Concurrent/Parallel Processing

David May: April 9, 2014

Introduction

The idea of using a collection of interconnected processing devices is not new.

Before the emergence of the modern stored program computer, complex problems were sometimes solved by a room full of human 'computers', passing data between each other on slips of paper.

John Von-Neumann studied the properties of large arrays of simple devices (cellular automata) and demonstrated self-reproducing patterns; it was subsequently shown that these arrays can implement Turing machines.

Over the last 50 years a wide variety of systems based on collections of processors have been developed and today such systems are used in products ranging from smartphones to supercomputers.

A note on terminology

The terms *concurrent* and *parallel* are often used interchangeably. Sometimes *parallel* is used to mean that the processors operate in step, all performing the same operation at the same time. Similarly, *concurrent* is sometimes used to mean that the processors operate independently, performing different operations.

Limits to sequential processing

The success of the Turing/Von-Neumann architecture has been largely due to its ability to scale; larger and larger problems can be solved by scaling up the memory and/or by executing more instructions. No change to the architecture is needed and in many cases no change to the program is needed.

This has been supported by technology scaling; as transistors are reduced in size, they operate faster and more of them can be used to increase memory capacity.

However, there are limits. As wires are reduced in width, they become slower, introducing delays in signalling between components such as the processor and its memory. And transistor operation ultimately become unreliable.

Key ideas in concurrent processing

It is useful to think of a concurrent system (hardware or software) as a collection of *processes* communicating with each other by sending and receiving *messages*. This gives rise to a number of useful structures:

- Graphs: a collection of processes connected via channels over which messages are sent forms a directed graph. The channels may provide either one-directional or bi-directional communication.
- Pipelines: data is fed through a series of processes, each of which performs some operation on it.
- Fat pipelines: a pipeline can fork into a number of identical sub-pipelines which re-join. Data arriving at the fork can be distributed to any of the sub-pipelines. The results from each sub-pipeline are merged at the re-join. It may be necessary to ensure that the order of the results corresponds to the order of the data entering the pipeline.
- Task farms: a process distributes sub-tasks to a collection of identical worker processes. As each worker completes its sub-task, it returns the result to the distributor and waits for a new task.
- Server farms: these are similar to task farms but the workers hold data structures and process requests to access and modify data. It is possible for there to be multiple clients.
- Domain decomposition: a collection of processes execute the same program on different segments of a large data structure.

These structures may be used alone or in combination to construct concurrent systems. In concurrent programs, they may be embedded within sequential or recursive program structures to enable introduction (and removal) of concurrency as the program executes.

Pipelines in sequential processors

It is possible to speed up execution of instructions in a sequential processor using a pipeline. In a simple case, the execution of an instruction is broken up into four stages:

- Fetch: fetches instructions from the memory/cache.

- Decode: decodes instructions and reads operands from the registers.
- Execute: executes instructions.
- Writeback: writes results back to the registers.

Each stage is performed by a corresponding hardware module and registers are used between the stages to hold the data associated with the instruction.

If the pipeline is clocked, a new instruction enters the pipeline and a completed one leaves it at each cycle. Note that it is possible for such a pipeline to operate without a clock, provided that a handshake is used to pass data from one stage to the next.

The pipeline will execute up to four instructions at the same time:

				Time →				
1:	Fetch	Decode	Execute	Write				
2:		Fetch	Decode	Execute	Write			
3:			Fetch	Decode	Execute	Write		
4:				Fetch	Decode	Execute	Write	
5:					Fetch	Decode	Execute	Write

Processors that use this technique normally have a set of addressable registers (for example, the ARM has 16) and instructions that specify them explicitly. In the instruction decode stage, a check is made to determine if there are dependencies between instructions such as that introduced by r3 in:

```
ADD r3, r2, r1
ADD r4, r3, r4
```

This is dealt with either by delaying the second instruction, causing a pipeline *bubble* and loss of performance, or by providing an additional *forwarding* path to allow the output of the execute stage to re-enter as an input to the execute stage at the next cycle.

