Parallel and Concurrent Computing

Loris Pauley
Gricelda Fawcett

First Edition, 2012

ISBN 978-81-323-1290-1

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Parallel Computing

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. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

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 program as a result of parallelization is observed 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

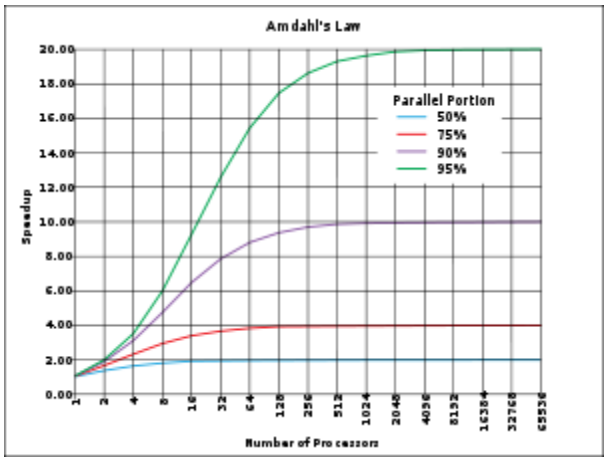
so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

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 computation-bounded 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



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.

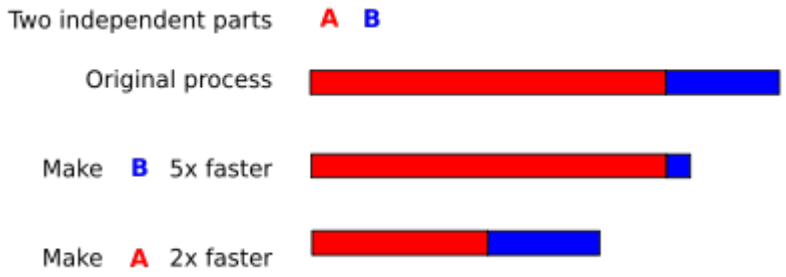
Optimally, the speed-up from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for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

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 α is the fraction of running time a sequential program spends on non-parallelizable parts, then

$$S = \frac{1}{\alpha}$$

is the maximum speed-up with parallelization of the program. If the sequential portion of a program accounts for 10% of the runtime, 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



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×).

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

Amdahl's law assumes a fixed problem size and that the running time of the sequential section of the program is independent of the number of processors, whereas Gustafson's law does not make these assumptions.

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 fragments. 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 statement producing a result used by the second statement. The second condition represents an anti-dependency, when the second statement (P_j) would overwrite a variable needed by the first expression (P_i). The third and final condition represents an output dependency: When two statements write to the same location, the final result must come from the logically last executed statement.

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

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

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 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, whether or not those instructions were using a single 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."

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. 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.

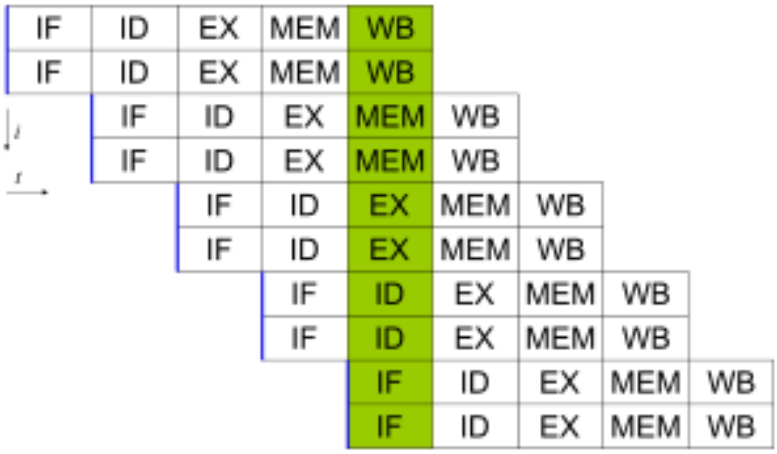
Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until recently (c. 2003–2004), with the advent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

Instruction-level parallelism

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

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.

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.



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.

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.

Data parallelism

Data parallelism is parallelism inherent in program loops, which focuses on distributing the data across different computing nodes to be processed in parallel. "Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure." Many scientific and engineering applications exhibit data parallelism.

A loop-carried dependency is the dependence of a loop iteration on the output of one or more previous iterations. Loop-carried dependencies prevent the parallelization of loops. For example, consider the following pseudocode that computes the first few Fibonacci numbers:

```
1:    PREV1 := 0
2:    PREV2 := 1
4:    do:
5:       CUR := PREV1 + PREV2
6:       PREV1 := PREV2
7:       PREV2 := CUR
8:    while (CUR < 10)
```

This loop cannot be parallelized because CUR depends on itself (PREV2) and PREV1, which are computed in each loop iteration. Since each iteration depends on the result of

the previous one, they cannot be performed in parallel. As the size of a problem gets bigger, the amount of data-parallelism available usually does as well.

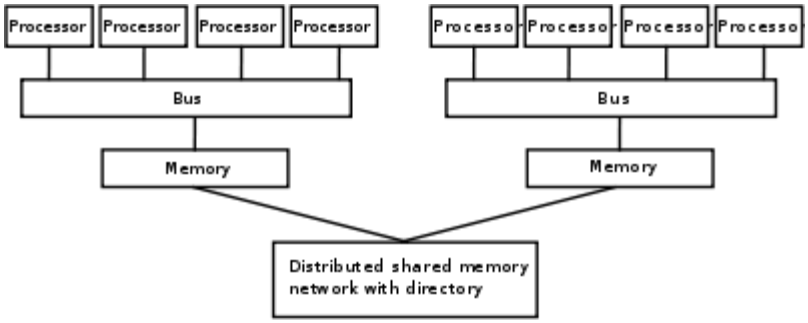
Task parallelism

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". 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.

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). 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.



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 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.

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

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); by contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. 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. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.

Symmetric multiprocessing

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

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

A Beowulf 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. 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. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. The vast majority of the TOP500 supercomputers are clusters.

Massive parallel processing

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. 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."

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.

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, grid computing typically deals only with embarrassingly parallel problems. Many grid 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 grid computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, grid 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/or 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. 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)

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. 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 CTM respectively. Other GPU programming languages are 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. 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.

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. (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

The Cray-1 is the most famous 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." They are closely related to Flynn's SIMD classification.

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

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. 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. 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. HMPP 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

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, and (for FPGAs) Mitrion-C.

Application checkpointing

The larger and more complex a computer is, the more that can go wrong and the shorter the mean time between failures. 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. For an application that may run for months, that is critical. Application checkpointing may be used to facilitate process migration.

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:

- 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

History

ILLIAC IV, "perhaps the most infamous of Supercomputers"

The origins of true (MIMD) parallelism go back to Federico Luigi, Conte Menabrea and his "Sketch of the Analytic Engine Invented by Charles Babbage". 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.

In April 1958, S. Gill (Ferranti) discussed parallel programming and the need for branching and waiting. 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. 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.

Chapter 2

Types of Parallelism

Bit-level parallelism

Bit-level parallelism is a form of parallel computing based on increasing processor word size. From the advent of very-large-scale integration (VLSI) computer chip fabrication technology in the 1970s until about 1986, advancements in computer architecture were done by increasing bit-level parallelism

Increasing the word size reduces the number of instructions the processor must execute in order to perform an operation on variables whose sizes are greater than the length of the word. (For example, consider a case where an 8-bit processor must add two 16-bit integers. The processor must first add the 8 lower-order bits from each integer, then add the 8 higher-order bits, requiring two instructions to complete a single operation. A 16-bit processor would be able to complete the operation with 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. Only recently, with the advent of x86-64 architectures, have 64-bit processors become commonplace.

On 32-bit processors, external data bus width continues to increase. For example, DDR1 SDRAM transfers 128 bits per clock cycle. DDR2 SDRAM transfers a minimum of 256 bits per burst.

About 55% of all CPUs sold in the world are 8-bit microcontrollers. Less than 10% of all the CPUs sold in the world are 32-bit or more.

Instruction-level parallelism

Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. Consider the following program:

```
1. e = a + b
2. f = c + d
3. g = 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 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, but they have not been actively researched since the 1980s.

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 360 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.

Data parallelism

Data parallelism (also known as **loop-level 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:

```
define foo
    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.

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 takes place usually to pass 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 runtime 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.

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.

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).

Chapter 3

Distributing Computing

Distributed computing is a field of computer science that studies distributed systems. A **distributed system** consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a **distributed program**, and **distributed programming** is the process of writing such programs.

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 computer.

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. 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, the following defining properties are commonly used:

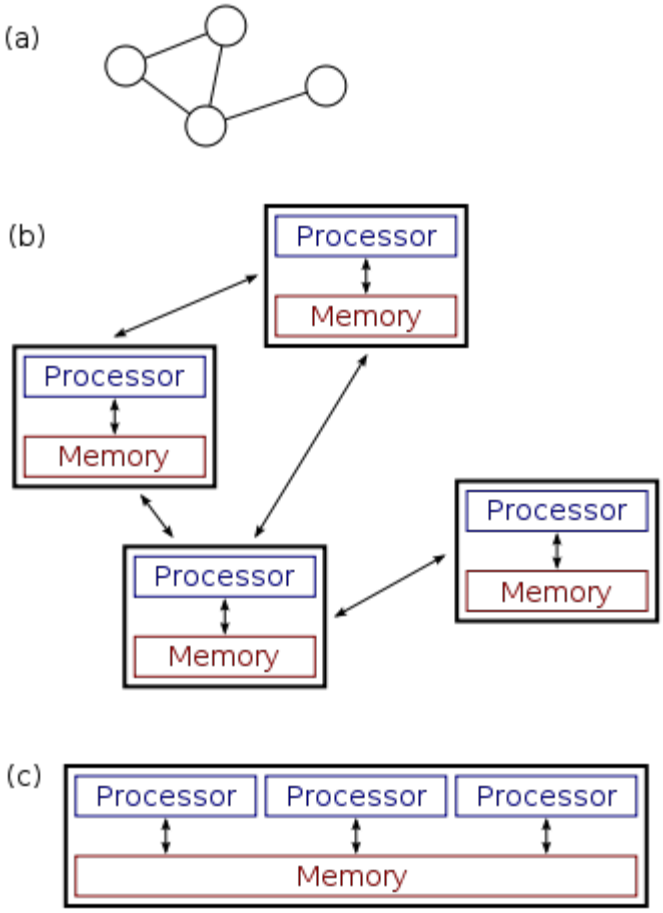
- There are several autonomous computational entities, each of which has its own local memory.
- The entities communicate with each other by message passing.

A distributed system may have a common goal, such as solving a large computational problem. 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.

Other typical properties of distributed systems include the following:

- The system has to tolerate failures in individual computers.

- 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.
- Each computer has only a limited, incomplete view of the system. Each computer may know only one part of the input.



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

Parallel or distributed computing?

The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterised both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly-coupled form of distributed computing, 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 have access to a shared memory. Shared memory can be used to exchange information between processors.
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

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 graph in which each node (vertex) is a computer and each edge (line between two 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

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

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, and it is probably the earliest example of a large-scale distributed application. In addition to ARPANET and its successor 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

There are two main reasons for using distributed systems and distributed computing. First, the very nature of the application may *require* the use of a communication network that connects several computers. For example, data is produced in one physical location and it is needed in another location.

Second, 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 be more reliable 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.

Examples of distributed systems and applications of distributed computing include the following:

- 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.
  - Distributed rendering in computer graphics.

Theoretical foundations

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 focusses 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. 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.

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. 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 colouring 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 neighbours 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 colour 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 colouring 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). 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

A centralised algorithm is efficient if it does not require much time (number of computational steps) or space (amount of memory). These complexity measures give rise to complexity classes such as P (decision problems solvable in polynomial time) and PSPACE (decision problems solvable in polynomial space).

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. 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. 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.

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.

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.

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, Byzantine fault tolerance, and self-stabilisation.

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.
- Logical clocks provide a causal happened-before ordering of events.
- Clock synchronization algorithms provide globally consistent physical time stamps.

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, 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.

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.
- Tightly 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.

Computer cluster

The Silicon Graphics Cluster-SGI; an example of a cluster computer

A **computer cluster** is a group of linked computers, working together closely thus in many respects forming a single computer. The components of a cluster are commonly, but not always, connected to each other through fast local area networks. Clusters are usually deployed to improve performance and/or availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability.

Cluster categorizations

High-availability (HA) clusters

High-availability clusters (also known as Failover Clusters) are implemented primarily for the purpose of improving the availability of services that the cluster provides. They operate by having redundant nodes, which are then used to provide service when system components fail. The most common size for an HA cluster is two nodes, which is the minimum requirement to provide redundancy. 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. The LanderCluster from Lander Software can run on Windows, Linux, and UNIX platforms.

Load-balancing clusters

Load-balancing is when multiple computers are linked together to share computational workload or function as a single virtual computer. Logically, from the user side, they are multiple machines, but function as a single virtual machine. Requests initiated from the user are managed by, and distributed among, all the standalone computers to form a cluster. This results in balanced computational work among different machines, improving the performance of the cluster systems.

Compute clusters

Often clusters are used primarily for computational purposes, rather than handling IO-oriented operations such as web service or databases. For instance, a cluster might support computational simulations of weather or vehicle crashes. The primary distinction within compute clusters is how tightly-coupled the individual nodes are. For instance, a single compute job may require frequent communication among nodes - this implies that the cluster shares a dedicated network, is densely located, and probably has homogenous nodes. This cluster design is usually referred to as Beowulf Cluster. The other extreme is where a compute job uses one or few nodes, and needs little or no inter-node communication. This latter category is sometimes called "Grid" computing. Tightly-coupled compute clusters are designed for work that might traditionally have been called "supercomputing". Middleware such as MPI (Message Passing Interface) or PVM (Parallel Virtual Machine) permits compute clustering programs to be portable to a wide variety of clusters.

Grid computing

Grids are usually computer clusters, but more focused on throughput like a computing utility rather than running fewer, tightly-coupled jobs. Often, grids will incorporate heterogeneous collections of computers, possibly distributed geographically, sometimes administered by unrelated organizations.

Grid computing is optimized for workloads which consist of many independent jobs or packets of work, which do not have to share data between the jobs during the computation process. Grids serve to manage the allocation of jobs to computers which will perform the work independently of the rest of the grid cluster. Resources such as storage may be shared by all the nodes, but intermediate results of one job do not affect other jobs in progress on other nodes of the grid.

An example of a very large grid is the Folding@home project. It is analyzing data that is used by researchers to find cures for diseases such as Alzheimer's and cancer. Another

large project is the SETI@home project, which may be the largest distributed grid in existence. It uses approximately three million home computers all over the world to analyze data from the Arecibo Observatory radiotelescope, searching for evidence of extraterrestrial intelligence. In both of these cases, there is no inter-node communication or shared storage. Individual nodes connect to a main, central location to retrieve a small processing job. They then perform the computation and return the result to the central server. In the case of the @home projects, the software is generally run when the computer is otherwise idle. UC Berkley has developed an open source application BOINC to allow individual users to contribute to the above and other projects such as lhc@home (Large Hadron Collider) from a single manager which can then be set to allocate a percentage of idle time to each of the projects a node is signed up for. The Software can be downloaded and a project list can be found here BOINC

The grid setup means that the nodes can take however many jobs they are able to process in one session and then return the results and acquire a new job from a central project server.

Implementations

The TOP500 organization's semiannual list of the 500 fastest computers usually includes many clusters. TOP500 is a collaboration between the University of Mannheim, the University of Tennessee, and the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory. As of November 17, 2009, the top supercomputer is the Department of Energy's Jaguar system with performance of 1759 TFlops measured with the High-Performance LINPACK benchmark.

Clustering can provide significant performance benefits versus price. The System X supercomputer at Virginia Tech, the 28th most powerful supercomputer on Earth as of June 2006, is a 12.25 TFlops computer cluster of 1100 Apple XServe G5 2.3 GHz dual-processor machines (4 GB RAM, 80 GB SATA HD) running Mac OS X and using InfiniBand interconnect. The cluster initially consisted of Power Mac G5s; the rack-mountable XServes are denser than desktop Macs, reducing the aggregate size of the cluster. The total cost of the previous Power Mac system was $5.2 million, a tenth of the cost of slower mainframe computer supercomputers. (The Power Mac G5s were sold off.)

The central concept of a Beowulf cluster is the use of commercial off-the-shelf (COTS) computers to produce a cost-effective alternative to a traditional supercomputer. One project that took this to an extreme was the Stone Soupercomputer.

However it is worth noting that Flops (floating point operations per second), aren't always the best metric for supercomputer speed. Clusters can have very high Flops, but they cannot access all data in the cluster as a whole at once. Therefore clusters are excellent for parallel computation, but much poorer than traditional supercomputers at non-parallel computation.

JavaSpaces is a specification from Sun Microsystems that enables clustering computers via a distributed shared memory.

Consumer game consoles

Due to the increasing computing power of each generation of game consoles, a novel use has emerged where they are repurposed into HPC clusters. Some examples of game console clusters are Sony PlayStation clusters and Microsoft Xbox clusters. It has been suggested on a news website that countries which are restricted from buying supercomputing technologies may be obtaining game systems to build computer clusters for military use.

History

The history of cluster computing is best captured by a footnote in Greg Pfister's *In Search of Clusters*: “Virtually every press release from DEC mentioning clusters says ‘DEC, who invented clusters…’. IBM did not invent them either. *Customers* invented clusters, as soon as they could not fit all their work on one computer, or needed a backup. The date of the first is unknown, but it would be surprising if it was not in the 1960s, or even late 1950s.”

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. Amdahl's Law describes mathematically the speedup one can expect from parallelizing any given otherwise serially performed task on a parallel architecture. Here we, defined the engineering basis for both multiprocessor computing and cluster computing, where the primary differentiator is whether or not the interprocessor communications are supported "inside" the computer (on for example a customized internal communications bus or network) or "outside" the computer on a *commodity* network.

