit of speed-up due to parallelism.

In 1969, US company Honeywell introduced its first Multics system, a symmetric multiprocessor system capable of running up to eight processors in parallel. C.mmp, a 1970s multi-processor project at Carnegie Mellon University, was "among the first multiprocessors with more than a few processors". "The first bus-connected multi-processor with snooping caches was the Synapse N+1 in 1984."

SIMD parallel computers can be traced back to the 1970s. The motivation behind early SIMD computers was to amortize the gate delay of the processor's control unit over multiple instructions.[28] In 1964, Slotnick had proposed building a massively parallel computer for the Lawrence Livermore National Laboratory. His design was funded by the US Air Force, which was the earliest SIMD parallel-computing effort, ILLIAC IV. The key to its design was a fairly high parallelism, with up to 256 processors, which allowed the machine to work on large datasets in what would later be known as vector processing. However, ILLIAC IV was called "the most infamous of Supercomputers", because the project was only one fourth completed, but took 11 years and cost almost four times the original estimate.[] When it was finally ready to run its first real application in 1976, it was outperformed by existing commercial supercomputers such as the Cray-1.

## References


[1] S.V. Adve et al. (November 2008). "Parallel Computing Research at Illinois: The UPCRC Agenda" (http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf) (PDF). Parallel@Illinois, University of Illinois at Urbana-Champaign. "The main techniques for these performance benefits – increased clock frequency and smarter but increasingly complex architectures – are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster."

**[2]** Asanovic et al. Old [conventional wisdom]: Power is free, but transistors are expensive. New [conventional wisdom] is [that] power is expensive, but transistors are "free".

[3] Asanovic, Krste et al. (December 18, 2006). "The Landscape of Parallel Computing Research: A View from Berkeley" (http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf) (PDF). University of California, Berkeley. Technical Report No. UCB/EECS-2006-183. "Old [conventional wisdom]: Increasing clock frequency is the primary method of improving processor performance. New [conventional wisdom]: Increasing parallelism is the primary method of improving processor performance ... Even representatives from Intel, a company generally associated with the 'higher clock-speed is better' position, warned that traditional approaches to maximizing performance through maximizing clock speed have been pushed to their limit."

**[4]** Patterson and Hennessy, p. 748.

**[5]** Culler et al. p. 15.

[6] Patt, Yale (April 2004). " The Microprocessor Ten Years From Now: What Are The Challenges, How Do We Meet Them? (http://users.ece.utexas.edu/~patt/Videos/talk_videos/cmu_04-29-04.wmv) (wmv). Distinguished Lecturer talk at Carnegie Mellon University. Retrieved on November 7, 2007.

**[7]** Culler et al. p. 124.

**[8]** Culler et al. p. 125.

[9] Patterson and Hennessy, p. 713.
[10] Hennessy and Patterson, p. 549.
[11] Patterson and Hennessy, p. 714.
[12] What is clustering? (http://www.webopedia.com/TERM/c/clustering.html) Webopedia computer dictionary. Retrieved on November 7, 2007.
[13] Beowulf definition. (http://www.pcmag.com/encyclopedia_term/0,2542,t=Beowulf&i=38548,00.asp) *PC Magazine*. Retrieved on November 7, 2007.
[14] Architecture share for 06/2007 (http://www.top500.org/stats/list/29/archtype). TOP500 Supercomputing Sites. Clusters make up 74.60% of the machines on the list. Retrieved on November 7, 2007.
[15] Hennessy and Patterson, p. 537.
[16] MPP Definition. (http://www.pcmag.com/encyclopedia_term/0,,t=mpp&i=47310,00.asp) *PC Magazine*. Retrieved on November 7, 2007.
[17] D'Amour, Michael R., Chief Operating Officer, DRC Computer Corporation. "Standard Reconfigurable Computing". Invited speaker at the University of Delaware, February 28, 2007.
[18] Boggan, Sha'Kia and Daniel M. Pressel (August 2007). GPUs: An Emerging Platform for General-Purpose Computation (http://www.arl.army.mil/arlreports/2007/ARL-SR-154.pdf) (PDF). ARL-SR-154, U.S. Army Research Lab. Retrieved on November 7, 2007.
[19] Maslennikov, Oleg (2002). "Systematic Generation of Executing Programs for Processor Elements in Parallel ASIC or FPGA-Based Systems and Their Transformation into VHDL-Descriptions of Processor Element Control Units". (http://www.springerlink.com/content/jjrdrb0lelyeu3e9/) *Lecture Notes in Computer Science*, **2328/2002:** p. 272.
[20] Kahng, Andrew B. (June 21, 2004) " Scoping the Problem of DFM in the Semiconductor Industry (http://www.future-fab.com/documents.asp?grID=353&d_ID=2596)." University of California, San Diego. "Future design for manufacturing (DFM) technology must reduce design [non-recoverable expenditure] cost and directly address manufacturing [non-recoverable expenditures] – the cost of a mask set and probe card – which is well over $1 million at the 90 nm technology node and creates a significant damper on semiconductor-based innovation."
[21] Patterson and Hennessy, p. 751.
[22] The Sidney Fernbach Award given to MPI inventor Bill Gropp (http://awards.computer.org/ana/award/viewPastRecipients.action?id=16) refers to MPI as "the dominant HPC communications interface"
[23] *Encyclopedia of Parallel Computing, Volume 4* by David Padua 2011 ISBN 0387097651 page 265
[24] Asanovic, Krste, et al. (December 18, 2006). The Landscape of Parallel Computing Research: A View from Berkeley (http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf) (PDF). University of California, Berkeley. Technical Report No. UCB/EECS-2006-183. See table on pages 17–19.
[25] Menabrea, L. F. (1842). Sketch of the Analytic Engine Invented by Charles Babbage (http://www.fourmilab.ch/babbage/sketch.html). Bibliothèque Universelle de Genève. Retrieved on November 7, 2007.
[26] Patterson and Hennessy, p. 753.
[27] Parallel Programming, S. Gill, The Computer Journal Vol. 1 #1, pp2-10, British Computer Society, April 1958.
[28] Patterson and Hennessy, p. 749.

## Further reading

- Rodriguez, C.; Villagra, M.; Baran, B. (29 August 2008). "Asynchronous team algorithms for Boolean Satisfiability" (http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610083). *Bio-Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd*: 66–69. doi: 10.1109/BIMNICS.2007.4610083 (http://dx.doi.org/10.1109/BIMNICS.2007.4610083).

## External links

- Go Parallel: Translating Multicore Power into Application Performance (http://goparallel.sourceforge.net/)
- Parallel computing (http://www.dmoz.org/Computers/Parallel_Computing/) at DMOZ
- Lawrence Livermore National Laboratory: Introduction to Parallel Computing (http://www.llnl.gov/computing/tutorials/parallel_comp/)
- Comparing programmability of Open MP and pthreads (http://www.futurechips.org/tips-for-power-coders/open-mp-pthreads.html)
- What makes parallel programming hard? (http://www.futurechips.org/tips-for-power-coders/parallel-programming.html)
- Designing and Building Parallel Programs, by Ian Foster (http://www-unix.mcs.anl.gov/dbpp/)

- Internet Parallel Computing Archive (http://wotug.ukc.ac.uk/parallel/)
- Parallel processing topic area at IEEE Distributed Computing Online (http://dsonline.computer.org/portal/site/dsonline/index.jsp?pageID=dso_level1_home&path=dsonline/topics/parallel&file=index.xml&xsl=generic.xsl)
- Parallel Computing Works Free On-line Book (http://www.new-npac.org/projects/cdroms/cewes-1998-05/copywrite/pcw/book.html)
- Frontiers of Supercomputing Free On-line Book Covering topics like algorithms and industrial applications (http://ark.cdlib.org/ark:/13030/ft0f59n73z/)
- Universal Parallel Computing Research Center (http://www.upcrc.illinois.edu/)
- Course in Parallel Programming at Columbia University (in collaboration with IBM T.J Watson X10 project) (http://ppppcourse.ning.com/)
- Parallel and distributed Gr obner bases computation in JAS (http://arxiv.org/PS_cache/arxiv/pdf/1008/1008.0011v1.pdf)
- Course in Parallel Computing at University of Wisconsin-Madison (http://sbel.wisc.edu/Courses/ME964/2011/index.htm)
- OpenHMPP, A New Standard for Manycore (http://www.openhmpp.org)

# Instruction-level parallelism

**Instruction-level parallelism** (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. The potential overlap among instructions is called instruction level parallelism.

There are two approaches to instruction level parallelism:

- Hardware
- Software

Hardware level works upon dynamic parallelism whereas, the software level works on static parallelism. The Pentium processor works on the dynamic sequence of parallel execution but the Itanium processor works on the static level parallelism.

Consider the following program:

1. e = a + b
2. f = c + d
3. m = e * f

Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed. However, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously. If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time, giving an ILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphics and scientific computing the amount can be very large. However, workloads such as cryptography may exhibit much less parallelism.

Micro-architectural techniques that are used to exploit ILP include:

- Instruction pipelining where the execution of multiple instructions can be partially overlapped.

- Superscalar execution, VLIW, and the closely related explicitly parallel instruction computing concepts, in which multiple execution units are used to execute multiple instructions in parallel.
- Out-of-order execution where instructions execute in any order that does not violate data dependencies. Note that this technique is independent of both pipelining and superscalar. Current implementations of out-of-order execution dynamically (i.e., while the program is executing and without any help from the compiler) extract ILP from ordinary programs. An alternative is to extract this parallelism at compile time and somehow convey this information to the hardware. Due to the complexity of scaling the out-of-order execution technique, the industry has re-examined instruction sets which explicitly encode multiple independent operations per instruction.
- Register renaming which refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations, used to enable out-of-order execution.
- Speculative execution which allow the execution of complete instructions or parts of instructions before being certain whether this execution should take place. A commonly used form of speculative execution is control flow speculation where instructions past a control flow instruction (e.g., a branch) are executed before the target of the control flow instruction is determined. Several other forms of speculative execution have been proposed and are in use including speculative execution driven by value prediction, memory dependence prediction and cache latency prediction.
- Branch prediction which is used to avoid stalling for control dependencies to be resolved. Branch prediction is used with speculative execution.

Dataflow architectures are another class of architectures where ILP is explicitly specified, for a recent example see the TRIPS architecture.

In recent years, ILP techniques have been used to provide performance improvements in spite of the growing disparity between processor operating frequencies and memory access times (early ILP designs such as the IBM System/360 Model 91 used ILP techniques to overcome the limitations imposed by a relatively small register file). Presently, a cache miss penalty to main memory costs several hundreds of CPU cycles. While in principle it is possible to use ILP to tolerate even such memory latencies the associated resource and power dissipation costs are disproportionate. Moreover, the complexity and often the latency of the underlying hardware structures results in reduced operating frequency further reducing any benefits. Hence, the aforementioned techniques prove inadequate to keep the CPU from stalling for the off-chip data. Instead, the industry is heading towards exploiting higher levels of parallelism that can be exploited through techniques such as multiprocessing and multithreading. [1]

## References

[1] Reflections of the Memory Wall (http://www.csl.cornell.edu/~sam/papers/cf04.pdf)

## External links

- Approaches to addressing the Memory Wall (http://www.itee.uq.edu.au/~philip/Publications/Techreports/2002/Reports/memory-wall-survey.pdf)
- Wired magazine article that refers to the above paper (http://www.wired.com/wired/archive/4.08/geek.html)

# Task parallelism

**Task parallelism** (also known as **function parallelism** and **control parallelism**) is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing execution processes (threads) across different parallel computing nodes. It contrasts to data parallelism as another form of parallelism.

## Description

In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication usually takes place by passing data from one thread to the next as part of a workflow.

As a simple example, if we are running code on a 2-processor system (CPUs "a" & "b") in a parallel environment and we wish to do tasks "A" and "B", it is possible to tell CPU "a" to do task "A" and CPU "b" to do task 'B" simultaneously, thereby reducing the run time of the execution. The tasks can be assigned using conditional statements as described below.

Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e. threads), as opposed to the data (data parallelism). Most real programs fall somewhere on a continuum between task parallelism and data parallelism.Wikipedia:Citation needed

## Example

The pseudocode below illustrates task parallelism:

```
program:
...
if CPU="a" then
   do task "A"
else if CPU="b" then
   do task "B"
end if
...
end program
```

The goal of the program is to do some net total task ("A+B"). If we write the code as above and launch it on a 2-processor system, then the runtime environment will execute it as follows.

- In an SPMD system, both CPUs will execute the code.
- In a parallel environment, both will have access to the same data.
- The "if" clause differentiates between the CPU's. CPU "a" will read true on the "if" and CPU "b" will read true on the "else if", thus having their own task.
- Now, both CPU's execute separate code blocks simultaneously, performing different tasks simultaneously.

Code executed by CPU "a":

```
program:
...
do task "A"
...
```

```
end program
```

Code executed by CPU "b":

```
program:
...
do task "B"
...
end program
```

This concept can now be generalized to any number of processors.

### JVM Example

Similar to the previous example, Task Parallelism is also possible using the Java Virtual Machine JVM.

The code below illustrates task parallelism on the JVM using the commercial third-party Ateji PX extension.[1] Statements or blocks of statements can be composed in parallel using the || operator inside a parallel block, introduced with square brackets:

```
[
   || a++;
   || b++;
]
```

or in short form:

```
[ a++; || b++; ]
```

Each parallel statement within the composition is called a branch. We purposely avoid using the terms task or process which mean very different things in different contexts.

## Languages

Examples of (fine-grained) task-parallel languages can be found in the realm of Hardware Description Languages like Verilog and VHDL, which can also be considered as representing a "code static" software paradigm where the program has a static structure and the data is changing - as against a "data static" model where the data is not changing (or changing slowly) and the processing (applied methods) change (e.g. database search).Wikipedia:Please clarify

## Notes

[1] http://www.ateji.com/px/patterns.html#task Task Parallelism using Ateji PX, an extension of Java

## References


- Quinn Michael J, Parallel Programming in C with MPI and OpenMP McGraw-Hill Inc. 2004. ISBN 0-07-058201-7
- D. Kevin Cameron (http://www.linkedin.com/in/kevcameron) coined terms "data static" and "code static". Wikipedia:Citation needed

# Data parallelism

**Data parallelism** is a form of parallelization of computing across multiple processors in parallel computing environments. Data parallelism focuses on distributing the data across different parallel computing nodes. It contrasts to task parallelism as another form of parallelism.

## Description

In a multiprocessor system executing a single set of instructions (SIMD), data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

For instance, consider a 2-processor system (CPUs A and B) in a parallel environment, and we wish to do a task on some data `d`. It is possible to tell CPU A to do that task on one part of `d` and CPU B on another part simultaneously, thereby reducing the duration of the execution. The data can be assigned using conditional statements as described below. As a specific example, consider adding two matrices. In a data parallel implementation, CPU A could add all elements from the top half of the matrices, while CPU B could add all elements from the bottom half of the matrices. Since the two processors work in parallel, the job of performing matrix addition would take one half the time of performing the same operation in serial using one CPU alone.

Data parallelism emphasizes the distributed (parallelized) nature of the data, as opposed to the processing (task parallelism). Most real programs fall somewhere on a continuum between task parallelism and data parallelism.

## Example

The program below expressed in pseudocode—which applies some arbitrary operation, `foo`, on every element in the array `d`—illustrates data parallelism:[1]

```
if CPU = "a"
   lower_limit := 1
   upper_limit := round(d.length/2)
else if CPU = "b"
   lower_limit := round(d.length/2) + 1
   upper_limit := d.length

for i from lower_limit to upper_limit by 1
   foo(d[i])
```

If the above example program is executed on a 2-processor system the runtime environment may execute it as follows:

- In an SPMD system, both CPUs will execute the code.
- In a parallel environment, both will have access to `d`.
- A mechanism is presumed to be in place whereby each CPU will create its own copy of `lower_limit` and `upper_limit` that is independent of the other.
- The `if` clause differentiates between the CPUs. CPU `"a"` will read true on the `if`; and CPU `"b"` will read true on the `else if`, thus having their own values of `lower_limit` and `upper_limit`.
- Now, both CPUs execute `foo(d[i])`, but since each CPU has different values of the limits, they operate on different parts of `d` simultaneously, thereby distributing the task among themselves. Obviously, this will be faster than doing it on a single CPU.

This concept can be generalized to any number of processors. However, when the number of processors increases, it may be helpful to restructure the program in a similar way (where `cpuid` is an integer between 1 and the number of CPUs, and acts as a unique identifier for every CPU):

```
for i from cpuid to d.length by number_of_cpus
   foo(d[i])
```

For example, on a 2-processor system CPU A (`cpuid` 1) will operate on odd entries and CPU B (`cpuid` 2) will operate on even entries.

### JVM Example

Similar to the previous example, Data Parallelism is also possible using the Java Virtual Machine JVM (using Ateji PX, an extension of Java).

The code below illustrates Data parallelism on the JVM: Branches in a parallel composition can be quantified. This is used to perform the || operator[2] on all elements of an array or a collection:

```
[
   // increment all array elements in parallel
   || (int i : N) array[i]++;
]
```

The equivalent sequential code would be:

```
[
   // increment all array elements one after the other
   for(int i : N) array[i]++;
]
```

Quantification can introduce an arbitrary number of generators (iterators) and filters. Here is how we would update the upper left triangle of a matrix:

```
[
   ||(int i:N, int j:N, if i+j<N) matrix[i][j]++;
]
```

## Notes

[1] Some input data (e.g. when `d.length` evaluates to 1 and `round` rounds towards zero [this is just an example, there are no requirements on what type of rounding is used]) will lead to `lower_limit` being greater than `upper_limit`, it's assumed that the loop will exit immediately (i.e. zero iterations will occur) when this happens.

[2] http://www.ateji.com/px/patterns.html#data Data Parallelism using Ateji PX, an extension of Java

## References


- Hillis, W. Daniel and Steele, Guy L., Data Parallel Algorithms (http://dx.doi.org/10.1145/7902.7903) Communications of the ACM December 1986
- Blelloch, Guy E, Vector Models for Data-Parallel Computing MIT Press 1990. ISBN 0-262-02313-X

# Uniform memory access

**Uniform memory access** (UMA) is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. Uniform memory access computer architectures are often contrasted with non-uniform memory access (NUMA) architectures. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.[1]

## Types of UMA architectures

1. UMA using bus-based symmetric multiprocessing (SMP) architectures
2. UMA using crossbar switches
3. UMA using multistage interconnection networks

## hUMA

In April 2013, the term "hUMA" (for heterogeneous Uniform Memory Access) began to appear in AMD promotional material to refer to CPU and GPU sharing the same system memory via cache coherent views. Advantages include an easier programming model and less copying of data between separate memory pools.[2]

## Notes

**[1]** Advanced Computer Architecture, Kai Hwang, ISBN 0-07-113342-9

[2] Peter Bright. AMD's "heterogeneous Uniform Memory Access" coming this year in Kaveri (http://arstechnica.com/information-technology/2013/04/amds-heterogeneous-uniform-memory-access-coming-this-year-in-kaveri/), Ars Technica, April 30, 2013.

# Non-uniform memory access

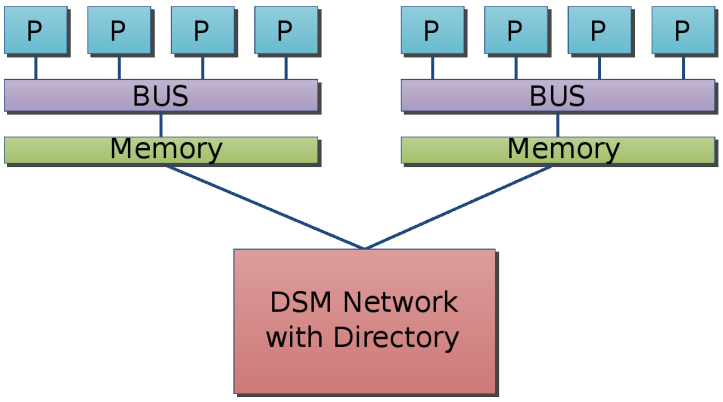
**Non-uniform memory access** (**NUMA**) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. They were developed commercially during the 1990s by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Honeywell Information Systems Italy (HISI) (later Groupe Bull), Silicon Graphics (later Silicon Graphics International), Sequent Computer Systems (later IBM), Data General (later EMC), and Digital (later Compaq, now HP). Techniques developed by these companies later featured in a variety of Unix-like operating systems, and to an extent in Windows NT.

The first commercial implementation of a NUMA-based Unix system was the Symmetrical Multi Processing XPS-100 family of servers, designed by Dan Gielan of VAST Corporation for Honeywell Information Systems Italy.

## Basic concept

Modern CPUs operate considerably faster than the main memory they use. In the early days of computing and data processing, the CPU generally ran slower than its own memory. The performance lines of processors and memory crossed in the 1960s with the advent of the first supercomputers. Since then, CPUs increasingly have found themselves "starved for data" and having to stall while waiting for data to arrive from memory. Many supercomputer designs of the 1980s and 1990s focused on providing high-speed memory access as opposed to faster processors, allowing the computers to work on large data sets at speeds other systems could not approach.




One possible architecture of a NUMA system. The processors connect to the bus or crossbar by connections of varying thickness/number. This shows that different CPUs have different access priorities to memory based on their relative location.

Limiting the number of memory accesses provided the key to extracting high performance from a modern computer. For commodity processors, this meant installing an ever-increasing amount of high-speed cache memory and using increasingly sophisticated algorithms to avoid cache misses. But the dramatic increase in size of the operating systems and of the applications run on them has generally overwhelmed these cache-processing improvements. Multi-processor systems without NUMA make the problem considerably worse. Now a system can starve several processors at the same time, notably because only one processor can access the computer's memory at a time.

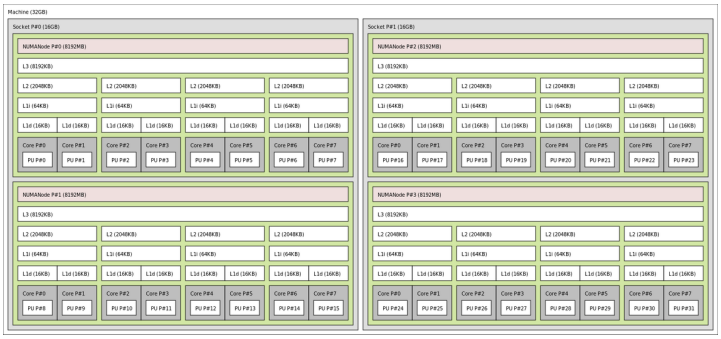
NUMA attempts to address this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory. For problems involving spread data (common for servers and similar applications), NUMA can improve the performance over a single shared memory by a factor of roughly the number of processors (or separate memory banks). Another approach to addressing this problem, utilized mainly by non-NUMA systems, is the multi-channel memory architecture; multiple memory channels are increasing the number of simultaneous memory accesses.

Of course, not all data ends up confined to a single task, which means that more than one processor may require the same data. To handle these cases, NUMA systems include additional hardware or software to move data between memory banks. This operation slows the processors attached to those banks, so the overall speed increase due to NUMA depends heavily on the nature of the running tasks.

Intel announced NUMA compatibility for its x86 and Itanium servers in late 2007 with its Nehalem and Tukwila CPUs.[1] Both CPU families share a common chipset; the interconnection is called Intel Quick Path Interconnect (QPI).[2] AMD implemented NUMA with its Opteron processor (2003), using HyperTransport. Freescale's NUMA for PowerPC is called CoreNet.

## Cache coherent NUMA (ccNUMA)

Nearly all CPU architectures use a small amount of very fast non-shared memory known as cache to exploit locality of reference in memory accesses. With NUMA, maintaining cache coherence across shared memory has a significant overhead. Although simpler to design and build, non-cache-coherent NUMA systems become prohibitively complex to program in the standard von Neumann architecture programming model.



Topology of a ccNUMA Bulldozer server.

Typically, ccNUMA uses inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, ccNUMA may perform poorly when multiple processors attempt to access the same memory area in rapid succession. Support for NUMA in operating systems attempts to reduce the frequency of this kind of access by allocating processors and memory in NUMA-friendly ways and by avoiding scheduling and locking algorithms that make NUMA-unfriendly accesses necessary.

Alternatively, cache coherency protocols such as the MESIF protocol attempt to reduce the communication required to maintain cache coherency. Scalable Coherent Interface (SCI) is an IEEE standard defining a directory-based cache coherency protocol to avoid scalability limitations found in earlier multiprocessor systems. For example, SCI is used as the basis for the NumaConnect technology.

As of 2011, ccNUMA systems are multiprocessor systems based on the AMD Opteron processor, which can be implemented without external logic, and the Intel Itanium processor, which requires the chipset to support NUMA. Examples of ccNUMA-enabled chipsets are the SGI Shub (Super hub), the Intel E8870, the HP sx2000 (used in the Integrity and Superdome servers), and those found in NEC Itanium-based systems. Earlier ccNUMA systems such as those from Silicon Graphics were based on MIPS processors and the DEC Alpha 21364 (EV7) processor.

## NUMA vs. cluster computing

One can view NUMA as a tightly coupled form of cluster computing. The addition of virtual memory paging to a cluster architecture can allow the implementation of NUMA entirely in software. However, the inter-node latency of software-based NUMA remains several orders of magnitude greater (slower) than that of hardware-based NUMA.

## Software support

Since NUMA largely influences memory access performance, certain software optimizations are needed to allow scheduling threads and processes close to their data.

- Microsoft Windows 7 and Windows Server 2008 R2 add support for NUMA architecture over 64 logical cores.[3]
- Java 7 added support for NUMA-aware memory allocator and garbage collector.[4]
- The Linux kernel 2.5 already had basic support built-in, which was further extended in subsequent releases. Linux kernel version 3.8 brought a new NUMA foundation which allowed more efficient NUMA policies to be built in the next kernel releases. Linux kernel version 3.13 brought numerous policies that attempt to put a process near its memory, together with handling of cases such as shared pages between processes, or transparent huge pages; new sysctl settings are allowing NUMA balancing to be enabled or disabled, as well as various NUMA memory balancing parameters to be configured.
- OpenSolaris models NUMA architecture with lgroups.

## References

[1] Intel Corp. (2008). Intel QuickPath Architecture [White paper]. Retrieved from http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf

[2] Intel Corporation. (September 18th, 2007). Gelsinger Speaks To Intel And High-Tech Industry's Rapid Technology Caden[Press release]. Retrieved from http://www.intel.com/pressroom/archive/releases/2007/20070918corp_b.htm

[3] NUMA Support (MSDN) (http://msdn.microsoft.com/en-us/library/windows/desktop/aa363804(v=vs.85).aspx)

[4] Java HotSpot™ Virtual Machine Performance Enhancements (http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#numa)



## External links

- NUMA FAQ (http://lse.sourceforge.net/numa/faq/)
- Page-based distributed shared memory (http://cs.gmu.edu/cne/modules/dsm/yellow/page_dsm.html)
- OpenSolaris NUMA Project (http://www.opensolaris.org/os/community/performance/numa/)
- Introduction video for the Alpha EV7 system architecture (http://h18002.www1.hp.com/alphaserver/nextgen/overview.wmv)
- More videos related to EV7 systems: CPU, IO, etc (http://www.alphaprocessors.com/)
- NUMA optimization in Windows Applications (http://developer.amd.com/pages/1162007106.aspx)
- NUMA Support in Linux at SGI (http://oss.sgi.com/projects/numa/)
- Intel Tukwila (http://www.realworldtech.com/page.cfm?NewsID=361&date=05-05-2006#361/)
- Intel QPI (CSI) explained (http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032)
- current Itanium NUMA systems (http://www.sql-server-performance.com/articles/per/high_call_volume_NUMA_p1.aspx)

# Crossbar switch

In electronics, a **crossbar switch** (also known as **cross-point switch**, **crosspoint switch**, or **matrix switch**) is a switch connecting multiple inputs to multiple outputs in a matrix manner. Originally the term was used literally, for a matrix switch controlled by a grid of crossing metal bars, and later was broadened to matrix switches in general. It is one of the principal switch architectures, together with a rotary switch, memory switch and a crossover switch.

## General properties

A crossbar switch is an assembly of individual switches between multiple inputs and multiple outputs. The switches are arranged in a matrix. If the crossbar switch has M inputs and N outputs, then a crossbar has a matrix with $M \times N$ cross-points or places where the "bars" cross. At each crosspoint is a switch; when closed, it connects one of $M$ inputs to one of $N$ outputs. A given crossbar is a single layer, non-blocking switch. "Non-blocking" means that other concurrent connections do not prevent connecting an arbitrary input to any arbitrary output. Collections of crossbars can be used to implement multiple layer and/or blocking switches. A crossbar switching system is also called a co-ordinate switching system.

## Applications

Crossbar switches are most famously used in information processing applications such as telephony and circuit switching, but they are also used in applications such as mechanical sorting machines.

The matrix layout of a crossbar switch is also used in some semiconductor memory devices. Here the "bars" are extremely thin metal "wires", and the "switches" are fusible links. The fuses are blown or opened using high voltage and read using low voltage. Such devices are called programmable read-only memory.[1] At the 2008 NSTI Nanotechnology Conference a paper was presented which discussed a nanoscale crossbar implementation of an adding circuit used as an alternative to logic gates for computation.

Furthermore, matrix arrays are fundamental to modern flat-panel displays. Thin-film-transistor LCDs have a transistor at each crosspoint, so they could be considered to include a crossbar switch as part of their structure.

For video switching in home and professional theater applications, a crossbar switch (or a matrix switch, as it is more commonly called in this application) is used to make the output of multiple video appliances available simultaneously to every monitor or every room throughout a building. In a typical installation, all the video sources are located on an equipment rack, and are connected as inputs to the matrix switch.

Where central control of the matrix is practical, a typical rack-mount matrix switch offers front-panel buttons to allow manual connection of inputs to outputs. An example of such a usage might be a sports bar, where numerous programs are displayed simultaneously. In order to accomplish this, a sports bar would ordinarily need to purchase / rent a separate cable or satellite box for each display for which independent control is desired. The matrix switch enables the signals to be re-routed on a whim, thus allowing the establishment to purchase / rent only those boxes needed to cover the total number of *unique* programs viewed anywhere in the building also it makes it easier to control and get sound from any program to the overall speaker / sound system.

Such switches are used in high-end home theater applications. Video sources typically shared include set-top cable/satellite receivers or DVD changers; the same concept applies to audio as well. The outputs are wired to televisions in individual rooms. The matrix switch is controlled via an Ethernet or RS-232 serial connection by a whole-house automation controller, such as those made by AMX, Crestron, or Control4 - which provides the user interface that enables the user in each room to select which appliance to watch. The actual user interface varies by system brand, and might include a combination of on-screen menus, touch-screens, and handheld remote controls. The system is necessary to enable the user to select the program they wish to watch from the same room they will watch it from, otherwise it would be necessary (and arguably absurd) for them to walk to the equipment rack.

The special crossbar switches used in distributing satellite TV signals are called multiswitches.

## Implementations

Historically, a crossbar switch consisted of metal bars associated with each input and output, together with some means of controlling movable contacts at each cross-point. In the later part of the 20th Century these literal crossbar switches declined and the term came to be used figuratively for rectangular array switches in general. Modern "crossbar switches" are usually implemented with semiconductor technology. An important emerging class of optical crossbars is being implemented with MEMS technology.

### Mechanical

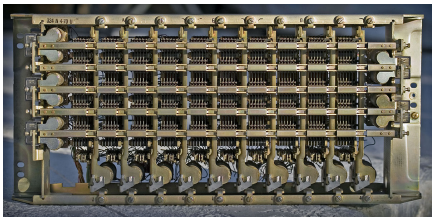
A type of middle 19th Century telegraph exchange consisted of a grid of vertical and horizontal brass bars with a hole at each intersection. The operator inserted a brass pin to connect one telegraph line to another.

### Electromechanical/telephony

A telephony crossbar switch is an electromechanical device for switching telephone calls. The first design of what is now called a crossbar switch was the Bell company Western Electric's "coordinate selector" of 1915. To save money on control systems, this system was organized on the stepping switch or selector principle rather than the link principle. It was little used in America, but the Televerket Swedish governmental agency manufactured its own design (the Gotthilf Betulander design from 1919, inspired by the Western Electric system), and used it in Sweden from 1926 until the digitalization in the 1980s in small and medium sized A204 model switches. The system design used in AT&T Corporation's 1XB crossbar exchanges, which entered revenue service from 1938, developed by Bell Telephone Labs, was inspired by the Swedish design but was based on the rediscovered link principle. In 1945, a similar design by Swedish Televerket was installed in Sweden, making it possible to increase the capacity of the A204 model switch. Delayed by the Second World War, several millions of urban 1XB lines were installed from the 1950s in the United States.

In 1950, the Ericsson Swedish company developed their own versions of the 1XB and A204 systems for the international market. In the early 1960s, the company's sales of crossbar switches exceeded those of their rotating 500-switching system, as measured in the number of lines. Crossbar switching quickly spread to the rest of the world, replacing most earlier designs like the Strowger (step-by-step) and Panel systems in larger installations in the U.S. Graduating from entirely electromechanical control on introduction, they were gradually elaborated to have full electronic control and a variety of calling features including short-code and speed-dialing. In the UK the Plessey Company produced a range of TXK crossbar exchanges, but their widespread rollout by the British Post Office began later than in other countries, and then was inhibited by the parallel development of TXE reed relay and electronic exchange systems, so they never achieved a large number of customer connections although they did find some success as tandem switch exchanges.

Crossbar switches use switching matrices made from a two-dimensional array of contacts arranged in an x-y format. These switching matrices are operated by a series of horizontal bars arranged over the contacts. Each such "select" bar can be rocked up or down by electromagnets to provide access to two levels of the matrix. A second set of vertical "hold" bars is set at right angles to the first (hence the name, "crossbar") and also operated by electromagnets. The select bars carry spring-loaded wire fingers that enable the hold bars to operate the contacts beneath the bars. When the select and then the hold electromagnets operate in sequence to move the bars, they trap one of the spring fingers to close the contacts beneath the point where two bars cross. This then makes the connection through the switch as part of setting up a calling path through the exchange. Once connected, the select magnet is then released so it can use its other fingers for other connections, while the hold magnet remains energized for the duration of the call to maintain the connection. The crossbar switching interface was referred to as the TXK or TXC switch (*T*elephone e*X*change *C*rossbar) - in the UK.

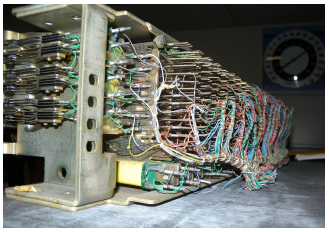


Western Electric 100 Point six-wire Type B crossbar switch

The Bell System *Type B* crossbar switch of the 1960s was made in the largest quantity. The majority were 200 point switches, with twenty verticals and ten levels of three wires, but this example is a 100 point six wire switch. Each select bar carries ten fingers so any of the ten circuits assigned to the ten verticals can connect to either of two levels. Five select bars, each able to rotate up or down, mean a choice of ten links to the next stage of switching. Each crosspoint in this particular model connected six wires. Note the *Vertical Off-Normal* contacts next to the hold magnets, lined up along the bottom of the switch. They perform logic and memory functions, and the hold bar keeps them in the active position as long as the connection is up. The *Horizontal Off Normals* on the sides of the switch are activated by the horizontal bars when the "butterfly magnets" rotate them. This only happens while the connection is being set up, since the butterflies are only energized then.

Late-model Western Electric crossbar switch

The majority of Bell System switches were made to connect three wires including the tip and ring of a balanced pair circuit and a sleeve lead for control. Many connected six wires, either for two distinct circuits or for a four wire circuit or other complex connection. The Bell System *Type C* miniature crossbar of the 1970s was similar, but the fingers projected forward from the back and the select bars held paddles to move them. The majority of type C had twelve levels; these were the less common ten level ones. The Northern Electric *Minibar* used in SP1 switch was similar but even smaller. The ITT Pentaconta Multiswitch of the same era had usually 22 verticals, 26 levels, and six to twelve wires. Ericsson crossbar switches sometimes had only five verticals.



Back of Type C

## Electromechanical/instrumentation

For instrumentation use, James Cunningham, Son and Company[2] made high-speed, very-long-life crossbar switches[3] with physically small mechanical parts which permitted faster operation than telephone-type crossbar switches. Many of their switches had the mechanical Boolean AND function of telephony crossbar switches, but other models had individual relays (one coil per crosspoint) in matrix arrays, connecting the relay contacts to [x] and [y] buses. These latter types were equivalent to separate relays; there was no logical AND function built in. Cunningham crossbar switches had precious-metal contacts capable of handling millivolt signals.

## Telephone exchange

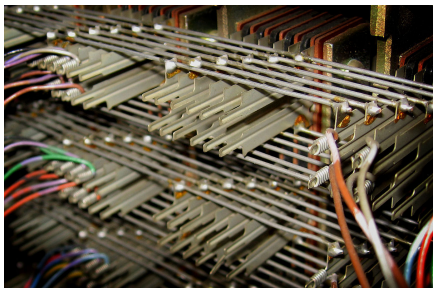
Early crossbar exchanges were divided into an originating side and a terminating side, while the later and prominent Canadian and US SP1 switch and 5XB switch were not. When a user picked up the telephone handset, the resulting line loop operating the user's line relay caused the exchange to connect the user's telephone to an originating sender, which returned the user a dial tone. The sender then recorded the dialed digits and passed them to the originating marker, which selected an outgoing trunk and operated the various crossbar switch stages to connect the calling user to it. The originating marker then passed the trunk call completion requirements (type of pulsing, resistance of the trunk, etc.) and the called party's details to the sender and released. The sender then relayed this information to a terminating sender (which could be on either the same or a different exchange). This sender then used a terminating marker to connect the calling user, via the selected incoming trunk, to the called user, and caused the controlling

relay set to pass intermittent ring voltage of about 90 V AC at 20 Hz to ring the called user's phone bell, and return ringing tone to the caller.

The crossbar switch itself was simple: exchange design moved all the logical decision-making to the common control elements, which were very reliable as relay sets. The design criterion was to have two hours of "downtime" for service every forty years, which was a huge improvement on earlier electromechanical systems. The exchange design concept lent itself to incremental upgrades, as the control elements could be replaced separately from the call switching elements. The minimum size of a crossbar exchange was comparatively large, but in city areas with a large installed line capacity the whole exchange occupied less space than other exchange technologies of equivalent capacity. For this reason they were also typically the first switches to be replaced with digital systems, which were even smaller and more reliable.

Two principles of using crossbar switches were used. One early method was based on the selector principle, and used the switches as functional replacement for Strowger or stepping switches. Control was distributed to the switches themselves. Call establishment progressed through the exchange stage by stage, as successive digits were dialed. With the selector principle, each switch could only handle its portion of one call at a time. Each moving contact of the array was multipledWikipedia:Please clarify to corresponding crosspoints on other switches to a selector in the next bank of switches. Thus an exchange with a hundred 10×10 switches in five stages could only have twenty conversations in progress. Distributed control meant there was no common point of failure, but also meant that the setup stage lasted for the ten seconds or so the caller took to dial the required number. In control occupancy terms this comparatively long interval degrades the traffic capacity of a switch.

Starting with the 1XB switch, the later and more common method was based on the link principle, and used the switches as crosspoints. Each moving contact was multipled to the other contacts on the same level by simpler "banjo" wires, to a link on one of the inputs of a switch in the next stage. The switch could handle its portion of as many calls as it had levels or verticals. Thus an exchange with forty 10×10 switches in four stages could have a hundred conversations in progress. The link principle was more efficient, but required a more complex control system to find idle links through the switching fabric.



"Banjo" wiring of a 100 point six wire Type B Bell System switch

This meant common control, as described above: all the digits were recorded, then passed to the common control equipment, the marker, to establish the call at all the separate switch stages simultaneously. A marker-controlled crossbar system had in the marker a highly vulnerable central control; this was invariably protected by having duplicate markers. The great advantage was that the control occupancy on the switches was of the order of one second or less, representing the operate and release lags of the X-then-Y armatures of the switches. The only downside of common control was the need to provide digit recorders enough to deal with the greatest forecast originating traffic level on the exchange.

The Plessey TXK1 or 5005 design used an intermediate form, in which a clear path was marked through the switching fabric by distributed logic, and then closed through all at once.

In some countries, no crossbar exchanges remain in revenue service. However, crossbar exchanges remain in use in countries like Russia, where some massive city telephone networks have not yet been fully upgraded to digital technology. Preserved installations may be seen in museums like The Museum of Communications [4] in Seattle, Washington, and the Science Museum in London.

Changing nomenclature can confuse: in current American terminology a "switch" now frequently refers to a system which is also called a "telephone exchange" (the usual term in English)—that is, a large collection of selectors of some sort within a building. For most of the twentieth century a "Strowger switch" or a "crossbar switch" referred to an individual piece of mechanical equipment making up part of an exchange. Hence the pictures above show a

"crossbar switch" using the earlier meaning.

### Semiconductor

Semiconductor implementations of crossbar switches typically consist of a set of input amplifiers or retimers connected to a series of metalizations or "bars" within a semiconductor device. A similar set of metalizations or "bars" are connected to output amplifiers or retimers. At each cross-point where the "bars" cross, a pass transistor is implemented which connects the bars. When the pass transistor is enabled, the input is connected to the output.

As computer technologies have improved, crossbar switches have found uses in systems such as the multistage interconnection networks that connect the various processing units in a uniform memory access parallel processor to the array of memory elements.

## Arbitration

A standard problem in using crossbar switches is that of setting the cross-points. In the classic telephony application of cross-bars, the crosspoints are closed and open as the telephone calls come and go. In Asynchronous Transfer Mode or packet switching applications, the crosspoints must be made and broken at each decision interval. In high-speed switches, the settings of all of the cross-points must be determined and then set millions or billions of times per second. One approach for making these decisions quickly is through the use of a wavefront arbiter.

## References

[1] Yong Chen, Gun-Young Jung, Douglas A A Ohlberg, Xuema Li, Duncan R Stewart, Jan O Jeppesen, Kent A Nielsen, and J Fraser Stoddart, Nanoscale molecular-switch crossbar circuits, 2003 *Nanotechnology* 14 462-468

[2] http://www.obs-us.com/people/karen/cunningham/chapter6.htm#47

[3] http://www.obs-us.com/people/karen/cunningham/47.htm

[4] http://www.museumofcommunications.org/

## External links

- 1XB (http://www.telephonetribute.com/switches_survey_chapter_6.html)
- 5XB (http://www.telephonetribute.com/switches_survey_chapter_7.html)
- 4XB (http://www.telephonetribute.com/switches_survey_chapter_8.html)
- XBT (http://www.telephonetribute.com/switches_survey_chapter_9.html)
- Ericsson ARF (in Dutch) (http://www.actw.nl/Openbare centrales/ARF.htm)
- Photos of Bell System crossbar switches and equipment (http://xy3.com/phone/your calls and mine.shtml)
- Crossbar switch diagram in paraller processing (http://img85.imageshack.us/img85/1050/crossbarswitcher.png)

# Mesh networking

A **mesh network** is a network topology in which each node (called a mesh node) relays data for the network. All nodes cooperate in the distribution of data in the network.

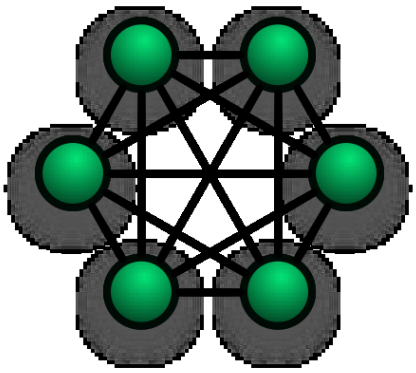


Illustration of a mesh network.

A mesh network can be designed using a *flooding* technique or a *routing* technique. When using a routing technique, the message is propagated along a path, by *hopping* from node to node until the destination is reached. To ensure all its paths' availability, a routing network must allow for continuous connections and reconfiguration around broken or blocked paths, using *self-healing* algorithms. A mesh network whose nodes are all connected to each other is a fully connected network. Fully connected wired networks have the advantages of security and reliability: problems in a cable affect only the two nodes attached to it. However, in such networks, the number of cables, and therefore the cost, goes up rapidly as the number of nodes increases.

Mesh networks can be seen as one type of ad hoc network. Mobile ad hoc networks (MANETs) and mesh networks are therefore closely related, but a MANET also must deal with problems introduced by the mobility of the nodes.

The self-healing capability enables a routing-based network to operate when one node breaks down or a connection goes bad. As a result, the network is typically quite reliable, as there is often more than one path between a source and a destination in the network. Although mostly used in wireless situations, this concept is also applicable to wired networks and software interaction.

## Wireless mesh networks

Wireless mesh networks were originally developed for military applications. Mesh networks are typically wireless. Over the past decade, the size, cost, and power requirements of radios has declined, enabling multiple radios to be contained within a single mesh node, thus allowing for greater modularity; each can handle multiple frequency bands and support a variety of functions as needed—such as client access, backhaul service, and scanning (required for high-speed handoff in mobile applications)—even customized sets of them.Wikipedia:Please clarify

Work in this field has been aided by the use of game theory methods to analyze strategies for the allocation of resources and routing of packets.

### Examples

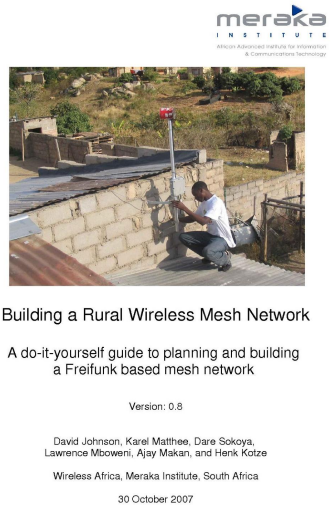
In rural Catalonia, Guifi.net was developed in 2004 as a response to the lack of broadband internet, where commercial internet providers weren't providing a connection or a very poor one. Nowadays with more than 30,000 nodes it is only halfway a fully connected network, but following a peer to peer agreement it remained an open, free and neutral network with extensive redundancy.

Zigbee digital radios are incorporated into some consumer appliances, including battery-powered appliances. Zigbee radios spontaneously organize a mesh network, using AODV routing; transmission and reception are synchronized. This means the radios can be off much of the time, and thus conserve power.

In early 2007, the US-based firm Meraki launched a mini wireless mesh router. This is an example of a wireless mesh network (on a claimed speed of up to 50 megabits per second). The 802.11 radio within the Meraki Mini has been optimized for long-distance communication, providing coverage over 250 metres.

This is an example of a single-radio mesh network being used within a community as opposed to multi-radio long range mesh networks like BelAir or MeshDynamics that provide multifunctional infrastructure, typically using tree based topologies and their advantages in O(n) routing.

The Naval Postgraduate School, Monterey CA, demonstrated such wireless mesh networks for border security. In a pilot system, aerial cameras kept aloft by balloons relayed real time high resolution video to ground personnel via a mesh network.




Building a Rural Wireless Mesh Network: A DIY Guide (PDF)

An MIT Media Lab project has developed the XO-1 laptop or "OLPC"(One Laptop per Child) which is intended for disadvantaged schools in developing nations and uses mesh networking (based on the IEEE 802.11s standard) to create a robust and inexpensive infrastructure. The instantaneous connections made by the laptops are claimed by the project to reduce the need for an external infrastructure such as the Internet to reach all areas, because a connected node could share the connection with nodes nearby. A similar concept has also been implemented by Greenpacket with its application called SONbuddy.

In Cambridge, UK, on 3 June 2006, mesh networking was used at the “Strawberry Fair” to run mobile live television, radio and Internet services to an estimated 80,000 people.

The Champaign-Urbana Community Wireless Network (CUWiN) project is developing mesh networking software based on open source implementations of the Hazy-Sighted Link State Routing Protocol and Expected Transmission Count metric. Additionally, the Wireless Networking Group in the University of Illinois at Urbana-Champaign are developing a multichannel, multi-radio wireless mesh testbed, called Net-X as a proof of concept implementation of some of the multichannel protocols being developed in that group. The implementations are based on an architecture that allows some of the radios to switch channels to maintain network connectivity, and includes protocols for channel allocation and routing.

FabFi is an open-source, city-scale, wireless mesh networking system originally developed in 2009 in Jalalabad, Afghanistan to provide high-speed internet to parts of the city and designed for high performance across multiple hops. It is an inexpensive framework for sharing wireless internet from a central provider across a town or city. A second larger implementation followed a year later near Nairobi, Kenya with a "freemium"Wikipedia:Please clarify pay model to support network growth. Both projects were undertaken by the Fablab users of the respective cities.

SMesh is an 802.11 multi-hop wireless mesh network developed by the Distributed System and Networks Lab at Johns Hopkins University. A fast handoff scheme allows mobile clients to roam in the network without interruption in connectivity, a feature suitable for real-time applications, such as VoIP.

Many mesh networks operate across multiple radio bands. For example Firetide and Wave Relay mesh networks have the option to communicate node to node on 5.2 GHz or 5.8 GHz, but communicate node to client on 2.4 GHz (802.11). This is accomplished using software-defined radio (SDR).

The SolarMESH project examined the potential of powering 802.11-based mesh networks using solar power and rechargeable batteries. Legacy 802.11 access points were found to be inadequate due to the requirement that they be continuously powered.[1] The IEEE 802.11s standardization efforts are considering power save options, but

solar-powered applications might involve single radio nodes where relay-link power saving will be inapplicable.

The WING project [2] (sponsored by the Italian Ministry of University and Research and led by CREATE-NET and Technion) developed a set of novel algorithms and protocols for enabling wireless mesh networks as the standard access architecture for next generation Internet. Particular focus has been given to interference and traffic aware channel assignment, multi-radio/multi-interface support, and opportunistic scheduling and traffic aggregation in highly volatile environments.

WiBACK Wireless Backhaul Technology has been developed by the Fraunhofer Institute for Open Communication Systems (FOKUS) in Berlin. Powered by solar cells and designed to support all existing wireless technologies, networks are due to be rolled out to several countries in sub-Saharan Africa in summer 2012.

Recent standards for wired communications have also incorporated concepts from Mesh Networking. An example is ITU-T G.hn, a standard that specifies a high-speed (up to 1 Gbit/s) local area network using existing home wiring (power lines, phone lines and coaxial cables). In noisy environments such as power lines (where signals can be heavily attenuated and corrupted by noise) it's common that mutual visibility between devices in a network is not complete. In those situations, one of the nodes has to act as a relay and forward messages between those nodes that cannot communicate directly, effectively creating a mesh network. In G.hn, relaying is performed at the Data Link Layer.

## References

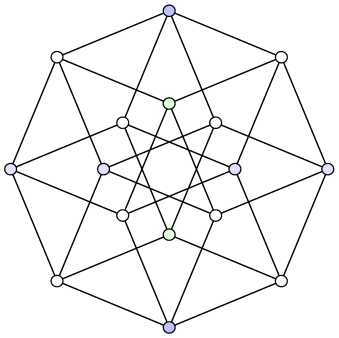
[1] Terence D. Todd, Amir A. Sayegh, Mohammed N. Smadi, and Dongmei Zhao. The Need for Access Point Power Saving in Solar Powered WLAN Mesh Networks (http://ieeexplore.ieee.org/search/freesrchabstract.jsp?arnumber=4519963&isnumber=4519956&punumber=65&k2dockey=4519963@ieeejrns). In IEEE Network, May/June 2008.

[2] http://www.wing-project.org WING

## External links

- Battelle Institute AoA (http://www.slideshare.net/DaCostaFrancis/battelle-aoa-evaluation-report-on-military-mesh-network-products) Comparative Ratings for popular mesh network providers, specific to mission critical military programs.
- MIT Roofnet (http://www.pdos.lcs.mit.edu/roofnet/) A research project at MIT that forms the basis of roofnet / Meraki mesh networks
- WING Project (http://www.wing-project.org) Wireless Mesh Network distribution based on the roofnet source code
- First, Second and Third Generation Mesh Architectures (http://www.meshdynamics.com/documents/MDThirdGenerationMesh.pdf) History and evolution of Mesh Networking Architectures
- DARPA's ITMANET program and the FLoWS Project (http://www.mit.edu/~medard/itmanet) Investigating Fundamental Performance Limits of MANETS
- Robin Chase discusses Zipcar and Mesh networking (http://www.ted.com/talks/view/id/212) Robin Chase talks at the Ted conference about the future of mesh networking and eco-technology
- Dynamic And Persistent Mesh Networks (http://www.slideshare.net/DaCostaFrancis/military-defense-and-public-safety-mesh-networks) Hybrid mesh networks for military, homeland security and public safety
- Mesh Networks Research Group (http://www.mesh-networks.org/) Projects and tutorials' compilation related to the Wireless Mesh Networks
- Phantom (http://code.google.com/p/phantom/) anonymous, decentralized network, isolated from the Internet
- Qaul Project (http://www.qaul.net/) Text messaging, file sharing and voice calls independent of internet and cellular networks.

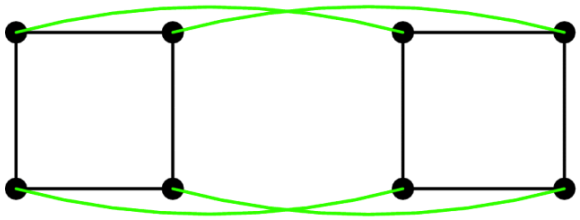
# Hypercube graph

| Hypercube graph | |
|---|---|
|  <br>The hypercube graph $Q_4$ | |
| **Vertices** | $2^n$ |
| **Edges** | $2^{n-1}n$ |
| **Diameter** | $n$ |
| **Girth** | 4 if $n \geq 2$ |
| **Automorphisms** | $n!\ 2^n$ |
| **Chromatic number** | 2 |
| **Spectrum** | $\{(n-2k)^{\binom{n}{k}}; k = 0, \ldots, n\}$ |
| **Properties** | Symmetric<br>Distance regular<br>Unit distance<br>Hamiltonian<br>Bipartite |
| **Notation** | $Q_n$ |
| • v<br>• t<br>• e [1] | |

In graph theory, the **hypercube graph** $Q_n$ is a regular graph with $2^n$ vertices, $2^{n-1}n$ edges, and $n$ edges touching each vertex. It can be obtained as the one-dimensional skeleton of the geometric hypercube; for instance, $Q_3$ is the graph formed by the 8 vertices and 12 edges of a three-dimensional cube. Alternatively, it can be obtained from the family of subsets of a set with $n$ elements, by making a vertex for each possible subset and joining two vertices by an edge whenever the corresponding subsets differ in a single element.

Hypercube graphs should not be confused with cubic graphs, which are graphs that have exactly three edges touching each vertex. The only hypercube that is a cubic graph is $Q_3$.

## Construction



Construction of $Q_3$ by connecting pairs of corresponding vertices in two copies of $Q_2$

The hypercube graph $Q_n$ may be constructed from the family of subsets of a set with $n$ elements, by making a vertex for each possible subset and joining two vertices by an edge whenever the corresponding subsets differ in a single element. Equivalently, it may be constructed using $2^n$ vertices labeled with $n$-bit binary numbers and connecting two vertices by an edge whenever the Hamming distance of their labels is 1. These two constructions are closely related: a binary number may be interpreted as a set (the set of positions where it has a 1 digit), and two such sets differ in a single element whenever the corresponding two binary numbers have Hamming distance 1.

Alternatively, $Q_{n+1}$ may be constructed from the disjoint union of two hypercubes $Q_n$, by adding an edge from each vertex in one copy of $Q_n$ to the corresponding vertex in the other copy, as shown in the figure. The joining edges form a perfect matching.

Another definition of $Q_n$ is the Cartesian product of $n$ two-vertex complete graphs $K_2$.

## Examples

The graph $Q_0$ consists of a single vertex, while $Q_1$ is the complete graph on two vertices and $Q_2$ is a cycle of length 4.

The graph $Q_3$ is the 1-skeleton of a cube, a planar graph with eight vertices and twelve edges.

The graph $Q_4$ is the Levi graph of the Möbius configuration.

## Properties

### Bipartiteness

Every hypercube graph is bipartite: it can be colored with only two colors. The two colors of this coloring may be found from the subset construction of hypercube graphs, by giving one color to the subsets that have an even number of elements and the other color to the subsets with an odd number of elements.

### Hamiltonicity

Every hypercube $Q_n$ with $n > 1$ has a Hamiltonian cycle, a cycle that visits each vertex exactly once. Additionally, a Hamiltonian path exists between two vertices $u,v$ if and only if have different colors in a 2-coloring of the graph. Both facts are easy to prove using the principle of induction on the dimension of the hypercube, and the construction of the hypercube graph by joining two smaller hypercubes with a matching.

Hamiltonicity of the hypercube is tightly related to the theory of Gray codes. More precisely there is a bijective correspondence between the set of $n$-bit cyclic Gray codes and the set of Hamiltonian cycles in the hypercube $Q_n$. An analogous property holds for acyclic $n$-bit Gray codes and Hamiltonian paths.

A lesser known fact is that every perfect matching in the hypercube extends to a Hamiltonian cycle. The question whether every matching extends to a Hamiltonian cycle remains an open problem.[2]

### Other properties

The hypercube graph $Q_n$ (n > 1) :

- is the Hasse diagram of a finite Boolean algebra.
- is a median graph. Every median graph is an isometric subgraph of a hypercube, and can be formed as a retraction of a hypercube.
- has more than $2^2$n-2 perfect matchings. (this is another consequence that follows easily from the inductive construction.)
- is arc transitive and symmetric. The symmetries of hypercube graphs can be represented as signed permutations.
- contains all the cycles of length 4, 6, ..., $2^n$ and is thus a bipancyclic graph.
- can be drawn as a unit distance graph in the Euclidean plane by choosing a unit vector for each set element and placing each vertex corresponding to a set *S* at the sum of the vectors in *S*.
- is a *n*-vertex-connected graph, by Balinski's theorem
- is planar (can be drawn with no crossings) if and only if **n** ≤ 3. For larger values of *n*, the hypercube has genus $(n-4)2^{n-3}+1$.
- has exactly $2^{2^n-n-1}\prod_{k=2}^{n} k^{\binom{n}{k}}$ spanning trees.
- The family $Q_n$ (n > 1) is a Lévy family of graphs
- The achromatic number of $Q_n$ is known to be proportional to $\sqrt{n2^n}$, but the constant of proportionality is not known precisely.
- The bandwidth of $Q_n$ is exactly $\sum_{i=0}^{n} \binom{n}{\lfloor n/2 \rfloor}$.[3]
- The eigenvalues of the adjacency matrix are (-n,-n+2,-n+4,...,n-4,n-2,n) and the eigenvalues of its Laplacian are (0,2,...,2n). The k-th eigenvalue has multiplicity $\binom{n}{k}$ in both cases.
- The isoperimetric number is h(G)=1

## Problems

The problem of finding the longest path or cycle that is an induced subgraph of a given hypercube graph is known as the snake-in-the-box problem.

Szymanski's conjecture concerns the suitability of a hypercube as an network topology for communications. It states that, no matter how one chooses a permutation connecting each hypercube vertex to another vertex with which it should be connected, there is always a way to connect these pairs of vertices by paths that do not share any directed edge.

## Notes

[1] http://en.wikipedia.org/w/index.php?title=Template:Infobox_graph&action=edit

[2] Ruskey, F. and Savage, C. Matchings extend to Hamiltonian cycles in hypercubes (http://garden.irmacs.sfu.ca/?q=op/matchings_extends_to_hamilton_cycles_in_hypercubes) on Open Problem Garden. 2007.

[3] Optimal Numberings and Isoperimetric Problems on Graphs, L.H. Harper, Journal of Combinatorial Theory, 1, 385–393,

## References


- Harary, F.; Hayes, J. P.; Wu, H.-J. (1988), "A survey of the theory of hypercube graphs", *Computers & Mathematics with Applications* **15** (4): 277–289, doi: 10.1016/0898-1221(88)90213-1 (http://dx.doi.org/10.1016/0898-1221(88)90213-1).