A further problem is branches; if a branch is encountered the instructions following the branch will be discarded and the pipeline will start to fill again with instructions from the new program location (the branch target).

As branches are very common (typically occurring every 5-10 instructions), it is normal to include hardware to fetch instructions ahead of when they are needed so that a branch can be identified early, the new program address calculated and the new instructions fetched. Also it is common to use a *branch predictor* to try to guess the outcome of a conditional branch.

Note that pipelines can have many more than four stages. Additional stages may be needed to support memory/cache access and complex arithmetical operations such as floating point. In Intel processors several stages are used to translate x86 instructions into an internal form (*micro-ops*) suitable for pipelined execution.

Fat pipelines in sequential processors

It is possible to increase the width of the pipeline in a sequential processor by providing multiple execution units. A typical arrangement is for the fetch and decode stages to operate on groups of four instructions; these are then sent (issued) to four execution units at the same time. This is known as *superscalar* processing.

The execution units are themselves pipelined. The instructions proceed through the execution pipelines at speeds depending on the operations being performed; for example delays may arise because of dependencies, cache misses or complex instructions such as division. The instructions will leave the execution pipelines in any order. However, it is normal to use a fat pipeline that preserves ordering and in a superscalar processor, the device that performs this is known as a *re-order buffer* that collects the results and writes them back to the registers in the same order as the corresponding instructions in the program. If this were not done, the results of the instructions would appear in the registers in a different order from that specified by the instruction sequence in the program. This would make the behaviour difficult to understand when an instruction resulting in an error (trap) occurs; some instructions before the error would not have produced results but some instructions after the error would have.

These techniques enable many instructions to be executed at the same time and make branch prediction even more important. A variety of schemes have been devised both to predict branch outcomes and to recover in the case of an incorrect prediction.

They also require a memory hierarchy capable of delivering several instructions at once as well as performing data accesses. Typically a pair of L1 caches are used, one for instructions and one for data. These are accompanied by a larger L2 cache holding both program and data.

An important issue when designing a superscalar processor is that at every stage, the instruction processing hardware must be able to operate on several instructions. If four instructions are fetched in parallel, four must be decoded in parallel, four must be issued in parallel, four must be executed in parallel and four must be written back in parallel. It is useful to analyse this by imagining that the processor has been *bisected* (cut in half) at each stage and working out how much data (and how many instructions) can flow from one half to the next.

Multiple processors

It has become common to include two or more processors on a chip, sharing access to the memory system. Typically each processor has its own L1 and L2 caches, but the processors share a large on-chip L3 cache.

This gives rise to a problem of *cache coherency*. Two processors P1 and P2 may both have a copy of a memory location in their L1 cache; if P1 writes to the location and then P2 reads it, it is important that P2 reads the most recent value, not its cached copy of the original value.

There are various schemes to maintain cache coherency. An example is as follows.

- The L1 and L2 aches are implemented as write-through. This means that any write will cause the written data to be written into the L3 cache.
- The L1 and L2 caches are arranged to observe *all* writes into the L3 cache by other processors (this is known as *snooping*). If a cache observes a write to an address that matches the address in one of its current entries, it invalidates the entry.

In the above example, the value written to the memory location by P1 will be written into the L3 cache. This write will be observed by the L2/L1 caches of P2, which will invalidate their copies of the location. When P2 now reads the location, it will be loaded from the L3 cache.

Normally, additional mechanisms (such as inter-processor interrupts) are provided to signal between the processors to avoid the need to continually read a shared location when performing communication between programs executed by different processors.

Data parallelism

Data parallelism has become an important technique to optimise execution of graphics and signal processing applications. It is provided as an extension to a conventional processor in one of two ways:

- In a *coprocessor*, sharing access to the memory system. A number of execution units are supplied with the same instruction sequence and all perform the same operation at the same time (but on different data). In some systems, there is a limited capability for conditional execution, usually by providing each execution unit with a condition register such that it will only execute the current instruction if the value of the condition register is *true*.