Consequently the history of early computer clusters is more or less directly tied into the history of early networks, as one of the primary motivation for the development of a network was to link computing resources, creating a de facto computer cluster. Packet switching networks were conceptually invented by the RAND corporation in 1962. Using the concept of a packet switched network, the ARPANET project succeeded in creating in 1969 what was arguably the world's first commodity-network based computer cluster by linking four different computer centers (each of which was something of a "cluster" in its own right, but probably not a *commodity* cluster). The ARPANET project grew into the Internet—which can be thought of as "the mother of all computer clusters" (as the union of nearly all of the computer resources, including clusters, that happen to be connected). It also established the paradigm in use by *all* computer clusters in the world today—the use of packet-switched networks to perform interprocessor communications between processor (sets) located in otherwise disconnected frames.

The development of customer-built and research clusters proceeded hand in hand with that of both networks and the Unix operating system from the early 1970s, as both TCP/IP and the Xerox PARC project created and formalized protocols for network-based communications. The Hydra operating system was built for a cluster of DEC PDP-11 minicomputers called C.mmp at Carnegie Mellon University in 1971. However, it was not until circa 1983 that the protocols and tools for *easily* doing remote job distribution and file sharing were defined (largely within the context of BSD Unix, as implemented by Sun Microsystems) and hence became generally available commercially, along with a shared filesystem.

The *first* commercial clustering product was ARCnet, developed by Datapoint in 1977. ARCnet was not a commercial success and clustering per se did not really take off until DEC 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. VAXcluster, now VMScluster, is still available on OpenVMS systems from HP running on Alpha and Itanium systems.

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

No history of commodity computer clusters would be complete without noting the pivotal role played by the development of Parallel Virtual Machine (PVM) software in 1989. This open source software based on TCP/IP communications enabled the *instant* creation of a virtual supercomputer—a high performance compute cluster—made out of any TCP/IP connected systems. Free form heterogeneous clusters built on top of this model rapidly achieved total throughput in FLOPS that greatly exceeded that available even with the most expensive "big iron" supercomputers. PVM and the advent of inexpensive networked PCs led, in 1993, to a NASA project to build supercomputers out of commodity clusters. In 1995 the invention of the "beowulf"-style cluster—a compute cluster built on top of a commodity network for the specific purpose of "being a supercomputer" capable of performing tightly coupled parallel HPC computations. This in turn spurred the independent development of Grid computing as a named entity, although Grid-style clustering had been around at least as long as the Unix operating system and the Arpanet, whether or not it, or the clusters that used it, were named.

Technologies

MPI is a widely-available communications library that enables parallel programs to be written in C, Fortran, Python, OCaml, and many other programming languages.

The GNU/Linux world supports various cluster software; for application clustering, there is Beowulf, 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. NCSA's recently installed Lincoln is a cluster of 450 Dell PowerEdge 1855 blade servers running Windows Compute Cluster Server 2003. This cluster debuted at #130 on the Top500 list in June 2006.

gridMathematica provides distributed computations over clusters including data analysis, computer algebra and 3D visualization. It can make use of other technologies such as Altair PBS Professional, Microsoft Windows Compute Cluster Server, Platform LSF and Sun Grid Engine.

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

Another example of consumer game products being added to high-performance computing is the Nvidia Tesla Personal Supercomputer workstation, which gets its processing power by harnessing the power of multiple graphics accelerator processor chips.

Algorithmic Skeletons are a high-level parallel programming model for parallel and distributed computing which take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones.

Massive parallel processing

Massive parallel processing (**MPP**) is a term used in computer architecture to refer to a computer system with many independent arithmetic units or entire microprocessors, that run in parallel. The term *massive* connotes hundreds if not thousands of such units. Early examples of such a system are the Distributed Array Processor, the Goodyear MPP, the Connection Machine, and the Ultracomputer.

Some years ago many of the most powerful supercomputers were *MPP* systems.

In this class of computing, all of the processing elements are connected together to be one very large computer. This is in contrast to distributed computing where massive numbers of separate computers are used to solve a single problem.

The earliest massively parallel processing systems all used serial computers as individual processing elements, in order to achieve the maximum number of independent units for a given size and cost.

Through advances in Moore's Law, single-chip implementations of massively parallel processor arrays are becoming cost effective, and finding particular application in high performance embedded systems applications such as video compression. Examples include chips from Ambric, Coherent Logix, picoChip, and Tilera.

Grid computing

Grid computing is a term referring to the combination of computer resources from multiple administrative domains to reach a common goal. The **Grid** can be thought of as a distributed system with non-interactive workloads that involve a large number of files. What distinguishes grid computing from conventional high performance computing systems such as cluster computing is that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed. Although a grid can be dedicated to a specialized application, it is more common that a single grid will be used for a variety of different purposes. Grids are often constructed with the aid of general-purpose grid software libraries known as middleware.

Grid size can vary by a considerable amount. Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform very large tasks. Furthermore, “Distributed” or “grid” computing in general is a special type of parallel computing that relies on complete computers (with onboard CPUs, storage, power supplies, network interfaces, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus.

Overview

Grid computing combines computer resources from multiple administrative domains to reach common goal. Grid computing (or the use of a computational grid) simultaneously applies the resources of many computers in a network to tackle a single problem, usually to solve a scientific or technical problem that requires a great number of computer processing cycles or access to large amounts of data. When many are brought together in this collaborative effort, they are known as Virtual Organizations (VOs). These VOs may be formed to solve a single task and may then disappear just as quickly.

One of the main strategies of grid computing is to use middleware to divide and apportion pieces of a program among several computers, sometimes up to many thousands. Grid computing involves computation in a distributed fashion, which may also involve the aggregation of large-scale cluster computing-based systems.

The size of a grid may vary from small—confined to a network of computer workstations within a corporation, for example—to large, public collaborations across many companies and networks. "The notion of a confined grid may also be known as an intra-nodes cooperation whilst the notion of a larger, wider grid may thus refer to an inter-nodes cooperation".

Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform very large tasks. This technology has been applied to computationally intensive scientific, mathematical, and academic problems through volunteer computing, and it is used in commercial enterprises for such diverse applications as drug discovery, economic forecasting, seismic analysis, and back office data processing in support for e-commerce and Web services.

Comparison of grids and conventional supercomputers

“Distributed” or “grid” computing in general is a special type of parallel computing that relies on complete computers (with onboard CPUs, storage, power supplies, network interfaces, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus.

The primary advantage of distributed computing is that each node can be purchased as commodity hardware, which, when combined, can produce a similar computing resource as multiprocessor supercomputer, but at a lower cost. This is due to the economies of scale of producing commodity hardware, compared to the lower efficiency of designing and constructing a small number of custom supercomputers. The primary performance disadvantage is that the various processors and local storage areas do not have high-speed connections. This arrangement is thus well-suited to applications in which multiple parallel computations can take place independently, without the need to communicate intermediate results between processors. The high-end scalability of geographically dispersed grids is generally favorable, due to the low need for connectivity between nodes relative to the capacity of the public Internet.

There are also some differences in programming and deployment. It can be costly and difficult to write programs that can run in the environment of a supercomputer, which may have a custom operating system, or require the program to address concurrency issues. If a problem can be adequately parallelized, a “thin” layer of “grid” infrastructure can allow conventional, standalone programs, given a different part of the same problem, to run on multiple machines. This makes it possible to write and debug on a single

conventional machine, and eliminates complications due to multiple instances of the same program running in the same shared memory and storage space at the same time.

Design considerations and variations

One feature of distributed grids is that they can be formed from computing resources belonging to multiple individuals or organizations (known as multiple administrative domains). This can facilitate commercial transactions, as in utility computing, or make it easier to assemble volunteer computing networks.

One disadvantage of this feature is that the computers which are actually performing the calculations might not be entirely trustworthy. The designers of the system must thus introduce measures to prevent malfunctions or malicious participants from producing false, misleading, or erroneous results, and from using the system as an attack vector. This often involves assigning work randomly to different nodes (presumably with different owners) and checking that at least two different nodes report the same answer for a given work unit. Discrepancies would identify malfunctioning and malicious nodes.

Due to the lack of central control over the hardware, there is no way to guarantee that nodes will not drop out of the network at random times. Some nodes (like laptops or dialup Internet customers) may also be available for computation but not network communications for unpredictable periods. These variations can be accommodated by assigning large work units (thus reducing the need for continuous network connectivity) and reassigning work units when a given node fails to report its results in expected time.

The impacts of trust and availability on performance and development difficulty can influence the choice of whether to deploy onto a dedicated computer cluster, to idle machines internal to the developing organization, or to an open external network of volunteers or contractors. In many cases, the participating nodes must trust the central system not to abuse the access that is being granted, by interfering with the operation of other programs, mangling stored information, transmitting private data, or creating new security holes. Other systems employ measures to reduce the amount of trust “client” nodes must place in the central system such as placing applications in virtual machines.

Public systems or those crossing administrative domains (including different departments in the same organization) often result in the need to run on heterogeneous systems, using different operating systems and hardware architectures. With many languages, there is a trade off between investment in software development and the number of platforms that can be supported (and thus the size of the resulting network). Cross-platform languages can reduce the need to make this trade off, though potentially at the expense of high performance on any given node (due to run-time interpretation or lack of optimization for the particular platform).

Various middleware projects have created generic infrastructure to allow diverse scientific and commercial projects to harness a particular associated grid or for the

purpose of setting up new grids. BOINC is a common one for various academic projects seeking public volunteers; more are listed at the end of the article.

In fact, the middleware can be seen as a layer between the hardware and the software. On top of the middleware, a number of technical areas have to be considered, and these may or may not be middleware independent. Example areas include SLA management, Trust and Security, Virtual organization management, License Management, Portals and Data Management. These technical areas may be taken care of in a commercial solution, though the cutting edge of each area is often found within specific research projects examining the field.

Market segmentation of the grid computing market

According to IT-Tude.com, for the segmentation of the grid computing market, two perspectives need to be considered: the provider side and the user side:

The provider side

The overall grid market comprises several specific markets. These are the grid middleware market, the market for grid-enabled applications, the utility computing market, and the software-as-a-service (SaaS) market.

Grid middleware is a specific software product, which enables the sharing of heterogeneous resources, and Virtual Organizations. It is installed and integrated into the existing infrastructure of the involved company or companies, and provides a special layer placed among the heterogeneous infrastructure and the specific user applications. Major grid middlewares are Globus Toolkit, gLite, and UNICORE.

Utility computing is referred to as the provision of grid computing and applications as service either as an open grid utility or as a hosting solution for one organization or a VO. Major players in the utility computing market are Sun Microsystems, IBM, and HP.

Grid-enabled applications are specific software applications that can utilize grid infrastructure. This is made possible by the use of grid middleware, as pointed out above.

Software as a service (SaaS) is “software that is owned, delivered and managed remotely by one or more providers.” (Gartner 2007) Additionally, SaaS applications are based on a single set of common code and data definitions. They are consumed in a one-to-many model, and SaaS uses a Pay As You Go (PAYG) model or a subscription model that is based on usage. Providers of SaaS do not necessarily own the computing resources themselves, which are required to run their SaaS. Therefore, SaaS providers may draw upon the utility computing market. The utility computing market provides computing resources for SaaS providers.

The user side

For companies on the demand or user side of the grid computing market, the different segments have significant implications for their IT deployment strategy. The IT deployment strategy as well as the type of IT investments made are relevant aspects for potential grid users and play an important role for grid adoption.

CPU scavenging

CPU-scavenging, **cycle-scavenging**, **cycle stealing**, or **shared computing** creates a "grid" from the unused resources in a network of participants (whether worldwide or internal to an organization). Typically this technique uses desktop computer instruction cycles that would otherwise be wasted at night, during lunch, or even in the scattered seconds throughout the day when the computer is waiting for user input or slow devices.

Volunteer computing projects use the CPU scavenging model almost exclusively.

In practice, participating computers also donate some supporting amount of disk storage space, RAM, and network bandwidth, in addition to raw CPU power. Heat produced by CPU power in rooms with many computers can be used for fine heating premises. Since nodes are likely to go "offline" from time to time, as their owners use their resources for their primary purpose, this model must be designed to handle such contingencies.

History

The term *grid computing* originated in the early 1990s as a metaphor for making computer power as easy to access as an electric power grid in Ian Foster's and Carl Kesselman's seminal work, "The Grid: Blueprint for a new computing infrastructure" (1999).

CPU scavenging and volunteer computing were popularized beginning in 1997 by distributed.net and later in 1999 by SETI@home to harness the power of networked PCs worldwide, in order to solve CPU-intensive research problems.

The ideas of the grid (including those from distributed computing, object-oriented programming, and Web services) were brought together by Ian Foster, Carl Kesselman, and Steve Tuecke, widely regarded as the "fathers of the grid". They led the effort to create the Globus Toolkit incorporating not just computation management but also storage management, security provisioning, data movement, monitoring, and a toolkit for developing additional services based on the same infrastructure, including agreement negotiation, notification mechanisms, trigger services, and information aggregation. While the Globus Toolkit remains the de facto standard for building grid solutions, a number of other tools have been built that answer some subset of services needed to create an enterprise or global grid.

In 2007 the term cloud computing came into popularity, which is conceptually similar to the canonical Foster definition of grid computing (in terms of computing resources being consumed as electricity is from the power grid). Indeed, grid computing is often (but not always) associated with the delivery of cloud computing systems as exemplified by the AppLogic system from 3tera.

Fastest virtual supercomputers

- BOINC – 5.128PFLOPS as of Apr 24th 2010.
- Folding@Home – 5 PFLOPS, as of March 17, 2009
- As of April 2010, MilkyWay@Home computes at over 1.6 PFLOPS, with a large amount of this work coming from GPUs.
- As of April 2010, SETI@Home computes data averages more than 730 TFLOPS.
- As of April 2010, Einstein@Home is crunching more than 210 TFLOPS.
- As of April 2010, GIMPS is sustaining 44 TFLOPS.

Current projects and applications

Grids computing offer a way to solve Grand Challenge problems such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling. Grids offer a way of using the information technology resources optimally inside an organization. They also provide a means for offering information technology as a utility for commercial and noncommercial clients, with those clients paying only for what they use, as with electricity or water.

Grid computing is being applied by the National Science Foundation's National Technology Grid, NASA's Information Power Grid, Pratt & Whitney, Bristol-Myers Squibb Co., and American Express.

One of the most famous cycle-scavenging networks is SETI@home, which was using more than 3 million computers to achieve 23.37 sustained teraflops (979 lifetime teraflops) as of September 2001.

As of August 2009 Folding@home achieves more than 4 petaflops on over 350,000 machines.

The European Union has been a major proponent of grid computing. Many projects have been funded through the framework programme of the European Commission. Many of the projects are highlighted below, but two deserve special mention: BEinGRID and Enabling Grids for E-sciencE.

BEinGRID (Business Experiments in Grid) is a research project partly funded by the European commission as an Integrated Project under the Sixth Framework Programme (FP6) sponsorship program. Started in June 1, 2006, the project will run 42 months, until November 2009. The project is coordinated by Atos Origin. According to the project fact

sheet, their mission is “to establish effective routes to foster the adoption of grid computing across the EU and to stimulate research into innovative business models using Grid technologies”. To extract best practice and common themes from the experimental implementations, two groups of consultants are analyzing a series of pilots, one technical, one business. The results of these cross analyzes are provided by the website IT-Tude.com. The project is significant not only for its long duration, but also for its budget, which at 24.8 million Euros, is the largest of any FP6 integrated project. Of this, 15.7 million is provided by the European commission and the remainder by its 98 contributing partner companies.

The Enabling Grids for E-sciencE project, which is based in the European Union and includes sites in Asia and the United States, is a follow-up project to the European DataGrid (EDG) and is arguably the largest computing grid on the planet. This, along with the LHC Computing Grid (LCG), has been developed to support the experiments using the CERN Large Hadron Collider. The LCG project is driven by CERN's need to handle huge amounts of data, where storage rates of several gigabytes per second (10 petabytes per year) are required. A list of active sites participating within LCG can be found online as can real time monitoring of the EGEE infrastructure. The relevant software and documentation is also publicly accessible. There is speculation that dedicated fiber optic links, such as those installed by CERN to address the LCG's data-intensive needs, may one day be available to home users thereby providing internet services at speeds up to 10,000 times faster than a traditional broadband connection.

Another well-known project is distributed.net, which was started in 1997 and has run a number of successful projects in its history.

The NASA Advanced Supercomputing facility (NAS) has run genetic algorithms using the Condor cycle scavenger running on about 350 Sun and SGI workstations.

Until April 27, 2007, United Devices operated the United Devices Cancer Research Project based on its Grid MP product, which cycle-scavenges on volunteer PCs connected to the Internet. As of June 2005, the Grid MP ran on about 3.1 million machines.

Another well-known project is the World Community Grid. The World Community Grid's mission is to create the largest public computing grid that benefits humanity. This work is built on belief that technological innovation combined with visionary scientific research and large-scale volunteerism can change our world for the better. IBM Corporation has donated the hardware, software, technical services, and expertise to build the infrastructure for World Community Grid and provides free hosting, maintenance, and support.

Definitions

Today there are many definitions of *grid computing*:

- In his article “What is the Grid? A Three Point Checklist”, Ian Foster lists these primary attributes:
    - Computing resources are not administered centrally.
    - Open standards are used.
    - Nontrivial quality of service is achieved.

- Plaszczak/Wellner define grid technology as "the technology that enables resource virtualization, on-demand provisioning, and service (resource) sharing between organizations."
- IBM defines grid computing as “the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across ‘multiple’ administrative domains based on their (resources) availability, capacity, performance, cost and users' quality-of-service requirements”.
- An earlier example of the notion of computing as utility was in 1965 by MIT's Fernando Corbató. Corbató and the other designers of the Multics operating system envisioned a computer facility operating “like a power company or water company”.
- Buyya/Venugopal define grid as "a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements".
- CERN, one of the largest users of grid technology, talk of **The Grid**: “a service for sharing computer power and data storage capacity over the Internet.”