# Multi-core processor

A **multi-core processor** is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing.[1] Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

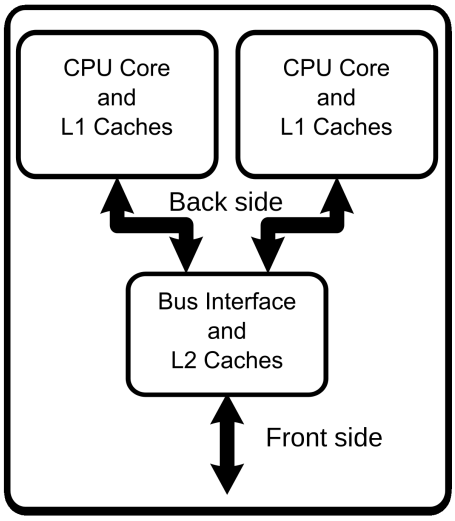


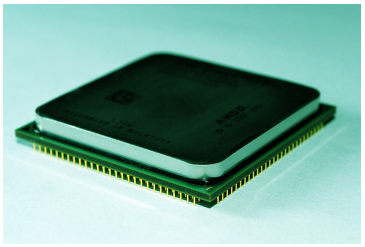

Diagram of a generic dual-core processor, with CPU-local level 1 caches, and a shared, on-die level 2 cache.

Processors were originally developed with only one core. Multi-core processors were developed in the early 2000s by Intel, AMD and others. Multicore processors may have two cores (dual-core CPUs, for example AMD Phenom II X2 and Intel Core Duo), four cores (quad-core CPUs, for example AMD Phenom II X4, Intel's i5 and i7 processors), six cores (hexa-core CPUs, for example AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X), eight cores (octo-core CPUs, for example Intel Xeon E7-2820 and AMD FX-8350), ten cores (for example, Intel Xeon E7-2850), or more. A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. *Homogeneous* multi-core systems include only identical cores, heterogeneous multi-core systems have cores that are not identical. Just as with single-processor systems, cores in multi-core systems may implement architectures such as superscalar, VLIW, vector processing, SIMD, or multithreading.

An Intel Core 2 Duo E6750 dual-core processor.

Multi-core processors are widely used across many application domains including general-purpose, embedded, network, digital signal processing (DSP), and graphics.

The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be run in parallel simultaneously on multiple cores; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main system memory. Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem. The parallelization of software is a significant ongoing topic of research.



An AMD Athlon X2 6400+ dual-core processor.

## Terminology

The terms *multi-core* and *dual-core* most commonly refer to some sort of central processing unit (CPU), but are sometimes also applied to digital signal processors (DSP) and system-on-a-chip (SoC). The terms are generally used only to refer to multi-core microprocessors that are manufactured on the *same* integrated circuit die; separate microprocessor dies in the same package are generally referred to by another name, such as *multi-chip module*. This article uses the terms "multi-core" and "dual-core" for CPUs manufactured on the *same* integrated circuit, unless otherwise noted.

In contrast to multi-core systems, the term *multi-CPU* refers to multiple physically separate processing-units (which often contain special circuitry to facilitate communication between each other).

The terms *many-core* and *massively multi-core* are sometimes used to describe multi-core architectures with an especially high number of cores (tens or hundreds).[2]

Some systems use many soft microprocessor cores placed on a single FPGA. Each "core" can be considered a "semiconductor intellectual property core" as well as a CPU coreWikipedia:Citation needed.

## Development

While manufacturing technology improves, reducing the size of individual gates, physical limits of semiconductor-based microelectronics have become a major design concern. These physical limitations can cause significant heat dissipation and data synchronization problems. Various other methods are used to improve CPU performance. Some *instruction-level parallelism* (ILP) methods such as superscalar pipelining are suitable for many applications, but are inefficient for others that contain difficult-to-predict code. Many applications are better suited to *thread level parallelism* (TLP) methods, and multiple independent CPUs are commonly used to increase a system's overall TLP. A combination of increased available space (due to refined manufacturing processes) and the demand for increased TLP led to the development of multi-core CPUs.

### Commercial incentives

Several business motives drive the development of multi-core architectures. For decades, it was possible to improve performance of a CPU by shrinking the area of the integrated circuit, which drove down the cost per device on the IC. Alternatively, for the same circuit area, more transistors could be utilized in the design, which increased functionality, especially for CISC architectures. Clock rates also increased by orders of magnitude in the decades of the late 20th century, from several megahertz in the 1980s to several gigahertz in the early 2000s.

As the rate of clock speed improvements slowed, increased use of parallel computing in the form of multi-core processors has been pursued to improve overall processing performance. Multiple cores were used on the same CPU chip, which could then lead to better sales of CPU chips with two or more cores. Intel has produced a 48-core

processor for research in cloud computing; each core has an X86 architecture. Intel has loaded Linux on each core.

## Technical factors

Since computer manufacturers have long implemented symmetric multiprocessing (SMP) designs using discrete CPUs, the issues regarding implementing multi-core processor architecture and supporting it with software are well known.

Additionally:

- Utilizing a proven processing-core design without architectural changes reduces design risk significantly.
- For general-purpose processors, much of the motivation for multi-core processors comes from greatly diminished gains in processor performance from increasing the operating frequency. This is due to three primary factors:
  1. The *memory wall*; the increasing gap between processor and memory speeds. This effect pushes cache sizes larger in order to mask the latency of memory. This helps only to the extent that memory bandwidth is not the bottleneck in performance.
  2. The *ILP wall*; the increasing difficulty of finding enough parallelism in a single instruction stream to keep a high-performance single-core processor busy.
  3. The *power wall*; the trend of consuming exponentially increasing power with each factorial increase of operating frequency. This increase can be mitigated by "shrinking" the processor by using smaller traces for the same logic. The *power wall* poses manufacturing, system design and deployment problems that have not been justified in the face of the diminished gains in performance due to the *memory wall* and *ILP wall*.

In order to continue delivering regular performance improvements for general-purpose processors, manufacturers such as Intel and AMD have turned to multi-core designs, sacrificing lower manufacturing-costs for higher performance in some applications and systems. Multi-core architectures are being developed, but so are the alternatives. An especially strong contender for established markets is the further integration of peripheral functions into the chip.

## Advantages

The proximity of multiple CPU cores on the same die allows the cache coherency circuitry to operate at a much higher clock-rate than is possible if the signals have to travel off-chip. Combining equivalent CPUs on a single die significantly improves the performance of cache snoop (alternative: Bus snooping) operations. Put simply, this means that signals between different CPUs travel shorter distances, and therefore those signals degrade less. These higher-quality signals allow more data to be sent in a given time period, since individual signals can be shorter and do not need to be repeated as often.

Assuming that the die can physically fit into the package, multi-core CPU designs require much less printed circuit board (PCB) space than do multi-chip SMP designs. Also, a dual-core processor uses slightly less power than two coupled single-core processors, principally because of the decreased power required to drive signals external to the chip. Furthermore, the cores share some circuitry, like the L2 cache and the interface to the front side bus (FSB). In terms of competing technologies for the available silicon die area, multi-core design can make use of proven CPU core library designs and produce a product with lower risk of design error than devising a new wider core-design. Also, adding more cache suffers from diminishing returns.Wikipedia:Citation needed

Multi-core chips also allow higher performance at lower energy. This can be a big factor in mobile devices that operate on batteries. Since each and every core in multi-core is generally more energy-efficient, the chip becomes more efficient than having a single large monolithic core. This allows higher performance with less energy. The challenge of writing parallel code clearly offsets this benefit.

### Disadvantages

Maximizing the utilization of the computing resources provided by multi-core processors requires adjustments both to the operating system (OS) support and to existing application software. Also, the ability of multi-core processors to increase application performance depends on the use of multiple threads within applications. The situation is improving: for example the Valve Corporation's Source engine offers multi-core support,[3][4] and Crytek has developed similar technologies for CryEngine 2, which powers their game, *Crysis*. Emergent Game Technologies' Gamebryo engine includes their Floodgate technology,[5] which simplifies multi-core development across game platforms. In addition, Apple Inc.'s OS X, starting with Mac OS X Snow Leopard, and iOS starting with iOS 4, have a built-in multi-core facility called Grand Central Dispatch.

Integration of a multi-core chip drives chip production yields down and they are more difficult to manage thermally than lower-density single-chip designs. Intel has partially countered this first problem by creating its quad-core designs by combining two dual-core on a single die with a unified cache, hence any two working dual-core dies can be used, as opposed to producing four cores on a single die and requiring all four to work to produce a quad-core. From an architectural point of view, ultimately, single CPU designs may make better use of the silicon surface area than multiprocessing cores, so a development commitment to this architecture may carry the risk of obsolescence. Finally, raw processing power is not the only constraint on system performance. Two processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage. It has been claimedWikipedia:Manual of Style/Words to watch#Unsupported attributions that if a single core is close to being memory-bandwidth limited, then going to dual-core might give 30% to 70% improvement; if memory bandwidth is not a problem, then a 90% improvement can be expected; however, Amdahl's law makes this claim dubious. It would be possible for an application that used two CPUs to end up running faster on one dual-core if communication between the CPUs was the limiting factor, which would count as more than 100% improvement.

## Hardware

### Trends

The general trend in processor development has moved from dual-, tri-, quad-, hex-, oct-core chips to ones with tens or even hundreds of cores.Wikipedia:Citation needed In addition, multi-core chips mixed with simultaneous multithreading, memory-on-chip, and special-purpose "heterogeneous" cores promise further performance and efficiency gains, especially in processing multimedia, recognition and networking applications. There is also a trend of improving energy-efficiency by focusing on performance-per-watt with advanced fine-grain or ultra fine-grain power management and dynamic voltage and frequency scaling (i.e. laptop computers and portable media players).

### Architecture

The composition and balance of the cores in multi-core architecture show great variety. Some architectures use one core design repeated consistently ("homogeneous"), while others use a mixture of different cores, each optimized for a different, "heterogeneous" role.

The article "CPU designers debate multi-core future" by Rick Merritt, EE Times 2008, includes these comments:

> Chuck Moore [...] suggested computers should be more like cellphones, using a variety of specialty cores to run modular software scheduled by a high-level applications programming interface.
>
> [...] Atsushi Hasegawa, a senior chief engineer at Renesas, generally agreed. He suggested the cellphone's use of many specialty cores working in concert is a good model for future multi-core designs.
>
> [...] Anant Agarwal, founder and chief executive of startup Tilera, took the opposing view. He said multi-core chips need to be homogeneous collections of general-purpose cores to keep the software model simple.

## Software impact

An outdated version of an anti-virus application may create a new thread for a scan process, while its GUI thread waits for commands from the user (e.g. cancel the scan). In such cases, a multi-core architecture is of little benefit for the application itself due to the single thread doing all the heavy lifting and the inability to balance the work evenly across multiple cores. Programming truly multithreaded code often requires complex co-ordination of threads and can easily introduce subtle and difficult-to-find bugs due to the interweaving of processing on data shared between threads (thread-safety). Consequently, such code is much more difficult to debug than single-threaded code when it breaks. There has been a perceived lack of motivation for writing consumer-level threaded applications because of the relative rarity of consumer-level demand for maximum use of computer hardware. Although threaded applications incur little additional performance penalty on single-processor machines, the extra overhead of development has been difficult to justify due to the preponderance of single-processor machines. Also, serial tasks like decoding the entropy encoding algorithms used in video codecs are impossible to parallelize because each result generated is used to help create the next result of the entropy decoding algorithm.

Given the increasing emphasis on multi-core chip design, stemming from the grave thermal and power consumption problems posed by any further significant increase in processor clock speeds, the extent to which software can be multithreaded to take advantage of these new chips is likely to be the single greatest constraint on computer performance in the future. If developers are unable to design software to fully exploit the resources provided by multiple cores, then they will ultimately reach an insurmountable performance ceiling.

The telecommunications market had been one of the first that needed a new design of parallel datapath packet processing because there was a very quick adoption of these multiple-core processors for the datapath and the control plane. These MPUs are going to replace[6] the traditional Network Processors that were based on proprietary micro- or pico-code.

Parallel programming techniques can benefit from multiple cores directly. Some existing parallel programming models such as Cilk Plus, OpenMP, OpenHMPP, FastFlow, Skandium, MPI, and Erlang can be used on multi-core platforms. Intel introduced a new abstraction for C++ parallelism called TBB. Other research efforts include the Codeplay Sieve System, Cray's Chapel, Sun's Fortress, and IBM's X10.

Multi-core processing has also affected the ability of modern computational software development. Developers programming in newer languages might find that their modern languages do not support multi-core functionality. This then requires the use of numerical libraries to access code written in languages like C and Fortran, which perform math computations faster than newer languages like C#. Intel's MKL and AMD's ACML are written in these native languages and take advantage of multi-core processing. Balancing the application workload across processors can be problematic, especially if they have different performance characteristics. There are different conceptual models to deal with the problem, for example using a coordination language and program building blocks (programming libraries or higher-order functions). Each block can have a different native implementation for each processor type. Users simply program using these abstractions and an intelligent compiler chooses the best implementation based on the context.

Managing concurrency acquires a central role in developing parallel applications. The basic steps in designing parallel applications are:

Partitioning

> The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.

Communication

> The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another

task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the communication phase of a design.

Agglomeration

In the third stage, development moves from the abstract toward the concrete. Developers revisit decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer. In particular, developers consider whether it is useful to combine, or agglomerate, tasks identified by the partitioning phase, so as to provide a smaller number of tasks, each of greater size. They also determine whether it is worthwhile to replicate data and computation.

Mapping

In the fourth and final stage of the design of parallel algorithms, the developers specify where each task is to execute. This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling.

On the other hand, on the server side, multi-core processors are ideal because they allow many users to connect to a site simultaneously and have independent threads of execution. This allows for Web servers and application servers that have much better throughput.

### Licensing

Some software may be licensed "per processor". This gives rise to an ambiguity, because a "processor" may be either a core or a combination of cores. Microsoft has stated that it would treat a socket as a single processor.[7] Oracle counts an AMD X2 or Intel dual-core CPU as a single processor but has other numbers for other types, especially for processors with more than two cores.Wikipedia:Citation needed

## Embedded applications

Embedded computing operates in an area of processor technology distinct from that of "mainstream" PCs. The same technological drivers towards multi-core apply here too. Indeed, in many cases the application is a "natural" fit for multi-core technologies, if the task can easily be partitioned between the different processors.

In addition, embedded software is typically developed for a specific hardware release, making issues of software portability, legacy code or supporting independent developers less critical than is the case for PC or enterprise computing. As a result, it is easier for developers to adopt new technologies and as a result there is a greater variety of multi-core processing architectures and suppliers.

As of 2010[8], multi-core network processing devices have become mainstream, with companies such as Freescale Semiconductor, Cavium Networks, Wintegra and Broadcom all manufacturing products with eight processors. For the system developer, a key challenge is how to exploit all the cores in these devices to achieve maximum networking performance at the system level, despite the performance limitations inherent in an SMP operating system. To address this issue, companies such as 6WIND provide portable packet processing software designed so that the networking data plane runs in a fast path environment outside the OS, while retaining full compatibility with standard OS APIs.[9]

In digital signal processing the same trend applies: Texas Instruments has the three-core TMS320C6488 and four-core TMS320C5441, Freescale the four-core MSC8144 and six-core MSC8156 (and both have stated they are working on eight-core successors). Newer entries include the Storm-1 family from Stream Processors, Inc [10] with 40 and 80 general purpose ALUs per chip, all programmable in C as a SIMD engine and Picochip with three-hundred processors on a single die, focused on communication applications.

## Hardware examples

### Commercial

- Adapteva Epiphany, a many-core processor architecture with up to 4096 processors on-chip
- Aeroflex Gaisler LEON3, a multi-core SPARC that also exists in a fault-tolerant version.
- Ageia PhysX, a multi-core physics processing unit.
- Ambric Am2045, a 336-core Massively Parallel Processor Array (MPPA)
- AMD
  - A-Series, dual-, triple-, and quad-core of Accelerated Processor Units (APU).
  - Athlon 64, Athlon 64 FX and Athlon 64 X2 family, dual-core desktop processors.
  - Athlon II, dual-, triple-, and quad-core desktop processors.
  - FX-Series, quad-, 6-, and 8-core desktop processors.
  - Opteron, dual-, quad-, 6-, 8-, 12-, and 16-core server/workstation processors.
  - Phenom, dual-, triple-, and quad-core processors.
  - Phenom II, dual-, triple-, quad-, and 6-core desktop processors.
  - Sempron X2, dual-core entry level processors.
  - Turion 64 X2, dual-core laptop processors.
  - Radeon and FireStream multi-core GPU/GPGPU (10 cores, 16 5-issue wide superscalar stream processors per core)
- Analog Devices Blackfin BF561, a symmetrical dual-core processor
- ARM MPCore is a fully synthesizable multi-core container for ARM11 MPCore and ARM Cortex-A9 MPCore processor cores, intended for high-performance embedded and entertainment applications.
- ASOCS ModemX, up to 128 cores, wireless applications.
- Azul Systems
  - Vega 1, a 24-core processor, released in 2005.
  - Vega 2, a 48-core processor, released in 2006.
  - Vega 3, a 54-core processor, released in 2008.
- Broadcom SiByte SB1250, SB1255 and SB1455.
- ClearSpeed
  - CSX700, 192-core processor, released in 2008 (32/64-bit floating point; Integer ALU)
- Cradle Technologies CT3400 and CT3600, both multi-core DSPs.
- Cavium Networks Octeon, a 16-core MIPS MPU.
- Freescale Semiconductor QorIQ series processors, up to 8 cores, Power Architecture MPU.
- Hewlett-Packard PA-8800 and PA-8900, dual core PA-RISC processors.
- IBM
  - POWER4, the world's first non-embedded dual-core processor, released in 2001.
  - POWER5, a dual-core processor, released in 2004.
  - POWER6, a dual-core processor, released in 2007.
  - POWER7, a 4,6,8-core processor, released in 2010.
  - POWER8, a 12-core processor, released in 2013.
  - PowerPC 970MP, a dual-core processor, used in the Apple Power Mac G5.
  - Xenon, a triple-core, SMT-capable, PowerPC microprocessor used in the Microsoft Xbox 360 game console.
- Kalray
  - MPPA-256, 256-core processor, released 2012 (256 usable VLIW cores, Network-on-Chip (NoC), 32/64-bit IEEE 754 compliant FPU)

- Sony/IBM/Toshiba's Cell processor, a nine-core processor with one general purpose PowerPC core and eight specialized SPUs (Synergystic Processing Unit) optimized for vector operations used in the Sony PlayStation 3
- Infineon Danube, a dual-core, MIPS-based, home gateway processor.
- Intel
  - Atom, single and dual-core processors for netbook systems.
  - Celeron Dual-Core, the first dual-core processor for the budget/entry-level market.
  - Core Duo, a dual-core processor.
  - Core 2 Duo, a dual-core processor.
  - Core 2 Quad, 2 dual-core dies packaged in a multi-chip module.
  - Core i3, Core i5 and Core i7, a family of multi-core processors, the successor of the Core 2 Duo and the Core 2 Quad.
  - Itanium 2, a dual-core processor.
  - Pentium D, 2 single-core dies packaged in a multi-chip module.
  - Pentium Extreme Edition, 2 single-core dies packaged in a multi-chip module.
  - Pentium Dual-Core, a dual-core processor.
  - Teraflops Research Chip (Polaris), a 3.16 GHz, 80-core processor prototype, which the company originally stated would be released by 2011.[11]
  - Xeon dual-, quad-, 6-, 8-, and 10-core processors.
  - Xeon Phi 57-core, 60-core and 61-core processors.
- IntellaSys
  - SEAforth 40C18, a 40-core processor[12]
  - SEAforth24, a 24-core processor designed by Charles H. Moore
- NetLogic Microsystems
  - XLP, a 32-core, quad-threaded MIPS64 processor
  - XLR, an eight-core, quad-threaded MIPS64 processor
  - XLS, an eight-core, quad-threaded MIPS64 processor
- Nvidia
  - GeForce 9 multi-core GPU (8 cores, 16 scalar stream processors per core)
  - GeForce 200 multi-core GPU (10 cores, 24 scalar stream processors per core)
  - Tesla multi-core GPGPU (10 cores, 24 scalar stream processors per core)
- Parallax Propeller P8X32, an eight-core microcontroller.
- picoChip PC200 series 200–300 cores per device for DSP & wireless
- Plurality HAL series tightly coupled 16-256 cores, L1 shared memory, hardware synchronized processor.
- Rapport Kilocore KC256, a 257-core microcontroller with a PowerPC core and 256 8-bit "processing elements".
- SiCortex "SiCortex node" has six MIPS64 cores on a single chip.
- Sun Microsystems
  - MAJC 5200, two-core VLIW processor
  - UltraSPARC IV and UltraSPARC IV+, dual-core processors.
  - UltraSPARC T1, an eight-core, 32-thread processor.
  - UltraSPARC T2, an eight-core, 64-concurrent-thread processor.
  - UltraSPARC T3, a sixteen-core, 128-concurrent-thread processor.
  - SPARC T4, an eight-core, 64-concurrent-thread processor.
  - SPARC T5, a sixteen-core, 128-concurrent-thread processor.
- Texas Instruments
  - TMS320C80 MVP, a five-core multimedia video processor.
  - TMS320TMS320C66, 2,4,8 core dsp.

- Tilera
  - TILE64, a 64-core 32-bit processor
  - TILE-Gx, a 72-core 64-bit processor
- XMOS Software Defined Silicon quad-core XS1-G4

### Free

- OpenSPARC

### Academic

- MIT, 16-core RAW [13] processor
- University of California, Davis, Asynchronous array of simple processors (AsAP)
  - 36-core 610 MHz AsAP
  - 167-core 1.2 GHz AsAP2
- University of Washington, Wavescalar [14] processor
- University of Texas, Austin, TRIPS processor
- Linköping University, Sweden, ePUMA processor

## Notes

1. ^ Digital signal processors (DSPs) have utilized multi-core architectures for much longer than high-end general-purpose processors. A typical example of a DSP-specific implementation would be a combination of a RISC CPU and a DSP MPU. This allows for the design of products that require a general-purpose processor for user interfaces and a DSP for real-time data processing; this type of design is common in mobile phones. In other applications, a growing number of companies have developed multi-core DSPs with very large numbers of processors.
2. ^ Two types of operating systems are able to utilize a dual-CPU multiprocessor: partitioned multiprocessing and symmetric multiprocessing (SMP). In a partitioned architecture, each CPU boots into separate segments of physical memory and operate independently; in an SMP OS, processors work in a shared space, executing threads within the OS independently.

## References

[1] CSA Organization (http://www.csa.com/discoveryguides/multicore/review.pdf)
[2] Programming Many-Core Chips. By András Vajda (http://books.google.com/books?id=pSxa_anfiG0C&pg=PA3&dq=several+tens), page 3
[3] Multi-core in the Source Engine (http://www.bit-tech.net/gaming/2006/11/02/Multi_core_in_the_Source_Engin/1.html)
[4] AMD: dual-core not for gamers... yet (http://www.theregister.co.uk/2005/04/22/amd_dual-core_games/)
[5] Gamebryo's Floodgate page (http://www.emergent.net/index.php/homepage/products-and-services/floodgate)
[6] Multicore packet processing Forum (http://multicorepacketprocessing.com/)
[7] Multicore Processor Licensing (http://www.microsoft.com/licensing/highlights/multicore.mspx)
[8] http://en.wikipedia.org/w/index.php?title=Multi-core_processor&action=edit
[9] Maximizing network stack performance (http://www.6wind.com/products-services/6windgate-software)
[10] http://www.streamprocessors.com
[11] 80-core prototype from Intel (http://techfreep.com/intel-80-cores-by-2011.htm)
[12] "40-core processor with Forth-based IDE tools unveiled" (http://www.embedded.com/products/integratedcircuits/210603674)
[13] http://groups.csail.mit.edu/cag/raw/
[14] http://wavescalar.cs.washington.edu/

## External links

- Embedded moves to multicore (http://embedded-computing.com/embedded-moves-multicore)
- Multicore News blog (http://www.multicorezone.com)
- IEEE: Multicore Is Bad News For Supercomputers (http://spectrum.ieee.org/nov08/6912)

# Symmetric multiprocessing

**Symmetric multiprocessing** (**SMP**) involves a symmetric multiprocessor system hardware and software architecture where two or more identical processors connect to a single, shared main memory, have full access to all I/O devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. Most multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

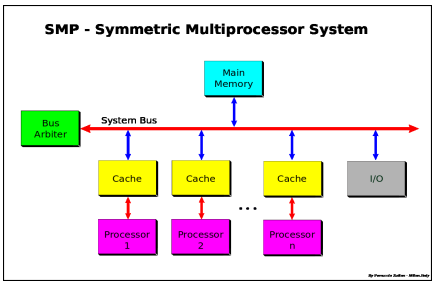


Diagram of a symmetric multiprocessing system

SMP systems are *tightly coupled multiprocessor systems* with a pool of homogeneous processors running independently, each processor executing different programs and working on different data and with capability of sharing common resources (memory, I/O device, interrupt system and so on) and connected using a system bus or a crossbar.

## Design

SMP systems have centralized shared memory called *Main Memory* (MM) operating under a single operating system with two or more homogeneous processors. Usually each processor has an associated private high-speed memory known as cache memory (or cache) to speed-up the MM data access and to reduce the system bus traffic.

Processors may be interconnected using buses, crossbar switches or on-chip mesh networks. The bottleneck in the scalability of SMP using buses or crossbar switches is the bandwidth and power consumption of the interconnect among the various processors, the memory, and the disk arrays. Mesh architectures avoid these bottlenecks, and provide nearly linear scalability to much higher processor counts at the sacrifice of programmability:

> Serious programming challenges remain with this kind of architecture because it requires two distinct modes of programming, one for the CPUs themselves and one for the interconnect between the CPUs. A single programming language would have to be able to not only partition the workload, but also comprehend the memory locality, which is severe in a mesh-based architecture.

SMP systems allow any processor to work on any task no matter where the data for that task are located in memory, provided that each task in the system is not in execution on two or more processors at the same time; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.

## History

IBM offered dual-processor computer systems based on its System/360 model 65 and the closely related model 67. and 67-2 The operating systems that ran on these machines were OS/360 M65MP[1] and TSS/360. Other software, developed at universities, used both CPUs—notably MTS. Both processors could access data channels and initiate I/O.

In OS/360 M65MP, since the operating system kernel ran on both processors (though with a "big lock" around the I/O handler) and peripherals could generally be attached to either processor.[2]

The MTS supervisor (UMMPS) ran on either or both CPUs of the IBM System/360 model 67-2. Supervisor locks were small and were used to protect individual common data structures that might be accessed simultaneously from either CPU.[3]

Digital Equipment Corporation's first multi-processor VAX system, the VAX-11/782, was asymmetric,[4] but later VAX multiprocessor systems were SMP.[5]

The first commercial Unix SMP implementation was the NUMA based Honeywell Information Systems Italy XPS-100 designed by Dan Gielan of VAST Corporation in 1985. Its design supported up to 14 processors although due to electrical limitations the largest marketed version was a dual processor system. The operating system was derived and ported by VAST Corporation from AT&T 3B20 Unix SysVr3 code used internally within AT&T.

## Uses

Time-sharing and server systems can often use SMP without changes to applications, as they may have multiple processes running in parallel, and a system with more than one process running can run different processes on different processors.

On personal computers, SMP is less useful for applications that have not been modified. If the system rarely runs more than one process at a time, SMP is useful only for applications that have been modified for multithreaded (multitasked) processing. Custom-programmed software can be written or modified to use multiple threads, so that it can make use of multiple processors. However, most consumer products such as word processors and computer games are written in such a manner that they cannot gain large benefits from concurrent systems. For games this is usually because writing a program to increase performance on SMP systems can produce a performance loss on uniprocessor systems. Recently[6], however, multi-core chips are becoming more common in new computers, and the balance between installed uni- and multi-core computers may change in the coming years.

Multithreaded programs can also be used in time-sharing and server systems that support multithreading, allowing them to make more use of multiple processors.

## Programming

Uniprocessor and SMP systems require different programming methods to achieve maximum performance. Programs running on SMP systems may experience a performance increase even when they have been written for uniprocessor systems. This is because hardware interrupts that usually suspend program execution while the kernel handles them can execute on an idle processor instead. The effect in most applications (e.g. games) is not so much a performance increase as the appearance that the program is running much more smoothly. Some applications, particularly compilers and some distributed computing projects, run faster by a factor of (nearly) the number of additional processorsWikipedia:Citation needed.

Systems programmers must build support for SMP into the operating system: otherwise, the additional processors remain idle and the system functions as a uniprocessor system.

SMP systems can also lead to more complexity regarding instruction sets. A homogeneous processor system typically requires extra registers for "special instructions" such as SIMD (MMX, SSE, etc.), while a heterogeneous

system can implement different types of hardware for different instructions/uses.

## Performance

When more than one program executes at the same time, an SMP system has considerably better performance than a uni-processor, because different programs can run on different CPUs simultaneously.

In cases where an SMP environment processes many jobs, administrators often experience a loss of hardware efficiency. Software programs have been developed to schedule jobs so that the processor utilization reaches its maximum potential. Good software packages can achieve this maximum potential by scheduling each CPU separately, as well as being able to integrate multiple SMP machines and clusters.

Access to RAM is serialized; this and cache coherency issues causes performance to lag slightly behind the number of additional processors in the system.

## Systems

### Entry-level systems

Before about 2006, entry-level servers and workstations with two processors dominated the SMP market. With the introduction of dual-core devices, SMP is found in most new desktop machines and in many laptop machines. The most popular entry-level SMP systems use the x86 instruction set architecture and are based on Intel's Xeon, Pentium D, Core Duo, and Core 2 Duo based processors or AMD's Athlon64 X2, Quad FX or Opteron 200 and 2000 series processors. Servers use those processors and other readily available non-x86 processor choices, including the Sun Microsystems UltraSPARC, Fujitsu SPARC64 III and later, SGI MIPS, Intel Itanium, Hewlett Packard PA-RISC, Hewlett-Packard (merged with Compaq, which acquired first Digital Equipment Corporation) DEC Alpha, IBM POWER and PowerPC (specifically G4 and G5 series, as well as earlier PowerPC 604 and 604e series) processors. In all cases, these systems are available in uniprocessor versions as well.

Earlier SMP systems used motherboards that have two or more CPU sockets. More recently[6], microprocessor manufacturers introduced CPU devices with two or more processors in one device, for example, the Itanium, POWER, UltraSPARC, Opteron, Athlon, Core 2, and Xeon all have multi-core variants. Athlon and Core 2 Duo multiprocessors are socket-compatible with uniprocessor variants, so an expensive dual socket motherboard is no longer needed to implement an entry-level SMP machine. It should also be noted that dual socket Opteron designs are technically ccNUMA designs, though they can be programmed as SMP for a slight loss in performance. Software based SMP systems can be created by linking smaller systems together. An example of this is the software developed by ScaleMP.

With the introduction of ARM Cortex-A9 multi-core SoCs, low-cost symmetric multiprocessing embedded systems began to flourish in the form of smartphones and tablet computers with a multi-core processor.

### Mid-level systems

The Burroughs D825 first implemented SMP in 1962. It was implemented later on other mainframes. Mid-level servers, using between four and eight processors, can be found using the Intel Xeon MP, AMD Opteron 800 and 8000 series and the above-mentioned UltraSPARC, SPARC64, MIPS, Itanium, PA-RISC, Alpha and POWER processors. High-end systems, with sixteen or more processors, are also available with all of the above processors.

Sequent Computer Systems built large SMP machines using Intel 80386 (and later 80486) processors. Some smaller 80486 systems existed, but the major x86 SMP market began with the Intel Pentium technology supporting up to two processors. The Intel Pentium Pro expanded SMP support with up to four processors natively. Later, the Intel Pentium II, and Intel Pentium III processors allowed dual CPU systems, except for the respective Celerons. This was followed by the Intel Pentium II Xeon and Intel Pentium III Xeon processors, which could be used with up to four

processors in a system natively. In 2001 AMD released their Athlon MP, or MultiProcessor CPU, together with the 760MP motherboard chipset as their first offering in the dual processor marketplace. Although several much larger systems were built, they were all limited by the physical memory addressing limitation of 64 GiB. With the introduction of 64-bit memory addressing on the AMD64 Opteron in 2003 and Intel 64 (EM64T) Xeon in 2005, systems are able to address much larger amounts of memory; their addressable limitation of 16 EiB is not expected to be reached in the foreseeable future.

## Alternatives

SMP using a single shared system bus represents one of the earliest styles of multiprocessor machine architectures, typically used for building smaller computers with up to 8 processors.

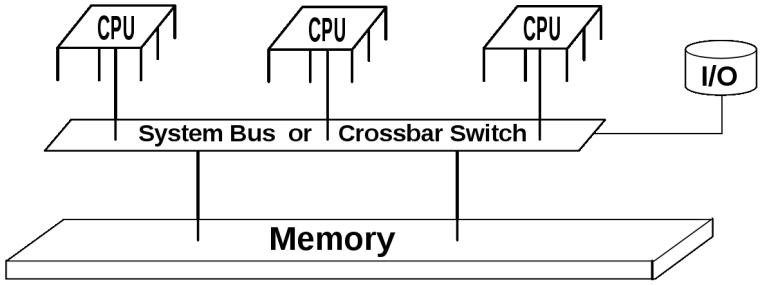



Diagram of a typical SMP system. Three processors are connected to the same memory module through a system bus or crossbar switch

Larger computer systems might use newer architectures such as NUMA (Non-Uniform Memory Access), which dedicates different memory banks to different processors. In a NUMA architecture, processors may access local memory quickly and remote memory more slowly. This can dramatically improve memory throughput as long as the data are localized to specific processes (and thus processors). On the downside, NUMA makes the cost of moving data from one processor to another, as in workload balancing, more expensive. The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

Finally, there is computer clustered multiprocessing (such as Beowulf), in which not all memory is available to all processors. Clustering techniques are used fairly extensively to build very large supercomputers.

## References

[1] M65MP: An Experiment in OS/360 multiprocessing (http://doi.acm.org/10.1145/800186.810634)

[2] IBM, "OS I/O Supervisor PLM" (http://bitsavers.org/pdf/ibm/360/os/R21.7_Apr73/plm/GY28-6616-9_OS_IO_Superv_PLM_R21.7_Apr73.pdf) - GY28-6616-9, Program Logic Manual, R21.7, April 1973

[3] *Time Sharing Supervisor Programs* (http://archive.michigan-terminal-system.org/documents/timeSharingSupervisorPrograms-1971.pdf?attredirects=0&d=1) by Mike Alexander (May 1971) has information on MTS, TSS, CP/67, and Multics

[4] VAX Product Sales Guide, pages 1-23 and 1-24 (http://www.bitsavers.org/pdf/dec/vax/EG-21731-18_VAX_Product_Sales_Guide_Apr82.pdf): the VAX-11/782 is described as an asymmetric multiprocessing system in 1982

[5] VAX 8820/8830/8840 System Hardware User's Guide (http://www.bitsavers.org/pdf/dec/vax/8800/EK-8840H-UG-001_88xx_System_Hardware_Users_Guide_Mar88.pdf): by 1988 the VAX operating system was SMP

[6] http://en.wikipedia.org/w/index.php?title=Symmetric_multiprocessing&action=edit

## External links

- History of Multi-Processing (http://ei.cs.vt.edu/~history/Parallel.html)
- Practical Parallel Programming in Pascal (http://www.cs.bris.ac.uk/~alan/book.html)
- Linux and Multiprocessing (http://www.ibm.com/developerworks/library/l-linux-smp/)
- Multicore News blog (http://www.multicorezone.com)
- AMD (http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118,00.html)

# Distributed computing

"Distributed Information Processing" redirects here. For the computer company, see DIP Research.

**Distributed computing** is a field of computer science that studies distributed systems. A *distributed system* is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components. Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications.

A computer program that runs in a distributed system is called a **distributed program**, and distributed programming is the process of writing such programs.[1] There are many alternatives for the message passing mechanism, including RPC-like connectors and message queues. An important goal and challenge of distributed systems is location transparency.

**Distributed computing** also refers to the use of distributed systems to solve computational problems. In *distributed computing*, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other by message passing.[2]

## Introduction

The word *distributed* in terms such as "distributed system", "distributed programming", and "distributed algorithm" originally referred to computer networks where individual computers were physically distributed within some geographical area.[3] The terms are nowadays used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by message passing. While there is no single definition of a distributed system,[4] the following defining properties are commonly used:

- There are several autonomous computational entities, each of which has its own local memory.[5]
- The entities communicate with each other by message passing.[6]

In this article, the computational entities are called *computers* or *nodes*.

A distributed system may have a common goal, such as solving a large computational problem.[7] Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users.[8]

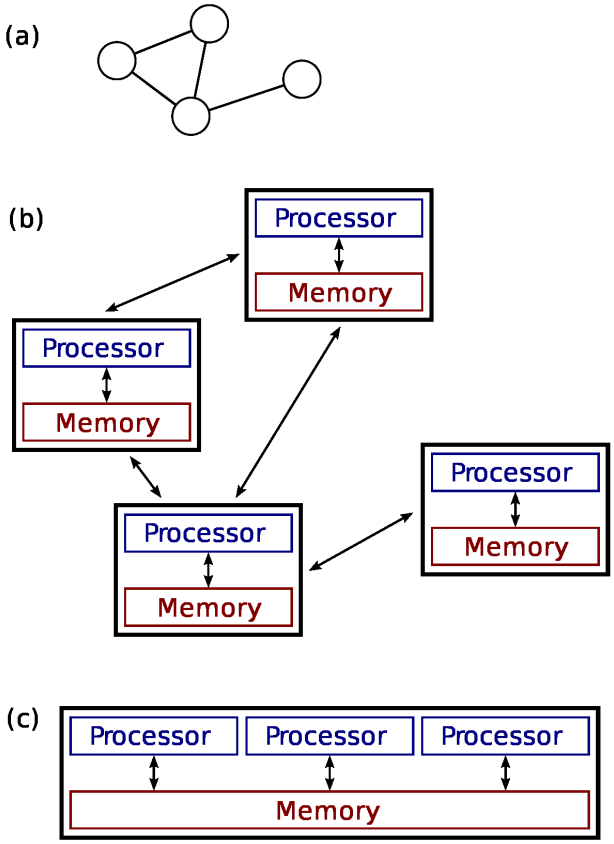
Other typical properties of distributed systems include the following:

- The system has to tolerate failures in individual computers.[9]
- The structure of the system (network topology, network latency, number of computers) is not known in advance, the system may consist of different kinds of computers and network links, and the system may change during the execution of a distributed program.[10]
- Each computer has only a limited, incomplete view of the system. Each computer may know only one part of the input.[11]

## Architecture

Client/Server System : The Client-server architecture is a way to dispense a service from a central source. There is a single server that provides a service, and many clients that communicate with the server to consume its products. In this architecture, clients and servers have different jobs. The server's job is to respond to service requests from clients, while a client's job is to use the data provided in response in order to perform some task.

Peer-to-Peer System : The term peer-to-peer is used to describe distributed systems in which labor is divided among all the components of the system. All the computers send and receive data, and they all contribute some processing power and memory. As a distributed system increases in size, its capacity of computational resources increases. In a peer-to-peer system, all components of the system contribute some processing power and memory to a distributed computation.




(a)–(b) A distributed system.
(c) A parallel system.

## Parallel and distributed computing

Distributed systems are groups of networked computers, which have the same goal for their work. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them.[12] The same system may be characterised both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel.[13] Parallel computing may be seen as a particular tightly coupled form of distributed computing,[14] and distributed computing may be seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange information between processors.[15]
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.[16]

The figure on the right illustrates the difference between distributed and parallel systems. Figure (a) is a schematic view of a typical distributed system; as usual, the system is represented as a network topology in which each node is a computer and each line connecting the nodes is a communication link. Figure (b) shows the same distributed system in more detail: each computer has its own local memory, and information can be exchanged only by passing messages from one node to another by using the available communication links. Figure (c) shows a parallel system in which each processor has a direct access to a shared memory.

The situation is further complicated by the traditional uses of the terms parallel and distributed *algorithm* that do not quite match the above definitions of parallel and distributed *systems*; see the section Theoretical foundations below for more detailed discussion. Nevertheless, as a rule of thumb, high-performance parallel computation in a shared-memory multiprocessor uses parallel algorithms while the coordination of a large-scale distributed system

uses distributed algorithms.

## History

During the earliest years of computing, any tasks that required large computations and massive processings were left up the government or a handful of large companies. These entities could afford to buy massive supercomputers and the infrastructure needed to support them.

As the price of personal computing declined rapidly and the Internet was introduced, an alternative was needed. Distributed Computing gave the answer.

The use of concurrent processes that communicate by message-passing has its roots in operating system architectures studied in the 1960s.[17] The first widespread distributed systems were local-area networks such as Ethernet, which was invented in the 1970s.[18]

ARPANET, the predecessor of the Internet, was introduced in the late 1960s, and ARPANET e-mail was invented in the early 1970s. E-mail became the most successful application of ARPANET,[19] and it is probably the earliest example of a large-scale distributed application. In addition to ARPANET, and its successor, the Internet, other early worldwide computer networks included Usenet and FidoNet from 1980s, both of which were used to support distributed discussion systems.

The study of distributed computing became its own branch of computer science in the late 1970s and early 1980s. The first conference in the field, Symposium on Principles of Distributed Computing (PODC), dates back to 1982, and its European counterpart International Symposium on Distributed Computing (DISC) was first held in 1985.

## Applications

Reasons for using distributed systems and distributed computing may include:

1. The very nature of an application may *require* the use of a communication network that connects several computers: for example, data produced in one physical location and required in another location.
2. There are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is *beneficial* for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer. A distributed system can provide more reliability than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uniprocessor system.[20]

Ghaemi *et al.* define a **distributed query** as a query "that selects data from databases located at multiple sites in a network" and offer as an SQL example:

```
SELECT ename, dname
FROM company.emp e, company.dept@sales.goods d
WHERE e.deptno = d.deptno
```

## Examples

Examples of distributed systems and applications of distributed computing include the following:[21]

- Telecommunication networks:
  - Telephone networks and cellular networks
  - Computer networks such as the Internet
  - Wireless sensor networks
  - Routing algorithms
- Network applications:
  - World wide web and peer-to-peer networks
  - Massively multiplayer online games and virtual reality communities
  - Distributed databases and distributed database management systems
  - Network file systems
  - Distributed information processing systems such as banking systems and airline reservation systems
- Real-time process control:
  - Aircraft control systems
  - Industrial control systems
- Parallel computation:
  - Scientific computing, including cluster computing and grid computing and various volunteer computing projects; see the list of distributed computing projects
  - Distributed rendering in computer graphics

## Theoretical foundations

Main article: Distributed algorithm

### Models

Many tasks that we would like to automate by using a computer are of question–answer type: we would like to ask a question and the computer should produce an answer. In theoretical computer science, such tasks are called computational problems. Formally, a computational problem consists of *instances* together with a *solution* for each instance. Instances are questions that we can ask, and solutions are desired answers to these questions.

Theoretical computer science seeks to understand which computational problems can be solved by using a computer (computability theory) and how efficiently (computational complexity theory). Traditionally, it is said that a problem can be solved by using a computer if we can design an algorithm that produces a correct solution for any given instance. Such an algorithm can be implemented as a computer program that runs on a general-purpose computer: the program reads a problem instance from input, performs some computation, and produces the solution as output. Formalisms such as random access machines or universal Turing machines can be used as abstract models of a sequential general-purpose computer executing such an algorithm.

The field of concurrent and distributed computing studies similar questions in the case of either multiple computers, or a computer that executes a network of interacting processes: which computational problems can be solved in such a network and how efficiently? However, it is not at all obvious what is meant by “solving a problem” in the case of a concurrent or distributed system: for example, what is the task of the algorithm designer, and what is the concurrent or distributed equivalent of a sequential general-purpose computer?

The discussion below focuses on the case of multiple computers, although many of the issues are the same for concurrent processes running on a single computer.

Three viewpoints are commonly used:

Parallel algorithms in shared-memory model

- All computers have access to a shared memory. The algorithm designer chooses the program executed by each computer.
- One theoretical model is the parallel random access machines (PRAM) that are used.[22] However, the classical PRAM model assumes synchronous access to the shared memory.
- A model that is closer to the behavior of real-world multiprocessor machines and takes into account the use of machine instructions, such as Compare-and-swap (CAS), is that of *asynchronous shared memory*. There is a wide body of work on this model, a summary of which can be found in the literature.[23]

Parallel algorithms in message-passing model

- The algorithm designer chooses the structure of the network, as well as the program executed by each computer.
- Models such as Boolean circuits and sorting networks are used.[24] A Boolean circuit can be seen as a computer network: each gate is a computer that runs an extremely simple computer program. Similarly, a sorting network can be seen as a computer network: each comparator is a computer.

Distributed algorithms in message-passing model

- The algorithm designer only chooses the computer program. All computers run the same program. The system must work correctly regardless of the structure of the network.
- A commonly used model is a graph with one finite-state machine per node.

In the case of distributed algorithms, computational problems are typically related to graphs. Often the graph that describes the structure of the computer network *is* the problem instance. This is illustrated in the following example.

## An example

Consider the computational problem of finding a coloring of a given graph *G*. Different fields might take the following approaches:

Centralized algorithms

- The graph *G* is encoded as a string, and the string is given as input to a computer. The computer program finds a coloring of the graph, encodes the coloring as a string, and outputs the result.

Parallel algorithms

- Again, the graph *G* is encoded as a string. However, multiple computers can access the same string in parallel. Each computer might focus on one part of the graph and produce a coloring for that part.
- The main focus is on high-performance computation that exploits the processing power of multiple computers in parallel.

Distributed algorithms

- The graph *G* is the structure of the computer network. There is one computer for each node of *G* and one communication link for each edge of *G*. Initially, each computer only knows about its immediate neighbors in the graph *G*; the computers must exchange messages with each other to discover more about the structure of *G*. Each computer must produce its own color as output.
- The main focus is on coordinating the operation of an arbitrary distributed system.

While the field of parallel algorithms has a different focus than the field of distributed algorithms, there is a lot of interaction between the two fields. For example, the Cole–Vishkin algorithm for graph coloring[25] was originally presented as a parallel algorithm, but the same technique can also be used directly as a distributed algorithm.

Moreover, a parallel algorithm can be implemented either in a parallel system (using shared memory) or in a distributed system (using message passing).[26] The traditional boundary between parallel and distributed algorithms (choose a suitable network vs. run in any given network) does not lie in the same place as the boundary between parallel and distributed systems (shared memory vs. message passing).

## Complexity measures

In parallel algorithms, yet another resource in addition to time and space is the number of computers. Indeed, often there is a trade-off between the running time and the number of computers: the problem can be solved faster if there are more computers running in parallel (see speedup). If a decision problem can be solved in polylogarithmic time by using a polynomial number of processors, then the problem is said to be in the class NC.[27] The class NC can be defined equally well by using the PRAM formalism or Boolean circuits − PRAM machines can simulate Boolean circuits efficiently and vice versa.[28]

In the analysis of distributed algorithms, more attention is usually paid on communication operations than computational steps. Perhaps the simplest model of distributed computing is a synchronous system where all nodes operate in a lockstep fashion. During each *communication round*, all nodes in parallel (1) receive the latest messages from their neighbours, (2) perform arbitrary local computation, and (3) send new messages to their neighbours. In such systems, a central complexity measure is the number of synchronous communication rounds required to complete the task.[29]

This complexity measure is closely related to the diameter of the network. Let $D$ be the diameter of the network. On the one hand, any computable problem can be solved trivially in a synchronous distributed system in approximately $2D$ communication rounds: simply gather all information in one location ($D$ rounds), solve the problem, and inform each node about the solution ($D$ rounds).

On the other hand, if the running time of the algorithm is much smaller than $D$ communication rounds, then the nodes in the network must produce their output without having the possibility to obtain information about distant parts of the network. In other words, the nodes must make globally consistent decisions based on information that is available in their *local neighbourhood*. Many distributed algorithms are known with the running time much smaller than $D$ rounds, and understanding which problems can be solved by such algorithms is one of the central research questions of the field.[30]

Other commonly used measures are the total number of bits transmitted in the network (cf. communication complexity).

## Other problems

Traditional computational problems take the perspective that we ask a question, a computer (or a distributed system) processes the question for a while, and then produces an answer and stops. However, there are also problems where we do not want the system to ever stop. Examples of such problems include the dining philosophers problem and other similar mutual exclusion problems. In these problems, the distributed system is supposed to continuously coordinate the use of shared resources so that no conflicts or deadlocks occur.

There are also fundamental challenges that are unique to distributed computing. The first example is challenges that are related to *fault-tolerance*. Examples of related problems include consensus problems,[31] Byzantine fault tolerance,[32] and self-stabilisation.[33]

A lot of research is also focused on understanding the *asynchronous* nature of distributed systems:

- Synchronizers can be used to run synchronous algorithms in asynchronous systems.[34]
- Logical clocks provide a causal happened-before ordering of events.[35]
- Clock synchronization algorithms provide globally consistent physical time stamps.[36]

### Properties of distributed systems

So far the focus has been on *designing* a distributed system that solves a given problem. A complementary research problem is *studying* the properties of a given distributed system.

The halting problem is an analogous example from the field of centralised computation: we are given a computer program and the task is to decide whether it halts or runs forever. The halting problem is undecidable in the general case, and naturally understanding the behaviour of a computer network is at least as hard as understanding the behaviour of one computer.

However, there are many interesting special cases that are decidable. In particular, it is possible to reason about the behaviour of a network of finite-state machines. One example is telling whether a given network of interacting (asynchronous and non-deterministic) finite-state machines can reach a deadlock. This problem is PSPACE-complete,[37] i.e., it is decidable, but it is not likely that there is an efficient (centralised, parallel or distributed) algorithm that solves the problem in the case of large networks.

## Coordinator Election

**Coordinator election** (sometimes called **leader election**) is the process of designating a single process as the organizer of some task distributed among several computers (nodes). Before the task is begun, all network nodes are either unaware which node will serve as the "coordinator" (or leader) of the task, or unable to communicate with the current coordinator. After a coordinator election algorithm has been run, however, each node throughout the network recognizes a particular, unique node as the task coordinator.

The network nodes communicate among themselves in order to decide which of them will get into the "coordinator" state. For that, they need some method in order to break the symmetry among them. For example, if each node has unique and comparable identities, then the nodes can compare their identities, and decide that the node with the highest identity is the coordinator.

The definition of this problem is often attributed to LeLann, who formalized it as a method to create a new token in a token ring network in which the token has been lost.

Coordinator election algorithms are designed to be economical in terms of total bytes transmitted, and time. The algorithm suggested by Gallager, Humblet, and Spira for general undirected graphs has had a strong impact on the design of distributed algorithms in general, and won the Dijkstra Prize for an influential paper in distributed computing.

Many other algorithms were suggested for different kind of network graphs, such as undirected rings, unidirectional rings, complete graphs, grids, directed Euler graphs, and others. A general method that decouples the issue of the graph family from the design of the coordinator election algorithm was suggested by Korach, Kutten, and Moran.

In order to perform coordination, distributed systems employ the concept of coordinators. The coordinator election problem is to choose a process from among a group of processes on different processors in a distributed system to act as the central coordinator. Several central coordinator election algorithms exist.

### Bully algorithm

When using the Bully algorithm, any process sends a message to the current coordinator. If there is no response within a given time limit, the process tries to elect itself as leader.

### Chang and Roberts algorithm

The Chang and Roberts algorithm (or "Ring Algorithm") is a ring-based election algorithm used to find a process with the largest unique identification number .

## Architectures

Various hardware and software architectures are used for distributed computing. At a lower level, it is necessary to interconnect multiple CPUs with some sort of network, regardless of whether that network is printed onto a circuit board or made up of loosely coupled devices and cables. At a higher level, it is necessary to interconnect processes running on those CPUs with some sort of communication system.

Distributed programming typically falls into one of several basic architectures or categories: client–server, 3-tier architecture, *n*-tier architecture, distributed objects, loose coupling, or tight coupling.

- Client–server: Smart client code contacts the server for data then formats and displays it to the user. Input at the client is committed back to the server when it represents a permanent change.
- 3-tier architecture: Three tier systems move the client intelligence to a middle tier so that stateless clients can be used. This simplifies application deployment. Most web applications are 3-Tier.
- *n*-tier architecture: *n*-tier refers typically to web applications which further forward their requests to other enterprise services. This type of application is the one most responsible for the success of application servers.
- highly coupled (clustered): refers typically to a cluster of machines that closely work together, running a shared process in parallel. The task is subdivided in parts that are made individually by each one and then put back together to make the final result.
- Peer-to-peer: an architecture where there is no special machine or machines that provide a service or manage the network resources. Instead all responsibilities are uniformly divided among all machines, known as peers. Peers can serve both as clients and servers.
- Space based: refers to an infrastructure that creates the illusion (virtualization) of one single address-space. Data are transparently replicated according to application needs. Decoupling in time, space and reference is achieved.

Another basic aspect of distributed computing architecture is the method of communicating and coordinating work among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master/slave relationship. Alternatively, a "database-centric" architecture can enable distributed computing to be done without any form of direct inter-process communication, by utilizing a shared database.

## Notes

[1] . , p. 10.
[2] , p. 291–292. , p. 5.
[3] , p. 1.
[4] , p. 10.
[5] , p. 8–9, 291. , p. 5. , p. 3. , p. xix, 1. , p. xv.
[6] , p. 291. , p. 3. , p. 4.
[7] , p. 3–4. , p. 1.
[8] , p. 4. , p. 2.
[9] , p. 4, 8. , p. 2–3. , p. 4.
[10] , p. 2. , p. 1.
[11] , p. 7. , p. xix, 2. , p. 4.
[12] , p. 10. .
[13] , p. xix, 1–2. , p. 1.
[14] , p. 1.
[15] , Chapter 15. .
[16] See references in Introduction.
[17] , p. 348.
[18] , p. 32.
[19] , The history of email (http://www.nethistory.info/History of the Internet/email.html).
[20] , Section 24.1.2.
[21] , p. 10–11. , p. 4–6. , p. xix, 1. , p. xv. , Section 24.
[22] , Section 30.
[23] , Chapters 2-6.
[24] , Sections 28 and 29.
[25] . , Section 30.5.
[26] , p. ix.
[27] , Section 6.7. , Section 15.3.
[28] , Section 15.2.
[29] , p. 17–23.
[30] , Sections 2.3 and 7. . .
[31] , Sections 5–7. , Chapter 13.
[32] , p. 99–102. , p. 192–193.
[33] . , Chapter 17.
[34] , Section 16. , Section 6.
[35] , Section 18. , Sections 6.2–6.3.
[36] , Section 6.4.
[37] , Section 19.3.

## References

Books


- Andrews, Gregory R. (2000), *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison–Wesley, ISBN 0-201-35752-6.
- Arora, Sanjeev; Barak, Boaz (2009), *Computational Complexity – A Modern Approach*, Cambridge, ISBN 978-0-521-42426-4.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990), *Introduction to Algorithms* (1st ed.), MIT Press, ISBN 0-262-03141-8.
- Dolev, Shlomi (2000), *Self-Stabilization*, MIT Press, ISBN 0-262-04178-2.
- Elmasri, Ramez; Navathe, Shamkant B. (2000), *Fundamentals of Database Systems* (3rd ed.), Addison–Wesley, ISBN 0-201-54263-3.
- Ghosh, Sukumar (2007), *Distributed Systems – An Algorithmic Approach*, Chapman & Hall/CRC, ISBN 978-1-58488-564-1.
- Lynch, Nancy A. (1996), *Distributed Algorithms*, Morgan Kaufmann, ISBN 1-55860-348-4.

- Herlihy, Maurice P.; Shavit, Nir N. (2008), *The Art of Multiprocessor Programming*, Morgan Kaufmann, ISBN 0-12-370591-6.
- Papadimitriou, Christos H. (1994), *Computational Complexity*, Addison–Wesley, ISBN 0-201-53082-1.
- Peleg, David (2000), *Distributed Computing: A Locality-Sensitive Approach* (http://www.ec-securehost.com/SIAM/DT05.html), SIAM, ISBN 0-89871-464-8.

Articles

- Cole, Richard; Vishkin, Uzi (1986), "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control* **70** (1): 32–53, doi: 10.1016/S0019-9958(86)80023-7 (http://dx.doi.org/10.1016/S0019-9958(86)80023-7).
- Keidar, Idit (2008), "Distributed computing column 32 – The year in review" (http://webee.technion.ac.il/~idish/sigactNews/#column 32), *ACM SIGACT News* **39** (4): 53–54, doi: 10.1145/1466390.1466402 (http://dx.doi.org/10.1145/1466390.1466402).
- Linial, Nathan (1992), "Locality in distributed graph algorithms", *SIAM Journal on Computing* **21** (1): 193–201, doi: 10.1137/0221015 (http://dx.doi.org/10.1137/0221015).
- Naor, Moni; Stockmeyer, Larry (1995), "What can be computed locally?", *SIAM Journal on Computing* **24** (6): 1259–1277, doi: 10.1137/S0097539793254571 (http://dx.doi.org/10.1137/S0097539793254571).

Web sites

- Godfrey, Bill (2002). "A primer on distributed computing" (http://www.bacchae.co.uk/docs/dist.html).
- Peter, Ian (2004). "Ian Peter's History of the Internet" (http://www.nethistory.info/History of the Internet/). Retrieved 2009-08-04.


## Further reading


Books

- Coulouris, George et al (2011), *Distributed Systems: Concepts and Design (5th Edition)*, Addison-Wesley ISBN 0-132-14301-1.
- Attiya, Hagit and Welch, Jennifer (2004), *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, Wiley-Interscience ISBN 0-471-45324-2.
- Faber, Jim (1998), *Java Distributed Computing* (http://docstore.mik.ua/orelly/java-ent/dist/index.htm), O'Reilly: Java Distributed Computing by Jim Faber, 1998 (http://docstore.mik.ua/orelly/java-ent/dist/index.htm)
- Garg, Vijay K. (2002), *Elements of Distributed Computing*, Wiley-IEEE Press ISBN 0-471-03600-5.
- Tel, Gerard (1994), *Introduction to Distributed Algorithms*, Cambridge University Press
- Chandy, Mani et al, *Parallel Program Design*

Articles

- Keidar, Idit; Rajsbaum, Sergio, eds. (2000–2009), "Distributed computing column" (http://webee.technion.ac.il/~idish/sigactNews/), *ACM SIGACT News*.
- Birrell, A. D.; Levin, R.; Schroeder, M. D.; Needham, R. M. (April 1982). "Grapevine: An exercise in distributed computing" (http://www.cs.ucsb.edu/~ravenben/papers/coreos/BLS+82.pdf). *Communications of the ACM* **25** (4): 260–274. doi: 10.1145/358468.358487 (http://dx.doi.org/10.1145/358468.358487).

Conference Papers

- C. Rodríguez, M. Villagra and B. Barán, Asynchronous team algorithms for Boolean Satisfiability (http://dx.doi.org/10.1109/BIMNICS.2007.4610083), Bionetics2007, pp. 66–69, 2007.

