Karlsruhe Institute of Technology

A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators

Rainer Buchty
Vincent Heuveline
Wolfgang Karl
Jan-Philipp Weiß

No. 2009-02

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.emcl.kit.edu

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
No. 2009-02

Impressum

Karlsruhe Institute of Technology (KIT)
Engineering Mathematics and Computing Lab (EMCL)

Fritz-Erler-Str. 23, building 01.86
76133 Karlsruhe
Germany

KIT – University of the State of Baden Wuerttemberg and
National Laboratory of the Helmholtz Association

Published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de .

A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators

Rainer Buchty[1], Vincent Heuveline[2], Wolfgang Karl[1] and Jan-Philipp Weiss[2,3]

[1] Chair for Computer Architecture, Institute of Computer Science and Engineering
[2] Engineering Mathematics Computing Lab
[3] Shared Research Group New Frontiers in High Performance Computing
Karlsruhe Institute of Technology, Kaiserstr. 12, 76128 Karlsruhe, Germany

Abstract. The paradigm shift towards multicore technologies is offering a great potential of computational power for scientific and industrial applications. It is, however, posing considerable challenges to software development. This problem is impaired by increasing heterogeneity of hardware platforms on both, processor level, and by adding dedicated accelerators. Performance gains for data- and compute-intensive applications can currently only be achieved by exploiting coarse- and fine-grained parallelism on all system levels, and improved scalability with respect to constantly increasing core counts. A key methodology is hardware-aware computing where in all production steps explicit knowledge of processor, memory, and network details is profitably utilized.
In this work we provide a survey on current multicore and accelerator technologies. We outline architectural features and show, how these features are exposed to the programmer and how they can be beneficially utilized in the application-mapping process. In particular, we characterize the discrepancy to conventional parallel platforms with respect to hierarchical memory sub-systems, fine-grained parallelism on several system levels, and chip- and system-level heterogeneity. We motivate the necessity of hardware-aware computing and summarize the challenges arising from high-performance heterogeneous computing. Furthermore, we investigate the interaction of hardware and application characteristics for selected applications in numerical simulation.

Keywords: Multicore processors; accelerators; heterogeneity; hardware-aware computing; performance optimization; parallel programming; numerical simulation

1 Introduction and motivation

During the 1990s and early 2000s, processors faced an enormous performance gain mainly due to the steady increase of clock rates; the associated speed-up for applications did not require any dedicated programming effort. This free lunch, however, did come to a grinding halt due to several technological issues such

as heat dissipation, leakage currents, and signal run-times. Even if these issues were solved, we still face the memory gap and synchronization constraints. With Moore's law still being valid, this observation led to a new paradigm: thread and data level parallelism based on replicated cores. For at least some more years, we will see exponentially increasing performance, where Moore's law on transistor density is no longer leading to more and more powerful unicore processors, but is rather translated into a corollary called *Moore's cores*: core count is doubling with every new generation. The multicore idea brings a duplication of processor cores running at lower clock with improved power efficiency. But now, further performance increase is based on extended parallelism with all applications and programmers affected.

Already today, multicore processors and accelerators offer a great potential of theoretically available computing power. And the future is about to promise further progression. Parallelism is spreading across several system levels, and data is held disjointly or redundantly in several stages of the memory sub-system ranging from small, therefore near and fast, to large, i.e. remote and slow memories. But while the cores are getting faster and faster, it becomes more and more difficult to "feed the beast" by quickly moving data in and out. Several concepts have been developed to mitigate the memory wall. On multicore CPUs, data is kept in a nested hierarchy of caches for maximizing data reuse by means of automatized and optimized protocols. Steadily increasing cache size has the implication that chip real estate is reserved for cache and less space is available for computing operations. Hence, more power is required for fewer flops. With the multicore concept and upcoming tiled manycore architectures, the problem of memory access becomes even more pressing. Solutions as pursued today are adding further hierarchy levels, e.g. by sharing memory resources between a number of cores. On Graphics Processing Units (GPUs), one uses fast context-switching of light-weight threads in order to hide latencies. According to Little's law, the latency-bandwidth product is giving an upper bound for concurrency [1]. A further issue with respect to reliability of hardware results from further miniaturization and a correspondingly increasing susceptibility to transient errors.

Current products include multicore CPUs (like IA32, x86-64, PowerPC, SPARC, or IA64), massively parallel GPUs, heterogeneous multicore platforms like the STI Cell processor, user-configurable Field Programmable Gate Arrays (FPGAs), coprocessor boards like ClearSpeed's line of products, and hybrid platforms like the Convey HC1 hybrid core. In the near future, other concepts like Intel's Larrabee or AMD's Fusion are expected to further diversify available products. Manycore devices like Intel's Single-chip Cloud Computer (SCC) [2] are subject to contemporary industrial research. Current approaches and technological solutions comprise a down-scaling of homogeneous computing clusters (e.g. Larrabee, SCC), heterogeneous and hierarchical architectures (Cell, Fusion), SIMD-array processors (GPUs, ClearSpeed), peer-to-peer connected processors (HyperTransport, QPI), hybrid platforms with accelerators/GPUs attached via PCIe, dedicated architecture integration (Convey

HC1, socket-replacement FPGAs), and hardware support for the programming model (e.g. transactional memory). In all areas, the hardware vendors are pushed towards novel techniques that promise further performance increase. Permanent innovation is the only sales pitch.

As a consequence of different ideas, we are facing diverging approaches in technology with different processing models and accordingly different programming approaches and environments. Each offered device comes up with its own characteristics and specific features. For the user, this translates to different effort, extra learning curves, isolated benefits, and unexpected drawbacks. The devices mainly differ in the configuration and arrangement of functional and control units, and the data flow from main memory to the compute cores is organized differently. As a consequence instruction sets and the generic or vendor-specific programming concepts differ. In some programming approaches, only parts of the hardware structure are exposed to the user and are accessible by the programmer. For some platforms parts of the internal processes are automatized by means of the runtime system and protocols, for other devices many things are left to the programmer – giving more flexibility but also more responsibility.

The great challenge is to exploit the available parallel potential for speeding up or even for enabling large-scale numerical simulation, image processing, real-time applications, or business-critical processes. Numerical simulation on high-performance computers has evolved to be the third pillar for gaining scientific cognition – besides theory and experiment. But for attaining maximal benefit from technology, mature parallel concepts need to be developed and utilized. We still have to explore suitable software frameworks and learn how to use current technology. Resulting from wide variety and fast advancements, first standards are about to evolve just recently such as e.g. OpenCL [3]. The experience over four decades in parallel programming is proving that there is a hard job to be done, and the general breakthrough is still to come. Due to the absence and impracticality of parallelizing compilers and the need for manual performance-tuning, all software-development processes need to be adapted to the new prerequisites: all kinds of multi-level parallelism, finer granularities, and a nested hierarchical memory system combined with software- or user-managed data transfers need to be exploited and expressed by the algorithms. Keeping the future perspective in mind, applications need to be designed with respect to scalability across several processor generations and concepts. Whether a parallel application can be mapped most profitably to a particular parallel platform while still keeping scalability potential depends on the characteristics and communication pattern of the underlying parallel algorithms. Best performance can be achieved for uniformly structured applications with high arithmetic intensity (ratio of performed flop per transferred byte), optimal locality, uniformity of operations, low bandwidth requirements, and the possibility to agglomerate or divide sub-blocks of data and computation over a large variety of granularities. Since these prerequisites are unrealistic for real application scenarios, the actual structure and properties of the algorithms need to be exploited at its best by organizing all data and computations in a platform-specific manner. However,

it is still an open question regarding the right platform to use and the right paradigm: as long as these questions are unresolved no one will invest the necessary amount of money, time and resources. Isolated approaches and specific solutions are likewise to be dead ends. On the other hand, these business risks do also come along with chances.

