WIKIPEDIA

Parallel computing

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.[1] Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has long been employed in high-performance computing, but it's gaining broader interest due to the physical constraints preventing frequency scaling.[2] As power consumption (and consequently heat generation) by computers has become a concern in recent years,[3] parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.[4]

IBM's Blue Gene/P massively parallel supercomputer.

Parallel computing is closely related to concurrent computing—they are frequently used together, and often conflated, though the two are distinct: it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU).[5][6] In parallel computing, a computational task is typically broken down into several, often many, very similar sub-tasks that can be processed independently and whose results are combined afterwards, upon completion. In contrast, in concurrent computing, the various processes often do not address related tasks; when they do, as is typical in distributed computing, the separate tasks may have a varied nature and often require some inter-process communication during execution.

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.

In some cases parallelism is transparent to the programmer, such as in bit-level or instruction-level parallelism, but explicitly parallel algorithms, particularly those that use concurrency, are more difficult to write than sequential ones,[7] 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.

A theoretical upper bound on the speed-up of a single program as a result of parallelization is given by Amdahl's law.

Contents

Background
- Amdahl's law and Gustafson's law
- Dependencies
- Race conditions, mutual exclusion, synchronization, and parallel slowdown
- Fine-grained, coarse-grained, and embarrassing parallelism
- Consistency models
- Flynn's taxonomy

Types of parallelism

Bit-level parallelism
Instruction-level parallelism
Task parallelism
Hardware
Memory and communication
Classes of parallel computers
Multi-core computing
Symmetric multiprocessing
Distributed computing
Specialized parallel computers
Software
Parallel programming languages
Automatic parallelization
Application checkpointing
Algorithmic methods
Fault tolerance
History
Biological brain as massively parallel computer
See also
References
Further reading
External links

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 one is executed.[8]

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.[8] Historically parallel computing was used for scientific computing and the simulation of scientific problems, particularly in the natural and engineering sciences, such as meteorology. This led to the design of parallel hardware and software, as well as high performance computing.[9]

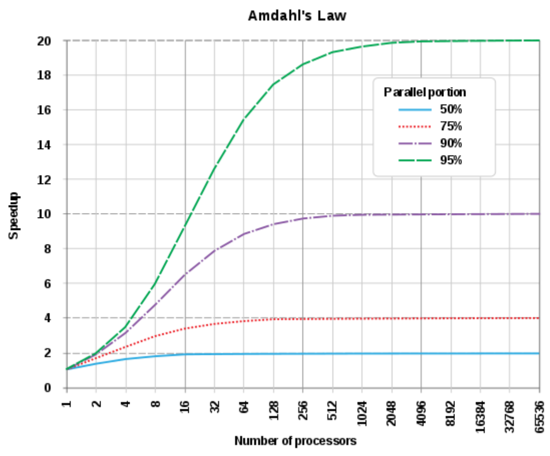
Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all compute-bound programs.[10] However, power consumption P by a chip is given by the equation $P = C \times V^2 \times F$, where 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).[11] Increases in frequency increase the amount of power used in a processor. Increasing processor power consumption led ultimately to Intel's May 8, 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.[12]

To deal with the problem of power consumption and overheating the major central processing unit (CPU or processor) manufacturers started to produce power efficient processors with multiple cores. The core is the computing unit of the processor and in multi-core processors each core is independent and can access the same memory concurrently. Multi-core processors have brought parallel computing to desktop computers. Thus parallelisation of serial programmes has become a mainstream programming task. In 2012 quad-core processors became standard for desktop computers, while servers have 10 and 12 core processors. From Moore's law it can be predicted that the number of cores per processor will double every 18–24 months. This could mean that after 2020 a typical processor will have dozens or hundreds of cores.[13]

An operating system can ensure that different tasks and user programmes are run in parallel on the available cores. However, for a serial software programme to take full advantage of the multi-core architecture the programmer needs to restructure and parallelise the code. A speed-up of application software runtime will no longer be achieved through frequency scaling, instead programmers will need to parallelise their software code to take advantage of the increasing computing power of multicore architectures.[14]

Amdahl's law and Gustafson's law

Optimally, the speedup 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 speedup. Most of them have a near-linear speedup 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 speedup 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 speedup using parallel computing would be 10 times no matter how many processors are used.