## External links

- Distributed computing (http://www.dmoz.org/Computers/Computer_Science/Distributed_Computing/) at DMOZ
- Distributed computing journals (http://www.dmoz.org/Computers/Computer_Science/Distributed_Computing/Publications/) at DMOZ

# Computer cluster

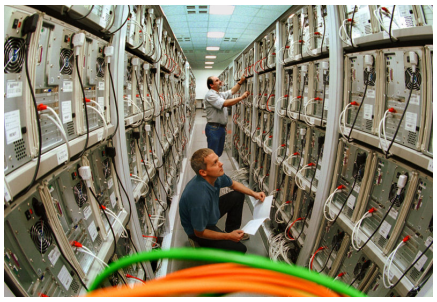
Not to be confused with data cluster or computer lab.

A **computer cluster** consists of a set of loosely connected or tightly connected computers that work together so that in many respects they can be viewed as a single system.

The components of a cluster are usually connected to each other through fast local area networks ("LAN"), with each *node* (computer used as a server) running its own instance of an operating system. Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing.

Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability.

Computer clusters have a wide range of applicability and deployment, ranging from small business clusters with a handful of nodes to some of the fastest supercomputers in the world such as IBM's Sequoia.



Technicians working on a large Linux cluster at the Chemnitz University of Technology, Germany


Sun Microsystems Solaris Cluster

## Basic concepts

A simple, home-built Beowulf cluster.

The desire to get more computing power and better reliability by orchestrating a number of low cost commercial off-the-shelf computers has given rise to a variety of architectures and configurations.

The computer clustering approach usually (but not always) connects a number of readily available computing nodes (e.g. personal computers used as servers) via a fast local area network.[1] The activities of the computing nodes are orchestrated by "clustering middleware", a software layer that sits atop the nodes and allows the users to treat the cluster as by and large one cohesive computing unit, e.g. via a single system image concept.

Computer clustering relies on a centralized management approach which makes the nodes available as orchestrated shared servers. It is distinct from other approaches such as peer to peer or grid computing which also use many nodes, but with a far more distributed nature.

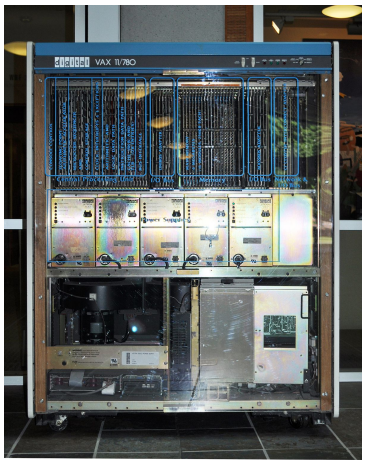
A computer cluster may be a simple two-node system which just connects two personal computers, or may be a very fast supercomputer. A basic approach to building a cluster is that of a Beowulf cluster which may be built with a few personal computers to produce a cost-effective alternative to traditional high performance computing. An early project that showed the viability of the concept was the 133 nodes Stone Soupercomputer. The developers used Linux, the Parallel Virtual Machine toolkit and the Message Passing Interface library to achieve high performance at a relatively low cost.

Although a cluster may consist of just a few personal computers connected by a simple network, the cluster architecture may also be used to achieve very high levels of performance. The TOP500 organization's semiannual list of the 500 fastest supercomputers often includes many clusters, e.g. the world's fastest machine in 2011 was the K computer which has a distributed memory, cluster architecture.[2][3]

## History

Main article: History of computer clusters

See also: History of supercomputing



A VAX 11/780, c. 1977

Greg Pfister has stated that clusters were not invented by any specific vendor but by customers who could not fit all their work on one computer, or needed a backup. Pfister estimates the date as some time in the 1960s. The formal engineering basis of cluster computing as a means of doing parallel work of any sort was arguably invented by Gene Amdahl of IBM, who in 1967 published what has come to be regarded as the seminal paper on parallel processing: Amdahl's Law.

The history of early computer clusters is more or less directly tied into the history of early networks, as one of the primary motivations for the development of a network was to link computing resources, creating a de facto computer cluster.

The first commercial clustering product was ARCnet, developed by Datapoint in 1977. Clustering per se did not really take off until Digital Equipment Corporation released their VAXcluster product in 1984 for the VAX/VMS operating system. The ARCnet and VAXcluster products not only supported parallel computing, but also shared file systems and peripheral devices. The idea was to provide the advantages of parallel processing, while maintaining

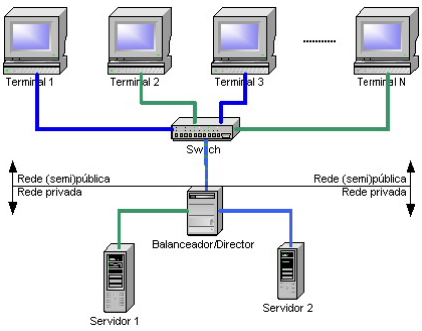
data reliability and uniqueness. Two other noteworthy early commercial clusters were the *Tandem Himalayan* (a circa 1994 high-availability product) and the *IBM S/390 Parallel Sysplex* (also circa 1994, primarily for business use).

Within the same time frame, while computer clusters used parallelism outside the computer on a commodity network, supercomputers began to use them within the same computer. Following the success of the CDC 6600 in 1964, the Cray 1 was delivered in 1976, and introduced internal parallelism via vector processing.[4] While early supercomputers excluded clusters and relied on shared memory, in time some of the fastest supercomputers (e.g. the K computer) relied on cluster architectures.

## Attributes of clusters

Computer clusters may be configured for different purposes ranging from general purpose business needs such as web-service support, to computation-intensive scientific calculations. In either case, the cluster may use a high-availability approach. Note that the attributes described below are not exclusive and a "compute cluster" may also use a high-availability approach, etc.

"Load-balancing" clusters are configurations in which cluster-nodes share computational workload to provide better overall performance. For example, a web server cluster may assign different queries to different nodes, so the overall response time will be optimized.[5] However, approaches to load-balancing may significantly differ among applications, e.g. a high-performance cluster used for scientific computations would balance load with different algorithms from a web-server cluster which may just use a simple round-robin method by assigning each new request to a different node.



A load balancing cluster with two servers and N user stations

"Computer clusters" are used for computation-intensive purposes, rather than handling IO-oriented operations such as web service or databases.[6] For instance, a computer cluster might support computational simulations of vehicle crashes or weather. Very tightly coupled computer clusters are designed for work that may approach "supercomputing".

"High-availability clusters" (also known as failover clusters, or HA clusters) improve the availability of the cluster approach. They operate by having redundant nodes, which are then used to provide service when system components fail. HA cluster implementations attempt to use redundancy of cluster components to eliminate single points of failure. There are commercial implementations of High-Availability clusters for many operating systems. The Linux-HA project is one commonly used free software HA package for the Linux operating system.

## Benefits

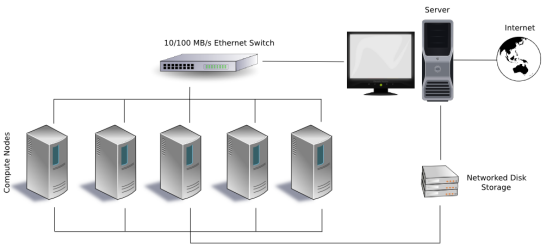
**Low Cost**: Customers can eliminate the cost and complexity of procuring, configuring and operating HPC clusters with low, pay-as-you-go pricing. Further, you can optimize costs by leveraging one of several pricing models: On Demand, Reserved or Spot Instances.

**Elasticity**: You can add and remove computer resources to meet the size and time requirements for your workloads.

**Run Jobs Anytime, Anywhere**: You can launch compute jobs using simple APIs or management tools and automate workflows for maximum efficiency and scalability. You can increase your speed of innovation by accessing computer resources in minutes instead of spending time in queues.

## Design and configuration

One of the issues in designing a cluster is how tightly coupled the individual nodes may be. For instance, a single computer job may require frequent communication among nodes: this implies that the cluster shares a dedicated network, is densely located, and probably has homogeneous nodes. The other extreme is where a computer job uses one or few nodes, and needs little or no inter-node communication, approaching grid computing.




A typical Beowulf configuration

In a Beowulf system, the application programs never see the computational nodes (also called slave computers) but only interact with the "Master" which is a specific computer handling the scheduling and management of the slaves. In a typical implementation the Master has two network interfaces, one that communicates with the private Beowulf network for the slaves, the other for the general purpose network of the organization. The slave computers typically have their own version of the same operating system, and local memory and disk space. However, the private slave network may also have a large and shared file server that stores global persistent data, accessed by the slaves as needed.

By contrast, the special purpose 144 node DEGIMA cluster is tuned to running astrophysical N-body simulations using the Multiple-Walk parallel treecode, rather than general purpose scientific computations.[7]

Due to the increasing computing power of each generation of game consoles, a novel use has emerged where they are repurposed into High-performance computing (HPC) clusters. Some examples of game console clusters are Sony PlayStation clusters and Microsoft Xbox clusters. Another example of consumer game product is the Nvidia Tesla Personal Supercomputer workstation, which uses multiple graphics accelerator processor chips.

Computer clusters have historically run on separate physical computers with the same operating system. With the advent of virtualization, the cluster nodes may run on separate physical computers with different operating systems which are painted above with a virtual layer to look similar.Wikipedia:Citation neededWikipedia:Please clarify The cluster may also be virtualized on various configurations as maintenance takes place. An example implementation is Xen as the virtualization manager with Linux-HA.[8]

# Data sharing and communication

## Data sharing

As the computer clusters were appearing during the 1980s, so were supercomputers. One of the elements that distinguished the three classes at that time was that the early supercomputers relied on shared memory. To date clusters do not typically use physically shared memory, while many supercomputer architectures have also abandoned it.

However, the use of a clustered file system is essential in modern computer clusters.Wikipedia:Citation needed Examples include the IBM General Parallel File System, Microsoft's Cluster Shared Volumes or the Oracle Cluster File System.

A NEC Nehalem cluster

## Message passing and communication

Main article: Message passing in computer clusters

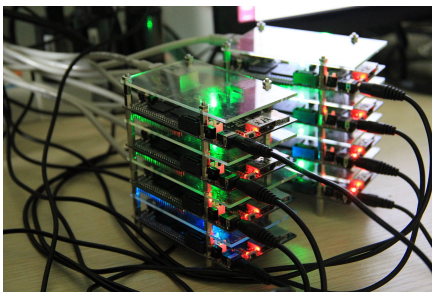
Two widely used approaches for communication between cluster nodes are MPI, the Message Passing Interface and PVM, the Parallel Virtual Machine.[9]

PVM was developed at the Oak Ridge National Laboratory around 1989 before MPI was available. PVM must be directly installed on every cluster node and provides a set of software libraries that paint the node as a "parallel virtual machine". PVM provides a run-time environment for message-passing, task and resource management, and fault notification. PVM can be used by user programs written in C, C++, or Fortran, etc.

MPI emerged in the early 1990s out of discussions among 40 organizations. The initial effort was supported by ARPA and National Science Foundation. Rather than starting anew, the design of MPI drew on various features available in commercial systems of the time. The MPI specifications then gave rise to specific implementations. MPI implementations typically use TCP/IP and socket connections. MPI is now a widely available communications model that enables parallel programs to be written in languages such as C, Fortran, Python, etc.[] Thus, unlike PVM which provides a concrete implementation, MPI is a specification which has been implemented in systems such as MPICH and Open MPI.

# Cluster management

One of the challenges in the use of a computer cluster is the cost of administrating it which can at times be as high as the cost of administrating N independent machines, if the cluster has N nodes.[10] In some cases this provides an advantage to shared memory architectures with lower administration costs. This has also made virtual machines popular, due to the ease of administration.


Low-cost and low energy tiny-cluster of Cubieboards, using Apache Hadoop on Lubuntu

## Task scheduling

When a large multi-user cluster needs to access very large amounts of data, task scheduling becomes a challenge. In a heterogeneous CPU-GPU cluster, which has a complex application environment the performance of each job depends on the

characteristics of the underlying cluster, mapping tasks onto CPU cores and GPU devices provides significant challenges. This is an area of ongoing research and algorithms that combine and extend MapReduce and Hadoop have been proposed and studied.[]

### Node failure management

When a node in a cluster fails, strategies such as "fencing" may be employed to keep the rest of the system operational.[11][12] Fencing is the process of isolating a node or protecting shared resources when a node appears to be malfunctioning. There are two classes of fencing methods; one disables a node itself, and the other disallows access to resources such as shared disks.

The STONITH method stands for "Shoot The Other Node In The Head", meaning that the suspected node is disabled or powered off. For instance, *power fencing* uses a power controller to turn off an inoperable node.

The *resources fencing* approach disallows access to resources without powering off the node. This may include *persistent reservation fencing* via the SCSI3, fibre Channel fencing to disable the fibre channel port, or global network block device (GNBD) fencing to disable access to the GNBD server.

## Software development and administration

### Parallel programming

Load balancing clusters such as web servers use cluster architectures to support a large number of users and typically each user request is routed to a specific node, achieving task parallelism without multi-node cooperation, given that the main goal of the system is providing rapid user access to shared data. However, "computer clusters" which perform complex computations for a small number of users need to take advantage of the parallel processing capabilities of the cluster and partition "the same computation" among several nodes.

Automatic parallelization of programs continues to remain a technical challenge, but parallel programming models can be used to effectuate a higher degree of parallelism via the simultaneous execution of separate portions of a program on different processors.[][13]

### Debugging and monitoring

The development and debugging of parallel programs on a cluster requires parallel language primitives as well as suitable tools such as those discussed by the *High Performance Debugging Forum* (HPDF) which resulted in the HPD specifications.[14] Tools such as TotalView were then developed to debug parallel implementations on computer clusters which use MPI or PVM for message passing.

The Berkeley NOW (Network of Workstations) system gathers cluster data and stores them in a database, while a system such as PARMON, developed in India, allows for the visual observation and management of large clusters.

Application checkpointing can be used to restore a given state of the system when a node fails during a long multi-node computation.[15] This is essential in large clusters, given that as the number of nodes increases, so does the likelihood of node failure under heavy computational loads. Checkpointing can restore the system to a stable state so that processing can resume without having to recompute results.

## Some implementations

The GNU/Linux world supports various cluster software; for application clustering, there is distcc, and MPICH. Linux Virtual Server, Linux-HA - director-based clusters that allow incoming requests for services to be distributed across multiple cluster nodes. MOSIX, openMosix, Kerrighed, OpenSSI are full-blown clusters integrated into the kernel that provide for automatic process migration among homogeneous nodes. OpenSSI, openMosix and Kerrighed are single-system image implementations.

Microsoft Windows Compute Cluster Server 2003 based on the Windows Server platform provides pieces for High Performance Computing like the Job Scheduler, MSMPI library and management tools.

gLite is a set of middleware technologies created by the Enabling Grids for E-sciencE (EGEE) project.

slurm is also used to schedule and manage some of the largest supercomputer clusters (see top500 list.)

## Other approaches

Although most computer clusters are permanent fixtures, attempts at flash mob computing have been made to build short-lived clusters for specific computations. However, larger scale volunteer computing systems such as BOINC-based systems have had more followers.

## References


[1] *Network-Based Information Systems: First International Conference, NBIS 2007* ISBN 3-540-74572-6 page 375

[2] TOP500 list (http://i.top500.org/sublist) To view all clusters on the TOP500 select "cluster" as architecture from the sublist menu.

[3] M. Yokokawa et al *The K Computer*, in "International Symposium on Low Power Electronics and Design" (ISLPED) 1-3 Aug. 2011, pages 371-372

[4] *Readings in computer architecture* by Mark Donald Hill, Norman Paul Jouppi, Gurindar Sohi 1999 ISBN 978-1-55860-539-8 page 41-48

[5] *High Performance Linux Clusters* by Joseph D. Sloan 2004 ISBN 0-596-00570-9 page

[6] *High Performance Computing for Computational Science - VECPAR 2004* by Michel Daydé, Jack Dongarra 2005 ISBN 3-540-25424-2 pages 120-121

[7] Hamada T. *et al.* (2009) A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs – towards cost effective, high performance N-body simulation. *Comput. Sci. Res. Development* 24:21-31.

[8] Maurer, Ryan: Xen Virtualization and Linux Clustering (http://www.linuxjournal.com/article/8812)

[9] *Distributed services with OpenAFS: for enterprise and education* by Franco Milicchio, Wolfgang Alexander Gehrke 2007, ISBN pages 339-341 (http://books.google.it/books?id=bKf4NBaIJI8C&pg=PA339&dq="message+passing"+computer+cluster+MPI+PVM&hl=en&sa=X&ei=-dD7ToZCj_bhBOOxvI0I&redir_esc=y#v=onepage&q="message passing" computer cluster MPI PVM&f=false)

[10] *Computer Organization and Design* by David A. Patterson and John L. Hennessy 2011 ISBN 0-12-374750-3 pages 641-642

[11] Alan Robertson *Resource fencing using STONITH*. IBM Linux Research Center, 2010 (ftp://ftp.telecom.uff.br/pub/linux/HA/ResourceFencing_Stonith.pdf)

[12] *Sun Cluster environment: Sun Cluster 2.2* by Enrique Vargas, Joseph Bianco, David Deeths 2001 ISBN page 58

[13] *Parallel Programming: For Multicore and Cluster Systems* by Thomas Rauber, Gudula Rünger 2010 ISBN 3-642-04817-X pages 94–95 (http://books.google.it/books?id=wWogxOmA3wMC&pg=PA94&dq="parallel+programming+language"+computer+cluster&hl=en&sa=X&ei=zfD8TpX_F5CQ4gSMvfXhBA&redir_esc=y#v=onepage&q="parallel programming language" computer cluster&f=false)

[14] *A debugging standard for high-performance computing* by Joan M. Francioni and Cherri Pancake, in the "Journal of Scientific Programming" Volume 8 Issue 2, April 2000 (http://dl.acm.org/citation.cfm?id=1239906)

[15] *Computational Science-- ICCS 2003: International Conference* edited by Peter Sloot 2003 ISBN 3-540-40195-4 pages 291-292


## Further reading


- Mark Baker, et al., *Cluster Computing White Paper* (http://arxiv.org/abs/cs/0004014), 11 Jan 2001.
- Evan Marcus, Hal Stern: *Blueprints for High Availability: Designing Resilient Distributed Systems*, John Wiley & Sons, ISBN 0-471-35601-8
- Greg Pfister: *In Search of Clusters*, Prentice Hall, ISBN 0-13-899709-8
- Rajkumar Buyya (editor): *High Performance Cluster Computing: Architectures and Systems*, Volume 1, ISBN 0-13-013784-7, and Volume 2, ISBN 0-13-013785-5, Prentice Hall, NJ, USA, 1999.

## External links

- IEEE Technical Committee on Scalable Computing (TCSC) (https://www.ieeetcsc.org/)
- Reliable Scalable Cluster Technology, IBM (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.rsct.doc/rsctbooks.html)
- Tivoli System Automation Wiki (https://www.ibm.com/developerworks/wikis/display/tivoli/Tivoli+System+Automation)

# Massively parallel (computing)

"Massively parallel" redirects here. For other uses, see Massively parallel (disambiguation).

In computing, **massively parallel** refers to the use of a large number of processors (or separate computers) to perform a set of coordinated computations in parallel.

In one approach, e.g., in grid computing the processing power of a large number of computers in distributed, diverse administrative domains, is opportunistically used whenever a computer is available.[1] An example is BOINC, a volunteer-based, opportunistic grid system.[2]

In another approach, a large number of processors are used in close proximity to each other, e.g., in a computer cluster. In such a centralized system the speed and flexibility of the interconnect becomes very important, and modern supercomputers have used various approaches ranging from enhanced Infiniband systems to three-dimensional torus interconnects.[3]

The term also applies to massively parallel processor arrays (MPPAs) a type of integrated circuit with an array of hundreds or thousands of CPUs and RAM banks. These processors pass work to one another through a reconfigurable interconnect of channels. By harnessing a large number of processors working in parallel, an MPPA chip can accomplish more demanding tasks than conventional chips. MPPAs are based on a software parallel programming model for developing high-performance embedded system applications.

Goodyear MPP was an early implementation of a massively parallel computer (MPP architecture). MPP architectures are the second most common supercomputer implementations after clusters, as of November 2013.[4]

## References

[1] *Grid computing: experiment management, tool integration, and scientific workflows* by Radu Prodan, Thomas Fahringer 2007 ISBN 3-540-69261-4 pages 1–4

[2] *Parallel and Distributed Computational Intelligence* by Francisco Fernández de Vega 2010 ISBN 3-642-10674-9 pages 65–68

[3] Knight, Will: "IBM creates world's most powerful computer", *NewScientist.com news service*, June 2007

[4] http://s.top500.org/static/lists/2013/11/TOP500_201311_Poster.png

# Reconfigurable computing

**Reconfigurable computing** is a computer architecture combining some of the flexibility of software with the high performance of hardware by processing with very flexible high speed computing fabrics like field-programmable gate arrays (FPGAs). The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the datapath itself in addition to the control flow. On the other hand, the main difference with custom hardware, i.e. application-specific integrated circuits (ASICs) is the possibility to adapt the hardware during runtime by "loading" a new circuit on the reconfigurable fabric.

## History and properties

The concept of reconfigurable computing has existed since the 1960s, when Gerald Estrin's landmark paper proposed the concept of a computer made of a standard processor and an array of "reconfigurable" hardware.[1][2] The main processor would control the behavior of the reconfigurable hardware. The latter would then be tailored to perform a specific task, such as image processing or pattern matching, as quickly as a dedicated piece of hardware. Once the task was done, the hardware could be adjusted to do some other task. This resulted in a hybrid computer structure combining the flexibility of software with the speed of hardware; unfortunately this idea was far ahead of its time in needed electronic technology.

In the 1980s and 1990s there was a renaissance in this area of research with many proposed reconfigurable architectures developed in industry and academia,[3] such as: COPACOBANA [4], Matrix, Garp [5],[6] Elixent, PACT XPP, Silicon Hive, Montium, Pleiades, Morphosys, PiCoGA.[7] Such designs were feasible due to the constant progress of silicon technology that let complex designs be implemented on one chip. The world's first commercial reconfigurable computer, the Algotronix CHS2X4, was completed in 1991. It was not a commercial success, but was promising enough that Xilinx (the inventor of the Field-Programmable Gate Array, FPGA) bought the technology and hired the Algotronix staff.[8]

## Reconfigurable computing as a paradigm shift: using the Anti Machine

***Table 1: Nick Tredennick's Paradigm Classification Scheme***

| **Early Historic Computers:** | |
|---|---|
| | **Programming Source** |
| Resources fixed | none |
| Algorithms fixed | none |
| **von Neumann Computer:** | |
| | **Programming Source** |
| Resources fixed | none |
| Algorithms variable | Software (instruction streams) |
| **Reconfigurable Computing Systems:** | |
| | **Programming Source** |
| Resources variable | Configware (configuration) |
| Algorithms variable | Flowware (data streams) |

Computer scientist Reiner Hartenstein describes reconfigurable computing in terms of an *anti machine* that, according to him, represents a fundamental paradigm shift away from the more conventional von Neumann machine.

[9] Hartenstein calls it **Reconfigurable Computing Paradox,** that software-to-configware migration (software-to-FPGA migration) results in reported speed-up factors of up to more than four orders of magnitude, as well as a reduction in electricity consumption by up to almost four orders of magnitude—although the technological parameters of FPGAs are behind the Gordon Moore curve by about four orders of magnitude, and the clock frequency is substantially lower than that of microprocessors. This paradox is due to a paradigm shift, and is also partly explained by the Von Neumann syndrome.

The fundamental model of the reconfigurable computing machine paradigm, the data-stream-based anti machine is well illustrated by the differences to other machine paradigms that were introduced earlier, as shown by Nick Tredennick's following classification scheme of computing paradigms (see "Table 1: Nick Tredennick's Paradigm Classification Scheme"). [10]

The fundamental model of a Reconfigurable Computing Machine, the data-stream-based anti machine (also called Xputer), is the counterpart of the instruction-stream-based von Neumann machine paradigm. This is illustrated by a simple reconfigurable system (not *dynamically* reconfigurable), which has no instruction fetch at run time. The reconfiguration (before run time) can be considered as a kind of *super instruction fetch*. An anti machine does not have a program counter. The anti machine has data counters instead, since it is data-stream-driven. Here the definition of the term *data streams* is adopted from the systolic array scene, which defines, at which time which data item has to enter or leave which port, here of the reconfigurable system, which may be fine-grained (e. g. using FPGAs) or coarse-grained, or a mixture of both.

The systolic array scene, originally (early 1980s) mainly mathematicians, only defined one half of the anti machine: the data path: the systolic array (also see Super systolic array). But they did not define nor model the data sequencer methodology, considering that this is not their job to take care where the data streams come from or end up. The data sequencing part of the anti machine is modeled as distributed memory, preferably on chip, which consists of auto-sequencing memory (ASM) blocks. Each ASM block has a sequencer including a data counter. An example is the Generic Address Generator (GAG), which is a generalization of the DMA.

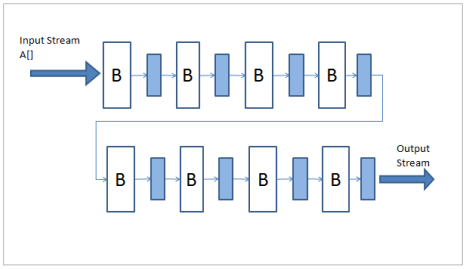
## Example of a streaming model of computation

Problem: We are given 2 character arrays of length 256: A[] and B[]. We need to compute the array C[] such that `C[i]=B[B[B[B[B[B[B[B[A[i]]]]]]]]]`. Though this problem is hypothetical, similar problems exist which have some applications.Wikipedia:Citation needed

Consider a software solution (C code) for the above problem:

```
for(int i=0;i<256;i++){
      char a=A[i];
      for(int j=0;j<8;j++)
            a=B[a];
      C[i]=a;
}
```

This program will take about 256*10*CPI cycles for the CPU, where CPI is the number of cycles per instruction.



Now, consider the hardware implementation shown here, say on an FPGA. Here, one element from the array 'A' is 'streamed' by a microprocessor into the circuit every cycle. The array 'B' is implemented as a ROM, perhaps in the BRAMs of the FPGA. The wire going into the ROMs labelled 'B' are the address lines and the wires out are the values stored in the ROM at that address. The blue boxes are registers used for storing temporary values Clearly, this is a pipeline and will output 1 value (a useful C[i] value) after the 8th cycle. Hence the output is also a 'stream'.

The hardware implementation takes 256+8 cycles. Hence, we can expect a speedup of about 10*CPI over the software implementation. However, the speedup is much less than this value due to the slow clock of the FPGA.

## Reconfigurable computing as a hobby

With the advent of affordable FPGA boards, there is an ever increasing number of students' and hobbyists' projects that seek to recreate vintage computers or implement more novel architectures, such as the RISC Wikipedia:Link rot .

An FPGA board is being used to recreate the Vector-06C computer


Game being played on the FPGA Vector-06C computer


The C-One emulating an Amstrad CPC

## References

[1] Estrin, G. 2002. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Ann. Hist. Comput.*24, 4 (Oct. 2002), 3–9. DOI=http://dx.doi.org/10.1109/MAHC.2002.1114865

[2] Estrin, G., "Organization of Computer Systems—The Fixed Plus Variable Structure Computer," *Proc. Western Joint Computer Conf.*, Western Joint Computer Conference, New York, 1960, pp. 33–40.

**[3]** C. Bobda: Introduction to Reconfigurable Computing: Architectures; Springer, 2007

[4] http://www.sciengines.com/copacobana/

[5] http://brass.cs.berkeley.edu/garp.html

[6] Hauser, John R. and Wawrzynek, John, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '97, April 16–18, 1997), pp. 24–33.

**[7]** Campi, F.; Toma, M.; Lodi, A.; Cappelli, A.; Canegallo, R.; Guerrieri, R., "A VLIW processor with reconfigurable instruction set for embedded applications," Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International , vol., no., pp. 250-491 vol.1, 2003

[8] Algotronix History (http://www.algotronix.com/people/tom/album.html)

[9] Hartenstein, R. 2001. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2001)* (Munich, Germany). W. Nebel and A. Jerraya, Eds. Design, Automation, and Test in Europe. IEEE Press, Piscataway, NJ, 642–649.

[10] N. Tredennick: The Case for Reconfigurable Computing; Microprocessor Report, Vol. 10 No. 10, 5 August 1996, pp 25–27.

## Further reading


- Cardoso, João M. P.; Hübner, Michael (Eds.), *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign* (http://www.springer.com/engineering/circuits+&+systems/book/978-1-4614-0060-8), Springer, 2011.
- S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*, Morgan Kaufmann, 2008.
- J. Henkel, S. Parameswaran (editors): Designing Embedded Processors. A Low Power Perspective; Springer Verlag, March 2007
- J. Teich (editor) et al.: Reconfigurable Computing Systems. Special Topic Issue of Journal *it — Information Technology*, Oldenbourg Verlag, Munich. Vol. 49(2007) Issue 3 (http://www.atypon-link.com/OLD/toc/itit/49/3)
- T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung, "Reconfigurable Computing: Architectures and Design Methods", IEEE Proceedings: Computer & Digital Techniques, Vol. 152, No. 2, March 2005, pp. 193–208.
- A. Zomaya (editor): Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies; Springer Verlag, 2006
- J. M. Arnold and D. A. Buell, "VHDL programming on Splash 2," in More FPGAs, Will Moore and Wayne Luk, editors, Abingdon EE & CS Books, Oxford, England, 1994, pp. 182–191. (Proceedings,International Workshop on Field-Programmable Logic, Oxford, 1993.)
- J. M. Arnold, D. A. Buell, D. Hoang, D. V. Pryor, N. Shirazi, M. R. Thistle, "Splash 2 and its applications, "Proceedings, International Conference on Computer Design, Cambridge, 1993, pp. 482–486.
- D. A. Buell and Kenneth L. Pocek, "Custom computing machines: An introduction," The Journal of Supercomputing, v. 9, 1995, pp. 219–230.


## External links

- The Fine-grained Computing Group at Information Sciences Institute (http://www3.isi.edu/research/research-divisions/div10-home/div10-research_overview/div10-research_fine_grain_computing.htm)
- Reconfigurable computing lectures and tutorials at Brown University (http://scale.engin.brown.edu/classes/EN2911XF07/)
- A Decade of Reconfigurable Computing: a Visionary Retrospective (http://www.ics.uci.edu/~dutt/ics212-wq05/hartenstein-recongtut-date01.pdf)
- Reconfigurable Computing: Coming of Age (http://pw1.netcom.com/~optmagic/reconfigure/fawcett.html)
- The University of South Carolina Reconfigurable Computing Laboratory (http://www.cse.sc.edu/~buell/Public_Data/reconlab.html)
- The Virginia Tech Configurable Computing Laboratory (http://www.ccm.ece.vt.edu/)
- Reconfigurable Systems Summer Institute (RSSI) (http://rssi.ncsa.uiuc.edu/)
- IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) (http://www.fccm.org/)
- International Conference on Field-Programmable Logic and Applications (FPL) (http://fpl.org/)
- BYU Configurable Computing Laboratory's FPGA CAD tool set (http://www.jhdl.org/)
- The Morphware Forum (http://morphware.net/)
- NSF Center for High-Performance Reconfigurable Computing (CHREC) (http://www.chrec.org/)
- The OpenFPGA effort (http://www.openfpga.org/)
- RC Education Workshop (http://helios.informatik.uni-kl.de/RCeducation/)
- Reconfigurable Architectures Workshop (http://xputers.informatik.uni-kl.de/raw/index_raw.html#raw_06)
- The George Washington University High Performance Computing Laboratory (http://hpcl.seas.gwu.edu/)

- The University of Florida High-Performance Computing & Simulation Research Laboratory (http://www.hcs.ufl.edu/)
- The University of Kansas Hybridthreads Project - OS for Hybrid CPU/FPGA chips (http://www.ittc.ku.edu/hybridthreads)
- Reconfigurable computing tools and O/S Support from the University of Wisconsin (http://www.ece.wisc.edu/~kati/research.html)
- Circuits and Systems Group, Imperial College London (http://cas.ee.ic.ac.uk)
- Why we need reconfigurable computing education (http://isvlsi06.itiv.uni-karlsruhe.de/RCe-i1.pdf)
- The on-line version of the 2D/3D MEANDER FPGA design framework (http://proteas.microlab.ntua.gr)
- FHPCA: FPGA High Performance Computing Alliance (http://www.fhpca.org/)
- Website of the DRESD (Dynamic Reconfigurability in Embedded System Design) research project (http://www.dresd.org/)
- Advanced topics in computer architecture: chip multiprocessors and polymorphic processors (2003) (http://www.stanford.edu/class/ee392c/)
- UT Austin TRIPS multiprocessor (http://www.cs.utexas.edu/users/cart/trips/)
- UNC Charlotte reconfigurable computing cluster (http://rcs.uncc.edu/wiki/index.php/Main_Page)
- XiRisc/PiCoGA project at University of Bologna, Italy (http://www.arces.unibo.it/content/view/31/243)
- COPACOBANA Project, Germany (http://www.sciengines.com/copacobana/)
- "Reconfigurable Computing" course in University of Erlangen, Germany (http://www12.informatik.uni-erlangen.de/edu/rc/)

# Field-programmable gate array

"FPGA" redirects here. It is not to be confused with Flip-chip pin grid array.

A **field-programmable gate array** (**FPGA**) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare).

A FPGA from Altera

Contemporary FPGAs have large resources of logic gates and RAM blocks to implement complex digital computations. As FPGA designs employ very fast I/Os and bidirectional data buses it becomes a challenge to verify correct timing of valid data within setup time and hold time. Floor planning enables resources allocation within FPGA to meet these time constraints. FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of a portion of the design and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications.[1]

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" – somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

FPGA from Xilinx

Some FPGAs have analog features in addition to digital functions. The most common analog feature is programmable slew rate and drive strength on each output pin, allowing the engineer to set slow rates on lightly loaded pins that would otherwise ring unacceptably, and to set stronger, faster rates on heavily loaded pins on high-speed channels that would otherwise run too slowly.[2][3] Another relatively common analog feature is differential comparators on input pins designed to be connected to differential signaling channels. A few "mixed signal FPGAs" have integrated peripheral analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) with analog signal conditioning blocks allowing them to operate as a system-on-a-chip.[4] Such devices blur the line between an FPGA, which carries digital ones and zeros on its internal programmable interconnect fabric, and field-programmable analog array (FPAA), which carries analog values on its internal programmable interconnect fabric.

## History

The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both had the option of being programmed in batches in a factory or in the field (field programmable). However programmable logic was hard-wired between logic gates.[5]

In the late 1980s the Naval Surface Warfare Department funded an experiment proposed by Steve Casselman to develop a computer that would implement 600,000 reprogrammable gates. Casselman was successful and a patent related to the system was issued in 1992.

Some of the industry's foundational concepts and technologies for programmable logic arrays, gates, and logic blocks are founded in patents awarded to David W. Page and LuVerne R. Peterson in 1985.[6][7]

Xilinx co-founders Ross Freeman and Bernard Vonderschmitt invented the first commercially viable field programmable gate array in 1985 – the XC2064.[8] The XC2064 had programmable gates and programmable interconnects between gates, the beginnings of a new technology and market.[9] The XC2064 boasted a mere 64 configurable logic blocks (CLBs), with two 3-input lookup tables (LUTs).[10] More than 20 years later, Freeman was entered into the National Inventors Hall of Fame for his invention.[11]

Xilinx continued unchallenged and quickly growing from 1985 to the mid-1990s, when competitors sprouted up, eroding significant market-share. By 1993, Actel was serving about 18 percent of the market.

The 1990s were an explosive period of time for FPGAs, both in sophistication and the volume of production. In the early 1990s, FPGAs were primarily used in telecommunications and networking. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications.

## Modern developments

A recent trend has been to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". This work mirrors the architecture by Ron Perlof and Hana Potash of Burroughs Advanced Systems Group which combined a reconfigurable CPU architecture on a single chip called the SB24. That work was done in 1982. Examples of such hybrid technologies can be found in the Xilinx Zynq™-7000 All Programmable SoC, which includes a 1.0 GHz dual-core ARM Cortex-A9 MPCore processor embedded within the FPGA's logic fabric or in the Altera Arria V FPGA which includes an 800 MHz dual-core ARM Cortex-A9 MPCore. The Atmel FPSLIC is another such device, which uses an AVR processor in combination with Atmel's programmable logic architecture. The Actel SmartFusion devices incorporate an ARM Cortex-M3 hard processor core (with up to 512 kB of flash and 64 kB of RAM) and analog peripherals such as a multi-channel ADC and DACs to their flash-based FPGA fabric.

In 2010, Xilinx Inc introduced the first All Programmable System on a Chip branded Zynq™-7000 that fused features of an ARM high-end microcontroller (hard-core implementations of a 32-bit processor, memory, and I/O) with an FPGA fabric to make FPGAs easier for embedded designers to use. By incorporating the ARM processor-based platform into a 28 nm FPGA family, the extensible processing platform enables system architects and embedded software developers to apply a combination of serial and parallel processing to their embedded system designs, for which the general trend has been to progressively increasing complexity. The high level of integration helps to reduce power consumption and dissipation, and the reduced parts count vs. using an FPGA with a separate CPU chip leads to a lower parts cost, a smaller system, and higher reliability since most failures in modern electronics occur on PCBs in the connections between chips instead of within the chips themselves.[12][13][14][15][16]

An alternate approach to using hard-macro processors is to make use of soft processor cores that are implemented within the FPGA logic. Nios II, MicroBlaze and Mico32 are examples of popular softcore processors.

As previously mentioned, many modern FPGAs have the ability to be reprogrammed at "run time," and this is leading to the idea of reconfigurable computing or reconfigurable systems — CPUs that reconfigure themselves to suit the task at hand.

Additionally, new, non-FPGA architectures are beginning to emerge. Software-configurable microprocessors such as the Stretch S5000 adopt a hybrid approach by providing an array of processor cores and FPGA-like programmable cores on the same chip.

## Gates

- 1982: 8192 gates, Burroughs Advances Systems Group, integrated into the S-Type 24 bit processor for reprogrammable I/O.
- 1987: 9,000 gates, Xilinx
- 1992: 600,000, Naval Surface Warfare Department
- Early 2000s: Millions

### Market size

- 1985: First commercial FPGA : Xilinx XC2064
- 1987: $14 million
- ~1993: >$385 million
- 2005: $1.9 billion[17]
- 2010 estimates: $2.75 billion

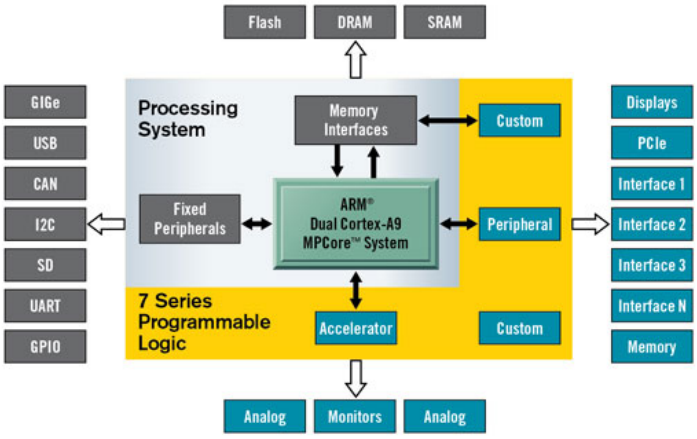
### FPGA design starts

- 2005: 80,000[18]
- 2008: 90,000[19]

## FPGA comparisons

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. An older study had shown that designs implemented on FPGAs need on average 40 times as much area, draw 12 times as much dynamic power, and run at one third the speed of corresponding ASIC implementations. More recently, FPGAs such as the Xilinx Virtex-7 or the Altera Stratix 5 have come to rival corresponding ASIC and ASSP solutions by providing significantly reduced power, increased speed, lower materials cost, minimal implementation real-estate, and increased possibilities for re-configuration 'on-the-fly'. Where previously a design may have included 6 to 10 ASICs, the same design can now be achieved using only one FPGA.

Advantages include the ability to re-program in the field to fix bugs, and may include a shorter time to market and lower non-recurring engineering costs. Vendors can also take a middle road by developing their hardware on ordinary FPGAs, but manufacture their final version as an ASIC so that it can no longer be modified after the design has been committed.



A Xilinx Zynq-7000 All Programmable System on a Chip.

Xilinx claims that several market and technology dynamics are changing the ASIC/FPGA paradigm:[20]

- Integrated circuit costs are rising aggressively
- ASIC complexity has lengthened development time
- R&D resources and headcount are decreasing
- Revenue losses for slow time-to-market are increasing
- Financial constraints in a poor economy are driving low-cost technologies

These trends make FPGAs a better alternative than ASICs for a larger number of higher-volume applications than they have been historically used for, to which the company attributes the growing number of FPGA design starts (see History).

Some FPGAs have the capability of partial re-configuration that lets one portion of the device be re-programmed while other portions continue running.

### Complex programmable logic devices (CPLD)

The primary differences between CPLDs (complex programmable logic devices) and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also far more complex to design for.

In practice, the distinction between FPGAs and CPLDs is often one of size as FPGAs are usually much larger in terms of resources than CPLDs. Typically only FPGA's contain more complex embedded functions such as adders, multipliers, memory, and serdes. Another common distinction is that CPLDs contain embedded flash to store their configuration while FPGAs usually, but not always, require an external nonvolatile memory.

### Security considerations

With respect to security, FPGAs have both advantages and disadvantages as compared to ASICs or secure microprocessors. FPGAs' flexibility makes malicious modifications during fabrication a lower risk.[21] Previously, for many FPGAs, the design bitstream is exposed while the FPGA loads it from external memory (typically on every power-on). All major FPGA vendors now offer a spectrum of security solutions to designers such as bitstream encryption and authentication. For example, Altera and Xilinx offer AES (up to 256 bit) encryption for bitstreams stored in an external flash memory.

FPGAs that store their configuration internally in nonvolatile flash memory, such as Microsemi's ProAsic 3 or Lattice's XP2 programmable devices, do not expose the bitstream and do not need encryption. In addition, flash memory for LUT provides SEU protection for space applications.Wikipedia:Please clarify

## Applications

Applications of FPGAs include digital signal processing, software-defined radio, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.[22][23]

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

Common FPGA Applications:

Aerospace and Defense

- Avionics/DO-254
- Communications
- Missiles & Munitions
- Secure Solutions
- Space

ASIC Prototyping

Audio

- Connectivity Solutions
- Portable Electronics
- Radio
- Digital Signal Processing (DSP)

Automotive

- High Resolution Video
- Image Processing
- Vehicle Networking and Connectivity
- Automotive Infotainment

Broadcast

- Real-Time Video Engine
- EdgeQAM
- Encoders
- Displays
- Switches and Routers

Consumer Electronics

- Digital Displays
- Digital Cameras
- Multi-function Printers
- Portable Electronics
- Set-top Boxes

Distributed Monetary Systems

- Transaction verification
- BitCoin Mining

Data Center

- Servers
- Security
- Routers
- Switches
- Gateways
- Load Balancing

High Performance Computing

- Servers
- Super Computers
- SIGINT Systems
- High-end RADARS
- High-end Beam Forming Systems
- Data Mining Systems

Industrial

- Industrial Imaging
- Industrial Networking
- Motor Control

Medical

- Ultrasound
- CT Scanner
- MRI
- X-ray
- PET
- Surgical Systems

Scientific Instruments

- Lock-in amplifiers
- Boxcar averagers
- Phase-locked loops

Security

- Industrial Imaging
- Secure Solutions
- Image Processing

Video & Image Processing

- High Resolution Video
- Video Over IP Gateway
- Digital Displays
- Industrial Imaging

Wired Communications

- Optical Transport Networks
- Network Processing
- Connectivity Interfaces

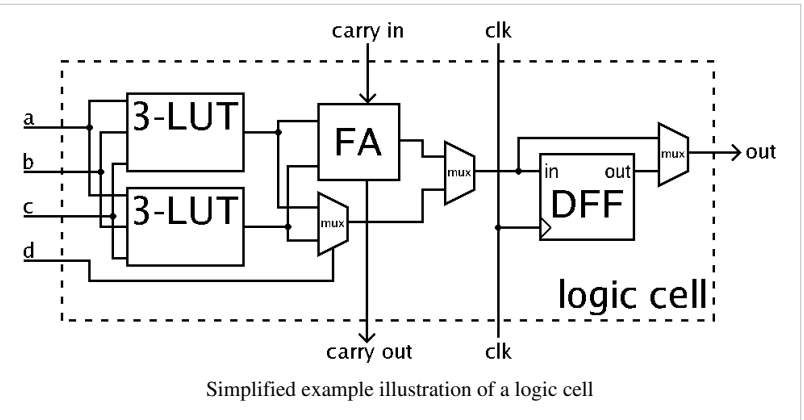
Wireless Communications

- Baseband
- Connectivity Interfaces
- Mobile Backhaul
- Radio

## Architecture

The most common FPGA architecture consists of an array of logic blocks (called Configurable Logic Block, CLB, or Logic Array Block, LAB, depending on vendor), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array.

An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs/LABs and I/Os required is easily determined from the design, the number of routing tracks needed may vary considerably even among designs with the same amount of logic. For example, a crossbar switch requires much more routing than a systolic array with the same gate count. Since unused routing tracks increase the cost (and decrease the performance) of the part without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs that will fit in terms of Lookup tables (LUTs) and I/Os can be routed. This is determined by estimates such as those derived from Rent's rule or by experiments with existing designs.



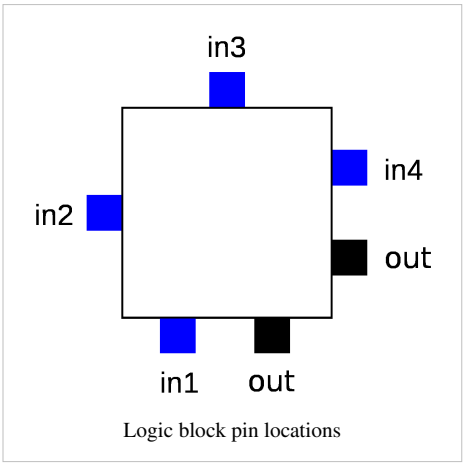
Simplified example illustration of a logic cell

In general, a logic block (CLB or LAB) consists of a few logical cells (called ALM, LE, Slice etc.). A typical cell consists of a 4-input LUT, a Full adder (FA) and a D-type flip-flop, as shown below. The LUTs are in this figure split into two 3-input LUTs. In *normal mode* those are combined into a 4-input LUT through the left mux. In *arithmetic* mode, their outputs are fed to the FA. The selection of mode is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space.[24][25]

ALMs and Slices usually contains 2 or 4 structures similar to the example figure, with some shared signals.

CLBs/LABs typically contains a few ALMs/LEs/Slices.

In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.



Logic block pin locations

Since clock signals (and often other high-fan-out signals) are normally routed via special-purpose dedicated routing networks (i.e. global buffers) in commercial FPGAs, they and other signals are separately managed.

For this example architecture, the locations of the FPGA logic block pins are shown to the right.

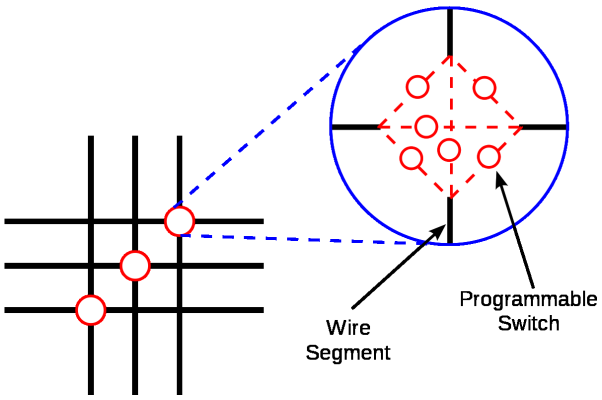
Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block.

Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

Whenever a vertical and a horizontal channel intersect, there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure on the right illustrates the connections in a switch box.



Switch box topology

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed I/O logic and embedded memories.

FPGAs are also widely used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time-to-market.

To shrink the size and power consumption of FPGAs, vendors such as Tabula and Xilinx have introduced new 3D or stacked architectures.[26][27] Following the introduction of its 28 nm 7-series FPGAs, Xilinx revealed that several of the highest-density parts in those FPGA product lines will be constructed using multiple dies in one package, employing technology developed for 3D construction and stacked-die assemblies. The technology stacks several (three or four) active FPGA dice side-by-side on a silicon interposer – a single piece of silicon that carries passive interconnect.[28]

## FPGA design and programming

To define the behavior of the FPGA, the user provides a hardware description language (HDL) or a schematic design. The HDL form is more suited to work with large structures because it's possible to just specify them numerically rather than having to draw every piece by hand. However, schematic entry can allow for easier visualisation of a design.

Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA. This file is transferred to the FPGA/CPLD via a serial interface (JTAG) or to an external memory device like an EEPROM.

The most common HDLs are VHDL and Verilog, although in an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level through the introduction of alternative languages. National Instruments' LabVIEW graphical programming language (sometimes referred to as "G") has an FPGA add-in module available to target and program FPGA hardware.

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly

called *IP cores*, and are available from FPGA vendors and third-party IP suppliers (rarely free, and typically released under proprietary licenses). Other predefined circuits are available from developer communities such as OpenCores (typically released under free and open source licenses such as the GPL, BSD or similar license), and other sources.

In a typical design flow, an FPGA application developer will simulate the design at multiple stages throughout the design process. Initially the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. Then, after the synthesis engine has mapped the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.

## Basic process technology types

- SRAM - based on static memory technology. In-system programmable and re-programmable. Requires external boot devices. CMOS. Currently Wikipedia:Manual of Style/Dates and numbers#Chronological items in use.
- Fuse - One-time programmable. Bipolar. Obsolete.
- Antifuse - One-time programmable. CMOS.
- PROM - Programmable Read-Only Memory technology. One-time programmable because of plastic packaging. Obsolete.
- EPROM - Erasable Programmable Read-Only Memory technology. One-time programmable but with window, can be erased with ultraviolet (UV) light. CMOS. Obsolete.
- EEPROM - Electrically Erasable Programmable Read-Only Memory technology. Can be erased, even in plastic packages. Some but not all EEPROM devices can be in-system programmed. CMOS.
- Flash - Flash-erase EPROM technology. Can be erased, even in plastic packages. Some but not all flash devices can be in-system programmed. Usually, a flash cell is smaller than an equivalent EEPROM cell and is therefore less expensive to manufacture. CMOS.

## Major manufacturers

Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percent of the market.

Both Xilinx and Altera provide free Windows and Linux design software (ISE and Quartus) which provides limited sets of devices.

Other competitors include Lattice Semiconductor (SRAM based with integrated configuration flash, instant-on, low power, live reconfiguration), Actel (now Microsemi, antifuse, flash-based, mixed-signal), SiliconBlue Technologies (extremely low power SRAM-based FPGAs with optional integrated nonvolatile configuration memory; acquired by Lattice in 2011), Achronix (SRAM based, 1.5 GHz fabric speed), and QuickLogic (handheld focused CSSP, no general purpose FPGAs).

In March 2010, Tabula announced their FPGA technology that uses time-multiplexed logic and interconnect that claims potential cost savings for high-density applications.

# References


[1] FPGA Architecture for the Challenge (http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html)

[2] FPGA Signal Integrity tutorial (http://wiki.altium.com/display/ADOH/FPGA+SI+Tutorial+-+Simulating+the+Reflection+Characteristics)

[3] NASA: FPGA drive strength (http://klabs.org/richcontent/fpga_content/DesignNotes/signal_quality/actel_drive_strength/index.htm)

[4] Mike Thompson. "Mixed-signal FPGAs provide GREEN POWER" (http://www.eetimes.com/showArticle.jhtml?articleID=200000777). EE Times, 2007-07-02.

[5] History of FPGAs (http://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm)

[6] Google Patent Search, " Re-programmable PLA (http://www.google.com/patents?id=BB4vAAAAEBAJ&dq=4508977)". Retrieved February 5, 2009.

[7] Google Patent Search, " Dynamic data re-programmable PLA (http://www.google.com/patents?id=1-gzAAAAEBAJ&dq=4524430)". Retrieved February 5, 2009.

[8] Peter Clarke, EE Times, " Xilinx, ASIC Vendors Talk Licensing (http://www.eetimes.com/story/OEG20010622S0091)." June 22, 2001. Retrieved February 10, 2009.

[9] Funding Universe. “ Xilinx, Inc. (http://www.fundinguniverse.com/company-histories/Xilinx-Inc-Company-History.html)” Retrieved January 15, 2009.

[10] Clive Maxfield, Programmable Logic DesignLine, " Xilinx unveil revolutionary 65nm FPGA architecture: the Virtex-5 family (http://www.pldesignline.com/products/187203173). May 15, 2006. Retrieved February 5, 2009.

[11] Press Release, " Xilinx Co-Founder Ross Freeman Honored as 2009 National Inventors Hall of Fame Inductee for Invention of FPGA (http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1255523&highlight)"

[12] McConnel, Toni. EETimes. " ESC - Xilinx All Programmable System on a Chip combines best of serial and parallel processing (http://www.eetimes.com/electronics-products/processors/4115537/ESC--Xilinx-Extensible-Processing-Platform-combines-best-of-serial-and-parallel-processing)." April 28, 2010. Retrieved February 14, 2011.

[13] Cheung, Ken, FPGA Blog. " Xilinx Extensible Processing Platform for Embedded Systems (http://fpgablog.com/posts/arm-cortex-mpcore/)." April 27, 2010. Retrieved February 14, 2011.

[14] Nass, Rich, EETimes. " Xilinx puts ARM core into its FPGAs (http://www.eetimes.com/electronics-products/processors/4115523/Xilinx-puts-ARM-core-into-its-FPGAs)." April 27, 2010. Retrieved February 14, 2011.

[15] Leibson, Steve, Design-Reuse. " Xilinx redefines the high-end microcontroller with its ARM-based Extensible Processing Platform - Part 1 (http://www.design-reuse.com/industryexpertblogs/23302/xilinx-arm-based-extensible-processing-platform.html)." May. 03, 2010. Retrieved February 15, 2011.

[16] Wilson, Richard, Electronics Weekly. " Xilinx acquires ESL firm to make FPGAs easier to use (http://www.electronicsweekly.com/Articles/2011/01/31/50386/xilinx-acquires-esl-firm-to-make-fpgas-easier-to-use.htm)." January 31, 2011. Retrieved February 15, 2011.

[17] Dylan McGrath, EE Times, " FPGA Market to Pass $2.7 Billion by '10, In-Stat Says (http://www.eetimes.com/news/design/business/showArticle.jhtml?articleID=188102617)". May 24, 2006. Retrieved February 5, 2009.

[18] Dylan McGrath, EE Times, " Gartner Dataquest Analyst Gives ASIC, FPGA Markets Clean Bill of Health (http://www.eetimes.com/conf/dac/showArticle.jhtml?articleID=164302400)". June 13, 2005. Retrieved February 5, 2009.

[19] Virtex-4 Family Overview (http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf)

[20] Tim Erjavec, White Paper, " Introducing the Xilinx Targeted Design Platform: Fulfilling the Programmable Imperative (http://www.xilinx.com/publications/prod_mktg/Targeted_Design_Platforms.pdf)." February 2, 2009. Retrieved February 2, 2009

[21] Huffmire Paper " Managing Security in FPGA-Based Embedded Systems (http://www2.computer.org/portal/web/csdl/doi/10.1109/MDT.2008.166)." Nov-Dec 2008. Retrieved Sept 22, 2009

[22] FPGA/DSP Blend Tackles Telecom Apps (http://www.bdti.com/articles/info_eet0207fpga.htm)

[23] Xilinx aims 65-nm FPGAs at DSP applications (http://www.eetimes.com/showArticle.jhtml?articleID=197001881)

[24] http://www.altera.com/literature/hb/cyc2/cyc2_cii51002.pdf

[25] http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[26] Dean Takahashi, VentureBeat. " Intel connection helped chip startup Tabula raise $108M (http://venturebeat.com/2011/05/02/intel-connection-helped-chip-startup-tabula-raise-108m)." May 2, 2011. Retrieved May 13, 2011.

[27] Lawrence Latif, The Inquirer. " FPGA manufacturer claims to beat Moore's Law (http://www.theinquirer.net/inquirer/news/1811460/fpga-manufacturer-claims-beat-moores-law)." October 27, 2010. Retrieved May 12, 2011.

[28] EDN Europe. " Xilinx adopts stacked-die 3D packaging (http://www.edn-europe.com/xilinxadoptsstackeddie3dpackaging+article+4461+Europe.html)." November 1, 2010. Retrieved May 12, 2011.

## Further reading


- Sadrozinski, Hartmut F.-W.; Wu, Jinyuan (2010). *Applications of Field-Programmable Gate Arrays in Scientific Research*. Taylor & Francis. ISBN 978-1-4398-4133-4.
- Wirth, Niklaus (1995). *Digital Circuit Design An Introduction Textbook*. Springer. ISBN 3-540-58577-X.


## External links

- University of North Carolina at Charlotte's Reconfigurable Computing Laboratory (http://rcs.uncc.edu/)
- Vaughn Betz's FPGA Architecture Page (http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html)
- Tutorials and Examples on FPGAs (http://www.fpgacenter.com)
- What is an FPGA? (https://www.youtube.com/watch?v=gUsHwi4M4xE) on YouTube

# General-purpose computing on graphics processing units

**General-Purpose Computing on Graphics Processing Units** (**GPGPU**, rarely **GPGP** or **GP²U**) is the utilization of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).[1][2][3] Any GPU providing a functionally complete set of operations performed on arbitrary bits can compute any computable value. Additionally, the use of multiple graphics cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing.[4]

OpenCL is the currently dominant open general-purpose GPU computing language. The dominant proprietary framework is Nvidia's CUDA.[5]

## Programmability

In principle, any boolean function can be built-up from a functionally complete set of logic operators. An early example of general purpose computing with a GPU involved performing additions by using an early stream processor called a blitter to invoke a special sequence of logical operations on bit vectors. Such methods are seldom used today as modern GPUs now include support for more advanced mathematical operations including addition, multiplication, and often certain transcendental functions.

The programmability of the pipelines have trendedWikipedia:Please clarify according to Microsoft's DirectX specification,Wikipedia:Citation needed with DirectX 8 introducing Shader Model 1.1, DirectX 8.1 Pixel Shader Models 1.2, 1.3 and 1.4, and DirectX 9 defining Shader Model 2.x and 3.0. Each shader model increased the programming model flexibilities and capabilities, ensuring the conforming hardware follows suit. The DirectX 10 specification introduces Shader Model 4.0 which unifies the programming specification for vertex, geometry ("Geometry Shaders" are new to DirectX 10) and fragment processing allowing for a better fit for unified shader hardware, thus providing one computational pool of programmable resource.Wikipedia:Vagueness

## Data types

Pre-DirectX 9 graphics cards only supported paletted or integer color types. Various formats are available, each containing a red element, a green element, and a blue element.Wikipedia:Citation needed Sometimes an additional alpha value is added, to be used for transparency. Common formats are:

- 8 bits per pixel – Sometimes palette mode, where each value is an index in a table with the real color value specified in one of the other formats. Sometimes three bits for red, three bits for green, and two bits for blue.
- 16 bits per pixel – Usually allocated as five bits for red, six bits for green, and five bits for blue.
- 24 bits per pixel – eight bits for each of red, green, and blue
- 32 bits per pixel – eight bits for each of red, green, blue, and alpha

For early fixed-function or limited programmability graphics (i.e. up to and including DirectX 8.1-compliant GPUs) this was sufficient because this is also the representation used in displays. This representation does have certain limitations, however. Given sufficient graphics processing power even graphics programmers would like to use better formats, such as floating point data formats, to obtain effects such as high dynamic range imaging. Many GPGPU applications require floating point accuracy, which came with graphics cards conforming to the DirectX 9 specification.