The multicore dilemma can only be overcome by holistic approaches in an interdisciplinary environment. For all steps of the full simulation cycle comprehensive hardware knowledge is required in order to achieve optimal throughput. A considerable trade-off exists between performance, productivity, portability, and maximal flexibility of an implementation, and its goal needs to be set individually as the design and parameter spaces are high-dimensional. Hardware-aware computing is a multi-disciplinary approach to identify the best combination of application, physical models, numerical schemes, parallel algorithms, and platform-specific implementations. Design goals are maximal flexibility with respect to software capabilities, mathematical quality, efficient resource utilization, performance, and portability of concepts.

The goal and the contribution of this work is to provide an overview of current technologies, their basic features, and their potential for deployment in high-performance computing and numerical simulation. We want to point out how corresponding parallel programming environments can expose or hide important hardware details and how applications and algorithms can be mapped to a considered device. Furthermore, we show basic application characteristics and the linkage between mathematical modeling, parallel implementation, and quality of the results. In addition, drawbacks and impact of algorithms and implementations are worked out specifically. Last but not least, we would like to give some advice to the users for their difficult decisions regarding appropriate choice of hardware platforms and programming environments that give the best fit for their application requirements and the amount of necessary work for achieving desired performance.

In recent years, an unmanageable amount of research work around specific multicore and accelerator technologies has evolved. While some papers describe architectural concepts [4], parallel programming methodologies and environments [5], most work focuses on selected applications and their dedicated implementation on specific architectures. Some impressive work is done by comparing basic application kernels on a multitude of multicore platforms [6–8]. In [9], an irregular and data-intensive application is evaluated on different multicore platforms. [10] details the design constraints for multicore and manycore architectures; furthermore, the author outlines the drawbacks of OpenMP, MPI, and hybrid MPI/OpenMP approaches for chip-level multiprocessing. The future of parallel processing landscape has been researched by an interdisciplinary team at Berkeley [11]. However, no general survey exists that combines aspects of applications, algorithms, programming approaches and platforms in a unified manner. In our presented work we try to tackle this task while concentrating on aspects of high-performance computing and numerical simulation.

This paper is organized as follows: Section 2 summarizes basic features and concepts of current multicore technologies. In Section 3 we provide an overview of contemporary multicore implementations. Section 4 gives a concise insight to parallel programming environments. In Section 5 we detail the necessity of hardware-aware computing and outline basic characteristics of selected applications. We conclude in Section 6.

2 Hardware features

Technological innovations across several decades have brought up ever more powerful processors; multicore processors are setting new milestones in these developments: performance is now based on thread-level and data-level parallelism. Per-thread performance will remain essentially static for the next processor generations, hence, software developers are challenged by increasing the number of threads in their code. In the following, we give an overview of the technological principles employed in multicore CPUs, how they influence program generation, and what challenges are to be expected.

Continuity of Moore's law is still giving reduced chip size for the same amount of functionality, or increased functionality at the same chip size. The rising number of available transistors can be used to build more components and more complex components on a single die. Either the control logic or the size of caches can be enhanced, or, as described, the number of complete cores on a chip can be further increased. **Replication of cores** can be done in two basic ways, either building multicores from few, quite complex cores, or less complex ones but many of them. As of now, we see both approaches in the field, e.g. in case of IA32-derived multicore processors currently offering two to eight cores, or massive multicore processors consisting of rather lightweight processing elements as in the case of GPUs, which currently comprise somewhere between 240 and 1600 individual processor cores.

Multicore architectures add another level of complexity to the design of **memory hierarchies**. Unlike popular belief, multicores are not just scaled-down clusters, but have dedicated requirements to be considered when designing an appropriate memory subsystem. Hence, we currently see a multitude of different approaches ranging from individual caches and hierarchies shared between two or more cores to mixed approaches where some cores feature scratchpad memories, others caches, or combinations of both concepts. Larger on-chip **caches** organized in several hierarchy levels are the basic concept to hide memory access latencies, therefore most modern desktop and server CPUs typically have at least three independent caches: a data cache to speed up data fetch and store, an instruction cache to speed up executable instruction fetch, and a translation lookaside buffer to speed up virtual-to-physical address translation for both, executable instructions and data. For uniprocessor cores, typically the cache infrastructure is heuristically tuned towards a general optimum with regard to the processor microarchitecture based on given benchmarks. While the use of cache memory is implicitly transparent and therefore usually does not require

dedicated consideration by the programmer, for high-performance applications the source code should be tuned individually to given cache configurations: doing so improves data locality by more efficiently using the present cache resources. An even more efficient resource use can be achieved by abandoning the concept of transparent caches but instead use so-called **scratchpad memory** (SPM). It can be shown that for same memory sizes the SPM concept achieves better utilization and energy efficiency [12] compared to using caches. However, SPM breaks with the concept of transparency and requires dedicated consideration by the programmer who needs to take care of data transfers between fast but small SPMs and main memory. In the relaxed **stream-processing model** on the CUDA architecture, we also find local memory shared by the stream processors of a multiprocessor: data can be loaded to and stored from the shared memory at any time of processing and is handled like a user-managed cache. Automated caches are also available for reading and writing data [13]. Additionally, texture and constant caches are good for read-only caching. In addition to memory hierarchy, also **memory attachment** needs to be considered: since main memory is attached to multicore processors locally, we have non-uniform memory access (NUMA) for processors where the memory controller is included on-chip (Intel Nehalem, AMD K10, SPARC); in previous generations of Intel processors (Core microarchitecture: Xeon, Core 2) the memory controller was located on the NorthBridge and memory access was uniform. Hence, **memory-aware programming** is strongly advised to achieve maximum performance. The actual memory architecture employed in individual nodes and the used programming model might lead to hybrid approaches such as e.g. required when targeting a cluster comprising of individual Cell BE nodes. Memory alignment and proper coalescing of data transfers is highly relevant for bandwidth-bound applications. Software and hardware prefetching should be enabled by appropriate structuring of code.