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law[15]

$$S_{\text{latency}}(s) = \frac{1}{1-p+\frac{p}{s}},$$

where

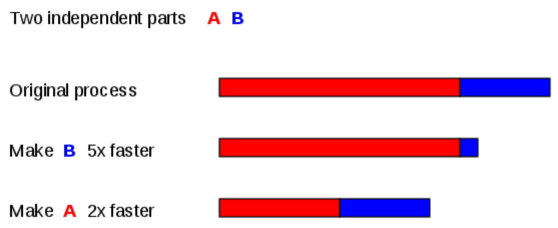
- S_{latency} is the potential speedup in latency of the execution of the whole task;
- s is the speedup in latency of the execution of the parallelizable part of the task;
- p is the percentage of the execution time of the whole task concerning the parallelizable part of the task *before parallelization*.

Since $S_{\text{latency}} < 1/(1 - p)$, it shows that a small part of the program which cannot be parallelized will limit the overall speedup available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (serial) parts. If the non-parallelizable part of a program accounts for 10% of the runtime (p = 0.9), we can get no more than a 10 times speedup, 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."[16]

Amdahl's law only applies to cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work.[17] In this case, Gustafson's law gives a less pessimistic and more realistic assessment of parallel performance:[18]

$$S_{\text{latency}}(s) = 1 - p + sp.$$

Both Amdahl's law and Gustafson's law assume that the running time of the serial part 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*.



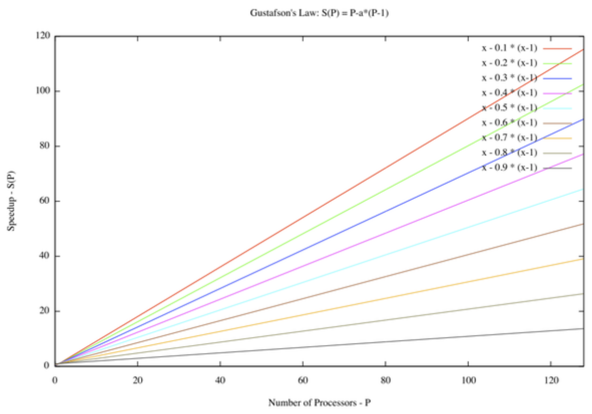
Assume that a task has two independent parts, *A* and *B*. Part *B* takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part *A* be twice as fast. This will make the computation much faster than by optimizing part *B*, even though part *B*'s speedup is greater by ratio, (5 times versus 2 times).

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[19] 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

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



A graphical representation of Gustafson's law.

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 produces a variable needed by the first segment. 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.[20]

