Multi-core architectures and their software landscape

Raphael ‘kena’ Poss

Note: The following text will be published in the *Computing Science Handbook*, Volume 1. This document is a draft copy. Do not distribute.

1 Introduction

In the decade 1990-2000, processor chip architectures have benefited from tremendous advances in manufacturing processes, enabling cheap performance increases from both increasing clock frequencies and decreasing gate size on silicon. These advances in turn enabled an explosive expansion of the software industry, with a large focus on computers based on general-purpose uni-processors. This architecture model, that of the Von Neumann computer, had emerged at the end of the 1980's as the *de facto* target of all software developments.

Until the turn of the 21st century, system engineers using uni-processors as building blocks could assume ever-increasing performance gains, by just substituting a processor by the next generation in new systems. Then they ran into two obstacles. One was the *memory wall* [45], i.e. the increasing divergence between the access time to memory and the execution time of single instructions. To overcome this wall, architects have designed increasingly complex uni-processors using techniques such as branch predictors and out-of-order execution to automatically find parallelism in single threaded programs and keep processor pipelines busy during memory accesses. The second is the *sequential performance wall* [1, 34], also called “Pollack's rule” [31], i.e. the increasing divergence in single processors between performance gains by architectural optimizations and the power-area cost of these optimizations.

To “cut the Gordian knot,” in the words of [34], the industry has since (post-2000) shifted towards multiplying the number of processors on chip, creating increasingly larger Chip Multi-Processors (CMPs) by processor counts, now called *cores*. The underlying motivation is to *exploit explicit concurrency* in software and distribute workloads across multiple processors to increase performance. The responsibility to find parallelism was pushed again to the software side, where it had been forgotten for fifteen years.

During the period 2000-2010, this shift to multi-core chips has caused a commotion in those software communities that had gotten used to transparent frequency increases and implicit Instruction-Level Parallelism (ILP) for sequential programs without ever questioning the basic machine model targeted by programming languages and complexity theory. “The free lunch is over” [40], and software ecosystems then had to acknowledge and understand explicit on-chip parallelism and energy constraints to fully utilise current and future hardware.

This transition was disruptive for audiences used to systems where the processor fetches instructions one after another following the control flow of *one program*. Yet the commotion was essentially specific to those traditional audiences of general-purpose uni-processors that had grown in the period 1990-2000. In most application niches, application-specific knowledge about available parallelism had long mandated dedicated support from the hardware and software towards increased performance: scientific and high-performance computing have long exploited dedicated Single Instruction, Multiple Data (SIMD) units, embedded applications routinely specialise components to program features to reduce logic feature size and power requirements, and server applications in data centres have been optimised towards servicing independent network streams, exploiting dedicated I/O channels and Hardware Multi-Threading (HMT) for throughput scalability. Moreover, a host of research on parallel systems had been performed in the previous period, up to the late 1980's, and best practices from this period are now surfacing in the software industry again.

In the rest of this chapter, we review the development of multi-core processor chips during the last decade and their upcoming challenges.

2 Underlying principles

2.1 Multi-core architecture principles

Two observations from circuit design have motivated the transition to multi-core processor chips.

As noted by [4], the scalability of multiple-instruction issue in conventional processors is constrained by the fact that ILP is not improved linearly with the addition of silicon. Scaling up implicit concurrency in superscalar processors gives very large circuit structures. For example, the logic required for out-of-order issue scales quadratically with issue width [22] and would eventually dominate the chip area and power consumption. This situation had been summarised by Pollack [31] by stating that the performance of a single core increases with the square root of its number of transistors.

Meanwhile, the power cost of single-core ILP is disadvantageous. Not only does Pollack's rule suggest more power consumption due to the growing silicon cost per core; to increase the Instructions Per Second (IPS) count, the processor's clock frequency must also increase. As noted in [34], Maximum power consumption ($Power$) is increased with the core operating voltage (V) and frequency (F) as follows:

$$Power = C \times V^2 \times F$$

where C is the effective load capacitance of all units and wires on core. Within some voltage range, F may go up with supply voltage V ($F = k \times V^{\alpha-1}$, $\alpha \leq 1$). This is a good way to gain performance, but power is also increased (proportional to $V^{2+\alpha}$). For a given core technology, this entails that a linear increase in IPS via frequency scaling requires at least a quadratic increase in power consumption.

From this circuit perspective, the advantage of explicit parallelism by investing transistor counts towards multiple, simpler cores becomes clear: assuming available concurrency in software, two cores running at half the frequency can perform together the same IPS count at less than half the power usage. Moreover, by keeping the cores simple, more transistors are available to increase the core count on chip and thus maximum IPS scalability.

This perspective also reveals the main challenge of multi-core chips: the purported scalability is strongly dependent on the ability of software to exploit the increasing core counts. This is the issue of programmability, covered in the rest of this chapter.

Beyond issues of power and performance, another factor has become visible in the last decade: fault management.

Both transient and permanent faults can be considered. Transient faults are caused mostly by unexpected charged particles traversing the silicon fabric, either emitted by atomic decay in the fabric itself or its surrounding packages, or by cosmic rays, or by impact from atmospheric neutrons; as the density of circuits increases, a single charged particle will impact more circuits. Permanent faults are caused by physical damage to the fabric, for example via heat-induced stress on the metal interconnect or atomic migration. While further research on energy efficiency will limit heat-induced stress, atomic migration unavoidably entails loss of function of some components over time. This effect increases as the technology density increases because the feature size, i.e. the number of atoms per transistor/gate, decreases.