DirectX 9 Shader Model 2.x suggested the support of two precision types: full and partial precision. Full precision support could either be FP32 or FP24 (floating point 32- or 24-bit per component) or greater, while partial precision was FP16. ATI's R300 series of GPUs supported FP24 precision only in the programmable fragment pipeline (although FP32 was supported in the vertex processors) while Nvidia's NV30 series supported both FP16 and FP32; other vendors such as S3 Graphics and XGI supported a mixture of formats up to FP24.

Shader Model 3.0 altered the specification, increasing full precision requirements to a minimum of FP32 support in the fragment pipeline. ATI's Shader Model 3.0 compliant R5xx generation (Radeon X1000 series) supports just FP32 throughout the pipeline while Nvidia's NV4x and G7x series continued to support both FP32 full precision and FP16 partial precisions. Although not stipulated by Shader Model 3.0, both ATI and Nvidia's Shader Model 3.0 GPUs introduced support for blendable FP16 render targets, more easily facilitating the support for High Dynamic Range Rendering.Wikipedia:Citation needed

The implementations of floating point on Nvidia GPUs are mostly IEEE compliant; however, this is not true across all vendors.[6] This has implications for correctness which are considered important to some scientific applications. While 64-bit floating point values (double precision float) are commonly available on CPUs, these are not universally supported on GPUs; some GPU architectures sacrifice IEEE compliance while others lack double-precision altogether. There have been efforts to emulate double-precision floating point values on GPUs; however, the speed tradeoff negates any benefit to offloading the computation onto the GPU in the first place.[7]

Most operations on the GPU operate in a vectorized fashion: one operation can be performed on up to four values at once. For instance, if one color <R1, G1, B1> is to be modulated by another color <R2, G2, B2>, the GPU can produce the resulting color <R1*R2, G1*G2, B1*B2> in one operation. This functionality is useful in graphics because almost every basic data type is a vector (either 2-, 3-, or 4-dimensional). Examples include vertices, colors, normal vectors, and texture coordinates. Many other applications can put this to good use, and because of their higher performance, vector instructions (SIMD) have long been available on CPUs.

In 2002, James Fung *et al* developed OpenVIDIA at University of Toronto, and demonstrated this work, which was later published in 2003, 2004, and 2005,[8] in conjunction with a collaboration between University of Toronto and nVIDIA. In November 2006 Nvidia launched CUDA, an SDK and API that allows using the C programming language to code algorithms for execution on Geforce 8 series GPUs. OpenCL, an open standard defined by the Khronos Group[9] provides a cross-platform GPGPU platform that additionally supports data parallel compute on CPUs. OpenCL is actively supported on Intel, AMD, Nvidia and ARM platforms. GPGPU compared, for example, to traditional floating point accelerators such as the 64-bit CSX700 boards from ClearSpeed that are used in today's

supercomputers, current top-end GPUs from AMD and Nvidia emphasize single-precision (32-bit) computation; double-precision (64-bit) computation executes more slowly.Wikipedia:Citation needed

# GPGPU programming concepts

GPUs are designed specifically for graphics and thus are very restrictive in operations and programming. Due to their design, GPUs are only effective for problems that can be solved using stream processing and the hardware can only be used in certain ways.

## Stream processing

Main article: Stream processing

GPUs can only process independent vertices and fragments, but can process many of them in parallel. This is especially effective when the programmer wants to process many vertices or fragments in the same way. In this sense, GPUs are stream processors – processors that can operate in parallel by running one kernel on many records in a stream at once.

A *stream* is simply a set of records that require similar computation. Streams provide data parallelism. *Kernels* are the functions that are applied to each element in the stream. In the GPUs, *vertices* and *fragments* are the elements in streams and vertex and fragment shaders are the kernels to be run on them. Since GPUs process elements independently there is no way to have shared or static data. For each element we can only read from the input, perform operations on it, and write to the output. It is permissible to have multiple inputs and multiple outputs, but never a piece of memory that is both readable and writable. Wikipedia:Vagueness

Arithmetic intensity is defined as the number of operations performed per word of memory transferred. It is important for GPGPU applications to have high arithmetic intensity else the memory access latency will limit computational speedup.[10]

Ideal GPGPU applications have large data sets, high parallelism, and minimal dependency between data elements.

## GPU programming concepts

### Computational resources

There are a variety of computational resources available on the GPU:

- Programmable processors – Vertex, primitive, and fragment pipelines allow programmer to perform kernel on streams of data
- Rasterizer – creates fragments and interpolates per-vertex constants such as texture coordinates and color
- Texture Unit – read only memory interface
- Framebuffer – write only memory interface

In fact, the programmer can substitute a write only texture for output instead of the framebuffer. This is accomplished either through Render to Texture (RTT), Render-To-Backbuffer-Copy-To-Texture (RTBCTT), or the more recent stream-out.

#### Textures as stream

The most common form for a stream to take in GPGPU is a 2D grid because this fits naturally with the rendering model built into GPUs. Many computations naturally map into grids: matrix algebra, image processing, physically based simulation, and so on.

Since textures are used as memory, texture lookups are then used as memory reads. Certain operations can be done automatically by the GPU because of this.

#### Kernels

Kernels can be thought of as the body of loops. For example, a programmer operating on a grid on the CPU might have code that looks like this:

```
// Input and output grids have 10000 x 10000 or 100 million elements.

void transform_10k_by_10k_grid(float in[10000][10000], float
out[10000][10000])
{
    for (int x = 0; x < 10000; x++) {
        for (int y = 0; y < 10000; y++) {
            // The next line is executed 100 million times
            out[x][y] = do_some_hard_work(in[x][y]);
        }
    }
}
```

On the GPU, the programmer only specifies the body of the loop as the kernel and what data to loop over by invoking geometry processing.

#### Flow control

In sequential code it is possible to control the flow of the program using if-then-else statements and various forms of loops. Such flow control structures have only recently been added to GPUs.[11] Conditional writes could be accomplished using a properly crafted series of arithmetic/bit operations, but looping and conditional branching were not possible.

Recent GPUs allow branching, but usually with a performance penalty. Branching should generally be avoided in inner loops, whether in CPU or GPU code, and various methods, such as static branch resolution, pre-computation, predication, loop splitting,[12] and Z-cull[13] can be used to achieve branching when hardware support does not exist.

### GPU methods

#### Map

Main article: Map (higher-order function)

The map operation simply applies the given function (the kernel) to every element in the stream. A simple example is multiplying each value in the stream by a constant (increasing the brightness of an image). The map operation is simple to implement on the GPU. The programmer generates a fragment for each pixel on screen and applies a fragment program to each one. The result stream of the same size is stored in the output buffer.

### Reduce

Main article: Fold (higher-order function)

Some computations require calculating a smaller stream (possibly a stream of only 1 element) from a larger stream. This is called a reduction of the stream. Generally a reduction can be accomplished in multiple steps. The results from the prior step are used as the input for the current step and the range over which the operation is applied is reduced until only one stream element remains.

### Stream filtering

Stream filtering is essentially a non-uniform reduction. Filtering involves removing items from the stream based on some criteria.

### Scatter

The scatter operation is most naturally defined on the vertex processor. The vertex processor is able to adjust the position of the vertex, which allows the programmer to control where information is deposited on the grid. Other extensions are also possible, such as controlling how large an area the vertex affects.

The fragment processor cannot perform a direct scatter operation because the location of each fragment on the grid is fixed at the time of the fragment's creation and cannot be altered by the programmer. However, a logical scatter operation may sometimes be recast or implemented with an additional gather step. A scatter implementation would first emit both an output value and an output address. An immediately following gather operation uses address comparisons to see whether the output value maps to the current output slot.

### Gather

The fragment processor is able to read textures in a random access fashion, so it can gather information from any grid cell, or multiple grid cells, as desired.Wikipedia:Vagueness

### Sort

The sort operation transforms an unordered set of elements into an ordered set of elements. The most common implementation on GPUs is using sorting networks.

### Search

The search operation allows the programmer to find a particular element within the stream, or possibly find neighbors of a specified element. The GPU is not used to speed up the search for an individual element, but instead is used to run multiple searches in parallel.Wikipedia:Citation needed

### Data structures

A variety of data structures can be represented on the GPU:

- Dense arrays
- Sparse arrays – static or dynamic
- Adaptive structures (union type)

# Applications

## Research: Higher Education and Supercomputing

### Computational Chemistry and Biology

### Bioinformatics[14]

| Application | Description | Supported Features | Expected Speed Up† | GPU‡ | Multi-GPU Support | Release Status |
|---|---|---|---|---|---|---|
| BarraCUDA | Sequence mapping software | Alignment of short sequencing reads | 6–10x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 0.6.2 |
| CUDASW++ | Open source software for Smith-Waterman protein database searches on GPUs | Parallel search of Smith-Waterman database | 10–50x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 2.0.8 |
| CUSHAW | Parallelized short read aligner | Parallel, accurate long read aligner – gapped alignments to large genomes | 10x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 1.0.40 |
| GPU-BLAST | Local search with fast *k*-tuple heuristic | Protein alignment according to blastp, multi CPU threads | 3–4x | T 2075, 2090, K10, K20, K20X | Single only | Available now Version 2.2.26 |
| GPU-HMMER | Parallelized local and global search with profile Hidden Markov models | Parallel local and global search of Hidden Markov Models | 60–100x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 2.3.2 |
| mCUDA-MEME | Ultrafast scalable motif discovery algorithm based on MEME | Scalable motif discovery algorithm based on MEME | 4–10x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 3.0.12 |
| SeqNFind | A GPU Accelerated Sequence Analysis Toolset | Reference assembly, blast, Smith–Waterman, hmm, de novo assembly | 400x | T 2075, 2090, K10, K20, K20X | Yes | Available now |
| UGENE | Opensource Smith–Waterman for SSE/CUDA, Suffix array based repeats finder and dotplot | Fast short read alignment | 6–8x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 1.11 |
| WideLM | Fits numerous linear models to a fixed design and response | Parallel linear regression on multiple similarly-shaped models | 150x | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 0.1-1 |

### Molecular Dynamics

| Application | Description | Supported Features | Expected Speed Up† | GPU‡ | Multi-GPU Support | Release Status |
|---|---|---|---|---|---|---|
| Abalone | Models molecular dynamics of biopolymers for simulations of proteins, DNA and ligands | Explicit and implicit solvent, Hybrid Monte Carlo | 4–29x | T 2075, 2090, K10, K20, K20X | Single Only | Available now Version 1.8.48 |
| ACEMD | GPU simulation of molecular mechanics force fields, implicit and explicit solvent | Written for use on GPUs | 160 ns/day GPU version only | T 2075, 2090, K10, K20, K20X | Yes | Available now |
| AMBER | Suite of programs to simulate molecular dynamics on biomolecule | PMEMD: explicit and implicit solvent | 89.44 ns/day JAC NVE | T 2075, 2090, K10, K20, K20X | Yes | Available now Version 12 + bugfix9 |
| DL-POLY | Simulate macromolecules, polymers, ionic systems, etc. on a distributed memory parallel computer | Two-body forces, Link-cell pairs, Ewald SPME forces, Shake VV | 4x | T 2075, 2090, K10, K20, K20X | Yes | Available now, Version 4.0 source only |
| CHARMM | MD package to simulate molecular dynamics on biomolecule. | Implicit (5x), Explicit (2x) Solvent via OpenMM | TBD | T 2075, 2090, K10, K20, K20X | Yes | In development Q4/12 |
| GROMACS | Simulation of biochemical molecules with complicated bond interactions | Implicit (5x), Explicit (2x) solvent | 165 ns/Day DHFR | T 2075, 2090, K10, K20, K20X | Single only | Available now version 4.6 in Q4/12 |
| HOOMD-Blue | Particle dynamics package written grounds up for GPUs | Written for GPUs | 2x | T 2075, 2090, K10, K20, K20X | Yes | Available now |
| LAMMPS | Classical molecular dynamics package | Lennard-Jones, Morse, Buckingham, CHARMM, Tabulated, Course grain SDK, Anisotropic Gay-Bern, RE-squared, “Hybrid”combinations | 3–18x | T 2075, 2090, K10, K20, K20X | Yes | Available now |
| NAMD | Designed for high-performance simulation of large molecular systems | 100M atom capable | 6.44 ns/days STMV 585x 2050s | T 2075, 2090, K10, K20, K20X | Yes | Available now, version 2.9 |
| OpenMM | Library and application for molecular dynamics for HPC with GPUs | Implicit and explicit solvent, custom forces | Implicit: 127–213 ns/day; Explicit: 18–55 ns/day DHFR | T 2075, 2090, K10, K20, K20X | Yes | Available now version 4.1.1 |

†Expected speedups are highly dependent on system configuration. GPU performance compared against multi-core x86 CPU socket. GPU performance benchmarked on GPU supported features and may be a kernel to kernel performance comparison. For details on configuration used, view application website. Speedups as per Nvidia in-house testing or ISV's documentation .

‡ Q=Quadro GPU, T=Tesla GPU. Nvidia recommended GPUs for this application. Please check with developer / ISV to obtain certification information.

The following are some of the areas where GPUs have been used for general purpose computing:

- MATLAB acceleration using the Parallel Computing Toolbox and MATLAB Distributed Computing Server, as well as third-party packages like Jacket.
- Computer clusters or a variation of a parallel computing (utilizing GPU cluster technology) for highly calculation-intensive tasks:
  - High-performance computing clusters (HPC clusters) (often termed supercomputers)
    - including cluster technologies like Message Passing Interface, and single-system image (SSI), distributed computing, and Beowulf
  - Grid computing (a form of distributed computing) (networking many heterogeneous computers to create a virtual computer architecture)
  - Load-balancing clusters (sometimes termed a server farm)
- Physical based simulation and physics engines (usually based on Newtonian physics models)
  - Conway's Game of Life, cloth simulation, incompressible fluid flow by solution of Navier–Stokes equations
- Statistical physics
  - Ising model
- Lattice gauge theory
- Segmentation – 2D and 3D
- Level-set methods
- CT reconstruction
- Fast Fourier transform
- k-nearest neighbor algorithm[15]
- fuzzy logic[16]
- Tone mapping
- Audio signal processing
  - Audio and Sound Effects Processing, to use a GPU for DSP (digital signal processing)
  - Analog signal processing
  - Speech processing
- Digital image processing
- Video processing
  - Hardware accelerated video decoding and post-processing
    - Motion compensation (mo comp)
    - Inverse discrete cosine transform (iDCT)
    - Variable-length decoding (VLD)
    - Inverse quantization (IQ)
    - In-loop deblocking
    - Bitstream processing (CAVLC/CABAC) using special purpose hardware for this task because this is a serial task not suitable for regular GPGPU computation
    - Deinterlacing

    - Spatial-temporal de-interlacing
  - Noise reduction
  - Edge enhancement
  - Color correction
- Hardware accelerated video encoding and pre-processing
- Global illumination – ray tracing, photon mapping, radiosity among others, subsurface scattering
- Geometric computing – constructive solid geometry, distance fields, collision detection, transparency computation, shadow generation
- Scientific computing
  - Monte Carlo simulation of light propagation[17]
  - Weather forecasting
  - Climate research
  - Molecular modeling on GPU
  - Quantum mechanical physics
  - Astrophysics[18]
- Bioinformatics[19]
- Computational finance
- Medical imaging
- Computer vision
- Digital signal processing / signal processing
- Control engineering
- Neural networks
- Database operations[20]
- Lattice Boltzmann methods
- Cryptography and cryptanalysis
  - Implementation of MD6
  - Implementation of AES[21][22]
  - Implementation of DES
  - Implementation of RSA[23]
  - Implementation of ECC
  - Password cracking
- Electronic Design Automation
- Antivirus software[24]
- Intrusion detection[25][26]
- Increase computing power for distributed computing project like SETI@home, Einstein@home

## References

[1] Fung etal, "Mediated Reality Using Computer Graphics Hardware for Computer Vision" (http://www.eyetap.org/~fungja/glorbits_final.pdf), Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002), Seattle, Washington, USA, 7–10 Oct 2002, pp. 83–89.

[2] An EyeTap video-based featureless projective motion estimation assisted by gyroscopic tracking for wearable computer mediated reality, ACM Personal and Ubiquitous Computing published by Springer Verlag, Vol.7, Iss. 3, 2003.

[3] "Computer Vision Signal Processing on Graphics Processing Units", Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004) (http://www.eyetap.org/papers/docs/procicassp2004.pdf): Montreal, Quebec, Canada, 17–21 May 2004, pp. V-93 – V-96

[4] "Using Multiple Graphics Cards as a General Purpose Parallel Computer: Applications to Computer Vision", Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004) (http://eyetap.org/papers/docs/procicpr2004.pdf), Cambridge, United Kingdom, 23–26 August 2004, volume 1, pages 805–808.

[5] http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html "As the two major programming frameworks for GPU computing, OpenCL and CUDA have been competing for mindshare in the developer community for the past few years."

[6] Mapping computational concepts to GPUs (http://doi.acm.org/10.1145/1198555.1198768): Mark Harris. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses (Los Angeles, California, 31 July – 4 August 2005). J. Fujii, Ed. SIGGRAPH '05. ACM Press, New York, NY, 50.

[7] Double precision on GPUs (Proceedings of ASIM 2005) (http://www.mathematik.uni-dortmund.de/lsiii/static/showpdffile_GoeddekeStrzodkaTurek2005.pdf): Dominik Goddeke, Robert Strzodka, and Stefan Turek. Accelerating Double Precision (FEM) Simulations with (GPUs). Proceedings of ASIM 2005 – 18th Symposium on Simulation Technique, 2005.

[8] James Fung, Steve Mann, Chris Aimone, "OpenVIDIA: Parallel GPU Computer Vision", Proceedings of the ACM Multimedia 2005, Singapore, 6–11 Nov. 2005, pages 849–852

[9] (http://www.khronos.org/opencl/):OpenCL at the Khronos Group

[10] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM 52(10) (2009) 56–67

[11] GPU Gems – Chapter 34, GPU Flow-Control Idioms (http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter34.html)

[12] (http://www.futurechips.org/tips-for-power-coders/basic-technique-to-help-branch-prediction.html): Future Chips. "Tutorial on removing branches", 2011

[13] GPGPU survey paper (http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=907): John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". Computer Graphics Forum, volume 26, number 1, 2007, pp. 80–113.

[14] http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf

[15] Fast *k*-nearest neighbor search using GPU. In Proceedings of the CVPR Workshop on Computer Vision on GPU, Anchorage, Alaska, USA, June 2008. V. Garcia and E. Debreuve and M. Barlaud.

[16] M. Cococcioni, R. Grasso, M. Rixen, *Rapid prototyping of high performance fuzzy computing applications using high level GPU programming for maritime operations support*, in Proceedings of the 2011 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), Paris, 11–15 April 2011

[17] E. Alerstam, T. Svensson & S. Andersson-Engels, "Parallel computing with graphics processing units for high speed Monte Carlo simulation of photon migration" (http://www.atomic.physics.lu.se/fileadmin/atomfysik/Biophotonics/Publications/Alerstam2008_JBOLetters.pdf), *J. Biomedical Optics* **13**, 060504 (2008) (http://dx.doi.org/10.1117/1.3041496)

[18] http://www.astro.lu.se/compugpu2010/

[19] Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A. (2007) High-throughput sequence alignment using Graphics Processing Units. (http://www.biomedcentral.com/1471-2105/8/474/) BMC Bioinformatics 8:474.

[20] GPU-based Sorting in PostgreSQL (http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/ngm/15-823/project/Final.pdf) Naju Mancheril, School of Computer Science – Carnegie Mellon University

[21] AES on SM3.0 compliant GPUs. (http://www.springerlink.com/content/v8010ju818508326/?p=fb25e7d2cdb94147885cfda7fdc6509d) Owen Harrison, John Waldron, AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In proceedings of CHES 2007.

[22] AES and modes of operations on SM4.0 compliant GPUs. (http://www.usenix.org/events/sec08/tech/harrison.html) Owen Harrison, John Waldron, Practical Symmetric Key Cryptography on Modern Graphics Hardware. In proceedings of USENIX Security 2008.

[23] RSA on SM4.0 compliant GPUs. (http://www.springerlink.com/content/v83j50l12p7446v2/?p=4faeca75397543a5a70ac3586840bcc2&pi=21) Owen Harrison, John Waldron, Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In proceedings of AfricaCrypt 2009.

[24] GrAVity: A Massively Parallel Antivirus Engine (http://www.ics.forth.gr/dcs/Activities/papers/gravity-raid10.pdf). Giorgos Vasiliadis and Sotiris Ioannidis, GrAVity: A Massively Parallel Antivirus Engine. In proceedings of RAID 2010.

[25] Gnort: High Performance Network Intrusion Detection Using Graphics Processors (http://www.ics.forth.gr/dcs/Activities/papers/gnort.raid08.pdf). Giorgos Vasiliadis et al, Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In proceedings of RAID 2008.

[26] Regular Expression Matching on Graphics Hardware for Intrusion Detection (http://www.ics.forth.gr/dcs/Activities/papers/gnort-regexp.raid09.pdf). Giorgos Vasiliadis et al, Regular Expression Matching on Graphics Hardware for Intrusion Detection. In proceedings of RAID 2009.

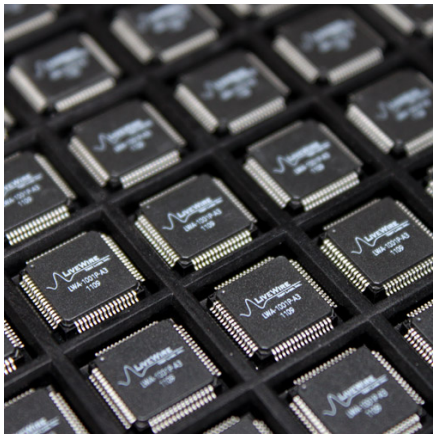
## External links

- openhmpp.org – New Open Standard for Many-Core (http://www.openhmpp.org/)
- OCLTools (http://www.clusterchimps.org/ocltools.html) Open Source OpenCL Compiler and Linker
- GPGPU.org – General-Purpose Computation Using Graphics Hardware (http://www.gpgpu.org/)
- GPGPU Wiki (http://www.gpgpu.org/w/index.php/Main_Page)
- SIGGRAPH 2005 GPGPU Course Notes (http://www.gpgpu.org/s2005)
- IEEE VIS 2005 GPGPU Course Notes (http://www.gpgpu.org/vis2005)
- NVIDIA Developer Zone (http://developer.nvidia.com)
- AMD GPU Tools (http://developer.amd.com/GPU/Pages/default.aspx)
- CPU vs. GPGPU (http://www.futurechips.org/chip-design-for-all/cpu-vs-gpgpu.html)
- What is GPU Computing? (http://www.nvidia.com/object/GPU_Computing.html)
- Tech Report article: "ATI stakes claims on physics, GPGPU ground" (http://techreport.com/onearticle.x/8887) by Scott Wasson
- GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model – porting a standard model to GPU hardware (http://dx.doi.org/10.1016/j.jcp.2009.03.018)
- GPGPU Computing @ Duke Statistical Science (http://www.stat.duke.edu/gpustatsci/)
- GPGPU Programming in F# (http://blogs.msdn.com/b/satnam_singh/archive/2009/12/15/gpgpu-and-x64-multicore-programming-with-accelerator-from-f.aspx) using the Microsoft Research Accelerator system.
- GPGPU Review (http://tobiaspreis.de/publications/p_epjst_2011_econophysics.pdf), Tobias Preis, European Physical Journal Special Topics **194**, 87–119 (2011)

# Application-specific integrated circuit

"ASIC" redirects here. For other uses, see ASIC (disambiguation).

An **application-specific integrated circuit** (**ASIC**) /ˈeɪsɪk/, is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. For example, a chip designed to run in a digital voice recorder or a high efficiency Bitcoin miner is an ASIC. Application-specific standard products (ASSPs) are intermediate between ASICs and industry standard integrated circuits like the 7400 or the 4000 series.



A tray of Application-specific integrated circuit (ASIC) chips.

As feature sizes have shrunk and design tools improved over the years, the maximum complexity (and hence functionality) possible in an ASIC has grown from 5,000 gates to over 100 million. Modern ASICs often include entire microprocessors, memory blocks including ROM, RAM, EEPROM, flash memory and other large building blocks. Such an ASIC is often termed a SoC (system-on-chip). Designers of digital ASICs often use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs.

Field-programmable gate arrays (FPGA) are the modern-day technology for building a breadboard or prototype from standard parts; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications. For smaller designs and/or lower production volumes, FPGAs may be more cost effective than an ASIC design even in production. The non-recurring engineering (NRE) cost of an ASIC can run into the millions of dollars.

## History

The initial ASICs used gate array technology. Ferranti produced perhaps the first gate-array, the ULA (Uncommitted Logic Array), around 1980. An early successful commercial application was the ULA circuitry found in the 8-bit ZX81 and ZX Spectrum low-end personal computers, introduced in 1981 and 1982. These were used by Sinclair Research (UK) essentially as a low-cost I/O solution aimed at handling the computer's graphics. Some versions of ZX81/Timex Sinclair 1000 used just four chips (ULA, 2Kx8 RAM, 8Kx8 ROM, Z80A CPU) to implement an entire mass-market personal computer with built-in BASIC interpreter.

Customization occurred by varying the metal interconnect mask. ULAs had complexities of up to a few thousand gates. Later versions became more generalized, with different base dies customised by both metal and polysilicon layers. Some base dies include RAM elements.

## Standard-cell designs

Main article: standard cell

In the mid-1980s, a designer would choose an ASIC manufacturer and implement their design using the design tools available from the manufacturer. While third-party design tools were available, there was not an effective link from the third-party design tools to the layout and actual semiconductor process performance characteristics of the various ASIC manufacturers. Most designers ended up using factory-specific tools to complete the implementation of their designs. A solution to this problem, which also yielded a much higher density device, was the implementation of standard cells. Every ASIC manufacturer could create functional blocks with known electrical characteristics, such as propagation delay, capacitance and inductance, that could also be represented in third-party tools. Standard-cell design is the utilization of these functional blocks to achieve very high gate density and good electrical performance. Standard-cell design fits between Gate Array and Full Custom design in terms of both its non-recurring engineering and recurring component cost.

By the late 1990s, logic synthesis tools became available. Such tools could compile HDL descriptions into a gate-level netlist. Standard-cell integrated circuits (ICs) are designed in the following conceptual stages, although these stages overlap significantly in practice.

1. A team of design engineers starts with a non-formal understanding of the required functions for a new ASIC, usually derived from requirements analysis.
2. The design team constructs a description of an ASIC (application specific integrated circuits) to achieve these goals using an HDL. This process is analogous to writing a computer program in a high-level language. This is usually called the RTL (register-transfer level) design.
3. Suitability for purpose is verified by functional verification. This may include such techniques as logic simulation, formal verification, emulation, or creating an equivalent pure software model (see Simics, for example). Each technique has advantages and disadvantages, and often several methods are used.
4. Logic synthesis transforms the RTL design into a large collection of lower-level constructs called standard cells. These constructs are taken from a standard-cell library consisting of pre-characterized collections of gates (such as 2 input nor, 2 input nand, inverters, etc.). The standard cells are typically specific to the planned manufacturer of the ASIC. The resulting collection of standard cells, plus the needed electrical connections between them, is called a gate-level netlist.
5. The gate-level netlist is next processed by a placement tool which places the standard cells onto a region representing the final ASIC. It attempts to find a placement of the standard cells, subject to a variety of specified constraints.
6. The routing tool takes the physical placement of the standard cells and uses the netlist to create the electrical connections between them. Since the search space is large, this process will produce a “sufficient” rather than “globally optimal” solution. The output is a file which can be used to create a set of photomasks enabling a

semiconductor fabrication facility (commonly called a 'fab') to produce physical ICs.

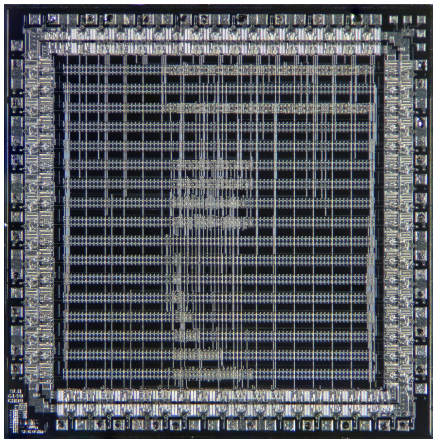
7. Given the final layout, circuit extraction computes the parasitic resistances and capacitances. In the case of a digital circuit, this will then be further mapped into delay information, from which the circuit performance can be estimated, usually by static timing analysis. This, and other final tests such as design rule checking and power analysis (collectively called signoff) are intended to ensure that the device will function correctly over all extremes of the process, voltage and temperature. When this testing is complete the photomask information is released for chip fabrication.

These steps, implemented with a level of skill common in the industry, almost always produce a final device that correctly implements the original design, unless flaws are later introduced by the physical fabrication process.

The design steps (or flow) are also common to standard product design. The significant difference is that standard-cell design uses the manufacturer's cell libraries that have been used in potentially hundreds of other design implementations and therefore are of much lower risk than full custom design. Standard cells produce a design density that is cost effective, and they can also integrate IP cores and SRAM (Static Random Access Memory) effectively, unlike Gate Arrays.

## Gate-array design

Gate-array design is a manufacturing method in which the diffused layers, i.e. transistors and other active devices, are predefined and wafers containing such devices are held in stock prior to metallization—in other words, unconnected. The physical design process then defines the interconnections of the final device. For most ASIC manufacturers, this consists of from two to as many as nine metal layers, each metal layer running perpendicular to the one below it. Non-recurring engineering costs are much lower, as photolithographic masks are required only for the metal layers, and production cycles are much shorter, as metallization is a comparatively quick process.



Microscope photograph of a gate-array ASIC showing the predefined logic cells and custom interconnections. This particular design uses less than 20% of available logic gates.

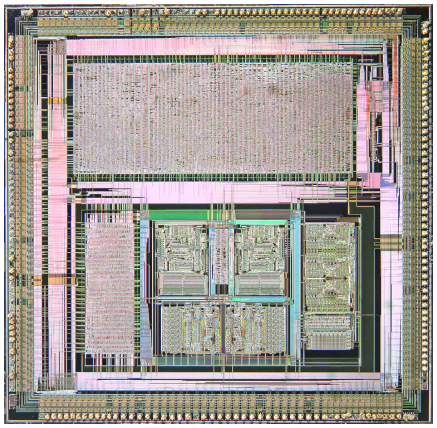
Gate-array ASICs are always a compromise as mapping a given design onto what a manufacturer held as a stock wafer never gives 100% utilization. Often difficulties in routing the interconnect require migration onto a larger array device with consequent increase in the piece part price. These difficulties are often a result of the layout software used to develop the interconnect.

Pure, logic-only gate-array design is rarely implemented by circuit designers today, having been replaced almost entirely by field-programmable devices, such as field-programmable gate arrays (FPGAs), which can be programmed by the user and thus offer minimal tooling charges non-recurring engineering, only marginally increased piece part cost, and comparable performance. Today, gate arrays are evolving into structured ASICs that consist of a large IP core like a CPU, DSP unit, peripherals, standard interfaces, integrated memories SRAM, and a block of reconfigurable, uncommited logic. This shift is largely because ASIC devices are capable of integrating such large blocks of system functionality and "system-on-a-chip" requires far more than just logic blocks.

In their frequent usages in the field, the terms "gate array" and "semi-custom" are synonymous. Process engineers more commonly use the term "semi-custom", while "gate-array" is more commonly used by logic (or gate-level) designers.

## Full-custom design

Main article: Full custom



Microscope photograph of custom ASIC (486 chipset) showing gate-based design on top and custom circuitry on bottom.

By contrast, full-custom ASIC design defines all the photolithographic layers of the device. Full-custom design is used for both ASIC design and for standard product design.

The benefits of full-custom design usually include reduced area (and therefore recurring component cost), performance improvements, and also the ability to integrate analog components and other pre-designed — and thus fully verified — components, such as microprocessor cores that form a system-on-chip.

The disadvantages of full-custom design can include increased manufacturing and design time, increased non-recurring engineering costs, more complexity in the computer-aided design (CAD) system, and a much higher skill requirement on the part of the design team.

For digital-only designs, however, "standard-cell" cell libraries, together with modern CAD systems, can offer considerable performance/cost benefits with low risk. Automated layout tools are quick and easy to use and also offer the possibility to "hand-tweak" or manually optimize any performance-limiting aspect of the design.

This is designed by using basic logic gates, circuits or layout specially for a design.

## Structured design

Main article: Structured ASIC platform

Structured ASIC design (also referred to as "platform ASIC design"), is a relatively new term in the industry, resulting in some variation in its definition. However, the basic premise of a structured ASIC is that both manufacturing cycle time and design cycle time are reduced compared to cell-based ASIC, by virtue of there being pre-defined metal layers (thus reducing manufacturing time) and pre-characterization of what is on the silicon (thus reducing design cycle time). One definition states that

> In a "structured ASIC" design, the logic mask-layers of a device are predefined by the ASIC vendor (or in some cases by a third party). Design differentiation and customization is achieved by creating custom metal layers that create custom connections between predefined lower-layer logic elements. "Structured ASIC" technology is seen as bridging the gap between field-programmable gate arrays and "standard-cell" ASIC designs. Because only a small number of chip layers must be custom-produced, "structured ASIC" designs have much smaller non-recurring expenditures (NRE) than "standard-cell" or "full-custom" chips, which require that a full mask set be produced for every design.Wikipedia:Citation needed

This is effectively the same definition as a gate array. What makes a structured ASIC different is that in a gate array, the predefined metal layers serve to make manufacturing turnaround faster. In a structured ASIC, the use of predefined metallization is primarily to reduce cost of the mask sets as well as making the design cycle time significantly shorter. For example, in a cell-based or gate-array design the user must often design power, clock, and test structures themselves; these are predefined in most structured ASICs and therefore can save time and expense for the designer compared to gate-array. Likewise, the design tools used for structured ASIC can be substantially lower cost and easier (faster) to use than cell-based tools, because they do not have to perform all the functions that cell-based tools do. In some cases, the structured ASIC vendor requires that customized tools for their device (e.g., custom physical synthesis) be used, also allowing for the design to be brought into manufacturing more quickly.

## Cell libraries, IP-based design, hard and soft macros

Cell libraries of logical primitives are usually provided by the device manufacturer as part of the service. Although they will incur no additional cost, their release will be covered by the terms of a non-disclosure agreement (NDA) and they will be regarded as intellectual property by the manufacturer. Usually their physical design will be pre-defined so they could be termed "hard macros".

What most engineers understand as "intellectual property" are IP cores, designs purchased from a third-party as sub-components of a larger ASIC. They may be provided as an HDL description (often termed a "soft macro"), or as a fully routed design that could be printed directly onto an ASIC's mask (often termed a hard macro). Many organizations now sell such pre-designed cores — CPUs, Ethernet, USB or telephone interfaces — and larger organizations may have an entire department or division to produce cores for the rest of the organization. Indeed, the wide range of functions now available is a result of the phenomenal improvement in electronics in the late 1990s and early 2000s; as a core takes a lot of time and investment to create, its re-use and further development cuts product cycle times dramatically and creates better products. Additionally, organizations such as OpenCores are collecting free IP cores, paralleling the open source software movement in hardware design.

Soft macros are often process-independent, i.e., they can be fabricated on a wide range of manufacturing processes and different manufacturers. Hard macros are process-limited and usually further design effort must be invested to migrate (port) to a different process or manufacturer.

## Multi-project wafers

Some manufacturers offer multi-project wafers (MPW) as a method of obtaining low cost prototypes. Often called shuttles, these MPW, containing several designs, run at regular, scheduled intervals on a "cut and go" basis, usually with very little liability on the part of the manufacturer. The contract involves the assembly and packaging of a handful of devices. The service usually involves the supply of a physical design data base i.e. masking information or Pattern Generation (PG) tape. The manufacturer is often referred to as a "silicon foundry" due to the low involvement it has in the process.

## References

## Sources


- Kevin Morris (23 November 2003). "Cost-Reduction Quagmire: Structured ASIC and Other Options" (http://www.fpgajournal.com/articles/20041123_quagmire.htm). *FPGA and Programmable Logic Journal*.
- Anthony Cataldo (26 March 2002). "Xilinx looks to ease path to custom FPGAs" (http://www.eetimes.com/story/OEG20020325S0060). *EE Times* (CMP Media, LLC).
- "Xilinx intros next-gen EasyPath FPGAs priced below structured ASICs" (http://www.thefreelibrary.com/Xilinx+intros+next-gen+EasyPath+FPGAs+priced+below+structured+ASICs.-a0125948398). *EDP Weekly's IT Monitor* (Millin Publishing, Inc.). 18 October 2004.
- Golshan, K. (2007). *Physical design essentials: an ASIC design implementation perspective.* New York: Springer. ISBN 0-387-36642-3.

# Vector processor

"Array processor" redirects here. It is not to be confused with Array processing.

A **vector processor**, or **array processor**, is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called *vectors*. This is in contrast to a scalar processor, whose instructions operate on single data items. Vector processors can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector machines appeared in the early 1970s and dominated supercomputer design through the 1970s into the 90s, notably the various Cray platforms. The rapid rise in the price-to-performance ratio of conventional microprocessor designs led to the vector supercomputer's demise in the later 1990s.

Today, most commodity CPUs implement architectures that feature instructions for a form of vector processing on multiple (vectorized) data sets, typically known as SIMD (**S**ingle **I**nstruction, **M**ultiple **D**ata). Common examples include VIS, MMX, SSE, AltiVec and AVX. Vector processing techniques are also found in video game console hardware and graphics accelerators. In 2000, IBM, Toshiba and Sony collaborated to create the Cell processor, consisting of one scalar processor and eight vector processors, which found use in the Sony PlayStation 3 among other applications.

Other CPU designs may include some multiple instructions for vector processing on multiple (vectorised) data sets, typically known as MIMD (**M**ultiple **I**nstruction, **M**ultiple **D**ata) and realized with VLIW. Such designs are usually dedicated to a particular application and not commonly marketed for general purpose computing. In the Fujitsu FR-V VLIW/*vector processor* both technologies are combined.

## History

### Early work

Vector processing development began in the early 1960s at Westinghouse in their **Solomon** project. Solomon's goal was to dramatically increase math performance by using a large number of simple math co-processors under the control of a single master CPU. The CPU fed a single common instruction to all of the arithmetic logic units (ALUs), one per "cycle", but with a different data point for each one to work on. This allowed the Solomon machine to apply a single algorithm to a large data set, fed in the form of an array.

In 1962, Westinghouse cancelled the project, but the effort was restarted at the University of Illinois as the ILLIAC IV. Their version of the design originally called for a 1 GFLOPS machine with 256 ALUs, but, when it was finally delivered in 1972, it had only 64 ALUs and could reach only 100 to 150 MFLOPS. Nevertheless it showed that the basic concept was sound, and, when used on data-intensive applications, such as computational fluid dynamics, the "failed" ILLIAC was the fastest machine in the world. The ILLIAC approach of using separate ALUs for each data element is not common to later designs, and is often referred to under a separate category, massively parallel computing.

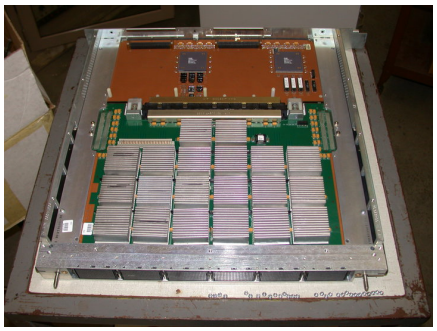
A computer for operations with functions was presented and developed by Kartsev in 1967.

### Supercomputers

The first *successful* implementation of vector processing appears to be the Control Data Corporation STAR-100 and the Texas Instruments Advanced Scientific Computer (ASC). The basic ASC (i.e., "one pipe") ALU used a pipeline architecture that supported both scalar and vector computations, with peak performance reaching approximately 20 MFLOPS, readily achieved when processing long vectors. Expanded ALU configurations supported "two pipes" or "four pipes" with a corresponding 2X or 4X performance gain. Memory bandwidth was sufficient to support these expanded modes. The STAR was otherwise slower than CDC's own supercomputers like the CDC 7600, but at data

related tasks they could keep up while being much smaller and less expensive. However the machine also took considerable time decoding the vector instructions and getting ready to run the process, so it required very specific data sets to work on before it actually sped anything up.

The vector technique was first fully exploited in 1976 by the famous Cray-1. Instead of leaving the data in memory like the STAR and ASC, the Cray design had eight "vector registers," which held sixty-four 64-bit words each. The vector instructions were applied between registers, which is much faster than talking to main memory. The Cray design used pipeline parallelism to implement vector instructions rather than multiple ALUs. In addition the design had completely separate pipelines for different instructions, for example, addition/subtraction was implemented in different hardware than multiplication. This allowed a batch of vector instructions themselves to be pipelined, a technique they called *vector chaining*. The Cray-1 normally had a performance of about 80 MFLOPS, but with up to three chains running it could peak at 240 MFLOPS – a respectable number even as of 2002.



Cray J90 processor module with four scalar/vector processors

Other examples followed. Control Data Corporation tried to re-enter the high-end market again with its ETA-10 machine, but it sold poorly and they took that as an opportunity to leave the supercomputing field entirely. In the early and mid-1980s Japanese companies (Fujitsu, Hitachi and Nippon Electric Corporation (NEC) introduced register-based vector machines similar to the Cray-1, typically being slightly faster and much smaller. Oregon-based Floating Point Systems (FPS) built add-on array processors for minicomputers, later building their own minisupercomputers. However Cray continued to be the performance leader, continually beating the competition with a series of machines that led to the Cray-2, Cray X-MP and Cray Y-MP. Since then, the supercomputer market has focused much more on massively parallel processing rather than better implementations of vector processors. However, recognising the benefits of vector processing IBM developed Virtual Vector Architecture for use in supercomputers coupling several scalar processors to act as a vector processor.

### SIMD

Vector processing techniques have since been added to almost all modern CPU designs, although they are typically referred to as SIMD. In these implementations, the vector unit runs beside the main scalar CPU, and is fed data from vector instruction aware programs.

## Description

In general terms, CPUs are able to manipulate one or two pieces of data at a time. For instance, most CPUs have an instruction that essentially says "add A to B and put the result in C". The data for A, B and C could be—in theory at least—encoded directly into the instruction. However, in efficient implementation things are rarely that simple. The data is rarely sent in raw form, and is instead "pointed to" by passing in an address to a memory location that holds the data. Decoding this address and getting the data out of the memory takes some time, during which the CPU traditionally would sit idle waiting for the requested data to show up. As CPU speeds have increased, this *memory latency* has historically become a large impediment to performance; see Memory wall.

In order to reduce the amount of time consumed by these steps, most modern CPUs use a technique known as instruction pipelining in which the instructions pass through several sub-units in turn. The first sub-unit reads the address and decodes it, the next "fetches" the values at those addresses, and the next does the math itself. With pipelining the "trick" is to start decoding the next instruction even before the first has left the CPU, in the fashion of an assembly line, so the address decoder is constantly in use. Any particular instruction takes the same amount of

time to complete, a time known as the *latency*, but the CPU can process an entire batch of operations much faster and more efficiently than if it did so one at a time.

Vector processors take this concept one step further. Instead of pipelining just the instructions, they also pipeline the data itself. The processor is fed instructions that say not just to add A to B, but to add all of the numbers "from here to here" to all of the numbers "from there to there". Instead of constantly having to decode instructions and then fetch the data needed to complete them, the processor reads a single instruction from memory, and it is simply implied in the definition of the instruction *itself* that the instruction will operate again on another item of data, at an address one increment larger than the last. This allows for significant savings in decoding time.

To illustrate what a difference this can make, consider the simple task of adding two groups of 10 numbers together. In a normal programming language one would write a "loop" that picked up each of the pairs of numbers in turn, and then added them. To the CPU, this would look something like this:

```
execute this loop 10 times
  read the next instruction and decode it
  fetch this number
  fetch that number
  add them
  put the result here
end loop
```

But to a vector processor, this task looks considerably different:

```
read instruction and decode it
fetch these 10 numbers
fetch those 10 numbers
add them
put the results here
```

There are several savings inherent in this approach. For one, only two address translations are needed. Depending on the architecture, this can represent a significant savings by itself. Another saving is fetching and decoding the instruction itself, which has to be done only one time instead of ten. The code itself is also smaller, which can lead to more efficient memory use.

But more than that, a vector processor may have multiple functional units adding those numbers in parallel. The checking of dependencies between those numbers is not required as a vector instruction specifies multiple independent operations. This simplifies the control logic required, and can improve performance by avoiding stalls.

As mentioned earlier, the Cray implementations took this a step further, allowing several different types of operations to be carried out at the same time. Consider code that adds two numbers and then multiplies by a third; in the Cray, these would all be fetched at once, and both added and multiplied in a single operation. Using the pseudocode above, the Cray did:

```
read instruction and decode it
fetch these 10 numbers
fetch those 10 numbers
fetch another 10 numbers
add and multiply them
put the results here
```

The math operations thus completed far faster overall, the limiting factor being the time required to fetch the data from memory.

Not all problems can be attacked with this sort of solution. Adding these sorts of instructions necessarily adds complexity to the core CPU. That complexity typically makes *other* instructions run slower—i.e., whenever it is **not** adding up many numbers in a row. The more complex instructions also add to the complexity of the decoders, which might slow down the decoding of the more common instructions such as normal adding.

In fact, vector processors work best only when there are large amounts of data to be worked on. For this reason, these sorts of CPUs were found primarily in supercomputers, as the supercomputers themselves were, in general, found in places such as weather prediction centres and physics labs, where huge amounts of data are "crunched".

## Real world example: vector instructions usage with the x86 architecture

Shown below is an actual x86 architecture example for vector instruction usage with the SSE instruction set. The example multiplies two arrays of single precision floating point numbers. It's written in the C language with inline assembly code parts for compilation with GCC (32bit).

```
//SSE simd function for vectorized multiplication of 2 arrays with
single-precision floatingpoint numbers
//1st param pointer on source/destination array, 2nd param 2. source
array, 3rd param number of floats per array
 void mul_asm(float* out, float* in, unsigned int leng)
 {    unsigned int count, rest;

      //compute if array is big enough for vector operation
      rest  = (leng*4)%16;
      count = (leng*4)-rest;

     // vectorized part; 4 floats per loop iteration
      if (count>0){
      __asm __volatile__  (".intel_syntax noprefix\n\t"
      "loop:                  \n\t"
      "movups xmm0,[ebx+ecx] ;loads 4 floats in first register
(xmm0)\n\t"
      "movups xmm1,[eax+ecx] ;loads 4 floats in second register
(xmm1)\n\t"
      "mulps xmm0,xmm1       ;multiplies both vector registers\n\t"
      "movups [eax+ecx],xmm0 ;write back the result to memory\n\t"
      "sub ecx,16            ;increase address pointer by 4 floats\n\t"
      "jnz loop              \n\t"
      ".att_syntax prefix    \n\t"
        : : "a" (out), "b" (in), "c"(count), "d"(rest): "xmm0","xmm1");
      }

      // scalar part; 1 float per loop iteration
      if (rest!=0)
      {
       __asm __volatile__  (".intel_syntax noprefix\n\t"
      "add eax,ecx           \n\t"
      "add ebx,ecx           \n\t"
```

```
      "rest:                  \n\t"
      "movss xmm0,[ebx+edx]  ;load 1 float in first register
(xmm0)\n\t"
      "movss xmm1,[eax+edx]  ;load 1 float in second register
(xmm1)\n\t"
      "mulss xmm0,xmm1       ;multiplies both scalar parts of
registers\n\t"
      "movss [eax+edx],xmm0  ;write back the result\n\t"
      "sub edx,4             \n\t"
      "jnz rest              \n\t"
      ".att_syntax prefix    \n\t"
        : : "a" (out), "b" (in), "c"(count), "d"(rest): "xmm0","xmm1");
      }
      return;
 }
```

## Programming heterogeneous computing architectures

Various machines were designed to include both traditional processors and vector processors, such as the Fujitsu AP1000 and AP3000. Programming such heterogeneous machines can be difficult since developing programs that make best use of characteristics of different processors increases the programmer's burden. It increases code complexity and decreases portability of the code by requiring hardware specific code to be interleaved throughout application code. Balancing the application workload across processors can be problematic, especially given that they typically have different performance characteristics. There are different conceptual models to deal with the problem, for example using a coordination language and program building blocks (programming libraries or higher order functions). Each block can have a different native implementation for each processor type. Users simply program using these abstractions and an intelligent compiler chooses the best implementation based on the context.

## References

## External links

- The History of the Development of Parallel Computing (http://ei.cs.vt.edu/~history/Parallel.html) (from 1955 to 1993)

# List of concurrent and parallel programming languages

This article lists concurrent and parallel programming languages, categorising them by a defining paradigm. A concurrent language is defined as one which uses the concept of simultaneously executing processes or threads of execution as a means of structuring a program. A parallel language is able to express programs that are executable on more than one processor. Both types are listed as concurrency is a useful tool in expressing parallelism, but it is not necessary. In both cases, the features must be part of the language syntax and not an extension such as a library.

The following categories aim to capture the main, defining feature of the languages contained, but they are not necessarily orthogonal.

## Actor model

Main article: Actor model

- Axum - a domain-specific language being developed by Microsoft.
- Elixir (runs on the Erlang VM)
- Erlang
- Janus
- SALSA
- Scala
- Smalltalk

## Coordination languages

- Glenda
- Linda coordination language
- Millipede
- CnC (Concurrent Collections)

## Dataflow

Main article: Dataflow programming

- E (also object-oriented)
- Joule (also distributed)
- SISAL
- Lustre (also synchronous)
- Signal (also synchronous)
- LabView (also synchronous)
- CAL

## Distributed

Main article: Distributed computing

- Limbo
- Oz - Multi-paradigm language with particular support for constraint and distributed programming.
- Sequoia
- Bloom
- Julia
- SR
- MPD

## Event-driven and hardware description

Main article: Event-driven programming

Main article: Hardware Description Language

- Verilog
- VHDL
- SystemC
- Verilog-AMS - math modeling of continuous time systems (#true concurrency)
- Esterel (also synchronous)

## Functional

Main article: Functional programming

- Concurrent Haskell
- Concurrent ML
- Clojure
- Elixir
- Erlang
- Id
- SequenceL
- MultiLisp

## GPU languages

Main article: General-Purpose Computing on Graphics Processing Units

- CUDA
- OpenCL
- OpenHMPP

## Logic programming

Main article: Logic programming

- Parlog

## Multi-threaded

Main article: Thread (computer science) § Multithreading

- C= [1]
- Cilk
- Cilk Plus
- Clojure
- Fork - Programming language for the PRAM model.
- Go
- Java
- ParaSail
- SequenceL

## Object-oriented

Main article: Object-orientated programming

- Ada
- C*
- C++ AMP
- Charm++
- D Programming Language
- Eiffel SCOOP (Simple Concurrent Object-Oriented Programming)
- Emerald (also distributed)
- Java
- ParaSail
- Smalltalk
- μC++

## Partitioned global address space (PGAS)

Main article: Partitioned global address space

- Chapel
- Coarray Fortran
- Fortress
- High Performance Fortran
- Titanium
- Unified Parallel C
- X10
- ZPL

## Process Calculi

Main article: Process calculus

- Occam
- Occam-π
- JoCaml
- Join Java
- Limbo
- Alef
- Ateji PX - An extension of Java with parallel primitives inspired from pi-calculus.
- Concurrent Pascal
- Ease
- FortranM
- Limbo (also distributed)
- Joyce
- Newsqueak
- PyCSP [2]
- SuperPascal
- XC - A C-based language developed by XMOS.

## APIs

These application programming interfaces support parallelism in host languages.

- OpenMP for C, C++, and Fortran (shared memory and attached GPUs)

## Unsorted

- NESL
- Orc
- ABCPL
- ActorScript
- Afnix
- APL
- Curry
- DAPPLE
- Emrald
- LabView
- Modula-3
- Nimrod
- Pict
- Polaris
- Reia
- Rust
- SR
- Stackless
- Zounds
- PCN

## References

[1] http://www.hoopoesnest.com/cstripes/cstripes-sketch.htm
[2] https://code.google.com/p/pycsp/

# Actor model

The **actor model** in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. The actor model originated in 1973. It has been used both as a framework for a theoretical understanding of computation, and as the theoretical basis for several practical implementations of concurrent systems. The relationship of the model to other work is discussed in Indeterminacy in concurrent computation and Actor model and process calculi.

## History

Main article: History of the Actor model

According to Carl Hewitt, unlike previous models of computation, the Actor model was inspired by physics including general relativity and quantum mechanics. It was also influenced by the programming languages Lisp, Simula and early versions of Smalltalk, as well as capability-based systems and packet switching. Its development was "motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network." Since that time, the advent of massive concurrency through multi-core computer architectures has revived interest in the Actor model.

Following Hewitt, Bishop, and Steiger's 1973 publication, Irene Greif developed an operational semantics for the Actors model as part of her doctoral research. Two years later, Henry Baker and Hewitt published a set of axiomatic laws for Actor systems. Other major milestones include William Clinger's 1981 dissertation introducing a denotational semantics based on power domains and Gul Agha's 1985 dissertation which further developed a transition-based semantic model complementary to Clinger's. This resulted in the full development of actor model theory.

Major software implementation work was done by Russ Atkinson, Giuseppe Attardi, Henry Baker, Gerry Barber, Peter Bishop, Peter de Jong, Ken Kahn, Henry Lieberman, Carl Manning, Tom Reinhardt, Richard Steiger and Dan Theriault in the Message Passing Semantics Group at Massachusetts Institute of Technology (MIT). Research groups led by Chuck Seitz at California Institute of Technology (Caltech) and Bill Dally at MIT constructed computer architectures that further developed the message passing in the model. See Actor model implementation.

Research on the Actor model has been carried out at California Institute of Technology, Kyoto University Tokoro Laboratory, MCC, MIT Artificial Intelligence Laboratory, SRI, Stanford University, University of Illinois at Urbana-Champaign, Pierre and Marie Curie University (University of Paris 6), University of Pisa, University of Tokyo Yonezawa Laboratory, Centrum Wiskunde & Informatica (CWI) and elsewhere.

## Fundamental concepts

The Actor model adopts the philosophy that *everything is an actor*. This is similar to the *everything is an object* philosophy used by some object-oriented programming languages, but differs in that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent.

An actor is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- designate the behavior to be used for the next message it receives.

There is no assumed sequence to the above actions and they could be carried out in parallel.

Decoupling the sender from communications sent was a fundamental advance of the Actor model enabling asynchronous communication and control structures as patterns of passing messages.[1]

Recipients of messages are identified by address, sometimes called "mailing address". Thus an actor can only communicate with actors whose addresses it has. It can obtain those from a message it receives, or if the address is for an actor it has itself created.

The Actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

## Formal systems

Over the years, several different formal systems have been developed which permit reasoning about systems in the Actor model. These include:

- Operational semantics
- Laws for Actor systems
- Denotational semantics
- Transition semantics

There are also formalisms that are not fully faithful to the Actor model in that they do not formalize the guaranteed delivery of messages including the following (See Attempts to relate Actor semantics to algebra and linear logic):

- Several different Actor algebras
- Linear logic

## Applications

The Actors model can be used as a framework for modelling, understanding, and reasoning about, a wide range of concurrent systems. For example:

- Electronic mail (e-mail) can be modeled as an Actor system. Accounts are modeled as Actors and email addresses as Actor addresses.
- Web Services can be modeled with SOAP endpoints modeled as Actor addresses.
- Objects with locks (*e.g.* as in Java and C#) can be modeled as a **Serializer**, provided that their implementations are such that messages can continually arrive (perhaps by being stored in an internal queue). A serializer is an important kind of Actor defined by the property that it is continually available to the arrival of new messages; every message sent to a serializer is guaranteed to arrive.
- Testing and Test Control Notation (TTCN), both TTCN-2 and TTCN-3, follows Actor model rather closely. In TTCN, Actor is a test component: either parallel test component (PTC) or main test component (MTC). Test components can send and receive messages to and from remote partners (peer test components or test system interface), the latter being identified by its address. Each test component has a behaviour tree bound to it; test

components run in parallel and can be dynamically created by parent test components. Built-in language constructs allow the definition of actions to be taken when an expected message is received from the internal message queue, like sending a message to another peer entity or creating new test components.

## Message-passing semantics

The Actor model is about the semantics of message passing.

### Unbounded nondeterminism controversy

Arguably, the first concurrent programs were interrupt handlers. During the course of its normal operation a computer needed to be able to receive information from outside (characters from a keyboard, packets from a network, *etc*). So when the information arrived the execution of the computer was "interrupted" and special code called an interrupt handler was called to *put* the information in a buffer where it could be subsequently retrieved.

In the early 1960s, interrupts began to be used to simulate the concurrent execution of several programs on a single processor. Having concurrency with shared memory gave rise to the problem of concurrency control. Originally, this problem was conceived as being one of mutual exclusion on a single computer. Edsger Dijkstra developed semaphores and later, between 1971 and 1973,[2] Tony Hoare[3] and Per Brinch Hansen[4] developed monitors to solve the mutual exclusion problem. However, neither of these solutions provided a programming-language construct that encapsulated access to shared resources. This encapsulation was later accomplished by the serializer construct ([Hewitt and Atkinson 1977, 1979] and [Atkinson 1980]).

The first models of computation (*e.g.* Turing machines, Post productions, the lambda calculus, *etc.*) were based on mathematics and made use of a global state to represent a computational *step* (later generalized in [McCarthy and Hayes 1969] and [Dijkstra 1976] see Event orderings versus global state). Each computational step was from one global state of the computation to the next global state. The global state approach was continued in automata theory for finite state machines and push down stack machines, including their nondeterministic versions. Such nondeterministic automata have the property of bounded nondeterminism; that is, if a machine always halts when started in its initial state, then there is a bound on the number of states in which it halts.

Edsger Dijkstra further developed the nondeterministic global state approach. Dijkstra's model gave rise to a controversy concerning *unbounded nondeterminism.* Unbounded nondeterminism (also called *unbounded indeterminacy*), is a property of concurrency by which the amount of delay in servicing a request can become unbounded as a result of arbitration of contention for shared resources *while still guaranteeing that the request will eventually be serviced*. Hewitt argued that the Actor model should provide the guarantee of service. In Dijkstra's model, although there could be an unbounded amount of time between the execution of sequential instructions on a computer, a (parallel) program that started out in a well defined state could terminate in only a bounded number of states [Dijkstra 1976]. Consequently, his model could not provide the guarantee of service. Dijkstra argued that it was impossible to implement unbounded nondeterminism.

Hewitt argued otherwise: there is no bound that can be placed on how long it takes a computational circuit called an *arbiter* to settle (see metastability in electronics).[5] Arbiters are used in computers to deal with the circumstance that computer clocks operate asynchronously with input from outside, *e.g.* keyboard input, disk access, network input, *etc.* So it could take an unbounded time for a message sent to a computer to be received and in the meantime the computer could traverse an unbounded number of states.

The Actor Model features unbounded nondeterminism which was captured in a mathematical model by Will Clinger using domain theory. There is no global state in the Actor model.Wikipedia:Disputed statement

### Direct communication and asynchrony

Messages in the Actor model are not necessarily buffered. This was a sharp break with previous approaches to models of concurrent computation. The lack of buffering caused a great deal of misunderstanding at the time of the development of the Actor model and is still a controversial issue. Some researchers argued that the messages are buffered in the "ether" or the "environment". Also, messages in the Actor model are simply sent (like packets in IP); there is no requirement for a synchronous handshake with the recipient.