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
```

In this example, instruction 3 cannot be executed before (or even in parallel with) instruction 2, because instruction 3 uses a result from instruction 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.[21] Threads will often need synchronized access to an object or other resource, for example when they must update a variable that is shared between them. Without synchronization, the instructions between the two threads 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 may be necessary to ensure correct program execution when threads must serialize access to resources, but their use can greatly slow a program and may affect its reliability.[22]

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.[23]

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a lock or a semaphore.[24] 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.[25]

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 or waiting on each other for access to resources.[26][27] Once the overhead from resource contention or communication dominates the time spent on other computation, further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This problem, known as parallel slowdown,[28] can be improved in some cases by software analysis and redesign.[29]

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 exhibits embarrassing parallelism if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

Consistency models

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".[30]

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. Introduced in 1962, Petri nets 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 π-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.

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."[31]

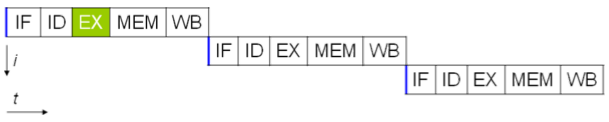
Types of parallelism

Bit-level parallelism

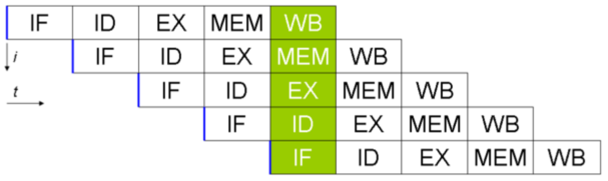
From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle.[32] Increasing the word size reduces 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 the early 2000s, with the advent of x86-64 architectures, did 64-bit processors become commonplace.

Instruction-level parallelism



A canonical processor without pipeline. It takes five clock cycles to complete one instruction and thus the processor can issue subscalar performance (IPC = 0.2 < 1).

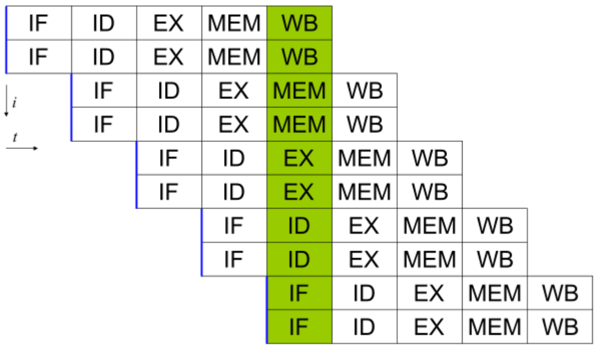


A canonical five-stage pipelined processor. In the best case scenario, it takes one clock cycle to complete one instruction and thus the processor can issue scalar performance (IPC = 1).

A computer program is, in essence, a stream of instructions executed by a processor. Without instruction-level parallelism, a processor can only issue less than one instruction per clock cycle (IPC < 1). These processors are known as *subscalar* processors. 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.[33]

All 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 and thus can issue one instruction per clock cycle (IPC = 1). These processors are known as *scalar* processors. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and register write back (WB). The Pentium 4 processor had a 35-stage pipeline.[34]

Most modern processors also have multiple execution units. They usually combine this feature with pipelining and thus can issue more than one instruction per clock cycle (IPC > 1). These processors 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 canonical five-stage pipelined superscalar processor. In the best case scenario, it takes one clock cycle to complete two instructions and thus the processor can issue superscalar performance (IPC = 2 > 1).

Task parallelism

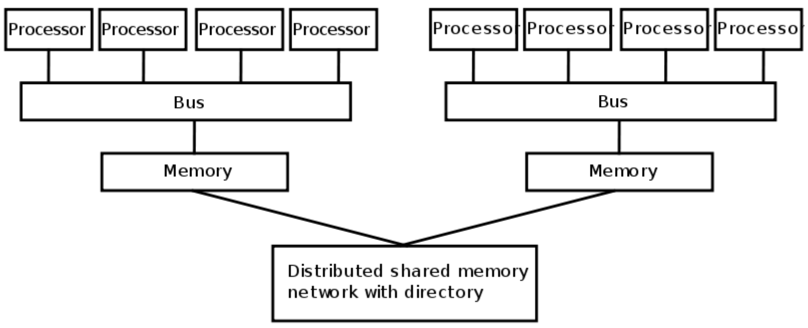
Task parallelisms is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data".[35] This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism involves the decomposition of a task into sub-tasks and then allocating each sub-task to a processor for execution. The processors would then execute these sub-tasks concurrently and often cooperatively. Task parallelism does not usually scale with the size of a problem.[36]

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).[37] 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. On the supercomputers, distributed shared memory space can be implemented using the programming model such as PGAS. This model allows processes on one compute node to transparently access the remote memory of another compute node. All compute nodes are also connected to an external shared memory system via high-speed interconnect, such as Infiniband, this external shared memory system is known as burst buffer, which is typically built from arrays of non-volatile memory physically distributed across multiple I/O nodes.

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



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.

Computer systems make use of caches—small and 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.[37]

