Era of Multi-Core Processors

Maheshkumar P Jagtap
Advanced Numerical Research & Analysis Group, Kanchanbagh, Hyderabad-560 075
Email: mahesh@anurag.drdo.in

ABSTRACT
Since the very beginning of integrated circuits development, processors were invented with ever-increasing clock frequencies and sophisticated in-build optimisation strategies. Due to physical limitations, this 'free lunch' of speedup has come to an end. Physical constraints put up multiple barriers in achieving high performance computing within power budgets. Computer architects are trying to cross these performance restricting walls using number of innovative processor architectures employing area concurrency. These processors are commonly known as multi-core processors. The following article gives a summary of recent trends and challenges in multi-core processor architectures. It discusses how 40 years of parallel computing research need to be considered in the upcoming multi-core era. I feel that the future research must be driven from two sides—a better expression of hardware structures, and a domain-specific understanding of software parallelism.

Keywords: Multi-core processors, high performance computing, multi-core processor architecture

1. HIGH PERFORMANCE AND ENERGY EFFICIENCY
Applications that require energy efficient high performance computing are becoming increasingly commonplace. These applications often comprise multiple computing tasks and are frequently key components in many systems such as: wired and wireless communications, multimedia, large-scale multi-dimensional signal processing (e.g., medical imaging, synthetic aperture radar), some large-scale scientific computing, remote sensing and processing; and medical/biological processing. Many of these applications are embedded and are strongly energy constrained (e.g., portable or remotely-located) and cost-constrained. In addition, many of them require very high throughputs, often dissipate a significant portion of the system power budget, and are therefore of considerable interest. There are several design approaches to achieve high performance computing solutions such as ASICs, programmable processors/DSPs and FPGAs. ASICs can provide very high performance and very high energy efficiency, but they have little programming flexibility. On the other hand, programmable processors/DSPs are easy to program but their performance and energy efficiency is normally 10–100 times lower than ASIC implementations. FPGAs fall somewhere in between. One-time fabrication costs for state of the art designs (e.g., 90 nm CMOS technology) are roughly one million dollars, and total design costs of modern chips can easily run into the tens of millions of dollars. Programmable processors are the platforms which allow high one-time fabrication costs and design costs to be shared among a variety of applications/users; but higher performance and energy efficiency are expected to be achieved using architectural innovations.

2. AREA-TIME-POWER TRADE-OFFS
To analyse the future microprocessor architecture, we must first understand the trade-offs among area (A), time (T), power (P) and technology (S). For a fixed process technology, higher performance (smaller T) inevitably requires more power (P) and area (A). The driving force in design innovation is the rapid advances in technology. As technology advances and feature size shrinks, the three design considerations benefit from one process generation to another, resulting in higher speed, smaller area, and reduced power consumption.

3. PERFORMANCE CONSIDERATION
Increasing the processor clock frequencies and using wide issue processor architectures have worked well to improve performance but recently have become significantly more challenging. Deeper pipelining is one of the key techniques to increase the clock frequency and performance, but the benefit of the deeper pipeline is eventually diminished when the inserted Flip-Flop’s delay is comparable to the combinational logic delay. Moreover, deeper pipeline stage increases cycles-per-instruction (CPI) and impacts negatively to the system performance. Performance increase by microarchitecture alone is governed by Pollack’s Rule, which states that performance increase is roughly proportional to square root of increase in complexity. In other words, if you double the logic in a processor core, then it delivers only 40 per cent more performance.
Researchers found that the depth per pipeline is approximately 8 Fanout-4 (FO4) inverter delays to obtain highest performance which corresponds to 20–30 pipeline stages. The pipeline delay of some modern processor is already close to 10 FO4 so that the deeper pipelining technique for high performance is reaching its limit. Also, increasing pipeline stages necessitates more registers and control logic, thereby further increasing design difficulty as well as power consumption. The optimum pipeline depth for maximum energy efficiency is about 22.5 FO4 delay (about 7 stage pipeline), using BIPS/Watt as the metric—BIPS are billions of instructions per second.

Invariably, a faster functional unit (such as an arithmetic logic unit or memory interface) is larger and more expensive than a slower functional unit. Ullman shows that an implementation is theoretically bounded by if volume (I/O, power dissipation, and so on) limits it; or \( \frac{A}{T^2} \) in implementation that are communications limited. So depending on the type of functional unit and circuit implementation, doubling processing speed could increase die size by 2 to 4 times

\[
A \times T^n = \text{cons} \tan t
\]

where \( n \) is typically between 1 and 2.