To mitigate the impact of faults, various mechanisms have been used to hide faults from software: redundancy, error correction, etc. However, a fundamental consequence of faults remains: as fault tolerance kicks in, either the *latency changes* (e.g. longer path through the duplicated circuit or error correction logic) or the *throughput changes* (e.g. one circuit used instead of two).

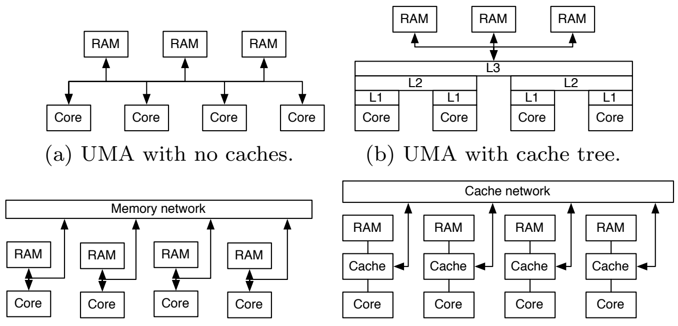
To summarise, the increasing number of faults is a source of unavoidable *dynamic heterogeneity* in larger chips. Either components will appear to enter or leave the system dynamically, for example when a core must stop due to temporary heat excess, or their characteristics will appear to evolve over time beyond the control of applications. This in turn requires to evolve the abstract model of the chip that programmers use when writing software.

Exposing the chip's structure in abstract models as a network of loosely coupled cores, i.e. a *distributed system on chip*, instead of a tightly-coupled "central processing unit," will facilitate the management of this dynamic heterogeneity in software.

2.2 Multi-core architecture models

Any computing system today is a composition of memories, caches, processors, I/O interfaces and data links as "basic blocks." A given architecture is a concrete assembly of these components. In contrast an *architecture model* is an abstract description of how components are connected together, to capture the general properties common to multiple concrete architectures. Architecture models are in turn characterised by their *typology*, their *topology* and the *concurrency management primitives* they expose at the hardware/software interface.

The *typology* reports which different types of components are used. In multi-core chips, we can distinguish *homogeneous* or "symmetric" designs, where all repeated components have the same type, from *heterogeneous* designs which may use e.g. general-purpose cores in combination with on-chip accelerators,



(a) UMA with no caches. (b) UMA with cache tree.

(c) NUMA with no caches. (d) NUMA with caches.

Figure 1: Example memory and cache topologies.

cores with different cache sizes, or cores with different instruction sets. Historically homogeneous designs have been preferred as they were simpler to program: most multi-cores prior to 2010 were so-called Symmetric Multi-Processor (SMP) systems. However, some heterogeneity has since become mandatory: not only as a way to manage faults and energy usage, as highlighted above, but also to increase execution efficiency for fixed applications by specialising some functional units to the most demanding software tasks.

The *topology* indicates how components are connected together. In multi-cores, the topology determines how cores can communicate with each other and the latency and bandwidth of communication. We can distinguish three design spectra for topologies.

The first is the *memory topology*, i.e. how cores are connected to main memory. At one end of this spectrum, “UMAs” describe interconnects where a single memory is shared symmetrically by all cores, and hence provides homogeneous latency/bandwidth constraints. At the other end, individual or groups of cores have their own local memory, and remote accesses become orders of magnitude more expensive than local accesses. This is where one finds “NUMAs.” The position of any specific architecture on this spectrum is a cost trade-off: UMAs are simpler to program but require more silicon and energy to provide increased bandwidth to the shared memory.

The second spectrum is the *cache topology*, i.e. how caches are connected to cores, memory and other caches. At one end of this spectrum, congruent with UMAs, the cache topology can be represented as a *cache tree*, where the common memory forms the root of the tree and the cores form the leaves. This topology fully preserves the notion of “cache hierarchy” from the perspective of individual cores, where there is only one path to main memory and local performance is determined only by the hit and miss rates at each level of requests emitted locally. At the other end, congruent with NUMAs, more diversity exists. With *distributed coherent cache* architectures, a network protocol between caches ensures that updates by one part of the system are consistently visible from all other parts. These are also called Cache-Coherent NUMAs (ccNUMAs) or Distributed Shared Memory (DSM) architectures. Again, the choice is a cost trade-off: cache trees provide a shorter latency of access to main memory on average, but cost more silicon and energy to operate than loosely coupled caches. Example memory and cache topologies are given in Figure 1.

The third spectrum is the *inter-core topology*, i.e. what direct links are available between cores for direct point-to-point communication. In most multi-cores, regardless of the memory architecture a dedicated *signalling network* is implemented to notify cores asynchronously upon unexpected events. The exchange of notifications across this network is commonly called Inter-Processor Interrupt (IPI) delivery. Inter-core Networks-on-Chip (NoCs) also exist that offer arbitrary data communication between cores; 2D mesh topologies are most common as they are cheap to implement in tiled designs.

Finally, the *concurrency management primitives* determine how software can exploit the hardware parallelism. There are three aspects to this interface: control, synchronisation, and data movement.