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 interconnected 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.

Multi-core computing

A multi-core processor is a processor that includes multiple processing units (called "cores") on the same chip. This processor differs from a superscalar processor, which includes multiple execution units and can issue multiple instructions per clock cycle from one instruction stream (thread); in contrast, a multi-core processor can issue multiple instructions per clock cycle from multiple instruction streams. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is a prominent multi-core processor. Each core in a multi-core processor can potentially be superscalar as well—that is, on every clock cycle, each core can issue multiple instructions from one thread.

Simultaneous multithreading (of which Intel's Hyper-Threading is the best known) was an early form of pseudo-multi-coreism. A processor capable of concurrent multithreading includes multiple execution units in the same processing unit—that is it has a superscalar architecture—and can issue multiple instructions per clock cycle from *multiple* threads. Temporal multithreading on the other hand includes a single execution unit in the same processing unit and can issue one instruction at a time from *multiple* threads.

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus.[38] Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors.[39] 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.[38]

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. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them.[40] The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel.[41]

Cluster computing

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.[42] 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.[43] Beowulf technology was originally developed by Thomas Sterling and Donald Becker. 87% of all Top500 supercomputers are clusters.[44] The remaining are Massively Parallel Processors, explained below.

A Beowulf cluster.

Because grid computing systems (described below) can easily handle embarrassingly parallel problems, modern clusters are typically designed to handle more difficult problems—problems that require nodes to share intermediate results with each other more often. This requires a high bandwidth and, more importantly, a low-latency interconnection network. Many historic and current supercomputers use customized high-performance network hardware specifically designed for cluster computing, such as the Cray Gemini network.[45] As of 2014, most current supercomputers use some off-the-shelf standard network hardware, often Myrinet, InfiniBand, or Gigabit Ethernet.

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.[46] In an 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."[47]

A cabinet from IBM's Blue Gene/L massively parallel supercomputer.

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

Grid computing

Grid 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.[48]

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.[49] 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."[49]

General-purpose computing on graphics processing units (GPGPU)

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.[50] Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.

Nvidia's Tesla GPGPU card

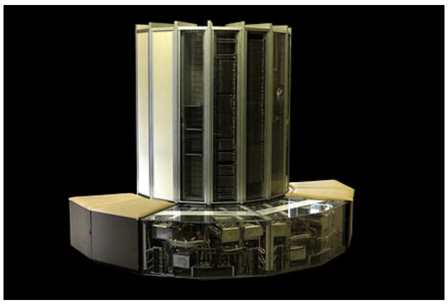
In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively. 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

Several application-specific integrated circuit (ASIC) approaches have been devised for dealing with parallel applications.[51][52][53]

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 UV photolithography. This process requires a mask set, which can be extremely expensive. A mask set can cost over a million US dollars.[54] (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.[49] 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 PFLOPS RIKEN MDGRAPE-3 machine which uses custom ASICs for molecular dynamics simulation.

Vector processors


The Cray-1 is a 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.[55] They are closely related to Flynn's SIMD classification.[55]

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 Freescale Semiconductor's AltiVec and Intel's Streaming SIMD Extensions (SSE).

Software

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 the most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API.[56] 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 hybrid multi-core parallel programming (HMPP) 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.

The rise of consumer GPUs has led to support for compute kernels, either in graphics APIs (referred to as compute shaders), in dedicated APIs (such as OpenCL), or in other language extensions.

Automatic parallelization

Automatic parallelization of a sequential program by a compiler is the "holy grail" of parallel computing, especially with the aforementioned limit of processor frequency. Despite decades of work by compiler researchers, automatic parallelization has had only limited success.[57]