Besides exploiting fabrication process advancements, computer architects employed wide issue processor architectures such as VLIW and superscalar approaches for high performance computation. While their benefit is also quickly diminished when the issue width is more than 10; since most applications don’t have so many independently parallel executable instructions in per fetch.

4. POWER CONSIDERATION

For most applications, power is the real price of performance. Processor power dissipation is the sum of dynamic switching power and static leakage power:

\[
\text{Power}_{\text{total}} = \text{switching power} + \text{leakage power} \tag{2}
\]

\[
= \frac{C V^2 f_{\text{clk}}}{2} + I_{\text{leakage}} V \tag{3}
\]

Switching power is directly proportional to capacitance and activity in the application programs. The leakage power increases exponentially with threshold voltage \( (V_f) \) creating a major problem.

Performance-oriented designs take advantage of the decreased capacitance that the process technology provides to improve switching time accordingly:

\[
\Delta t = \frac{C \times \Delta V}{I} \tag{4}
\]

Large switching current \( I \) significantly improve switching times. If \( I \) is small compared with supply voltage, the switching current is roughly proportional to the square of the supply voltage. Thus, the switching frequency is roughly proportional to the supply voltage, which implies that there is a cubic relationship between \( T \) and power consumption \( (P) \) at the same feature size

\[
P \times T^3 = \text{cons} \tan t \tag{5}
\]

Based on this cubic rule, it requires roughly eight times more power to reduce processing time by half.

The trade-off of cost, performance, and power consumption is fundamental to any system design. Although there are numerous design possibilities in this area-time-power continuum, the application always drives the optimum design point. Most high performance technologies, such as increasing clock frequencies and increasing processor issues (which mean increasing number of circuits and increasing capacitance) result in higher power consumption. Here architects face three major performance bottlenecks or walls:

Memory wall

The access time to memory depends on wire delay that is relatively constant with scaling. As processor clock rates increases, so do cache miss times, creating a memory wall on performance. So far, significant processor hardware support (such as increased cache size and branch tables) and an increased emphasis on memory performance has helped designers manage cache misses and branches, reducing access to memory requirements and alleviating the impacts of memory wall. Nevertheless, the memory wall is still a serious issue.

Frequency wall

The maximum pipeline segment size (the number of logic gates in one pipeline stage) determines clock frequency. Reductions in segment size beyond a certain point are difficult, creating a frequency wall. Here, the question is how to improve performance without reducing segment size.

Power wall

Higher frequency implies greater power density. As clock speed increases, the cost of removing the resulting heat limits a design and its effectiveness at some point. Although these walls arise from physical constraints and application program behaviors, new processor and memory paradigms need to be explored to improve memory access and access predictability. For example, area concurrency-not clock frequency- can help improve performance.

5. FUTURE FABRICATION TECHNOLOGIES

Future fabrication technologies are also imposing new challenges such as wiring and parameter variations. In the early days of CMOS technology, wires could be treated as ideal. They transmit signals with infinite speed, without power consumption, and without coupling effect. This assumption is no longer true. For global wires, their length is nearly constant along with the technology scaling if the chip size stays the same; which makes their delay nearly constant. Compared to gate delay which scales down with the technology, the delay of the global and long wires scales up with the technology. A global wire 1cm long delay in modern 65 nm technology is around 100 FO4 delay.
delay; which corresponds to more than one clock cycle period in modern processors and the global wires have to be pipelined or be eliminated through architecture level innovations. Similar with the delay, the power consumption of the long wires also scales up along with the technology compared to the gates. Besides the effect on delay and power consumption, the inductive and capacitive coupling between wires adds signal noise and impacts system reliability.

Furthermore, future fabrication technologies are expected to have tremendous variability compared to current technologies in both gates and wires. Fundamental issues of statistical fluctuations for submicron MOSFETs are not completely understood, but the variation increases leakage of transistor and causes a variation of the speed of individual transistors, which in turn leads to IC timing issues. Reported chips fabricated in advanced nanometer technologies can easily have 30 per cent variation in chip frequencies. This variance will be present at time of fabrication and also have a time-varying component.