Grids can be categorized with a three stage model of departmental grids, enterprise grids and global grids. These correspond to a firm initially utilising resources within a single group i.e. an engineering department connecting desktop machines, clusters and equipment. This progresses to enterprise grids where nontechnical staff's computing resources can be used for cycle-stealing and storage. A global grid is a connection of enterprise and departmental grids that can be used in a commercial or collaborative manner.

Chapter 4

Models and Techniques in Parallel Computing

Parallel programming model

A **parallel programming model** is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. It encloses the areas of applications, programming languages, compilers, libraries, communications systems, and parallel I/O. Due to the difficulties in automatic parallelization today, people have to choose a proper parallel programming model or a form of mixture of them to develop their parallel applications on a particular platform.

Parallel models are implemented in several ways: as libraries invoked from traditional sequential languages, as language extensions, or complete new execution models. They are also roughly categorized for two kinds of systems: shared-memory system and distributed-memory system, though the lines between them are largely blurred nowadays.

A programming model is usually judged by its expressibility and simplicity, which are by all means conflicting factors. The ultimate goal is to improve productivity of programming.

Example parallel programming models

Models

- Algorithmic Skeletons
- Components
- Distributed Objects
- Remote Method Invocation
- Workflows

Libraries

- POSIX Threads

- MPI
- Global Arrays
- SHMEM
- PVM
- TBB
- KAAPI

Languages

- Ada
- Ateji PX
- C*
- Cilk
- Charm++
- Partitioned global address space languages:
  - Co-array Fortran,
  - Unified Parallel C,
  - Titanium
- High Performance Fortran
- Haskell
- Occam
- Event-driven programming & Hardware Description Languages:
  - Verilog
  - VHDL
  - SystemC
  - ParC Parallel C++ through language extensions
- Ease
- Erlang
- Linda coordination language
- Oz
- CUDA
- OpenCL
- Jacket
- NESL
- Scala

Unsorted

- OpenMP
- Global Arrays
- Intel Ct
- Pervasive DataRush
- ProActive
- Parallel Random Access Machine
- Stream processing
- Structural Object Programming Model (SOPM)

- Pipelining
- ZPL

Other research-level models are:

- Cray's Chapel
- Sun’s Fortress
- IBM’s X10

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. A parallelizing compiler tries to split up a loop so that its iterations can be executed on separate processors 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

The Fortran code below 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
```

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.

Difficulties

Automatic parallelization by compilers or tools is very difficult due to the following reasons:

- dependence analysis is hard for code using indirect addressing, pointers, recursion, and 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.

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 for shared memory systems.
- Build an interactive system between programmers and parallelizing tools/compilers. Notable examples are Vector Fabrics' vfAnalyst, SUIF Explorer (The Stanford University Intermediate Format compiler), the Polaris compiler, and ParaWise (formally CAPTools).
- Hardware-supported speculative multithreading.

Historical parallelizing compilers

Most research compilers for automatic parallelization consider Fortran programs, because Fortran makes stronger guarantees about aliasing than languages such as C. Typical examples are:

- Rice Fortran D compiler
- Vienna Fortran compiler
- Paradigm compiler
- Polaris compiler
- SUIF compiler

Application checkpointing

Checkpointing is a technique for inserting fault tolerance into computing systems. It basically consists of storing a snapshot of the current application state, and later on, use it for restarting the execution in case of failure.

Technique properties

There are many different points of view and techniques for achieving application checkpointing. Depending on the specific implementation, a tool can be classified as having several properties:

- *Amount of state saved*: This property refers to the abstraction level used by the technique to analyze an application. It can range from seeing each application as a

black box, hence storing all application data, to selecting specific relevant cores of data in order to achieve a more efficient and portable operation.

- *Automatization level*: Depending on the effort needed to achieve fault tolerance through the use of a specific checkpointing solution.

- *Portability*: Whether or not the saved state can be used on different machines to restart the application.

- *System architecture*: How is the checkpointing technique implemented: inside a library, by the compiler or at operating system level.

Each design decision made affects the properties and efficiency of the final product. For instance, deciding to store the entire application state will allow for a more straightforward implementation, since no analysis of the application will be needed, but it will deny the portability of the generated state files, due to a number of non-portable structures (such as application stack or heap) being stored along with application data.

Use in distributed shared memory systems

In distributed shared memory, checkpointing is a technique that helps tolerate the errors leading to losing the effect of work of long-running applications. The main property which should be induced by checkpointing techniques in such systems is in preserving system consistency in case of failure. There are two main approaches to checkpointing in such systems: coordinated checkpointing, in which all cooperating processes work together to establish coherent checkpoint; and communication induced (called also dependency induced) independent checkpointing.

It must be stressed that simply forcing processes to checkpoint their state at fixed time intervals is not sufficient to ensure global consistency. Even if we postulate the existence of global clock, the checkpoints made by different processes still may not form a consistent state. The need for establishing a consistent state may force other process to roll back to their checkpoints, which in turn may cause other processes to roll back to even earlier checkpoints, which in the most extreme case may mean that the only consistent state found is the initial state (the so called *domino effect*).

In the coordinated checkpointing approach, processes must ensure that their checkpoints are consistent. This is usually achieved by some kind of two-phase commit protocol algorithm. In communication induced checkpointing, each process checkpoints its own state independently whenever this state is exposed to other processes (that is, for example whenever a remote process reads the page written to by the local process).

The system state may be saved either locally, in stable storage, or in a distant node's memory.

Practical implementations for Linux/Unix

A number of practical checkpointing packages have been developed for the Linux/Unix family of operating systems. These checkpointing packages may be divided into two classes, those which operate in user space, examples of which include the checkpointing package used by Condor and the portable checkpointing library developed by The University of Tennessee. User space checkpointing packages are highly portable and can typically be compiled and run on any modern Unix (e.g. Linux, FreeBSD, OpenBSD, Darwin etc). In contrast, kernel based checkpointing packages such as Chpox and the checkpointing algorithms developed for the MOSIX cluster computing environment tend to be highly operating system dependent. Most kernel based checkpointing packages developed to date run under either the 2.4 or 2.6 subfamilies of the Linux kernel on i686 architectures.

Cryopid

Modern checkpointing packages such as **Cryopid** are capable of checkpointing a ***process pod***, that is a parent process and all its associated children, and of dealing with file system abstractions such as sockets and pipes (FIFO's) in addition to regular files. In the case of Cryopid, there is also provision to roll all dynamic libraries, open files, sockets and FIFO's associated with the process into the checkpoint. This is very useful when the checkpointed process is to be restarted in a heterogeneous environment (e.g. the machine on which the checkpoint is restarted has libraries and file system which differ from the host on which the process was checkpointed). Note that Cryopid seems to be unmaintained and does not compile on recent (2.6.22) Linux x86-64 kernels.

Note Cryopid is still maintained and is now available via the **SourceForge** project **Cryopid2**. This version of Cryopid will compile on all Linux kernels up to 2.6.27 for 32-bit kernels. Work is in hand to get Cryopid2 working on 64-bit kernels. The cryopid2 package extends Benard Blackhams original Cryopid package in a number of significant ways. For example, it allows the state of Linux real time signals to be preserved when a checkpoint is taken and also is capable of inter-operating with ssh via a **portal daemon** in order to implement **full** process migration (and of any associated pod processes) between Linux hosts. Cryopid2 also has the capability to roll up its environment (e.g. the bodies of open files) into the checkpoints it produces. This facilitates the migration of processes onto *foreign* hosts which present an arbitrary file system environment to an inbound migrating process. **Pipes** are also preserved in a similar manner: their contents are sucked into the migrating process prior to migration (so they form part of the checkopoint) and spat out into the kernel of the new host when execution is resumed. Cryopid2 is inter-operable with the P3 Organic computing environment which uses its services for both persistence and process migration.

DMTCP

DMTCP (Distributed MultiThreaded Checkpointing) is a tool to transparently checkpointing the state of an arbitrary group of programs spread across many machines

and connected by sockets. It does not modify the user's program nor the operating system.

Among the applications supported by DMTCP are Open MPI, MATLAB, Python, Perl, and many programming languages and shell scripting languages. With the use of TightVNC, it can also checkpoint and restart X Window applications, as long as they do not use extensions (e.g.: no OpenGL, no video). Among the Linux features supported by DMTCP are open file descriptors, pipes, sockets, signal handlers, process id and thread id virtualization (ensure old pids and tids continue to work upon restart), ptys, fifos, process group ids, session ids, terminal attributes, and mmap/mprotect (including mmap-based shared memory).

DMTCP is also the basis for **URDB**, the Universal Reversible Debugger. URDB is still experimental. Nevertheless, it currently adds reversibility to gdb, MATLAB, Python (pdb), and Perl (perl -d). It also supports reverse expression watchpoints, a form of temporal search within a process lifetime.

OpenVz

OpenVZ kernel has an ability to checkpoint and restart a virtual private server (VPS), i.e. a set of processes and all the data structures associated with those processes (opened files, sockets, IPC objects, network connections, etc.). The primary use of checkpointing is "live migration", a move of a VPS from one physical server to another without a need to shut down and restart it. OpenVZ supports checkpointing on x86, x86-64 and IA-64 architectures.

Chapter 5

Introduction to Concurrent Computing

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that *may* be executed in parallel. Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network. The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes. A number of different methods can be used to implement concurrent programs, such as implementing each computational process as an operating system process, or implementing the computational processes as a set of threads within a single operating system process.

Pioneers in the field of concurrent computing include Edsger Dijkstra, Per Brinch Hansen, and C.A.R. Hoare.

Concurrent interaction and communication

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

Coordinating access to resources

One of the major issues in concurrent computing is preventing concurrent processes from interfering with each other. For example, consider the following algorithm for making withdrawals from a checking account represented by the shared resource `balance`:

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

```
8      return false;
9   }
```

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

Because concurrent systems rely on the use of shared resources (including communication media), concurrent computing in general requires the use of some form of arbiter somewhere in the implementation to mediate access to these resources.

Unfortunately, while many solutions exist to the problem of a conflict over one resource, many of those "solutions" have their own concurrency problems such as deadlock when more than one resource is involved.

Advantages

- Increased application throughput - parallel execution of a concurrent program allows the number of tasks completed in certain time period to increase.
- High responsiveness for input/output - input/output-intensive applications mostly wait for input or output operations to complete. Concurrent programming allows the time that would be spent waiting to be used for another task.
- More appropriate program structure - some problems and problem domains are well-suited to representation as concurrent tasks or processes.

Concurrent programming languages

Concurrent programming languages are programming languages that use language constructs for concurrency. These constructs may involve multi-threading, support for distributed computing, message passing, shared resources (including shared memory) or futures (known also as *promises*). Such languages are sometimes described as Concurrency Oriented Languages or Concurrency Oriented Programming Languages (COPL).

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

Many concurrent programming languages have been developed more as research languages (e.g. Pict) rather than as languages for production use. However, languages such as Erlang, Limbo, and occam have seen industrial use at various times in the last 20 years. Languages in which concurrency plays an important role include:

- ActorScript – theoretical purely actor-based language defined in terms of itself
- Ada
- Afnix – concurrent access to data is protected automatically (previously called *Aleph*, but unrelated to *Alef*)
- Alef – concurrent language with threads and message passing, used for systems programming in early versions of Plan 9 from Bell Labs
- Alice – extension to Standard ML, adds support for concurrency via futures.
- Ateji PX – an extension to Java with parallel primitives inspired from pi-calculus
- Axum – domain specific concurrent programming language, based on the Actor model and on the .NET Common Language Runtime using a C-like syntax.
- Chapel – a parallel programming language being developed by Cray Inc.
- Charm++ – C++-like language for thousands of processors.
- Cilk – a concurrent C
- Cω – C Omega, a research language extending C#, uses asynchronous communication
- Clojure – a modern Lisp targeting the JVM
- Concurrent Clean – a functional programming language, similar to Haskell
- Concurrent Haskell – lazy, pure functional language operating concurrent processes on shared memory
- Concurrent ML – a concurrent extension of Standard ML
- Concurrent Pascal – by Per Brinch Hansen
- Curry
- E – uses promises, ensures deadlocks cannot occur
- Eiffel – through its SCOOP mechanism based on the concepts of Design by Contract
- Erlang – uses asynchronous message passing with nothing shared
- Faust – Realtime functional programming language for signal processing. The Faust compiler provides automatic parallelization using either OpenMP or a specific work-stealing scheduler.
- Go – systems programming language with explicit support for concurrent programming
- Io – actor-based concurrency
- Janus features distinct "askers" and "tellers" to logical variables, bag channels; is purely declarative
- JoCaml
- Join Java – concurrent language based on the Java programming language
- Joule – dataflow language, communicates by message passing
- Joyce – a concurrent teaching language built on Concurrent Pascal with features from CSP by Per Brinch Hansen

- LabVIEW – graphical, dataflow programming language, in which functions are nodes in a graph and data is wires between those nodes. Includes object oriented language extensions.
- Limbo – relative of Alef, used for systems programming in Inferno (operating system)
- MultiLisp – Scheme variant extended to support parallelism
- Modula-3 – modern language in Algol family with extensive support for threads, mutexes, condition variables.
- Newsqueak – research language with channels as first-class values; predecessor of Alef
- occam – influenced heavily by Communicating Sequential Processes (CSP).
    - occam-π – a modern variant of occam, which incorporates ideas from Milner's π-calculus
- Orc – a heavily concurrent, nondeterministic language based on Kleene algebra.
- Oz – multiparadigm language, supports shared-state and message-passing concurrency, and futures
    - Mozart Programming System – multiplatform Oz
- Pict – essentially an executable implementation of Milner's π-calculus
- Perl with AnyEvent and Coro
- Python with greenlet and gevent.
- Reia – uses asynchronous message passing between shared-nothing objects
- SALSA – actor language with token-passing, join, and first-class continuations for distributed computing over the Internet
- Scala – a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way
- SR – research language
- Stackless Python
- SuperPascal – a concurrent teaching language built on Concurrent Pascal and Joyce by Per Brinch Hansen
- Unicon – Research language.
- Termite Scheme adds Erlang-like concurrency to Scheme
- TNSDL – a language used at developing telecommunication exchanges, uses asynchronous message passing
- VHDL – VHSIC Hardware Description Language, aka IEEE STD-1076
- XC – a concurrency-extended subset of the C programming language developed by XMOS based on Communicating Sequential Processes. The language also offers built-in constructs for programmable I/O.

Many other languages provide support for concurrency in the form of libraries (on level roughly comparable with the above list).

Models of concurrency

There are several models of concurrent computing, which can be used to understand and analyze concurrent systems. These models include:

- Actor model
  - Object-capability model for security
- Petri nets
- Process calculi such as
  - Ambient calculus
  - Calculus of Communicating Systems (CCS)
  - Communicating Sequential Processes (CSP)
  - π-calculus

Chapter 6

Shared Memory and Message Passing

Shared memory

In computing, **shared memory** is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Depending on context, programs may run on a single processor or on multiple separate processors. Using memory for communication inside a single program, for example among its multiple threads, is generally not referred to as *shared memory*.

In hardware

In computer hardware, **shared memory** refers to a (typically) large block of random access memory that can be accessed by several different central processing units (CPUs) in a multiple-processor computer system.

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location.

The issue with shared memory systems is that many CPUs need fast access to memory and will likely cache memory, which has two complications:

- CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.
- Cache coherence: Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data. Such coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processors. On the other hand they can sometimes become overloaded and become a bottleneck to performance

The alternatives to shared memory are distributed memory and distributed shared memory, each having a similar set of issues.

In software

In computer software, *shared memory* is either

- a method of inter-process communication (IPC), i.e. a way of exchanging data between programs running at the same time. One process will create an area in RAM which other processes can access, *or*
- a method of conserving memory space by directing accesses to what would ordinarily be copies of a piece of data to a single instance instead, by using virtual memory mappings or with explicit support of the program in question. This is most often used for shared libraries and for XIP.

Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication (as opposed to other mechanisms of IPC such as named pipes, Unix domain sockets or CORBA). On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (whereas other IPC methods can use a computer network), and care must be taken to avoid issues if processes sharing memory are running on separate CPUs and the underlying architecture is not cache coherent.

IPC by shared memory is used for example to transfer images between the application and the X server on Unix systems, or inside the IStream object returned by CoMarshalInterThreadInterfaceInStream in the COM libraries under Windows.

Dynamic libraries are generally held in memory once and mapped to multiple processes, and only pages that had to be customized for the individual process (because a symbol resolved differently there) are duplicated, usually with a mechanism that transparently copies the page when a write is attempted, and then lets the write succeed on the private copy.

POSIX provides a standardized API for using shared memory, *POSIX Shared Memory*. This uses the function shm_open from sys/mman.h.

Unix System 5 provides an API for shared memory as well. This uses shmget from sys/shm.h.

BSD systems provide "anonymous mapped memory" which can be used by several processes.

Recent 2.6 Linux kernel builds have started to offer /dev/shm as shared memory in the form of a RAM disk, more specifically as a world-writable directory that is stored in memory. /dev/shm support is completely optional within the kernel configuration file. It is included by default in both Fedora and Ubuntu distributions.

Message passing

Message passing in computer science is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Overview

Message passing is the paradigm of communication where messages are sent from a sender to one or more recipients. Forms of messages include (remote) method invocation, signals, and data packets. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

Prominent theoretical foundations of concurrent computation, such as the Actor model and the process calculi are based on message passing. Implementations of concurrent systems that use message passing can either have message passing as an integral part of the language, or as a series of library calls from the language. Examples of the former include many distributed object systems. Examples of the latter include Microkernel operating systems pass messages between one kernel and one or more server blocks, and the Message Passing Interface used in High Performance Computing.

Message passing systems and models

Distributed object and remote method invocation systems like ONC RPC, Corba, Java RMI, DCOM, SOAP, .NET Remoting, CTOS, QNX Neutrino RTOS, OpenBinder, D-Bus and similar are message passing systems.

Message passing systems have been called "shared nothing" systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

Message passing model based programming languages typically define messaging as the (usually asynchronous) sending (usually by copy) of a data item to a communication endpoint (Actor, process, thread, socket, *etc.*). Such messaging is used in Web Services by SOAP. This concept is the higher-level version of a datagram except that messages

can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.

Messages are also commonly used in the same sense as a means of interprocess communication; the other common technique being streams or pipes, in which data are sent as a sequence of elementary data items instead (the higher-level version of a virtual circuit).

Synchronous versus Asynchronous Message Passing

Synchronous message passing systems require the sender and receiver to wait for each other to transfer the message. That is, the sender will not continue until the receiver has received the message.

Synchronous communication has two advantages. The first advantage is that reasoning about the program can be simplified in that there is a synchronisation point between sender and receiver on message transfer. The second advantage is that no buffering is required. The message can always be stored on the receiving side, because the sender will not continue until the receiver is ready.

Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready. The advantage of asynchronous communication is that the sender and receiver can overlap their computation because they do not wait for each other.

Synchronous communication can be built on top of asynchronous communication by ensuring that the sender always wait for an acknowledgement message from the receiver before continuing.

The buffer required in asynchronous communication can cause problems when it is full. A decision has to be made whether to block the sender or whether to discard future messages. If the sender is blocked, it may lead to an unexpected deadlock. If messages are dropped, then communication is no longer reliable.

Message passing versus Calling

Message passing should be contrasted with the alternative communication method for passing information between programs - the Call. In a traditional `Call`, arguments are passed to the "callee" (the receiver) typically by one or more general purpose registers or in a parameter list containing the addresses of each of the arguments. This form of communication differs from message passing in at least three crucial areas -

- total memory usage
- transfer time
- locality

In message passing, each of the arguments has to have sufficient available *extra* memory for copying the existing argument into a portion of the new message. This applies irrespective of the size of the original arguments - so if one of the arguments is (say) an HTML string of 31,000 octets describing a web page, it has to be copied in its entirety (and perhaps even transmitted) to the receiving program (if not a local program).

By contrast, for the call method, only an address of say 4 or 8 bytes needs to be passed for each argument and may even be passed in a general purpose register requiring zero additional storage and zero "transfer time". This of course is not possible for distributed systems since an (absolute) address - in the callers address space - is normally meaningless to the remote program (however, a relative address might in fact be usable if the callee had an *exact* copy of, at least some of, the callers memory in advance).

Examples of message passing style

- Actor model implementation
- Amorphous computing
- Antiobjects
- Flow-based programming
- SOAP (protocol)
- Smalltalk

Influences on other programming models

In the terminology of some object-oriented programming languages, a message is the single means to pass control to an object. If the object "responds" to the message, it has a method for that message. In pure object-oriented programming, message passing is performed exclusively through a dynamic dispatch strategy.

Objects can send messages to other objects from within their method bodies. Message passing enables extreme late binding in systems. Sending the same message to an object twice will usually result in the object applying the method twice. Two messages are considered to be the same message type, if the name and the arguments of the message are identical. Some languages support the forwarding or delegation of method invocations from one object to another if the former has no method to handle the message, but "knows" another object that may have one.

Alan Kay has argued that message passing is more important than objects in OOP, and that objects themselves are often over-emphasized. The live distributed objects programming model builds upon this observation; it uses the concept of a distributed data flow to characterize the behavior of a complex distributed system in terms of message patterns, using high-level, functional-style specifications.

Chapter 7

Actor Model and Process Calculus

Actor model

In computer science, the **Actor model** 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

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 rekindled 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 dissertation, in 1981, 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, Beppe 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.

Research on the Actor model has been carried out at Caltech Computer Science, Kyoto University Tokoro Laboratory, MCC, MIT Artificial Intelligence Laboratory, SRI, Stanford University, University of Illinois at Urbana-Champaign Open Systems Laboratory, Pierre and Marie Curie University (University of Paris 6), University of Pisa, University of Tokyo Yonezawa Laboratory 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.

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 :

- 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.

Models prior to the Actor model

The Actor model builds on previous models of computation.

Lambda calculus

The lambda calculus of Alonzo Church can be viewed as the earliest message passing programming language. For example, the lambda expression below implements a tree data structure when supplied with parameters for a `leftSubTree` and `rightSubTree`. When such a tree is given a parameter message `"getLeft"`, it returns `leftSubTree` and likewise when given the message `"getRight"` it returns `rightSubTree`.