### Actor creation plus addresses in messages means variable topology

A natural development of the Actor model was to allow addresses in messages. Influenced by packet switched networks [1961 and 1964], Hewitt proposed the development of a new model of concurrent computation in which communications would not have any required fields at all: they could be empty. Of course, if the sender of a communication desired a recipient to have access to addresses which the recipient did not already have, the address would have to be sent in the communication.

A computation might need to send a message to a recipient from which it would later receive a response. The way to do this is to send a communication which has the message along with the address of another actor called the *resumption* (sometimes also called continuation or stack frame) along with the message. The recipient could then cause a response message to be sent to the resumption.

Actor creation plus the inclusion of the addresses of actors in messages means that Actors have a potentially variable topology in their relationship to one another much as the objects in Simula also had a variable topology in their relationship to one another.

### Inherently concurrent

As opposed to the previous approach based on composing sequential processes, the Actor model was developed as an inherently concurrent model. In the Actor model sequentiality was a special case that derived from concurrent computation as explained in Actor model theory.

### No requirement on order of message arrival

Hewitt argued against adding the requirement that messages must arrive in the order in which they are sent to the Actor. If output message ordering is desired, then it can be modeled by a queue Actor that provides this functionality. Such a queue Actor would queue the messages that arrived so that they could be retrieved in FIFO order. So if an Actor `X` sent a message `M1` to an Actor `Y`, and later `X` sent another message `M2` to `Y`, there is no requirement that `M1` arrives at `Y` before `M2`.

In this respect the Actor model mirrors packet switching systems which do not guarantee that packets must be received in the order sent. Not providing the order of delivery guarantee allows packet switching to buffer packets, use multiple paths to send packets, resend damaged packets, and to provide other optimizations.

For example, Actors are allowed to pipeline the processing of messages. What this means is that in the course of processing a message `M1`, an Actor can designate the behavior to be used to process the next message, and then in fact begin processing another message `M2` before it has finished processing `M1`. Just because an Actor is allowed to pipeline the processing of messages does not mean that it *must* pipeline the processing. Whether a message is pipelined is an engineering tradeoff. How would an external observer know whether the processing of a message by an Actor has been pipelined? There is no ambiguity in the definition of an Actor created by the possibility of pipelining. Of course, it is possible to perform the pipeline optimization incorrectly in some implementations, in which case unexpected behavior may occur.

## Locality

Another important characteristic of the Actor model is locality.

Locality means that in processing a message an Actor can send messages only to addresses that it receives in the message, addresses that it already had before it received the message and addresses for Actors that it creates while processing the message. (But see Synthesizing Addresses of Actors.)

Also locality means that there is no simultaneous change in multiple locations. In this way it differs from some other models of concurrency, *e.g.*, the Petri net model in which tokens are simultaneously removed from multiple locations and placed in other locations.

## Composing Actor Systems

The idea of composing Actor systems into larger ones is an important aspect of modularity that was developed in Gul Agha's doctoral dissertation, developed later by Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott.

## Behaviors

A key innovation was the introduction of *behavior* specified as a mathematical function to express what an Actor does when it processes a message including specifying a new behavior to process the next message that arrives. Behaviors provided a mechanism to mathematically model the sharing in concurrency.

Behaviors also freed the Actor model from implementation details, *e.g.*, the Smalltalk-72 token stream interpreter. However, it is critical to understand that the efficient implementation of systems described by the Actor model require *extensive* optimization. See Actor model implementation for details.

## Modeling other concurrency systems

Other concurrency systems (*e.g.* process calculi) can be modeled in the Actor model using a two-phase commit protocol.[6]

## Computational Representation Theorem

There is a *Computational Representation Theorem* in the Actor model for systems which are closed in the sense that they do not receive communications from outside. The mathematical denotation denoted by a closed system $S$ is constructed from an initial behavior $\bot_S$ and a behavior-approximating function $\mathbf{progression}_S$. These obtain increasingly better approximations and construct a denotation (meaning) for $S$ as follows [Hewitt 2008; Clinger 1981]:

$$\mathbf{Denote}_S \equiv \lim_{i \to \infty} \mathbf{progression}_S^{\,i}(\bot_S)$$

In this way, $S$ can be mathematically characterized in terms of all its possible behaviors (including those involving unbounded nondeterminism). Although $\mathrm{Denote}_S$ is not an implementation of $S$, it can be used to prove a generalization of the Church-Turing-Rosser-Kleene thesis [Kleene 1943]:

A consequence of the above theorem is that a finite Actor can nondeterministically respond with an uncountable[*clarify*] number of different outputs.

## Relationship to logic programming

One of the key motivations for the development of the actor model was to understand and deal with the control structure issues that arose in development of the Planner programming language.Wikipedia:Citation needed Once the actor model was initially defined, an important challenge was to understand the power of the model relative to Robert Kowalski's thesis that "computation can be subsumed by deduction". Kowalski's thesis turned out to be false for the concurrent computation in the actor model (see Indeterminacy in concurrent computation). This result

reversed previous expectations because Kowalski's thesis is true for sequential computation and even some kinds of parallel computation, such as the lambda calculus.

Nevertheless attempts were made to extend logic programming to concurrent computation. However, Hewitt and Agha [1991] claimed that the resulting systems were not deductive in the following sense: computational steps of the concurrent logic programming systems do not follow deductively from previous steps (see Indeterminacy in concurrent computation). Recently, logic programming has been integrated into the actor model in a way that maintains logical semantics.

## Migration

Migration in the Actor model is the ability of Actors to change locations. *E.g.*, in his dissertation, Aki Yonezawa modeled a post office that customer Actors could enter, change locations within while operating, and exit. An Actor that can migrate can be modeled by having a location Actor that changes when the Actor migrates. However the faithfulness of this modeling is controversial and the subject of research.Wikipedia:Citation needed

## Security

The security of Actors can be protected in the following ways:

- hardwiring in which Actors are physically connected
- computer hardware as in Burroughs B5000, Lisp machine, *etc.*
- virtual machines as in Java virtual machine, Common Language Runtime, *etc.*
- operating systems as in capability-based systems
- signing and/or encryption of Actors and their addresses

## Synthesizing addresses of actors

A delicate point in the Actor model is the ability to synthesize the address of an Actor. In some cases security can be used to prevent the synthesis of addresses (see Security). However, if an Actor address is simply a bit string then clearly it can be synthesized although it may be difficult or even infeasible to guess the address of an Actor if the bit strings are long enough. SOAP uses a URL for the address of an endpoint where an Actor can be reached. Since a URL is a character string, it can clearly be synthesized although encryption can make it virtually impossible to guess.

Synthesizing the addresses of Actors is usually modeled using mapping. The idea is to use an Actor system to perform the mapping to the actual Actor addresses. For example, on a computer the memory structure of the computer can be modeled as an Actor system that does the mapping. In the case of SOAP addresses, it's modeling the DNS and rest of the URL mapping.

## Contrast with other models of message-passing concurrency

Robin Milner's initial published work on concurrency[7] was also notable in that it was not based on composing sequential processes. His work differed from the Actor model because it was based on a fixed number of processes of fixed topology communicating numbers and strings using synchronous communication. The original Communicating Sequential Processes model[8] published by Tony Hoare differed from the Actor model because it was based on the parallel composition of a fixed number of sequential processes connected in a fixed topology, and communicating using synchronous message-passing based on process names (see Actor model and process calculi history). Later versions of CSP abandoned communication based on process names in favor of anonymous communication via channels, an approach also used in Milner's work on the Calculus of Communicating Systems and the π-calculus.

These early models by Milner and Hoare both had the property of bounded nondeterminism. Modern, theoretical CSP ([Hoare 1985] and [Roscoe 2005]) explicitly provides unbounded nondeterminism.

Petri nets and their extensions (e.g. coloured Petri nets) are like actors in that they are based on asynchronous message passing and unbounded nondeterminism, while they are like early CSP in that they define fixed topologies of elementary processing steps (transitions) and message repositories (places).

## Influence

The Actor Model has been influential on both theory development and practical software development.

### Theory

The Actor Model has influenced the development of the Pi-calculus and subsequent Process calculi. In his Turing lecture, Robin Milner wrote:

> *Now, the pure lambda-calculus is built with just two kinds of thing: terms and variables. Can we achieve the same economy for a process calculus? Carl Hewitt, with his Actors model, responded to this challenge long ago; he declared that a value, an operator on values, and a process should all be the same kind of thing: an Actor.*
>
> *This goal impressed me, because it implies the homogeneity and completeness of expression ... But it was long before I could see how to attain the goal in terms of an algebraic calculus...*
>
> *So, in the spirit of Hewitt, our first step is to demand that all things denoted by terms or accessed by names--values, registers, operators, processes, objects--are all of the same kind of thing; they should all be processes.*

### Practice

The Actor Model has had extensive influence on commercial practice. For example Twitter has used actors for scalability. Also, Microsoft has used the Actor Model in the development of its Asynchronous Agents Library.[9] There are numerous other Actor libraries listed in the Actor Libraries and Frameworks section below.

## Current issues

According to Hewitt [2006], the Actor model faces issues in computer and communications architecture, concurrent programming languages, and Web Services including the following:

- scalability: the challenge of scaling up concurrency both locally and nonlocally.
- transparency: bridging the chasm between local and nonlocal concurrency. Transparency is currently a controversial issue. Some researchers Wikipedia:Avoid weasel words have advocated a strict separation between local concurrency using concurrent programming languages (e.g. Java and C#) from nonlocal concurrency using SOAP for Web services. Strict separation produces a lack of transparency that causes problems when it is desirable/necessary to change between local and nonlocal access to Web Services (see distributed computing).
- inconsistency: Inconsistency is the norm because all very large knowledge systems about human information system interactions are inconsistent. This inconsistency extends to the documentation and specifications of very large systems (e.g. Microsoft Windows software, etc.), which are internally inconsistent.

Many of the ideas introduced in the Actor model are now also finding application in multi-agent systems for these same reasons [Hewitt 2006b 2007b]. The key difference is that agent systems (in most definitions) impose extra constraints upon the Actors, typically requiring that they make use of commitments and goals.

The Actor model is also being applied to client cloud computing.

## Early Actor researchers

There is a growing community of researchers working on the Actor Model as it is becoming commercially more important. Early Actor researchers included:

- Important contributions to the semantics of Actors have been made by: Gul Agha, Beppe Attardi, Henry Baker, Will Clinger, Irene Greif, Carl Hewitt, Carl Manning, Ian Mason, Ugo Montanari, Maria Simi, Scott Smith, Carolyn Talcott, Prasanna Thati, and Akinori Yonezawa.
- Important contributions to the implementation of Actors have been made by: Bill Athas, Russ Atkinson, Beppe Attardi, Henry Baker, Gerry Barber, Peter Bishop, Nanette Boden, Jean-Pierre Briot, Bill Dally, Peter de Jong, Jessie Dedecker, Travis Desell, Ken Kahn, Carl Hewitt, Henry Lieberman, Carl Manning, Tom Reinhardt, Chuck Seitz, Richard Steiger, Dan Theriault, Mario Tokoro, Carlos Varela, Darrell Woelk.

## Programming with Actors

A number of different programming languages employ the Actor model or some variation of it. These languages include:

### Early Actor programming languages

- Act 1, 2 and 3
- Acttalk[10]
- Ani[11]
- Cantor[12]
- Rosette[13]

### Later Actor programming languages

- ABCL
- AmbientTalk[14]
- Axum
- CAL Actor Language
- D
- E
- Elixir
- Erlang
- Fantom
- Humus
- Io
- Ptolemy Project
- Rebeca Modeling Language
- Reia
- Rust
- SALSA
- Scala
- Scratch

### Actor libraries and frameworks

Actor libraries or frameworks have also been implemented to permit actor-style programming in languages that don't have actors built-in. Among these frameworks are:

| Name | Status | Latest release | License | Languages |
|---|---|---|---|---|
| Actor [15] | Active | 2013-05-30 | MIT | Java |
| Actor Framework [16] | Active | 2013-11-13 | Apache 2.0 | .NET |
| Akka (toolkit) [17] | Active | 2014-03-26 | Apache 2.0 | Java and Scala |
| Akka.NET [18] | Active | 2014-02-01 | Apache 2.0 | .NET |
| Remact.Net [19] | Active | 2014-03-23 | MIT | .NET |
| Ateji PX [20] | Active | ? | ? | Java |
| F# MailboxProcessor [21] | Active | same as F# (built-in core library) | Apache License | F# |
| Korus [22] | Active | 2010-02-01 | GPL 3 | Java |
| Kilim [23] | Active | 2011-10-13 | MIT | Java |
| ActorFoundry (based on Kilim) | Active? | 2008-12-28 | ? | Java |
| ActorKit [24] | Active | 2011-09-14 | BSD | Objective-C |
| Cloud Haskell [25] | Active | 2012-11-07 | BSD | Haskell |
| CloudI [26] | Active | 2014-02-02 | BSD | Erlang, C/C++, Java, Python, Ruby |
| NAct [27] | Active | 2012-02-28 | LGPL 3.0 | .NET |
| Retlang [28] | Active | 2011-05-18 | New BSD | .NET |
| JActor [29] | Active | 2013-01-22 | LGPL | Java |
| Jetlang [30] | Active | 2012-02-14 | New BSD | Java |
| Haskell-Actor [31] | Active? | 2008 | New BSD | Haskell |
| GPars [32] | Active | 2012-12-19 | Apache 2.0 | Groovy |
| PARLEY [33] | Active? | 2007-22-07 | GPL 2.1 | Python |
| Pulsar [34] | Active | 2013-06-30 | New BSD | Python |
| Pulsar [35] | Active | 2013-07-19 | LGPL/Eclipse | Clojure |
| Pykka [36] | Active | 2013-07-15 | Apache 2.0 | Python |
| Termite Scheme [37] | Active? | 2009 | LGPL | Scheme (Gambit implementation) |
| Theron [38] | Active | 2012-08-20 | MIT | C++ |
| Quasar [39] | Active | 2013-07-19 | LGPL/Eclipse | Java |
| Libactor [40] | Active? | 2009 | GPL 2.0 | C |
| Actor-CPP [41] | Active | 2012-03-10 | GPL 2.0 | C++ |
| S4 [42] | Active | 2011-11-28 | Apache 2.0 | Java |
| libcppa [43] | Active | 2014-04-21 | LGPL 3.0 | C++11 |

| | | | | |
|---|---|---|---|---|
| Celluloid [44] | Active | 2012-07-17 | MIT | Ruby |
| LabVIEW Actor Framework [45] | Active | 2012-03-01 | ? | LabVIEW |
| QP frameworks for real-time embedded systems | Active | 2012-09-07 | GPL 2.0 and commercial (dual licensing) | C and C++ |
| libprocess [46] | Active | 2012-07-13 | Apache 2.0 | C++ |
| SObjectizer [47] | Active | 2014-01-04 | New BSD | C++ |

Please note that not all frameworks and libraries are listed here.

# References


[1] Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages* Journal of Artificial Intelligence. June 1977.

[2] Per Brinch Hansen, *Monitors and Concurrent Pascal: A Personal History*, Comm. ACM 1996, pp 121-172

[3] C.A.R. Hoare, *Monitors: An Operating System Structuring Concept*, Comm. ACM Vol. 17, No. 10. October 1974, pp. 549-557

[4] Brinch Hansen, P., *Operating System Principles*, Prentice-Hall, July 1973.

[5] Carl Hewitt, "What is computation? Actor Model versus Turing's Model", *A Computable Universe: Understanding Computation & Exploring Nature as Computation*. Dedicated to the memory of Alan M. Turing on the 100th anniversary of his birth. Edited by Hector Zenil. World Scientific Publishing Company. 2012

**[6]** Frederick Knabe. A Distributed Protocol for Channel-Based Communication with Choice PARLE 1992.

**[7]** Robin Milner. Processes: A Mathematical Model of Computing Agents in Logic Colloquium 1973.

[8] C.A.R. Hoare. Communicating sequential processes (http://portal.acm.org/citation.cfm?id=359585&dl=GUIDE&coll=GUIDE&CFID=19884966&CFTOKEN=55490895) CACM. August 1978.

**[9]** "Actor-Based Programming with the Asynchronous Agents Library" MSDN September 2010.

**[10]** Jean-Pierre Briot. Acttalk: A framework for object-oriented concurrent programming-design and experience 2nd France-Japan workshop. 1999.

**[11]** Ken Kahn. A Computational Theory of Animation MIT EECS Doctoral Dissertation. August 1979.

**[12]** William Athas and Nanette Boden Cantor: An Actor Programming System for Scientific Computing in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.

**[13]** Darrell Woelk. Developing InfoSleuth Agents Using Rosette: An Actor Based Language Proceedings of the CIKM '95 Workshop on Intelligent Information Agents. 1995.

[14] Dedecker J., Van Cutsem T., Mostinckx S., D'Hondt T., De Meuter W. Ambient-oriented Programming in AmbientTalk. In “Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP), Dave Thomas (Ed.), Lecture Notes in Computer Science Vol. 4067, pp. 230-254, Springer-Verlag.”, 2006

[15] https://github.com/edescourtis/actor

[16] http://actorfx.codeplex.com

[17] http://akka.io

[18] https://github.com/rogeralsing/Pigeon

[19] https://github.com/steforster/Remact.Net

[20] http://www.ateji.com/px

[21] http://en.wikibooks.org/wiki/F_Sharp_Programming/MailboxProcessor

[22] http://code.google.com/p/korus/

[23] http://kilim.malhar.net/

[24] https://github.com/stevedekorte/ActorKit

[25] http://haskell-distributed.github.com/wiki.html

[26] http://cloudi.org

[27] http://code.google.com/p/n-act/

[28] http://code.google.com/p/retlang/

[29] http://jactorconsulting.com/product/jactor/

[30] http://code.google.com/p/jetlang/

[31] http://code.google.com/p/haskellactor/

[32] http://gpars.codehaus.org/

[33] http://osl.cs.uiuc.edu/parley/

[34] http://quantmind.github.io/pulsar/

[35] https://github.com/puniverse/pulsar

[36] http://pykka.readthedocs.org/en/latest/index.html
[37] http://code.google.com/p/termite/
[38] http://www.theron-library.com/
[39] https://github.com/puniverse/quasar
[40] https://code.google.com/p/libactor/
[41] http://code.google.com/p/actor-cpp/
[42] http://incubator.apache.org/s4/
[43] http://libcppa.blogspot.com/
[44] http://github.com/celluloid/celluloid/
[45] http://ni.com/actorframework
[46] https://github.com/libprocess/libprocess
[47] http://sourceforge.net/projects/sobjectizer/

## Further reading


- Gul Agha. **Actors: A Model of Concurrent Computation in Distributed Systems**. MIT Press 1985.
- Paul Baran. **On Distributed Communications Networks** IEEE Transactions on Communications Systems. March 1964.
- William A. Woods. **Transition network grammars for natural language analysis** CACM. 1970.
- Carl Hewitt. **Procedural Embedding of Knowledge In Planner** IJCAI 1971.
- G.M. Birtwistle, Ole-Johan Dahl, B. Myhrhaug and Kristen Nygaard. **SIMULA Begin** Auerbach Publishers Inc, 1973.
- Carl Hewitt, *et al.* **Actor Induction and Meta-evaluation** Conference Record of ACM Symposium on Principles of Programming Languages, January 1974.
- Carl Hewitt, *et al.* **Behavioral Semantics of Nonrecursive Control Structure** Proceedings of Colloque sur la Programmation, April 1974.
- Irene Greif and Carl Hewitt. **Actor Semantics of PLANNER-73** Conference Record of ACM Symposium on Principles of Programming Languages. January 1975.
- Carl Hewitt. **How to Use What You Know** IJCAI. September, 1975.
- Alan Kay and Adele Goldberg. **Smalltalk-72 Instruction Manual** (http://www.bitsavers.org.nyud.net/pdf/xerox/parc/techReports/Smalltalk-72_Instruction_Manual_Mar76.pdf) Xerox PARC Memo SSL-76-6. May 1976.
- Edsger Dijkstra. **A discipline of programming** Prentice Hall. 1976.
- Carl Hewitt and Henry Baker **Actors and Continuous Functionals** (http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-194.pdf) Proceeding of IFIP Working Conference on Formal Description of Programming Concepts. August 1–5, 1977.
- Carl Hewitt and Russ Atkinson. **Synchronization in Actor Systems** (http://portal.acm.org/citation.cfm?id=512975&coll=portal&dl=ACM) Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977
- Carl Hewitt and Russ Atkinson. **Specification and Proof Techniques for Serializers** IEEE Journal on Software Engineering. January 1979.
- Ken Kahn. **A Computational Theory of Animation** MIT EECS Doctoral Dissertation. August 1979.
- Carl Hewitt, Beppe Attardi, and Henry Lieberman. **Delegation in Message Passing** Proceedings of First International Conference on Distributed Systems Huntsville, AL. October 1979.
- Nissim Francez, C.A.R. Hoare, Daniel Lehmann, and Willem-Paul de Roever. **Semantics of nondetermiism, concurrency, and communication** Journal of Computer and System Sciences. December 1979.
- George Milne and Robin Milner. **Concurrent processes and their syntax** JACM. April 1979.
- Daniel Theriault. **A Primer for the Act-1 Language** MIT AI memo 672. April 1982.
- Daniel Theriault. **Issues in the Design and Implementation of Act 2** MIT AI technical report 728. June 1983.

- Henry Lieberman. **An Object-Oriented Simulator for the Apiary** Conference of the American Association for Artificial Intelligence, Washington, D. C., August 1983
- Carl Hewitt and Peter de Jong. **Analyzing the Roles of Descriptions and Actions in Open Systems** Proceedings of the National Conference on Artificial Intelligence. August 1983.
- Carl Hewitt and Henry Lieberman. **Design Issues in Parallel Architecture for Artificial Intelligence** MIT AI memo 750. Nov. 1983.
- C.A.R. Hoare. **Communicating Sequential Processes** (http://www.usingcsp.com/) Prentice Hall. 1985.
- Carl Hewitt. **The Challenge of Open Systems** Byte Magazine. April 1985. Reprinted in *The foundation of artificial intelligence---a sourcebook* Cambridge University Press. 1990.
- Carl Manning. **Traveler: the actor observatory** ECOOP 1987. Also appears in Lecture Notes in Computer Science, vol. 276.
- William Athas and Charles Seitz **Multicomputers: message-passing concurrent computers** IEEE Computer August 1988.
- William Athas and Nanette Boden **Cantor: An Actor Programming System for Scientific Computing** in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.
- Jean-Pierre Briot. **From objects to actors: Study of a limited symbiosis in Smalltalk-80** Rapport de Recherche 88-58, RXF-LITP, Paris, France, September 1988
- William Dally and Wills, D. **Universal mechanisms for concurrency** PARLE 1989.
- W. Horwat, A. Chien, and W. Dally. **Experience with CST: Programming and Implementation** PLDI. 1989.
- Carl Hewitt. **Towards Open Information Systems Semantics** Proceedings of 10th International Workshop on Distributed Artificial Intelligence. October 23–27, 1990. Bandera, Texas.
- Akinori Yonezawa, Ed. **ABCL: An Object-Oriented Concurrent System** MIT Press. 1990.
- K. Kahn and Vijay A. Saraswat, " Actors as a special case of concurrent constraint (logic) programming (http://doi.acm.org/10.1145/97946.97955)", in SIGPLAN *Notices*, October 1990. Describes Janus.
- Carl Hewitt. **Open Information Systems Semantics** Journal of Artificial Intelligence. January 1991.
- Carl Hewitt and Jeff Inman. **DAI Betwixt and Between: From "Intelligent Agents" to Open Systems Science** IEEE Transactions on Systems, Man, and Cybernetics. Nov./Dec. 1991.
- Carl Hewitt and Gul Agha. **Guarded Horn clause languages: are they deductive and Logical?** International Conference on Fifth Generation Computer Systems, Ohmsha 1988. Tokyo. Also in *Artificial Intelligence at MIT*, Vol. 2. MIT Press 1991.
- William Dally, *et al.* **The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms** IEEE Micro. April 1992.
- S. Miriyala, G. Agha, and Y.Sami. **Visulatizing actor programs using predicate transition nets** Journal of Visual Programming. 1992.
- Carl Hewitt and Carl Manning. **Negotiation Architecture for Large-Scale Crisis Management** AAAI-94 Workshop on Models of Conflict Management in Cooperative Problem Solving. Seattle, WA. Aug. 4, 1994.
- Carl Hewitt and Carl Manning. **Synthetic Infrastructures for Multi-Agency Systems** Proceedings of ICMAS '96. Kyoto, Japan. December 8–13, 1996.
- S. Frolund. **Coordinating Distributed Objects: An Actor-Based Approach for Synchronization** MIT Press. November 1996.
- W. Kim. **ThAL: An Actor System for Efficient and Scalable Concurrent Computing** PhD thesis. University of Illinois at Urbana Champaign. 1997.
- Jean-Pierre Briot. **Acttalk: A framework for object-oriented concurrent programming-design and experience** (http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/ec89/ec890109.pdf) 2nd France-Japan workshop. 1999.

- N. Jamali, P. Thati, and G. Agha. **An actor based architecture for customizing and controlling agent ensembles** IEEE Intelligent Systems. 14(2). 1999.
- Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, Dave Winer. **Simple Object Access Protocol (SOAP) 1.1** W3C Note. May 2000.
- M. Astley, D. Sturman, and G. Agha. **Customizable middleware for modular distributed software** CACM. 44(5) 2001.
- Edward Lee, S. Neuendorffer, and M. Wirthlin. **Actor-oriented design of embedded hardware and software systems** (http://ptolemy.eecs.berkeley.edu/papers/02/actorOrientedDesign/newFinal.pdf) *Journal of Circuits, Systems, and Computers*. 2002.
- P. Thati, R. Ziaei, and G. Agha. **A Theory of May Testing for Actors** Formal Methods for Open Object-based Distributed Systems. March 2002.
- P. Thati, R. Ziaei, and G. Agha. **A theory of may testing for asynchronous calculi with locality and no name matching** Algebraic Methodology and Software Technology. Springer Verlag. September 2002. LNCS 2422.
- Stephen Neuendorffer. **Actor-Oriented Metaprogramming** (http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/ERL-05-1.pdf) PhD Thesis. University of California, Berkeley. December, 2004
- Carl Hewitt (2006a) **The repeated demise of logic programming and why it will be reincarnated** What Went Wrong and Why: Lessons from AI Research and Applications. Technical Report SS-06-08. AAAI Press. March 2006.
- Carl Hewitt (2006b) *What is Commitment? Physical, Organizational, and Social* (http://www.pcs.usp.br/~coin-aamas06/10_commitment-43_16pages.pdf) COIN@AAMAS. April 27, 2006b.
- Carl Hewitt (2007a) **What is Commitment? Physical, Organizational, and Social (Revised)** Pablo Noriega .et al. editors. LNAI 4386. Springer-Verlag. 2007.
- Carl Hewitt (2007b) **Large-scale Organizational Computing requires Unstratified Paraconsistency and Reflection** COIN@AAMAS'07.
- D. Charousset, T. C. Schmidt, R. Hiesgen and M. Wählisch. *Native actors: a scalable software platform for distributed, heterogeneous environments* (http://dx.doi.org/10.1145/2541329.2541336) in AGERE! '13 Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control.


# External links

## Videos

- Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know, but were afraid to ask) (http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask) Microsoft Channel 9. April 9, 2012.

## Articles

- Actors on the JVM (http://drdobbs.com//229402193) Dr. Dobb's, April 2011
- A now dated set of speculations by Paul Mackay can be found at *Why has the actor model not succeeded?* (http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/)

### Procedural Libraries

- JavAct (http://www.irit.fr/PERSONNEL/SMAC/arcangeli/JavAct.html) - a Java library for programming concurrent, distributed, and mobile applications using the actor model (and open implementation principles).
- Functional Java (http://functionaljava.org/) - a Java library of that includes an implementation of concurrent actors with code examples in standard Java and Java 7 BGGA style.
- ActorFoundry (http://osl.cs.uiuc.edu/af) - a Java-based library for Actor programming. The familiar Java syntax, an ant build file and a bunch of example make the entry barrier very low.
- ActiveJava (http://tristan.aubrey-jones.com/code/?project=third_year_project&dir=/) - a prototype Java language extension for Actor programming.
- Akka (http://akka.io) - Actor based library in Scala and Java, from Typesafe Inc..
- GPars (http://gpars.codehaus.org) - a concurrency library for Groovy and Java
- Asynchronous Agents Library (http://msdn.microsoft.com/en-us/library/dd492627.aspx) - Microsoft actor library for Visual C++. "The Agents Library is a C++ template library that promotes an actor-based programming model and in-process message passing for coarse-grained dataflow and pipelining tasks. "

# Linda (coordination language)

In computer science, **Linda** is a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory. Linda was developed by David Gelernter and Nicholas Carriero at Yale University.

## Model

This model is implemented as a "coordination language" in which several primitives operating on ordered sequence of typed data objects, "tuples," are added to a sequential language, such as C, and a logically global associative memory, called a tuplespace, in which processes store and retrieve tuples.

The original Linda model requires four operations that individual workers perform on the tuples and the tuplespace:

- **in** atomically reads and removes—consumes—a tuple from tuplespace
- **rd** non-destructively reads a tuplespace
- **out** produces a tuple, writing it into tuplespace
- **eval** creates new processes to evaluate tuples, writing the result into tuplespace

## Evaluation

Compared to other parallel-processing models, Linda is more orthogonal in treating process coordination as a separate activity from computation, and it is more general in being able to subsume various levels of concurrency—uniprocessor, multi-threaded multiprocessor, or networked—under a single model. Its orthogonality allows processes computing in different languages and platforms to interoperate using the same primitives. Its generality allows a multi-threaded Linda system to be distributed across multiple computers without change.

Whereas message-passing models require tightly-coupled processes sending messages to each other in some sequence or protocol, Linda processes are decoupled from other processes, communicating only through the tuplespace; a process need have no notion of other processes except for the kinds of tuples consumed or produced (data coupling).

Researchers have proposed more primitives to support different types of communication and co-ordination between (open distributed) computer systems, and to solve particular problems arising from various uses of the model. Researchers have also experimented with various means of implementing the virtual shared memory for this model.

Many of these researchers proposed larger modifications to the original Linda model, developing a family of systems known as Linda-like systems and implemented as orthogonal technology (unlike original version). An example of this is the language Ease designed by Steven Ericsson-Zenith.

## Implementations

Linda was originally implemented in C and Fortran, but has since been implemented in many programming languages, including:

- C: C-Linda, TCP-Linda [1], LinuxTuples [2]
- C++: CppLinda [3]
- Erlang: Erlinda [4]
- Java: JavaSpaces, TSpaces [5], LightTS [6], LIME [7]
- Lisp
- Lua: LuaTS [8] Lua Lanes [9]
- Prolog: SICStus Prolog Linda [10]
- Python: PyLinda
- Ruby: Rinda

Some of the more notable Linda implementations include:

- C-Linda or TCP-Linda - the earliest commercial and a widespread implementation of virtual shared memory for supercomputers and clustered systems from Scientific Computing Associates, founded by Martin Schultz.
- JavaSpaces - a Java-based tuplespace implementation that helped popularize distributed computing.
- TSpaces [5] - a Java-based tuplespace platform from IBM.Wikipedia:Undue weight

## Criticisms

Criticisms of Linda from the multiprocessing community tend to focus on the decreased speed of operations in Linda systems as compared to MPI systems.Wikipedia:Citation needed While not without justification, these claims were largely refuted for an important class of problems. Detailed criticisms of the Linda model can also be found in Steven Ericsson-Zenith's book *Process Interaction Models*.

## Name

Linda is named after Linda Lovelace, an actress in the porn movie Deep Throat, a pun on Ada's tribute to Ada Lovelace.

## Publications


- Gelernter, David; Carriero, Nicholas (1992). "Coordination Languages and their Significance". Communications of the ACM. doi:10.1145/129630.129635 [11].
- Carriero, Nicholas; Gelernter, David; Mattson, Timothy; Sherman, Andrew (1994). "The Linda Alternative to Message-Passing systems". Parallel Computing. doi:10.1016/0167-8191(94)90032-9 [12].
- Wells, George. "Coordination Languages: Back to the Future with Linda" [13] (PDF). Rhodes University.
- Sluga, Thomas Arkadius. "Modern C++ Implementation of the LINDA coordination language". University of Hannover.

## References

[1] http://lindaspaces.com/products/linda.html
[2] http://linuxtuples.sourceforge.net/
[3] http://sourceforge.net/projects/cpplinda/
[4] http://code.google.com/p/erlinda/
[5] http://www.almaden.ibm.com/cs/TSpaces/
[6] http://lights.sourceforge.net/
[7] http://lime.sourceforge.net/index.html
[8] http://www.jucs.org/jucs_9_8/luats_a_reactive_event/Leal_M_A.pdf
[9] http://kotisivu.dnainternet.net/askok/bin/lanes/
[10] http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_29.html
[11] http://dx.doi.org/10.1145%2F129630.129635
[12] http://dx.doi.org/10.1016%2F0167-8191%2894%2990032-9
[13] http://wcat05.unex.es/Documents/Wells.pdf

## External links

- Coordination Language (http://www.jpaulmorrison.com/cgi-bin/wiki.pl?CoordinationLanguage) - A small discussion about the differences between the approach of Linda and that of Flow-based programming
- Linda for C++ (https://sourceforge.net/projects/cpplinda/)
- Linda for C (http://www.comp.nus.edu.sg/~wongwf/linda.html)
- Erlinda (for Erlang) (http://code.google.com/p/erlinda/)
- Linda for Java (http://www.cs.bris.ac.uk/Publications/Papers/2000380.pdf)
- Linda for Prolog (http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_29.html)
- PyLinda (for Python) (http://code.google.com/p/pylinda/)
- Rinda (for Ruby) (http://segment7.net/projects/ruby/drb/rinda/)
- Linda in a Mobile Environment (LIME) (http://lime.sourceforge.net/) (for nesC)

# Scala (programming language)

**Scala**

| | |
|---|---|
| **Paradigm(s)** | Multi-paradigm: functional, object-oriented, imperative, concurrent |
| **Designed by** | Martin Odersky |
| **Developer** | Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne |
| **Appeared in** | 2003 |
| **Stable release** | 2.11.1 / May 21, 2014 |
| **Preview release** | 2.11.0-RC4 / April 8, 2014 |
| **Typing discipline** | static, strong, inferred, structural |
| **Influenced by** | Eiffel, Erlang, Haskell, Java, Lisp, Pizza,[1] Standard ML, OCaml, Scheme, Smalltalk, Oz |
| **Influenced** | Fantom, Ceylon, Lasso, Kotlin |
| **Implementation language** | Scala |
| **Platform** | JVM, LLVM |
| **License** | Scala License (similar to BSD license) |
| **Filename extension(s)** | .scala |
| **Website** | www.scala-lang.org [2] |
| • Scala at Wikibooks | |

**Scala** (/ˈskɑːlə/ ***SKAH**-lə*) is an object-functional programming and scripting language for general software applications. Scala has full support for functional programming (including currying, pattern matching, algebraic data types, lazy evaluation, tail recursion, immutability, etc.) and a very strong static type system. This allows programs written in Scala to be very concise and thus smaller in size than most general purpose programming languages. Many of Scala's design decisions were inspired by criticism over the shortcomings of Java.

Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Java libraries may be used directly in Scala code, and vice versa. Like Java, Scala is statically typed and object-oriented, and uses a curly-brace syntax reminiscent of the C programming language. Unlike Java, Scala has many features of functional programming languages like Scheme, Standard ML and Haskell, including anonymous functions, type inference, list comprehensions, and lazy initialization. Scala also has extensive language and library support for avoiding side-effects, pattern matching, delimited continuations, higher-order types, and covariance and contravariance. Scala has a "unified type system", meaning that all types (including primitive types like integer and boolean) are subclasses of the type `Any`. This is similar to C# but unlike Java. Scala likewise has other features present in C# but not Java, including anonymous types, operator overloading, optional parameters, named parameters, raw strings, and no checked exceptions.

The name Scala is a portmanteau of "scalable" and "language", signifying that it is designed to grow with the demands of its users.

## History

The design of Scala started in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) by Martin Odersky, following on from work on Funnel, a programming language combining ideas from functional programming and Petri nets.[3] Odersky had previously worked on Generic Java and javac, Sun's Java compiler.

Scala was released internally in late 2003, before being released publicly in early 2004 on the Java platform, and on the .NET platform in June 2004.[4] A second version of the language, v2.0, was released in March 2006. The .NET support was officially dropped in 2012.[5]

On 17 January 2011 the Scala team won a five-year research grant of over €2.3 million from the European Research Council.[6] On 12 May 2011, Odersky and collaborators launched Typesafe Inc., a company to provide commercial support, training, and services for Scala. Typesafe received a $3 million investment from Greylock Partners.

## Platforms and license

Scala runs on the Java platform (Java Virtual Machine) and is compatible with existing Java programs. It also runs on Android smartphones.[7] There was an alternative implementation for the .NET platform, but it was dropped.

The Scala software distribution, including compiler and libraries, is released under a BSD license.

## Examples

### "Hello World" example

The Hello World program written in Scala has this form:

```
object HelloWorld extends App {
  println("Hello, World!")
}
```

Unlike the stand-alone Hello World application for Java, there is no class declaration and nothing is declared to be static; a singleton object created with the **object** keyword is used instead.

With the program saved in a file named `HelloWorld.scala`, it can be compiled from the command line:

```
$ scalac HelloWorld.scala
```

To run it:

`$ scala HelloWorld` (You may need to use the "-cp" key to set the classpath like in Java).

This is analogous to the process for compiling and running Java code. Indeed, Scala's compilation and execution model is identical to that of Java, making it compatible with Java build tools such as Ant.

A shorter version of the "Hello World" Scala program is:

```
println("Hello, World!")
```

Saved in a file named `HelloWorld2.scala`, this can be run as a script without prior compilation using:

```
$ scala HelloWorld2.scala
```

Commands can also be entered directly into the Scala interpreter, using the option **-e**:

```
$ scala -e 'println("Hello, World!")'
```

### A basic example

The following example shows the differences between Java and Scala syntax:

```
font size="7.63">
/ Java:
t mathFunction(int num) {
  int numSquare = num*num;
  return (int) (Math.cbrt(numSquare) +
    Math.log(numSquare));

font>
```

```
font size="7.63">
/ Scala: Direct conversion from Java

/ no import needed; scala.math
/ already imported as `math`
ef mathFunction(num: Int): Int = {
 var numSquare: Int = num*num
 return (math.cbrt(numSquare) + math.log(numSquare)).
   asInstanceOf[Int]

font>
```

```
<font size="7.63">
// Scala: More idiomatic
// Uses type inference, omits `return` statemen
// uses `toInt` method

import math._
def intRoot23(num: Int) = {
  val numSquare = num*num
  (cbrt(numSquare) + log(numSquare)).toInt
}
</font>
```

Some syntactic differences in this code are:

- Scala does not require semicolons to end statements.
- Value types are capitalized: `Int, Double, Boolean` instead of `int, double, boolean`.
- Parameter and return types follow, as in Pascal, rather than precede as in C.
- Functions must be preceded by `def`.
- Local or class variables must be preceded by `val` (indicates an immutable variable) or `var` (indicates a mutable variable).
- The `return` operator is unnecessary in a function (although allowed); the value of the last executed statement or expression is normally the function's value.
- Instead of the Java cast operator `(Type) foo`, Scala uses `foo.asInstanceOf[Type]`, or a specialized function such as `toDouble` or `toInt`.
- Instead of Java's `import foo.*;`, Scala uses `import foo._`.
- Function or method `foo()` can also be called as just `foo`; method `thread.send(signo)` can also be called as just `thread send signo`; and method `foo.toString()` can also be called as just `foo toString`.

These syntactic relaxations are designed to allow support for domain-specific languages.

Some other basic syntactic differences:

- Array references are written like function calls, e.g. `array(i)` rather than `array[i]`. (Internally in Scala, both arrays and functions are conceptualized as kinds of mathematical mappings from one object to another.)
- Generic types are written as e.g. `List[String]` rather than Java's `List<String>`.
- Instead of the pseudo-type `void`, Scala has the actual singleton class `Unit` (see below).

### An example with classes

The following example contrasts the definition of classes in Java and Scala.

```
<font size="7.94">
// Java:
public class Point {
  private final double x, y;

  public Point(final double X, final double Y) {
    x = X;
    y = Y;
  }

  public double getX() {
    return x;
  }

  public double getY() {
    return y;
  }

  public Point(
    final double X, final double Y,
    final boolean ADD2GRID
  ) {
    this(X, Y);

    if (ADD2GRID)
      grid.add(this);
  }

  public Point() {
    this(0.0, 0.0);
  }

  double distanceToPoint(final Point OTHER) {
    return distanceBetweenPoints(x, y,
      OTHER.x, OTHER.y);
  }

  private static Grid grid = new Grid();

  static double distanceBetweenPoints(
      final double X1, final double Y1,
      final double X2, final double Y2
  ) {
    return Math.hypot(X1 - X2, Y1 - Y2);
  }
}
</font>
```

```
<font size="7.94">
// Scala
class Point(
    val x: Double, val y: Double,
    addToGrid: Boolean = false
) {
  import Point._

  if (addToGrid)
    grid.add(this)

  def this() {
    this(0.0, 0.0)
  }

  def distanceToPoint(other: Point) =
    distanceBetweenPoints(x, y, other.x, other.y)
}

object Point {
  private val grid = new Grid()

  def distanceBetweenPoints(x1: Double, y1: Double,
      x2: Double, y2: Double) = {
    math.hypot(x1 - x2, y1 - y2)
  }
}
</font>
```

The above code shows some of the conceptual differences between Java and Scala's handling of classes:

- Scala has no static variables or methods. Instead, it has *singleton objects*, which are essentially classes with only one object in the class. Singleton objects are declared using `object` instead of `class`. It is common to place static variables and methods in a singleton object with the same name as the class name, which is then known as a *companion object*. (The underlying class for the singleton object has a `$` appended. Hence, for `class Foo` with companion object `object Foo`, under the hood there's a class `Foo$` containing the companion object's

code, and a single object of this class is created, using the singleton pattern.)

- In place of constructor parameters, Scala has *class parameters*, which are placed on the class itself, similar to parameters to a function. When declared with a `val` or `var` modifier, fields are also defined with the same name, and automatically initialized from the class parameters. (Under the hood, external access to public fields always goes through accessor (getter) and mutator (setter) methods, which are automatically created. The accessor function has the same name as the field, which is why it's unnecessary in the above example to explicitly declare accessor methods.) Note that alternative constructors can also be declared, as in Java. Code that would go into the default constructor (other than initializing the member variables) goes directly at class level.
- Default visibility in Scala is `public`.

## Features (with reference to Java)

Scala has the same compilation model as Java and C#, namely separate compilation and dynamic class loading, so that Scala code can call Java libraries, or .NET libraries in the .NET implementation.

Scala's operational characteristics are the same as Java's. The Scala compiler generates byte code that is nearly identical to that generated by the Java compiler. In fact, Scala code can be decompiled to readable Java code, with the exception of certain constructor operations. To the JVM, Scala code and Java code are indistinguishable. The only difference is a single extra runtime library, `scala-library.jar`.

Scala adds a large number of features compared with Java, and has some fundamental differences in its underlying model of expressions and types, which make the language theoretically cleaner and eliminate a number of "corner cases" in Java. From the Scala perspective, this is practically important because a number of additional features in Scala are also available in C#. Examples include:

### Syntactic flexibility

As mentioned above, Scala has a good deal of syntactic flexibility, compared with Java. The following are some examples:

- Semicolons are unnecessary; lines are automatically joined if they begin or end with a token that cannot normally come in this position, or if there are unclosed parentheses or brackets.
- Any method can be used as an infix operator, e.g. `"%d apples".format(num)` and `"%d apples" format num` are equivalent. In fact, arithmetic operators like `+` and `<<` are treated just like any other methods, since function names are allowed to consist of sequences of arbitrary symbols (with a few exceptions made for things like parens, brackets and braces that must be handled specially); the only special treatment that such symbol-named methods undergo concerns the handling of precedence.
- Methods `apply` and `update` have syntactic short forms. `foo()`—where `foo` is a value (singleton object or class instance)—is short for `foo.apply()`, and `foo() = 42` is short for `foo.update(42)`. Similarly, `foo(42)` is short for `foo.apply(42)`, and `foo(4) = 2` is short for `foo.update(4, 2)`. This is used for collection classes and extends to many other cases, such as STM cells.
- Scala distinguishes between no-parens (`def foo = 42`) and empty-parens (`def foo() = 42`) methods. When calling an empty-parens method, the parentheses may be omitted, which is useful when calling into Java libraries which do not know this distinction, e.g., using `foo.toString` instead of `foo.toString()`. By convention a method should be defined with empty-parens when it performs side effects.
- Method names ending in colon (`:`) expect the argument on the left-hand-side and the receiver on the right-hand-side. For example, the `4 :: 2 :: Nil` is the same as `Nil.::(2).::(4)`, the first form corresponding visually to the result (a list with first element 4 and second element 2).
- Class body variables can be transparently implemented as separate getter and setter methods. For `trait FooLike { var bar: Int }`, an implementation may be `object Foo extends FooLike { private var x = 0; def bar = x; def bar_=(value: Int) { x = value }}`. The call

site will still be able to use a concise `foo.bar = 42`.

- The use of curly braces instead of parentheses is allowed in method calls. This allows pure library implementations of new control structures.[8] For example, `breakable { ... if (...) break() ... }` looks as if `breakable` was a language defined keyword, but really is just a method taking a thunk argument. Methods that take thunks or functions often place these in a second parameter list, allowing to mix parentheses and curly braces syntax: `Vector.fill(4) { math.random }` is the same as `Vector.fill(4)(math.random)`. The curly braces variant allows the expression to span multiple lines.
- For-expressions (explained further down) can accommodate any type that defines monadic methods such as `map`, `flatMap` and `filter`.

By themselves, these may seem like questionable choices, but collectively they serve the purpose of allowing domain-specific languages to be defined in Scala without needing to extend the compiler. For example, Erlang's special syntax for sending a message to an actor, i.e. `actor ! message` can be (and is) implemented in a Scala library without needing language extensions.

### Unified type system

Java makes a sharp distinction between primitive types (e.g. `int` and `boolean`) and reference types (any class). Only reference types are part of the inheritance scheme, deriving from `java.lang.Object`. In Scala, however, all types inherit from a top-level class `Any`, whose immediate children are `AnyVal` (value types, such as `Int` and `Boolean`) and `AnyRef` (reference types, as in Java). This means that the Java distinction between primitive types and boxed types (e.g. `int` vs. `Integer`) is not present in Scala; boxing and unboxing is completely transparent to the user. Scala 2.10 allows for new value types to be defined by the user.

### For-expressions

Instead of the Java "foreach" loops for looping through an iterator, Scala has a much more powerful concept of `for`-expressions. These are similar to list comprehensions in a languages such as Haskell, or a combination of list comprehensions and generator expressions in Python. For-expressions using the `yield` keyword allow a new collection to be generated by iterating over an existing one, returning a new collection of the same type. They are translated by the compiler into a series of `map`, `flatMap` and `filter` calls. Where `yield` is not used, the code approximates to an imperative-style loop, by translating to `foreach`.

A simple example is:

```
val s = for (x <- 1 to 25 if x*x > 50) yield 2*x
```

The result of running it is the following vector:

```
Vector(16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50)
```

(Note that the expression `1 to 25` is not special syntax. The method `to` is rather defined in the standard Scala library as an extension method on integers, using a technique known as implicit conversions that allows new methods to be added to existing types.)

A more complex example of iterating over a map is:

```
// Given a map specifying Twitter users mentioned in a set of tweets,
// and number of times each user was mentioned, look up the users
// in a map of known politicians, and return a new map giving only the
// Democratic politicians (as objects, rather than strings).
val dem_mentions = for {
    (mention, times) <- mentions
```

```
    account          <- accounts.get(mention)
    if account.party == "Democratic"
  } yield (account, times)
```

Expression `(mention, times) <- mentions` is an example of pattern matching (see below). Iterating over a map returns a set of key-value tuples, and pattern-matching allows the tuples to easily be destructured into separate variables for the key and value. Similarly, the result of the comprehension also returns key-value tuples, which are automatically built back up into a map because the source object (from the variable `mentions`) is a map. Note that if `mentions` instead held a list, set, array or other collection of tuples, exactly the same code above would yield a new collection of the same type.

## Functional tendencies

While supporting all of the object-oriented features available in Java (and in fact, augmenting them in various ways), Scala also provides a large number of capabilities that are normally found only in functional programming languages. Together, these features allow Scala programs to be written in an almost completely functional style, and also allow functional and object-oriented styles to be mixed.

Examples are:

- No distinction between statements and expressions
- Type inference
- Anonymous functions with capturing semantics (i.e. closures)
- Immutable variables and objects
- Lazy evaluation
- Delimited continuations (since 2.8)
- Higher-order functions
- Nested functions
- Currying
- Pattern matching
- Algebraic data types (through "case classes")
- Tuples

### Everything is an expression

Unlike C or Java, but similar to languages such as Lisp, Scala makes no distinction between statements and expressions. All statements are in fact expressions that evaluate to some value. Functions that would be declared as returning `void` in C or Java, and statements like `while` that logically do not return a value, are in Scala considered to return the type `Unit`, which is a singleton type, with only one object of that type. Functions and operators that never return at all (e.g. the `throw` operator or a function that always exits non-locally using an exception) logically have return type `Nothing`, a special type containing no objects; that is, a bottom type, i.e. a subclass of every possible type. (This in turn makes type `Nothing` compatible with every type, allowing type inference to function correctly.)

Similarly, an `if-then-else` "statement" is actually an expression, which produces a value, i.e. the result of evaluating one of the two branches. This means that such a block of code can be inserted wherever an expression is desired, obviating the need for a ternary operator in Scala:

```
<font size="7.78">
// Java:
int hexDigit = x >= 10 ? x + 'A' - 10 : x + '0';
</font>
```

```
<font size="7.78">
// Scala:
val hexDigit = if (x >= 10) x + 'A' - 10 else x + '0
</font>
```

For similar reasons, `return` statements are unnecessary in Scala, and in fact are discouraged. As in Lisp, the last expression in a block of code is the value of that block of code, and if the block of code is the body of a function, it will be returned by the function.

To make it clear that all expressions are functions, even methods that return `Unit` are written with an equals sign

```
def printValue(x: String): Unit = {
  println("I ate a %s".format(x))
}
```

or equivalently (with type inference, and omitting the unnecessary braces):

```
def printValue(x: String) = println("I ate a %s" format x)
```

**Type inference**

Due to type inference, the type of variables, function return values, and many other expressions can typically be omitted, as the compiler can deduce it. Examples are `val x = "foo"` (for an immutable, constant variable or immutable object) or `var x = 1.5` (for a variable whose value can later be changed). Type inference in Scala is essentially local, in contrast to the more global Hindley-Milner algorithm used in Haskell, ML and other more purely functional languages. This is done to facilitate object-oriented programming. The result is that certain types still need to be declared (most notably, function parameters, and the return types of recursive functions), e.g.

```
def formatApples(x: Int) = "I ate %d apples".format(x)
```

or (with a return type declared for a recursive function)

```
def factorial(x: Int): Int =
  if (x == 0)
    1
  else
    x*factorial(x - 1)
```

**Anonymous functions**

In Scala, functions are objects, and a convenient syntax exists for specifying anonymous functions. An example is the expression `x => x < 2`, which specifies a function with a single parameter, that compares its argument to see if it is less than 2. It is equivalent to the Lisp form `(lambda (x) (< x 2))`. Note that neither the type of `x` nor the return type need be explicitly specified, and can generally be inferred by type inference; but they can be explicitly specified, e.g. as `(x: Int) => x < 2` or even `(x: Int) => (x < 2): Boolean`.

Anonymous functions behave as true closures in that they automatically capture any variables that are lexically available in the environment of the enclosing function. Those variables will be available even after the enclosing function returns, and unlike in the case of Java's "anonymous inner classes" do not need to be declared as final. (It is even possible to modify such variables if they are mutable, and the modified value will be available the next time the anonymous function is called.)

An even shorter form of anonymous function uses placeholder variables: For example, the following:

```
list map { x => sqrt(x) }
```

can be written more concisely as

```
list map { sqrt(_) }
```

### Immutability

Scala enforces a distinction between immutable (unmodifiable, read-only) variables, whose value cannot be changed once assigned, and mutable variables, which can be changed. A similar distinction is made between immutable and mutable objects. The distinction must be made when a variable is declared: Immutable variables are declared with `val` while mutable variables use `var`. Similarly, all of the collection objects (container types) in Scala, e.g. linked lists, arrays, sets and hash tables, are available in mutable and immutable variants, with the immutable variant considered the more basic and default implementation. The immutable variants are "persistent" data types in that they create a new object that encloses the old object and adds the new member(s); this is similar to how linked lists are built up in Lisp, where elements are prepended by creating a new "cons" cell with a pointer to the new element (the "head") and the old list (the "tail"). Persistent structures of this sort essentially remember the entire history of operations and allow for very easy concurrency — no locks are needed as no shared objects are ever modified.

### Lazy (non-strict) evaluation

Evaluation is strict ("eager") by default. In other words, Scala evaluates expressions as soon as they are available, rather than as needed. However, you can declare a variable non-strict ("lazy") with the `lazy` keyword, meaning that the code to produce the variable's value will not be evaluated until the first time the variable is referenced. Non-strict collections of various types also exist (such as the type `Stream`, a non-strict linked list), and any collection can be made non-strict with the `view` method. Non-strict collections provide a good semantic fit to things like server-produced data, where the evaluation of the code to generate later elements of a list (that in turn triggers a request to a server, possibly located somewhere else on the web) only happens when the elements are actually needed.

### Tail recursion

Functional programming languages commonly provide tail call optimization to allow for extensive use of recursion without stack overflow problems. Limitations in Java bytecode complicate tail call optimization on the JVM. In general, a function that calls itself with a tail call can be optimized, but mutually recursive functions cannot. Trampolines have been suggested as a workaround.[9] Trampoline support has been provided by the Scala library with the object `scala.util.control.TailCalls` since Scala 2.8.0 (released July 14, 2010).

### Case classes and pattern matching

Scala has built-in support for pattern matching, which can be thought of as a more sophisticated, extensible version of a switch statement, where arbitrary data types can be matched (rather than just simple types like integers, booleans and strings), including arbitrary nesting. A special type of class known as a *case class* is provided, which includes automatic support for pattern matching and can be used to model the algebraic data types used in many functional programming languages. (From the perspective of Scala, a case class is simply a normal class for which the compiler automatically adds certain behaviors that could also be provided manually—e.g. definitions of methods providing for deep comparisons and hashing, and destructuring a case class on its constructor parameters during pattern matching.)

An example of a definition of the quicksort algorithm using pattern matching is as follows:

```
def qsort(list: List[Int]): List[Int] = list match {
  case Nil => Nil
  case pivot :: tail =>
    val (smaller, rest) = tail.partition(_ < pivot)
    qsort(smaller) ::: pivot :: qsort(rest)
}
```

The idea here is that we partition a list into the elements less than a pivot and the elements not less, recursively sort each part, and paste the results together with the pivot in between. This uses the same divide-and-conquer strategy of

mergesort and other fast sorting algorithms.

The `match` operator is used to do pattern matching on the object stored in `list`. Each `case` expression is tried in turn to see if it will match, and the first match determines the result. In this case, `Nil` only matches the literal object `Nil`, but `pivot :: tail` matches a non-empty list, and simultaneously *destructures* the list according to the pattern given. In this case, the associated code will have access to a local variable named `pivot` holding the head of the list, and another variable `tail` holding the tail of the list. Note that these variables are read-only, and are semantically very similar to variable bindings established using the `let` operator in Lisp and Scheme.

Pattern matching also happens in local variable declarations. In this case, the return value of the call to `tail.partition` is a tuple — in this case, two lists. (Tuples differ from other types of containers, e.g. lists, in that they are always of fixed size and the elements can be of differing types — although here they are both the same.) Pattern matching is the easiest way of fetching the two parts of the tuple.

The form `_ < pivot` is a declaration of an anonymous function with a placeholder variable; see the section above on anonymous functions.

The list operators `::` (which adds an element onto the beginning of a list, similar to `cons` in Lisp and Scheme) and `:::` (which appends two lists together, similar to `append` in Lisp and Scheme) both appear. Despite appearances, there is nothing "built-in" about either of these operators. As specified above, any string of symbols can serve as function name, and a method applied to an object can be written "infix"-style without the period or parentheses. The line above as written:

```
qsort(smaller) ::: pivot :: qsort(rest)
```

could also be written as follows:

```
qsort(rest).:::(pivot).::(qsort(smaller))
```

in more standard method-call notation. (Methods that end with a colon are right-associative and bind to the object to the right.)

### Partial functions

In the pattern-matching example above, the body of the `match` operator is a partial function, which consists of a series of `case` expressions, with the first matching expression prevailing, similar to the body of a switch statement. Partial functions are also used in the exception-handling portion of a `try` statement:

```
try {
  ...
} catch {
  case nfe:NumberFormatException => { println(nfe); List(0) }
  case _ => Nil
}
```

Finally, a partial function can be used by itself, and the result of calling it is equivalent to doing a `match` over it. For example, the previous code for quicksort can be written as follows:

```
val qsort: List[Int] => List[Int] = {
  case Nil => Nil
  case pivot :: tail =>
    val (smaller, rest) = tail.partition(_ < pivot)
    qsort(smaller) ::: pivot :: qsort(rest)
}
```

Here a read-only *variable* is declared whose type is a function from lists of integers to lists of integers, and bind it to a partial function. (Note that the single parameter of the partial function is never explicitly declared or named.)

However, we can still call this variable exactly as if it were a normal function:

```
scala> qsort(List(6,2,5,9))
res32: List[Int] = List(2, 5, 6, 9)
```

### Object-oriented extensions

Scala is a pure object-oriented language in the sense that every value is an object. Data types and behaviors of objects are described by classes and traits. Class abstractions are extended by subclassing and by a flexible mixin-based composition mechanism to avoid the problems of multiple inheritance.

Traits are Scala's replacement for Java's interfaces. Interfaces in Java versions under 8 are highly restricted, able only to contain abstract function declarations. This has led to criticism that providing convenience methods in interfaces is awkward (the same methods must be reimplemented in every implementation), and extending a published interface in a backwards-compatible way is impossible. Traits are similar to mixin classes in that they have nearly all the power of a regular abstract class, lacking only class parameters (Scala's equivalent to Java's constructor parameters), since traits are always mixed in with a class. The `super` operator behaves specially in traits, allowing traits to be chained using composition in addition to inheritance. The following example is a simple window system:

```
abstract class Window {
  // abstract
  def draw()
}

class SimpleWindow extends Window {
  def draw() {
    println("in SimpleWindow")
    // draw a basic window
  }
}

trait WindowDecoration extends Window { }

trait HorizontalScrollbarDecoration extends WindowDecoration {
  // "abstract override" is needed here in order for "super()" to work
because the parent
  // function is abstract. If it were concrete, regular "override"
would be enough.
  abstract override def draw() {
    println("in HorizontalScrollbarDecoration")
    super.draw()
    // now draw a horizontal scrollbar
  }
}

trait VerticalScrollbarDecoration extends WindowDecoration {
  abstract override def draw() {
    println("in VerticalScrollbarDecoration")
    super.draw()
    // now draw a vertical scrollbar
```

```
  }
}

trait TitleDecoration extends WindowDecoration {
  abstract override def draw() {
    println("in TitleDecoration")
    super.draw()
    // now draw the title bar
  }
}
```

A variable may be declared as follows:

```
val mywin = new SimpleWindow with VerticalScrollbarDecoration with
HorizontalScrollbarDecoration with TitleDecoration
```

The result of calling `mywin.draw()` is

```
in TitleDecoration
in HorizontalScrollbarDecoration
in VerticalScrollbarDecoration
in SimpleWindow
```

In other words, the call to `draw` first executed the code in `TitleDecoration` (the last trait mixed in), then (through the `super()` calls) threaded back through the other mixed-in traits and eventually to the code in `Window` itself, *even though none of the traits inherited from one another*. This is similar to the decorator pattern, but is more concise and less error-prone, as it doesn't require explicitly encapsulating the parent window, explicitly forwarding functions whose implementation isn't changed, or relying on run-time initialization of entity relationships. In other languages, a similar effect could be achieved at compile-time with a long linear chain of implementation inheritance, but with the disadvantage compared to Scala that one linear inheritance chain would have to be declared for each possible combination of the mix-ins.