In most designs that have emerged from the grouping of cores previously designed for single-core execution, such as most general-purpose SMP chips in use today, the interface for control and synchronisation is quite basic. For control, cores execute their flow of instructions until either a “halt” instruction is encountered, or an IPI is delivered that stops the current instruction flow and starts another. The only primitive for inter-core control is IPI delivery and regular load/store operations to a shared memory. In this context, the only synchronisation mechanisms available to software are either *busy loops* that access a memory location until its contents are changed by another core, or *passive waiting* that stops the current core, to be awakened by a subsequent IPI from another core. Additional mechanisms may be present but are still unusual. For example, *hardware primitives for synchronisation barriers* may be available, whereby two or more cores that execute a barrier will automatically wait for one another.

For data movement, the near universal primitives are still memory load and store instructions: using a shared memory, programs running across multiple cores can *emulate virtual channels by using buffer data structures* at known common locations. In addition to loads and stores, dedicated *messaging primitives* may exist to send a data packet to a named target core or wait upon reception of a packet, although they are still uncommon.

In any case, whichever primitives are available are typically abstracted by the operating systems in software to present a standardised programming interface to applications, such as those described in the remainder of this chapter. Thanks to this abstraction layer, most of the diversity in concurrency management interfaces is hidden to application programmers. However, it is still often necessary to obtain knowledge about which underlying primitives are provided by an architecture to understand its cost/performance trade-offs.

2.3 Multi-cores with intra-core parallelism

Independently and prior to the introduction of multi-cores, architects had enhanced individual cores to offer internal parallelism. The purpose of internal parallelism is to increase utilisation of the processor pipeline, by enabling the overlap of computations with waiting instructions such as I/O or memory accesses. When cores with internal parallelism are combined together to form a multi-core chip, two scales of parallelism exist and their interaction must thus be considered.

Out-of-order execution

In processors with Out-of-Order Execution (OoOE), instructions wait at the start of the pipeline until their input operands are available, and ready instructions are allowed to enter the pipeline in a different order than program order. Result order is then restored at the end of the pipeline. The key concept is that missing data operands do not prevent the pipeline from executing unrelated, ready instructions, and *utilisation* is increased. For more details, see [12, Chap. 2&3] and the Chapter on Performance enhancements in this volume.

OoOE introduces new challenges for multi-core synchronisation. For example, a common idiom is to place a computation result into a known memory cell, then write a flag "the result has been computed" into another. If memory stores are performed in order, another core can perform a busy loop, reading the flag until it changes, with the guarantee that the computation result is available afterwards. With OoOE, this pattern is invalidated: although the *program* on the producer core specifies to write the result and only then write the flag, instruction reordering may invert the two stores.

To address this type of situation, new primitives must be introduced to *protect the order of externally visible side effects* in presence of OoOE. The most common are memory barriers, or *fences*. These must be used by programs explicitly between memory operations used for multi-core synchronisation. When the processor encounters a fence, it will block further instructions until the memory operations prior to the fence have completed. This restores the effect order required by the program, at the expense of less ILP in the pipeline and thus lower utilisation.

Hardware multi-threading

The key motivation for multi-threading in a single core is to exploit the waiting time of blocked threads by running instructions from other threads [35, 33]. This is called Thread-Level Parallelism (TLP); it can tolerate longer waiting times than ILP. To enable this benefit of TLP even for small waiting times like individual memory loads or Floating-Point Unit (FPU) operations, multi-threading can be implemented in hardware (HMT). With HMT, a processor core will contain multiple Program Counters (PCs) active simultaneously, together with independent sets of physical registers for each running hardware thread. The fetch unit is then responsible for feeding the core pipeline with instructions from different threads over time, switching as necessary when threads become blocked [41, 38, 39, 42, 23].

Because each hardware thread executes an independent instruction stream via its own PC, operating systems in software typically register the hardware threads as independent *virtual processors* in the system. Subsequently, from the perspective of software, care must be taken to distinguish virtual from hardware processors when enumerating hardware resources prior to parallel work distribution. Indeed, when work is distributed to two or more hardware threads sharing the same core pipeline, performance can only increase *until all waiting times in that pipeline are filled with work.* Once a pipeline is fully utilised, no more performance can be gained with hardware threads on that core even though there may be some idle hardware threads available.

2.4 Programming principles

As with all parallel computing systems, multi-core programming is ultimately constrained by Amdahl's and Gustafson's laws.

In [2], Amdahl explains that that the performance of one program, i.e. its time to result or *latency*, will stay fundamentally limited by its longest chain of dependent computations, i.e. its *critical path*, regardless of how much platform parallelism is available. The first task of the programmer is thus to shorten the critical path and instead exposing more concurrent computations that can be parallelised. When the critical path cannot be shortened, the latency cannot be reduced further with parallelism. However, Gustafson's law [11] in turn suggests that the problem sizes of the parallel sections can be expanded instead, to increase use of the available parallelism and increase *throughput*, i.e. computations per second, at constant latency.

Within these boundaries, software design for multi-cores involves the following concerns:

- programmers and software frameworks *expose concurrency* in applications. This activity takes two forms. An existing program is *relaxed from ordering constraints* to add concurrency; for example, a sequential loop may be annotated to indicate it can be carried out in parallel. Alternatively, new code can be composed from *concurrent building blocks*, such as primitive map/reduce operators. This activity typically occurs statically, during software development.
- meanwhile, software frameworks and operating systems *map and schedule program concurrency to the available hardware parallelism.* This activity typically occurs at run-time, to carry out program execution over the available cores.