6. SOLUTION: MULTI-CORE SYSTEMS

To address the challenges in performance, power and future technologies, innovations on computer architecture and design are needed; and multi-core systems are one of the most, or the most promising technology. As was also pointed out by a computer architecture group at EECS department of UC Berkeley the "shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures".

Deep submicron fabrication technologies enable very high levels of integration such as a dual-core chip with 1.7 billion transistors, thus reaching a key milestone in the level of circuit complexity possible on a single chip. A highly promising approach to efficiently use these circuit resources is the integration of multiple processors onto a single chip to achieve higher performance through parallel processing, which is called a multi-core system or a chip multiprocessor.

Multi-core systems can provide high energy efficiency since they can allow the clock frequency and supply voltage to be reduced together to dramatically reduce power dissipation during periods when full rate computation is not needed. Giving a simple example, assuming one uniprocessor is capable of computing one application with clock rate \( F \) and voltage \( V \), and consuming power \( P \); now if using a dual core system and assuming the application can be partitioned into two cores without any overhead, then each core only needs a clock rate \( F/2 \) and the voltage can be reduced accordingly; assuming a linear relation between voltage and clock frequency and the voltage is reduced to \( V/2 \), then the power dissipation of the dual core system will only be about \( P/4 \).

Multi-core systems also potentially provide the opportunity to independently control each processor’s clock and voltage to achieve higher energy efficiency, if different processors are in separate clock and voltage domains.

Furthermore, multi-core systems are suitable for future technologies. The distributed feature can potentially constrain the wires into one core and eliminate the global (long) wires. The multi-core systems also provide flexible approaches to treat each core differently by adjusting the mapped application, supply voltage, and clock rate; to utilise each core’s specific features due to variations. For example, when one processor in the chip is much slower than the others, a low workload can be mapped on it without affecting system performance. The multi-core systems have high scalability since a single processor can be designed and the system can be obtained by combining multiple processors. Thus the systems can be easily adjusted according to the required performance and cost constraints by changing the number of cores; which is much easier than changing the issue-width or the pipelining of uni-processors.

A multi-core CPU combines multiple independent execution units into one processor chip, in order to execute multiple instructions in a truly parallel fashion. The cores of a multi-core processor are sometimes also denoted as processing element or computational engine. According to Flynn’s taxonomy, the resulting systems are true multiple-instruction-multiple-data (MIMD) machines, able to process multiple threads of execution at the same time.

Achieving processing speedup by using parallel execution units is obviously nothing new. The concept of the first multiprocessor computer goes back to the 60’s with initial architectures such as the ILLIAC IV. The resulting challenges were also discussed to large extend, sometimes even 25 years ago. What is the difference now?

The true paradigm shift today is not the realization of CMP (Chip Multiprocessor) architectures, but their spreading in all computer markets. Embedded systems, mobile phones, desktop systems and server systems now include multiple cores out of the box. A parallel computer is no longer a dedicated hardware setup for special purposes. It is commodity.

While parallel computing of the past was only intended for a specific set of problem domains, it is now relevant for every scientific, industrial or private application running on a computer. Parallel computing now becomes visible for millions of industrial software developers in practice, who usually lack experience in the creation and handling of fine-grained parallel activities. For academia, the shift will influence both future teaching and research in computer science and software engineering. Many traditional areas such as computer hardware architecture, programming languages, design patterns, scheduling theory and parallel algorithms will gain more attention in the future, since the upcoming research challenges demand answers from these fields. In the following text, an attempt is made to provide a high-level overview about some of the identified issues in this area. I base the argumentation on one fundamental statement:
"40 years of parallel computing need to be considered."

7. PARALLEL HARDWARE

The support for multiple parallel activities (multithreading) in hardware is realised on different levels in today’s computer systems (Fig. 1). The scheduling of parallel activities must now consider this given 'stack' of execution units. Even though modern operating systems are meanwhile aware of this effect, they still cannot consider data dependencies between parallel activities in their scheduling decision. The application therefore has to support the operating system scheduler in the placement with its specific knowledge.

On the lowest level, the execution unit itself can have a super-scalar architecture. A hardware-controlled parallel usage of execution unit components enables the execution of multiple processor instructions during one clock cycle. This approach of instruction-level parallelism (ILP) is limited, but widely realized approach in modern processor designs.