```
λ(leftSubTree,rightSubTree)
  λ(message)
    if (message == "getLeft") then leftSubTree
    else if (message == "getRight") then rightSubTree
```

However, the semantics of the lambda calculus were expressed using variable substitution in which the values of parameters were substituted into the body of an invoked lambda expression. The substitution model is unsuitable for concurrency because it does not allow the capability of sharing of changing resources. Inspired by the lambda calculus, the interpreter for the programming language Lisp made use of a data structure called an environment so that the values of parameters did not have to be substituted into the body of an invoked lambda expression. This allowed for sharing of the effects of updating shared data structures but did not provide for concurrency.

Simula

Simula 67 pioneered using message passing for computation, motivated by discrete event simulation applications. These applications had become large and unmodular in previous simulation languages. At each time step, a large central program would have to go through and update the state of each simulation object that changed depending on the state of whichever simulation objects that it interacted with on that step. Kristen Nygaard and Ole-Johan Dahl developed the idea (first described in an IFIP workshop in 1967) of having methods on each object that would update its own local state based on messages from other objects. In addition they introduced a class structure for objects with inheritance. Their innovations considerably improved the modularity of programs.

However, Simula used coroutine control structure instead of true concurrency.

Smalltalk

Alan Kay was influenced by message passing in the pattern-directed invocation of Planner in developing Smalltalk-71. Hewitt was intrigued by Smalltalk-71 but was put off by the complexity of communication that included invocations with many fields including *global*, *sender*, *receiver*, *reply-style*, *status*, *reply*, *operator selector*, *etc.*

In 1972 Kay visited MIT and discussed some of his ideas for Smalltalk-72 building on the Logo work of Seymour Papert and the "little person" model of computation used for teaching children to program. However, the message passing of Smalltalk-72 was quite

complex. Code in the language was viewed by the interpreter as simply a stream of tokens. As Dan Ingalls later described it:

The first (token) encountered (in a program) was looked up in the dynamic context, to determine the receiver of the subsequent message. The name lookup began with the class dictionary of the current activation. Failing there, it moved to the sender of that activation and so on up the sender chain. When a binding was finally found for the token, its value became the receiver of a new message, and the interpreter activated the code for that object's class.

Thus the message passing model in Smalltalk-72 was closely tied to a particular machine model and programming language syntax that did not lend itself to concurrency. Also, although the system was bootstrapped on itself, the language constructs were not formally defined as objects that respond to **Eval** messages.This led some to believe that a new mathematical model of concurrent computation based on message passing should be simpler than Smalltalk-72.

Subsequent versions of the Smalltalk language largely followed the path of using the virtual methods of Simula in the message passing structure of programs. However Smalltalk-72 made primitives such as integers, floating point numbers, *etc.* into objects. The authors of Simula had considered making such primitives into objects but refrained largely for efficiency reasons. Java at first used the expedient of having both primitive and object versions of integers, floating point numbers, *etc.* The C# programming language (and later versions of Java, starting with Java 1.5) adopted the less elegant solution of using *boxing* and *unboxing*, a variant of which had been used earlier in some Lisp implementations.

The Smalltalk system went on to become very influential, innovating in bitmap displays, personal computing, the class browser interface, and many other ways.

Petri nets

Prior to the development of the Actor model, Petri nets were widely used to model nondeterministic computation. However, they were widely acknowledged to have an important limitation: they modeled control flow but not data flow. Consequently they were not readily composable thereby limiting their modularity. Hewitt pointed out another difficulty with Petri nets: simultaneous action. *I.e.*, the atomic step of computation in Petri nets is a transition in which tokens *simultaneously* disappear from the input places of a transition and appear in the output places. The physical basis of using a primitive with this kind of simultaneity seemed questionable to him. Despite these apparent difficulties, Petri nets continue to be a popular approach to modelling concurrency, and are still the subject of active research.

Threads, locks, and buffers (channels)

Prior to the Actor model, concurrency was defined in low level machine terms of threads, locks and buffers(channels). It certainly is the case that implementations of the Actor model typically make use of these hardware capabilities. However, there is no reason that the model could not be implemented directly in hardware without exposing any hardware threads and locks. Also, there is no necessary relationship between the number of Actors, threads, and locks that might be involved in a computation. Implementations of the Actor model are free to make use of threads and locks in any way that is compatible with the laws for Actors.

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, 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, Tony Hoare and Per Brinch Hansen 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. 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.

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 model. 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.

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.

Modeling other concurrency systems

Other concurrency systems (*e.g.* process calculi) can be modeled in the Actor model using a two-phase commit protocol.

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 increasingly better approximations from an initial behavior called $\bot_S$ using a behavior approximating function $\mathbf{progression}_S$ to construct a denotation (meaning) for S as follows [Hewitt 2008; Clinger 1981]:

$$\mathbf{Denote}_S \equiv \sqcup_{i \in \omega} \mathbf{progression}_S^{\,i}(\bot_S)$$

In this way, S can be mathematically characterized in terms of all its possible behaviors (including those involving unbounded nondeterminism). Although $\mathtt{Denote}_S$ is not an implementation of S, it can be used to prove a generalization of the Church-Turing-Rosser-Kleene thesis [Kleene 1943]:

Enumeration Theorem: If the primitive Actors of a closed Actor system are effective, then its possible outputs are recursively enumerable.
Proof: Follows immediately from the Representation Theorem.

Relationship to mathematical logic

The development of the Actor model has an interesting relationship to mathematical logic. One of the key motivations for its development was to understand and deal with the control structure issues that arose in development of the Planner programming language. 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. This result is still somewhat controversial and it reversed previous expectations because Kowalski's thesis is true for sequential computation and even some kinds of parallel computation, *e.g.* 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.

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.

Security

The security of Actors can be protected in the following ways:

- hardwiring in which Actors are physically connected
- 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. 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 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 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. 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.

Current importance

Forty years after the publication of Moore's Law, hardware development is furthering local and nonlocal massive concurrency. Local concurrency is enabled by new hardware for 64-bit multi-core (Platform 2015 Unveiled at IDF Spring 2005) microprocessors, multi-chip modules, and high performance interconnect. Nonlocal concurrency is being enabled by new hardware for wired and wireless broadband packet switched communications. Both local and nonlocal storage capacities are growing exponentially.

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 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.

- 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 .

Actor researchers

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 Aki 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.

Process calculus

In computer science, the **process calculi** (or **process algebras**) are a diverse family of related approaches to formally modelling concurrent systems. Process calculi provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes (e.g., using bisimulation). Leading examples of process calculi include CSP, CCS, ACP, and LOTOS. More recent additions to the family include the π-calculus, the ambient calculus, PEPA and the fusion calculus.

Essential features

While the variety of existing process calculi is very large (including variants that incorporate stochastic behaviour, timing information, and specializations for studying

molecular interactions), there are several features that all process calculi have in common:

- Representing interactions between independent processes as communication (message-passing), rather than as the modification of shared variables
- Describing processes and systems using a small collection of primitives, and operators for combining those primitives
- Defining algebraic laws for the process operators, which allow process expressions to be manipulated using equational reasoning

Mathematics of processes

To define a **process calculus**, one starts with a set of *names* (or *channels*) whose purpose is to provide means of communication. In many implementations, channels have rich internal structure to improve efficiency, but this is abstracted away in most theoretic models. In addition to names, one needs a means to form new processes from old. The basic operators, always present in some form or other, allow:

- parallel composition of processes
- specification of which channels to use for sending and receiving data
- sequentialization of interactions
- hiding of interaction points
- recursion or process replication

Parallel composition

Parallel composition of two processes *P* and *Q*, usually written $P|Q$, is the key primitive distinguishing the process calculi from sequential models of computation. Parallel composition allows computation in *P* and *Q* to proceed simultaneously and independently. But it also allows interaction, that is synchronisation and flow of information from *P* to *Q* (or vice versa) on a channel shared by both. Crucially, an agent or process can be connected to more than one channel at a time.

Channels may be synchronous or asynchronous. In the case of a synchronous channel, the agent sending a message waits until another agent has received the message. Asynchronous channels do not require any such synchronization. In some process calculi (notably the π-calculus) channels themselves can be sent in messages through (other) channels, allowing the topology of process interconnections to change. Some process calculi also allow channels to be *created* during the execution of a computation.

Communication

Interaction can be (but isn't always) a *directed* flow of information. That is, input and output can be distinguished as dual interaction primitives. Process calculi that make such distinctions typically define an input operator (*e.g.* $x(v)$) and an output operator (*e.g.* $x\langle y\rangle$), both of which name an interaction point (here x) that is used to synchronise with a dual interaction primitive.

Should information be exchanged, it will flow from the outputting to the inputting process. The output primitive will specify the data to be sent. In $x\langle y\rangle$, this data is y. Similarly, if an input expects to receive data, one or more bound variables will act as place-holders to be substituted by data, when it arrives. In $x(v)$, v plays that role. The choice of the kind of data that can be exchanged in an interaction is one of the key features that distinguishes different process calculi.

Sequential composition

Sometimes interactions must be temporally ordered. For example, it might be desirable to specify algorithms such as: *first receive some data on x and then send that data on y*. *Sequential composition* can be used for such purposes. It is well known from other models of computation. In process calculi, the sequentialisation operator is usually integrated with input or output, or both. For example, the process $x(v) \cdot P$ will wait for an input on x. Only when this input has occurred will the process P be activated, with the received data through x substituted for identifier v.

Reduction semantics

The key operational reduction rule, containing the computational essence of process calculi, can be given solely in terms of parallel composition, sequentialization, input, and output. The details of this reduction vary among the calculi, but the essence remains roughly the same. The reduction rule is:

$$x\langle y\rangle \cdot P \mid x(v) \cdot Q \longrightarrow P \mid Q[{}^{y}/_{v}]$$

The interpretation of this reduction rule is:

1. The process $x\langle y\rangle \cdot P$ sends a message, here y, along the channel x. Dually, the process $x(v) \cdot Q$ receives that message on channel x.
2. Once the message has been sent, $x\langle y\rangle \cdot P$ becomes the process P, while $x(v) \cdot Q$ becomes the process $Q[{}^{y}/_{v}]$, which is Q with the place-holder v substituted by y, the data received on x.

The class of processes that P is allowed to range over as the continuation of the output operation substantially influences the properties of the calculus.

Hiding

Processes do not limit the number of connections that can be made at a given interaction point. But interaction points allow interference (i.e. interaction). For the synthesis of compact, minimal and compositional systems, the ability to restrict interference is crucial. *Hiding* operations allow control of the connections made between interaction points when composing agents in parallel. Hiding can be denoted in a variety of ways. For example, in the π-calculus the hiding of a name *x* in *P* can be expressed as $(\nu\ x)P$, while in CSP it might be written as $P \setminus \{x\}$.

Recursion and replication

The operations presented so far describe only finite interaction and are consequently insufficient for full computability, which includes non-terminating behaviour. *Recursion* and *replication* are operations that allow finite descriptions of infinite behaviour. Recursion is well known from the sequential world. Replication !*P* can be understood as abbreviating the parallel composition of a countably infinite number of *P* processes:

$$!P = P|!P$$

Null process

Process calculi generally also include a *null process* (variously denoted as *nil*, 0, *STOP*, δ, or some other appropriate symbol) which has no interaction points. It is utterly inactive and its sole purpose is to act as the inductive anchor on top of which more interesting processes can be generated.

Discrete and continuous process algebra

Process algebra has been studied for discrete time and continuous time (real time or dense time)

History

In the first half of the 20th century, various formalisms were proposed to capture the informal concept of a *computable function*, with μ-recursive functions, Turing Machines and the lambda calculus possibly being the best-known examples today. The surprising fact that they are essentially equivalent, in the sense that they are all encodable into each other, supports the Church-Turing thesis. Another shared feature is more rarely commented on: they all are most readily understood as models of *sequential* computation. The subsequent consolidation of computer science required a more subtle formulation of the notion of computation, in particular explicit representations of concurrency and communication. Models of concurrency such as the process calculi, Petri-Nets in 1962, and the Actor model in 1973 emerged from this line of enquiry.

Research on process calculi began in earnest with Robin Milner's seminal work on the Calculus of Communicating Systems (CCS) during the period from 1973 to 1980. C.A.R. Hoare's Communicating Sequential Processes (CSP) first appeared in 1978, and was subsequently developed into a fully-fledged process calculus during the early 1980s. There was much cross-fertilization of ideas between CCS and CSP as they developed. In 1982 Jan Bergstra and Jan Willem Klop began work on what came to be known as the Algebra of Communicating Processes (ACP), and introduced the term *process algebra* to describe their work. CCS, CSP, and ACP constitute the three major branches of the process calculi family: the majority of the other process calculi can trace their roots to one of these three calculi.

Current research

Various different process calculi have been studied and not all of them fit the paradigm sketched here. The most prominent example may be the ambient calculus. This is to be expected as process calculi are an active field of study. Currently research on process calculi focuses on the following problems.

- Developing new process calculi for better modeling of computational phenomena.

- Finding well-behaved subcalculi of a given process calculus. This is valuable because (1) most calculi are fairly *wild* in the sense that they are rather general and not much can be said about arbitrary processes; and (2) computational applications rarely exhaust the whole of a calculus. Rather they use only processes that are very constrained in form. Constraining the shape of processes is mostly studied by way of type systems.

- Logics for processes that allow one to reason about (essentially) arbitrary properties of processes, following the ideas of Hoare logic.

- Behavioural theory: what does it mean for two processes to be the same? How can we decide whether two processes are different or not? Can we find representatives for equivalence classes of processes? Generally, processes are considered to be the same if no context, that is other processes running in parallel, can detect a difference. Unfortunately, making this intuition precise is subtle and mostly yields unwieldy characterisations of equality (which in most cases must also be undecidable, as a consequence of the halting problem). Bisimulations are a technical tool that aids reasoning about process equivalences.

- Expressivity of calculi. Programming experience shows that certain problems are easier to solve in some languages than in others. This phenomenon calls for a more precise characterisation of the expressivity of calculi modeling computation than that afforded by the Church-Turing thesis. One way of doing this is to consider encodings between two formalisms and see what properties encodings can potentially preserve. The more properties can be preserved, the more expressive the target of the encoding is said to be. For process calculi, the

celebrated results are that the synchronous π-calculus is more expressive than its asynchronous variant, has the same expressive power as the higher-order π-calculus, but is less than the ambient calculus.

- Using process calculus to model biological systems. It is thought by some that the compositionality offered by process-theoretic tools can help biologists to organise their knowledge more formally.

Relationship to other models of concurrency

The history monoid is the free object that is generically able to represent the histories of individual communicating processes. A process calculus is then a formal language imposed on a history monoid in a consistent fashion. That is, a history monoid can only record a sequence of events, with synchronization, but does not specify the allowed state transitions. Thus, a process calculus is to a history monoid what a formal language is to a free monoid (a formal language is a subset of the set of all possible finite-length strings of an alphabet generated by the Kleene star).

The use of channels for communication is one of the features distinguishing the process calculi from other models of concurrency, such as Petri nets and the Actor model. One of the fundamental motivations for including channels in the process calculi was to enable certain algebraic techniques, thereby making it easier to reason about processes algebraically.

Chapter 8

Concurrency Control and Critical Section

Concurrency control

In computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

Computer systems, both software and hardware, consist of modules, or components. Each component is designed to operate correctly, i.e., to obey to or meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction. Operation consistency and correctness should be achieved with as good as possible efficiency, without reducing performance below reasonable.

Concurrency control in databases

Comments:

1. This section is applicable to all transactional systems, i.e., to all systems that use *database transactions* (*atomic transactions*; e.g., transactional objects in Systems management and in networks of smartphones), not only database management systems (DBMSs).
2. DBMSs need to deal also with concurrency control issues not typical just to database transactions but rather to operating systems in general. These issues are out of the scope of this section.

Concurrency control in Database management systems (DBMS; e.g., Bernstein et al. 1987, Weikum and Vossen 2001), other transactional objects, and related distributed

applications (e.g., Grid computing and Cloud computing) ensures that *database transactions* are performed concurrently without violating the data integrity of the respective databases. Thus concurrency control is an essential element for correctness in any system where two database transactions or more, executed with time overlap, can access the same data, e.g., virtually in any general-purpose database system. A well established concurrency control theory exists for database systems: serializability theory, which allows to effectively design and analyze concurrency control methods and mechanisms.

To ensure correctness, A DBMS usually guarantees that only *serializable* transaction schedules are generated, unless *serializability* is intentionally relaxed to increase performance, but only in cases that application correctness is not harmed. For maintaining correctness in cases of failed (aborted) transactions (which can always happen for many reasons) schedules also need to have the *recoverability* property. A DBMS also guarantees that no effect of *committed* transactions is lost, and no effect of *aborted* (rolled back) transactions remains in the related database. Overall transaction characterization is usually summarized by the ACID rules below. As databases become distributed, or cooperate in distributed environments (e.g., Cloud computing), the effective distribution of concurrency control mechanisms receives special attention.

Database transaction and the ACID rules

The concept of a *database transaction* (or *atomic transaction*) has evolved in order to enable both a well understood database system behavior in a faulty environment where crashes can happen any time, and *recovery* from a crash to a well understood database state. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands). Every database transaction obeys the following rules (by support in the database system; i.e., a database system is designed to guarantee them for the transactions it runs):

- **Atomicity** - Either the effects of all or none of its operations remain when a transaction is completed (*committed* or *aborted* respectively). In other words, to the outside world a committed transaction appears to be indivisible, atomic. A transaction is a unit of work that appears as if it is either performed in its entirety, or not performed at all ("all or nothing" semantics).
- **Consistency** - Every transaction must leave the database in a consistent state, i.e., maintain the predetermined integrity rules of the database (constraints upon and among the database's objects). A transaction must transform a database from one consistent state to another consistent state.
- **Isolation** - Transactions cannot interfere with each other. Moreover, usually the effects of an incomplete transaction are not visible to another transaction. Providing isolation is the main goal of concurrency control.

- **Durability** - Effects of successful (committed) transactions must persist through crashes (typically by recording the transaction's effects and its commit event in a non-volatile memory).

Why is concurrency control needed?

If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected result may occur. Here are some typical examples:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. The incorrect summary problem: While one transaction takes a summary over values of a repeated data-item, a second transaction updated some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether a certain update result has been included in the summary or not.

Concurrency control mechanisms

Types of mechanisms

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations, and then abort a transaction if the desired rules are violated.
- **Pessimistic** - Block an operation of a transaction if it may cause violation of the rules until the possibility of violation disappears.
- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking to transaction's end, as done with optimistic.

Many methods for concurrency control exist. Most of them can be implemented within either main category above. Major methods, which have each many variants, and in some cases may overlap or be combined, include:

- Two-phase locking (2PL) - Controlling access to data by locks assigned to the data
- Serialization (also called Serializability, or Conflict, or Precedence) graph checking - Checking for cycles in the schedule's graph.
- Timestamp ordering (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.
- Commitment ordering (or Commit ordering; CO) - Controlling or checking transactions' order of commit events to be compatible with their respective precedence order.

Other major concurrency control types that are utilized in conjunction with the methods above include:

- Multiversion concurrency control (MVCC) - Increasing concurrency and performance by maintaining several last generational values (versions) for each data item (datum; database object).
- Index concurrency control - Synchronizing access operations to indexes, rather than to user data.

The most common mechanism type in database systems since their early days in the 1970s has been *Strong strict Two-phase locking* (SS2PL; also called *Rigorous scheduling* or *Rigorous 2PL*) which is a special case (variant) of both Two-phase locking (2PL) and Commitment ordering (CO). In spite of its long name (for historical reasons) the idea of the mechanism is simple: "Release all locks applied by a transaction only after the transaction has ended."

Major goals

Serializability

For correctness, a common major goal of most concurrency control mechanisms is generating schedules with the *Serializability* property. Without serializability undesirable phenomena may occur, e.g., money may disappear from accounts, or be generated from nowhere. Serializability of a schedule means equivalence to some *serial* schedule with the same transactions (i.e., in which transactions are sequential with no overlap in time). Serializability is considered the highest level of isolation between database transactions, and the major correctness criterion for concurrent transactions. In some cases compromised, relaxed forms of serializability are allowed for better performance (e.g., the popular *Snapshot isolation* mechanism), if application's correctness is not violated by the relaxation.

Almost all implemented concurrency control mechanisms achieve serializability by providing *Conflict serializablity*, a broad special case of serializability (i.e., it covers, enables most serializable schedules; does not impose significant additional delay-causing constraints) which can be implemented effectively.

Recoverability

Concurrency control also ensures the *Recoverability* property for maintaining correctness in cases of aborted transactions (which can always happen for many reasons). Recoverability means that committed transactions have not read data written by aborted transactions. Unlike Serializability, Recoverability cannot be compromised, relaxed at any case, since any relaxation results in quick database integrity violation upon aborts. The major mechanisms listed above are serializability mechanisms. None of them in its general form automatically provides recoverability, and special considerations and mechanism enhancements are needed to support recoverability. A commonly utilized special case of recoverability is *Strictness*, which allows efficient database recovery from failure (but excludes optimistic implementations; e.g, Strict CO (SCO) does not not allow an optimistic implementation, but allows semi-optimistic).

Comment: Note that the *Recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is rather needed to correctly automatically handle transaction aborts, which may be unrelated to database failure and recovery from it.

Distribution: Distributed serializability and Commitment ordering

As database systems become distributed, or cooperate in distributed environments (e.g., in Grid computing, Cloud computing, and networks with smartphones), a need exists for effective distributed concurrency control mechanisms. Achieving the Serializability property of a distributed system's schedule effectively poses special challenges typically not met by most local serializability mechanisms. This is especially due to a need in costly distribution of concurrency control information amid communication and computer latency. The *Commitment ordering* (Commit ordering, CO; Raz 1992) technique, disclosed publicly in 1991, which does not require the distribution of such information, provides a general effective solution (reliable, high-performance, and scalable) for both distributed and global serializability, also in a heterogeneous environment with database systems (or other transactional objects) with different (any) concurrency control mechanisms. The existence of such a solution has been considered "unlikely" until 1991, and by many experts also later, due to misunderstanding of the CO solution.

SS2PL mentioned above is a variant of CO and thus also effective to achieve distributed serializability. It also provides Strictness and thus Recoverability. Possessing all these desired properties, together with known efficient locking based implementations, explains SS2PL's popularity. SS2PL has been utilized to efficiently achieve Distributed and Global serializability since the 1980. However, SS2PL is blocking and constraining

(pessimistic), and with the proliferation of distribution and utilization of systems different from traditional database systems (e.g., as in Cloud computing), less constraining types of CO (e.g., Optimistic CO) may be needed for better performance.

Comment: *Distributed conflict serializability* in its general form is difficult to achieve efficiently, and it is achieved efficiently via *Distributed CO*. Unlike Serializability, *Distributed recoverability* and *Distributed strictness* can be achieved efficiently in a straightforward way, similarly to the way Distributed CO is achieved (Raz 1992, page 307). Differently from the general Distributed CO, *Distributed SS2PL* exists automatically when all local components are SS2PL based, a fact known and utilized for efficient (Distributed SS2PL, which implies) Distributed serializability and strictness since the 1980s. Less constrained Distributed serializability and strictness can be efficiently achieved by Distributed Strict CO (SCO), or by a mix of SS2PL and SCO based local components.

Critical section

In concurrent programming a **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.

By carefully controlling which variables are modified inside and outside the critical section (usually, by accessing important state only from within), concurrent access to that state is prevented. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time.

How critical sections are implemented varies among operating systems.

The simplest method is to prevent any change of processor control inside the critical section. On uni-processor systems, this can be done by disabling interrupts on entry into the critical section, avoiding system calls that can cause a context switch while inside the section and restoring interrupts to their previous state on exit. Any thread of execution entering any critical section anywhere in the system will, with this implementation, prevent any other thread, including an interrupt, from getting the CPU and therefore from entering any other critical section or, indeed, any code whatsoever, until the original thread leaves its critical section.

This brute-force approach can be improved upon by using semaphores. To enter a critical section, a thread must obtain a semaphore, which it releases on leaving the section. Other threads are prevented from entering the critical section at the same time as the original thread, but are free to gain control of the CPU and execute other code, including other critical sections that are protected by different semaphores.

Some confusion exists in the literature about the relationship between different critical sections in the same program. In general, a resource that must be protected from concurrent access may be accessed by several pieces of code. Each piece must be guarded by a common semaphore. Is each piece now a critical section or are all the pieces guarded by the same semaphore in aggregate a single critical section? This confusion is evident in definitions of a critical section such as "... a piece of code that can only be executed by one process or thread at a time". This only works if all access to a protected resource is contained in one "piece of code", which requires either the definition of a piece of code or the code itself to be somewhat contrived.

Application Level Critical Sections

Application-level critical sections reside in the memory range of the process and are usually modifiable by the process itself. This is called a user-space object because the program run by the user (as opposed to the kernel) can modify and interact with the object. However the functions called may jump to kernel-space code to register the user-space object with the kernel.

Example Code For Critical Sections with POSIX pthread library