### Expressive type system

Scala is equipped with an expressive static type system that enforces the safe and coherent use of abstractions. In particular, the type system supports:

- Classes and abstract types as object members
- Structural types
- Path-dependent types
- Compound types
- Explicitly typed self references
- Generic classes
- Polymorphic methods
- Upper and lower type bounds
- Variance
- Annotation
- Views

Scala is able to infer types by usage. This makes most static type declarations optional. Static types need not be explicitly declared unless a compiler error indicates the need. In practice, some static type declarations are included for the sake of code clarity.

### Type enrichment

A common technique in Scala, known as "enrich my library" (formerly "pimp my library", now discouraged due to its connotation), allows new methods to be used as if they were added to existing types. This is similar to the C# concept of extension methods but more powerful, because the technique is not limited to adding methods and can for instance also be used to implement new interfaces. In Scala, this technique involves declaring an implicit conversion from the type "receiving" the method to a new type (typically, a class) that wraps the original type and provides the additional method. If a method cannot be found for a given type, the compiler automatically searches for any applicable implicit conversions to types that provide the method in question.

This technique allows new methods to be added to an existing class using an add-on library such that only code that *imports* the add-on library gets the new functionality, and all other code is unaffected.

The following example shows the enrichment of type `Int` with methods `isEven` and `isOdd`:

```
object MyExtensions {
  implicit class IntPredicates(i: Int) {
    def isEven = i % 2 == 0
    def isOdd  = !isEven
  }
}

import MyExtensions._  // bring implicit enrichment into scope
4.isEven  // -> true
```

Importing the members of `MyExtensions` brings the implicit conversion to extension class `IntPredicates` into scope.[10]

## Concurrency

Scala standard library includes support for the actor model, in addition to the standard Java concurrency APIs. The company called *Typesafe* provides a stack that includes Akka,[11] a separate open source framework that provides actor-based concurrency. Akka actors may be distributed or combined with software transactional memory ("transactors"). Alternative CSP implementations for channel-based message passing are Communicating Scala Objects,[12] or simply via JCSP.

An Actor is like a thread instance with a mailbox. It can be created by `system.actorOf`, overriding the `receive` method to receive messages and using the `!` (exclamation point) method to send a message.[13] The following example shows an EchoServer which can receive messages and then print them.

```
val echoServer = actor(new Act {
  become {
    case msg => println("echo " + msg)
  }
})
echoServer ! "hi"
```

Scala also comes with built-in support for data-parallel programming in the form of Parallel Collections integrated into its Standard Library since version 2.9.0.

The following example shows how to use Parallel Collections to improve performance.[14]

```
val urls = List("http://scala-lang.org",
"https://github.com/scala/scala")
```

```
def fromURL(url: String) = scala.io.Source.fromURL(url)
  .getLines().mkString("\n")

val t = System.currentTimeMillis()
urls.par.map(fromURL(_))
println("time: " + (System.currentTimeMillis - t) + "ms")
```

## Cluster computing

Two significant cluster computing solutions are based on Scala: the open source Apache Spark and the commercial GridGain [15]. Additionally, Apache Kafka, the publish-subscribe message queue popular with Spark and other stream processing technologies, is written in Scala.

## Testing

There are several ways to test code in Scala:

- ScalaTest [16] supports multiple testing styles and can integrate with Java-based testing frameworks
- ScalaCheck [17], a library similar to Haskell's QuickCheck
- specs2 [18], a library for writing executable software specifications
- ScalaMock [19] provides support for testing high-order and curried functions
- JUnit or TestNG, two popular testing frameworks written in Java

## Versions

| Version | Released | Features | Status | Notes |
|---|---|---|---|---|
| **2.0** [20] | 12-Mar-2006 | _ | _ | _ |
| **2.1.8** [20] | 23-Aug-2006 | _ | _ | _ |
| **2.3.0** [20] | 23-Nov-2006 | _ | _ | _ |
| **2.4.0** [20] | 09-Mar-2007 | _ | _ | _ |
| **2.5.0** [20] | 02-May-2007 | _ | _ | _ |
| **2.6.0** [20] | 27-Jul-2007 | _ | _ | _ |
| **2.7.0** [20] | 07-Feb-2008 | _ | _ | _ |
| **2.8.0** [21] | 14-Jul-2010 | Revision the common, uniform, and all-encompassing framework for collection types. | _ | _ |
| **2.9.0** [22] | 12-May-2011 | _ | _ | _ |

| | | | | |
|---|---|---|---|---|
| 2.10 [23] | 04-Jan-2013 | • Value Classes [24]<br>• Implicit Classes [25]<br>• String Interpolation [26]<br>• Futures and Promises [27]<br>• Dynamic and applyDynamic [28]<br>• Dependent method types: * def identity(x: AnyRef): x.type = x // the return type says we return exactly what we got<br>• New ByteCode emitter based on ASM: Can target JDK 1.5, 1.6 and 1.7 / Emits 1.6 bytecode by default / Old 1.5 backend is deprecated<br>• A new Pattern Matcher: rewritten from scratch to generate more robust code (no more exponential blow-up!) / code generation and analyses are now independent (the latter can be turned off with -Xno-patmat-analysis)<br>• Scaladoc Improvements<br>• Implicits (-implicits flag)<br>• Diagrams (-diagrams flag, requires graphviz)<br>• Groups (-groups)<br>• Modularized Language features [29]<br>• Parallel Collections [30] are now configurable with custom thread pools<br>• Akka Actors now part of the distribution\\scala.actors have been deprecated and the akka implementation is now included in the distribution.<br>• Performance Improvements: Faster inliner / Range#sum is now O(1)<br>• Update of ForkJoin library<br>• Fixes in immutable TreeSet/TreeMap<br>• Improvements to PartialFunctions<br>• Addition of ??? and NotImplementedError<br>• Addition of IsTraversableOnce + IsTraversableLike type classes for extension methods<br>• Deprecations and cleanup<br>• Floating point and octal literal syntax deprecation<br>• Removed scala.dbc<br><br>Experimental features<br>• Scala Reflection [31]<br>• Macros [32] | – | – |
| **2.10.2 [33]** | 06-Jun-2013 | – | – | – |
| **2.10.3 [34]** | 01-Oct-2013 | – | – | – |
| **2.11.0 [35]** | 21-Apr-2014 | – | Current | – |

## Comparison with other JVM languages

Scala is often compared with Groovy and Clojure, two other programming languages also using the JVM. Substantial differences between these languages are found in the type system, in the extent to which each language supports object-oriented and functional programming, and in the similarity of their syntax to the syntax of Java.

Scala is statically typed, while both Groovy and Clojure are dynamically typed. This makes the type system more complex and difficult to understand but allows almost all type errors to be caught at compile-time and can result in significantly faster execution. By contrast, dynamic typing requires more testing to ensure program correctness and is generally slower in order to allow greater programming flexibility and simplicity. In regard to speed differences, current versions of Groovy and Clojure allow for optional type annotations to help programs avoid the overhead of dynamic typing in cases where types are practically static. This overhead is further reduced when using recent versions of the JVM, which has been enhanced with an "invoke dynamic" instruction for methods that are defined

with dynamically typed arguments. These advances reduce the speed gap between static and dynamic typing, although a statically typed language, like Scala, is still the preferred choice when excecution efficiency is very important.

In regard to programming paradigms, Scala inherits the object-oriented model of Java and extends it in various ways. Groovy, while also strongly object-oriented is more focused in reducing verbosity. In Clojure, object-oriented programming is deemphasised with fuctional programming being the primary strength of the language. Scala also has many functional programming facilities, including features found in advanced functional languages like Haskell, and tries to be agnostic between the two paradigms, letting the developer choose between the two paradigms or, more frequently, some combination thereof.

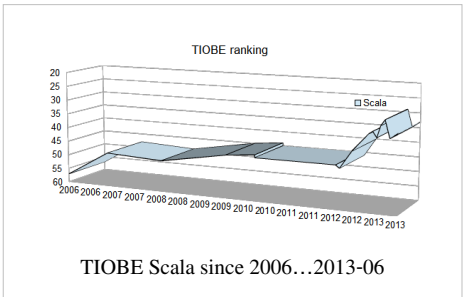
In regard to syntax similarity with Java, Scala inherits a lot of Java's syntax, as is the case with Groovy. Clojure on the other hand follows the Lisp syntax, which is very different in both appearence and philosophy, making the transition from Java very difficult. However, learning Scala is also considered difficult because of its many advanced features. This is not the case with Groovy, despite the fact that it is also a feature-rich language, mainly because it was designed to be primarily a scripting language.

# Adoption

## Language rankings

Scala was voted the most popular JVM scripting language at the 2012 JavaOne conference.

As of 2013[36], all JVM-based derivatives (Scala/Groovy/Clojure) are significantly less popular than the original Java language itself which is usually ranked first or second, and which is also simultaneously evolving over time.




TIOBE Scala since 2006…2013-06

As of December 2013, the TIOBE index of programming language popularity shows Scala in 31st place with 0.342% of programmer mindshare (as measured by internet search engine rankings and similar publication-counting), while it was below the top 50 threshold the year before. Scala is now ahead of functional languages Haskell (50th) and Erlang (>50), as well as JVM competitors Groovy (47th) and Clojure (>50).

Another measure, the RedMonk Programming Language Rankings, as of June 2013 placed Scala 12th, based on 15th position in terms of number of GitHub projects and 16th in terms of number of questions tagged on Stack Overflow. (Groovy was 18th place; Clojure was 22nd.) Here, Scala is shown clearly behind a first-tier group of 11 languages (including Java, C, Python, PHP, Ruby, etc.), but leading a second-tier group.

The ThoughtWorks Technology Radar, which is an opinion based half-yearly report of a group of senior technologists, recommends Scala adoption in its languages and frameworks category.

According to Indeed.com Job Trends, Scala demand has been rapidly increasing [37] since 2010, trending ahead of Clojure but behind Groovy.

## Companies

| Website online | | | | Back office | | | |
|---|---|---|---|---|---|---|---|
| Atlassian | … | … | … | … | … | … | … |
| eBay Inc. | … | … | … | CSC | EDF Trading | S&P Capital IQ | … |
| Foursquare | … | … | … | … | | HSBC | |
| Heluna [38] | HolidayCheck [39] | … | Klout | IBM | Intel | Juniper Networks | Marktplaats.nl |
| LinkedIn [40] | LivingSocial | … | Metafor Software | Mind Candy | Micronautics Research | ITV [41] | |
| Meetup.com | Novell | | OPower [42] | Nasa | Nature | New York Times | newBrandAnalytics |
| Plot | precog.com | Process Street [43] | Reaktor [44] | | Office Depot | Peerius.com | Quora |
| Remember The Milk | SAP AG | Secondmarket.com | Siemens AG | Reverb Technologies, Inc. | Rhinofly | Sears | Sony |
| Thatcham Motor [45] | The Guardian | TicketFly | Twitter | StackMob | Stanford PPL [46] | TomTom | UBS |
| Wattzon.com | Wordnik.com | Workday.com | Yahoo! | Walmart | Xebia [47] | Xerox | Zeebox [48] |

In April 2009, Twitter announced that it had switched large portions of its backend from Ruby to Scala and intended to convert the rest.

Gilt uses Scala and Play Framework.[49]

Foursquare uses Scala and Lift.[50]

In April 2011, *The Guardian* newspaper's website guardian.co.uk announced that it was switching from Java to Scala, starting with the Content API for selecting and collecting news content. The website is one of the highest-traffic English-language news websites and, according to its editor, has the second largest online readership of any English-language newspaper in the World, after the *New York Times*.

Swiss bank UBS approved Scala for general production usage.

LinkedIn uses the Scalatra microframework to power its Signal API.

Meetup uses Unfiltered toolkit for real-time APIs.

Remember the Milk uses Unfiltered toolkit, Scala and Akka for public API and real time updates.

## Criticism

In November, 2011, Yammer moved away from Scala for reasons that included the learning curve for new team members and incompatibility from one version of the Scala compiler to the next. Others criticize Scala for lack of readability regarding implicit parameters, that it is not possible to avoid Scala's more arcane features once one needs to pull in a library that uses them, and that overall, "Scala's difficulty outweighs its value."

# References


**[1]** Martin Odersky et al., An Overview of the Scala Programming Language, 2nd Edition

[2] http://www.scala-lang.org/

[3] Martin Odersky, "A Brief History of Scala" (http://www.artima.com/weblogs/viewpost.jsp?thread=163733), Artima.com weblogs, June 9, 2006

**[4]** Martin Odersky, "The Scala Language Specification Version 2.7"

[5] Expunged the .net backend. by paulp · Pull Request #1718 · scala/scala · GitHub (https://github.com/scala/scala/pull/1718). Github.com (2012-12-05). Retrieved on 2013-11-02.

[6] Scala Team Wins ERC Grant (http://www.scala-lang.org/node/8579)

[7] Scala IDE for Eclipse: Developing for Android (http://www.assembla.com/wiki/show/scala-ide/Developing_for_Android)

[8] Scala's built-in control structures such as `if` or `while` cannot be re-implemented. There is a research project, Scala-Virtualized, that aimed at removing these restrictions: Adriaan Moors, Tiark Rompf, Philipp Haller and Martin Odersky. Scala-Virtualized (http://dl.acm.org/citation.cfm?id=2103769). *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, 117–120. July 2012.

[9] Tail calls, @tailrec and trampolines (http://blog.richdougherty.com/2009/04/tail-calls-tailrec-and-trampolines.html)

[10] Implicit classes were introduced in Scala 2.10 to make method extensions more concise. This is equivalent to adding a method `implicit def IntPredicate(i: Int) = new IntPredicate(i)`. The class can also be defined as `implicit class IntPredicates(val i: Int) extends AnyVal { ... }`, producing a so-called *value class*, also introduced in Scala 2.10. The compiler will then eliminate actual instantiations and generate static methods instead, allowing extension methods to have virtually no performance overhead.

[11] What is Akka? (http://doc.akka.io/docs/akka/snapshot/intro/what-is-akka.html), Akka online documentation

[12] Communicating Scala Objects (http://users.comlab.ox.ac.uk/bernard.sufrin/CSO/cpa2008-cso.pdf), Bernard Sufrin, Communicating Process Architectures 2008

[13] http://www.scala-tour.com/#/using-actor

[14] http://www.scala-tour.com/#/parallel-collection

[15] http://gridgain.com/

[16] http://www.scalatest.org/

[17] https://github.com/rickynils/scalacheck

[18] http://specs2.org/

[19] http://scalamock.org/

[20] http://www.scala-lang.org/download/changelog.html#2.

[21] http://www.scala-lang.org/download/changelog.html#2.8.0

[22] http://www.scala-lang.org/download/changelog.html#2.9.0

[23] http://www.scala-lang.org/download/changelog.html

[24] http://docs.scala-lang.org/overviews/core/value-classes.html

[25] http://docs.scala-lang.org/sips/pending/implicit-classes.html

[26] http://docs.scala-lang.org/overviews/core/string-interpolation.html

[27] http://docs.scala-lang.org/overviews/core/futures.html

[28] http://docs.scala-lang.org/sips/pending/type-dynamic.html

[29] http://docs.scala-lang.org/sips/pending/modularizing-language-features.html

[30] http://docs.scala-lang.org/overviews/parallel-collections/overview.html

[31] http://docs.scala-lang.org/overviews/reflection/overview.html

[32] http://docs.scala-lang.org/overviews/macros/overview.html

[33] http://www.scala-lang.org/news/2013/06/06/release-notes-v2.10.2.html

[34] http://www.scala-lang.org/news/2013/10/01/release-notes-v2.10.3.html

[35] http://scala-lang.org/news/2014/04/21/release-notes-2.11.0.html

[36] http://en.wikipedia.org/w/index.php?title=Scala_(programming_language)&action=edit

[37] http://www.indeed.com/trendgraph/jobgraph.png?q=scala&relative=1

[38] https://heluna.com/

[39] http://www.holidaycheck.com/

[40] http://www.scala-lang.org/node/1658#LinkedIn
[41] http://www.itv.com/
[42] http://www.scala-lang.org/node/1658#opower
[43] http://process.st/
[44] http://www.scala-lang.org/node/1658#Reaktor
[45] http://www.scala-lang.org/node/1658#thatcham
[46] http://www.scala-lang.org/node/3200
[47] http://www.scala-lang.org/node/1658#Xebia
[48] http://www.scala-lang.org/node/11428
[49] July 15, 2013 | Play Framework, Akka and Scala at Gilt Groupe | Typesafe (https://www.typesafe.com/blog/play_framework_akka_and_scala_at_gilt)
[50] Scala, Lift, and the Future (http://www.grenadesandwich.com/blog/steven/2009/11/27/scala-lift-and-future)

## Further reading

- Suereth, Joshua D. (Spring 2011). *Scala in Depth*. Manning Publications. p. 225. ISBN 978-1-935182-70-2.
- Meredith, Gregory (2011). *Monadic Design Patterns for the Web* (http://github.com/leithaus/XTrace/blob/monadic/src/main/book/content/monadic.pdf) (1st ed.). p. 300.
- Raychaudhuri, Nilanjan (Fall 2011). *Scala in Action* (1st ed.). Manning. p. 525. ISBN 978-1-935182-75-7.
- Wampler, Dean; Payne, Alex (September 15, 2009). *Programming Scala: Scalability = Functional Programming + Objects* (http://oreilly.com/catalog/9780596155957/) (1st ed.). O'Reilly Media. p. 448. ISBN 0-596-15595-6.
- Odersky, Martin; Spoon, Lex; Venners, Bill (December 13, 2010). *Programming in Scala: A Comprehensive Step-by-step Guide* (http://www.artima.com/shop/programming_in_scala_2ed) (2nd ed.). Artima Inc. pp. 883/852. ISBN 978-0-9815316-4-9.
- Pollak, David (May 25, 2009). *Beginning Scala* (http://www.apress.com/book/view/9781430219897/) (1st ed.). Apress. p. 776. ISBN 1-4302-1989-0.
- Perrett, Tim (July 2011). *Lift in Action* (1st ed.). Manning. p. 450. ISBN 978-1-935182-80-1.
- Loverdos, Christos; Syropoulos, Apostolos (September 2010). *Steps in Scala: An Introduction to Object-Functional Programming* (http://www.cambridge.org/gb/knowledge/isbn/item2713441/?site_locale=en_GB) (1st ed.). Cambridge University Press. pp. xviii + 485. ISBN 978-0-521-74758-5.
- Subramaniam, Venkat (July 28, 2009). *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine* (http://www.pragprog.com/titles/vsscala/programming-scala) (1st ed.). Pragmatic Bookshelf. p. 250. ISBN 1-934356-31-X.
- Horstmann, Cay (March 2012). *Scala for the Impatient* (http://www.informit.com/title/0321774094) (1st ed.). Addison-Wesley Professional. p. 360. ISBN 0-321-77409-4.

## External links

- Official website (http://www.scala-lang.org)
- Typesafe company website (http://typesafe.com/)
- Scala Forum (http://scala-forum.org/)
- Scala communities around the globe (http://www.scala-tribes.org/)
- Scala IDE (http://www.scala-ide.org/), open source Scala IDE for Eclipse
- Scala Tour (http://www.scala-tour.com/), open source Scala Tour
- Interactive Tour (http://www.scalatutorials.com/tour), a tour of Scala

# Event-driven programming

**Programming paradigms**

- Action
- Agent-oriented
- Aspect-oriented
- Automata-based
- Concurrent computing
  - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
  - Constraint
  - Dataflow
    - Flow-based
    - Cell-oriented (spreadsheets)
    - Reactive
  - Functional
    - Functional logic
  - Logic
    - Abductive logic
    - Answer set
    - Constraint logic
    - Functional logic
    - Inductive logic
- End-user programming
- Event-driven
  - Service-oriented
  - Time-driven
- Expression-oriented
- Feature-oriented
- Function-level (contrast: Value-level)
- Generic
- Imperative (contrast: Declarative)
  - Procedural
- Language-oriented
  - Natural language programming
  - Discipline-specific
  - Domain-specific
  - Grammar-oriented
    - Dialecting
  - Intentional
- Metaprogramming
  - Automatic
  - Reflective
    - Attribute-oriented
  - Homoiconic
  - Template
    - Policy-based

- Non-structured (contrast: Structured)
  - Array
- Nondeterministic
- Parallel computing
  - Process-oriented
- Point-free style
  - Concatenative
- Semantic
- Structured (contrast: Non-structured)
  - Block-structured
  - Modular (contrast: Monolithic)
  - Object-oriented (OOP)
    - By separation of concerns:
      - Aspect-oriented
      - Role-oriented
      - Subject-oriented
    - Class-based
    - Prototype-based
  - Recursive
- Value-level (contrast: Function-level)
- Probabilistic
- Concept

- v
- t
- e [1]

In computer programming, **event-driven programming** is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads. Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered around performing certain actions in response to user input.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected. In embedded systems the same may be achieved using hardware interrupts instead of a constantly running main loop. Event-driven programs can be written in any programming language, although the task is easier in languages that provide high-level abstractions, such as closures.

# Event handlers

## A trivial event handler

Because the code for checking for events and the main loop do not depend on the application, many programming frameworks take care of their implementation and expect the user to provide only the code for the event handlers. In this simple example there may be a call to an event handler called `OnKeyEnter()` that includes an argument with a string of characters, corresponding to what the user typed before hitting the ENTER key. To add two numbers, storage outside the event handler must be used. The implementation might look like below.

```
globally declare the counter K and the integer T.
OnKeyEnter(character C)
{
   convert C to a number N
   if K is zero store N in T and increment K
```

```
   otherwise add N to T, print the result and reset K to zero
}
```

While keeping track of history is straightforward in a batch program, it requires special attention and planning in an event-driven program.

### Exception handlers

In PL/1, even though a program itself may not be predominantly event driven, certain abnormal events such as a hardware error, overflow or "program checks" may occur that possibly prevent further processing. Exception handlers may be provided by "ON statements" in (unseen) callers to provide housekeeping routines to clean up afterwards before termination.

### Creating event handlers

The first step in developing an event-driven program is to write a series of subroutines, or methods, called event-handler routines. These routines handle the events to which the main program will respond. For example, a single left-button mouse-click on a command button in a GUI program may trigger a routine that will open another window, save data to a database or exit the application. Many modern day programming environments provide the programmer with event templates so that the programmer only needs to supply the event code.

The second step is to bind event handlers to events so that the correct function is called when the event takes place. Graphical editors combine the first two steps: double-click on a button, and the editor creates an (empty) event handler associated with the user clicking the button and opens a text window so you can edit the event handler.

The third step in developing an event-driven program is to write the main loop. This is a function that checks for the occurrence of events, and then calls the matching event handler to process it. Most event-driven programming environments already provide this main loop, so it need not be specifically provided by the application programmer. RPG, an early programming language from IBM, whose 1960s design concept was similar to event driven programming discussed above, provided a built-in main I/O loop (known as the "program cycle") where the calculations responded in accordance to 'indicators' (flags) that were set earlier in the cycle.

## Criticism and best practice

Event-driven programming is widely used in graphical user interfaces, for instance the Android concurrency frameworks are designed using the Half-Sync/Half-Async pattern, where a combination of a single-threaded event loop processing (for the main UI thread) and synchronous threading (for background threads) is used. This is because the UI-widgets are not thread-safe, and while they are extensible, there is no way to guarantee that all the implementations are thread-safe, thus single-thread model alleviates this issue.

The design of those toolkits has been criticized, e.g., by Miro Samek, for promoting an over-simplified model of event-action, leading programmers to create error prone, difficult to extend and excessively complex application code. He writes,

> Such an approach is fertile ground for bugs for at least three reasons:
>
> 1. It always leads to convoluted conditional logic.
> 2. Each branching point requires evaluation of a complex expression.
> 3. Switching between different modes requires modifying many variables, which all can easily lead to inconsistencies.
>
> — Miro Samek, *Who Moved My State?* [2], C/C++ Users Journal, The Embedded Angle column (April 2003)

and advocates the use of state machines as a viable alternative.Wikipedia:Please clarify

## Stackless threading

An event driven approach is used in hardware description languages. A thread context only needs a cpu stack while actively processing an event, once done the cpu can move on to process other event-driven threads, that allows an extremely large number of threads to be handled. This is essentially a Finite-state machine approach.

## References

[1] http://en.wikipedia.org/w/index.php?title=Template:Programming_paradigms&action=edit
[2] http://www.ddj.com/cpp/184401643

## External links

- Description (http://c2.com/cgi/wiki?EventDrivenProgramming) from Portland Pattern Repository
- Event-Driven Programming: Introduction, Tutorial, History (http://eventdrivenpgm.sourceforge.net/), tutorial by Stephen Ferg
- Event Driven Programming (http://www.alan-g.me.uk/l2p/tutevent.htm), tutorial by Alan Gauld
- Event Collaboration (http://www.martinfowler.com/eaaDev/EventCollaboration.html), article by Martin Fowler
- Transitioning from Structured to Event-Driven Programming (http://www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=504), article by Ben Watson
- Rethinking Swing Threading (http://today.java.net/pub/a/today/2003/10/24/swing.html), article by Jonathan Simon
- The event driven programming style (http://www.csse.uwa.edu.au/cnet/eventdriven.html), article by Chris McDonald
- Event Driven Programming using Template Specialization (http://codeproject.com/cpp/static-callbacks.asp), article by Christopher Diggins
- Concepts and Architecture of Vista - a Multiparadigm Programming Environment (http://www.swe.uni-linz.ac.at/people/schiffer/se-94-17/se-94-17.htm), article by Stefan Schiffer and Joachim Hans Fröhlich
- Event-Driven Programming and Agents (http://docs.eiffel.com/book/method/8-event-driven-programming-and-agents), chapter
- LabWindows/CVI Resources (http://zone.ni.com/devzone/devzone.nsf/webcategories/FCE7EA7ECA51169C862567A9005878EA)
- Distributed Publish/Subscribe Event System (http://www.codeplex.com/pubsub), an open source example which is in production on MSN.com and Microsoft.com
- Sinelabore.com (http://www.sinelabore.com/), for C-Code generation from UML State-Charts for Embedded Systems
- StateWizard (http://www.intelliwizard.com), a ClassWizard-like event-driven state machine framework and tool running in popular IDEs under open-source license

# Concurrent Haskell

**Concurrent Haskell** extends[1] Haskell 98 with explicit concurrency. The two main concepts underpinning Concurrent Haskell are:

- A primitive type `MVar a` implementing a bounded/single-place asynchronous channel, which is either empty or holds a value of type `a`.
- The ability to spawn a concurrent thread via the `forkIO` primitive.

Built atop this is a collection of useful concurrency and synchronisation abstractions[2] such as unbounded channels, semaphores and sample variables.

Default Haskell threads have very low overheads: creation, context-switching and scheduling are all internal to the Haskell runtime. These Haskell-level threads are mapped onto a configurable number of OS-level threads, usually one per processor core.

## Software Transactional Memory

The recently introduced Software Transactional Memory (STM)[3] extension[4] to the Glasgow Haskell Compiler reuses the process forking primitives of Concurrent Haskell. STM however:

- eschews `MVars` in favour of `TVars`.
- introduces the `retry` and `orElse` primitives, allowing alternative atomic actions to be *composed* together.

## STM monad

The STM monad is an implementation of Software Transactional Memory in Haskell. It is implemented in the GHC compiler, and allows for mutable variables to be modified in transactions. An example of a transaction might be in a banking application. One function that would be needed would be a transfer function, which takes money from one account, and puts it into another account. In the IO monad, this might look like this:

```
type Account = IORef Integer

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    fromVal <- readIORef from
    toVal   <- readIORef to
    writeIORef from (fromVal - amount)
    writeIORef to (toVal + amount)
```

This might work some of the time, but causes problems in concurrent situations where multiple transfers might be taking place on the same account at the same time. If there were two transfers transferring money from account 'from', and both calls to transfer ran the

```
fromVal <- readIORef from
```

line before either of them had written their new values, it is possible that money would be put into the other two accounts, with only one of the amounts being transferred being removed from account 'from', thus creating a race condition. This would leave the banking application in an inconsistent state. A traditional solution to such a problem is locking. For instance, one could place locks around modifications to an account to ensure that credits and debits occur atomically. In Haskell, locking is accomplished with MVars:

```haskell
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)
```

Using such procedures will ensure that money will never be lost or gained due to improper interleaving of reads and writes to any individual account. However, if one tries to compose them together to create a procedure like transfer:

```haskell
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit amount from
    credit amount to
```

, a race condition still exists: the first account may be debited, then execution of the thread may be suspended, leaving the accounts as a whole in an inconsistent state. Thus, additional locks must be added to ensure correctness of composite operations, and in the worst case, one might need to simply lock all accounts regardless of how many are used in a given operation.

To avoid this, one can use the STM monad, which allows one to write atomic transactions. This means that all operations inside the transaction fully complete, without any other threads modifying the variables that our transaction is using, or it fails, and the state is rolled back to where it was before the transaction was begun. In short, atomic transactions either complete fully, or it is as if they were never run at all. The lock-based code above translates in a relatively straightforward way:

```haskell
type Account = TVar Integer

credit :: Integer -> Account -> STM ()
credit amount account = do
    current <- readTVar account
    writeTVar account (current + amount)

debit :: Integer -> Account -> STM ()
debit amount account = do
    current <- readTVar account
    writeTVar account (current - amount)

transfer :: Integer -> Account -> Account -> STM ()
transfer amount from to = do
    debit amount from
    credit amount to
```

The return types of `STM ()` may be taken to indicate that we are composing scripts for transactions. When the time comes to actually execute such a transaction, a function `atomically :: STM a -> IO a` is used. The above

implementation will make sure that no other transactions interfere with the variables it is using (from and to) while it is executing, allowing the developer to be sure that race conditions like that above are not encountered. More improvements can be made to make sure that some other "business logic" is maintained in the system, i.e. that the transaction should not try to take money from an account until there is enough money in it:

```
transfer :: Integer -> Account -> Account -> STM ()
transfer amount from to = do
    fromVal <- readTVar from
    if (fromVal - amount) >= 0
        then do
                debit amount from
                credit amount to
        else retry
```

Here the `retry` function has been used, which will roll back a transaction, and try it again. Retrying in STM is smart in that it won't try to run the transaction again until one of the variables it references during the transaction has been modified by some other transactional code. This makes the STM monad quite efficient.

An example program using the transfer function might look like this:

```
module Main where

import Control.Concurrent (forkIO)
import Control.Concurrent.STM
import Control.Monad (forever)
import System.Exit (exitSuccess)

type Account = TVar Integer

main = do
    bob <- newAccount 10000
    jill <- newAccount 4000
    repeatIO 2000 $ forkIO $ atomically $ transfer 1 bob jill
    forever $ do
        bobBalance <- atomically $ readTVar bob
        jillBalance <- atomically $ readTVar jill
        putStrLn ("Bob's balance: " ++ show bobBalance ++ ", Jill's
balance: " ++ show jillBalance)
        if bobBalance == 8000
            then exitSuccess
            else putStrLn "Trying again."

repeatIO :: Integer -> IO a -> IO a
repeatIO 1 m = m
repeatIO n m = m >> repeatIO (n - 1) m

newAccount :: Integer -> IO Account
newAccount amount = newTVarIO amount

transfer :: Integer -> Account -> Account -> STM ()
```

```
transfer amount from to = do
    fromVal <- readTVar from
    if (fromVal - amount) >= 0
        then do
               debit amount from
               credit amount to
        else retry

credit :: Integer -> Account -> STM ()
credit amount account = do
    current <- readTVar account
    writeTVar account (current + amount)

debit :: Integer -> Account -> STM ()
debit amount account = do
    current <- readTVar account
    writeTVar account (current - amount)
```

'

which should print out "Bill's balance: 8000, Jill's balance: 6000". Here the `atomically` function has been used to run STM actions in the IO monad.

## References

[1] Simon Peyton Jones, Andrew D. Gordon, and Sigbjorn Finne. Concurrent Haskell (http://www.haskell.org/ghc/docs/papers/concurrent-haskell.ps.gz). *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL).* 1996. (Some sections are out of date with respect to the current implementation.)

[2] The Haskell Hierarchical Libraries (http://www.haskell.org/ghc/docs/latest/html/libraries/), Control.Concurrent (http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent.html)

[3] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions (http://research.microsoft.com/Users/simonpj/papers/stm/stm.pdf). *ACM Symposium on Principles and Practice of Parallel Programming 2005* (PPoPP'05). 2005.

[4] Control.Concurrent.STM (http://hackage.haskell.org/packages/archive/stm/latest/doc/html/Control-Concurrent-STM.html)

# Software transactional memory

In computer science, **software transactional memory** (**STM**) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It is an alternative to lock-based synchronization. A transaction in this context occurs when a piece of code executes a series of reads and writes to shared memory. These reads and writes logically occur at a single instant in time; intermediate states are not visible to other (successful) transactions. The idea of providing hardware support for transactions originated in a 1986 paper by Tom Knight.[1] The idea was popularized by Maurice Herlihy and J. Eliot B. Moss.[2] In 1995 Nir Shavit and Dan Touitou extended this idea to software-only transactional memory (STM).[3] Since 2005, STM has been the focus of intense research and support for practical implementations is growing.

## Performance

Unlike the locking techniques used in most modern multithreaded applications, STM is very optimistic: a thread completes modifications to shared memory without regard for what other threads might be doing, recording every read and write that it is performing in a log. Instead of placing the onus on the writer to make sure it does not adversely affect other operations in progress, it is placed on the reader, who after completing an entire transaction verifies that other threads have not concurrently made changes to memory that it accessed in the past. This final operation, in which the changes of a transaction are validated and, if validation is successful, made permanent, is called a *commit*. A transaction may also *abort* at any time, causing all of its prior changes to be rolled back or undone. If a transaction cannot be committed due to conflicting changes, it is typically aborted and re-executed from the beginning until it succeeds.

The benefit of this optimistic approach is increased concurrency: no thread needs to wait for access to a resource, and different threads can safely and simultaneously modify disjoint parts of a data structure that would normally be protected under the same lock.

However, in practice STM systems also suffer a performance hit compared to fine-grained lock-based systems on small numbers of processors (1 to 4 depending on the application). This is due primarily to the overhead associated with maintaining the log and the time spent committing transactions. Even in this case performance is typically no worse than twice as slow. Advocates of STM believe this penalty is justified by the conceptual benefits of STMWikipedia:Citation needed.

Theoretically, the worst case space and time complexity of $n$ concurrent transactions is $O(n)$. Actual needs depend on implementation details (one can make transactions fail early enough to avoid overhead), but there will also be cases, albeit rare, where lock-based algorithms have better time complexity than software transactional memory.

## Conceptual advantages and disadvantages

In addition to their performance benefits, STM greatly simplifies conceptual understanding of multithreaded programs and helps make programs more maintainable by working in harmony with existing high-level abstractions such as objects and modules. Lock-based programming has a number of well-known problems that frequently arise in practice:

- Locking requires thinking about overlapping operations and partial operations in distantly separated and seemingly unrelated sections of code, a task which is very difficult and error-prone.
- Locking requires programmers to adopt a locking policy to prevent deadlock, livelock, and other failures to make progress. Such policies are often informally enforced and fallible, and when these issues arise they are insidiously difficult to reproduce and debug.
- Locking can lead to priority inversion, a phenomenon where a high-priority thread is forced to wait for a low-priority thread holding exclusive access to a resource that it needs.

In contrast, the concept of a memory transaction is much simpler, because each transaction can be viewed in isolation as a single-threaded computation. Deadlock and livelock are either prevented entirely or handled by an external transaction manager; the programmer needs hardly worry about it. Priority inversion can still be an issue, but high-priority transactions can abort conflicting lower priority transactions that have not already committed.

On the other hand, the need to abort failed transactions also places limitations on the behavior of transactions: they cannot perform any operation that cannot be undone, including most I/O. Such limitations are typically overcome in practice by creating buffers that queue up the irreversible operations and perform them at a later time outside of any transaction. In Haskell, this limitation is enforced at compile time by the type system.

## Composable operations

In 2005, Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy described an STM system built on Concurrent Haskell that enables arbitrary atomic operations to be composed into larger atomic operations, a useful concept impossible with lock-based programming. To quote the authors:

> Perhaps the most fundamental objection [...] is that *lock-based programs do not compose*: correct fragments may fail when combined. For example, consider a hash table with thread-safe insert and delete operations. Now suppose that we want to delete one item A from table t1, and insert it into table t2; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement. [...] In short, operations that are individually correct (insert, delete) cannot be composed into larger correct operations.
>
> —Tim Harris et al., "Composable Memory Transactions", Section 2: Background, pg.2

With STM, this problem is simple to solve: simply wrapping two operations in a transaction makes the combined operation atomic. The only sticking point is that it's unclear to the caller, who is unaware of the implementation details of the component methods, when they should attempt to re-execute the transaction if it fails. In response, the authors proposed a **retry** command which uses the transaction log generated by the failed transaction to determine which memory cells it read, and automatically retries the transaction when one of these cells is modified, based on the logic that the transaction will not behave differently until at least one such value is changed.

The authors also proposed a mechanism for composition of *alternatives*, the **orElse** function. It runs one transaction and, if that transaction does a *retry*, runs a second one. If both retry, it tries them both again as soon as a relevant change is made.Wikipedia:Please clarify This facility, comparable to features such as the POSIX networking *select()* call, allows the caller to wait on any one of a number of events simultaneously. It also simplifies programming interfaces, for example by providing a simple mechanism to convert between blocking and nonblocking operations.

This scheme has been implemented in the Glasgow Haskell Compiler.

## Proposed language support

The conceptual simplicity of STMs enables them to be exposed to the programmer using relatively simple language syntax. Tim Harris and Keir Fraser's "Language Support for Lightweight Transactions" proposed the idea of using the classical *conditional critical region* (CCR) to represent transactions. In its simplest form, this is just an "atomic block", a block of code which logically occurs at a single instant:

```
// Insert a node into a doubly linked list atomically
  atomic {
     newNode->prev = node;
     newNode->next = node->next;
     node->next->prev = newNode;
     node->next = newNode;
```

```
 }
```

When the end of the block is reached, the transaction is committed if possible, or else aborted and retried. CCRs also permit a *guard condition*, which enables a transaction to wait until it has work to do:

```
atomic (queueSize > 0) {
     remove item from queue and use it
 }
```

If the condition is not satisfied, the transaction manager will wait until another transaction has made a *commit* that affects the condition before retrying. This loose coupling between producers and consumers enhances modularity compared to explicit signaling between threads. "Composable Memory Transactions" took this a step farther with its **retry** command (discussed above), which can, at any time, abort the transaction and wait until *some value* previously read by the transaction is modified before retrying. For example:

```
atomic {
     if (queueSize > 0) {
         remove item from queue and use it
     } else {
         retry
     }
 }
```

This ability to retry dynamically late in the transaction simplifies the programming model and opens up new possibilities.

One issue is how exceptions behave when they propagate outside of transactions. In "Composable Memory Transactions", the authors decided that this should abort the transaction, since exceptions normally indicate unexpected errors in Concurrent Haskell, but that the exception could retain information allocated by and read during the transaction for diagnostic purposes. They stress that other design decisions may be reasonable in other settings.

## Transactional locking

STM can be implemented as a lock-free algorithm or it can use locking. There are two types of locking schemes: In encounter-time locking (Ennals, Saha, and Harris), memory writes are done by first temporarily acquiring a lock for a given location, writing the value directly, and logging it in the undo log. Commit-time locking locks memory locations only during the commit phase.

A commit-time scheme named "Transactional Locking II" implemented by Dice, Shalev, and Shavit uses a global version clock. Every transaction starts by reading the current value of the clock and storing it as the read-version. Then, on every read or write, the version of the particular memory location is compared to the read-version; and, if it's greater, the transaction is aborted. This guarantees that the code is executed on a consistent snapshot of memory. During commit, all write locations are locked, and version numbers of all read and write locations are re-checked. Finally, the global version clock is incremented, new write values from the log are written back to memory and stamped with the new clock version.

An increasingly utilized method to manage transactional conflicts in Transactional memory, and especially in STM, is Commitment ordering (also called Commit ordering; CO). It is utilized for achieving serializability optimistically (i.e., without blocking upon conflict, and only locking for commit) by "commit order" (e.g., Ramadan et al. 2009,[4] and Zhang et al. 2006[5]). Serializability is the basis for the correctness of (concurrent transactions and) transactional memory. Tens of STM articles on "commit order" have already been published, and the technique is encumbered by a number of patents.

With CO the desired serializability property is achieved by committing transactions only in chronological order that is compatible with the precedence order (as determined by chronological orders of operations in conflicts) of the respective transactions. To enforce CO some implementation of the *Generic local CO algorithm* needs to be utilized. The patent abstract quoted above describes a general implementation of the algorithm with a pre-determined commit order (this falls into the category of "CO generic algorithm with real-time constraints").

## Implementation issues

One problem with implementing software transactional memory with optimistic reading is that it's possible for an incomplete transaction to read inconsistent state (that is, to read a mixture of old and new values written by another transaction). Such a transaction is doomed to abort if it ever tries to commit, so this does not violate the consistency condition enforced by the transactional system, but it's possible for this "temporary" inconsistent state to cause a transaction to trigger a fatal exceptional condition such as a segmentation fault or even enter an endless loop, as in the following contrived example from Figure 4 of "Language Support for Lightweight Transactions":

| Transaction A | Transaction B |
|---|---|
| `atomic {`<br>`  if (x != y)`<br>`  while (true) {`<br>`  }`<br>`}` | `atomic {`<br>`  x++;`<br>`  y++;`<br>`}` |

Provided *x*=*y* initially, neither transaction above alters this invariant, but it's possible transaction A will read *x* after transaction B updates it but read *y* before transaction B updates it, causing it to enter an infinite loop. The usual strategy for dealing with this is to intercept any fatal exceptions and abort any transaction that is not valid.

One way to deal with these issues is to detect transactions that execute illegal operations or fail to terminate and abort them cleanly; another approach is the transactional locking scheme.

## Implementations

A number of STM implementations (on varying scales of quality and stability) have been released, many under liberal licenses. These include:

### C/C++

- TinySTM [6] a time-based STM and Tanger [7] to integrate STMs with C and C++ via LLVM.
- The Lightweight Transaction Library [8] (LibLTX), a C implementation by Robert Ennals focusing on efficiency and based on his papers "Software Transactional Memory Should Not Be Obstruction-Free" and "Cache Sensitive Software Transactional Memory".
- LibCMT [9], an open-source implementation in C by Duilio Protti based on "Composable Memory Transactions". The implementation also includes a C# binding [10].
- TARIFA [11] is a prototype that brings the "atomic" keyword to C/C++ by instrumenting the assembler output of the compiler.
- Intel STM Compiler Prototype Edition [12] implements STM for C/C++ directly in a compiler (the Intel Compiler) for Linux or Windows producing 32 or 64 bit code for Intel or AMD processors. Implements atomic keyword as well as providing ways to decorate (declspec) function definitions to control/authorize use in atomic sections. A substantial implementation in a compiler with the stated purpose to enable large scale experimentation in any size C/C++ program. Intel has made four research releases of this special experimental version of its product compiler.

- stmmap [13] An implementation of STM in C, based on shared memory mapping. It is for sharing memory between threads and/or processes (not just between threads within a process) with transactional semantics. The multi-threaded version of its memory allocator is in C++.
- CTL [14] An implementation of STM in C, based on TL2 but with many extensions and optimizations.
- The TL2 lock-based STM from the Scalable Synchronization [15] research group at Sun Microsystems Laboratories, as featured in the DISC 2006 article "Transactional Locking II".
- Several implementations by Tim Harris & Keir Fraser [16], based on ideas from his papers "Language Support for Lightweight Transactions", "Practical Lock Freedom", and an upcoming unpublished work.
- RSTM [17] The University of Rochester STM written by a team of researchers led by Michael L. Scott.
- G++ 4.7 [18] now supports STM for C/C++ directly in the compiler. The feature is still listed as "experimental", but may still provide the necessary functionality for testing.

### C#

- Shielded [19] A strict and mostly obstruction-free STM for .NET, written in C#. Features include: conditional transactions, commutable (low conflict) operations, transactional collection types, and automatic generation of transactional proxy-subclasses for POCO objects.
- STMNet [20] A pure C#, open-source, lightweight software transactional memory API.
- SXM, an implementation of transactions for C# by Microsoft Research. Documentation [21], Download page [22] Discontinued.
- LibCMT [9], an open-source implementation in C by Duilio Protti based on "Composable Memory Transactions". The implementation also includes a C# binding [10].
- NSTM [23], .NET Software Transactional Memory written entirely in C# offering truly nested transactions and even integrating with System.Transactions.
- MikroKosmos [24] A Verification-Oriented Model Implementation of an STM in C#.
- ObjectFabric [25] cf. Java implementations.
- Sasa.TM [26] A pure C# implementation of software transactional memory.

### Clojure

- Clojure has STM support built into the core language

### Common Lisp

- CL-STM [27] A multi-platform STM implementation for Common Lisp.
- STMX [28] An open-source, actively maintained concurrency library providing both software and hardware memory transactions for Common Lisp.

### F#

- F# have them through [29] FSharpX - sample at [30] F#

### Groovy

- GPars [32] - The Gpars [31] framework contains support for STM leveraging the Java Multiverse [32] implementation.

### Haskell

- The STM [33] library, as featured in "Composable Memory Transactions" [34], is part of the Haskell Platform.
- The DSTM [35] library, a distributed STM, based on the above library.

### Java

- SCAT research group [36]'s implementation of AtomJava.
- JVSTM [37] implements the concept of Versioned Boxes [38] proposed by João Cachopo and António Rito Silva, members of the Software Engineering Group - INESC-ID [39]. Beginning from version 2.0, JVSTM is completely lock-free.
- Deuce [40] A runtime environment for Java Software Transactional Memory using byte code manipulation.
- Multiverse [32] Is a Java 1.6+ based Software Transactional Memory (STM) implementation that uses Multi Version Concurrency Control (MVCC) as concurrency control mechanism.
- DSTM2 [41] Sun Lab's Dynamic Software Transactional Memory Library
- ObjectFabric [25] is an open source implementation for Java and .NET. It can be turned into a Distributed STM through an extension mechanism. Other extensions allow logging, change notification, and persistence.
- ScalaSTM [42] - A library-based STM written in Scala that additionally provides a Java-focused API to allow use with Runnable and Callable objects.

### JavaScript

- AtomizeJS [43] implements Distributed Software Transactional Memory to web browsers with a single NodeJS server to validate the effects of transactions.

### OCaml

- coThreads [44], a concurrent programming library of OCaml, offers STM (originally STMLib [45]) as a module. Just like any other components in this library, the STM module can be used uniformly with VM-level threads, system threads and processes.

### Perl

- STM for Perl 6 has been implemented in Pugs via the Glasgow Haskell Compiler's STM library.

### Python

- Fork of CPython with atomic locks [46] - Armin Rigo explains his patch to CPython in an email to the pypy-dev list [47].
- PyPy STM with Threads [48] announcement from Armin Rigo for PyPy.

### Scala

- scala-stm [49] - An implementation of a software transactional memory framework in Scala
- ScalaSTM [50] - A draft proposal along with reference implementation CCSTM[51] to be included in the Scala standard library
- Akka STM - The Akka [52] framework contains support for STM in both Scala & Java
- MUTS [53] - Manchester University Transactions for Scala.[54]

### Smalltalk

- GemStone/S [55] A Transactional Memory Object Server for Smalltalk.
- STM [56] for the open-source Smalltalk (MIT Licence) Pharo [57]

### Other languages

- Fortress is a language developed by Sun that uses DSTM2 [41]
- STM.NET [58]

## References


[1] Tom Knight. *An architecture for mostly functional languages.* (http://web.mit.edu/mmt/Public/Knight86.pdf) Proceedings of the 1986 ACM conference on LISP and functional programming.

[2] Maurice Herlihy and J. Eliot B. Moss. *Transactional memory: architectural support for lock-free data structures.* Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93). Volume 21, Issue 2, May 1993.

[3] Nir Shavit and Dan Touitou. *Software transactional memory.* Distributed Computing. Volume 10, Number 2. February 1997.

[4] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, Emmett Witchel (2009): "Committing conflicting transactions in an STM" (http://portal.acm.org/citation.cfm?id=1504201) *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (PPoPP '09), ISBN 978-1-60558-397-6

[5] Lingli Zhang, Vinod K.Grover, Michael M. Magruder, David Detlefs, John Joseph Duffy, Goetz Graefe (2006): *Software transaction commit order and conflict management* (http://www.freepatentsonline.com/7711678.html) United States Patent 7711678, Granted 05/04/2010.

[6] http://www.tmware.org/tinystm

[7] http://tinystm.org/tanger

[8] https://sourceforge.net/projects/libltx

[9] http://sourceforge.net/projects/libcmt

[10] http://libcmt.sourceforge.net/index.php?page=CSharp_API

[11] http://www.hackshack.de/tarifa/

[12] http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/

[13] http://github.com/skaphan/stmmap

[14] http://asc.di.fct.unl.pt/~jml/Research/Software

[15] http://research.sun.com/scalable

[16] http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/

[17] http://www.cs.rochester.edu/research/synchronization/rstm/

[18] http://gcc.gnu.org/wiki/TransactionalMemory

[19] http://github.com/jbakic/Shielded

[20] http://github.com/SepiaGroup/STMNet

[21] http://www.cs.brown.edu/~mph/SXM/README.doc

[22] ftp://ftp.research.microsoft.com/downloads/fbe1cf9a-c6ac-4bbb-b5e9-d1fda49ecad9/SXM1.1.zip

[23] http://weblogs.asp.net/ralfw/archive/tags/Software+Transactional+Memory/default.aspx

[24] http://research.microsoft.com/en-us/downloads/c282dbde-01b1-4daa-8856-98876e513462/

[25] http://objectfabric.com

[26] http://higherlogics.blogspot.com/2011/09/software-transactional-memory-in-pure-c.html

[27] http://common-lisp.net/project/cl-stm/

[28] http://cliki.net/stmx

[29] https://github.com/fsharp/fsharpx/blob/master/src/FSharpx.Core/Stm.fs

[30] https://github.com/fsharp/fsharpx/blob/master/samples/StmSample.fsx

[31] http://gpars.codehaus.org/STM

[32] http://multiverse.codehaus.org/overview.html

[33] http://hackage.haskell.org/package/stm

[34] http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/stm.pdf

[35] http://hackage.haskell.org/package/DSTM

[36] http://wasp.cs.washington.edu/wasp_scat.html

[37] http://www.esw.inesc-id.pt/~jcachopo/jvstm/

[38] http://urresearch.rochester.edu/handle/1802/2101

[39] http://www.esw.inesc-id.pt/wikiesw

[40] https://sites.google.com/site/deucestm/

[41] http://www.sun.com/download/products.xml?id=453fb28e

[42] http://nbronson.github.io/scala-stm/index.html

[43] http://atomizejs.github.com
[44] http://cothreads.sourceforge.net
[45] http://www.pps.jussieu.fr/~li/software#stmlib
[46] https://bitbucket.org/arigo/cpython-withatomic
[47] http://mail.python.org/pipermail/pypy-dev/2011-August/008153.html
[48] http://morepypy.blogspot.com/2012/06/stm-with-threads.html
[49] http://github.com/djspiewak/scala-stm/tree/master
[50] http://nbronson.github.com/scala-stm
[51] N.G. Bronson, H. Chafi and K. Olukotun, CCSTM: A library-based STM for Scala (http://ppl.stanford.edu/ccstm/site/scaladays2010bronson.pdf). Proceedings of the 2010 Scala Days Workshop (Lausanne).
[52] http://akka.io/docs/akka/1.2/scala/stm.html
[53] http://apt.cs.man.ac.uk/projects/TERAFLUX/MUTS/
[54] D. Goodman, B. Khan, S. Khan, C. Kirkham, M. Luján and Ian Watson, MUTS: Native Scala Constructs for Software Transactional Memory (http://apt.cs.manchester.ac.uk/people/salkhan/Home_files/scaladays11.pdf). Proceedings of the 2011 Scala Days Workshop (Stanford).
[55] http://www.gemstone.com/products/smalltalk/
[56] http://source.lukas-renggli.ch/transactional.html
[57] http://www.pharo-project.org/
[58] http://msdn.microsoft.com/en-us/devlabs/ee334183.aspx

## External links

- Morry Katz, PARATRAN: A transparent transaction based runtime mechanism for parallel execution of Scheme (http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-454.pdf), MIT LCS, 1989
- Nir Shavit and Dan Touitou. Software Transactional Memory (http://citeseer.ist.psu.edu/shavit95software.html). *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213. August 1995. The paper originating STM.
- Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures (http://dl.acm.org/citation.cfm?id=872048). *Proceedings of the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 92–101. July 2003.
- Tim Harris and Keir Fraser. Language Support for Lightweight Transactions (http://citeseer.ist.psu.edu/harris03language.html). *Object-Oriented Programming, Systems, Languages, and Applications*, pp. 388–402. October 2003.
- Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.3507&rep=rep1&type=pdf).
- Michael L. Scott et al. Lowering the Overhead of Nonblocking Software Transactional Memory (http://www.cs.rochester.edu/u/scott/papers/2006_TRANSACT_RSTM.pdf) gives a good introduction not only to the RSTM but also about existing STM approaches.
- Torvald Riegel and Pascal Felber and Christof Fetzer, A Lazy Snapshot Algorithm with Eager Validation (http://se.inf.tu-dresden.de/pubs/papers/riegel2006lsa.pdf) introduces the first time-based STM.
- Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II (http://home.comcast.net/~pjbishop/Dave/GVTL-TL2-Disc06-060711-Camera.pdf).
- Knight, TF, An architecture for mostly functional languages (http://portal.acm.org/citation.cfm?id=319854&coll=portal&dl=ACM), ACM Lisp and Functional Programming Conference, August, 1986.
- Knight, TF, System and method for parallel processing with mostly functional languages, US Patent 4,825,360, April, 1989.
- Ali-Reza Adl-Tabatabai, Christos Kozyrakis, Bratin Saha, Unlocking concurrency (http://portal.acm.org/citation.cfm?id=1189288), ACM Queue 4, 10 (December 2006), pp 24–33. Ties multicore processors and the research/interest in STM together.
- James R Larus, Ravi Rajwar, Transactional Memory (http://www.morganclaypool.com/toc/cac/1/1), Morgan and Claypool Publishers, 2006.

- Cambridge lock-free group (http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/)
- Software transactional memory Description; Derrick Coetzee (http://blogs.msdn.com/devdev/archive/2005/10/20/483247.aspx)
- Transactional Memory Bibliography (http://www.cs.wisc.edu/trans-memory/biblio/index.html)
- Critical discussion of STM in conjunction with imperative languages (http://enfranchisedmind.com/blog/posts/the-problem-with-stm-your-languages-still-suck/)
- JVSTM – Java Versioned Software Transactional Memory (http://web.ist.utl.pt/~joao.cachopo/jvstm/)
- Deuce - Java Software Transactional Memory (http://sites.google.com/site/deucestm/)
- Blog about STM and the TLII algorithm (http://bartoszmilewski.wordpress.com/2010/09/11/beyond-locks-software-transactional-memory/)
- Flexviews - materialized views act like software transactional memory for database relations (http://flexvie.ws/)

# Go (programming language)

Not to be confused with Go! (programming language), an agent-based language released in 2003.

**Go**

| | |
|---|---|
| **Paradigm(s)** | compiled, concurrent, imperative, structured |
| **Designed by** | Robert Griesemer<br>Rob Pike<br>Ken Thompson |
| **Developer** | Google Inc. |
| **Appeared in** | 2009 |
| **Stable release** | version 1.2 / 1 December 2013 |
| **Typing discipline** | strong, static, inferred, nominal |
| **Major implementations** | gc (8g, 6g, 5g), gccgo |
| **Influenced by** | C, Limbo, Modula, Newsqueak, Oberon, Pascal, Python |
| **Implementation language** | C, Go |
| **OS** | Linux, Mac OS X, FreeBSD, NetBSD, OpenBSD, MS Windows, Plan 9 |
| **License** | BSD-style + Patent grant |
| **Filename extension(s)** | .go |
| **Website** | golang.org [1] |

**Go**, also called **golang**, is a programming language initially developed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is a statically-typed language with syntax loosely derived from that of C, adding garbage collected memory management, type safety, some dynamic-typing capabilities, additional built-in types such as variable-length arrays and key-value maps, and a large standard library.

The language was announced in November 2009 and is now used in some of Google's production systems. Go's "gc" compiler targets the Linux, Mac OS X, FreeBSD, OpenBSD, Plan 9, and Microsoft Windows operating systems and the i386, amd64, and ARM processor architectures. A second compiler, gccgo, is a GCC frontend.

## History

Ken Thompson states that, initially, Go was purely an experimental project. Referring to himself along with the other original authors of Go, he states:

> When the three of us [Thompson, Rob Pike, and Robert Griesemer] got started, it was pure research. The three of us got together and decided that we hated C++. [laughter] ... [Returning to Go,] we started off with the idea that all three of us had to be talked into every feature in the language, so there was no extraneous garbage put into the language for any reason.

The history of the language before its first release, back to 2007, is covered in the language's FAQ.

## Conventions and language tools

Go has a standard style covering indentation, spacing, and many other details, usually applied to user code by the `go fmt` tool in the Go distribution. Go *requires* that programs not contain unused variables or imports, omit `return` statements, or discard the results of evaluating certain built-in functions.[2] Banning unnecessary imports is particularly important because Go does not support circular dependencies.

Go also comes with `godoc`, a tool that generates text or HTML documentation from comments in source code, `go vet`, which analyzes code searching for common stylistic problems and mistakes. A profiler, `gdb` debugging support, and a race condition tester are also in the distribution.

As with many languages, there is an ecosystem of tools that add to the standard distribution, such as `gocode`, which enables code autocompletion in many text editors, `goimports` (by a Go team member), which automatically adds/removes package imports as needed, `errcheck`, which detects code that might unintentionally ignore errors, and more. Plugins exist to add language support in widely used text editors, and at least one IDE, LiteIDE [3], targets Go in particular.

## Concurrency

Go provides facilities for writing concurrent programs that share state by communicating.[4][5][6] Concurrency refers not only to multithreading and CPU parallelism, which Go supports, but also to asynchrony: letting slow operations like a database or network read run while the program does other work, as is common in event-based servers.[7]

## Language design

Go is recognizably in the tradition of C, but makes many changes aimed at conciseness, simplicity, and safety. The following is a brief overview of the features which define Go (for more information see the language specification [8]):

- A syntax and environment adopting patterns more common in dynamic languages:
  - Concise variable declaration and initialization through type inference (`x := 0` not `int x = 0;`).
  - Fast compilation times.
  - Remote package management (`go get`)[9] and online package documentation.[10]
- Distinctive approaches to particular problems.
  - Built-in concurrency primitives: light-weight processes (goroutines), channels, and the `select` statement.
  - An interface system in place of virtual inheritance, and type embedding instead of non-virtual inheritance.
  - A toolchain that, by default, produces statically linked native binaries without external dependencies.
- A desire to keep the language specification [8] simple enough to hold in a programmer's head,[11][12] in part by omitting features common to similar languages:
  - no type inheritance
  - no method or operator overloading
  - no circular dependencies among packages
  - no pointer arithmetic
  - no assertions
  - no generic programming

## Syntax

Go's syntax includes changes from C aimed at keeping code concise and readable. The programmer needn't specify the types of expressions, allowing just `i := 3` or `w := "some words"` to replace C's `int i = 3;` or `char* s = "some words";`. Semicolons at the end of lines aren't required. Functions may return multiple, named values, and returning a `result, err` pair is the standard way to handle errors in Go.[13] Go adds literal syntaxes for initializing struct parameters by name, and for initializing maps and slices. As an alternative to C's three-statement `for` loop, Go's `range` expressions allow concise iteration over arrays, slices, strings, and maps.