- As an additional set of instructions which treat the contents of each register in a conventional processor as a set of values. For example, on a processor with 64-bit registers, it is possible to provide instructions that treat the register contents as 8 8-bit values, 4 16-bit values or 2 32-bit values. Then, for example, an ADD8 instruction would add the corresponding pairs of 8-bit values from two source registers to produce the 8-bit values in a result register.

This is an example of the use of the domain decomposition technique noted above.

Scalable Multiprocessors

For systems with many interconnected processors, a key architectural issue is *scaling*. Ideally, performance should scale linearly with the number of processors used. In executing a concurrent program, all of the concurrent processors may need to communicate at the same time. This means that the interconnection network must be able to perform concurrent communication and that its performance should scale linearly with the number of processors.

It is not practical to connect every processor directly to every other processor; even for 100 processors this would require 9,900 communication links. Instead, networks are constructed using *switches* to route messages from their source to their destination. The use of switches reduces the number of connections but increases the network *latency* (the time for a message to be transmitted through the network). For realistic networks connecting p processors, this latency scales as $log(p)$.

The increase in latency can be *hidden*, for example by giving each processor several processes to execute (instead of just one). Whenever a process is delayed waiting for a message, other processes are available for execution.

For a network with latency $log(p)$, each message will use $log(p)$ network capacity in traversing it. So to support p concurrent messages, the network capacity will need to be $p \times log(p)$; the cost of the network grows faster than the cost of the processors. This ultimately limits the use of parallelism to increase performance. However, several network structures which can be scaled to connect large numbers of processors do have capacity growing as $p \times log(p)$.

An important practical issue is to choose an appropriate balance between communication throughput and processing throughput, both for each processor and for the system as a whole. If the processor can maintain a ratio of 1:10 (for example communicating at 1 billion operands/second and computing at 10 billion operations/second), then the ratio of network throughput to total processing throughput

should also be 1:10. Once this ratio is fixed, it will determine the extent to which parallelism can be used to increase performance.

A simple example illustrates this. A two dimensional array is to be processed by performing an operation in which each element is replaced by a function of its current value and those of its 4 neighbouring elements, involving a total of about 10 arithmetic operations. The array can be cut into tiles of size $n \times n$ and each tile given to a processor. Then each processor will perform $10n^2$ operations and $4n$ communications. For a 1:10 ratio, to keep all of the processors busy, $(4n \times 10) < 10n^2$, or $n > 4$. If $n < 4$, the processors will idle some of the time waiting for communications.

Interconnection Networks

A variety of network structures have been used (and are being used) for concurrent computing. These include:

- n-dimensional grids. Typically $n = 2$ or $n = 3$. These do not inherently provide the required capacity for scaling but are relatively easy to construct. For scaling, it is necessary to increase the capacity of the network links as the network size increases, or to add additional networks operating in parallel.
- toroidally connected n-dimensional grids. The toroidal connection improves performance by directly linking the switches on each edge to those on the opposite edge. For scaling, it is necessary to increase the capacity of the network links as the network size increases, or to add additional networks operating in parallel.
- hypercubes. These provide the required capacity for scaling.
- indirect networks. These are variations of Clos networks, originally developed for telephone switching. They can be easily be configured to provide the required capacity for scaling. There are effective ways to implement them using high capacity links and switches in the network core (this structure is sometimes known as a fat-tree).

An important issue when choosing a network structure is the potential throughput, especially in the case where every processor is communicating at the same time. It is useful to analyse this by imagining that the has been *bisected* (cut in half) and working out how much data can flow from one half to the other.

Protocols and routing

The network links use a protocol that provides flow-control to ensure that the receiver has space for incoming messages; it may also provide for error detection and/or correction. Sometimes it allows several messages to flow at the same time, sharing the link.