```
/* Sample C/C++, Unix/Linux */
#include <pthread.h>

/* This is the critical section object (statically allocated). */
static pthread_mutex_t cs_mutex = PTHREAD_MUTEX_INITIALIZER;

void f()
{
    /* Enter the critical section -- other threads are locked out */
    pthread_mutex_lock( &cs_mutex );

    /* Do some thread-safe processing! */

    /*Leave the critical section -- other threads can now
pthread_mutex_lock()  */
    pthread_mutex_unlock( &cs_mutex );
}
```

Example Code For Critical Sections with Win32 API

```
/* Sample C/C++, Windows, link to kernel32.dll */
#include <windows.h>
```

```
static CRITICAL_SECTION cs; /* This is the critical section object --
once initialized,
                               it cannot be moved in memory */
                             /* If you program in OOP, declare this as a
non-static member in your class */

/* Initialize the critical section before entering multi-threaded
context. */
InitializeCriticalSection(&cs);

void f()
{
    /* Enter the critical section -- other threads are locked out */
    EnterCriticalSection(&cs);

    /* Do some thread-safe processing! */

    /* Leave the critical section -- other threads can now
EnterCriticalSection() */
    LeaveCriticalSection(&cs);
}

/* Release system object when all finished -- usually at the end of the
cleanup code */
DeleteCriticalSection(&cs);
```

Note that on Windows NT (not 9x/ME), the function **TryEnterCriticalSection()** can be used to attempt to enter the critical section. This function returns immediately so that the thread can do other things if it fails to enter the critical section (usually due to another thread having locked it). With the pthreads library, the equivalent function is **pthread_mutex_trylock()**. Note that the use of a CriticalSection is not the same as a Win32 Mutex, which is an object used for *inter-process* synchronization. A Win32 CriticalSection is for *intra-process* synchronization (and is much faster as far as lock times), however it cannot be shared across processes.

Kernel Level Critical Sections

Typically, critical sections prevent process and thread migration between processors and the preemption of processes and threads by interrupts and other processes and threads.

Critical sections often allow nesting. Nesting allows multiple critical sections to be entered and exited at little cost.

If the scheduler interrupts the current process or thread in a critical section, the scheduler will either allow the process or thread to run to completion of the critical section, or it will schedule the process or thread for another complete quantum. The scheduler will not migrate the process or thread to another processor, and it will not schedule another process or thread to run while the current process or thread is in a critical section.

Similarly, if an interrupt occurs in a critical section, the interrupt's information is recorded for future processing, and execution is returned to the process or thread in the critical section. Once the critical section is exited, and in some cases the scheduled quantum completes, the pending interrupt will be executed.

Since critical sections may execute only on the processor on which they are entered, synchronization is only required within the executing processor. This allows critical sections to be entered and exited at almost zero cost. No interprocessor synchronization is required, only instruction stream synchronization. Most processors provide the required amount of synchronization by the simple act of interrupting the current execution state. This allows critical sections in most cases to be nothing more than a per processor count of critical sections entered.

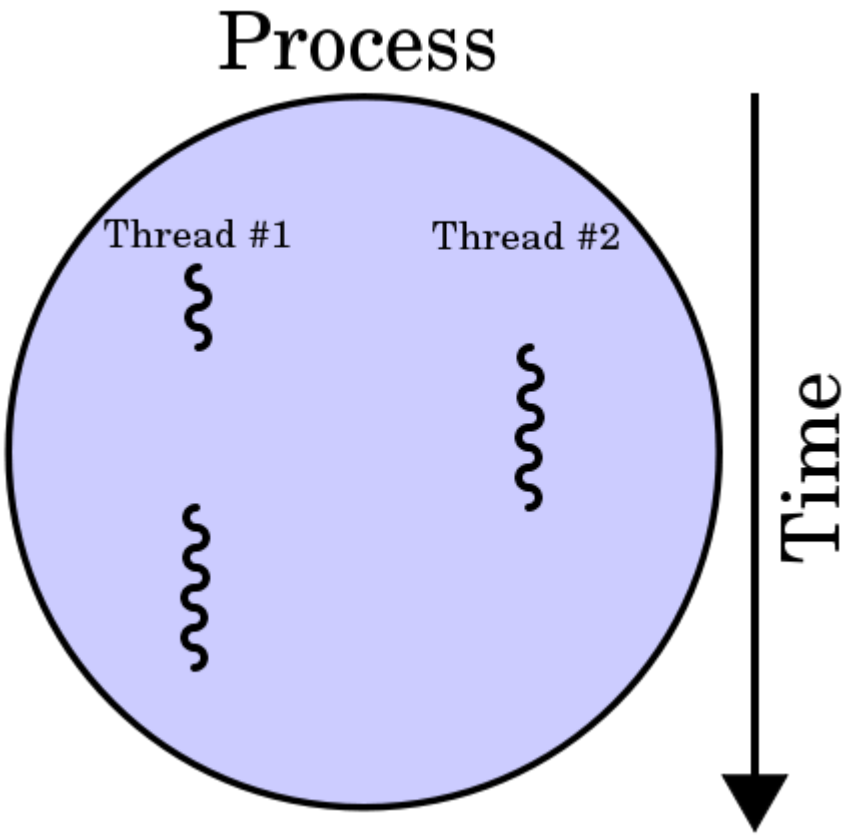
Performance enhancements include executing pending interrupts at the exit of all critical sections and allowing the scheduler to run at the exit of all critical sections. Furthermore, pending interrupts may be transferred to other processors for execution.

Critical sections should not be used as a long-lived locking primitive. They should be short enough that the critical section will be entered, executed, and exited without any interrupts occurring, neither from hardware much less the scheduler.

Chapter 9

Thread (Computer Science) and Communicating Sequential Processes

Thread (computer science)



A process with two threads of execution

In computer science, a **thread of execution** is the smallest unit of processing that can be scheduled by an operating system. It generally results from a fork of a computer program into two or more concurrently running tasks. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is

contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment). To give an analogy, multiple threads in a process are like multiple cooks reading off the same cook book and following its instructions, not necessarily from the same page.

On a single processor, **multithreading** generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multi-core system, the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The kernel of an operating system allows programmers to manipulate threads via the system call interface. Some implementations are called a *kernel thread*, whereas a *lightweight process* (LWP) is a specific type of kernel thread that shares the same state and information.

Programs can have *user-space threads* when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of ad-hoc time-slicing.

Threads compared with processes

Threads differ from traditional multitasking operating system processes in that:

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerable state information, whereas multiple threads within a process share state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms.
- Context switching between threads in the same process is typically faster than context switching between processes.

Systems like Windows NT and OS/2 are said to have "cheap" threads and "expensive" processes; in other operating systems there is not so great a difference except the cost of address space switch which implies a TLB flush.

Multithreading: Advantages/Uses

Multithreading as a widespread programming and execution model allows multiple threads to exist within the context of a single process. These threads share the process' resources but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps

the most interesting application of the technology is when it is applied to a *single* process to enable *parallel execution* on a *multiprocessor* system.

This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines — because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually-exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.

Another advantage of multithreading, even for single-CPU systems, is the ability for an application to remain responsive to input. In a single threaded program, if the main execution thread blocks on a long running task, the entire application can appear to freeze. By moving such long running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background.

Operating systems schedule threads in one of two ways:

1. *Preemptive multithreading* is generally considered the superior approach, as it allows the operating system to determine when a context switch should occur. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing priority inversion or other negative effects which may be avoided by cooperative multithreading.
2. *Cooperative multithreading*, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available.

Traditional mainstream computing hardware did not have much support for multithreading, because switching between threads was generally already quicker than full process context switches. Processors in embedded systems, which have higher requirements for real-time behaviors, might support multithreading by decreasing the thread-switch time, perhaps by allocating a dedicated register file for each thread instead of saving/restoring a common register file. In the late 1990s, the idea of executing instructions from multiple threads simultaneously has become known as simultaneous multithreading. This feature was introduced in Intel's Pentium 4 processor, with the name *hyper threading.*

Processes, kernel threads, user threads, and fibers

A *process* is the "heaviest" unit of kernel scheduling. Processes own resources allocated by the operating system. Resources include memory, file handles, sockets, device

handles, and windows. Processes do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way. Processes are typically preemptively multitasked.

A *kernel thread* is the "lightest" unit of kernel scheduling. At least one kernel thread exists within each process. If multiple kernel threads can exist within a process, then they share the same memory and file resources. Kernel threads are preemptively multitasked if the operating system's process scheduler is preemptive. Kernel threads do not own resources except for a stack, a copy of the registers including the program counter, and thread-local storage (if any). The kernel can assign one thread to each logical core in a system (because each core splits itself up into multiple logical cores if it supports multithreading, or only support one logical core per physical core if it does not support multithreading), and can swap out threads that get blocked. However, kernel threads take much longer than user threads to be swapped.

Threads are sometimes implemented in userspace libraries, thus called *user threads*. The kernel is not aware of them, so they are managed and scheduled in userspace. Some implementations base their *user threads* on top of several *kernel threads* to benefit from multi-processor machines (N:M model). Here the term "thread" (without kernel or user qualifier) defaults to referring to kernel threads. User threads as implemented by virtual machines are also called green threads. User threads are generally fast to create and manage, but cannot take advantage of multithreading or multiprocessing and get blocked if all of their associated kernel threads get blocked even if there are some user threads that are ready to run.

Fibers are an even lighter unit of scheduling which are cooperatively scheduled: a running fiber must explicitly "yield" to allow another fiber to run, which makes their implementation much easier than kernel or user threads. A fiber can be scheduled to run in any thread in the same process. This permits applications to gain performance improvements by managing scheduling themselves, instead of relying on the kernel scheduler (which may not be tuned for the application). Parallel programming environments such as OpenMP typically implement their tasks through fibers.

Thread and fiber issues

Concurrency and data structures

Threads in the same process share the same address space. This allows concurrently-running code to couple tightly and conveniently exchange data without the overhead or complexity of an IPC. When shared between threads, however, even simple data structures become prone to race hazards if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. Bugs caused by race hazards can be very difficult to reproduce and isolate.

To prevent this, threading APIs offer synchronization primitives such as mutexes to lock data structures against concurrent access. On uniprocessor systems, a thread running into a locked mutex must sleep and hence trigger a context switch. On multi-processor systems, the thread may instead poll the mutex in a spinlock. Both of these may sap performance and force processors in SMP systems to contend for the memory bus, especially if the granularity of the locking is fine.

I/O and scheduling

User thread or fiber implementations are typically entirely in userspace. As a result, context switching between user threads or fibers within the same process is extremely efficient because it does not require any interaction with the kernel at all: a context switch can be performed by locally saving the CPU registers used by the currently executing user thread or fiber and then loading the registers required by the user thread or fiber to be executed. Since scheduling occurs in userspace, the scheduling policy can be more easily tailored to the requirements of the program's workload.

However, the use of blocking system calls in user threads (as opposed to kernel threads) or fibers can be problematic. If a user thread or a fiber performs a system call that blocks, the other user threads and fibers in the process are unable to run until the system call returns. A typical example of this problem is when performing I/O: most programs are written to perform I/O synchronously. When an I/O operation is initiated, a system call is made, and does not return until the I/O operation has been completed. In the intervening period, the entire process is "blocked" by the kernel and cannot run, which starves other user threads and fibers in the same process from executing.

A common solution to this problem is providing an I/O API that implements a synchronous interface by using non-blocking I/O internally, and scheduling another user thread or fiber while the I/O operation is in progress. Similar solutions can be provided for other blocking system calls. Alternatively, the program can be written to avoid the use of synchronous I/O or other blocking system calls.

SunOS 4.x implemented "light-weight processes" or LWPs. NetBSD 2.x+, and DragonFly BSD implement LWPs as kernel threads (1:1 model). SunOS 5.2 through SunOS 5.8 as well as NetBSD 2 to NetBSD 4 implemented a two level model, multiplexing one or more user level threads on each kernel thread (M:N model). SunOS 5.9 and later, as well as NetBSD 5 eliminated user threads support, returning to a 1:1 model. FreeBSD 5 implemented M:N model. FreeBSD 6 supported both 1:1 and M:N, user could choose which one should be used with a given program using /etc/libmap.conf. Starting with FreeBSD 7, the 1:1 became the default. FreeBSD 8 no longer supports the M:N model.

The use of kernel threads simplifies user code by moving some of the most complex aspects of threading into the kernel. The program doesn't need to schedule threads or explicitly yield the processor. User code can be written in a familiar procedural style, including calls to blocking APIs, without starving other threads. However, kernel

threading on uniprocessor systems may force a context switch between threads at any time, and thus expose race hazards and concurrency bugs that would otherwise lie latent. On SMP systems, this is further exacerbated because kernel threads may literally execute concurrently on separate processors.

Models

1:1 (Kernel-level threading)

Threads created by the user are in 1-1 correspondence with schedulable entities in the kernel. This is the simplest possible threading implementation. Win32 used this approach from the start. On Linux, the usual C library implements this approach (via the NPTL or older LinuxThreads). The same approach is used by Solaris, NetBSD and FreeBSD.

N:1 (User-level threading)

An N:1 model implies that all application-level threads map to a single kernel-level scheduled entity; the kernel has no knowledge of the application threads. With this approach, context switching can be done very fast and, in addition, it can be implemented even on simple kernels which do not support threading. One of the major drawbacks however is that it cannot benefit from the hardware acceleration on multi-threaded processors or multi-processor computers: there is never more than one thread being scheduled at the same time. It is used by GNU Portable Threads.

N:M (Hybrid threading)

N:M maps some N number of application threads onto some M number of kernel entities, or "virtual processors." This is a compromise between kernel-level ("1:1") and user-level ("N:1") threading. In general, "N:M" threading systems are more complex to implement than either kernel or user threads, because changes to both kernel and user-space code are required. In the N:M implementation, the threading library is responsible for scheduling user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls. However, this increases complexity and the likelihood of priority inversion, as well as suboptimal scheduling without extensive (and expensive) coordination between the userland scheduler and the kernel scheduler.

Implementations

There are many different and incompatible implementations of threading. These include both kernel-level and user-level implementations. They however often follow more or less closely the POSIX Threads interface.

Kernel-level implementation examples

- Light Weight Kernel Threads (LWKT) in various BSDs

- M:N threading
- Native POSIX Thread Library (NPTL) for Linux, an implementation of the POSIX Threads (pthreads) standard
- Apple Multiprocessing Services version 2.0 and later, uses the built-in nanokernel in Mac OS 8.6 and later which was modified to support it.
- Microsoft Windows from Windows 95 and Windows NT onwards.

User-level implementation examples

- GNU Portable Threads
- FSU Pthreads
- Apple Inc.'s Thread Manager
- REALbasic (includes an API for cooperative threading)
- Netscape Portable Runtime (includes a user-space fibers implementation)
- State threads

Hybrid implementation examples

- Scheduler activations used by the NetBSD native POSIX threads library implementation (an N:M model as opposed to a 1:1 kernel or userspace implementation model)
- Marcel from the PM2 project.
- The OS for the Tera/Cray MTA
- Microsoft Windows 7

Fiber implementation examples

Fibers can be implemented without operating system support, although some operating systems or libraries provide explicit support for them.

- Win32 supplies a fiber API (Windows NT 3.51 SP3 and later)
- Ruby as Green threads

Programming Language Support

Many programming languages support threading in some capacity. Many implementations of C and C++ do not provide direct support for threading on their own, but provide access to the native threading APIs provided by the operating system. Some higher level (and usually cross platform) programming languages such as Java, Python, and .NET, expose threading to the developer while abstracting the platform specific differences in threading implementations in the runtime to the developer. A number of other programming languages also try to abstract the concept of concurrency and threading from the developer altogether (Cilk, OpenMP, MPI...). Some languages are designed to parallelism (Ateji PX, CUDA).

A few interpreted programming languages such as CPython and Ruby support threading, but have a limitation that is known as a Global Interpreter Lock (GIL). The GIL is a mutual exclusion lock held by the interpreter that prevents the interpreter from concurrently interpreting the applications code on two or more threads at the same time, which effectively limits the concurrency on multiple core systems.

Communicating sequential processes

In computer science, **Communicating Sequential Processes** (**CSP**) is a formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi. CSP was highly influential in the design of the occam programming language, and also influenced the design of programming languages such as Limbo and Go.

CSP was first described in a 1978 paper by C. A. R. Hoare, but has since evolved substantially. CSP has been practically applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems, such as the T9000 Transputer, as well as a secure ecommerce system. The theory of CSP itself is also still the subject of active research, including work to increase its range of practical applicability (e.g., increasing the scale of the systems that can be tractably analyzed).

History

The version of CSP presented in Hoare's original 1978 paper was essentially a concurrent programming language rather than a process calculus. It had a substantially different syntax than later versions of CSP, did not possess mathematically defined semantics, and was unable to represent unbounded nondeterminism. Programs in the original CSP were written as a parallel composition of a fixed number of sequential processes communicating with each other strictly through synchronous message-passing. In contrast to later versions of CSP, each process was assigned an explicit name, and the source or destination of a message was defined by specifying the name of the intended sending or receiving process. For example the process

```
COPY = *[c:character; west?c → east!c]
```

repeatedly receives a character from the process named `west`, and then sends that character to process named `east`. The parallel composition