## Types

Go adds some basic types not present in C for safety and convenience:

- *Slices* (written `[]`*`type`*) point into an array of objects in memory, storing a pointer to the start of the slice, a length, and a *capacity* specifying when new memory needs to be allocated to expand the array. Slice contents are passed by reference, and their contents are always mutable.
- Go's immutable `string` type typically holds UTF-8 text (though it can hold arbitrary bytes as well).
- `map[`*`keytype`*`]`*`valtype`* provides a hashtable.
- Go also adds *channel types*, which support concurrency and are discussed in the Concurrency section, and *interfaces*, which replace virtual inheritance and are discussed in Interface system section.

Structurally, Go's type system has a few differences from C and most C derivatives. Unlike C `typedef`s, Go's named `type`s are not aliases for each other, and rules limit when different types can be assigned to each other without explicit conversion.[14] Unlike in C, conversions between number types are explicit; to ensure that doesn't create verbose conversion-heavy code, numeric constants in Go represent abstract, untyped numbers.[15] Finally, in place of non-virtual inheritance, Go has a feature called *type embedding* in which one object can contain others and pick up their methods.

## Package system

In Go's package system, each package has a path (e.g., `"compress/bzip2"` or `"code.google.com/p/go.net/html"`) and a name (e.g., `bzip2` or `html`). References to other packages' definitions must *always* be prefixed with the other package's name, and only the *capitalized* names from other modules are accessible: `io.Reader` is public but `bzip2.reader` is not. The `go get` command can retrieve packages stored in a remote repository such as Github or Google Code, and package paths often look like partial URLs for compatibility.[16]

## Omissions

Go deliberately omits certain features common in other languages, including generic programming, assertions, pointer arithmetic, and inheritance. After initially omitting exceptions, the language added the `panic`/`recover` mechanism, but it is only meant for rare circumstances.[17][18]

The Go authors express an openness to generic programming, explicitly argue against assertions and pointer arithmetic, while defending the choice to omit type inheritance as giving a more useful language, encouraging heavy use of interfaces instead.

### Goroutines, channels, and `select`

Go's concurrency-related syntax and types include:

- The `go` statement, `go func()`, starts a function in a new light-weight process, or *goroutine*
- *Channel types*, `chan type`, provide a type-safe, synchronized, optionally buffered channels between goroutines, and are useful mostly with two other facilities:
  - The *send statement*, `ch <- x` sends `x` over `ch`
  - The *receive operator*, `<- ch` receives a value from `ch`
  - Both operations block until the other goroutine is ready to communicate
- The `select` statement uses a `switch`-like syntax to wait for communication on any of a set of channels[19]

From these tools one can build concurrent constructs like worker pools, pipelines (in which, say, a file is decompressed and parsed as it downloads), background calls with timeout, "fan-out" parallel calls to a set of services, and others.[20] Channels have also found uses further from the usual notion of interprocess communication, like serving as a concurrency-safe list of recycled buffers,[21] implementing coroutines (which helped inspire the name *goroutine*),[22] and implementing iterators.[23]

While communicating-processes model is the favored one in Go, it isn't the only one: memory can be shared across goroutines (see below), and the standard `sync` module provides locks and other primitives.[24]

### Safety

There are no restrictions on how goroutines access shared data, making race conditions possible. Specifically, unless a program explicitly synchronizes via channels or mutexes, writes from one goroutine might be partly, entirely, or not at all visible to another, often with no guarantees about ordering of writes. Furthermore, Go's *internal data structures* like interface values, slice headers, and string headers are not immune to race conditions, so type and memory safety can be violated in multithreaded programs that modify shared instances of those types without synchronization.[25][26]

Idiomatic Go minimizes sharing of data (and thus potential race conditions) by communicating over channels, and a race-condition tester is included in the standard distribution to help catch unsafe behavior. Still, it is important to realize that while Go provides *building blocks* that can be used to write correct, comprehensible concurrent code, arbitrary code isn't *guaranteed* to be safe.

Some concurrency-related structural conventions of Go (channels and alternative channel inputs) are derived from Tony Hoare's communicating sequential processes model. Unlike previous concurrent programming languages such as occam or Limbo (a language on which Go co-designer Rob Pike worked[27]), Go does not provide any built-in notion of safe or verifiable concurrency.

## Interface system

In place of virtual inheritance, Go uses *interfaces*. An interface declaration is nothing but a list of required methods: for example, implementing `io.Reader` requires a `Read` method that takes a `[]byte` and returns a count of bytes read and any error.[28] Code calling `Read` needn't know whether it's reading from an HTTP connection, a file, an in-memory buffer, or any other source.

Go's standard library defines interfaces for a number of concepts: input sources [29] and output sinks, [30] sortable collections, [31] objects printable as strings, [32] cryptographic hashes [33], and so on.

Besides calling methods via interfaces, Go allows converting interface values to other types with a run-time type check. The language constructs to do so are the *type assertion*,[34] which checks against a single potential type, and the *type switch*,[35] which checks against multiple types.

Go types don't declare which interfaces they implement: having the required methods *is* implementing the interface. In formal language, Go's interface system provides structural rather than nominal typing.

`interface{}`, the *empty interface*, is an important corner case because it can refer to an item of *any* concrete type, including primitive types like `string`. Code using the empty interface can't simply call methods (or built-in operators) on the referred-to object, but it can store the `interface{}` value, try to convert it to a more useful type via a type assertion or type switch, or inspect it with Go's `reflect` package.[36] Because `interface{}` can refer to any value, it's a limited way to escape the restrictions of static typing, like `void*` in C but with additional run-time type checks.

The example below uses the `io.Reader` and `io.Writer` interfaces to test Go's implementation of SHA-256 on a standard test input, 1,000,000 repeats of the character "a". `RepeatByte` implements an `io.Reader` yielding an infinite stream of repeats of a byte, similar to Unix `/dev/zero`. The `main()` function uses `RepeatByte` to stream a million repeats of "a" into the hash function, then prints the result, which matches the expected value published online.[37] Even though both reader and writer interfaces are needed to make this work, the code needn't mention either; the compiler infers what types implement what interfaces:

```
package main

import (
    "fmt"
    "io"
    "crypto/sha256"
)

type RepeatByte byte

func (r RepeatByte) Read(p []byte) (n int, err error) {
    for i := range p {
        p[i] = byte(r)
    }
    return len(p), nil
}

func main() {
    testStream := RepeatByte('a')
    hasher := sha256.New()
    io.CopyN(hasher, testStream, 1000000)
    fmt.Printf("%x", hasher.Sum(nil))
}
```

(Run or edit this example online. [38])

Also `type RepeatByte` is defined as a `byte`, not a `struct`. Named types in Go needn't be `structs`, and any named type can have methods defined, satisfy interfaces, and act, for practical purposes, as objects; the standard library, for example, stores IP addresses in byte slices.[39]

Interface values are stored in memory as a pointer to data and a second pointer to run-time type information. Like other pointers in Go, interface values are `nil` if uninitialized.[40] Unlike in environments like Java's virtual machine, there is no object header; the run-time type information is only attached to interface values. So, the system imposes no per-object memory overhead for objects not accessed via interface, similar to C `structs` or C#

`ValueTypes`.

Go does not have interface inheritance, but one interface type can *embed* another; then the embedding interface requires all of the methods required by the embedded interface.

## Examples

### Hello world

Here is a Hello world program in Go:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

(Run or edit this example online. [41])

### Echo

This imitates the Unix echo command in Go:

```
package main

import (
    "flag"
    "fmt"
    "strings"
)

func main() {
    var omitNewline bool
    flag.BoolVar(&omitNewline, "n", false, "don't print final newline")
    flag.Parse() // Scans the arg list and sets up flags.

    str := strings.Join(flag.Args(), " ")
    if omitNewline {
        fmt.Print(str)
    } else {
        fmt.Println(str)
    }
}
```

# Community, conferences and users of Go

## Community

- Gopher Academy [42], Gopher Academy is a group of developers working to educate and promote the golang community.
- Golangprojects.com [43], Programming jobs and projects where companies are looking for people that know Go

## Conferences

- GopherCon [44] The first Go conference. Denver, Colorado, USA April 24-26 2014
- dotGo [45] European conference. Paris, France October 10 2014

## Notable users

Some notable open-source applications in Go include:

- vitess [46], a sharding MySQL proxy used by YouTube
- Docker, a set of tools for deploying Linux containers
- Packer [47], a tool for packaging virtual machine images for multiple platforms
- goread.io [48], an RSS reader
- Flynn [49], a PaaS powered by Docker
- Juju, a service orchestration tool by Canonical, packagers of Ubuntu Linux
- nsq [50], a message queue by bit.ly
- Doozer [51], a lock service by managed hosting provider Heroku
- Sky [52], a database designed for behavioral data such as clickstreams
- heka [53], a stream-processing tool by Mozilla
- GoConvey [54], "is awesome Go testing". A Go testing tool that works with 'go test'.
- btcd [55], An alternative full node bitcoin implementation.

Go has many open-source libraries, some of which include:

- Go's standard library [56] covers a lot of fundamental functionality:
  - Algorithms: compression, cryptography, sorting, math, indexing, and text and string manipulation.
  - External interfaces: I/O, network clients and servers, parsing and writing common formats, running system calls, and interacting with C code.
  - Development tools: reflection, runtime control, debugging, profiling, unit testing, synchronization, and parsing Go.
- Third-party libraries built on top of it, with more specialized tools:
  - Web toolkits, including the Gorilla Web Toolkit [57], Revel [58], and goweb [59]
  - Database, stream, and caching tools, including groupcache [60] and kv [61] and ql [62]
  - Parsers for common formats, such as HTML [63], JSON [64], and Google Protocol Buffers [65]
  - Protocol implementations, such as ssh [66], SPDY [67], and websocket [68]
  - Database drivers, such as sqlite3 [69], mysql [70], and redis [71]
  - Bindings to C libraries, such as cgzip [72], qml [73], and GTK [74]
  - Specialized tools like biogo [75] for bioinformatics, meeus [76] for astronomy, and gogeos [77] for GIS
- Some sites help index the many free libraries outside the Go distribution:
  - godoc.org [78]
  - GitHub's most starred repositories in Go [79]
  - The Go Wiki's project page [80]

Other notable Go users (generally together with other languages, not exclusively) include:[81][82]

- Google, for many projects, notably including download server dl.google.com[83][84][85]
- CloudFlare, for their delta-coding proxy Railgun [86], their distributed DNS service, as well as tools for cryptography, logging, stream processing, and accessing SPDY sites.[87][88]
- SoundCloud, for "dozens of systems"[89]
- Secret [90], a mobile app enabling users to share anonymously with their friends[91]
- The BBC, in some games and internal projects
- Novartis, for an internal inventory system
- Cloud Foundry, a PaaS with various components implemented in Go
- ngrok [92], a tool for remote access to Web development environments
- Pond5 [93], a stock media marketplace
- Poptip [94], a social analytics company
- Splice [95], a music collaboration service
- Vimeo [96], several components of the video processing infrastructure
- SmartyStreets [97], an address verification provider who rewrote all of their services in Go

## Reception

Go's initial release led to much discussion.

Michele Simionato wrote in an article for artima.com:

> Here I just wanted to point out the design choices about interfaces and inheritance. Such ideas are not new and it is a shame that no popular language has followed such particular route in the design space. I hope Go will become popular; if not, I hope such ideas will finally enter in a popular language, we are already 10 or 20 years too late :-(

Dave Astels at Engine Yard wrote:

> Go is extremely easy to dive into. There are a minimal number of fundamental language concepts and the syntax is clean and designed to be clear and unambiguous. Go is still experimental and still a little rough around the edges.

*Ars Technica* interviewed Rob Pike, one of the authors of Go, and asked why a new language was needed. He replied that:

> It wasn't enough to just add features to existing programming languages, because sometimes you can get more in the long run by taking things away. They wanted to start from scratch and rethink everything. ... [But they did not want] to deviate too much from what developers already knew because they wanted to avoid alienating Go's target audience.

Go was named Programming Language of the Year by the TIOBE Programming Community Index in its first year, 2009, for having a larger 12-month increase in popularity (in only 2 months, after its introduction in November) than any other language that year, and reached 13th place by January 2010, surpassing established languages like Pascal. As of December 2013[98], it ranked 35th in the index. Go is already in commercial use by several large organizations.

Bruce Eckel stated:

> The complexity of C++ (even more complexity has been added in the new C++), and the resulting impact on productivity, is no longer justified. All the hoops that the C++ programmer had to jump through in order to use a C-compatible language make no sense anymore -- they're just a waste of time and effort. Now, Go makes much more sense for the class of problems that C++ was originally intended to solve.

## Mascot

Go's mascot is a gopher designed by Renée French, who also designed Glenda, the Plan 9 Bunny. The logo and mascot are licensed under Creative Commons Attribution 3.0 license.

## Naming dispute

On the day of the general release of the language, Francis McCabe, developer of the Go! programming language (note the exclamation point), requested a name change of Google's language to prevent confusion with his language. The issue was closed by a Google developer on 12 October 2010 with the custom status "Unfortunate" and with the following comment: "there are many computing products and services named Go. In the 11 months since our release, there has been minimal confusion of the two languages."

## Notes

[1] http://golang.org
[2] Expression statements - The Go Programming Language Specification (http://golang.org/ref/spec#Expression_statements)
[3] https://github.com/visualfc/liteide
[4] Share by communicating - Effective Go (http://golang.org/doc/effective_go.html#sharing)
[5] Andrew Gerrand, Share memory by communicating (http://blog.golang.org/share-memory-by-communicating)
[6] Andrew Gerrand, Codewalk: Share memory by communicating (http://golang.org/doc/codewalk/sharemem/)
[7] For more discussion, see Rob Pike, Concurrency is not Parallelism (http://vimeo.com/49718712)
[8] http://golang.org/ref/spec
[9] Download and install packages and dependencies - go - The Go Programming Language (http://golang.org/cmd/go/#hdr-Download_and_install_packages_and_dependencies); see godoc.org (http://godoc.org) for addresses and documentation of some packages
[10] godoc.org (http://godoc.org) and, for the standard library, golang.org/pkg (http://golang.org/pkg)
[11] Rob Pike, on The Changelog (http://5by5.tv/changelog/100) podcast
[12] Rob Pike, Less is exponentially more (http://commandcenter.blogspot.de/2012/06/less-is-exponentially-more.html)
[13] The `result, err` pair is analogous to the pair of standard streams `stdout` and `stderr` in Unix interprocess communication, and solves the same semipredicate problem; see Semipredicate problem: Multivalued return.
[14] Assignability - the Go Language Specification (http://golang.org/ref/spec#Assignability)
[15] Constants - the Go Language Specification (http://golang.org/ref/spec#Constants)
[16] Download and install packages and dependencies - go - The Go Programming Language (http://golang.org/cmd/go/#hdr-Download_and_install_packages_and_dependencies)
[17] Panic And Recover (https://code.google.com/p/go-wiki/wiki/PanicAndRecover), Go wiki
[18] Release notes, 30 March 2010 (http://golang.org/doc/devel/weekly.html#2010-03-30)
[19] The Go Programming Language Specification (http://golang.org/ref/spec). This deliberately glosses over some details in the spec: `close`, channel range expressions, the two-argument form of the receive operator, unidirectional channel types, and so on.
[20] Concurrency patterns in Go (http://talks.golang.org/2012/concurrency.slide)
[21] John Graham-Cumming, Recycling Memory Buffers in Go (http://blog.cloudflare.com/recycling-memory-buffers-in-go)
[22] tree.go (http://golang.org/doc/play/tree.go)
[23] Ewen Cheslack-Postava, Iterators in Go (http://ewencp.org/blog/golang-iterators/)
[24] sync - The Go Programming Language (http://golang.org/pkg/sync/)
[25] Russ Cox, Off to the Races (http://research.swtch.com/gorace)
[26] "There is one important caveat: Go is not purely memory safe in the presence of concurrency."
[27] Brian W. Kernighan, A Descent Into Limbo (http://www.vitanuova.com/inferno/papers/descent.html)
[28] Reader - io - The Go Programming Language (http://golang.org/pkg/io/#Reader)
[29] http://golang.org/pkg/io/#Reader
[30] http://golang.org/pkg/io/#Writer
[31] http://golang.org/pkg/sort/#Interface
[32] http://golang.org/pkg/fmt/#Stringer
[33] http://golang.org/pkg/hash/#Hash
[34] Type Assertions - The Go Language Specification (http://golang.org/ref/spec#Type_assertions)
[35] Type switches - The Go Language Specification (http://golang.org/ref/spec#Type_switches)
[36] reflect.ValueOf(i interface{}) (http://golang.org/pkg/reflect/#ValueOf) converts an `interface{}` to a `reflect.Value` that can be further inspected

[37] SHA-256 test vectors (https://www.cosic.esat.kuleuven.be/nessie/testvectors/hash/sha/Sha-2-256.unverified.test-vectors), set 1, vector #8
[38] http://play.golang.org/p/MIaP4AXV_G
[39] src/pkg/net/ip.go (http://golang.org/src/pkg/net/ip.go)
[40] Interface types - The Go Programming Language Specification (http://golang.org/ref/spec#Interface_types)
[41] http://play.golang.org/p/6wn73kqMxi
[42] http://gopheracademy.com/
[43] http://golangprojects.com/
[44] http://www.gophercon.com/
[45] http://www.dotgo.eu/
[46] https://github.com/youtube/vitess
[47] http://www.packer.io/
[48] https://github.com/mjibson/goread/
[49] https://github.com/flynn
[50] https://github.com/bitly/nsq
[51] http://xph.us/2011/04/13/introducing-doozer.html
[52] http://skydb.io/
[53] https://github.com/mozilla-services/heka
[54] https://github.com/smartystreets/goconvey
[55] https://github.com/conformal/btcd
[56] http://golang.org/pkg
[57] http://www.gorillatoolkit.org/
[58] http://revel.github.io/
[59] https://github.com/stretchr/goweb
[60] https://github.com/golang/groupcache
[61] https://github.com/cznic/kv
[62] https://github.com/cznic/ql
[63] https://code.google.com/p/go.net/html
[64] https://github.com/bitly/go-simplejson
[65] https://code.google.com/p/goprotobuf/
[66] https://code.google.com/p/go.crypto/ssh
[67] https://github.com/amahi/spdy
[68] https://code.google.com/p/go.net/websocket
[69] https://code.google.com/p/go-sqlite/
[70] https://github.com/go-sql-driver/mysql
[71] https://github.com/garyburd/redigo/
[72] https://github.com/youtube/vitess/tree/master/go/cgzip
[73] https://github.com/niemeyer/qml
[74] https://github.com/mattn/go-gtk
[75] https://code.google.com/p/biogo/
[76] https://github.com/soniakeys/meeus
[77] http://paulsmith.github.io/gogeos/
[78] http://godoc.org/
[79] https://github.com/search?l=Go&o=desc&q=stars%3A%3E50&ref=searchresults&s=stars&type=Repositories
[80] https://code.google.com/p/go-wiki/wiki/Projects
[81] Erik Unger, The Case For Go (https://gist.github.com/ungerik/3731476)
[82] Andrew Gerrand, Four years of Go (http://blog.golang.org/4years), The Go Blog
[83] dl.gogle.com: Powered by Go (http://talks.golang.org/2013/oscon-dl.slide)
[84] Matt Welsh, Rewriting a Large Production System in Go (http://matt-welsh.blogspot.com/2013/08/rewriting-large-production-system-in-go.html)
[85] David Symonds, High Performance Apps on Google App Engine (http://talks.golang.org/2013/highperf.slide)
[86] https://www.cloudflare.com/railgun
[87] John Graham-Cumming, Go at CloudFlare (http://blog.cloudflare.com/go-at-cloudflare)
[88] John Graham-Cumming, What we've been doing with Go (http://blog.cloudflare.com/what-weve-been-doing-with-go)
[89] Peter Bourgon, Go at SoundCloud (http://backstage.soundcloud.com/2012/07/go-at-soundcloud/)
[90] https://www.secret.ly/
[91] David Byttow, Demystifying Secret (https://medium.com/secret-den/12ab82fda29f), Medium, 14 February 2014. Retrieved 24 February 2014.
[92] https://ngrok.com/

[93] http://pond5.com/
[94] http://poptip.com/
[95] http://splice.com/
[96] http://vimeo.com
[97] http://smartystreets.com/
[98] http://en.wikipedia.org/w/index.php?title=Go_(programming_language)&action=edit

## References



## External links

- Official website (http://golang.org)
- A Tour of Go (http://tour.golang.org/) (official)
- Go Programming Language Resources (http://go-lang.cat-v.org/) (unofficial)
- Pike, Rob (28 April 2010). "Another Go at Language Design" (http://www.stanford.edu/class/ee380/Abstracts/100428.html). *Stanford EE Computer Systems Colloquium*. Stanford University. ( video (http://ee380.stanford.edu/cgi-bin/videologger.php?target=100428-ee380-300.asx)) — A university lecture

# Automatic parallelization

**Automatic parallelization**, also **auto parallelization**, **autoparallelization**, or **parallelization**, the last one of which implies automation when used in context, refers to converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine. The goal of automatic parallelization is to relieve programmers from the tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation.

The programming control structures on which autoparallelization places the most focus are loops, because, in general, most of the execution time of a program takes place inside some form of loop. There are two main approaches to parallelization of loops: pipelined multi-threading and cyclic multi-threading.[1]

For example, consider a loop that on each iteration applies a hundred operations, runs for a thousand iterations. This can be thought of as a grid of 100 columns by 1000 rows, a total of 100,000 operations. Cyclic multi-threading assigns each row to a different thread. Pipelined multi-threading assigns each column to a different thread.

## Cyclic multi-threading

A cyclic multi-threading parallelizing compiler tries to split up a loop so that each iteration can be executed on a separate processor concurrently.

### Compiler parallelization analysis

The *compiler* usually conducts two passes of analysis before actual parallelization in order to determine the following:

- Is it safe to parallelize the loop? Answering this question needs accurate dependence analysis and alias analysis
- Is it worthwhile to parallelize it? This answer requires a reliable estimation (modeling) of the program workload and the capacity of the parallel system.

The first pass of the compiler performs a data dependence analysis of the loop to determine whether each iteration of the loop can be executed independently of the others. Data dependence can sometimes be dealt with, but it may incur additional overhead in the form of message passing, synchronization of shared memory, or some other method of processor communication.

The second pass attempts to justify the parallelization effort by comparing the theoretical execution time of the code after parallelization to the code's sequential execution time. Somewhat counterintuitively, code does not always benefit from parallel execution. The extra overhead that can be associated with using multiple processors can eat into the potential speedup of parallelized code.

### Example

A loop is called DOALL if all of its iterations, in any given invocation, can be executed concurrently. The Fortran code below is DOALL, and can be auto-parallelized by a compiler because each iteration is independent of the others, and the final result of array `z` will be correct regardless of the execution order of the other iterations.

```
do i = 1, n
  z(i) = x(i) + y(i)
enddo
```

There are many pleasingly parallel problems that have such DOALL loops. For example, when rendering a ray-traced movie, each frame of the movie can be independently rendered, and each pixel of a single frame may be independently rendered.

On the other hand, the following code cannot be auto-parallelized, because the value of `z(i)` depends on the result of the previous iteration, `z(i - 1)`.

```
do i = 2, n
  z(i) = z(i - 1)*2
enddo
```

This does not mean that the code cannot be parallelized. Indeed, it is equivalent to

```
do i = 2, n
  z(i) = z(1)*2**(i - 1)
enddo
```

However, current parallelizing compilers are not usually capable of bringing out these parallelisms automatically, and it is questionable whether this code would benefit from parallelization in the first place.

## Pipelined multi-threading

Main article: software pipelining

A pipelined multi-threading parallelizing compiler tries to break up the sequence of operations inside a loop into a series of code blocks, such that each code block can be executed on separate processors concurrently.

There are many pleasingly parallel problems that have such relatively independent code blocks, in particular systems using pipes and filters. For example, when producing live broadcast television, the following tasks must be performed many times a second:

1. Read a frame of raw pixel data from the image sensor,
2. Do MPEG motion compensation on the raw data,
3. Entropy compress the motion vectors and other data,
4. Break up the compressed data into packets,
5. Add the appropriate error correction and do a FFT to convert the data packets into COFDM signals, and
6. Send the COFDM signals out the TV antenna.

A pipelined multi-threading parallelizing compiler could assign each of these 6 operations to a different processor, perhaps arranged in a systolic array, inserting the appropriate code to forward the output of one processor to the next processor.

## Difficulties

Automatic parallelization by compilers or tools is very difficult due to the following reasons:

- dependence analysis is hard for code that uses indirect addressing, pointers, recursion, or indirect function calls;
- loops have an unknown number of iterations;
- accesses to global resources are difficult to coordinate in terms of memory allocation, I/O, and shared variables;
- *irregular algorithms* that use input-dependent indirection interfere with compile-time analysis and optimization.

## Workaround

Due to the inherent difficulties in full automatic parallelization, several easier approaches exist to get a parallel program in higher quality. They are:

- Allow programmers to add "hints" to their programs to guide compiler parallelization, such as HPF for distributed memory systems and OpenMP or OpenHMPP for shared memory systems.
- Build an interactive system between programmers and parallelizing tools/compilers. Notable examples are Vector Fabrics' Pareon, SUIF Explorer (The Stanford University Intermediate Format compiler), the Polaris compiler, and ParaWise (formally CAPTools).
- Hardware-supported speculative multithreading.

## Historical parallelizing compilers

See also: Automatic parallelization tool

Most research compilers for automatic parallelization consider Fortran programs,Wikipedia:Citation needed because Fortran makes stronger guarantees about aliasing than languages such as C. Typical examples are:

- Paradigm compiler
- Polaris compiler
- Rice Fortran D compiler
- SUIF compiler
- Vienna Fortran compiler

## References

[1] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu-Yeon Wei, David Brooks. "The HELIX Project: Overview and Directions" (http://helix.eecs.harvard.edu/index.php/DAC2012). 2012.

# SystemC

**SystemC** is a set of C++ classes and macros which provide an event-driven simulation interface in C++ (see also discrete event simulation). These facilities enable a designer to *simulate* concurrent processes, each described using plain C++ syntax. SystemC processes can communicate in a *simulated* real-time environment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as user defined. In certain respects, SystemC deliberately mimics the hardware description languages VHDL and Verilog, but is more aptly described as a *system-level modeling language*.

SystemC is applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. SystemC is often associated with electronic system-level (ESL) design, and with transaction-level modeling (TLM).

SystemC is defined and promoted by the Open SystemC Initiative (OSCI), and has been approved by the IEEE Standards Association as IEEE 1666-2005[1] - the SystemC Language Reference Manual (LRM). The LRM provides the definitive statement of the semantics of SystemC. OSCI also provide an open-source proof-of-concept simulator (sometimes incorrectly referred to as the reference simulator), which can be downloaded from the OSCI website.[2] Although it was the intent of OSCI that commercial vendors and academia could create original software compliant to IEEE 1666, in practice most SystemC implementations have been at least partly based on the OSCI proof-of-concept simulator.

SystemC has semantic similarities to VHDL and Verilog, but may be said to have a syntactical overhead compared to these when used as a hardware description language. On the other hand, it offers a greater range of expression, similar to object-oriented design partitioning and template classes. Although strictly a C++ class library, SystemC is sometimes viewed as being a language in its own right. Source code can be compiled with the SystemC library (which includes a simulation kernel) to give an executable. The performance of the OSCI open-source implementation is typically less optimal than commercial VHDL/Verilog simulators when used for register transfer level simulation.

SystemC version 1 included common hardware-description language features such as structural hierarchy and connectivity, clock-cycle accuracy, delta cycles, four-valued logic (0, 1, X, Z), and bus-resolution functions. From version 2 onward, the focus of SystemC has moved to communication abstraction, transaction-level modeling, and virtual-platform modeling. SystemC version 2 added abstract ports, dynamic processes, and timed event notifications.

## History

- 1999-09-27 Open SystemC Initiative announced
- 2000-03-01 SystemC V0.91 released
- 2000-03-28 SystemC V1.0 released
- 2001-02-01 SystemC V2.0 specification and V1.2 Beta source code released
- 2003-06-03 SystemC 2.0.1 LRM (language reference manual) released
- 2005-06-06 SystemC 2.1 LRM and TLM 1.0 transaction-level modeling standard released
- 2005-12-12 IEEE approves the IEEE 1666–2005 standard for SystemC
- 2007-04-13 SystemC v2.2 released
- 2008-06-09 TLM-2.0.0 library released

- 2009-07-27 TLM-2.0 LRM released, accompanied by TLM-2.0.1 library
- 2010-03-08 SystemC AMS extensions 1.0 LRM released
- 2011-11-10 IEEE approves the IEEE 1666–2011 standard for SystemC[3]

SystemC traces its origins to work on Scenic programming language described in a DAC 1997 paper.

ARM Ltd., CoWare, Synopsys and CynApps teamed up to develop SystemC (CynApps later became Forte Design Systems) to launch it first draft version in 1999.[4][5] The chief competitor at the time was SpecC another C++ based open source package developed by UC Irvine personnel and some Japanese companies.

In June 2000, a standards group known as the Open SystemC Initiative was formed to provide an industry neutral organization to host SystemC activities and to allow Synopsys' largest competitors, Cadence and Mentor Graphics, democratic representation in SystemC development.

## Language features

### Modules

Modules are the basic building blocks of a SystemC design hierarchy. A SystemC model usually consists of several modules which communicate via ports. The modules can be thought of as a building block of SystemC.

### Ports

Ports allow communication from inside a module to the outside (usually to other modules) via channels.

### Exports

Exports incorporate channels and allow communication from inside a module to the outside (usually to other modules).

### Processes

Processes are the main computation elements. They are concurrent.

### Channels

Channels are the communication elements of SystemC. They can be either simple wires or complex communication mechanisms like FIFOs or bus channels.

Elementary channels:

- signal: the equivalent of a wire
- buffer
- fifo
- mutex
- semaphore

### Interfaces

Ports use interfaces to communicate with channels.

### Events

Events allow synchronization between processes and must be defined during initialization.

### Data types

SystemC introduces several data types which support the modeling of hardware.

Extended standard types:

- sc_int<*n*> *n*-bit signed integer
- sc_uint<*n*> *n*-bit unsigned integer
- sc_bigint<*n*> *n*-bit signed integer for *n*>64
- sc_biguint<*n*> *n*-bit unsigned integer for *n*>64

Logic types:

- sc_bit 2-valued single bit
- sc_logic 4-valued single bit
- sc_bv<*n*> vector of length *n* of sc_bit
- sc_lv<*n*> vector of length *n* of sc_logic

Fixed point types:

- sc_fixed<> templated signed fixed point
- sc_ufixed<> templated unsigned fixed point
- sc_fix untemplated signed fixed point
- sc_ufix untemplated unsigned fixed point

## Example

Example code of an adder:

```
#include "systemc.h"

SC_MODULE(adder)           // module (class) declaration
{
  sc_in<int> a, b;         // ports
  sc_out<int> sum;

  void do_add()            // process
  {
    sum.write(a.read() + b.read()); //or just sum = a + b
  }

  SC_CTOR(adder)           // constructor
  {
    SC_METHOD(do_add);     // register do_add to kernel
    sensitive << a << b;   // sensitivity list of do_add
  }
};
```

## Power/Energy estimation in SystemC

The Power/Energy estimation can be accomplished in SystemC by means of simulations. Powersim[6] is a SystemC class library aimed to the calculation of power and energy consumption of hardware described at system level. To this end, C++ operators are monitored and different energy models can be used for each SystemC data type. Simulations with Powersim do not require any change in the application source code.

## Vendors supporting SystemC

- Aldec
- AutoESL [7]
- Cadence Design Systems
- HCL Technologies [8]: ESL Services
- Calypto
- CircuitSutra [9]: SystemC Modeling Services
- CoFluent Design
- CoSynth Synthesizer [10]
- CoWare
- Forte Design Systems
- Mentor Graphics
- OVPsim provided by Open Virtual Platforms initiative with over 100 embedded processor core models used in SystemC based virtual platforms
- NEC CyberWorkBench [11]
- Imperas embedded software development tools using SystemC based virtual platforms developed by Imperas Software [12]
- Synopsys
- SystemCrafter [13]
- JEDA Technologies [14]
- Catapult C from Calypto supports SystemC, C++ and C as input, and also generates SystemC for verification.
- HIFSuite [15] from EDALab [16] allows SystemC code manipulation like conversion, automatic abstraction, verification and any custom workflow
- Dynalith Systems [17]: SystemC-FPGA co-simulation, which runs hardware block in the FPGA along with SystemC simulator through PCI, PCIe, or USB.
- verilog2systemc [18]: Free Verilog to SystemC Translator from EDA Utils [19].
- VLAB Works [20]
- VisualSim [21]

## Notes

[1] IEEE 1666 Standard SystemC Language Reference Manual (http://standards.ieee.org/getieee/1666/)

[2] www.systemc.org, the Open SystemC Initiative website (http://www.systemc.org)

[3] IEEE Approves Revised IEEE 1666™ “SystemC Language” Standard for Electronic System-Level Design, Adding Support for Transaction-level Modeling -- http://www.businesswire.com/news/home/20111109006054/en/IEEE-Approves-Revised-IEEE-1666%E2%84%A2-%E2%80%9CSystemC-Language%E2%80%9D

[4] Synopsys and Co-Ware Inc., which did much of the work behind the SystemC -- http://www.electronicsweekly.com/Articles/1999/12/07/13906/stm-synopsys-in-3-year-rampd-deal.htm

[5] "ARM is pleased that Synopsys, CoWare and other companies have come together on SystemC, because if it is taken up by the industry, it simplifies our world," said Tudor Brown, chief technology officer of ARM Ltd" in Babel of languages competing for role in SoC - http://www.eetimes.com/ip99/ip99story1.html

[6] http://sourceforge.net/projects/powersim/

[7] http://www.autoesl.com

[8] http://www.hcltech.com
[9] http://www.circuitsutra.com
[10] http://www.cosynth.com/index.php?page=cosynth-synthesizer&hl=en_US
[11] http://www.necst.co.jp/product/cwb/english/
[12] http://www.imperas.com
[13] http://www.systemcrafter.com
[14] http://www.jedatechnologies.net
[15] http://hifsuite.edalab.it
[16] http://www.edalab.it
[17] http://www.dynalith.com/doku.php?id=brochure
[18] http://www.edautils.com/verilog2systemc.html
[19] http://www.edautils.com
[20] http://www.vworks.com/
[21] http://www.visualsim.com/

## References


- *1666-2005 — IEEE Standard System C Language Reference Manual*. 2006. doi: 10.1109/IEEESTD.2006.99475 (http://dx.doi.org/10.1109/IEEESTD.2006.99475). ISBN 0-7381-4871-7.
- *1666-2011 — IEEE Standard for Standard SystemC Language Reference Manual*. 2012. doi: 10.1109/IEEESTD.2012.6134619 (http://dx.doi.org/10.1109/IEEESTD.2012.6134619). ISBN 978-0-7381-6801-2.
- T. Grötker, S. Liao, G. Martin, S. Swan, *System Design with SystemC*. Springer, 2002. ISBN 1-4020-7072-1
- A SystemC based Linux Live CD with C++/SystemC tutorial (http://sclive.wordpress.com/)
- J. Bhasker, *A SystemC Primer*, Second Edition, Star Galaxy Publishing, 2004. ISBN 0-9650391-2-9
- D. C. Black, J. Donovan, *SystemC: From the Ground Up*, 2nd ed., Springer 2009. ISBN 0-387-69957-0
- Frank Ghenassia (Editor), *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, Springer 2006. ISBN 0-387-26232-6
- Stan Y. Liao, Steven W. K. Tjiang, Rajesh K. Gupta: An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment. DAC 1997: 70-75


## External links

- Official website (http://www.systemc.org/home/)
- IEEE 1666 Standard SystemC Language Reference Manual (http://standards.ieee.org/getieee/1666/)
- SystemC Tutorial (http://www.asic-world.com/systemc/tutorial.html)
- ESCUG (http://www-ti.informatik.uni-tuebingen.de/~systemc/) - European SystemC Users Group
- NASCUG (http://www.nascug.org) - North American SystemC User's Group
- LASCUG (http://www.lascug.org) - Latin American SystemC User's Group
- ISCUG (http://www.iscug.in/) - Indian SystemC User's Group
- EDA Playground (http://www.edaplayground.com) - Free web browser-based C++/SystemC IDE

# List of important publications in concurrent, parallel, and distributed computing

Further information: Edsger W. Dijkstra Prize in Distributed Computing

This is a list of **important publications** in concurrent, parallel, and distributed computing, organized by field.

Some reasons why a particular publication might be regarded as important:

- **Topic creator** – A publication that created a new topic
- **Breakthrough** – A publication that changed scientific knowledge significantly
- **Influence** – A publication which has significantly influenced the world or has had a massive impact on the teaching of concurrent, parallel, and distributed computing.

## Consensus, synchronisation, and mutual exclusion

Synchronising concurrent processes. Achieving consensus in a distributed system in the presence of faulty nodes, or in a wait-free manner. Mutual exclusion in concurrent systems.

**Dijkstra: "Solution of a problem in concurrent programming control"**

Dijkstra, E. W. (1965). "Solution of a problem in concurrent programming control". *Communications of the ACM* **8** (9): 569. doi:10.1145/365559.365617 [1].

This paper presented the first solution to the mutual exclusion problem. Leslie Lamport writes that this work "started the field of concurrent and distributed algorithms".[2]

**Pease, Shostak, Lamport: "Reaching agreement in the presence of faults"**
**Lamport, Shostak, Pease: "The Byzantine generals problem"**

Pease, Marshall; Shostak, Robert; Lamport, Leslie (1980), "Reaching agreement in the presence of faults", *Journal of the ACM* **27** (1): 228–234, doi:10.1145/322186.322188 [3].

Lamport, Leslie; Shostak, Robert; Pease, Marshall (1982), "The Byzantine generals problem", *ACM Transactions on Programming Languages and Systems* **4** (3): 382–401, doi:10.1145/357172.357176 [4].

These two papers introduced and studied the problem that is nowadays known as Byzantine fault tolerance. The 1980 paper presented the classical lower bound that agreement is impossible if at least 1/3 of the nodes are faulty; it received the Edsger W. Dijkstra Prize in Distributed Computing in 2005. The highly cited 1982 paper gave the problem its present name, and also presented algorithms for solving the problem.

**Herlihy, Shavit: "The topological structure of asynchronous computation"**
**Saks, Zaharoglou: "Wait-free *k*-set agreement is impossible …"**

Herlihy, Maurice; Shavit, Nir (1999), "The topological structure of asynchronous computation" [5], *Journal of the ACM* **46** (6): 858–923, doi:10.1145/331524.331529 [6]. Gödel prize lecture [7].

Saks, Michael; Zaharoglou, Fotios (2000), "Wait-free *k*-set agreement is impossible: The topology of public knowledge"", *SIAM Journal on Computing* **29** (5): 1449–1483, doi:10.1137/S0097539796307698 [8].

These two papers study wait-free algorithms for generalisations of the consensus problem, and showed that these problems can be analysed by using topological properties and arguments. Both papers received the Gödel Prize in 2004.

## Foundations of distributed systems

Fundamental concepts such as time and knowledge in distributed systems.

**Halpern, Moses: "Knowledge and common knowledge in a distributed environment"**

Halpern, Joseph; Moses, Yoram (1990), "Knowledge and common knowledge in a distributed environment", *Journal of the ACM* **37** (3): 549–587, doi:10.1145/79147.79161 [9].

This paper formalised the notion of "knowledge" in distributed systems, demonstrated the importance of the concept of "common knowledge" in distributed systems, and also proved that common knowledge cannot be achieved if communication is not guaranteed. The paper received the Gödel Prize in 1997 and the Edsger W. Dijkstra Prize in Distributed Computing in 2009.

## Notes

[1] http://dx.doi.org/10.1145%2F365559.365617
[2] did not receive the PODC Award or Dijkstra Prize but was nevertheless mentioned twice in the descriptions of the winning papers, in 2002 (http://www.podc.org/influential/2002.html) and in 2006 (http://www.podc.org/dijkstra/2006.html).
[3] http://dx.doi.org/10.1145%2F322186.322188
[4] http://dx.doi.org/10.1145%2F357172.357176
[5] http://www.cs.brown.edu/~mph/HerlihyS99/p858-herlihy.pdf
[6] http://dx.doi.org/10.1145%2F331524.331529
[7] http://www.cs.brown.edu/~mph/finland.pdf
[8] http://dx.doi.org/10.1137%2FS0097539796307698
[9] http://dx.doi.org/10.1145%2F79147.79161

## External links

- "Top-ranked Papers in "Distributed and Parallel Computing"" (http://libra.msra.cn/CSDirectory/Paper_category_16.htm), *Microsoft Academic Search*

# Article Sources and Contributors

**Parallel computing** *Source*: http://en.wikipedia.org/w/index.php?oldid=604920288 *Contributors*: *drew, 5 albert square, 6a4fe8aa039615ebd9ddb83d6acf9a1dc1b684f7, ACSE, AManWithNoPlan, APH, Abutorsam007, Adamstevenson, Agirault, Ajaxartar, Alansohn, Aldie, Alfio, AlistairMcMillan, Allan McInnes, Allmightyduck, Almeria.raul, Almkglor, Amux, Amwebb, Anderston, Angela, Antaeus Feldspar, Anthiety, Apantomimehorse, Arbor to SJ, Arcann, Art LaPella, Artoftransformation, Arvindn, Asafshelly, Ashelly, Auryon, Autopilot, AvicAWB, Awaterl, BMF81, Beland, Bichito, BigFatBuddha, Biruitorul, BlueCannon, Bogey97, Boggie, BorgQueen, Bovineone, Br'er Rabbit, Brendanwood, Brighterorange, Brona, Brvman, Butwhatdoiknow, Can't sleep, clown will eat me, Caniago, CarlHewitt, Catgut, Ceilican, Ceiling Cat, Charleswarner, Charliepeck, Chris j wood, Chris.schaeffer, Ckatz, Cmdrjameson, Commander Shepard, CrADHD, Cswierkowski, Cybercobra, Dabcanboulet, Damiansoul, DariuszT, Darklilac, Dave Bass, DavidCary, Decentprady001, Dekisugi, Delightm, Dgies, Dhuss, Diego Moya, Disavian, DocWatson42, Docu, DragonLord, Dsigal, Dwarf Kirlston, Dyl, EchetusXe, Edderso, Egil, Ehheh, Eluchil, Emperorbma, EncMstr, Epbr123, Expatrick, Ezekiel56, Fantasy, Fredrik, Furrykef, Fæ, GDallimore, Gaius Cornelius, Galoubet, Gantlord, Garfieldspence, Gastonhillar, Gdr, GermanX, Giftlite, Gimmetrow, Glenn Maddox, GrEp, Gracefool, GrahamColm, Grahamec, Guarani.py, Guy Harris, Gvaz151, Gwen-chan, Gwizard, HRV, Hadal, Hairy Dude, Handheldpenguin, Hariharan wiki, Hcberkowitz, Henrik, Heron, Hmeleka, Hollylilholly, Howard McCay, Hslayer, Ideogram, IvanM89, J'raxis, J.delanoy, JForget, JRS, Jack007, Jaganath, JamieHanlon, Jason Quinn, Jesse V., Jesse Viviano, Jevansen, Jimmi Hugh, Jmacaulay, John of Reading, Johnnyw, JonHarder, JoshuacUK, Kbdank71, Kellen, Kku, Knotnic, Koffieyahoo, Kumioko (renamed), Kuru, Kuszi, LaggedOnUser, Laser brain, Leiting, LiDaobing, Liao, Lina6329, LinguistAtLarge, Liridon, Loltowne, M Stocker199, Mad031683, Malleus Fatuorum, Mangst, Manuel Anastácio, Marcoacostareyes, Marek69, Martarius, Martijnthie, Martychen, Matt Deres, Mattisse, Maury Markowitz, Maximus Rex, Mblumber, Mdd, Mechonbarsa, Mfc, Michael Devore, Michael Hardy, Michael Suess, Miym, Mlpkr, Mm40, Mobius, Mr Rav, MrOllie, Nacho Insular, Nbarth, Neelix, NickGarvey, Nixdorf, Nixeagle, Nk, Noetica, Nuttycoconut, Ojigiri, Oleg Alexandrov, Omeros, Omicronpersei8, Ottojschlosser, Out4thecount, P99am, ParallelWolverine, Pdecalculus, Persian Poet Gal, Piano non troppo, Piledhigheranddeeper, Pnm, Prathikh, PseudoSudo, Psy guy, Qiq, Qlmatrix, Quad4rax, Quaeler, Quod, Qwertyus, RTC, RU.Siriuz, RainbowOfLight, Rama, Ramu50, Raul654, RaulMetumtam, Raysonho, Razimantv, Rbonvall, Realist2, RedWolf, Ripe, Rjwilmsi, Robert Mijaković, RobertG, Robertgreer, Romainhk, Romanlezner, Ronz, Rpwoodbu, Ruud Koot, Sam Hocevar, Sander, SandyGeorgia, Scapler, Sciurinæ, Siddhant, SimonP, Skizzik, Socreddy, Sofia Koutsouveli, SoledadKabocha, Soumyasch, StaticVision, Statsone, Stephane Simard, Stevag, Stevage, Stickee, StitchProgramming, Stow44, Strait, TKD, Talayco, Tedickey, The Anome, The Thing That Should Not Be, TheAMmollusc, Thedemonhog, ThomHImself, TimBentley, Tlroche, Tomas e, Tony Sidaway, Tony1, Treisijs, Trekphiler, Ugur Basak, Uvainio, Uwsbel, VanishedUserABC, Vi2, Viajero, Vincnet, Walden, Wapcaplet, Waraqa, Wavelength, Widefox, Wikibarista, Wiwaxia, Woohookitty, Ykhwong, Zadneram, Zenioss, ZeroOne, باسم, ගිනිත සඳකාවන, 357 anonymous edits

**Instruction-level parallelism** *Source*: http://en.wikipedia.org/w/index.php?oldid=601956087 *Contributors*: Abdull, Adrianwn, Alansohn, Alfredo ougaowen, AndyGlew, Anrie Nord, Anthony Appleyard, Chris G, Chub, Cybercobra, DavidWBrooks, Davidhorman, Dgies, Dionyziz, Dougher, Dyl, Fullerene, GeorgeBills, Guy Harris, Harold Oxley, Hellisp, HenkeB, Int21h, Intgr, Ipsign, Jackson Peebles, Jamelan, Jj137, JonHarder, Kbdank71, Kevin Baas, Levin, Michael Hardy, Neilc, Nux, PsyMar, Radagast83, Raffamaiden, Raul654, SimonP, Skamille, Suruena, The1physicist, VampWillow, Zundark, 13 anonymous edits

**Task parallelism** *Source*: http://en.wikipedia.org/w/index.php?oldid=580498764 *Contributors*: Abdull, Amr.rs, Brorson, ChrisGualtieri, Christian75, Cjmclark, Cspan64, Cybercobra, Dcoetzee, Dgies, Erget2005, Fram, Khazadum, Paddles, Raul654, StitchProgramming, Tvk, 24 anonymous edits

**Data parallelism** *Source*: http://en.wikipedia.org/w/index.php?oldid=569298327 *Contributors*: Aervanath, Aktsu, Amr.rs, Bgwhite, Brorson, Cic, Cspan64, Davew haverford, Dcoetzee, Dgies, Donkeydonkeydonkeydonkey, Dougher, Hairy Dude, Khazadum, Mandarax, MrMarmite, Raul654, Spakin, StitchProgramming, Wikipelli, Wws, Ylem, 30 anonymous edits

**Uniform memory access** *Source*: http://en.wikipedia.org/w/index.php?oldid=598841581 *Contributors*: Almeida, Cspan64, Dgies, Dhruvbird, DreamRunnerMoshi, Edans.sandes, Expent, Greenrd, Harmil, JonHarder, Keithathaide, Kubanczyk, Mike Capp, Myasuda, PigFlu Oink, Raul654, Rchandra, Simeondahl, Tabletop, ThePowerofX, VanishedUserABC, Vichindocha, Zhenqinli, Ὁ οἶστρος, 31 anonymous edits

**Non-uniform memory access** *Source*: http://en.wikipedia.org/w/index.php?oldid=611303860 *Contributors*: (, 15.253, 18.94, Arch dude, Arjayay, AxelBoldt, BBCWatcher, Becker553, Beland, BenjaminHare, Bigfoot00, Bjacob, CCFreak2K, CanisRufus, Cdills, ChrisGualtieri, Dr unix, Dsimic, Długosz, Electricmuffin11, Emperorbma, Entheta, Etorus, Ewlyahoocom, Ferry24.Milan, Fuhghettaboutit, GL1zdA, Gardar Rurak, Gilliam, Hellisp, Hfastedge, Iain.mcclatchie, Isidore, Jagjag, JailaBailaBoo, Jan Traspe, Jerryobject, Jharrell, JonHarder, Jonesey95, K0zka, Ken g6, KenSharp, KentOlsen, Kglavin, Khazadum, Kinema, Letdorf, Lightmouse, Lkcl, Luís Felipe Braga, MJA, Maury Markowitz, Moop2000, Mulad, NegativeK, Pawlex, Pedant17, PhilKnight, Pion, Pleeplee, Plugwash, Psantimauro, Raanoo, RadioActive, Ramu50, Ravedave, Raysonho, Rchandra, RedWolf, Rilak, Sander van Malssen, SecureDBA, SteinbDJ, Suruena, Tanvir Ahmmed, Thad McCotter, The Anome, Thumperward, Togo, Trustable, Vincnet, Winterst, Xycaleth, Zadneram, Zalnas, 89 anonymous edits

**Crossbar switch** *Source*: http://en.wikipedia.org/w/index.php?oldid=588909136 *Contributors*: After Midnight, Altenmann, AnnaFrance, Anthony717, AvicAWB, BD2412, Bhadani, Bill william compton, Brholden, BriskWiki, CTZMSC3, Chmod007, CommonsDelinker, DavidCary, Doublebackslash, Elkman, EncMstr, Foxxygirltamara, Frap, Gareth Jones, Harryzilber, Hooperbloob, Hugo999, Jh-Furth, Jim.henderson, Jimb11568, Kiore, Kubanczyk, Longhair, MRSC, Mange01, MarkMLl, MatthewVanitas, Mattopia, Mauls, Maury Markowitz, Merovingian, Muhandes, Mysid, N Yo FACE, Nick Number, Nikevich, Ntsimp, Numbo3, OnBeyondZebrax, Perry Bebbington, Peter Ellis, Petri Krohn, Plasticup, RM21, Radiojon, Rapido, Rchandra, Reedy, Reswobslc, Ricky81682, Saltine, Sam Hocevar, SamHathaway, Securiger, Sfan00 IMG, Snigbrook, StuRat, UltraMagnus, Viswaprabha, Welsh, Wik, X570, Yeatesh, 68 anonymous edits

**Mesh networking** *Source*: http://en.wikipedia.org/w/index.php?oldid=610769008 *Contributors*: A. B., A5b, Aaron Rotenberg, AaronKaplan, AbsolutDan, Ale jrb, Alias Flood, Allan McInnes, Aneah, Arjayay, Asdlfkjasdfkj, Athaba, AxelBoldt, Azimsamjani, Bawolff, Belegur, Bender235, BiH, Binitdubey1, Birczanin, Bkraemer1, Blinken, Blinking Spirit, Bobo192, Borgx, Bracton, Burningstarsfall, CLazop, Caiaffa, Casmith 789, Cburnett, Celia.periza, Charles Matthews, Chris j wood, ClaudeVanMouse, Cody Cooper, Coudrinp, Cousinit99, Custoo, Cwolfsheep, CynicalCoder, DaL33T, Dadu, Dainomite, Davidlchandler, Dawnseeker2000, Dicklyon, Discospinster, Dmbasso, Dmunene, Drstuart, Dwmendes, ENeville, Edward, Eliz81, Ellipsis, Err404, Fdacosta, Frank, Geek2003, Gibhenry, Glenn, Haha1221, Heighest depth, Hexene, Hobsonlane, Hyunmimaust, Iamajpeg, Ilija.pecelj, Imroy, Ioanis Nikolaidis, Ionosphera, Itusg15q4user, Ivolve, JaapB, Jake Nelson, Jalo, Jamelan, Jamessungjin.kim, Javier cardona, Jec, Jeffme, JensAlfke, Jflabourdette, Jim.henderson, Jrincayc, Keithonearth, Kevin B12, Kgfleischmann, Kgrr, Koman90, Kostmo, Kozuch, Kri, Ksyrie, Kuji, Kvng, Landon1980, Lanilsson, Lesser Cartographies, Levin, Mardeg, Mario Behling, MariusG, Matt0401, Matt1299, Maulvie, Mckoss, Mdann52, Michael Hardy, Mikebb, Mindmatrix, Misternuvistor, Mobius, Modster, Mpleasant, Mzmadmike, NGC 2736, Nasa-verve, Nelson50, Nikola Smolenski, Oli Filth, Omegatron, Oranjelo100, Pengo, Peter Chastain, Pink.up, Prashanth2209, Quaestor23, RTG, Rabarberski, Radioactive afikomen, Radon210, Ralucam, Ray Van De Walker, Rcassidy, Rdikeman, Rendsworld, RenniePet, Rich Farmbrough, Richdrich, Ringbang, Rjwilmsi, Rl, Robert Brockway, Robert Cassidy, Rogerb67, Rorro, Sabrinaalexis, Sandeepreddyus, Sepiraph, Sir Nicholas de Mimsy-Porpington, Smee56, Soumyasch, Speck-Made, Stw, Teapeat, Tedwardo2, Temujin9, Tercespogiako, Thegreenj, Tide rolls, Tktechs, Treekids, TreveX, Twilight Realm, Unixxx, Utolotu, Vegaswikian, Veinor, Wavehunter, Wavelength, Wikimaniac20, Willking1979, WriterHound, Wrs1864, Wsnplanet, Yellowaj, Yurik, Zac67, Zacharoo12, Zarfmouse, Zeptomoon, 303 anonymous edits

**Hypercube graph** *Source*: http://en.wikipedia.org/w/index.php?oldid=605420108 *Contributors*: ANONYMOUS COWARD0xC0DE, Bender2k14, Bharel, Booyabazooka, David Eppstein, Ekuurh, G.perarnau, Gaius Cornelius, Giftlite, Grafen, Hermel, Herr Satz, InverseHypercube, Itai, JLeander, JoergenB, Kjkjava, Koko90, LokiClock, Magioladitis, Mysid, Rjwilmsi, Robinh, Salix alba, Stdazi, Tetracube, Tktktk, Tomo, Tomruen, Trovatore, Twri, Xnn, Zaslav, 15 anonymous edits

**Multi-core processor** *Source*: http://en.wikipedia.org/w/index.php?oldid=610269852 *Contributors*: 16@r, 28bytes, 7265, A5b, Aaron McDaid, Abce2, Acalamari, AcidPenguin9873, Adavis444, Addps4cat, Airplaneman, Aldaron, AlefZet, Alejandrocaro35, Alkivar, Alpha Quadrant (alt), Altenmann, Ancheta Wis, Andyluciano, Anetode, Aniru919, Arch dude, Archer3, Auric, Aviadbd, Az29, Babylonfive, Bahnpirat, Barfolomio, Barts1a, Bdsatish, Bender235, Berkut, Bevo, Beyond My Ken, Bikeman333, Binksternet, Bkell, Bmmxc damo, Bobo192, Braincricket, Bsadowski1, Bubba73, CBM, CQJ, Caltas, Carlosguitar, CattleGirl, Cdog44, Centrx, CesarB, Chaitanya.lala, Charles Matthews, Charlie6WIND, Chharper1, Chrisforster, Christian CHABRERIE, Cirt, Closedmouth, Cmbay, Coffee2theorems, Coralmizu, CountingPine, Crazycomputers, Cswierkowski, Cynical, Czarkoff, DARTH SIDIOUS 2, DGJM, DWM, DaGizza, Danorux, Danperryy, DarthShrine, DaveRunner, David Eppstein, DavidCary, DavidLeighEllis, Dawnseeker2000, Dbader, Ddxc, Decora, Dennis Brown, DerHexer, DigitalMediaSage, Disavian, Donner60, Donreed, Dragons flight, Drewster1829, Drtechmaster, Dsimic, Dyl, E946, EMG Blue, EagleFan, Edward, Ehoogerhuis, Elemesh, Ellwd, Elockid, Enisbayramoglu, Epolk, Erpert, Ettrig, Evil Prince, Evilgohan2, FSHL, Falcon Kirtaran, Falcon9x5, Felipe1982, Fernando S. Aldado, Fir0002, Fitzhugh, Flewis, Florian Blaschke, Flyer22, Fram, Frap, Fuhghettaboutit, Furrykef, GFauvel, Gal872875, Gamer2325, Gary, Gastonhillar, Gene Nygaard, Geniac, Geoyo, Gg7777, Giftlite, Gilderien, Gimpy530, Gjd001, Glacialfox, Glenn Maddox, Gnumer, Goldenthree, Goodone121, Gordonrox24, Gorgalore, GorillaWarfare, Gracefool, GreenReaper, GuitarFreak, Guy Harris, Guy2007, Gwern, Gwib, Hairy Dude, Haoao, Harizotoh9, Haseo9999, Hcucu, Hectoruk, Helwr, Hemant wikikosh, Henrik, Henriok, Hervegirod, Hibernian, Hoary, Homo sapiens, Hu12, Hubbabridge, HuskyHuskie, Hz.tiang, IOLJeff, ITMADOG, Ian Pitchford, IanOsgood, Imirman, Imperator3733, Iridescent, Ixfd64, J.delanoy, JHunterJ, JIP, JLaTondre, JPH-FM, Jack Boyce, JagSeal, Jagged 85, Jakohn, JamesBWatson, Jarble, JasonTWL, Jasper Deng, Jeff Song, Jeffmeisel, Jerryobject, Jesse V., Jesse Viviano, Jim1138, Joffeloff, Joshxyz, Jsanthara, Jspiegler, Julesd, JzG, Karlhendrikse, Kcordina, Keithathaide, Kellyprice, Ketiltrout, Kiteinthewind, Klower, Koza1983, Kozuch, Kyle wood, L Kensington, LA Songs, Lambiam, Lerdsuwa, Letowskie, Leuko, Liao, LilHelpa, Littleman TAMU, LizardJr8, Lord British, Ludootje, MBbjv, MacintoshWriter, Mahjongg, Marasmusine, Mark hermeling, Markuswiki, Master Thief Garrett, Matt Britt, MattKingston, Mazin07, Mdegive, Meaghan, Michael Anon, Michaelbarreto, Midgrid, Mikael Häggström, Mike4ty4, Minadasa, Mindmatrix, Miym, Mmernex, Monkeyegg, Monkeyman, MrOllie, Msrill, Mwastrod, Myscrnnm, NJA, NagabhushanReddy, Narcisse, Natamas, NawlinWiki, Nczempin, Neelix, Neilrieck, NerdyNSK, NimbusNiner, Nixdorf, Nodulation, Noelhurley, Nolansdad95120, Nono1234, Nurg, Od1n, Op47, OpenSystemsPublishing, Oren0, P3+J3^u!, ParallelWolverine, Parallelized, Peyre, Pgk1, Piano non troppo, Picklecolor2, Pinpoint23, Plasticboob, Pol098, Polarscribe, Powo, Prari, Proofreader77, Pvsuresh, Qiq, Quaeler, Qwertyus, Radical.bison, Ramurf, RaulMetumtam, Rbarreira, Real NC, Red66, Reinthal, Remi0o, Rich Farmbrough, Rilak, Rmashhadi, Rockstone35, Rogergummer, RoninDusette, Ronz, RoyBoy, Rupert baines, SECProto, SMC, SabbaZ, Sapox, Scientus, ScorpSt, Seaphoto, Serketan, Sha-256, Shalewagner, Shalom Yechiel, Shandris, Sherbrooke, Sigmajove, Simetrical, Simonriley, SirGrant, Skizatch, SlightlyMad, Smithfarm, Smkoehl, Snippy the heavily-templated snail, SnowRaptor, Sofia Koutsouveli, Solipsist, Sonic Hog, Soumyasch, Splintax,