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, SequenceL, System C (for FPGAs), Mitrion-C, VHDL, and Verilog.

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.[58]

Algorithmic methods

As parallel computers become larger and faster, we are now able to solve problems that had previously taken too long to run. Fields as varied as bioinformatics (for protein folding and sequence analysis) and economics (for mathematical finance) have taken advantage of parallel computing. Common types of problems in parallel computing applications include:[59]

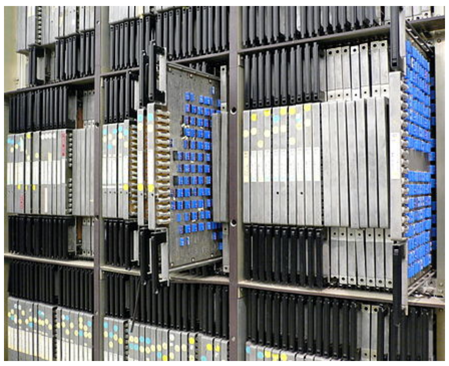
- Dense linear algebra
- Sparse linear algebra
- Spectral methods (such as Cooley–Tukey fast Fourier transform)
- *N*-body problems (such as Barnes–Hut simulation)
- structured grid problems (such as Lattice Boltzmann methods)
- Unstructured grid problems (such as found in finite element analysis)
- Monte Carlo method
- 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

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 fails, and also allows automatic error detection and error correction if the results differ. These methods can be used to help prevent single-event upsets caused by transient errors.[60] Although additional measures may be required in embedded or specialized systems, this method can provide a cost effective approach to achieve n-modular redundancy in commercial off-the-shelf systems.

History

The origins of true (MIMD) parallelism go back to Luigi Federico Menabrea and his *Sketch of the Analytic Engine Invented by Charles Babbage*.[62][63][64]


ILLIAC IV, "the most infamous of supercomputers".[61]

In April 1958, S. Gill (Ferranti) discussed parallel programming and the need for branching and waiting.[65] Also in 1958, IBM researchers John Cocke and Daniel Slotnick discussed the use of parallelism in numerical calculations for the first time.[66] Burroughs Corporation introduced the D825 in 1962, a four-processor computer that accessed up to 16 memory modules through a crossbar switch.[67] In 1967, Amdahl and Slotnick published a debate about the feasibility of parallel processing at American

Federation of Information Processing Societies Conference.[66] It was during this debate that Amdahl's law was coined to define the limit of speed-up due to parallelism.

In 1969, Honeywell introduced its first Multics system, a symmetric multiprocessor system capable of running up to eight processors in parallel.[66] C.mmp, a multi-processor project at Carnegie Mellon University in the 1970s, was among the first multiprocessors with more than a few processors. The first bus-connected multiprocessor with snooping caches was the Synapse N+1 in 1984.[63]

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.[68] In 1964, Slotnick had proposed building a massively parallel computer for the Lawrence Livermore National Laboratory.[66] His design was funded by the US Air Force, which was the earliest SIMD parallel-computing effort, ILLIAC IV.[66] 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.[61] 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.

Biological brain as massively parallel computer

In the early 1970s, at the MIT Computer Science and Artificial Intelligence Laboratory, Marvin Minsky and Seymour Papert started developing the *Society of Mind* theory, which views the biological brain as massively parallel computer. In 1986, Minsky published *The Society of Mind*, which claims that “mind is formed from many little agents, each mindless by itself”.[69] The theory attempts to explain how what we call intelligence could be a product of the interaction of non-intelligent parts. Minsky says that the biggest source of ideas about the theory came from his work in trying to create a machine that uses a robotic arm, a video camera, and a computer to build with children's blocks.[70]

Similar models (which also view the biological brain as a massively parallel computer, i.e., the brain is made up of a constellation of independent or semi-independent agents) were also described by:

- Thomas R. Blakeslee,[71]
- Michael S. Gazzaniga,[72][73]
- Robert E. Ornstein,[74]
- Ernest Hilgard,[75][76]
- Michio Kaku,[77]
- George Ivanovich Gurdjieff,[78]
- Neurocluster Brain Model.[79]

See also

- Concurrency (computer science)
- Content Addressable Parallel Processor
- List of distributed computing conferences
- List of important publications in concurrent, parallel, and distributed computing
- Manycore
- Multi tasking
- Parallel programming model
- Serializability
- Synchronous programming
- Transputer
- Vector processing

References

1. Gottlieb, Allan; Almasi, George S. (1989). *Highly parallel computing* (http://dl.acm.org/citation.cfm?id=160438). Redwood City, Calif.: Benjamin/Cummings. ISBN 978-0-8053-0177-9.
2. S.V. Adve *et al.* (November 2008). "Parallel Computing Research at Illinois: The UPCRC Agenda" (https://graphics.cs.illinois.edu/sites/default/files/upcrc-wp.pdf) Archived (https://web.archive.org/web/20180111165735/https://graphics.cs.illinois.edu/sites/default/files/upcrc-wp.pdf) 2018-01-11 at the Wayback Machine (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."