The connection point between these two activities is the *parallel programming model.* Different languages or software libraries will offer different programming models; each offers both *programming abstractions* towards programmers to specify "what to do" and *operational semantics* that provide an intuition of "how the program will behave" at run-time. Parallel programming models typically diverge from traditional programming models in that they avoid letting the programmer specify "how" to carry out computations, so as to give maximum flexibility to the underlying platform.

Parallel programming models can be categorised along two axes, illustrated in Figure 2, depending on how they expose computations and communication.

In one corner, *fork-join parallelism* and *Bulk-Synchronous Parallelism (BSP)* [44] are the most common. With fork-join, a program exposes concurrency by specifying at which points separate threads can start (fork) to compute separate parts of a computation; sequence is then enforced by expressing synchronisation on termination (join) of previously created threads. In the higher-level variant BSP, an overall repetitive computation is expressed as a sequence of wide parallel sections, with computations occurring during the parallel section and communication occurring during the synchronisation step. With these models, control is specified by the structure of the program, while typically communication is left implicit.

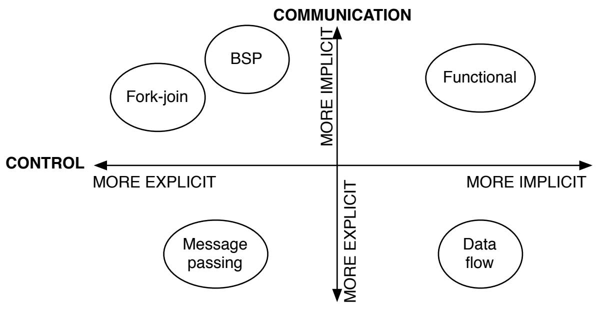


Figure 2: High-level classification of parallel programming models.

At the corner of explicit control and communication, *message passing* describes a collection of idioms where a program specifies independent processes that act collectively by exchanging data across explicit channels. Message passing stems from decades of research over communicating processes [13, 25, 26, 27]. It encompasses the most versatile techniques to program multi-cores, at the expense of a more explicit and error-prone model for programmers.

Diagonally opposed, *functional programming models* encompass the programming style of pure functional languages, where programmers are stimulated to specify only the input-output data relationships of their algorithms using symbolic operations, without introducing or assuming knowledge about the execution environment. These models provide the most flexibility for the platform by removing programmer involvement from mapping and scheduling concurrency, although it thus incurs the expense of a greater technology challenge to fully optimise execution.

Finally, *data-flow programming models* are related to functional models in that the programmer does not specify how to carry out computations. However, whereas with functional programs the data relationships may be implicitly carried by using references (pointers) in data structures, data-flow programs make data edges between computations explicit. This in turn simplifies the mapping of communication to the platform's topology.

A given programming language or software library may expose multiple programming models. The choice of one over another in applications is mostly driven by the trade-off between programmer productivity, technological maturity of the platform and performance: more implicit models are easier to program with, but more difficult to manage for the platform.

3 Impact on practice

After the High Performance Computing (HPC) and general-purpose computing industries stumbled on the sequential performance wall in the last decade, the move to multi-core chips has enabled the following breakthroughs:

- for HPC, the grouping of multiple cores on the same chip has enabled an increase of interconnect bandwidths by an order of magnitude, in turn enabling yet higher throughput scalability of supercomputers. As of 2012, nearly all systems from the supercomputer TOP500[1] are based on multi-core chips;

- for general-purpose computing, the investment of silicon towards multiple cores in combination with the general trend towards mobile computing has altered the marketing of processor products dramatically: instead of advertising on frequency alone, most manufacturers now effectively report on a combination of multi-application benchmarks and energy consumption.

For personal computing, even without considering performance gains in individual applications, the democratisation of multi-core chips has enabled a separation of workload between user applications and system tasks such as user interface management, networking or disk accesses, and thus results in a smoother user experience overall. This effect is noticeable for desktop computers, smart phones and tablets, which all nowadays use general-purpose cores. For selected application domains, such as video games and image processing, significant performance gains have been reached successfully; this effect is even multiplied when combined with many-core compute accelerators and Graphical Processing Units (GPUs).

In data centres, multi-cores have enabled a finer grain control over the resources handed over to client applications. By separating client applications over separate cores, interference between clients is reduced and resource billing is simplified (resources can be billed per unit of space and time, instead of per actual workload which is more difficult to compute). Some throughput increases have also been possible thanks to multi-cores, most noticeably for database servers and networked processes such as web servers. However, the preferred way to increase throughput in data centres is still to extend the number of network nodes instead of replacing existing nodes by new nodes with larger core counts.

Meanwhile, multi-core processors are a success story for embedded systems, where they are commonly called Multi-Processor Systems-on-Chip (MPSoCs). In these systems, processor chips are typically co-designed with software applications; different cores are implemented on chip to support specific application components. For example, a mobile phone processor chip may contain separate cores for managing wireless networks, encoding/decoding multimedia streams and general-purpose application support. By specialising a processor design to an application, hardware and energy costs are reduced while taking advantage of the parallelism to improve performance. While the embedded landscape is still much focuses on application-specific chip designs, it is expected that the field's expertise with mapping and managing application components over heterogeneous resources will propagate to all other uses of multi-cores during the upcoming decade.

3.1 Current landscape of multi-cores chips

All major technology vendors now have multi-core product offerings and continue to invest towards increasing core counts.