```
[west::DISASSEMBLE || X::COPY || east::ASSEMBLE]
```

assigns the names `west` to the `DISASSEMBLE` process, `X` to the `COPY` process, and `east` to the `ASSEMBLE` process, and executes these three processes concurrently.

Following the publication of the original version of CSP, Hoare, Stephen Brookes, and A. W. Roscoe developed and refined the *theory* of CSP into its modern, process algebraic form. The approach taken in developing CSP into a process algebra was influenced by Robin Milner's work on the Calculus of Communicating Systems (CCS), and vice versa. The theoretical version of CSP was initially presented in a 1984 article by Brookes, Hoare, and Roscoe, and later in Hoare's book *Communicating Sequential Processes*, which was published in 1985. In September 2006, that book was still the third-most cited computer science reference of all time according to Citeseer (albeit an unreliable source due to the nature of its sampling). The theory of CSP has undergone a few minor changes since the publication of Hoare's book. Most of these changes were motivated by the advent of automated tools for CSP process analysis and verification. Roscoe's *The Theory and Practice of Concurrency* describes this newer version of CSP.

Applications

An early and important application of CSP was its use for specification and verification of elements of the INMOS T9000 Transputer, a complex superscalar pipelined processor designed to support large-scale multiprocessing. CSP was employed in verifying the correctness of both the processor pipeline, and the Virtual Channel Processor which managed off-chip communications for the processor.

Industrial application of CSP to software design has usually focused on dependable and safety-critical systems. For example, the Bremen Institute for Safe Systems and Daimler-Benz Aerospace modeled a fault management system and avionics interface (consisting of some 23,000 lines of code) intended for use on the International Space Station in CSP, and analyzed the model to confirm that their design was free of deadlock and livelock. The modeling and analysis process was able to uncover a number of errors that would have been difficult to detect using testing alone. Similarly, Praxis High Integrity Systems applied CSP modeling and analysis during the development of software (approximately 100,000 lines of code) for a secure smart-card Certification Authority to verify that their design was secure and free of deadlock. Praxis claims that the system has a much lower defect rate than comparable systems.

Since CSP is well-suited to modeling and analyzing systems that incorporate complex message exchanges, it has also been applied to the verification of communications and security protocols. A prominent example of this sort of application is Lowe’s use of CSP and the FDR refinement-checker to discover a previously unknown attack on the Needham-Schroeder public-key authentication protocol, and then to develop a corrected protocol able to defeat the attack.

Informal description

As its name suggests, CSP allows the description of systems in terms of component processes that operate independently, and interact with each other solely through message-passing communication. However, the *"Sequential"* part of the CSP name is now something of a misnomer, since modern CSP allows component processes to be

defined both as sequential processes, and as the parallel composition of more primitive processes. The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators. Using this algebraic approach, quite complex process descriptions can be easily constructed from a few primitive elements.

Primitives

CSP provides two classes of primitives in its process algebra:

Events
: Events represent communications or interactions. They are assumed to be indivisible and instantaneous. They may be atomic names (e.g. *on*, *off*), compound names (e.g. *valve.open*, *valve.close*), or input/output events (e.g. *mouse*?*xy*, *screen*!*bitmap*).

Primitive processes
: Primitive processes represent fundamental behaviors: examples include *STOP* (the process that communicates nothing, also called deadlock), and *SKIP* (which represents successful termination)

Algebraic operators

CSP has a wide range of algebraic operators. The principal ones are:

Prefix
: The prefix operator combines an event and a process to produce a new process. For example,
$a \rightarrow P$
is the process which is willing to communicate *a* with its environment, and, after *a*, behaves like the process *P*.

Deterministic Choice
: The deterministic (or external) choice operator allows the future evolution of a process to be defined as a choice between two component processes, and allows the environment to resolve the choice by communicating an initial event for one of the processes. For example,
$(a \rightarrow P) \Box (b \rightarrow Q)$
is the process which is willing to communicate the initial events *a* and *b*, and subsequently behaves as either *P* or *Q* depending on which initial event the environment chooses to communicate. If both *a* and *b* were communicated simultaneously the choice would be resolved nondeterministically.

Nondeterministic Choice
: The nondeterministic (or internal) choice operator allows the future evolution of a process to be defined as a choice between two component processes, but does not allow the environment any control over which of the component processes will be selected. For example,
$(a \rightarrow P) \sqcap (b \rightarrow Q)$

can behave like either $(a \rightarrow P)$or $(b \rightarrow Q)$. It can refuse to accept *a* or *b*, and is only obliged to communicate if the environment offers both *a* and *b*. Nondeterminism can be inadvertently introduced into a nominally deterministic choice if the initial events of both sides of the choice are identical. So, for example,

$$(a \rightarrow a \rightarrow STOP) \Box (a \rightarrow b \rightarrow STOP)$$

is equivalent to

$$a \rightarrow ((a \rightarrow STOP) \sqcap (b \rightarrow STOP))$$

Interleaving

The interleaving operator represents completely independent concurrent activity. The process

$$P \,|||\, Q$$

behaves as both *P* and *Q* simultaneously. The events from both processes are arbitrarily interleaved in time.

Interface Parallel

The interface parallel operator represents concurrent activity that requires synchronization between the component processes – any event in the interface set can only occur when *all* component processes are able to engage in that event. For example, the process

$$P \,|[\{a\}]|\, Q$$

requires that *P* and *Q* must both be able to perform event *a* before that event can occur. So, for example, the process

$$(a \rightarrow P) \,|[\{a\}]|\, (a \rightarrow Q)$$

can engage in event *a*, and become the process

$$P \,|[\{a\}]|\, Q$$

while

$$(a \rightarrow P) \,|[\{a, b\}]|\, (b \rightarrow Q)$$

will simply deadlock.

Hiding

The hiding operator provides a way to abstract processes, by making some events unobservable. A trivial example of hiding is

$$(a \rightarrow P) \setminus \{a\}$$

which, assuming that the event *a* doesn't appear in *P*, simply reduces to

P

Examples

One of the archetypal CSP examples is an abstract representation of a chocolate vending machine, and its interactions with a person wishing to buy some chocolate. This vending machine might be able to carry out two different events, “coin” and “choc” which represent the insertion of payment and the delivery of a chocolate respectively. A machine which demands payment before offering a chocolate can be written as:

$$VendingMachine = coin \rightarrow choc \rightarrow STOP$$

A person who might choose to use a coin or card to make payments could be modelled as:

$$Person = (coin \rightarrow STOP) \Box (card \rightarrow STOP)$$

These two processes can be put in parallel, so that they can interact with each other. The behaviour of the composite process depends on the events that the two component processes must synchronise on. Thus,

$$VendingMachine \,||[\{coin, card\}]|| \, Person \equiv coin \rightarrow choc \rightarrow STOP$$

whereas if synchronization was only required on "coin", we would obtain

$$VendingMachine \,||[\{coin\}]|| \, Person \equiv (coin \rightarrow choc \rightarrow STOP) \Box (card \rightarrow STOP)$$

If we abstract this latter composite process by hiding the "coin" and "card" events, i.e.

$$((coin \rightarrow choc \rightarrow STOP) \Box (card \rightarrow STOP)) \setminus \{coin, card\}$$

we get the nondeterministic process

$$(choc \rightarrow STOP) \sqcap STOP$$

This is a process which either offers a "choc" event and then stops, or just stops. In other words, if we treat the abstraction as an external view of the system (e.g., someone who does not see the decision reached by the person), nondeterminism has been introduced.

Formal definition

Syntax

The syntax of CSP defines the "legal" ways in which processes and events may be combined. Let *e* be an event, and *X* be a set of events. Then the basic syntax of CSP can be defined as:

$$
\begin{array}{rcll}
Proc & ::= & STOP & \\
& | & SKIP & \\
& | & e \rightarrow Proc & (prefixing) \\
& | & Proc \,\Box\, Proc & (external\ choice) \\
& | & Proc \,\sqcap\, Proc & (nondeterministic\ choice) \\
& | & Proc \,|||\, Proc & (interleaving) \\
& | & Proc \,|[\{X\}]|\, Proc & (interface\ parallel) \\
& | & Proc \setminus X & (hiding) \\
& | & Proc; Proc & (sequential\ composition) \\
& | & \text{if } b \text{ then } Proc \text{ else } Proc & (boolean\ conditional) \\
& | & Proc \,\triangleright\, Proc & (timeout) \\
& | & Proc \,\triangle\, Proc & (interrupt)
\end{array}
$$

Note that, in the interests of brevity, the syntax presented above omits the $\mathbf{div}$process, which represents divergence, as well as various operators such as alphabetized parallel, piping, and indexed choices.

Formal semantics

CSP has been imbued with several different formal semantics, which define the *meaning* of syntactically correct CSP expressions. The theory of CSP includes mutually consistent denotational semantics, algebraic semantics, and operational semantics.

Denotational semantics

The three major denotational models of CSP are the *traces* model, the *stable failures* model, and the *failures/divergences* model. Semantic mappings from process expressions to each of these three models provide the denotational semantics for CSP.

The *traces model* defines the meaning of a process expression as the set of sequences of events (traces) that the process can be observed to perform. For example,

- $traces\,(STOP) = \{\langle\rangle\}$since *STOP* performs no events
- $traces\,(a \rightarrow b \rightarrow STOP) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$since the process $(a \rightarrow b \rightarrow STOP)$can be observed to have performed no events, the event *a*, or the sequence of events *a* followed by *b*

More formally, the meaning of a process *P* in the traces model is defined as $traces\,(P) \subseteq \Sigma^*$such that:

1. $\langle\rangle \in traces\,(P)$(i.e. $traces\,(P)$contains the empty sequence)

2. $s_1 \frown s_2 \in traces\,(P) \implies s_1 \in traces\,(P)$ (i.e. $traces\,(P)$ is prefix-closed)

where Σ^* is the set of all possible finite sequences of events.

The *stable failures model* extends the traces model with refusal sets, which are sets of events $X \subseteq \Sigma$ that a process can refuse to perform. A *failure* is a pair (s, X), consisting of a trace *s*, and a refusal set *X* which identifies the events that a process may refuse once it has executed the trace *s*. The observed behavior of a process in the stable failures model is described by the pair $(traces\,(P)\,, failures\,(P))$. For example,

- $failures\,((a \to STOP) \,\Box\, (b \to STOP)) = \{(\langle\rangle, \emptyset)\,, (\langle a\rangle, \{a, b\})\,, (\langle b\rangle, \{a, b\})\}$
- $failures\,((a \to STOP) \sqcap (b \to STOP)) = \{(\langle\rangle, \{a\})\,, (\langle\rangle, \{b\})\,, (\langle a\rangle, \{a, b\})\,, (\langle b\rangle, \{a, b\})\}$

The *failures/divergence model* further extends the failures model to handle divergence. A process in the failures/divergences model is a pair $(failures_\perp\,(P)\,, divergences\,(P))$ where $divergences\,(P)$ is defined as the set of all traces that can lead to divergent behavior and $failures_\perp\,(P) = failures\,(P) \cup \{(s, X)\,|s \in divergences\,(P)\}$.

Tools

Over the years, a number of tools for analyzing and understanding systems described using CSP have been produced. Early tool implementations used a variety of machine-readable syntaxes for CSP, making input files written for different tools incompatible. However, most CSP tools have now standardized on the machine-readable dialect of CSP devised by Bryan Scattergood, sometimes referred to as CSP_M. The CSP_M dialect of CSP possesses a formally defined operational semantics, which includes an embedded functional programming language.

The most well-known CSP tool is probably *Failures/Divergence Refinement 2* (*FDR2*), which is a commercial product developed by Formal Systems (Europe) Ltd. FDR2 is often described as a model checker, but is technically a *refinement* checker, in that it converts two CSP process expressions into Labelled Transition Systems (LTSs), and then determines whether one of the processes is a refinement of the other within some specified semantic model (traces, failures, or failures/divergence). FDR2 applies various state-space compression algorithms to the process LTSs in order to reduce the size of the state-space that must be explored during a refinement check.

The *Adelaide Refinement Checker* (*ARC*) is a CSP refinement checker developed by the Formal Modelling and Verification Group at The University of Adelaide. ARC differs from FDR2 in that it internally represents CSP processes as Ordered Binary Decision Diagrams (OBDDs), which alleviates the state explosion problem of explicit LTS

representations without requiring the use of state-space compression algorithms such as those used in FDR2.

The *ProB* project, which is hosted by the Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, was originally created to support analysis of specifications constructed in the B method. However, it also includes support for analysis of CSP processes both through refinement checking, and LTL model-checking. ProB can also be used to verify properties of combined CSP and B specifications.

The *Process Analysis Toolkit* (PAT) is a CSP analysis tool developed in the School of Computing at the National University of Singapore. PAT is able to perform refinement checking, LTL model-checking, and simulation of CSP and Timed CSP processes. The PAT process language extends CSP with support for mutable shared variables, asynchronous message passing, and a variety of fairness and quantitative time related process constructs such as `deadline` and `waituntil`. The underlying design principle of the PAT process language is to combine a high-level specification language with procedural programs (e.g. an event in PAT may be a sequential program or even an external C# library call) for greater expressiveness. Mutable shared variables and asynchronous channels provide a convenient syntactic sugar for well-known process modelling patterns used in standard CSP. The PAT syntax is similar, but not identical, to CSP_M. The principal differences between the PAT syntax and standard CSP_M are the use of semicolons to terminate process expressions, the inclusion of syntactic sugar for variables and assignments, and the use of slightly different syntax for internal choice and parallel composition.

CSPsim is a lazy simulator. It does not model check CSP, but is useful for exploring very large (potentially infinite) systems.

Related formalisms

Several other specification languages and formalisms have been derived from, or inspired by, the classic untimed CSP, including:

- Timed CSP, which incorporates timing information for reasoning about real-time systems
- Receptive Process Theory, a specialization of CSP that assumes an asynchronous (i.e. nonblocking) send operation
- CSPP
- HCSP
- Wright, an architecture description language
- TCOZ, an integration of Timed CSP and Object Z
- Circus, an integration of CSP and Z based on the Unifying Theories of Programming
- CspCASL, an extension of CASL that integrates CSP

Comparison with the Actor Model

In as much as it is concerned with concurrent processes that exchange messages, the Actor model is broadly similar to CSP. However, the two models make some fundamentally different choices with regard to the primitives they provide:

- CSP processes are anonymous, while actors have identities.