Each execution unit can additionally support the concept of a logical processor, which allows a simultaneous multi-threading (SMT) inside the processor pipeline. This approach allows to hide data access latencies for some of the threads, by utilizing the execution unit for other tasks during a blocking period. This realizes a virtual sharing of one execution unit within a processor. SMT is better known under the marketing term hyperthreading from Intel. It maintains the architectural state of the execution unit (mostly CPU registers) separately per logical processor, in order to allow context switching in hardware.

A set of execution units can be put together to form a chip multi-processing (CMP) architecture. Most such processors contain separate L1 caches for each of the units, and a shared L2 cache for the whole processor chip. CMP forms the latest trend in processor design, even though earlier attempts already gained some experience with this approach.

The different ways of exploiting parallelism inside one processor chip are then summarized as chip multi-threading (CMT) capabilities. The next higher level is symmetric multi-processing (SMP) with multiple processors, and ultimately the realization of a computer cluster as one multicomputer.

The widely promoted new era of multi-core systems

Figure 1. Hardware parallelism hierarchy.
basically focuses on the introduction of CMP features in standard desktop processors. Even though there is a high amount of daily news on recent hardware development, some common trends can be identified in this development.

Most sources agree that the number of execution units per chip will be in the magnitude of thousands in the next 5-10 years. A recent report from Berkeley predicts CMP processors with thousands of parallel execution units as the mainstream hardware of the future. The "Intel Terascale Computing" initiative designed in 2008 an 80-core research prototype to investigate future challenges for the company. The MIT-originated TILE64 processor architecture today provides a grid of 64 low-speed (866 MHz) execution units on a chip, interconnected by a mesh network. In general, the trend with such homogeneous multi-core architectures occurs to be a high number of less complex execution units, working together by some high-speed connection grid.

The opposite approach is a heterogeneous multi-core architecture. One example is the Cradle Technologies' CT3616 processor. It combines two quads of processing cluster and each quad contains eight Digital Signal Engines (DSE) and four Processing Engines (PE). The relevant property here is the heterogeneous set of execution units combined to one chip. It allows a prepared algorithm to use an optimised core for the computational task, while doing different activities in parallel. This kind of task-level parallelism is already common for modern personal computers with their dedicated chips for graphic processing and I/O, but now also reaches the CPU itself. Another well-known example is the z series mainframe concept, combining different processing units on one multi chip module with a shared L2 cache. Some of the units are dedicated as fault tolerance spares or diagnostic chip. This demands specialized compilers and an operating system prepared for the according processor platform.

Still underestimated hardware problem is the connectivity of the execution units. The overall chip still must be connected to memory and I/O devices, which are an order of magnitude slower in their latency time. Industry provides some solutions for this issue, which still need to be evaluated for higher numbers of execution units.

The overall idea of many slow processing units connected by a high-speed bus is also not completely new. Computer hardware history shows the INMOS Transputer concept, classical vector computers or massive parallel processing (MPP) installations. It is therefore necessary to consider the tremendous existing knowledge from these areas in the future.

A problem example in the hardware category is the contention of shared resources. With the increased introduction of parallelism in software layers, the processor faces a high number of threads with different memory access patterns and cache utilization profiles. Proposals by Intel suggest the introduction of a QoS-aware memory hierarchy, were the operating system prioritizes some threads in their cache and memory bandwidth usage. Other strategies still need to be investigated. This shows the general importance of better software-to-hardware mapping. Above the level of superscalar instruction processing, all parallelisation coordination must now be done in software. This coordination must consider the given parallel hardware layers, without binding itself too much on one particular processor architecture. It is therefore urgent to express modern hardware design in a more generic way, to develop appropriate scheduling and data management approaches for parallel applications.

The history of high-performance computing provides the idea of an abstract MIMD machine model, which is also applicable for modern CMT architectures. Such models support the runtime performance analysis for a given parallel algorithm, based on a very abstract understanding of a parallel execution environment. Famous examples are the Log P model, the bulk synchronous parallel (BSP) computer model and the parallel random access machine (PRAM) model. Most of the existing models imply unlimited space or cycle time, and can even lead to contrary efficiency numbers for the same algorithm. Other models such as the multicomputer focus only on distributed systems with multiple processors. It is therefore necessary to find a better abstraction of multi-core hardware architectures. This includes the consideration of timing effects reasoned by caches and mutual exclusion, which more and more turn out to be the performance-relevant factor.