After introducing its first mainstream dual-core offerings around 2005 via its Core and Xeon product lines, Intel's processors now nearly all feature a minimum of 2 to 4 cores. As of this writing, the most popular Core i5 and i7

[1]`http://top500.org/`

processors, based on the Sandy Bridge micro-architecture [46], feature 4 cores running at about 3GHz. These chips use 32nm silicon technology and nearly a billion transistors. They integrate an on-chip GPU accelerator with 6 to 12 additional compute cores; the general-purpose cores also optionally feature HyperThreading [23] for a maximum of 2 hardware threads per core. The next generation based on the Ivy Bridge micro-architecture is expected to offer similar features at a reduced silicon and power budget.

Meanwhile, AMD has reduced its competition push on single-threaded performance and pushed its multi-core road map further. Its Opteron product line currently contains chips with up to 12 general-purpose cores based on the Bulldozer micro-architecture [8]. Frequencies range from 1 to 3 GHz; the chips also use the 32nm technology. The upcoming Fusion product line, which uses general-purpose cores based on both Bulldozer and the new Bobcat micro-architecture [7], invest more silicon real estate towards on-chip accelerator cores (e.g. 80 in Brazos chips, 160-400 in Lynx/Sabine chips).

On the embedded/mobile computing market, ARM leads the way towards the generalisation of multi-core platforms. The most licensed architecture towards general-purpose applications is the Cortex-A design, now available with 2 to 4 cores on chip and frequencies up to 2GHz in its “MPCore” variant. The most visible user of Cortex-A is currently Apple, which equips its smart phone and tablet offerings with its own A4 and A5 chips based on Cortex-A. The upcoming ARM design Cortex-A15 is planned to feature up to 8 cores on chip, together with optional on-chip accelerators depending on vendor requirements.

On the server market, Oracle (previously Sun Microsystems) has stepped forward with its Niagara [21] micro-architecture. Niagara processors combine multiple cores with HMT, resulting in high core counts per chip: the most recent product, the SPARC T4 [37], exposes 64 hardware threads to software. Although Niagara was previously advertised for throughput due to its lower initial single-thread performance, the latest generations running around 3GHz with OoOE now compete across all general-purpose workloads.

As can be seen in this overview, the trends suggest a continued increase of core counts on the main processor chip, together with the integration of accelerators. However, separately packaged many-core accelerator chips are still being developed. The two major vendors are nowadays NVidia and AMD, the latter having acquired ATI in 2006. With clock frequencies below 1GHz but core counts in the hundreds, their accelerator chips deliver orders of magnitude higher peak floating-point performance than general-purpose cores with twice the frequency at the same generation. The main challenge to these designs is bandwidth to memory, where communication between the accelerator and the main processor chip is constrained by the inter-chip system bus. This bottleneck constitutes the main push towards integration with general-purpose cores on the same silicon die.

3.2 Shared memory multi-programming for multi-cores

For shared memory multi-programming, the common substrate underlying programming languages and libraries is constituted by *threads*, directly inherited from the era of time sharing on uni-processors. On multi-cores, threads execute simultaneously instead of interleaved, but these two abstractions remain otherwise identical to their original definition: programs create or *spawn* threads, then

```
void scale(int n, int a[]) {
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        a[i] = 2 * i;
}
```

Figure 3: Example OpenMP program fragment.

the operating system selects cores to execute the workload. The leading low-level Application Programming Interfaces (APIs) to manage threads are the POSIX interface [15] ("pthreads") and the Java virtual machine interface. To enable more fine-grained control over thread-to-core mappings, some operating systems also offer APIs to *pin* threads to specific cores (e.g. `pthread_setaffinity`). These basic interfaces are there to stay: as of 2011, new standards for the C [18] and C++ [17] languages has been published which integrate a native standard threading API similar to POSIX.

Upon these basic interfaces, different programming languages and libraries provide different parallel programming models for application developers. With less than a decade of renewed interest in multi-programming, these software frameworks for multi-cores have not yet stabilised and a large diversity of approach still exists across vendors, hardware platforms and operating systems.

For performance-oriented applications, the leading interfaces are currently OpenMP [29] for C, C++ and FORTRAN, and Intel's Threading Building Blocks (TBB) [32, 16] for C++. OpenMP is specialised towards the parallelisation of existing sequential code by the addition of annotations, or *pragmas*, to indicate which portions of code can be executed concurrently. An example is given in Figure 3: a loop is annotated to declare it can be run in parallel when the function "scale" is called; at run-time the value of "n" is inspected and the workload is distributed across the available cores.

In contrast to OpenMP, TBB is oriented towards the acceleration of new code, where programmers use TBB's control and data structures directly. Primitive constructs are provided for parallel map/reduce, searches, pipelines, sorting algorithm, as well as parallel implementations of container data structures (queues, vectors, hash maps). An example is given in Figure 4: the object "`scaler`" is responsible for carrying out the computation over sub-ranges of the array, and TBB ensures that `scaler`'s operator is called in parallel over the available cores.

Both OpenMP and TBB manage program concurrency in a similar fashion. When reached during execution, the program code generated by the compiler for concurrency constructs causes calls to the language run-time systems to define *tasks*. The run-time system in turn runs a *task scheduler* which spreads the tasks defined by the program over *worker threads*, which it has previously configured to match the number of underlying cores. In both interfaces, primitives are available to control the task scheduler and query the number of worker threads.