In recent multicore-based systems parallelism is spreading across several systems levels. Some of them need to be carefully addressed by the programmer by decomposition of data and tasks, others can only be influenced implicitly. **Dynamic ILP exploitation** (instruction level parallelism) – as the lowest-level and important form of parallelism – is a common feature of microprocessors for over a decade. The second source of parallelism is based on **parallel execution on short vectors**. This concept is known as *SIMD Within A Register* (SWAR). Instructions are executed on a group of registers – typically of size 128 bit – at the same time. SWAR parallelism can be found on general purpose multicore-CPUs (SSE), Cell's SPEs and in ClearSpeed's accelerators. On Intel's planned Larrabee processor SWAR width will be extended to 512 bit. The third level of parallelism is hardware-supported multithreading – e.g. HyperThreading (HT), Symmetric Multi-Threading (SMT) – where single cores may appear as multiple cores by replicating control flow logic and sharing execution units in time-slicing mode. On another level, **SIMD processing** exploits further data parallelism by applying an instruction to all elements of a collection of data in parallel. The main advantage is that only the data-path needs to be replicated but not the

control unit. However, SIMD processing typically has poor support for control flow. SIMD functionality is the basis of modern GPUs (with control flow extensions) and the ClearSpeed accelerators (with additional SWAR parallelism). The fifth level of parallelism is **core-level parallelism**. In multicore CPUs, the same kind of core is arranged in groups of two, four, six, or eight and builds a homogeneous multicore processor with strong coupling. The next level of parallelism is **socket-level parallelism**: on a single node of a large computer cluster several sockets may contain various multicore CPUs or FPGAs as socket replacement chips. Further devices like Cell, GPUs, or ClearSpeed accelerators can be attached via PCIe connections. Different instruction sets and different interfaces put particular emphasis on the socket-level heterogeneity. The highest level of parallelism is provided on the **node level**. Up to hundreds of nodes might be connected by specific topologies, e.g. via Infiniband or Gigabit Ethernet, into a huge cluster.

In order to achieve maximum benefit, exploiting these levels of parallelism adds significant complexity to the overall architecture. On lowest level, ILP collects a bunch of techniques that empower parallel processing of control-flow instructions at run-time; it was added by extending the already existing principles, i.e. from scalar to superscalar pipeline to out-of-order execution. Now, the per-cycle performance has reached saturation, as it is increasingly difficult to extract ILP from instruction streams. Moreover, ILP is inefficient for many applications, e.g. in case of superscalar pipelining where it is difficult to predict code behavior in branches. Techniques like out-of-order execution lead to more complex logic contradicting energy efficiency. Hence, ILP is resource-intensive and accounts for considerable heat generation. For keeping the pipelines busy and fully-loaded, the prediction of branches is critical in the case of dependent jumps (e.g. loops and if-branches). Branch predictors can be static or dynamic, and need to be accounted for by program development. On modern GPUs, divergent threads are reordered into convergent blocks; other simplified cores like the Cell's SPEs do not support branch prediction: in this case loops need to be unrolled or branches need to be detailed with branch-hint instructions and directives. One interesting observation was that despite ILP exploitation unicore processors still could not be fully exploited. This led to the introduction of the so-called HyperThreading (HT) technology being the first widely deployed thread level parallelism (TLP) support in general purpose processors. For HT, basically only support for two thread states was introduced into the architecture by adding required registers and control logic. That way, memory access latencies can be hidden by fast context switching. Even more acceleration can be achieved with dedicated multithreading support, where also the remaining microarchitecture is multiplied to fully support a number of threads (Alpha, Sun Niagara) [14]. With increasing numbers of threads thread management also becomes increasingly costly, hence architectures for massively multithreading therefore employ hardware thread managers, like e.g. the NVIDIA CUDA architecture.

Considering memory interconnection of individual cores, multicore architectures can be categorized by a distinction into hierarchical designs, pipeline de-

signs, network-based designs and array designs. In a **hierarchical design**, several cores share tree-organized caches where the number of caches rises from root to the leaves: we typically find 3-level or 2-level hierarchies where the root represents the connection to the main memory. For further increasing core counts, cache coherency is posing considerable limitations on scalability since data may be kept in different vertical and horizontal levels at the same time. In **pipeline designs**, data is processed step-wise and data is routed from core to core sequentially where each core may accomplish different work. This was the former principle of graphics processors when dealing with the graphics pipeline. In **network-based designs**, cores and main memory are connected via a fast communication network. Network-based designs, like e.g. the Cell processor, do have benefits over hierarchical designs in case of a large number of cores. In **array designs**, several processing elements are connected in a specific topology where groups of cores typically obey a unified control flow. These groups read and write data from main memory in a coordinated fashion and can communicate directly with neighbors in their groups. Network-based and array designs are guiding the way towards manycore processors. In hybrid designs different approaches are merged into new ones. The CUDA architecture of NVIDIA GPUs, for instance, allows communication of thread groups via shared memory whereas in the ClearSpeed accelerator board data can be communicated across the linear so-called swazzle path. A different approach based on dynamic heterogeneity is provided by FPGAs where different logic cells, look-up tables, flip-flops, and built-in fixed-function and memory blocks are combined to build application-specific configurations.

With increasingly smaller integration, not only dynamic aspects from program behavior and memory interaction need to be considered, but also **transient hardware errors** resulting from high-energy particles and fabrication flaws. While certain applications, e.g. playback of streaming media, are not hampered by such errors, others are highly sensitive. Error Correction Code (ECC) is an error-detection and -correction protocol capable of detecting and correcting single-bit errors on the fly. Error-tolerance in electronic circuits can be introduced by adding redundancy to the system. So-called dual modular redundancy (DMR) can detect errors by comparison, but cannot correct them. By adding a further instance of the component (triple modular redundancy, TMR), a decision unit can correct errors. While complex cores offer error correction capabilities, similar mechanism on GPUs are under development. **Reproducibility of results** in floating-point-dominated numerical codes across diverse platforms is a major design goal. To that end, standards like the IEEE-754 norm define floating-point data formats, rounding modes, denormal treatment, and overflow behavior. However, not all available hardware implementations are fully compliant with this standard. The impact of different hardware behavior on numerical codes in a heterogeneous environment is described in [15]. A further technique of multicore processors is partial shut-down of unused cores which gives rise to further improved energy efficiency. For example, Intel's new research chip SCC enables fine-granular energy management. Another concept is to speed up some

cores by overclocking in case of imbalanced workloads due to non-multithreaded code (cf. TurboMode on Intel Nehalem).

3 Modern multicore systems

Already today we see a multitude of approaches towards multicore architectures – combining different approaches and technologies. They all integrate several computing cores onto the same die but still exemplify conceptual differences. Neither there is no hard definition of multi- vs. manycore, nor does homogeneity vs. heterogeneity lead to a proper differentiation of architectures and their specific fields of use. Current implementations of multicore processors mainly differ in the setup of the cores, their interaction, their level of coupling, their communication infrastructure and topology, their memory sub-system providing data, and their associated processing model. On top of this, different programming models can be implemented with a considerable impact on the actual appearance and the level of hardware abstraction. In the end, only the programming model is the user view to the hardware – helping to simplify hardware handling or hampering detailed management of hardware behavior. In the following we like to give an overview of selected architectures, their genealogy and field of use, and specific properties such as used interconnection technologies.