Sreven.Nevets, Starkiller88, StaticVision, Steedhorse, Steinj, Stephenb, Stux, Stylemaster, Sulomania, Super48paul, Superchad, Swanner, SwisterTwister, Taroaldo, Taurius, Techedgeezine, TheAMmollusc, TheDoober, Thebestofall007, Thiseye, Threadman, Thucydides411, Thumperward, Tide rolls, Time2zone, Tommy2010, Truthordaretoblockme, Ttrevers, Ukexpat, UkillaJJ, Ulner, Un Piton, Versus22, VetteDude, Vincecate, Vintei, Virtimo, Vjardin, Vossanova, WhartoX, WhiteTimberwolf, Widefox, Widr, Wikicat, WikipedianMarlith, WinampLlama, Winchelsea, Winterst, Woohookitty, Yamamoto Ichiro, Yworo, Zodon, Чръный человек, 911 anonymous edits

**Symmetric multiprocessing** *Source*: http://en.wikipedia.org/w/index.php?oldid=604661808 *Contributors*: 119, Aaron Brenneman, Alainkaa, Alfio, AlistairMcMillan, Amwebb, Anrie Nord, AnthonyQBachler, Aqualize, Arch dude, Artoftransformation, Autopilot, BBCWatcher, BMF81, Bad Byte, Beland, Bjacob, Bjankuloski06en, Brian Geppert, Brianski, Butros, CanisRufus, Cffrost, Chris Roy, Chris the speller, Cncxbox, Ctrlfreak13, DagErlingSmørgrav, Daniel.robert.franklin, DataWraith, Deleting Unnecessary Words, Dgies, Dr unix, Dyl, EagleOne, Easwarno1, Ecapoccia, Edans.sandes, Edward, Eemars, Elizium23, Evilgohan2, Ewlyahoocom, EwokiWiki, Fatfool, Ferry24.Milan, Fred J, Furrykef, GSTQ21C, Gardar Rurak, Garysixpack, Ghettoblaster, Goldenthree, Guy Harris, Harnisch 54667, Harryboyles, Hathawayc, Hcobb, Henriok, Hns, Homerjay, Hymyly, Iam8up, Inarius, Ipsign, JLaTondre, JRWoodwardMSW, JWA74, Jabberwoch, Jabowery, Jayrvox, Jec, Jerryobject, Jesse V., Joffeloff, John Sauter, JonHarder, Jsnx, Jtrob704, Kalpase, Khazadum, Kntkadosh, Krallja, Kubanczyk, LAMurakami, LilHelpa, Magioladitis, Martarius, Matt Britt, Maury Markowitz, Maximus Rex, Mr Bill, Mr Echo, Myasuda, Neilc, NocNokNeo, Npx122sy, Nuno Tavares, Pakaran, ParallelWolverine, Pathoschild, Pavel Vozenilek, Pearle, Plouiche, Poweroid, Qiq, Qviri, R'n'B, Raulizahi, Raysonho, Rcbarnes, Rdash, Realg187, RedWolf, Richard Russell, Ricsi, Rilak, Rocketshiporion, Ronz, Rwwww, SKopp, SabbaZ, Slon02, Smithfarm, Solarix, Soosed, Spearhead, Splash, Stannered, Stephanvaningen, Szeder, The Anome, Thumperward, Tietew, Tothwolf, Twajn, Urhixidur, Vertigo Acid, Victor.gattegno, Vina, Vinograd19, W Nowicki, Wbm1058, Wikicaz, Wwskier, Xcvista, Yggdræsil, Yuyudevil, Zoicon5, Тиверополник, 260 anonymous edits

**Distributed computing** *Source*: http://en.wikipedia.org/w/index.php?oldid=610426051 *Contributors*: 64.104.217.xxx, APH, Adam Conover, Advuser14, Aervanath, Ahmed abbas helmy, Aitias, Ajtouchstone, Akiezun, AlbertCahalan, AlexChurchill, Alfio, AlisonW, Allan McInnes, Amwebb, Andrewpmk, Andyjsmith, Anonymous56789, AreThree, Ariedartin, Atlant, Autopilots, Barneyboo, Beetstra, Bejnar, Beland, Bender235, Bernfarr, Bihzad, Bjankuloski06en, Bobdoe, Boufal, Bovineone, Brent Gulanowski, Brighterorange, BrokenSegue, Buyya, Cadence-, Capricorn42, CaptainMooseInc, CarlHewitt, CatherineMunro, Cdiggins, Chocmah, ChrisGualtieri, Chrischan, Chtfn, ChuckPheatt, Cincybluffa, Cmdrjameson, Codepro, Cognitivecarbon, Conversion script, CosineKitty, Cpiral, CrypticBacon, Curps, Cybercobra, D104, D6, DLJessup, DVdm, Daira Hopwood, Damian Yerrick, Darkness Productions, Darkwind, David Eppstein, David Shay, David Woolley, Dawidl, Dbabbitt, Dbenbenn, Dbroadwell, Decrease789, DennisDaniels, Derek, Diannaa, Dominikiii, Donarreiskoffer, Dori, Doulos Christos, DragonLord, Dreadstar, Drektor2oo3, Dtngo, Dust Filter, EdH, Edcolins, Edward, Eggstasy, El C, Ellisonch, EncMstr, Eng azza, Evice, Evil Monkey, Ewlyahoocom, EwokiWiki, Ezrakilty, FayssalF, Felix.rivas, Flata, Flubeca, Flyer22, Formulax, Frazzydee, Freakofnurture, Fred Bradstadt, Fredrik, Frehley, Gaius Cornelius, Gangesmaster, Garas, Georgewilliamherbert, Geozapf, Ghewgill, Giftlite, Gilderien, Gilliam, Glenn, Glenn L, Gorffy, Gortsack, Goudron, GrabBrain, Greg Lindahl, Greg Ward, Guarani.py, Guy Harris, Gyll, Hadal, Haham hanuka, Hahutch298, Hala54, Hao2lian, Hazzeryoda, Hello71, Herbee, Heron, Hervegirod, Howdoesthiswo, Ichatz, Ideogram, Iflipti, InShaneee, Indiedan, Iosef aetos, Irdepesca572, Ixfd64, JCLately, JFromm, Jacob grace, Jacobko, Janakan86, Jandalhandler, Jarble, Jeff3000, Jeh, Jesse V., Jguard18, Jni, JoanneB, John Nixon, Jomifica, JonH, JordiGH, Joysofpi, Julianhyde, Juliano, Karmachrome, Kasperd, Kbdank71, Kbrose, Khalid hassani, Khiladi 2010, Kinshuk jpr19, Kitisco, Kizor, Kku, Koyaanis Qatsi, Kristof vt, Kurt Jansson, Kuru, Kuszi, LaggedOnUser, LedgendGamer, Lee Carre, Lee.Sailer, Lexor, LightningDragon, Lupine1647, M1ss1ontomars2k4, Ma2369, Magioladitis, Malhelo, Maria C Mosak, Mario.virtu, Mark Zinthefer, Markov12, Marvinfreeman, Matt Crypto, Matthiaspaul, Maxcommejesus, Maximaximax, Mboverload, Mdd, Mdsmedia, MelbourneStar, Mellorf, Metz2000, Michael2, Miguel in Portugal, Mika au, MikeHearn, Miym, Mkbnett, Monkeypooisgood, MrJones, MrOllie, Mrdude, Mtriana, Nanshu, Narendra22, Neilksomething, Nethgirb, Neum, Nickptar, Nigini, Ninly, Nit634, Nixdorf, NonNobisSolum, Nonlinear149, Nuno Tavares, Nurg, Only2sea, Optim, Page Up, Palaeovia, Papipaul, Pascal76, Paullaw, Philip Trueman, PlasmaTime, Pmerson, Powo, PrimeHunter, Proofreader77, Quaeler, Qwertyus, R3m0t, RDBrown, Rajrajmarley, Ramu50, Raul654, RealityDysfunction, Reddi, Redxiv, Riley Huntley, Rjwilmsi, Rob Hooft, Rotem Dan, Ruud Koot, Rwwww, S.K., Sae1962, Sahkuhnder, Scwlong, SebastianHelm, Shafigoldwasser, Shamjithkv, SimonHova, SimonP, SirDuncan, Skabraham, Slashem, Snorre, Softguyus, Sohil it, Some jerk on the Internet, Sorry Go Fish, Spiffy sperry, Spinningspark, Spl, Statsone, Stephan Leeds, Stevag, SuperMidget, Suruena, Szopen, TallMagic, Tanvir Ahmmed, Tedickey, The Thing That Should Not Be, Thv, Tim Watson, Timwi, Toonsuperlove, Trappist the monk, Trey56, TwoOneTwo, Unfactual POV, Unobjectionable, Vague Rant, VanishedUserABC, Vary, Vespristiano, Viriditas, Vladrassvet, Wakebrdkid, Walden, WalterGR, Wapcaplet, Warrior4321, Wavelength, Waxmop, Wayfarer, Whkoh, Widr, Wifione, WikHead, WikiLaurent, Wikiant, Wizardist, Wonderfulsnow, Worldwidegrid, Wsiegmund, Ww, Xe7al, YUL89YYZ, Yunshui, Île flottante, ساجد امجد ساجد, 470 anonymous edits

**Computer cluster** *Source*: http://en.wikipedia.org/w/index.php?oldid=611694196 *Contributors*: 0x6adb015, 195.162.205.xxx, 1exec1, 74s181, Abrech, Adamantios, Adamd1008, Agentbla, Airconswitch, Akadruid, AlanR, Albing, Alex R S, Ali.marjovi, AlistairMcMillan, Altenmann, Amwebb, Andrey4763913, Angryllama11, Ardahal.nitw, Arnvidr, Arny, Arthena, Atlant, Atomota, Bbryce, Bdiscoe, Beetstra, Beginning, Beland, BenFrantzDale, Bender235, Blaxthos, Blinkin1, Bmitov, Bogey97, Borgx, Bovineone, Brian G. Wilson, Burschik, Butros, Buyya, CRGreathouse, CSWarren, Can't sleep, clown will eat me, Carl Caputo, Cellis3, Channelsurfer, Charles Matthews, Chowbok, Chris 73, Chris Chittleborough, Chris Roy, Chuunen Baka, Closedmouth, Cloudruns, Cmr08, Codename Lisa, Cogiati, CogitoErgoSum14, ConcernedVancouverite, Conversion script, Craigy144, CrypticBacon, Cwmccabe, DARTH SIDIOUS 2, DNewhall, Danakil, Daniel7066, DanielLeicht, DavidBailey, DavidCary, Davidkazuhiro, Decltype, DeeperQA, Demiurge1000, Deskana, Dgies, Dhuss, Disavian, Discospinster, Dmsar, Dsimic, Dyl, Dylan Lake, Edivorce, Edward, Edward Z. Yang, Egil, Ekilfeather, EliasTorres, Elkhashab, Enatron, Epan88, Epbr123, ErkinBatu, Etienne.navarro, Frasmacon, Freemyer, Friedo, From-cary, Funkysh, Fyyer, Gaius Cornelius, Gamester17, Gardrek, Gauravd05, Gbeeker, Geni, Georgewilliamherbert, Gerard Czadowski, Giftlite, Giggy, Gihansky, Gilliam, Glenn, Gm246, Goudzovski, Greg Lindahl, GridMeUp, Ground, Guaka, Guoguo12, H.maghsoudy, HJ Mitchell, Haakon, Haseo9999, Hgz168, Hu12, HughesJohn, I already forgot, Ilanrab, Iluvcapra, Indivara, Izzyu, J.delanoy, JCLately, Jack007, Jacob.utc, Jano-r, Japo, Jeeves, JenniferForUnity, Jeremy Visser, Jesse V., Jfmantis, Jgrun300, Jimmi Hugh, JonHarder, Jopsen, Joy, Jth299, Kbdank71, Kcordina, Kestasjk, Ketil, Kevin B12, Kooo, Kross, Kutsy, Kvng, Kyng, LFaraone, Laurentius, Lear's Fool, Ljvillanueva, LonHohberger, Louzada, Lupinoid, Lurker, MJA, MOmarFarooq, MainFrame, Makwy2, Maria C Mosak, Masticate, MatthewWilcox, Matty, Mav, Mdd, Mdebets, Melizg, Mercy11, Metazargo, Mika au, Minghong, Missy Prissy, Mitchanimugen, Miym, Mkellner@penguincomputing.com, Mmazur, Moa3333, MorganCribbs, MrOllie, Mwilensky, Myasuda, Nabber00, Nakakapagpabagabag, Nataliia Matskiv, NawlinWiki, Neilc, NeoChaosX, Ningauble, Nivanov, Nixdorf, Notmuchtotell, Numa, Nuno Tavares, Nurg, Nv8200p, Oleszkie, Optikos, Orange Suede Sofa, Ortolan88, Paladin1979, Pat Berry, Pengo, Phil Boswell, Phr, Pillefj, Pleasantville, Pnm, Polarscribe, Popolon, Prgururaj, Psneog, Quadell, Quaeler, Quarl, Qwertyus, R'n'B, RA0808, RHaworth, Raysonho, Razimantv, RedWolf, Redgolpe, RenamedUser01302013, Rgbatduke, Rich Farmbrough, Rimmon, Rjstott, Rmarsha3, RobHutten, Robert Brockway, Ronz, Roopa prabhu, Rossami, RoyBoy, Rwwww, Samwb123, Schapel, ScotXW, Sdpinpdx, Selket, Shevel2, Shigdon, Sietse Snel, SimonDeDanser, SimonP, Slashyguigui, Smallpond, Socialservice, Software War Horse, Solitude, Sollosonic, Sophis, Squingynaut, Sriharsh1234, StaticGull, Stbalbach, SteinbDJ, Stevenj, Stevertigo, Strait, Sumdeus, Sumitidentity, Sunil Mohan, Sunray, Superm401, Susvolans, Szopen, TCorp, TenOfAllTrades, The Anome, Theanthrope, Thomasgl, Thpierce, Thumperward, Titodutta, Tobi, Tobias Bergemann, Trescott2000, Troels Arvin, Ukexpat, Uwe Girlich, VanishedUserABC, Vegaswikian, Veggies, Vhiruz, Vivewiki, Vny, W Nowicki, Wagino 20100516, Wavelength, Wensong, Whispering, Wickethewok, Wiki alf, WikiLeon, WikiNickEN, Wikipelli, Wimt, Winterheart, Wizardman, Wmahan, Wonderfl, Wysdom, X201, Xyb, Yahia.barie, Yamaha5, Ylee, Zachlipton, Zavvyone, Zigger, 633 anonymous edits

**Massively parallel (computing)** *Source*: http://en.wikipedia.org/w/index.php?oldid=590793180 *Contributors*: Cogiati, Felixgers, Gardrek, Magioladitis, Mbutts, Paulexyn0, Spaceknarf, Tony1, VanishedUserABC, 4 anonymous edits

**Reconfigurable computing** *Source*: http://en.wikipedia.org/w/index.php?oldid=607940746 *Contributors*: @modi, A5b, A876, Aapo Laitinen, Ancheta Wis, Angr, Anujkaliaiitd, Appusom, Arch dude, Archivist, Arpingstone, Ash ok7, AshtonBenson, Asr2010, Audigyz, Becritical, Breadshe, Brianski, Brighterorange, Christopher Thomas, Circeus, Cj211, Colonies Chris, Dan100, DavidCary, Develash, Dicklyon, Dsav, Ebaychatter0, Eequor, Electron9, Febbets, Frappucino, Gaius Cornelius, Gantlord, GeorgeVacek, Ghewgill, Glenn, GoingBatty, Gp-sci, Hardnett, Henrik, Heron, Hfastedge, JRS, JamesBWatson, Jantangring, Jars99, Jobarts, Joel Saks, Jpbowen, Jpp, JustAGal, Kamenraider, Kapeed irs, Karada, Khalid hassani, Knotwork, Ksiop, Kuru, LucianoLavagno, Magioladitis, Maja Gordic, Maury Markowitz, Mboverload, Mbutts, Michael Hardy, Mizaoku, Moonriddengirl, Mrand, Nixdorf, Npcomp, Numbermaniac, Nvgranny, OWV, OccamzRazor, Offname, Orborde, P millet lab1, Partha.Maji, Pvranade, Raanoo, Rafaelfonte, Rainier3, Rainier34, RainierH, RainierHa, Rdschwarz, RetroTechie, Richard Taytor, Robya, Rpmasson, Rsprattling, Rudderpost, Rwwww, SNIyer12, Sam Hocevar, Sbaum technischeinformatik, Schandho, Sckchui, Seidenstud, Soap, Someone42, Sondag, Stevertigo, Su-steve, Swilton, Tackat, Togo, Tosito, Tvancourt, Ufgatorman, Veledan, Vikramtheone, Vsmith, Wernher, Xe7al, Yahya Abdal-Aziz, Yworo, ZeroOne, Zeus, ^demon, Σ, أحمد, 137 anonymous edits

**Field-programmable gate array** *Source*: http://en.wikipedia.org/w/index.php?oldid=609719441 *Contributors*: 2001:db8, 4twenty42o, =Josh.Harris, A5b, AManWithNoPlan, Abdull, Adrianwn, Alex.muller, Alexf, Altenmann, AnOddName, Ancheta Wis, AndyGaskell, Angela, Angr, Ansell, Anujkaliaiitd, Appusom, Arda Xi, Arkrishna, Arny, Arrowsmaster, Asdf39, Ashok rudra, AshtonBenson, Atlant, Aurelien.desombre, Ausinha, AustinLiveOak, B kkn, Bamakhrama, Baudway, Beetstra, Belchman, Bender235, Bigmantonyd, BlaiseFEgan, Bobo192, Boldupdater, Bomac, Bongwarrior, Brian55555, Briancarlton, Brouhaha, Cadre, Can't sleep, clown will eat me, CanisRufus, Cataclysm, Catverine, Cburnett, Chancheelam, Cheese Sandwich, Chendy, Chris hutchings, ChrisGualtieri, ChristianBk, Christopher Thomas, Ciotog, Ckape, Cmdrjameson, Cnbattson, CobbleCC, Colin Marquardt, CommonsDelinker, Compfreak7, Conquerist, Conversion script, Cooperised, Cordell, CorporateM, Crazygideon, Cutullus, Cybercobra, DV8 2XL, DabMachine, Darragjm, Denniss, Devbisme, Dicklyon, Digihoe, Dimmu1313, Don4of4, Drjimbonobo, Dsimic, Dspguy, Dysprosia, Edcolins, Edward, Edwtie, Eeinfo2008, Eklovya Sharma, Electron20, Electron9, Eludom, Emote, EnOreg, Engineer1, Enginsrts, Enon, Erdot, EvanKroske, FPGAtalk, Falcorian, Faradayplank, FatalError, Fbetti9, Femto, Firefly4342, Fleminra, Fpgacenter, Fpganut, Fraggle81, Frappucino, Freshgod, From92714, FuelWagon, Furrykef, Gabrielldiego, Gadfium, Gaius Cornelius, Gantlord, Gauravm, GeorgeVacek, Gf uip, Giftlite, Glenn, Glrx, Goalie Ca, Goudzovski, Grafen, Gsmcolect, Hdante, Headbomb, Henrik, Henriok, Heron, Hga, Hjamleh, Honkhiam, Hooperbloob, Hu12, ICE77, Ignaciomella, Igor Markov, Islam almasri, Itai, Ixfd64, Jacj, Jack liaotianyu, Jamelan, Jan.gray, Jantangring, Jarble, Jauricchio, Javalenok, Jcarroll, Jeff.johnson.au, Jehochman, Jianhui67, Jidan, Jiekeren, Jihanj, Jinxynix, Joel Saks, Joel7687, Joelby, Joelholdsworth, Jon Awbrey, JonathonReinhart, Julesd, Karada, Karl Stroetmann, Karulont, Katana, Kenyon, KerryVeenstra, Khalid hassani, Killiondude, Kleinash, Kompere, Kostisl, Kphowell, Ktkoson, Kyokpae, Lala jee786, Lannm, Leon7, Lissajous, LouScheffer, Lukeno94, Luojingyuan, Ma Baker, Mamidanna, Mandarax, Mark1liver, Mark83, Markpipe, Masgatotkaca, Mats Randgaard, Matt Britt, Maxalot, Mdodge1982, Menotti, Metaeducation, Michael Hardy, MichaelKipper, Michaelwood, Michagal, Microsp, Mike1024, Mikeabundo, Mirror Vax, Misterpib, Miterdale, Mixel, Mohawkjohn, Mortense, Moxfyre, Mrand, My Other Head, MySchizoBuddy, Mysid, Navasj, Neildmartin, Nixdorf, Nkendrick, Noishe, Nyanya711477, Nylex, OSP Editor, Ogai, Olegos, Olmaltr, Omegatron, On5deu, PacificJuls, Pagemillroad, Pak21, Panscient, Parasyte, Paulcsweeney, Pellucidity, Petter.kallstrom, Pgan002, PhilKnight, Phronk, Plasticspork, Privatechef, Pro crast in a tor, Prophile, QTCaptain, Quadell, RTC, RTucker, Radagast83, Radiojon, RainierHa, Rdschwarz, Red Act, RedWolf, Requestion, Reubentg, RevRagnarok, Rich Farmbrough, Rigmahroll, Rknopman, RobertYu, Rocketrod1960, Rodrigob, Romanm, Ronz, Roo72, Rspanton, Rsriprac, Rudderpost, Rufty, Russvdw, S2cinc, SJK, SanGatiche, Sangwine, Satchmo, Sbmeirow, Sceptre, Sean Whitton, Shaddack, Sintaku, SkerHawx,

Skr15081997, Smsarmad, So-logic, Sobreira, Someguy1221, SoquelCreek, Speedevil, Srleffler, Sslemons, Stangaa, Stephan Leeds, Stevewilton, StevoX, Svick, Symon, Tbc, Techeditor1, Telecineguy, The Anome, TheAMmollusc, Thecheesykid, Thequbit, Thisara.d.m, Thluckoftheirish, Thumperward, Timwi, Togo, Trefoil44, Trioflex, Tyblu, Uchipit, Ultramandk, Unapiedra, VSteiger, Vanlegg, Vasywriter, VictorAnyakin, Villarinho, Vocaro, Welsh, Werieth, Whaa?, WhiteDragon, Wikihitech, Williamv1138, Wjl, Wk muriithi, Wmwmurray, Wolfmankurd, Woutput, Xilinx2425, Xjordanx, Xonus, Xordidi, Yahya Abdal-Aziz, Ygtai, Yworo, Ziegfeldf, Лев Дубовой, ماني, 721 anonymous edits

**General-purpose computing on graphics processing units** *Source*: http://en.wikipedia.org/w/index.php?oldid=609950043 *Contributors*: 1exec1, Adam majewski, Aeusoes1, Ahoerstemeier, AlexHajnal, Andreas Kaufmann, Andy.m.jost, Argaen, Auric, Autodmc, Avihu, BD2412, BalthaZar, BenJWoodcroft, Biasoli, Bjsmithdotus, BlueNovember, Bluemoose, Bovineone, Bsilverthorn, Burpen, Calaka, CanisRufus, Celtechm, Cffrost, Cswierkowski, Curph, Cwolfsheep, Dancter, Dannyniu, Dark Shikari, Darkwind, David Eppstein, DellTechWebGuy, Disavian, Doctorevil64, Dogcow, Dr satnam singh, Dvijaydev46, Echeslack, Efa, Em27, Erencexor, ErikNilsson, Evert Mouw, Ewlyahoocom, FERcsI, Favonian, Fchinchilla, Fernvale, Fiskbil, Frap, Fsantos222, Funandtrvl, Furrykef, GVnayR, Gaius Cornelius, Gamester17, Gargnano, Gioto, GlasGhost, Gpucomputingguru, Gracefool, Grendelkhan, Hans Lundmark, Harryboyles, Headbomb, Hpcanswers, Hulagutten, J04n, Jack Greenmaven, Jak86, Jamesmcmahon0, Jarble, Jerryobject, Jesse V., Jesse Viviano, Jfmantis, Josh Parris, Jowens, Joy, Justin W Smith, Kaihsu, Kbdank71, Kendo70133, Keonnikpour, Koarl0815, Kostmo, Kozuch, Kri, Kudret abi, Lasertomas, Lorem Ip, MER-C, Maghnus, Magioladitis, MaxDZ8, MaxEnt, Mc6809e, Melonakos, Menesergul, Mild Bill Hiccup, Modest Genius, Monkeypox37, Mortense, Mr.Z-man, MrOllie, Mschatz, NYCDA, Nameless23, Neolandes, Netspider, Noegenesis, Now3d, Opencl, Osndok, P99am, Paddles, Piotrus, Plouiche, Potatoswatter, QuasarTE, Qwertyus, R'n'B, Rangoon11, Rarkenin, Raysonho, Research 2010, Rjwilmsi, RockMFR, Ronaldo Aves, Rprabhu13, Sahrin, Sambcool, ScotXW, SeanAhern, Selket, Serg3d2, Sfingram, Shannernanner, SiegeLord, Sigmundur, Simoneau, Skierpage, SkyWalker, Soulhack, SpK, Svetlin, Tanadeau, The GP-you Group, Theblur, Tigeron, Trappist the monk, Tuankiet65, Vanished user ty12kl89jq10, Vleo64, Vyvyan Ade Basterd, WRK, Wasell, Wernher, WikiUserPedia, WipEout!, Wmahan, Your Lord and Master, Yrithinnd, Zarex, Ὁ οἶστρος, 242 anonymous edits

**Application-specific integrated circuit** *Source*: http://en.wikipedia.org/w/index.php?oldid=609483546 *Contributors*: A5b, Adrianwn, Adrory, Aleenf1, Altenmann, Antoinebercovici, Anubhab91, Arpa, Atlant, Ausinha, Avoidhours23423, Aweinstein, Banglore orifice, Barrylb, Behnam Ghiaseddin, BenTremblay, Berserkerus, Bigdumbdinosaur, Blenda, Bradams, Buster79, Bvankuik, CanisRufus, Cannolis, Carbuncle, Carlossuarez46, Chris hutchings, Chris the speller, Cody3019, Coleman89, Cspan64, DRAGON BOOSTER, Damian Yerrick, Danhash, David Eppstein, Deflective, Dgrant, Dicklyon, Djlipkin23, Edward, Elikcohen, Enochlau, Eras-mus, Everyking, Ewlyahoocom, Fattires1995, Favonian, Femto, Floridamicro, Fluffernutter, Frostholm, Ft1, FuelWagon, Fuzzie, GB fan, GRAHAMUK, Garamond Lethe, Gerry Ashton, Giftlite, Gigoptixcxsa, Glenn, Gporter1974, Gurch, HenkeB, Henriok, Hu12, ICE77, Ian Pitchford, Imphil, Imran, Indon, Irondesk, Itai, J04n, Jasonsophia, Jc3s5h, Jcoffland, Jeff G., Jehochman, Jmmooney, Jon Awbrey, JonHarder, Jonathan de Boyne Pollard, Jpbowen, Kanwar rajan, Katiemurphy1, Khalid hassani, Kkmurray, Korg, Kulp, Lcapcara, Le pro du 94 :), Liftarn, Lorkki, LouScheffer, Lzur, MARIODOESBREAKFAST, Madison Alex, Maelwys, Mamidanna, Marcika, Martarius, Maxalot, Mcleodm, Mdamrau, Melvin Simon, Michael Hardy, Missy Prissy, Mogigoma, Moonman333, Mortense, MrOllie, Mtmcdaid, Murugango, Nafis ru, NavarroJ, Ndokos, Nixdorf, OSP Editor, Ohnoitsjamie, Oleg Alexandrov, Overmaster net, Palica, Panscient, Parikshit Narkhede, Patrickyip, PaulN, Pdfpdf, Pdq, Pedant17, PeterBrooks, PeterJohnBishop, Phoebe, Phoenix-forgotten, Plasticup, Pradeep.esg, Pranav tailor, R'n'B, Rahulkants, Rajesh4091989, Randyest, Requestion, Rhebus, Rich Farmbrough, Richard mount, Rilak, Robofish, Rocksheavy2234, Rosenbluh, Rwender, Saltine, Sanjay.iitd2004, Scientus, SelkirkRanch, Shaddack, SteinbDJ, SteveLetwin, StuGeiger, Sudhirsathiyan, Synchronism, Tagremover, Targaryen, Tauheed ashraf, The Anome, Thumperward, Triddle, Trjumpet, Tuankiet65, Victor Snesarev, VivaLaPandaz, VoxLuna, WikHead, Wikipelli, Williamv1138, Wizontheweb, WriterHound, Wsuverkropp, Wwmbes, XP1, Xasodfuih, 305 anonymous edits

**Vector processor** *Source*: http://en.wikipedia.org/w/index.php?oldid=600640043 *Contributors*: Ahoerstemeier, Al Fuentes, Alistair1978, Altonbr, Alvin-cs, Andrewhayes, Anrie Nord, Arch dude, Auric, Bobo192, Bongwarrior, Boud, C777, CanisRufus, Carnildo, Collabi, Coolcaesar, CortalUX, Corti, Crdillon, DMahalko, DavidCary, DavidWBrooks, Dekimasu, Dex1337, Dgies, Diwas, Dmyersturnbull, Drphilharmonic, Dweaver, Eeekster, Eyreland, Flyer22, Frozenport, Gareth Griffith-Jones, GeoGreg, Gzornenplatz, Hajatvrc, Harald Hansen, Hdante, Hellisp, Henriok, Hgrobe, Jarble, Jin, JonHarder, Jyril, Kenyon, Ketiltrout, Kevmeister68, Lambiam, Liao, MatthewWilcox, Maury Markowitz, Memotype, Mipadi, Moolsan, Mu301, Mwtoews, NEMT, Neilc, Niemeyerstein en, Nikai, NotAnonymous0, Ocarroll, Oicumayberight, Pak21, Phe, Phrozenfire, Pishogue, Ramu50, Ravik, RedWolf, Rich257, RichiH, Rilak, Robert Brockway, RucasHost, Saigesploki, Sam Hocevar, Sanxiyn, Shaddam, Shirik, Sofia Koutsouveli, Spbooth, Sunsprite, Sven Manguard, Tagremover, Takanoha, Taxman, Tedder, Terryn3, The Anome, UncleTod, Vanessadannenberg, VanishedUserABC, Vasil', Victoror, Wbm1058, WhiteHatLurker, Winterspan, Wtshymanski, Xpclient, Yaronf, ZFU738, Zhenqinli, 110 anonymous edits

**List of concurrent and parallel programming languages** *Source*: http://en.wikipedia.org/w/index.php?oldid=605476075 *Contributors*: Anniemeyer, Bearcat, Bunnyhop11, CD-Host, Cswierkowski, Dekart, Diku454, Elldekaa, Ivo Balbaert, JamieHanlon, Jhellerstein, Lawrencewaugh, Necklace, NelsonRushton, Niceguyedc, R'n'B, Raul654, Rchrd, ReiniUrban, StitchProgramming, Sttaft, Therealjoefordham, VanishedUserABC, Welsh, Yworo, 53 anonymous edits

**Actor model** *Source*: http://en.wikipedia.org/w/index.php?oldid=608832930 *Contributors*: Aembleton, AgadaUrbanit, Alan.poindexter, Allan McInnes, AmigoNico, Ancechu, Angus Lepper, Anon126, Anonymouser, ArthurDenture, Arthuredelstein, Attys, Ayswaryacv, BD2412, BMF81, Beef, Ben Standeven, Bobo192, Bombnumber20, CBM, CSTAR, CanadianLinuxUser, CarlHewitt, Cfeet77, Cmdrjameson, Cosmotron, Cxbrx, DNewhall, Dafnapu, Daira Hopwood, David Humphreys, DerHexer, Derek R Bullamore, Dibblego, Disavian, Dleonard, Donhalcon, Dougher, DuncanCragg, Edward, ElHef, Ems57fcva, Ermey, FatalError, Finlay McWalter, FrankSanMiguel, Fresheneesz, Frietjes, Gauge, Gibbja, Gracefool, Guy Harris, Hairy Dude, Hillman, Hinrik, Hoodow, Indil, Jack Waugh, Janm67, Jantangring, Jcttrll, Jkeene, Jodal, Jonasboner, Joris.deguet, Josh Parris, Joswig, Jpbowen, Juanco, Junkblocker, Karol Langner, Kku, Knotwork, Koffieyahoo, Kpalsson, Kthejoker, Kusma, La goutte de pluie, Laforge49, Linas, Llywrch, Lradrama, Lsbardel, Lunalot, Madair, Madmediamaven, Magicmonster, Martijn Hoekstra, Marudubshinki, Mathrick, Mboverload, Mdann52, Mdd, Meatsgains, Michael Hardy, Mild Bill Hiccup, Mimi.vx, Mipadi, Mirosamek, Modify, Necromantiarian, Ojw, Okeuday, Older and ... well older, Oli Filth, Oluies, Orthologist, Pauncho.dog, Peterdjones, Pgan002, Piet Delport, PoweredByLolcats, Prof. Hewitt, Ptrelford, Pxma, Qwertyus, Rabarberski, Ravn, Rijkbenik, Rivimey, Rjwilmsi, Rkumar8, RobinMessage, Roggan76, Rohan Jayasekera, Rosiestep, Rp, Ruud Koot, Ryan Roos, Sam Pointon, Sam Staton, Samsara, SarekOfVulcan, Shanewholloway, Sietse Snel, SimonP, Siskus, Slicky, SpuriousQ, Sstrader, StAnselm, SteForster, Stevedekorte, Stuart Morrow, TAnthony, TRBP, Tajmiester, Tbhotch, Terryn3, Texture, TimBentley, Tobias Bergemann, Tony Sidaway, Tonyrex, Torc2, Trevyn, Untalker, Vaclav.pech, VanishedUserABC, Vonkje, Wbm1058, Welsh, Worrydream, Xaliqen, Xan2, Xaonon, 313 anonymous edits

**Linda (coordination language)** *Source*: http://en.wikipedia.org/w/index.php?oldid=611521684 *Contributors*: A5b, Akavel, AllanBz, Antonielly, Arto B, BookLubber, Bovineone, Britical, Chrismiceli, Ciphergoth, Clark89, Cwolfsheep, Darklock, David Woodward, Decultured, Dekart, Dmanning, Ejrrjs, Fuzzie, Ghais, Ghoseb, Gwizard, Headbomb, Ilion2, Jamelan, Kane5187, Kjwillia, LodeRunner, Lucduponcheel, Lyndonnixon, Ms.wiki.us, PapayaSF, RetroGamer65, Rijkbenik, Rjwilmsi, Rolloffle, Ronaldo, Saper, So-called Genius, Tgmattso, The Anome, The Thing That Should Not Be, UnitedStatesian, Wilfried Elmenreich, Zoicon5, 52 anonymous edits

**Scala (programming language)** *Source*: http://en.wikipedia.org/w/index.php?oldid=611534979 *Contributors*: AJim, Adrianwn, Agentilini, Al3xpayne, Alagwiki, Ale And Quail, AlecTaylor, Alexmadon, AllenZh, AmigoNico, Amolwriteme, Andreas Kaufmann, Andy Dingley, Andyhowlett, Anna Frodesiak, Antoras, Asymmetric, AugPi, Autarch, Authalic, Avesus, Awpc, Axaluss, AxelBoldt, Bdesham, Bekuletz, Bendiolas, Benwing, Bgwhite, Billy gat, BjKa, Bjorn Helgaas, Black Falcon, Blacshark, Bootjack2k, Bovineone, Bpdp, BrownHairedGirl, Brycehughes, BurntSky, BytterPT, C45207, CRGreathouse, CanisRufus, Captain Conundrum, Carl.antuar, Carlo Bonamico, Cccrrraaaiiiggg, Chaecker, Chocolateboy, ChrisEich, ChrisGualtieri, Cleared as filed, Cogiati, Compfreak7, Conaclos, Cybercobra, D, DG, DNewhall, Dacut, DalekCaan42, Danakil, Daniel5Ko, DaveCEO, DavidBiesack, Dcoetzee, Deflective, Delirium, Denisarona, DevOpera, Disnet2007, Download, Dpv, Dvasi, Długosz, Edward, EdwardH, Eed3si9n, Eje211, EngineerScotty, Enum, Eranation, Erdavila, Erwan, Etorreborre, Euphoria, Fadereu, Fayimora, Fela Winkelmolen, Finchsw17, Frap, Frecklefoot, Frietjes, Futurechimp, Garkbit, Gene Ward Smith, Gf uip, Ghettoblaster, Goystein, Graham87, Greenrd, Grshiplett, Gudeldar, Gwern, Haakon, Hairy Dude, HairyFotr, Hello5959us, Hervegirod, Hmains, Hosamaly, HowardBGolden, Huffers, Hydrology, Iggymwangi, Ilya, ItemState, JLaTondre, JWB, Jaffachief, Janto, Jarble, Jbolden1517, Jdforrester, Jdluz, Jkl, Jonabbey, Jonhanson, Joshdmiller, Joswig, Jtnystrom, Juan Miguel Cejuela, Junkblocker, JustWinBaby, Justinnhli, Kaiyannju, Karim.rayani, Katieh5584, Kbrose, Kevinarpe, Krystian.Nowak, Ktpenrose, KuwarOnline, Kwamikagami, Last Contrarian, Ldo, Leotohill, Levin, Lupin, MER-C, MadLex, Madacs, Magioladitis, MalcolmGorman, Mandarax, Matt Crypto, Matthewdunsdon, Mblumber, Mctpyt, Merosonox, Michael miceli, Michaelmalak, Midinastasurazz, Mike Linksvayer, Mike Rosoft, Mkenney, MoreNet, Mortense, Mottainai, Mr. Credible, Mrfly911, Mstroeck, Munjaros, Neuralwarp, NickGarvey, Nivanov, Nxavar, Nymf, Oliver H, Oluies, Omnipaedista, Paercebal, Paul Foxworthy, Paulmillr, Pavel Senatorov, PerLundberg, Pgan002, PieRRoMaN, Pinar, Propensive, Psychonaut, Qrilka, Qutezuce, Qwertyus, RC Howe, RLE64, Ras, Rcaloras, Reidhoch, Rexlen, Rfl, Rich Farmbrough, RickBeton, Rjwilmsi, Rl, Romanc19s, Royote, Runtime, S.Örvarr.S, Sae1962, Salvan, Sam Pointon, Samkass, Samspade79, Sanspeur, Sbmehta, Scarpy, Seanstephens, Seezer, Senorcarbone, Serketan, SethTisue, Sgoder, Sgonyea, Shiki2, Siskus, Sjames1958, Soc88, SpikeTorontoRCP, Spoon!, Squids and Chips, Strake, TJRC, TakuyaMurata, Talandor, Thumperward, ThurnerRupert, Titamation, Tmillsclare, Tobym, Tom Morris, Torc2, Toreau, TripleF, Tritium6, Trond.olsen, TutterMouse, Vincenzo.vitale, Wael Ellithy, Wavelength, Whenning, Wickorama, William Avery, Windofkeltia, Winterstein, Wmwallace, XP1, Yadavjpr, Yukoba, Zaxebo1, Zirco, Zootm, 356 anonymous edits

**Event-driven programming** *Source*: http://en.wikipedia.org/w/index.php?oldid=610804350 *Contributors*: Abdull, AdamDavies50, AnAj, Ashenai, Beland, BenFrantzDale, Bernard Ladenthin, Bigbluefish, Bobox, Bobstay, Bongwarrior, Bradams, Brick Thrower, Bugone, Burschik, Caiyu, Caltas, CanadianLinuxUser, CanisRufus, DVdm, Danakil, DarkFalls, Davelong, Dckelly, Deb, Dfletter, Dicklyon, Diego Moya, Dojarca, DoubleBlue, Dougher, DouglasGreen, DrCroco, Elfguy, Epbr123, Erik Sandberg, Eyu100, FatalError, Flyer22, Fuchsias, Furrykef, Ge0nk, Gioto, Hellozejaykay, Hippietrail, Hmains, Iain99, Ixfd64, J.delanoy, JRWoodwardMSW, Jabbba, Jarble, Jason Quinn, Jmilgram, JonHarder, Keltis2001, Ken Birman, Kirilian, Kku, Krish.roony, Kstevenham, LOL, LilHelpa, LinguistAtLarge, MER-C, MTayel, Mange01, Marjaei, Mark Renier, Mcclain, Mesoderm, Mike Rosoft, Mirosamek, Mortense, MrOllie, Mrfelis, Mrkwpalmer, Mrwaddle34, Naddy, Nanshu, Natalie Mak, Nbarth, Nevyn, NewEnglandYankee, Nh05cs69, Nickg, Nigel V Thomas, PaterMcFly, Pengo, Peter Campbell, Pinar, Pomoxis, Poppafuze, Porton, Possum, Pradeeban, PurpleAluminiumPoodle, QuadrivialMind, Quanticle, R'n'B, Raf1qu3, Retired username, Rhobite, Rich Farmbrough, RichardVeryard, Rrelf, Ruud Koot, S.K., Sankalp.gautam, Satellizer, Sgeureka, Snow Blizzard, Spoonboy42, Stephenb, Swiedenroth, Sysy, Taka, Tardis, Tassedethe, Tevildo, The Epopt, The Widdow's Son, Tide rolls, Tobias Bergemann, Tom harrison, Tresiden, Tumble, Uncle G, Victarus, W Hukriede, Waisbrot, WalterGR, Wapcaplet, Wbm1058, Wpdkc, Wrp103, Xu2008, Xysboybear, ZEJAYKAY, Zarrandreas, Zr40, 292 anonymous edits

**Concurrent Haskell** *Source*: http://en.wikipedia.org/w/index.php?oldid=559727983 *Contributors*: Alksentrs, Alquantor, Awagner83, Axman6, Cae prince, D6, David Eppstein, DdEe4Aai, Fijal, Janm67, Jbolden1517, Jesse V., Liyang, Mandarax, Mild Bill Hiccup, Miym, Sam Pointon, 7 anonymous edits

**Software transactional memory** *Source*: http://en.wikipedia.org/w/index.php?oldid=610334565 *Contributors*: Akim.demaille, Asd.and.Rizzo, Ashley Y, Bartosz, Bcward, Beland, Bkerr, Brouhaha, Carbo1200, Catamorphism, Chris the speller, Colonies Chris, Comps, Computaquark, Crockjack, Cyplm, Damian Yerrick, Damiansoul, DavePeixotto, Dcoetzee, Duncan.Hull, Edward, Ehamberg, EricBarre, Estrabd, Fijal, Fingers-of-Pyrex, Flanfl, Fredrik, Giftlite, Gldnspud, Gurch, Gwern, Harold f, Hernan.wilkinson, Hga, Indil, IronGargoyle, JCLately, JLD, Janm67, Jcea, Joao.m.lourenco, Joswig, Jsled, K sudarsun, K.lee, Kilo-Lima, Kimchy77, Kocio, Liao, Liyang, Maniac18, MattGiuca, MementoVivere, Metageek, Misterwilliam, Miym, Mkcmkc, Moony22, Mortense, Msackman, Neilc, Omnipaedista, ParallelWolverine, Peterdjones, PoisonedQuill, Professor Tournesol, Rich Farmbrough, Rjwilmsi, Rlovtangen, Ruud Koot, Shafigoldwasser, Skaphan, Soumyasch, Spayrard, StefanoC, Svick, Sylvain Pion, Tmcw, Tobias Bergemann, Tomdo08, Ujmmmjjuu, Vyadh, Wavetossed, Will Faught, Yamla, Ykhwong, Кирилл Русанов, 179 anonymous edits

**Go (programming language)** *Source*: http://en.wikipedia.org/w/index.php?oldid=611688333 *Contributors*: .p2502, 1exec1, Abednigo, Abledsoe78, Aclassifier, Adamstac, Aeiuthhiet, Ahmed Fatoum, Aidoor, AlecTaylor, Alexandre Bouthors, Alfredo ougaowen, Am088, Amniarix, Andrewman327, ArglebargleIV, Atomician, Banaticus, BarryNorton, Biasoli, Bkkbrad, Bla5er89, Blaisorblade, Blue Em, Boshomi, Bounce1337, Brianski, Btx40, Chickencha, Choas, Chris, ClaudeX, Codename Lisa, Compfreak7, Curly Turkey, Cybercobra, DAGwyn, DMacks, Daniel.Cardenas, Darxus, Davcamer, Dchestnykh, Diwas, Dl2000, Doceddi, Dratman, Drewlesueur, Dsimic, Egmetcalfe, Elimisteve, Ender409, ErikvanB, Espadrine, Fig wright, Filemon, Flyer22, Fragglet, Fryed-peach, Fshahriar, Garkbit, Gerardohc, Glenn, Glyn normington, Goplexian, Gracefool, Groogle, Guy Harris, Gzhao, Happyrabbit, Hervegirod, Hexene, Hmains, Huihermit, Hydrology, Hydrox, IMneme, Ian13, Isaac B Wagner, JC Chu, JaGa, JamesBrownJr, Jarble, Jcvernaleo, Jerryobject, Jeysaba, Jknacnud, JnRouvignac, JonathanMayUK, Jonovision, JorgePeixoto, Josephmarty, Juancnuno, Julesd, Kendall-K1, Kinema, Kintamanimatt, Letdorf, Loadmaster, Lopifalko, Lost.goblin, Lt. John Harper, Magioladitis, Mark viking, MarsRover, Martijn Hoekstra, Masharabinovich, Matt Crypto, Mdemare, Melnakeeb, Meltonkt, Michael miceli, Mild Bill Hiccup, Mindmatrix, Mkcmkc, Mmautner, Mogism, Mu Mind, MuffledThud, Nagle, Nasnema, Nbarth, Nealmcb, Northgrove, Odie5533, OsamaK, PBS, PatrickFisher, Pdone, Perey, Petter Strandmark, Pgan002, Pgr94, Philwiki, PieterDeBruijn, Pointillist, Porttikivi, Pratyya Ghosh, Prodicus, ProfCoder, Prolog, Qwertyus, R'n'B, RKT, Rahulrulez, Raysonho, Rezonansowy, Rl, Robennals, RoodyAlien, Rthangam77, SF007, Sct72, Shii, ShuklaSannidhya, Sigmundur, Skulldyvan, Slartidan, Soni master, Stefan.karpinski, Stephenalexbrowne, Steve.ruckdashel, Stokito, Strcat, Stybn, Svaksha, Sverdrup, Swehack, Syp, TakuyaMurata, There is a T101 in your kitchen, Thumperward, Tsuba, Tuggler, Tuxcantfly, Twilsonb, Vanished user ty12kl89jq10, Waldir, Wavelength, Wdscxsj, Wei2912, Wickorama, WikiLaurent, Wikipelli, Wrackey, XP1, Xan2, Yves Junqueira, ZacBowling, Zephyrtronium, மதனாஹரன், 364 anonymous edits

**Automatic parallelization** *Source*: http://en.wikipedia.org/w/index.php?oldid=609868443 *Contributors*: A5b, Adavidb, Adrianwn, Alca Isilon, AliveFreeHappy, Beland, Cdiggins, CheekyMonkey, Chip Zero, Chris Pickett, Cswierkowski, DMacks, Davew haverford, DavidCary, DougBoost, Eduard Urbach, Forderud, Furrykef, Glrx, Indil, John of Reading, Kristiankolev, Liao, Magioladitis, Mark viking, Marudubshinki, Mattisse, Michael miceli, Mirstan, Nixeagle, Oleg Alexandrov, Peytonbland, Quuxplusone, Rich Farmbrough, Rjwilmsi, Ruud Koot, Sae1962, Simetrical, SimonP, Simone Campanoni, Vincnet, Wiki ipplepen, Wimt, Woohookitty, 38 anonymous edits

**SystemC** *Source*: http://en.wikipedia.org/w/index.php?oldid=609183766 *Contributors*: Abdull, Agasta, Aijnsleij, Alvin-cs, Armando, Banglore orifice, BasilF, Bcrules82, Beta16, Boing! said Zebedee, Cbdorsett, Cedar101, CesarB, Chowbok, Cnwilliams, Cocolargol, D, Dawn Bard, Dcblack56, Denisarona, Dougher, Edward321, Eschudy, Ferritecore, Firumaru, Forderud, FrankOFFIS, Frap, Gaage, Gaius Cornelius, Greatestrowerever, Greg95060, GregorB, Grey Zz, Gv250, Ihpark5803, Isnow, J04n, Jni, Jpape, Jpbowen, Kanai L Ghosh, Kleidersack, Kulkarniy2k, LisaAHarvey, LodeRunner, Loren.wilton, LouScheffer, Markusaachen, Mikv, Mortense, Mukis, Mushroom, Myn99, Orcioni, Pburka, Phil Boswell, Pmokeefe, ProtocolOH, Qamlof, Quantumor, Reeshtya, SN74LS00, Sam Hocevar, SelfishGenie, Solidoccur, TheDukeVIP, Tijfo098, Umesh sisodia, Underroads4534, Veinor, Veralift, Viteau, Volodymyr Obrizan, Vxl119, Walter.vendraminetto, Weichaoliu, Wimt, Xionglingfeng, Yacitus, Zvar, 150 anonymous edits

**List of important publications in concurrent, parallel, and distributed computing** *Source*: http://en.wikipedia.org/w/index.php?oldid=583006603 *Contributors*: Curb Chain, Cybercobra, Dream Focus, Khazar2, Miym, RobinK, Ruud Koot, StephanNaro, Timotheus Canens

# Image Sources, Licenses and Contributors

**File:IBM Blue Gene P supercomputer.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:IBM_Blue_Gene_P_supercomputer.jpg *License*: Creative Commons Attribution-Sharealike 2.0 *Contributors*: Argonne National Laboratory's Flickr page

**File:AmdahlsLaw.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:AmdahlsLaw.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: AnonMoos, Bender235, Ebraminio, JRGomà, McZusatz, Phatom87, Senator2029, Utar

**Image:Optimizing-different-parts.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Optimizing-different-parts.svg *License*: Public Domain *Contributors*: Gorivero

**Image:Fivestagespipeline.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:Fivestagespipeline.png *License*: GNU Free Documentation License *Contributors*: User:Poil

**Image:Superscalarpipeline.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Superscalarpipeline.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Amit6, original version (File:Superscalarpipeline.png) by User:Poil

**Image:Numa.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Numa.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Raul654

**Image:Beowulf.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Beowulf.jpg *License*: GNU General Public License *Contributors*: User Linuxbeak on en.wikipedia

**Image:BlueGeneL cabinet.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:BlueGeneL_cabinet.jpg *License*: GNU Free Documentation License *Contributors*: Morio, Raul654, Yaleks

**Image:NvidiaTesla.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:NvidiaTesla.jpg *License*: Public Domain *Contributors*: Adamantios, Calvinsouto, Cirt, Hyins, Jaceksoci68, Qurren, WikipediaMaster

**File:Cray 1 IMG 9126.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Cray_1_IMG_9126.jpg *License*: Creative Commons Attribution-Sharealike 2.0 *Contributors*: User:Rama/use_my_images

**Image:ILLIAC 4 parallel computer.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:ILLIAC_4_parallel_computer.jpg *License*: Creative Commons Attribution 2.0 *Contributors*: Edward, FAEP, FlickrLickr, Hike395, Shizhao

**Image:NUMA.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:NUMA.svg *License*: Public Domain *Contributors*: Moop2000

**Image:Hwloc.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:Hwloc.png *License*: unknown *Contributors*: The Portable Hardware Locality (hwloc) Project. (Screenshot by the Open Source Grid Engine Project)

**File:Crossbar-hy1.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Crossbar-hy1.jpg *License*: Public Domain *Contributors*: Jim.henderson, Yeatesh

**File:Crossbar-mini-hy2.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Crossbar-mini-hy2.jpg *License*: Public Domain *Contributors*: Jim.henderson, Yeatesh

**File:C Xbr sw HON jeh.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:C_Xbr_sw_HON_jeh.jpg *License*: Creative Commons Zero *Contributors*: Jim.henderson

**File:Crossbar-banjo2-hy.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Crossbar-banjo2-hy.jpg *License*: GNU Free Documentation License *Contributors*: Yeatesh

**Image:NetworkTopology-FullyConnected.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:NetworkTopology-FullyConnected.png *License*: Public Domain *Contributors*: Foobaz, Glosser.ca, Kilom691, LoStrangolatore, 1 anonymous edits

**File:Building a Rural Wireless Mesh Network - A DIY Guide v0.8.pdf** *Source*: http://en.wikipedia.org/w/index.php?title=File:Building_a_Rural_Wireless_Mesh_Network_-_A_DIY_Guide_v0.8.pdf *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: David Johnson, Karel Matthee, Dare Sokoya, Lawrence Mboweni, Ajay Makan, and Henk Kotze (Wireless Africa, Meraka Institute, South Africa)

**Image:Hypercubestar.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Hypercubestar.svg *License*: Public Domain *Contributors*: CommonsDelinker, Incnis Mrsi, Itai, Mate2code, 4 anonymous edits

**File:Hypercubeconstruction.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:Hypercubeconstruction.png *License*: Public Domain *Contributors*: Original uploader was Stdazi at en.wikipedia

**Image:Dual Core Generic.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Dual_Core_Generic.svg *License*: Public Domain *Contributors*: Original uploader was CountingPine at en.wikipedia

**Image:E6750bs8.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:E6750bs8.jpg *License*: Public Domain *Contributors*: Original uploader was GuitarFreak at en.wikipedia

**Image:Athlon64x2-6400plus.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Athlon64x2-6400plus.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Babylonfive David W. Smith

**File:SMP - Symmetric Multiprocessor System.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:SMP_-_Symmetric_Multiprocessor_System.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Ferry24.Milan

**Image:Shared memory.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Shared_memory.svg *License*: GNU Free Documentation License *Contributors*: en:User:Khazadum, User:Stannered

**File:Distributed-parallel.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Distributed-parallel.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Miym

**File:MEGWARE.CLIC.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:MEGWARE.CLIC.jpg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: MEGWARE Computer GmbH

**File:Sun Microsystems Solaris computer cluster.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Sun_Microsystems_Solaris_computer_cluster.jpg *License*: Creative Commons Attribution 2.0 *Contributors*: FAEP, Ferdinand Porsche, FlickreviewR, Foroa, NapoliRoma

**File:Beowulf.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Beowulf.jpg *License*: GNU General Public License *Contributors*: User Linuxbeak on en.wikipedia

**File:SPEC-1 VAX 05.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:SPEC-1_VAX_05.jpg *License*: GNU Free Documentation License *Contributors*: Joe Mabel

**File:Balanceamento de carga (NAT).jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Balanceamento_de_carga_(NAT).jpg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Duesentrieb, It Is Me Here, Joolz, Mdd, Nuno Tavares, Ustas

**File:beowulf.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:Beowulf.png *License*: Public Domain *Contributors*: Mukarramahmad

**File:Nec-cluster.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Nec-cluster.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Hindermath

**File:Cubieboard HADOOP cluster.JPG** *Source*: http://en.wikipedia.org/w/index.php?title=File:Cubieboard_HADOOP_cluster.JPG *License*: Public Domain *Contributors*: Popolon

**File:StreamingModelExample.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:StreamingModelExample.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Anujkaliaiitd

**File:FPGARetrocomputing.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:FPGARetrocomputing.jpg *License*: Creative Commons Attribution-Sharealike 2.0 *Contributors*: Viacheslav Slavinsky

**File:Vector06cc_On-Screen_Display_Menu.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Vector06cc_On-Screen_Display_Menu.jpg *License*: Creative Commons Attribution-Sharealike 2.0 *Contributors*: Viacheslav Slavinsky

**File:An Amstrad CPC 464...in hardware.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:An_Amstrad_CPC_464...in_hardware.jpg *License*: Creative Commons Attribution 2.0 *Contributors*: Blake Patterson

**File:Altera StratixIVGX FPGA.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Altera_StratixIVGX_FPGA.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Altera Corporation

**File:Fpga xilinx spartan.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Fpga_xilinx_spartan.jpg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Dake

**File:Xilinx Zynq-7000 AP SoC.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Xilinx_Zynq-7000_AP_SoC.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Fpganut

**File:FPGA cell example.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:FPGA_cell_example.png *License*: Public Domain *Contributors*: Petter.kallstrom

**File:logic block pins.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Logic_block_pins.svg *License*: Public Domain *Contributors*: User:Joelholdsworth

**File:switch box.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Switch_box.svg *License*: GNU Free Documentation License *Contributors*: Traced by User:Stannered from an work by en:User:Ignaciomella

**File:SSDTR-ASIC technology.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:SSDTR-ASIC_technology.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Livewireinnovation

**File:S-MOS_Systems_ASIC_SLA6140.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:S-MOS_Systems_ASIC_SLA6140.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Antoinebercovici

**File:VLSI_VL82C486_Single_Chip_486_System_Controller_HV.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:VLSI_VL82C486_Single_Chip_486_System_Controller_HV.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Antoinebercovici

**File:Cray J90 CPU module.jpg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Cray_J90_CPU_module.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Mu301

**file:Wikibooks-logo-en-noslogan.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Wikibooks-logo-en-noslogan.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Bastique, User:Ramac et al.

**File:TIOBE Scala Ranking.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:TIOBE_Scala_Ranking.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Siskus

**File:Flag of Australia.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_Australia.svg *License*: Public Domain *Contributors*: Anomie, Mifter

**File:Flag of the United States.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_the_United_States.svg *License*: Public Domain *Contributors*: Anomie

**File:Flag of the Netherlands.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_the_Netherlands.svg *License*: Public Domain *Contributors*: Zscout370

**File:Flag of France.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_France.svg *License*: Public Domain *Contributors*: Anomie

**File:Flag of the United Kingdom.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_the_United_Kingdom.svg *License*: Public Domain *Contributors*: Anomie, Good Olfactory, MSGJ, Mifter

**File:Flag of Switzerland.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_Switzerland.svg *License*: Public Domain *Contributors*: User:Marc Mongenet Credits: User:-xfi- User:Zscout370

**File:Flag of Canada.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_Canada.svg *License*: Public Domain *Contributors*: Anomie

**File:Flag of Finland.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_Finland.svg *License*: Public Domain *Contributors*: Drawn by User:SKopp

**File:Flag of Germany.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_Germany.svg *License*: Public Domain *Contributors*: Anomie

**File:Flag of Japan.svg** *Source*: http://en.wikipedia.org/w/index.php?title=File:Flag_of_Japan.svg *License*: Public Domain *Contributors*: Anomie

**File:golang.png** *Source*: http://en.wikipedia.org/w/index.php?title=File:Golang.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Artem Korzhimanov, Denniss, Zhantongz

# License