When a message arrives at a switch, the first few bits (or bytes) are used to determine the outgoing link to be used. Some switches provide for the message to be forwarded via the outgoing link whilst the remainder of the message is still being received. This is known as *wormhole* routing.

An important issue is *deadlock freedom*. A deadlock occurs when two or more concurrent processes are waiting for each other. It is possible for this to happen when routing messages in a network, but for all of the network structures above, there are deadlock-free routing techniques. For example, in a two-dimensional grid, routing messages first left-right and then top-bottom will prevent deadlocks; this extends to n-dimensional grids.

Interconnect Performance

As noted above, a concurrent interconnect should be able to support scalable concurrent communication. An important case is where every processor sends a message to another processor, with no two messages sent to the same processor. This is often referred to as a *permutation routing* operation because the destinations $(d_0, d_1, ...d_n)$ of the messages is a permutation of the sources $(s_0, s_1, ...s_n)$ of them.

Some communication patterns can give rise to congestion at certain points in the interconnect. Suppose that a two-dimensional grid of processors is used and each holds a tile of a two-dimensional array. If the array is transposed, and the messages move first left-right and then top-bottom, each node along the diagonal will carry all of the messages originating in the row and column that it occupies. This congestion will reduce performance.

One way to avoid congestion is *randomisation*; messages are first sent to a randomly chosen intermediate point from which they continue to their final destination. Used in conjunction with networks with sufficient capacity (such as Clos networks and hypercubes), this will eliminate the congestion points and provide scalable communication.

Sharing Data

An important use of scalable parallel computers is manipulation of large data sets. It is often (incorrectly) assumed that this requires parallel computers based on shared memory (in which all processors have read/write access to a global memory). However, for large systems, access to large shared data sets is normally implemented using message passing and data-farms.

Many of the ideas from shared memory computing are still applicable, especially the parallel random access machine (PRAM) models:

- Exclusive Read Exclusive Write (EREW): No two processors access the same address at the same time.
- Concurrent Read Exclusive Write (CREW): Several processors may read the same address at the same time, but no two processors may write to the same address at the same time.
- Concurrent Read Concurrent Write (CRCW): Several processors may access the same address at the same time.

For any of these, it is possible that many accesses are made to the same server (but to different data) at the same time, causing congestion and reducing performance. The effects of this can be minimised by using hash functions to distribute data across the servers.

For the CREW and CRCW models, it is also possible that many accesses are made to the same data at the same time. This will potentially result in severe congestion (similar to a denial-of-service attack!). The effect can be minimised by combining the requests as they converge towards the location of the data and only forwarding a single request. This may require a layer of server processes in front of the processes forming the data-farm.

Current Issues

For several years, there has been no significant increase in the clock speeds of processors, with the result that concurrency has become essential for increased performance. Today, the number of processors per chip (or system) is growing rapidly. The main issues are:

- Communication between processors and memory remains a bottleneck. One promising solution, especially for communication between processors and

memory, is the use of *stacked* chips. This allows a large number of connections to be made vertically between a chip full of processors and a chip full of high-density memory. Also, because this can operate at higher throughput than a normal memory interface, the memory hierarchy can be simplified by removing some of the caches.

- Communication between chips is a bottleneck, especially in general purpose systems. An interesting possibility is *silicon photonics*, which enables optical devices such as modulators and demodulators to be combined with electronics on silicon chips. This enables inter-chip communications to be performed optically, which is potentially much more efficient than electronics. It is likely that communication chips will be stacked, as outlined above.

- Energy consumption has become a problem, and is likely to remain one. It affects everything from the battery life of mobile devices to the cost of running supercomputers. This can to some extent be improved by architecture but by far the largest gains would come from more efficient software and algorithms.

- Programming languages have not caught up. There are no widely used concurrent programming languages. Most systems are currently programmed using specialised languages and/or libraries.