The general multicore idea is to replace cores with complex internal organization by replicated cores with simpler organization. But the notion of a core is imprecise. Some say, it is a core if instruction flow can be executed independently. Others define a core as a single unit producing results by means of integer or floating-point computations on its own registers where data is accessible from peripheral devices like main memory. While some cores are able to run their own OS, other cores are orchestrated in a SIMD-like fashion. The cores typically interact via shared memory, message passing, DMA transfers, or via SIMD access to a global memory. Keeping these differences in mind, we currently find two to eight powerful cores in general purpose IA32, x86-64, or related CPUs. The Intel Larrabee processor will probably have 32 cores (based on a simpler core design), while the Intel Rock Creek research chip has 48 cores in a cluster-like network on a single die. The ClearSpeed accelerator board has 192 cores, while current GPUs from NVIDIA have 240 cores (Fermi will have 512 cores) and AMD/ATI GPUs have up to 1600 cores (with different meaning and capabilities).

What we see in the field of general-purpose multicores is typically a homogeneous approach, i.e. the replication of individual and identical general-purpose uniprocessors. Heterogeneous architectures, in term, combine cores of different ability, e.g. dedicated network or cryptographic processors, or a balanced set of cores of different computational power. Dedicated accelerator architectures provide powerful computing engines that are used for complementing existing systems and enlarging their capabilities. They are targeting specific functionalities e.g. “supercomputer on a chip” (NVIDIA Tesla, Cell BE), floating-point accelerators (ClearSpeed), or configurable logic cells (FPGAs). Currently, there are two opponent possibilities for multicore configurations: you can either build

larger and more complex cores with a lot of control logic, where the core count is kept comparably low – this circumvents the complexity wall by exploiting coarse-grained parallelism –, or you can build a lot of simpler cores with a high core count and possibly less reliable features. Retaining core complexity is owed to the fact that many applications are dominated by sequential parts and constrained by bandwidth bottlenecks. Huge on-chip caches and complex prefetch and control logic shall alleviate possible deficiencies. The configuration and size of the cores in size and capability has a direct impact on possible speedups [16]. A possible approach to reduce complexity is to retreat from out-of-order superscalar architecture and build simpler and faster, 1-wide in-order processors with high degree of speculation. As a future direction one has to explore system integration and SoC systems in order to pave the road to exaflop computing and overcome the power constraints. We will possibly see supercomputers on a chip: future graphics chips like the NVIDIA GT300 (Fermi) or Intel's SCC [2] are upcoming examples.

General-purpose multicore architectures typically evolved from general-purpose uniprocessors, therefore employing a low number of rather heavyweight, highly complex unicores and big nested caches organized in a two- or three-level hierarchy with full cache coherence across all cores in a node. Focus of these is typically multitasking environments where the use of several identical cores not only increases responsiveness of the overall system, but also simplifies task and thread management. **Intel** multicore CPUs have been built upon various microarchitectures (MAs) altering between short pipelines – with high IPC rates and moderate clock rates – and long pipelines – with a simple architecture, low IPC rates but high clock rates. The latest Nehalem MA is characterized by an integrated memory controller, return to HyperThreading, and the QPI interconnect to CPUs in the same node. It supports a three-level cache hierarchy with private L1 and L2 caches, and a shared inclusive L3 cache. Intel Itanium architecture (formerly called IA64) is basically different from common x86. It is based on explicit and static ILP where the main responsibility is left to the compiler. **AMD** processors based on the AMD64 ISA (x86-64) are basically 32-bit processor with 64-bit register extensions. The AMD Opteron processors possess an integrated memory controller supporting DDR, DDR2, or DDR3 SDRAM. This concept reduces the latency penalty for accessing the main RAM and eliminates the need for a separate NorthBridge chip. Each CPU can access the main memory of another processor – transparent to the programmer – via its HyperTransport links. The recent approach to multi-processing is not the same as standard symmetric multiprocessing; instead of having one bank of memory for all CPUs, each CPU has its own memory, resulting in a NUMA architecture. **IBM**'s Power architecture makes use of an approach called reduced instruction set computer (RISC) to process instructions. Power processors typically run at a higher clock rate than their counterparts from other vendors. The 64-bit Power6 in-order dual-core processor has two superscalar 2-way SMT cores served by an 8 Mbyte shared L2-cache. There is a two-stage pipeline supporting two independent 32-bit reads or one 64-bit write per cycle. There is also support for

up to 32 Mbyte of extern L3 cache providing data at 80 Gbyte/s. The Power7 with up to eight cores will be released in 2010. There are 12 execution units and 4-way SMT per core and VMX vector capabilities. Data is cached in private 256 kbyte L2 caches per core with lower latencies and 32 Mbyte L3 embedded DRAM (eDRAM) on chip. There will be maximal performance of 258.6 Gflop/s per chip. As an example of **Sun**'s SPARC architecture, the UltraSparcT2 Plus (Victoria Falls) is a highly multithreaded processor with 8 cores running up to 64 hardware-supported threads per core with up to 1.6 GHz. Each core has two integer ALUs and a fully pipelined floating-point unit. It has two integrated dual-channel FBDIMM memory controllers aiming at superior bandwidth.

Contrasting the main-stream multicore processors, alternative designs rely on simpler cores (mostly RISC) with less complicated and smaller caches or local stores, and more floating-point units. **MIPS processors** have their fundament in the embedded area, but there were ambitious projects to make a stand in the HPC domain. **SiCortex** has created a radical example of a tightly integrated Linux cluster supercomputer based on the MIPS64 architecture and a high-performance interconnect based on the Kautz digraph topology. Its multicore processing node integrates six power efficient MIPS64 cores, a crossbar memory controller, an interconnect DMA engine, and controllers on a single chip. Up to 972 6-core units result in 8.2 Tflop/s of peak performance. The **Cell Broadband Engine** (BE) is a processor architecture relying on innovative concepts for heterogeneous processing and memory management. Specialized cores are added to a main unit motivated by increasing both, computational power and memory bandwidth. The main unit of the Cell BE is the Power Processing Element (PPE) running at 3.2 GHz. The PPE mainly acts as controller for the eight Synergistic Processing Elements (SPEs) which usually handle the computational workload. The SPEs are 128-bit SIMD processors with 128 entry register files and 256 kbytes local, software-controlled memory called Local Store (LS). Data is transferred by means of user- or software-controlled Direct Memory Access (DMA). Each SPE has seven execution units and reaches 25.6 Gflop/s performance for single precision (SP) instructions and 12.8 Gflop/s double precision (DP) performance in the PowerXCell 8i. The eight SPEs are connected via the Element Interconnect Bus delivering aggregate peak bandwidth of 204 Gbyte/s and an aggregate performance of 204.9 (102.4) Gflop/s. Theoretical peak bandwidth to the local main memory is 25.6 Gbyte/s. **ClearSpeed** offered accelerator boards attached by PCIe with fast SIMD processors particularly designed for double-precision applications. The control unit of the CSX700 dispatches instructions to 192 processing elements (PEs). The Single-chip Cloud Computer (SCC) **Rock Creek** is an experimental manycore CPU from Intel that mimics cloud computing at the chip-level by a network of 2-core nodes. It is a cluster-on-a-chip system where each of its 48 Pentium55C-like cores can boot their own operating system. Its 48 IA32-compatible cores are organized into 24 tiles with their own router and message-buffer. The tiles are organized in a 6-by-4 mesh network with 256 Gbyte/s bisection bandwidth. It will be fed by four DDR3 memory controllers. Hardware-cache coherency is replaced by software-managed cache

handling. Further feature is a fine-grained power management scaling from 25 to 125 Watt.