- CSP message-passing fundamentally involves a rendezvous between the processes involved in sending and receiving the message, i.e. the sender cannot transmit a message until the receiver is ready to accept it. In contrast, message-passing in actor systems is fundamentally asynchronous, i.e. message transmission and reception do not have to happen at same time, and senders may transmit messages before receivers are ready to accept them. These approaches may be considered duals of each other, in the sense that rendezvous-based systems can be used to construct buffered communications that behave as asynchronous messaging systems, while asynchronous systems can be used to construct rendezvous-style communications by using a message/acknowledgement protocol to synchronize senders and receivers.

- CSP uses explicit channels for message passing, whereas actor systems transmit messages to named destination actors. These approaches may also be considered duals of each other, in the sense that processes receiving through a single channel effectively have an identity corresponding to that channel, while the name-based coupling between actors may be broken by constructing actors that behave as channels.

Chapter 10

Unbounded Nondeterminism and Indeterminacy in Concurrent Computation

Unbounded nondeterminism

In computer science, **unbounded nondeterminism** or **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*. Unbounded nondeterminism became an important issue in the development of the denotational semantics of concurrency, and later became part of research into the theoretical concept of hypercomputation.

Fairness

Discussion of unbounded nondeterminism tends to get involved with discussions of *fairness*. The basic concept is that all computation paths must be "fair" in the sense that if the machine enters a state infinitely often, it must take every possible transition from that state. This amounts to requiring that the machine be guaranteed to service a request if it can, since an infinite sequence of states will only be allowed if there is no transition that leads to the request being serviced. Equivalently, every possible transition must occur eventually in an infinite computation, although it may take an unbounded amount of time for the transition to occur. This concept is to be distinguished from the local fairness of flipping a "fair" coin, by which it is understood that it is possible (in some views; it almost surely won't happen) for the outcome always to be heads.

An example of the role of fair or unbounded nondeterminism in the merging of strings was given by William D. Clinger, in his 1981 thesis. He defined a "fair merge" of two strings to be a third string in which each character of each string must occur eventually. He then considered the set of all fair merges of two strings $merge(S, T)$, assuming it to be a monotone function. Then he argued that $merge(\bot,1^\omega) \subseteq merge(0,1^\omega)$, where $\bot$ is the empty stream. Now $merge(\bot,1^\omega) = 1^\omega$, so it must be that 1^ω is an element of $merge(0,1^\omega)$, a contradiction. He concluded that:

It appears that a fair merge cannot be written as a nondeterministic data flow program operating on streams.

On the possibility of implementing unbounded nondeterminism

Edsger Dijkstra [1976] argued that it is impossible to implement systems with unbounded nondeterminism. For this reason, Tony Hoare [1978] suggested that "an efficient implementation should try to be reasonably fair."

Nondeterministic automata

Nondeterministic Turing machines have only bounded nondeterminism. Likewise sequential programs containing guarded commands as the only sources of nondeterminism have only bounded nondeterminism (Edsger Dijkstra [1976]). Briefly, choice nondeterminism is bounded. Gordon Plotkin gave a proof in his original 1976 paper on powerdomains:

Now the set of initial segments of execution sequences of a given nondeterministic program `P`, starting from a given state, will form a tree. The branching points will correspond to the choice points in the program. Since there are always only finitely many alternatives at each choice point, the branching factor of the tree is always finite. That is, the tree is finitary. Now König's lemma says that if every branch of a finitary tree is finite, then so is the tree itself. In the present case this means that if every execution sequence of `P` terminates, then there are only finitely many execution sequences. So if an output set of `P` is infinite, it must contain [a nonterminating computation].

Indeterminacy versus nondeterministic automata

William Clinger [1981] provided the following analysis of the above proof by Plotkin:

This proof depends upon the premise that if every node `x` of a certain infinite branch can be reached by some computation `c`, then there exists a computation `c` that visits every node `x` on the branch. ... Clearly this premise follows not from logic but rather from the interpretation given to choice points. This premise fails for arrival nondeterminism [in the arrival of messages in the Actor model] because of finite delay [in the arrival of messages]. Though each node on an infinite branch must lie on a branch with a limit, the infinite branch need not itself have a limit. Thus the existence of an infinite branch does not necessarily imply a nonterminating computation.

Unbounded nondeterminism and noncomputability

Spaan et al. [1989] have argued that it is possible for an unboundedly nondeterministic program to solve the halting problem; their algorithm consists of two parts defined as follows:

The first part of the program requests a natural number from the second part; after receiving it, it will iterate the desired Turing machine for that many steps, and accept or reject according to whether the machine has yet halted.

The second part of the program nondeterministically chooses a natural number on request. The number is stored in a variable which is initialized to 0; then the program repeatedly chooses whether to increment the variable, or service the request. The fairness constraint requires that the request eventually be serviced, for otherwise there is an infinite loop in which only the "increment the variable" branch is ever taken.

Clearly, if the machine does halt, this algorithm has a path which accepts. If the machine does not halt, this algorithm will always reject, no matter what number the second part of the program returns.

Arguments for dealing with unbounded nondeterminism

Clinger and Carl Hewitt have developed a model (known as the Actor model) of concurrent computation with the property of unbounded nondeterminism built in [Clinger 1981; Hewitt 1985; Hewitt and Agha 1991; Hewitt 2006b]; this allows *computations* that cannot be implemented by Turing Machines, as seen above. However, these researchers emphasize that their model of concurrent computations cannot implement any *functions* that are outside the class of recursive functions defined by Church, Kleene, Turing, *etc.*

Hewitt [2006] justified his use of unbounded nondeterminism by arguing that there is no bound that can be placed on how long it takes a computational circuit called an *arbiter* to settle. 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.

He further argued that Electronic mail enables unbounded nondeterminism since mail can be stored on servers indefinitely before being delivered, and that Communication links to servers on the Internet can likewise be out of service indefinitely. This gave rise to the Unbounded nondeterminism controversy

Hewitt's analysis of fairness

Hewitt argued that issues in fairness derive in part from the global state point of view. The oldest models of computation (e.g.. Turing machines, Post productions, the lambda calculus, etc.) are based on mathematics that makes use of a global state to represent a computational *step*. Each computational step is 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. All of these models have the property of bounded nondeterminism: if a machine always halts when started in its initial state, then there is a bound on the number of states in which it can halt.

Hewitt argued that there is a fundamental difference between choices in global state nondeterminism and the arrival order indeterminacy (nondeterminism) of his Actor model. In global state nondeterminism, a "choice" is made for the "next" global state. In arrival order indeterminacy, arbitration locally decides each arrival order in an unbounded amount of time. While a local arbitration is proceeding, unbounded activity can take place elsewhere. There is no global state and consequently no "choice" to be made as to the "next" global state.

Indeterminacy in concurrent computation

Indeterminacy in concurrent computation is concerned with the effects of indeterminacy in concurrent computation.

Computation is an area in which indeterminacy is becoming increasingly important because of the massive increase in concurrency due to networking and the advent of many-core computer architectures. These computer systems make use of arbiters which give rise to indeterminacy.

A limitation of logic programming

Patrick Hayes [1973] argued that the "usual sharp distinction that is made between the processes of computation and deduction, is misleading". Robert Kowalski developed the thesis that *computation could be subsumed by deduction* and quoted with approval "Computation is controlled deduction." which he attributed to Hayes in his 1988 paper on the early history of Prolog. Contrary to Kowalski and Hayes, Carl Hewitt claimed that logical deduction was incapable of carrying out concurrent computation in open systems.

Hewitt [1985], Hewitt and Agha [1991], and other published work argued that mathematical models of concurrency did not determine particular concurrent

computations as follows: The Actor model makes use of arbitration (often in the form of notional Arbiters) for determining which message is next in the arrival ordering of an Actor that is sent multiple messages concurrently. This introduces indeterminacy in the arrival order. Since the arrival orderings are indeterminate, they cannot be deduced from prior information by mathematical logic alone. Therefore mathematical logic can not implement concurrent computation in open systems.

The authors note that although mathematical logic cannot, in their view, implement general concurrency it can implement some special cases of concurrent computation, *e.g.,* sequential computation and some kinds of parallel computation including the lambda calculus.

Arrival order indeterminacy

According to Hewitt [2008], in concrete terms for Actor systems, typically we cannot observe the details by which the arrival order of messages for an Actor is determined. Attempting to do so affects the results and can even push the indeterminacy elsewhere. Instead of observing the internals of arbitration processes of Actor computations, we await outcomes. Indeterminacy in arbiters produces indeterminacy in Actors. The reason that we await outcomes is that we have no alternative because of indeterminacy.

It is important to be clear about the basis for the published claim about the limitation of mathematical logic. It was not just that Actors could not in general be implemented in mathematical logic. The published claim was that because of the indeterminacy of the physical basis of the Actor model, that no kind of deductive mathematical logic could escape the limitation. This became important later when researchers attempted to extend Prolog (which had some basis in logic programming) to concurrent computation using message passing.

What does the mathematical theory of Actors have to say about this? A *closed* system is defined to be one which does not communicate with the outside. Actor model theory provides the means to characterize all the possible computations of a closed Actor system using the Representation Theorem [Hewitt 2007] as follows:

The mathematical denotation denoted by a closed system S is found by constructing increasingly better approximations from an initial behavior called $\bot_S$ using a behavior approximating function $\mathbf{progression}_S$ to construct a denotation (meaning) for S as follows :

$$\mathbf{Denote}_S \equiv \sqcup_{i \in \omega} \mathbf{progression}_S^{\,i}(\bot_S)$$

In this way, the behavior of S can be mathematically characterized in terms of all its possible behaviors (including those involving unbounded nondeterminism).

So mathematical logic can characterize (as opposed to implement) all the possible computations of a closed Actor system.

A limitation of logic due to lack of information

An open Actor system s is one in which the addresses of outside Actors can be passed into s in the middle of computations so that s can communicate with these outside Actors. These outside Actors can then in turn communicate with Actors internal to s using addresses supplied to them by s. Due to limitation of the inability to deduce arrival orderings, knowledge of what messages are sent from outside would not enable the response of s to be deduced. When other models of concurrent systems (*e.g.,* process calculi) are used to implement open systems, these systems also can have behavior that depends on arrival time orderings and so cannot be implemented by logical deduction.

Prolog-like concurrent systems were claimed to be based on mathematical logic

Keith Clark, Hervé Gallaire, Steve Gregory, Vijay Saraswat, Udi Shapiro, Kazunori Ueda, etc. developed a family of Prolog-like concurrent message passing systems using unification of shared variables and data structure streams for messages. Claims were made that these systems were based on mathematical logic. This kind of system was used as the basis of the Japanese Fifth Generation Project (ICOT).

Carl Hewitt and Gul Agha [1991] argued that these Prolog-like concurrent systems were neither deductive nor logical: like the Actor model, the Prolog-like concurrent systems were based on message passing and consequently were subject to the same indeterminacy.

Logical operations and system efficiency

Hewitt maintained that a basic lesson can be learned from Prolog and the Prolog-like concurrent systems: a universal model of concurrent computation is limited by having any mandatory overhead in the basic communication mechanisms. This is an argument against including pattern-directed invocation using unification and extraction of messages from data structure streams as fundamental primitives. But compare Shapiro's survey of Prolog-like concurrent programming languages for arguments for inclusion.

Indeterminacy in other models of computation

Arbitration is the basis of the indeterminacy in the Actor model of concurrent computation. It may also play a role in other models of concurrent systems such as process calculi.

Chapter 11

Erlang (Programming Language) and Joyce (Programming Language)

Erlang (programming language)

Erlang

Paradigm	multi-paradigm: concurrent, functional
Appeared in	1986
Designed by	Ericsson
Developer	Ericsson
Stable release	R14B (September 15, 2010; 2 months ago)

Typing discipline	dynamic, strong
Major implementations	Erlang
Influenced by	Prolog
Influenced	Clojure, Scala
License	Modified MPL

Erlang is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing. For concurrency it follows the Actor model. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. The first version was developed by Joe Armstrong in 1986. It supports hot swapping, thus code can be changed without stopping a system. It was originally a proprietary language within Ericsson, but was released as open source in 1998.

While threads are considered a complicated and error-prone topic in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks.

History

The name "Erlang", attributed to Bjarne Däcker, has been understood either as a reference to Danish mathematician and engineer Agner Krarup Erlang, or alternatively, as an abbreviation of "Ericsson Language".

Erlang was designed with the aim of improving the development of telephony applications. The initial version of Erlang was implemented in Prolog.

In 1998, the Ericsson AXD301 switch was announced, containing over a million lines of Erlang, and reported to achieve a reliability of nine "9"s. Shortly thereafter, Erlang was banned within Ericsson Radio Systems for new products, citing a preference for non-proprietary languages. The ban caused Armstrong and others to leave Ericsson. The implementation was open sourced at the end of the year. The ban at Ericsson was eventually lifted, and Armstrong was re-hired by Ericsson in 2004.

In 2006, native symmetric multiprocessing support was added to the runtime system and virtual machine.

Philosophy

The philosophy used to develop Erlang fits equally well with the development of Erlang-based systems. Quoting Mike Williams, one of the three inventors of Erlang:

1. Find the right methods—Design by Prototyping.
2. It is not good enough to have ideas, you must also be able to implement them and know they work.
3. Make mistakes on a small scale, not in a production project.

Functional language

A factorial algorithm implemented in Erlang:

```
-module(fact).    % This is the file 'fact.erl', the module and the
filename MUST match
-export([fac/1]). % This exports the function 'fac' of arity 1 (1
parameter, no type, no name)

fac(0) -> 1; % If 0, then return 1, otherwise (note the semicolon ;
meaning 'else')
fac(N) -> N * fac(N-1).
% Recursively determine, then return the result
% (note the period . meaning 'endif' or 'function end')
```

A sorting algorithm (similar to quicksort):

```
%% qsort:qsort(List)
%% Sort a list of items
-module(qsort).     % This is the file 'qsort.erl'
-export([qsort/1]). % A function 'qsort' with 1 parameter is exported
(no type, no name)

qsort([]) -> []; % If the list [] is empty, return an empty list
(nothing to sort)
qsort([Pivot|Rest]) ->
    % Compose recursively a list with 'Front' for all elements that
should be before 'Pivot'
    % then 'Pivot' then 'Back' for all elements that should be after
'Pivot'
    qsort([Front || Front <- Rest, Front < Pivot])
    ++ [Pivot] ++
    qsort([Back || Back <- Rest, Back >= Pivot]).
```

The above example recursively invokes the function `qsort` until nothing remains to be sorted. The expression `[Front || Front <- Rest, Front < Pivot]` is a list comprehension, meaning "Construct a list of elements `Front` such that `Front` is a member of `Rest`, and `Front` is less than `Pivot`." `++` is the list concatenation operator.

A comparison function can be used for more complicated structures for the sake of readability.

The following code would sort lists according to length:

```
% This is file 'listsort.erl' (the compiler is made this way)
-module(listsort).
% Export 'by_length' with 1 parameter (don't care of the type and name)
-export([by_length/1]).

by_length(Lists) -> % Use 'qsort/2' and provides an anonymous function
as a parameter
   qsort(Lists, fun(A,B) when is_list(A), is_list(B) -> length(A) <
length(B) end).

qsort([], _)-> []; % If list is empty, return an empty list (ignore the
second parameter)
qsort([Pivot|Rest], Smaller) ->
    % Partition list with 'Smaller' elements in front of 'Pivot' and
not-'Smaller' elements
    % after 'Pivot' and sort the sublists.
    qsort([X || X <- Rest, Smaller(X,Pivot)], Smaller)
    ++ [Pivot] ++
    qsort([Y ||Y <- Rest, not(Smaller(Y, Pivot))], Smaller).
```

Here again, a `Pivot` is taken from the first parameter given to `qsort()` and the rest of `Lists` is named `Rest`. Note that the expression

```
[X || X <- Rest, Smaller(X,Pivot)]
```

is no different in form from

```
[Front || Front <- Rest, Front < Pivot]
```

(in the previous example) except for the use of a comparison function in the last part, saying “Construct a list of elements `X` such that `X` is a member of `Rest`, and `Smaller` is true", with `Smaller` being defined earlier as

```
fun(A,B) when is_list(A), is_list(B) -> length(A) < length(B) end
```

Note also that the anonymous function is named `Smaller` in the parameter list of the second definition of `qsort` so that it can be referenced by that name within that function. It is not named in the first definition of `qsort`, which deals with the base case of an empty list and thus has no need of this function, let alone a name for it.

Data structures

Erlang has eight primitive data types:

1. **Integers**: integers are written as sequences of decimal digits, for example, 12, 12375 and -23427 are integers. Integer arithmetic is exact and only limited by available memory on the machine.
2. **Atoms**: atoms are used within a program to denote distinguished values. They are written as strings of consecutive alphanumeric characters, the first character being a small letter. Atoms can obtain any character if they are enclosed within single quotes and an escape convention exists which allows any character to be used within an atom.
3. **Floats**: floating point numbers are represented as IEEE 754 64-bit floating point numbers. Real numbers in the range $\pm 10^{308}$ can be represented by an Erlang float.
4. **References**: references are globally unique symbols whose only property is that they can be compared for equality. They are created by evaluating the Erlang primitive `make_ref()`.
5. **Binaries**: a binary is a sequence of bytes. Binaries provide a space-efficient way of storing binary data. Erlang primitives exist for composing and decomposing binaries and for efficient input/output of binaries.
6. **Pids**: Pid is short for Process Identifier—a Pid is created by the Erlang primitive `spawn(...)` Pids are references to Erlang processes.
7. **Ports**: ports are used to communicate with the external world. Ports are created with the built-in function (BIF) `open_port`. Messages can be sent to and received from ports, but these message must obey the so-called "port protocol."
8. **Funs** : Funs are function closures. Funs are created by expressions of the form: `fun(...) -> ... end`.

And two compound data types:

1. **Tuples** : tuples are containers for a fixed number of Erlang data types. The syntax `{D1,D2,...,Dn}` denotes a tuple whose arguments are `D1, D2, ... Dn`. The arguments can be primitive data types or compound data types. The elements of a tuple can be accessed in constant time.
2. **Lists** : lists are containers for a variable number of Erlang data types. The syntax `[Dh|Dt]` denotes a list whose first element is `Dh`, and whose remaining elements are the list `Dt`. The syntax `[]` denotes an empty list. The syntax `[D1,D2,..,Dn]` is short for `[D1|[D2|..|[Dn|[]]]]`. The first element of a list can be accessed in constant time. The first element of a list is called the *head* of the list. The remainder of a list when its head has been removed is called the *tail* of the list.

Two forms of syntactic sugar are provided:

1. **Strings** : strings are written as doubly quoted lists of characters, this is syntactic sugar for a list of the integer ASCII codes for the characters in the string, thus for example, the string "cat" is shorthand for `[99,97,116]`.
2. **Records** : records provide a convenient way for associating a tag with each of the elements in a tuple. This allows us to refer to an element of a tuple by name and not by position. A pre-compiler takes the record definition and replaces it with the appropriate tuple reference.

Concurrency and distribution orientation

Erlang's main strength is support for concurrency. It has a small but powerful set of primitives to create processes and communicate among them. Processes are the primary means to structure an Erlang application. Erlang processes loosely follow the communicating sequential processes (CSP) model. They are neither operating system processes nor operating system threads, but lightweight processes somewhat similar to Java's original “green threads”. Like operating system processes (and unlike green threads and operating system threads) they have no shared state between them. The estimated minimal overhead for each is 300 words, thus many of them can be created without degrading performance: a benchmark with 20 million processes has been successfully performed. Erlang has supported symmetric multiprocessing since release R11B of May 2006.

Inter-process communication works via a shared-nothing asynchronous message passing system: every process has a “mailbox”, a queue of messages that have been sent by other processes and not yet consumed. A process uses the `receive` primitive to retrieve messages that match desired patterns. A message-handling routine tests messages in turn against each pattern, until one of them matches. When the message is consumed and removed from the mailbox the process resumes execution. A message may comprise any Erlang structure, including primitives (integers, floats, characters, atoms), tuples, lists, and functions.

The code example below shows the built-in support for distributed processes:

```
 % Create a process and invoke the function web:start_server(Port,
MaxConnections)
 ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

 % Create a remote process and invoke the function
 % web:start_server(Port, MaxConnections) on machine RemoteNode
 RemoteProcess = spawn(RemoteNode, web, start_server, [Port,
MaxConnections]),

 % Send a message to ServerProcess (asynchronously). The message
consists of a tuple
 % with the atom "pause" and the number "10".
 ServerProcess ! {pause, 10},

 % Receive messages sent to this process
 receive
         a_message -> do_something;
         {data, DataContent} -> handle(DataContent);
         {hello, Text} -> io:format("Got hello message: ~s", [Text]);
         {goodbye, Text} -> io:format("Got goodbye message: ~s",
[Text])
 end.
```

As the example shows, processes may be created on remote nodes, and communication with them is transparent in the sense that communication with remote processes works exactly as communication with local processes.

Concurrency supports the primary method of error-handling in Erlang. When a process crashes, it neatly exits and sends a message to the controlling process which can take action. This way of error handling increases maintainability and reduces complexity of code.

Implementation

The Ericsson Erlang implementation loads virtual machine bytecode which is converted to threaded code at load time. It also includes a native code compiler on most platforms, developed by the High Performance Erlang Project (HiPE) at Uppsala University. It also supports interpreting, directly from source code via abstract tree, via script as of R11B-5.

Hot code loading and modules

Code is loaded and managed as "module" units; the module is a compilation unit. The system can keep two versions of a module in memory at the same time, and processes can concurrently run code from each. The versions are referred to as the "new" and the "old" version. A process will not move into the new version until it makes an external call to its module.

An example of the mechanism of hot code loading:

```
  %% A process whose only job is to keep a counter.
  %% First version
  -module(counter).
  -export([start/0, codeswitch/1]).

  start() -> loop(0).

  loop(Sum) ->
    receive
       {increment, Count} ->
          loop(Sum+Count);
       {counter, Pid} ->
          Pid ! {counter, Sum},
          loop(Sum);
       code_switch ->
          ?MODULE:codeswitch(Sum)
          % Force the use of 'codeswitch/1' from the latest MODULE
version
    end.

  codeswitch(Sum) -> loop(Sum).
```

For the second version, we add the possibility to reset the count to zero.

```
%% Second version
-module(counter).
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
  receive
     {increment, Count} ->
        loop(Sum+Count);
     reset ->
        loop(0);
     {counter, Pid} ->
        Pid ! {counter, Sum},
        loop(Sum);
     code_switch ->
        ?MODULE:codeswitch(Sum)
  end.

codeswitch(Sum) -> loop(Sum).
```

Only when receiving a message consisting of the atom 'code_switch' will the loop execute an external call to codeswitch/1 (`?MODULE` is a preprocessor macro for the current module). If there is a new version of the "counter" module in memory, then its codeswitch/1 function will be called. The practice of having a specific entry-point into a new version allows the programmer to transform state to what is required in the newer version. In our example we keep the state as an integer.

In practice, systems are built up using design principles from the Open Telecom Platform which leads to more code upgradable designs. Successful hot code loading is a tricky subject; code needs to be written to make use of Erlang's facilities.

Distribution

In 1998, Ericsson released Erlang as open source to ensure its independence from a single vendor and to increase awareness of the language. Erlang, together with libraries and the real-time distributed database Mnesia, forms the Open Telecom Platform (OTP) collection of libraries. Ericsson and a few other companies offer commercial support for Erlang.

Since the open source release, Erlang has been used by several firms worldwide, including Nortel and T-Mobile. Although Erlang was designed to fill a niche and has remained an obscure language for most of its existence, its popularity is growing due to demand for concurrent services.

Erlang is available for many Unix-like operating systems, including Mac OS X, and for Microsoft Windows.

Projects using Erlang

Projects using Erlang include:

- CouchDB, a document based database that uses MapReduce
- ejabberd, an Extensible Messaging and Presence Protocol (XMPP) instant messaging server
    - Facebook chat system, based on ejabberd
- Erlyvideo, high efficiency multiprotocol videostreaming server
- GitHub egitd, a replacement for stock git-daemon that ships with Git
- Issuu, an online digital publisher
- Membase, database management system optimized for storing data behind interactive web applications.
- RabbitMQ, an implementation of Advanced Message Queuing Protocol (AMQP)
- SimpleDB, a distributed database that is part of Amazon Web Services
- Twitterfall, a service to view trends and patterns from Twitter
- Wings 3D, a 3D modeller
- Yaws web server

Clones

Erlang has inspired clones of its concurrency facilities for other languages:

- Reia
- Scala

Joyce (programming language)

Joyce	
Paradigm	concurrent, imperative, structured
Appeared in	1987
Designed by	Brinch Hansen
Stable release	1 (1987)
Typing discipline	Strong
Influenced by	Communicating Sequential Processes,

Pascal, Concurrent Pascal

Influenced SuperPascal

Joyce is a secure, concurrent programming language designed by Per Brinch Hansen in the 1980s. It is based on the sequential language Pascal and the principles of Communicating Sequential Processes (CSP). It was created to address the shortcomings of CSP to be applied itself as a programming language, and to povide a tool, primarily for teaching, for distributed system implementation.

The language is based around the concept of *agents*; concurrently executed pocesses that communicate only by the use of channels and message passing. Agents may activate sub-agents dynamically and recursively. The development of Joyce formed the foundation of the language SuperPascal, also developed by Brinch Hansen around 1993.

Features

Joyce is based on a small subset of Pascal, extended with features inspired from CSP for concurrency. The following sections describe the some of the more novel features that were introduced.

Agents

An agent is a procedure consisting of a set of statements and possibly nested definitions of other agents. An agent may dynamically *activate* sub-agents which execute concurrently with their *creator*. An agent can terminate only when all of its sub-agents have also terminated. For example, an agent `process2` activates `process1`:

```
agent process1(x, y: integer);
begin
    ...
end;

agent process2();
use process1;
begin
    process1(9, 17);
end;
```

The activation of an agent creates new instances of all local variables and the value of each formal parameter is copied to a local variable. Hence, agents cannot access variables of other agents and are allowed only to communicate through the use of channels. This restriction prevents problems associated with the use of shared variables such as race conditions.

Communication

Agents communicate through entities called *channels*. Channels have an alphabet, defining the set of symbols which may be transmitted. Channels are created dynamically and accessed through the use of *port* variables. A port type is defined by a distinct set of symbols constituting its alphabet. Symbols with multiple values are defined with a specific type. For example:

```
stream = [int(integer), eos];
```

The symbol `int(integer)` denotes a *message* symbol called `int` of any integer value. The second typeless symbol declaration `eos` (end of stream) is known as a *signal*. Once a port type has been defined, a port variable of that type can be declared:

```
out : stream
in  : stream
```

And then a channel entity, internal to the agent creating it, can be activated as follows:

```
+out;
```

Symbols can then be sent and received on channels using the CSP-style input and output operators `?` and `!` respectively. A communication can only occur if there is a receiving agent matching the sending agent. The receiving agent must expect to receive the symbol type being sent. For example, the value 9 followed by the `eos` symbol is sent on port `out`:

```
out ! int(9)
out ! eos
```

And an integer message is received into a variable of a matching type, followed by the `eos`:

```
received : integer
in ? int(received)
in ? eos
```

Polling statements

Polling statements are based the CSP concept of guarded alternatives. A polling statement is made up of a set of statements, each guarded by an input channel statement. When a communication is matched between a transmitting agent and a guard, the guard is executed, followed by the corresponding statement. For example:

```
poll
    in ? X -> x := x + 1 |
    in ? Y -> y := y + 1
end
```

Where the port `in` is monitored for the signals `X` or `Y`, on a matching communication, the corresponding variables `x` or `y` are incremented.

Security

Joyce was designed to be a *secure* language in the sense that a compiler would be able to detect all violations of the language rules.

Example program

The following is a complete example program, taken from the original paper introducing the Joyce programming language, implementing an algorithm to generate prime numbers based on a sieving technique. A `sieve` agent is sent a stream of integers from its predecessor, the first being a prime. It removes all multiples of this prime from the stream and activates a successor. This continues until the `eos` signal is propagated along the set of sieves.

```
agent sieve(inp, out: stream);
var more: boolean; x, y: integer;
  succ: stream;
begin
    poll
        inp?int(x) -> +succ;
            sieve(succ, out); more := true |
        inp?eos    -> out!eos; more := false
    end;
    while more do
        poll
            inp?int(y) ->
                if y mod x <> 0 then succ!int(y) |
            inp?eos    -> out!int(x);
                succ!eos; more := false
        end;
end;
```

The following agent initialises the set of sieve agents and inputs into them a stream of integers between 3 and 9999.

```
agent primes;
use generate, sieve, print;
var a, b: stream;
begin
    +a; +b; generate(a, 3, 2, 4999);
    sieve(a, b); print(b)
end;
```

Implementation

Stack allocation

Due to concurrent execution of agent procedures, a conventional sequential stack allocation scheme cannot be used as the activation records of the agent calls do not follow a last-in first-out pattern. Instead, the creator-subagent relationships form a tree-structured stack. A simple scheme is used to implement this behaviour, which works by allocating new activation records at the top of the stack, and linking subagents' activation records to their creator's record. These records are freed only when the agent has terminated and they are at the top of the stack. The effectiveness of this scheme depends on the structure and behaviour of a program, which in some cases will result in poor memory usage. A more effective scheme was implemented in the SuperPascal language.