8. PARALLEL SOFTWARE

Multi-core systems are true parallel computers. The new trend therefore leads to the wider recognition of partially well-known and partially largely software problems. Industry and research already agree upon the fact that the existing programming paradigms and languages as well as design patterns do not align to modern processor design. This leads to the fact that the original hardware scalability problem is about to be replaced by a software parallelisation problem. Challenges that are well known to the parallel computing community are now the problem for every software developer.

This is nothing less than a paradigm shift in education, training, and daily practice of software development. Developers must get a basic understanding of parallel programming from the very beginning, treating a sequential program only as special case. This relates not only to tools and languages, but also to design patterns, algorithmic thinking, testing strategies and software architecture principles. The usage of multiple execution units has even influence on dependability issues, meaning that it also becomes a relevant aspect for research on fault-tolerant systems.

9. BASIC PRINCIPLES

The basic principles of parallel respectively concurrent programming are known for a long time. The mutual exclusion problem, deadlock, livelock and starvation are well-known questions for most computer science students and researchers. The challenge here is the consideration of such problems for the 'average' programmer. Due to the industrialisation
and wide-spread of software development, more and more developers do not have a solid scientific background in parallel computing. It is therefore necessary to assist these people in their development effort by appropriate testing mechanism, programming environment and high-level helper functionality.

There are widely accepted principles for the speedup achievable by parallelisation. The two corner stones are Amdahls and Gustafsons laws. Amdahls law puts an upper limit on the parallelization speedup achievable with a constant problem size.

Parallel Speedup = \frac{1}{(1-Serial\%) + \frac{1-Serial\%}{N}} \quad (6)

Figure 2 shows the speedup achievable by more execution units is always limited, even with a mostly parallelized application (P=95%). Amdahls law suggests that ultimate speed up is function of algorithm, not number of processors.

Gustafsons law relaxes this upper bound by expressing that speedup can always be achieved by the shift to bigger problems where parallel component scales as problem size which scales as number of processors and the serial component of code is independent of problem size. The speed up is given by equation 7 and it scales linearly.

Parallel speedup = \frac{1}{(1-Serial\% + \frac{1-Serial\%}{N})} 

These two intuitive facts are still very valid in the modern multi-core era, and therefore need to be considered in all related research. This demands a proper analysis of parallelization strategies for software. We propose that this kind of analysis can be for application domains in a whole, in order to derive generic parallelization principles and approaches. One example for this strategy is the traditional parallel database research, another one is the new area of parallelized XML processing. Future work needs to identify more such application domains and should provide according parallelization strategies for them. This of course demands a heavy involvement of the respective user communities, which is a new chance for computer science to be better, connected to other sciences.

10. PARALLELISATION SUPPORT

Parallelisation support, already implemented by somebody else, can be provided through the compiler, the operating system or third-party libraries. Such support functionality is used by the application developer either implicitly (e.g., automated loop parallelization) or explicitly through new language constructs respectively a dedicated parallel language.

A compiler can try to formulate the assembler code in a way that multiple execution units are automatically used. In the perfect world, even the operating system developer would then be freed from the consideration of parallel execution strategies. Recent projects in this area still show that this remains a grand challenge of computer science. Since the parallelizing compiler has to determine possible side effects automatically, it becomes extremely hard to still generate valid code. Automatic parallelization mainly focuses on loops, since the identification of coarse grained parallel chunks remains application-dependent. It is therefore a commonly accepted fact that parallelizing compilers can only provide limited help with speedup by parallelization.

The next level of software parallelism support is the operating system, which currently acts as common glue between hardware parallelism and application parallelism (Fig. 3). All modern systems support preemptive scheduling of parallel activities, commonly named as threads. This does not free the application from actually creating and maintaining these parallel activities, but it allows an abstract usage regardless of the particular underlying hardware. Multithreading is still the major parallelization paradigm, support by standardized libraries and virtual runtime environments. While early multi-threading libraries performed the task scheduling by them, it is meanwhile common to map application-level parallelism directly on to operating system threads.

Running more and more software in such a multi-threaded fashion leads to different kind of processing workloads, all with different memory access patterns. One example is virtualisation products, where a highly varying load from the guest operating system is mapped to one operating system process in the host system. Iyer et al. 6 predict this to be a major problem for optimal resource utilization. As one possible solution, they propose the prioritization of some threads on hardware level. This allows an optimized usage of cache space and memory bandwidth, but demands according hints from the operating system.

On application level, there is a variety of possibilities to express concurrent activities. They can be roughly categorized in dedicated parallel programming languages and sequential language extensions.