The importance of **Graphics Processing Units** (GPUs) in HPC is growing fast mainly due to the associated computing power and bandwidth that outpace values known from general purpose CPUs by at least an order of magnitude. The main reason for this development is that more transistors are devoted to data processing rather than data caching, flow control, and error correction. Furthermore, GPUs are comparatively cheap due to their high volume. The basic functional block of modern GPUs consists of several stream processors with powerful SIMD elements coupled to texture units and several types of caches. The pool of available floating-point engines can be allocated to particular types of processing tasks as needed. Drawbacks arise if the SIMD approach is violated by heavy branching or non-uniform data access patterns, but with every new generation these obstacles seem to be more and more resolved. It has to be stressed that the huge computing power of GPU comes up with increased demand for power, although the overhead of complex logic design of CPUs can be avoided. **NVIDIA** GPUs are abstracted by means of the Compute Unified Device Architecture (CUDA) and associated programming model. The basic difference to previous GPU generations is that it supports scattered reads from arbitrary addresses in memory, it provides shared memory for communication amongst threads, and supports integer and bitwise operations. NVIDIA's GT200b graphics chip (10 series) is built of 10 Thread Processing Clusters (TPC) with an L1 texture memory cache, texture units, and three processor units called a streaming multiprocessor (SM). Each SM has eight stream-processing units (SP) sharing a small piece of 16 kbyte RAM which is the basic difference to the GeForce 8 and 9 series. Altogether, each TPC has 24 SPs and eight texture-filtering units or texture-mapping units (80 in total). An SP is a fully pipelined, single-issue, in-order microprocessor equipped with two ALUs and a SP FPU working on 2048 local registers of 8 kbyte size. Besides, the ALUs basic functionality is executing one FMAD per cycle. A super-function unit executes more sophisticated operations like root extraction or supports the calculation with up to four FP MULs. This results in an average SP performance of 3 flop/cycle in single precision. With 1296 MHz clock rate overall SP performance equals 933 Gflop/s. Each SM has a 64-bit floating point unit (30 units per chip) that can perform one FMAD per cycle in accordance with IEEE 754R. Here, the overall performance sums up to 77.76 Gflop/s. The new GeForce 300 series is to be released in 2010: it will feature a cGPU design that is much closer to traditional CPUs and will be radically different from previous generations. The GT300 chipset will possibly use MIMD instead of SIMD. **AMD/ATI** has introduced the RV870 graphics processor built of 20 SIMD cores with 16 stream processors per core. Apart from NVIDIA, AMD/ATI's stream processors have a superscalar design where each processor incorporates five ALUs that are served by automatic HW-managed sharing of scalar instruction streams. Four of these ALUs are general-purpose ALUs, the fifth ALU is a special-purpose ALU capable of executing complex instructions. Besides the ALUs, each shader processor also contains a branch-

control unit and an array of general-purpose registers. Each core is serviced by dedicated logic and has four texture processors and an L1 cache at its disposal.

Field Programmable Gate Arrays (FPGAs) are somewhat in between Application Specific Integrated Circuits (ASICs) and processors. From ASICs they have borrowed fine-granular use of logic resources and the integration of complex logic functions. For the major part, FPGAs are designed for domain-specific purposes. On the other hand, FPGAs are programmable (i.e. configurable) by the user and they are incorporating standardized and tested products. In today's platform FPGAs, a lot of dedicated functional units are integrated: MACs and DSPs are speeding up and simplifying specific functionalities whereas block-RAMs (in size of a typical L2-cache) enable local storage of larger portions of data. But compared to typical microprocessors, FPGAs run at a much slower clock rate, typically only a few hundred MHz. Convey offers a hybrid Intel Xeon / FPGA system based on a coherent memory space where the alien FPGA programming approach based on hardware-description languages is replaced by application-specific instruction sets. In an industry-standard programming approach it is left to the compiler to schedule operations either to the FPGA or the CPU.

4 Parallel programming models and environments

An interesting observation with current approaches towards multicore systems and especially heterogeneous parallel systems is the fact that established programming models do not suit the specific needs of such systems: while high-level parallel programming models deliver an abstract view as required by today's complex applications, they do not feature fine-grained control over hardware mapping, resulting in poor hardware resource use. Strict hardware-aware approaches, in term, enable such fine-grained control but will put the focus of application programming to hardware mapping. As a result, certain hybrid systems emerged during the last couple of years, aiming at a beneficial combination of both aspects.

Conventional **parallel programming models** can be split into the following three fields which are shared-memory, message-passing, and data-parallel approaches with according standardized programming environments like (in order of fields mentioned) OpenMP [17], MPI [18], or High-performance Fortran [19]. These programming models focus on exploitation of certain levels of parallelism, support only marginal flexibility, and typically do not enable fine-grained architecture mapping. Of the mentioned programming models, **message-passing** enables the highest level of controlling the architecture mapping of an application. However, it is also the model which forces the programmer into detailed partitioning and orchestration with respect to a given system infrastructure. Heterogeneity of the hardware is typically not expressed within MPI. Certain flexibility is supported by **shared-memory programming model** OpenMP, which allows dynamically changing the number of threads to be created. PGAS (Partitioned Global Address Space) [20] is a programming model that defines a

global address space on a possibly distributed system. It focuses on reference-locality exploitation: with PGAS portions of the shared memory space may have an affinity for a particular thread. Examples of the PGAS model are Unified Parallel C (UPC) and Co-array Fortran, but also recent and industry-driven approaches like Chapel (Cray) and X10 (IBM).

Especially heterogeneous platforms require fine-grained ways of matching application code to the given platform on lowest hardware level: a most prominent example of such is the Cell BE [21], which forces the programmer to explicitly partition the program into individual chunks to be executed on the single vector units called SPEs. Here, communication must be explicitly formulated using so-called **DMA streams**, i.e. enabling the individual SPEs to fetch required data and write back result data. This careful matching of computation and communication may lead to speedups of orders of magnitude compared to a naïve implementation. It is however a quite tedious, error-prone, and time-consuming task. **Hardware-aware programming techniques** are also required on conventional homogeneous multicore platforms in order to minimize communication overhead, e.g. by optimizing data locality and appropriate prefetching. With current and upcoming multicore architectures this becomes even more prevalent as such architectures feature distinct schemes of cache sharing and interconnection technologies.