When accelerators are involved, typically the accelerator cores cannot run regular application code because they do not support recursion or arbitrary synchronisation. To program them, it is still customary to use a different set of APIs. The current leading standards are NVidia's CUDA interface [20], specialised towards its own chips, and OpenCL [19] which intends to provide a uni-

```
struct scaler {
  vector<int>& a_;
  scaler(vector<int>& a) : a_(a) {}
  void operator()(const blocked_range<size_t>& r) const {
     for (size_t i = r.begin(); i != r.end(); ++i)
          a_[i] = 2 * i;
  }
};
void scale(int n, vector<int>& a) {
  parallel_for(blocked_range<size_t>(0, n), scaler(a));
}
```

Figure 4: Example TBB program fragment.

```
// the following defines the kernel.
__kernel void scale_kernel(__global int *a) {
   size_t i = get_global_id(0);
   a[i] = 2 * i;
}
// the kernel is used as follows:
void scale(int n, int a[]) {
   /* need to copy the data to GPU memory first */
   void *gpu_mem = gcl_malloc(sizeof(int)*n, a,
                 CL_MEM_COPY_HOST_PTR|CL_MEM_READ_WRITE);
   /* then define a range to operate over the data */
   cl_ndrange r = { 1, {0}, {n,0,0}, {0} };
   /* then call the kernel */
   scale_kernel(&r, gpu_mem);
   /* then copy back the data from GPU to main memory */
   gcl_memcpy(a, gpu_mem, sizeof(int)*n);
   /* then release the GPU memory */
   gcl_free(gpu_mem);
}
```

Figure 5: Example OpenCL program fragment.

fied interface to accelerators. With both interfaces, the application programmer defines *computation kernels* which can execute on the accelerator cores, and uses the interface's API to trigger data movement and computations using kernels.

An example is given in Figure 5. As the example suggests, the main challenge of accelerator-based programming is data movement between the accelerator and main memory. While it is often possible to combine accelerator computations and thus reduce the need for communication, many cases exists where the application structure prevents keeping the data on the accelerator's memory. Again, the industry is moving towards tighter integration of accelerators and general-purpose cores on the same chip, in an effort to alleviate this communication overhead.

3.3 Distributed programming on chip

It is also possible to consider a multi-core chip as a network of single-core nodes sharing a very efficient interconnect. Using this model, a multi-core chip can be programmed using *explicitly communicating processes* instead of threads communicating implicitly via shared memory.

Abstractions to program multiple processing units using communicating processes had existed for decades and are now coming back with the advent of multi-cores. The typical interface for scientific computing, coming from the HPC community, is MPI [24]: it exposes a process management and message passing interface to C, C++, FORTRAN and Java code. Implementations of MPI are nowadays able to distribute workloads over multiple cores in a single chip as well as over a network of nodes. Meanwhile, the advent of networked applications in the last decade has caused a large diversity of other frameworks for inter-process *message queuing and brokering* in business applications: Java Message Service (JMS), Microsoft Message Queuing (MSMQ), WebSphere Message Broker from IBM, Apache ActiveMQ are examples. These interfaces are nowadays commonly used to drive processes running over separate cores in the same chip.

Post-2010: the era of multi-scale concurrency

While general-purpose programmers have been struggling to identify, extract and/or expose concurrency in programs during the last decade, a large amount of untapped higher-level concurrency has also appeared in applications, ready to be exploited. This is a consequence of the increasing number of features, or *services* integrated into user-facing applications in the age of the Internet and ever-increasing support of computers for human activities. For example, while a user's focus may be geared towards the decoding of a film, another activity in the system may be dedicated to downloading the next stream, while yet another may be monitoring the user's blood nutrient levels to predict when to order food online, while yet another may be responsible for backing up the day's collection of photographs on an online social platform, etc.

Even programs that are fundamentally sequential are now used in applications with high-level concurrency at scales that were unexpected. For example, the compilation of program source code to machine code is mostly sequential as each pass is dependent on the previous pass' output. However, meanwhile entire applications have become increasingly large in terms of their number of program source files, so even though one individual compilation cannot be accelerated via parallelism it is possible to massively parallelize an entire application build. While this form of parallelism had been known in large enterprise projects, the advent of multi-cores makes it accessible to any programmer working with commodity platforms.

4 Research issues

As this book gets published, the multi-core programming challenge has taken a new form. Explicit concurrency has appeared in software, both from increased understanding by programmers and by new technology in compilers and software run-time systems to discover concurrency automatically in applications.

Meanwhile, CMPs now contain dozens of cores for general-purpose computations and accelerators offer hundreds of smaller cores for specialised computing, and the trends suggest at least a ten-fold increase before the end of silicon scaling. The main challenge for software engineers is thus now less to find concurrency, but rather to *express* and *map* it efficiently to the available parallel hardware. Meanwhile, the main challenge for architects is to balance the need of software practitioners to manipulate *simple machine models* while providing *scalable* systems. The architecture and software engineering communities have thus started to work together to overcome the following new challenges:

- choose which *parallel machine models* to communicate to programmers to give them an intuition of the underlying hardware resources;
- choose which *programming abstractions* to offer through the software stack to describe concurrency in applications;
- determine how to *schedule concurrency over parallel resources* in operating software;
- for known application or fields, determine how to *co-design hardware and software* so that the hardware parallelism aligns with application concurrency.

4.1 The platform challenge: communication costs

As the example from Figure 5 illustrates, the cost of communication is becoming a growing design constraint for algorithms. In upcoming multi-core chips, the latency to access memory from cores, or even to communicate between cores, will become large compared to the pipeline cycle time. Any non-local data access will become a serious energy and time expenditure in computation. This is a new conceptual development compared to the last decade, where processor speed was still the main limiting factor and memory access latencies were kept under control using ever growing caches.