3. 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".
4. 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 limits."
5. "Concurrency is not Parallelism", *Waza conference* Jan 11, 2012, Rob Pike (slides (https://talks.golang.org/2012/waza.slide)) (video (https://vimeo.com/49718712))
6. "Parallelism vs. Concurrency" (https://wiki.haskell.org/Parallelism_vs._Concurrency). *Haskell Wiki*.
7. Hennessy, John L.; Patterson, David A.; Larus, James R. (1999). *Computer organization and design: the hardware/software interface* (https://archive.org/details/computerorganiz000henn) (2. ed., 3rd print. ed.). San Francisco: Kaufmann. ISBN 978-1-55860-428-5.
8. Barney, Blaise. "Introduction to Parallel Computing" (https://www.llnl.gov/computing/tutorials/parallel_comp/). Lawrence Livermore National Laboratory. Retrieved 2007-11-09.
9. Thomas Rauber; Gudula Rünger (2013). *Parallel Programming: for Multicore and Cluster Systems*. Springer Science & Business Media. p. 1. ISBN 9783642378010.
10. Hennessy, John L.; Patterson, David A. (2002). *Computer architecture / a quantitative approach* (3rd ed.). San Francisco, Calif.: International Thomson. p. 43. ISBN 978-1-55860-724-8.
11. Rabaey, Jan M. (1996). *Digital integrated circuits : a design perspective*. Upper Saddle River, N.J.: Prentice-Hall. p. 235. ISBN 978-0-13-178609-7.
12. Flynn, Laurie J. (8 May 2004). "Intel Halts Development Of 2 New Microprocessors" (https://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007). *New York Times*. Retrieved 5 June 2012.
13. Thomas Rauber; Gudula Rünger (2013). *Parallel Programming: for Multicore and Cluster Systems*. Springer Science & Business Media. p. 2. ISBN 9783642378010.
14. Thomas Rauber; Gudula Rünger (2013). *Parallel Programming: for Multicore and Cluster Systems*. Springer Science & Business Media. p. 3. ISBN 9783642378010.
15. Amdahl, Gene M. (1967). "Validity of the single processor approach to achieving large scale computing capabilities" (http://dl.acm.org/citation.cfm?id=160438). *Proceeding AFIPS '67 (Spring) Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*: 483–485. doi:10.1145/1465482.1465560 (https://doi.org/10.1145%2F1465482.1465560).
16. Brooks, Frederick P. (1996). *The mythical man month essays on software engineering* (https://archive.org/details/mythicalmonth00broo) (Anniversary ed., repr. with corr., 5. [Dr.] ed.). Reading, Mass. [u.a.]: Addison-Wesley. ISBN 978-0-201-83595-3.
17. Michael McCool; James Reinders; Arch Robison (2013). *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier. p. 61.

18. Gustafson, John L. (May 1988). "Reevaluating Amdahl's law" (https://web.archive.org/web/20070927040654/http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html). *Communications of the ACM*. **31** (5): 532–533. CiteSeerX 10.1.1.509.6892 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.509.6892). doi:10.1145/42411.42415 (https://doi.org/10.1145%2F42411.42415). Archived from the original (http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html) on 2007-09-27.
19. Bernstein, A. J. (1 October 1966). "Analysis of Programs for Parallel Processing". *IEEE Transactions on Electronic Computers*. EC-15 (5): 757–763. doi:10.1109/PGEC.1966.264565 (https://doi.org/10.1109%2FPGEC.1966.264565).
20. Roosta, Seyed H. (2000). *Parallel processing and parallel algorithms : theory and computation*. New York, NY [u.a.]: Springer. p. 114. ISBN 978-0-387-98716-3.
21. "Processes and Threads" (https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx). *Microsoft Developer Network*. Microsoft Corp. 2018. Retrieved 2018-05-10.
22. Krauss, Kirk J (2018). "Thread Safety for Performance" (http://www.developforperformance.com/ThreadSafetyForPerformance.html). *Develop for Performance*. Retrieved 2018-05-10.
23. Tanenbaum, Andrew S. (2002-02-01). "Introduction to Operating System Deadlocks" (http://www.informit.com/articles/article.aspx?p=25193). *Informit*. Pearson Education, Informit. Retrieved 2018-05-10.
24. Cecil, David (2015-11-03). "Synchronization internals – the semaphore" (https://www.embedded.com/design/operating-systems/4440752/Synchronization-internals----the-semaphore). *Embedded*. AspenCore. Retrieved 2018-05-10.
25. Preshing, Jeff (2012-06-08). "An Introduction to Lock-Free Programming" (http://preshing.com/20120612/an-introduction-to-lock-free-programming/). *Preshing on Programming*. Retrieved 2018-05-10.
26. "What's the opposite of "embarrassingly parallel"?" (https://stackoverflow.com/questions/806569/whats-the-opposite-of-embarrassingly-parallel). *StackOverflow*. Retrieved 2018-05-10.
27. Schwartz, David (2011-08-15). "What is thread contention?" (https://stackoverflow.com/questions/1970345/what-is-thread-contention). *StackOverflow*. Retrieved 2018-05-10.
28. Kukanov, Alexey (2008-03-04). "Why a simple test can get parallel slowdown" (https://software.intel.com/en-us/blogs/2008/03/04/why-a-simple-test-can-get-parallel-slowdown). Retrieved 2015-02-15.