Focusing on programming scenarios where a control thread forks potentially accelerable computation threads, certain industrial support exists for host/accelerator platforms. One well-known example is NVIDIA's **CUDA** approach [22]. Programming-wise, CUDA features an extension to the C programming language enabling programmers to easily write data-parallel programs to be either executed on conventional general-purpose processors or in a data-parallel manner on GPU hardware. This approach is picked up and extended by OpenCL [3], which offers a similar C extension and an according run-time system, but is not limited to a single vendor's hardware: with OpenCL, a first standard and unified interface for kernel invocation on a multitude of hardware devices is now available in its initial release. In theory, it offers portability of implementations over a multitude of hybrid computing systems. But efficient utilization is tightly connected to platform-specific optimizations – contradicting performance portability. A related approach is the **RapidMind** programming environment [23] that is now being merged with Intel's **Ct** (*C for throughput computing*) [24], combining basic features of both. RapidMind specifically targets program development for and execution on dynamically changing heterogeneous systems. Intel Ct, in its pre-merged version, takes a similar approach for covering dynamic parallelism, but is more explicitly vectorized: C++ templates are used to describe so-called *throughput vectors* (TVEC) for vectorized operations. Compiler and runtime system divide the work into units and distribute the work units across the available processing elements, enabling dynamic changes to parallelism granularity by splitting or merging individual threads on demand.

A number of academic projects exist which specifically focus on **dynamic aspects** in heterogeneous systems. The EXOCHI project [25], for instance, de-

livers a system abstraction layer similar to that found in superscalar processors so that arbitrary programs may be distributed onto the various execution units. This is e.g. exploited by the MERGE project [26], which uses the EXOCHI back-end for executing parallel programs on heterogeneous parallel systems. A likewise ambitious project is IBM's LIME [27] targeting not only program execution on a given heterogeneous hardware platform but even on-demand synthesis of functional units when employing reconfigurable hardware.

One problem of all mentioned approaches is the necessity of additional runtime layers and often language-centric extensions. In addition, none of the mentioned approaches focus on application requirements, i.e. perform dynamic execution on heterogeneous parallel hardware with regard to required quality of service. Such could e.g. be time frames to be met, required minimum resolution of computations, or a minimum throughput to be achieved. It is, however, possible to carefully extend already present system layers to support not only dynamic execution but also employ required self-management in order to follow application requirements [28]. It can be shown that such an approach is inherently language- and OS-agnostic and not only compatible but even complementary to existing parallel programming models [29]. Exploiting existing features of compiler technology and binary formats leads to an easy and compatible way of describing application requirements to be evaluated and followed by the run-time system [30].

5 Hardware-aware computing and application cases

The transition from unicore to multicore and manycore comes along with a disrupture in well-established concepts and methodologies. In numerical simulation, the momentousness becomes prominently apparent: all parts of the application code need to be compliant with multi-level parallel units, a nested and hierarchical memory subsystem, and heterogeneous components of the compute units. All aspects need to be expressed explicitly in the algorithms and implementations since there are only little software support and a few mechanisms available that may assist automatic and optimized utilization of resources, as well as hide away hardware details without affecting performance. A basic observation in the landscape of parallel computing is the trend towards heterogeneous platforms consisting of hybrid techniques on the node level and heterogeneous approaches on the chip level. But the concept of heterogeneity still lacks expression within the algorithms and applications. This does not only mean offloading of compute-intensive parts to dedicated accelerator units, but rather a cooperative concept for a joint interaction of different entities within a hybrid system. Current programming techniques mainly rely on minimal-invasive approaches where local parts of the application are identified for acceleration and offloaded to particular compute engines like GPUs or Cell processors in a master-slave oriented way. For the overall benefit additional communication via narrow bottlenecks have to be considered. The difficulty lies in defining clear interfaces between different parts of the hardware that do not inherit any kind of performance bottleneck.

The major profit from experimental programming approaches by platform-specific solutions comes from learning for unified approaches in the future. Some issues are resolved within particular projects, but the corresponding solutions are isolated and application-specific. In the long run, there is a strong need for flexible, generic, and modular solutions. Portability is mainly excluded due to hardware-specific optimization strategies. Since the parameter space becomes more and more intractable, there is no way around autotuners. For selected kernels autotuning is giving impressive results and ensures portability across diverse platforms. Promising examples are ATLAS (BLAS) [31], FFTW (FFT), Spiral (DSP), OSKI (Sparse BLAS), and PhiPAC (BLAS). For a profitable deployment of modern concepts and technologies there needs to be a deep insight into the hardware and into the application characteristics. In general, this comprehensive task is no longer manageable by domain specialists, who usually stick to problem-adapted but hardware-agnostic coding. In order to overcome the described challenges and difficulties, there needs to be an holistic approach taking into account all aspects of physical problem description, mathematical modeling, numerical methods, parallel programming, efficient implementation, hardware design, tool support, performance analysis, code tuning, and assessment of the results with an interactive feedback on the full simulation and development cycle. In this section the corresponding aspects shall be detailed more precisely.

On all kinds of computer systems best performance can only be achieved by optimal utilization of the designated cores and functional units with only limited amount of data movement. The roofline model in [32] describes performance degradation for several processor types by conflicting or not obeying SIMD operations, ILP, and balanced FP operations. It further shows, that many application kernels with low arithmetic intensity are constrained by the DRAM bandwidth and by not utilizing prefetching techniques, memory affinity, NUMA effects, and unit stride memory access. The utilization of concepts for optimizing the arrangement of computation (like vectorization, loop unrolling, re-ordering, branch elimination), optimizing the data structures (array access patterns, memory alignment, coalescing of data) and the optimization of data transfers (blocking for spatial and temporal locality, caching and cache-bypassing, in-place algorithms) is of paramount importance for good performance of a particular application. But before optimizing existing code, problem modeling has to be reconsidered. Even more important for an optimal run time – and often neglected in actual practice – is the choice of the adequate mathematical and algorithmic models and the best available numerical solving scheme with respect to arithmetic complexity and parallel time complexity. The choice of the mathematical solution method is also fundamental with respect to robustness of the method and quality of the final simulation result. Current hardware trends highlight the necessity for intensive research on novel numerical techniques focusing on parallel preconditioning, parallel smoothers, domain decomposition approaches, and bandwidth-optimal numerical schemes in general.