From the software architect's perspective, this *communication challenge* takes two forms. For one, either programmers or the concurrency management logic in operating software must become increasingly *aware of the topology* of the platform, so as to match the dependencies between application components to the actual communication links present in hardware. This requirement will require new abstractions and investments in programming languages and operating software, since the current technology landscape still mostly assumes SMPs and UMAs. The second aspect is that the *cost of computations* is no more a function only of the number of "compute steps," correlated with CPU time; it must also involve the "communication steps" correlated with on-chip network latencies. This is a major conceptual shift that will require *advances in complexity theory* before algorithm specifications can be correlated to actual program behaviour in massively parallel chips.

From the hardware architect's perspective, the communication challenge takes three forms. One is to develop *new memory architectures* able to serve the bandwidth requirements of growing core numbers. Indeed, the energy and area costs of central caches that serve all cores symmetrically grow quadratically with their capacity and number of clients; caches will thus dominate silicon usage until more distributed cache systems are developed. However, distributed caches in turn require *weaker memory consistency* semantics [28] to be cost advantageous; a generalisation of weakly consistent memory architectures will in turn have a dramatic impact on software ecosystems. The second aspect to be covered by hardware architect is *latency tolerance*: as the time cost of non-local data accesses grows, individual cores must provide mechanism to overlap computation with communication. A step in this direction is HMT, which will be increasingly complemented with hardware support for point-to-point messaging between cores, such as found in the recent TILE architecture [5]. The third aspect is *dark silicon* [10]: any given design will be fully utilised only by some applications, while most of the silicon will be under-utilised by most applications. The role of architects will thus be to determine the best trade-offs between investing silicon real estate towards cores or towards communication links.

4.2 The software challenge: matching abstractions to requirements

A lot of attention has been given on the parallelisation of existing software, and comparatively less on the improvement of software to better program parallel platforms. There are, in effect, three broad strategies to *optimise performance and cost* on multi-core systems.

The first is provide better abstractions *to programmers* to compose sub-programs so that the resulting critical path becomes shorter—i.e. decrease the amount of synchronisation programmers use—for a given functional specification of the input-output relationship. This is the classical effort towards increasing the amount of concurrency, which must continue as the amount of on-chip parallelism increases; beyond parallelization of individual algorithms, this effort must now also take place at the level of entire applications.

The second strategy is to determine ways to shorten programs to *describe less computations* to be performed at run-time, i.e. simplify the input-output relationship. There are two known strategies to do this:

- let *programmers* use domain-specific knowledge that reduce expectations on program outputs, for example reduce output "quality" in image processing by introducing non-determinism when the difference is not perceptible. This is the domain of *approximate programming* [43, 36] and Domain-Specific Languages (DSLs);
- when *compilers and software run-time systems* transform programs, exploit extra application-level knowledge to remove excess code from the individual sub-programs being combined [3].

The third strategy is for programmers to *remove unnecessary constraints* on the execution of algorithms. This strategy stems from the observation that most control and data structures in use today have been designed at a time where computers were predominantly sequential, and thus may carry implicit requirements to preserve ordering even when it is not relevant to the application. For example, many programmers use lists and arrays as containers, which implicitly carry a strong ordering guarantee. In contrast, languages like C++ or Haskell provide high-level *type classes* (contracts) which enable the programmers to state their requirements, e.g. an non-iterable associative container, and let the implementation choose a suitable parallelizable implementation, e.g. a distributed heap or hash table. Research is still ongoing in this direction [30, 6, 9].

5 Summary

The move towards increasing core counts on chip was both an answer to overcome the sequential performance wall [1, 34] and to bring higher communication bandwidths to parallel computing. The expected benefits of multi-cores were both higher performance and lower energy costs, thanks to frequency scaling.

The foremost challenge with multi-cores is not new, as it was shared by early practitioners with parallel architectures until the 1980's. This "concurrency challenge" requires software engineers to acknowledge platform parallelism and spend extra effort to express concurrency in applications. In response to this, new language and operating software technology has been developed in a short time, resulting in a large diversity of platforms, which have not yet matured nor stabilized. While this diversity creates opportunities in the highly dynamic IT industry, it also means that experience gained by practitioners in the last decade will likely need to be revisited in the coming ten years.

The move to CMPs, especially with increasing core counts and accelerators on chip also entails new technological and conceptual issues. More active components in the system imply faults or otherwise resource heterogeneity that must be understood and modeled. Communication links between cores and between cores and memory must be accounted for when mapping application components to the chip's resources. Real parallelism between application components imply that programmers cannot stop an application and observe a consistent global state. The benefits of multiple cores on performance can only be reaped by reducing synchronization, which for some applications means decreasing reliance on determinism. These issue in turn require new abstractions to describe and manipulate the computing system at a high level, and research has barely started to characterize which general aspects of multi-core parallelism will be relevant in the next era of growing software concurrency.

Defining terms

Concurrency vs. parallelism "Concurrency is non-determinism with regards to the order in which events may occur. Parallelism is the degree to which events occur simultaneously." [14]

Operating software software composed of operating systems, compilers and language run-time systems, in charge of mapping the concurrency expressed in software to the available parallel resources in hardware.