29. Krauss, Kirk J (2018). "Threading for Performance" (http://www.developforperformance.com/ThreadingForPerformance.html). *Develop for Performance*. Retrieved 2018-05-10.
30. Lamport, Leslie (1 September 1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". *IEEE Transactions on Computers*. **C-28** (9): 690–691. doi:10.1109/TC.1979.1675439 (https://doi.org/10.1109%2FTC.1979.1675439).
31. Patterson and Hennessy, p. 748.
32. Singh, David Culler; J.P. (1997). *Parallel computer architecture* ([Nachdr.] ed.). San Francisco: Morgan Kaufmann Publ. p. 15. ISBN 978-1-55860-343-1.
33. Culler et al. p. 15.
34. 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) Archived (https://web.archive.org/web/20080414141000/http://users.ece.utexas.edu/~patt/Videos/talk_videos/cmu_04-29-04.wmv) 2008-04-14 at the Wayback Machine (wmv). Distinguished Lecturer talk at Carnegie Mellon University. Retrieved on November 7, 2007.
35. Culler et al. p. 124.
36. Culler et al. p. 125.
37. Patterson and Hennessy, p. 713.
38. Hennessy and Patterson, p. 549.
39. Patterson and Hennessy, p. 714.
40. Ghosh (2007), p. 10. Keidar (2008).
41. Lynch (1996), p. xix, 1–2. Peleg (2000), p. 1.
42. What is clustering? (http://www.webopedia.com/TERM/c/clustering.html) Webopedia computer dictionary. Retrieved on November 7, 2007.
43. Beowulf definition. (https://www.pcmag.com/encyclopedia_term/0,2542,t=Beowulf&i=38548,00.asp) *PC Magazine*. Retrieved on November 7, 2007.

44. "List Statistics | TOP500 Supercomputer Sites" (https://www.top500.org/statistics/list/). *www.top500.org*. Retrieved 2018-08-05.
45. "Interconnect" (https://www.nersc.gov/users/computational-systems/hopper/configuration/interconnect/) Archived (https://web.archive.org/web/20150128133120/https://www.nersc.gov/users/computational-systems/hopper/configuration/interconnect/) 2015-01-28 at the Wayback Machine.
46. Hennessy and Patterson, p. 537.
47. MPP Definition. (https://www.pcmag.com/encyclopedia_term/0,,t=mpp&i=47310,00.asp) *PC Magazine*. Retrieved on November 7, 2007.
48. Kirkpatrick, Scott (2003). "COMPUTER SCIENCE: Rough Times Ahead". *Science*. **299** (5607): 668–669. doi:10.1126/science.1081623 (https://doi.org/10.1126%2Fscience.1081623). PMID 12560537 (https://www.ncbi.nlm.nih.gov/pubmed/12560537).
49. D'Amour, Michael R., Chief Operating Officer, DRC Computer Corporation. "Standard Reconfigurable Computing". Invited speaker at the University of Delaware, February 28, 2007.
50. 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.
51. 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.
52. Shimokawa, Y.; Fuwa, Y.; Aramaki, N. (18–21 November 1991). "A parallel ASIC VLSI neurocomputer for a large number of neurons and billion connections per second speed" (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=170708). *International Joint Conference on Neural Networks*. **3**: 2162–2167. doi:10.1109/IJCNN.1991.170708 (https://doi.org/10.1109%2FIJCNN.1991.170708). ISBN 978-0-7803-0227-3.
53. Acken, Kevin P.; Irwin, Mary Jane; Owens, Robert M. (July 1998). "A Parallel ASIC Architecture for Efficient Fractal Image Coding". *The Journal of VLSI Signal Processing*. **19** (2): 97–113. doi:10.1023/A:1008005616596 (https://doi.org/10.1023%2FA%3A1008005616596).
54. 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) Archived (https://web.archive.org/web/20080131221732/http://www.future-fab.com/documents.asp?grID=353&d_ID=2596) 2008-01-31 at the Wayback Machine." 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."
55. Patterson and Hennessy, p. 751.
56. 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"
57. Shen, John Paul; Mikko H. Lipasti (2004). *Modern processor design : fundamentals of superscalar processors* (1st ed.). Dubuque, Iowa: McGraw-Hill. p. 561. ISBN 978-0-07-057064-1. "However, the holy grail of such research—automated parallelization of serial programs—has yet to materialize. While automated parallelization of certain classes of algorithms has been demonstrated, such success has largely been limited to scientific and numeric applications with predictable flow control (e.g., nested loop structures with statically determined iteration counts) and statically analyzable memory access patterns. (e.g., walks over large multidimensional arrays of float-point data)."
58. *Encyclopedia of Parallel Computing, Volume 4* by David Padua 2011 ISBN 0387097651 page 265
59. 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.
60. Dobel, B., Hartig, H., & Engel, M. (2012) "Operating system support for redundant multithreading". *Proceedings of the Tenth ACM International Conference on Embedded Software*, 83–92. doi:10.1145/2380356.2380375 (https://doi.org/10.1145%2F2380356.2380375)

61. Patterson and Hennessy, pp. 749–50: "Although successful in pushing several technologies useful in later projects, the ILLIAC IV failed as a computer. Costs escalated from the $8 million estimated in 1966 to $31 million by 1972, despite the construction of only a quarter of the planned machine . It was perhaps the most infamous of supercomputers. The project started in 1965 and ran its first real application in 1976."