For typical applications in numerical simulation, basic kernels have a computational intensity of only order one with respect to the problem size. Accordingly,

only a low fraction of peak performance can be achieved on most bandwidth-constrained systems. Hence, sole focus on compute complexity is inadequate since flops turn out to be for free but bytes are really expensive. Due to the unsurmountable memory wall the structure of the algorithms needs to be shifted towards reduced communication at the cost of additional and probably redundant computation. Since these issues become even more pronounced within the concept of multicore processors, there is an urgent demand for sophisticated methods for reduction of data transfers and increasing locality of computations. Memory transfer reduction strategies mainly comprise algorithmic re-arrangements, but also modified implementation mechanisms. They include restriction to single precision with a basic performance gain of a factor of two. Matrix operations on regular grids may be reduced to application of fixed stencil routines preventing matrices to be transferred. Temporal blocking techniques like loop skewing and circular queue [7], [33], [34] give significant benefits for stencil applications but are intrinsically based on explicit solution schemes – with all well-known drawbacks. In the spirit of overcoming communication bottlenecks, [35] proposes to exchange communication-bound kernels in sparse iterative Krylov-space solvers by matrix power kernels and block-operated orthogonalization with an associated reformulation of the algorithm. This method successfully shows how cost-efficiency of additionally performed computations can give considerable benefits by means of reduced bandwidth. In [36] a pipelined CG version is proposed that reduces the number of synchronization points by adding an additional scalar product. In this setting, several vector updates can be performed in parallel. Computational intensity of kernels in finite element methods (FEM) may be increased by using higher order elements and restriction to local dense matrices. Another approach to circumvent memory traffic on the implementation level is to avoid cache-write misses by utilizing SIMD intrinsics for cache-bypassing or to work with in-place implementations with a single in- and output array instead of out-of-place implementations [37].

Memory and data organization is of crucial importance for efficient performance. Data has to be transferred in huge contiguous chunks that are aligned to appropriate boundaries. Applications based on structured grids and regular data without indirect addressing are an essential part for predictable data access patterns. This is especially true for the Cell processor or other upcoming software-cache based architectures where each data transfer has to be user-controlled and -initiated. The multi-facetted exigencies for highly capable numerical simulation codes give a somewhat different picture of the organization of data structures. The simulation of turbulent flows in complex geometries or large-range effects in climate simulation or electrodynamics give rise to a discrete problem description based on unstructured grids in a finite element context. Only unstructured and problem-adapted locally refined grids based on deformable geometric primitives are able to resolve curvilinear boundaries, aggregated errors, and singularities in the solution. In this approach, careful application of goal-oriented adaptive refinement can increase simulation accuracy while keeping the amount of degrees of freedom at a moderate level. As a consequence data structures need to be

arranged in a flexible manner giving rise to linked lists, indirect addressing, non-contiguous memory access, pointer chasing and short messages in a distributed memory environment. In [38] the execution time of a typical application on unstructured grids is reported to be dominated by mesh-related operations at a fraction of more than 70 %. However, it is of general agreement that flexibility, generality and problem-adaptation in the FEM codes outweigh the associated disadvantages.

Hardware-aware computing not only comprises highly-optimized platform-specific implementations, design of communication-optimized data structures, and maximizing data reuse by temporal and spatial blocking techniques. It also relies on the choice and development of adequate mathematical models that express multilevel-parallelism and scalability up to a huge number of cores. The models need to be adapted to a wide range of platforms and programming environments. Data and problem decomposition on heterogeneous platforms is a further objective. Memory transfer reduction strategies are an important concept for facing the constraints of current platforms. Since hardware reality and mathematical cognition give rise to different implementation concepts, hardware-aware computing means also to find a balance between the associated opponent guiding lines while keeping the best mathematical quality. All solutions need to be designed in a reliable, robust and future-proof context. The goal is not to design isolated solutions for particular configurations but to develop methodologies and concepts that apply to a wide range of problem classes and architectures or that can be easily extended or adapted.

5.1 Application cases

Physical processes are typically described by systems of partial differential equations (PDEs) that can either be discretized by finite difference, finite volume or finite element methods. The discretization process results in sets of large and sparse discrete equations that express the local and global coupling of physical effects on a discrete level. A possible parallelization method relies on domain decomposition and domain sub-structuring methods where the computational domain (and hence the problem complexity) is divided into several subdomains. The associated local solutions are typically much simpler but need to undergo some interleaved global or neighbor-local coupling procedure. The second approach relies on parallelization of the basic building blocks from linear algebra that form the global solving scheme [39].

A typical model problem in numerical simulation is the solution of a Poisson problem. The resulting discrete linear system of equations can be solved by a wide variety of numerical solution methods differing in work complexity, general applicability, and degree of parallelism. These methods include Gauss-like methods of non-optimal complexity, direct solvers like FFT or multi-frontal methods, and a large class of iterative solvers like the Conjugate Gradient (CG) method. For improving work complexity in terms of necessary iterations, CG-like methods need to be accompanied with preconditioning techniques. Since preconditioning based on ILU and SOR-methods are highly recursive and sequential,

other approaches with less preconditioning efficiency but higher degree of parallelism need to be developed. Domain decomposition techniques are known not be scalable in an algorithmic sense, unless some coupling on a global coarse grid level is used. Asymptotically optimal multigrid methods rely on a combination of a hierarchy of nested grids. In both cases, operations on coarse grids yield poor parallel efficiency due to relative communication overhead, short loops and vectors, short messages, and kernel call overheads. The Hybrid Hierarchical Grid (HHG) approach described in [40] aims to close the gap between finite element flexibility on unstructured grids and the capabilities of multigrid methods on structured grids. A similar approach with distinction of structured and unstructured data and a generalized multigrid and domain decomposition concept is taken in the FEAST project [41]. Coprocessors like GPUs are integrated on the level of local subproblems for accelerating local multigrid problems. The limitations of this approach due to Amdahl's law are reported in [42]. While the subproblems of a vector-valued non-linear Navier-Stokes solver are accelerated by a factor of twelve, the global solver reaches a speedup of only two. A similar observation is done for a solid mechanics code [43].

Dense linear algebra (DLA) kernels have been subject of thorough investigation over a long period of time. Their basic implementation is provided by the Basic Linear Algebra Subprograms (BLAS) collection, a highly optimized and platform-specific API, and its parallel PBLAS version. Machine peak flop rates can be attained up to 60% to 90% with dense matrix kernels like the BLAS 3 based LINPACK benchmark. In order to face the changed situation in hardware technologies, the PLASMA project [44] aims at redesigning the DLA packages LAPACK and ScaLAPACK for shared memory computers based on multicore processors. Relying on tile algorithms based on fine-grained parallelism and directed acyclic graphs, performance benefits arise due to asynchronous out-of-order scheduling of operations. Data reuse is optimized by a pruned parameter search. The MAGMA project [44] pursues a similar approach for heterogeneous and hybrid architectures departing from mixed multicore and GPU systems. Within this approach new techniques for trading speed against accuracy are investigated. It shows that increased communication costs have to be reflected by software development.