For the large base of sequential programs written in C- or Java-style languages, it is common to use libraries or abstractions for threads. One popular example is the OpenMP language extension; other examples are Cilk and Concurrent C.

More radical approaches criticize threads as wrong kind of abstraction, and rely instead on approaches such as pure data-parallel programming, functional programming or reactive programming. A third class of researchers elaborates on the notion of domain-specific languages, in order to encapsulate the parallelization in high-level language constructs. Popular examples in this category are SQL, MapReduce,
or Simulink. It remains an open research question which of these approaches is most suitable for the programming of scalable commodity applications based on thousands of execution units. In fact, most implementations still end up in mapping the activities to operating system threads. Alternative solutions mostly demand a virtual runtime environment for the context switching.

In general, application developers have to decide upon the degree of control they want to have, starting from low-level locking primitives up to implicit parallelism in functional languages. The low-level approach is still the dominating idea in high-performance parallel computing, since it is the only way to get maximum performance. The more abstract solutions are about to become more popular for the broader mass of developers. The choice also influences the design of compatible debugging facilities—low-level parallelisation support demands a relevant support for investigating problematic parallel activities.

From a theoretical perspective, the best approach for a parallel application is the usage of a dedicated parallel programming language. Many past initiatives in high-performance computing tried to introduce compatible solutions, in order to get the maximum benefit from the given compute power. This led to a long history of libraries and parallel programming languages, all trying to abstract basic concurrency and synchronization issues for the developer. A few of them remained successful, most of them are forgotten. Future research for multi-core-enabled software needs to remember the successful and—especially—the failed attempts for the abstraction of parallel processing issues, in order to learn from the past. A number of language proposals are still promising, and now need to be considered for future programming concepts. At the moment, there is still no agreement about the feasibility of such languages for average industrial developers.

Beside the way of expressing parallel activities, it is also still relevant to work on scalable algorithms. Latest research on lock-free data structures and software transactional memory show that there is still a lot of potential in scalable algorithm and data structure design. The high-performance computing field has a long tradition in this area, such as with linear algebra computation, spectral methods, matrix calculations, atmosphere modeling based on partial differential equations, VLSI floor plan optimization or graph traversal. It is therefore necessary to consider this huge body of knowledge for the upcoming era of multi-core processors.

11. COMPUTER EDUCATION: FOCUS ON CONCURRENCY

Extracting increased single-program throughput from this newly provided hardware will require far more programmer intervention than any other recent processor advancement. While many advances have been made over the years in parallel programming tools, it is still necessary for programmers to have a working knowledge of fundamental concurrency concepts to make productive use of multiple processors. Parallel programming practitioners in the past have been limited to a small, specialized community of computer scientists, as access to multiprocessor systems was only available to those working in the research departments of large corporations, universities, or national laboratories. Because of this, the curricula of most computer science departments have only taught parallel programming as an advanced elective course, if at all.

The current computer science workforce is not yet equipped to handle this shift in skills, and industry recognizes this. As part of their “Terascale Initiative”, Intel has put out a call to programmers and educators alike to expand their skill sets and their curricula to include multiprocessing. These concepts are becoming increasingly important in ensuring the short-term and long-term successes of our students.

Instead of spending time on teaching students specialised parallel programming languages, focus can be on having students learn and apply principles of concurrency within languages with which they are already comfortable, such
as Java and its object-oriented use of threading. By giving students a clear process for exploiting parallelism and practice at implementing it, they will gain confidence in taking advantage of new multi-core hardware. The curriculum should focus on to provide students with the skills they will need to succeed as we enter the multi-core era.

12. SUMMARY

Multi-core architectures are the state-of-the-art in IT hardware. Computers of the future will contain thousands of execution units per chip, either homogeneous or specialized for particular purposes. The parallelisation of application workload is about to become the most relevant strategy for speedup and scale up in every kind of application. Since single thread performance is no longer given for free, it is unavoidable to think and program in a parallel fashion.

The future research for multi-core enabled applications must be driven from two sides – a better expression of hardware, and a better design of software. The increasing variety of CMP processor architectures must be abstracted in better models, in order to allow an appropriate software design. Software on the other side must become analysable and designable according to the parallel workload it produces. Some applications will be scalable by default - 3D graphics, scientific computing or high-throughput server computing. The interesting challenges are all the other ones.

REFERENCES