62. 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. quote: "when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes."
63. Patterson and Hennessy, p. 753.
64. R.W. Hockney, C.R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms, Volume 2* (https://books.google.com/books?id=6HcBQ67-Fb4C). 1988. p. 8 quote: "The earliest reference to parallelism in computer design is thought to be in General L. F. Menabrea's publication in… 1842, entitled *Sketch of the Analytical Engine Invented by Charles Babbage*".
65. "Parallel Programming", S. Gill, *The Computer Journal* Vol. 1 #1, pp2-10, British Computer Society, April 1958.
66. Wilson, Gregory V. (1994). "The History of the Development of Parallel Computing" (http://ei.cs.vt.edu/~history/Parallel.html). Virginia Tech/Norfolk State University, Interactive Learning with a Digital Library in Computer Science. Retrieved 2008-01-08.
67. Anthes, Gry (November 19, 2001). "The Power of Parallelism" (https://web.archive.org/web/20080131205427/http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=65878). *Computerworld*. Archived from the original (http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=65878) on January 31, 2008. Retrieved 2008-01-08.
68. Patterson and Hennessy, p. 749.
69. Minsky, Marvin (1986). *The Society of Mind* (https://archive.org/details/societyofmind00marv/page/17). New York: Simon & Schuster. pp. 17 (https://archive.org/details/societyofmind00marv/page/17). ISBN 978-0-671-60740-1.
70. Minsky, Marvin (1986). *The Society of Mind* (https://archive.org/details/societyofmind00marv/page/29). New York: Simon & Schuster. pp. 29 (https://archive.org/details/societyofmind00marv/page/29). ISBN 978-0-671-60740-1.
71. Blakeslee, Thomas (1996). *Beyond the Conscious Mind. Unlocking the Secrets of the Self*. pp. 6–7.
72. Gazzaniga, Michael; LeDoux, Joseph (1978). *The Integrated Mind*. pp. 132–161.
73. Gazzaniga, Michael (1985). *The Social Brain. Discovering the Networks of the Mind*. pp. 77–79.
74. Ornstein, Robert (1992). *Evolution of Consciousness: The Origins of the Way We Think*. p. 2.
75. Hilgard, Ernest (1977). *Divided consciousness: multiple controls in human thought and action*. New York: Wiley. ISBN 978-0-471-39602-4.
76. Hilgard, Ernest (1986). *Divided consciousness: multiple controls in human thought and action (expanded edition)*. New York: Wiley. ISBN 978-0-471-80572-4.
77. Kaku, Michio (2014). *The Future of the Mind*.
78. Ouspenskii, Pyotr (1992). "Chapter 3". *In Search of the Miraculous. Fragments of an Unknown Teaching*. pp. 72–83.
79. "Official Neurocluster Brain Model site" (http://neuroclusterbrain.com). Retrieved July 22, 2017.

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 (https://doi.org/10.1109%2FBIMNICS.2007.4610083).
- Sechin, A.; Parallel Computing in Photogrammetry. GIM International. #1, 2016, pp. 21–23.

External links

- Instructional videos on CAF in the Fortran Standard by John Reid (see Appendix B) (http://www.nd.edu/~dbalsara/Numerical-PDE-Course)
- Parallel computing (https://curlie.org/Computers/Parallel_Computing/) at Curlie

- Lawrence Livermore National Laboratory: Introduction to Parallel Computing (https://www.llnl.gov/computing/tutorials/parallel_comp/)
- Designing and Building Parallel Programs, by Ian Foster (http://www-unix.mcs.anl.gov/dbpp/)
- Internet Parallel Computing Archive (https://web.archive.org/web/20021012122919/http://wotug.ukc.ac.uk/parallel/)
- Parallel processing topic area at IEEE Distributed Computing Online (https://web.archive.org/web/20110928211245/http://dsonline.computer.org/portal/site/dsonline/index.jsp?pageID=dso_level1_home&path=dsonline%2Ftopics%2Fparallel&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 (https://web.archive.org/web/20081020052247/http://www.upcrc.illinois.edu/)
- Course in Parallel Programming at Columbia University (in collaboration with IBM T.J. Watson X10 project) (https://web.archive.org/web/20100122110043/http://ppppcourse.ning.com/)
- Parallel and distributed Gröbner bases computation in JAS (https://arxiv.org/PS_cache/arxiv/pdf/1008/1008.0011v1.pdf), see also Gröbner basis
- Course in Parallel Computing at University of Wisconsin-Madison (https://web.archive.org/web/20110826190010/http://sbel.wisc.edu/Courses/ME964/2011/index.htm)
- Berkeley Par Lab: progress in the parallel computing landscape (http://research.microsoft.com/en-us/collaboration/parlab/contents.aspx), Editors: David Patterson, Dennis Gannon, and Michael Wrinn, August 23, 2013
- The trouble with multicore (http://spectrum.ieee.org/computing/software/the-trouble-with-multicore), by David Patterson, posted 30 Jun 2010
- The Landscape of Parallel Computing Research: A View From Berkeley (https://web.archive.org/web/20080705232106/http://view.eecs.berkeley.edu/) (one too many dead link at this site)
- Introduction to Parallel Computing (https://computing.llnl.gov/tutorials/parallel_comp/)
- Coursera: Parallel Programming (https://www.coursera.org/learn/parprog1)