Programming model conceptual model available to users of a given programming language. Consists of programming abstractions that allow programmers to specify "what to do" and operational semantics that give programmers an intuition of "how the program will behave." Parallel programming models are special in that they discourage programmers from specifying how to carry out computation, so as to leave operating software maximum flexibility to map and schedule the program's concurrency.

Scalability ability of a system to approximate a factor N performance improvement for a factor N cost investment (e.g. silicon area, number of cores, energy, frequency).

Throughput vs. latency throughput is the number of computations achieved by unit of time, whereas latency is the number of seconds necessary to achieve a unit of computation. Parallelism can decrease latency down to a program's critical path, whereas throughput typically remains scalable as the workload on concurrent sections can be arbitrarily increased.

Topology of architecture models the topology of an architecture consists of how components are connected to each other. On multi-core processor chips, one can consider separately the memory topology, ie. how cores are connected to main memory, the cache topology, ie. how caches are connected to cores, main memory and each other, and the inter-core topology.

Typology of architecture models the typology of an architecture consists of the set of component types that participate in the design. For example, heterogeneous multi-core architectures have more than one processor type.

Acronyms

API	Application Programming Interface
BSP	Bulk-Synchronous Parallelism
ccNUMA	Cache-Coherent NUMA
CMP	Chip Multi-Processor
DSL	Domain-Specific Language
DSM	Distributed Shared Memory
FPU	Floating-Point Unit
GPU	Graphical Processing Unit
HMT	Hardware Multi-Threading
HPC	High Performance Computing
ILP	Instruction-Level Parallelism
IPI	Inter-Processor Interrupt
IPS	Instructions Per Second
MIMD	Multiple Instruction, Multiple Data
MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
NUMA	Non-Uniform Memory Architecture
OoOE	Out-of-Order Execution
PC	Program Counter
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multi-Processor
SPMD	Single Program, Multiple Data
TBB	Threading Building Blocks
TLP	Thread-Level Parallelism
UMA	Uniform Memory Architecture

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *SIGARCH Comput. Archit. News*, 28:248–259, May 2000.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 41–52, September 2009.

[4] Ian Bell, Nabil Hasasneh, and Chris Jesshope. Supporting microthread scheduling and synchronisation in CMPs. *International Journal of Parallel Programming*, 34:343–381, 2006.

[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *IEEE International Solid-State Circuits Conference, 2008 (ISSCC 2008). Digest of Technical Papers.*, pages 88–598. IEEE, February 2008.

[6] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, September 2004.

[7] B. Burgess, B. Cohen, M. Denman, J. Dundas, D. Kaplan, and J. Rupley. Bobcat: Amd's low-power x86 processor. *Micro, IEEE*, 31(2):16–25, March/April 2011.

[8] M. Butler, L. Barnes, D.D. Sarma, and B. Gelinas. Bulldozer: An approach to multithreaded compute performance. *Micro, IEEE*, 31(2):6–15, March/April 2011.

[9] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[10] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proc. 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[11] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[12] John L. Henessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Publishers Inc., fourth edition, 2007.

[13] C. A. R. Hoare. *Communicating sequential processes.* Prentice Hall, 1985.

[14] Philip Kaj Ferdinand Hölzenspies. *On run-time exploitation of concurrency.* PhD thesis, University of Twente, Enschede, the Netherlands, April 2010.

[15] IEEE Standards Association. *IEEE Std. 1003.1-2008, Information Technology – Portable Operating System Interface (POSIX®).* IEEE, 2008.

[16] Intel Corporation. *Intel® Threading Building Blocks Reference Manual*, 2011.

[17] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 14882:2011, Programming languages – C++*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, first edition, September 2011.

[18] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 9899:2011, Programming Languages – C*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, first edition, December 2011.

[19] Khronos OpenCL Working Group. The OpenCL specification, version 1.0.43, 2009.

[20] David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 103–104, New York, NY, USA, 2007. ACM.

[21] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[22] M.H. Lipasti and J.P. Shen. Superspeculative microarchitecture for beyond AD 2000. *Computer*, 30(9):59–66, September 1997.

[23] D. T. Marr, Frank Binns, D. L. Hill, Glenn Hinton, D. A. Koufaty, J. A. Miller, and Michael Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

[24] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009.

[25] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1980.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[27] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[28] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, 1993.

[29] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0, 2008.

[30] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ACM.

[31] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address). In *Proc. 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, Washington, DC, USA, 1999. IEEE Computer Society.

[32] J. Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Series. O'Reilly, 2007.

[33] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17:365–375, July 1974.

[34] R. Ronen, A. Mendelson, K. Lai, Shih-Lien Lu, F. Pollack, and J.P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, mar 2001.

[35] Jerome H. Saltzer. CTSS technical notes. Technical Report MAC-TR-16, Massachusetts Institute of Technology – Project MAC, 1965.

[36] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011.

[37] M. Shah, R. Golla, P. Jordan, G. Grohoski, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. SPARC T4: A dynamically threaded server-on-a-chip. *IEEE Micro*, PP(99):1, 2012.

[38] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. *Proc. SPIE Int. Soc. Opt. Eng.; (United States)*, 298:241–248, 1981.

[39] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society.

[40] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[41] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, AFIPS '64 (Fall, part II), pages 33–40, New York, NY, USA, 1965. ACM.

[42] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23:392–403, May 1995.

[43] David Ungar and Sam S. Adams. Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 19–26, New York, NY, USA, 2010. ACM.

[44] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, aug. 1990.

[45] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23:20–24, March 1995.

[46] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266, February 2011.