Sparse matrix operation on dense vectors are of outstanding importance in numerical simulation. In iterative solvers, sparse matrix vector multiplication (SpMV) is the basic kernel that is repeated a hundred to thousands of times. The sparsity pattern is typically condensed into compressed formats. On unstructured grids, assembly of these matrices takes a remarkable amount of time. For unstructured sparse matrices the imbalance of line lengths is conflicting vectorization approaches. Due to limited data reuse, low locality and computational intensity only a low fraction of peak can be observed. On modern multicore platforms, performance numbers between 0.5 and 12 Gflop/s are observed on 4-core to 16-core systems utilizing automatically tuned implementations [7]. On recent NVIDIA GPUs, on-device performance for unstructured SpMV goes up to 17 Gflop/s in double precision [45]. Utilizing regular structure and symmetries in

fixed stencil operations, performance of highly-tuned implementations on modern multicore systems reaches between 2.5 and 16 Gflop/s on 8-core and 16-core systems [6, 37]. On NVIDIA GPUs on-device performance reaches 35 Gflop/s in double precision for 7-pt stencils [7] and 18 Gflop/s in diagonal SpMV format [45].

An involved model problem is the solution of the incompressible Navier-Stokes equations, a non-linear and vector-valued saddle-point problem. Typical Navier-Stokes solvers are based on Chorin-type projection methods with similar solution characteristics like Krylov-subspace methods. The major concerns are the nonlinear Newton-like procedure in the advection step and the solution of a Poisson equation for the pressure in each time step. Solutions by means of operator splitting give rise to explicit or implicit solution schemes. While explicit schemes can make use of temporal blocking methods reducing the bandwidth requirement, they are often impaired by time step limitations. Implicit methods are proven to be more robust and reliable and do not have time step constraints in most cases. As an alternative to Navier-Stokes-like problems on the modeling level, Lattice-Boltzmann methods (LBMs) can be utilized instead. LBMs are based on a mesoscopic particle description and hide the non-linearity in the right hand side of an explicit solution scheme based on equidistant cartesian grids, where all interactions are strictly local. Due to the structure of the data impressive results are obtained on accelerators like GPUs or Cell. Scalability to a huge number of cores is proven as well. *N-Body problems* and *molecular dynamics* describe the interaction of an agglomerate of particles or molecules by means of particle collisions, mutual reaction and external forces. Due to the large number of particles long-range-effects and fine resolution of the particles are usually canceled. N-body problems can be solved by numerically integrating up the differential equations of motion. But for a large number of particles these approaches become intractable and simplifications are required. Typical arithmetic complexity is of order $O(N^2)$ while the amount of data is only of order $O(N)$. Tree structured algorithms (e.g. *Barnes-Hut*) can reduce complexity to $O(N \log(N))$.

Genome sequenzing is a typical application in bioinformatics. Well-known algorithms for local sequence alignment and determination of similar regions in nucleotide or protein sequences are BLAST and *Smith-Waterman*. In financial mathematics rapid computation of exponentials and logarithms based on fixed and floating point binary coded decimal arithmetic is required – stressing advantages of FPGAs. The *Black Scholes* model for option pricing can be combined with Monte Carlo methods for numerical integration of prices for call and put options in a highly parallel manner. *Digital signal processing* (DSP) applications require high bandwidth but only limited accuracy handled in fixed-point arithmetics. Due to their specific capabilities FPGAS can give considerable benefits for applications like ray tracing, discrete cosine transform, rice coding, and convolution. Random number generation (Mersenne Twister) is another typical application field for FPGAs.

6 Conclusion

Driven by Moore's law and technological side-effects, we see a constant increase in parallelism and heterogeneity delivering ever rising performance rates. However, programmability of hardware devices and scalability of software solutions need to be further improved for a beneficial utilization of theoretically available performance. This collision of architecture complexity and complexity of harnessing its power is actively hampering development and acceptance of such systems: so far, certain promising processor technologies have been canceled (e.g. Sun Rock), delayed (e.g. Larrabee), or the manufacturer does not exist in its original form anymore (e.g. ClearSpeed, SiCortex).

As of now, there seems to be convergence towards a number of mainstream-acceptable concepts such as GPU-accelerated clusters. However, as long as there is no true convergence towards unified programming approaches that are applicable to a variety of different technologies and platforms, investments into particular solutions by means of money and manpower are still risky. Until then, much of the responsibility is put on the programmer. "Magic" compilers and auto-parallelization concepts are still out of reach. But the joint effort of participating disciplines can result in multicore-aware algorithms, multicore-enabled libraries, and multicore-capable tool chains. Some of the complexities can possibly be hidden in libraries; however, manual implementations are slow and expensive, and a new library is required for each architecture. This disadvantage might be overcome by autotuners searching parameter space for optimal performance configurations.

A lot of potential can be found within the algorithms. Strong research efforts are necessary for exploring new algorithms that are well-adapted to the prerequisites of emerging hardware. The main focus has to be set to communication-avoiding strategies based on data-locality and possibly replicated computations. Scalability towards upcoming architectures with ever-increasing core counts is a vital ingredient for all concepts. The approach of hardware-aware computing is trying to find a balance between structurally opponent cognition from best-performing implementations and the side conditions of maximally flexible software packages and mathematical quality. In the end, there is no way around changes in the modeling process, better algorithms, and platform-specific tuning. Restructuring of algorithms is strongly required for minimization of communication, couplings and load imbalance, and optimization of data locality. Efficient data management is the key to good performance in bandwidth-bound applications. Data operations need to be grouped and merged for minimizing memory traffic, and traffic needs to be re-structured and optimized by tiling, aligning, SIMDizing, and compressing.

Hence, many investments need to be made by all contributors: programming languages and environments need to be further developed for more support of the developers and hardware-reluctant domain experts. The full development and tool chain must be complemented by compilers, tools, operating systems, and runtime environments that give full support for platform-specific implementations without conflicting the deployment of portable concepts. Performance

bottlenecks need to highlighted and addressed. The algorithms have to be re-engineered with respect to multi-level parallelism, nested memory sub-systems, and heterogeneous platforms. Communication has to be minimized while keeping scalability over the next generations of hardware. Unified concepts and the convergence towards widely acceptable parallel frameworks are of critical importance. On the academia side, parallelism has to be taught in undergraduate courses across all disciplines in order to build a broad base for the next generation of programmers. A general view to the multi-facetted appearance of parallel thinking, high-performance computing, numerical simulation, multi-threaded algorithms, hardware characteristics, and the tool set has to be conveyed. Comprehensive software packages for the years 2015-2020 need to be designed now.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and the industrial collaboration partner Hewlett-Packard.

References

1. Bailey DH. Little's law and high performance computing. *RNR Technical Report*, 1997.
2. Intel *RockCreek* Single-chip Cloud Computer. `http://download.intel.com/pressroom/pdf/rockcreek/SCC_Announcement_JustinRattner.pdf` [December 20 2009]
3. Khronos OpenCL Working Group. *OpenCL 1.0 Standard*. `http://www.khronos.org/opencl/` [December 20 2009].
4. Ungerer T, Robič B, Šilc, J. A survey of processors with explicit multithreading. *ACM Comput. Surv.* 2003; **35**(1):29–63.
5. McCool, MD. Scalable programming models for massively multicore processors. In *Proc. IEEE* 2008; 816–831.
6. Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms In *SC'07: Proc. 2007 ACM/IEEE Conf. on Supercomputing*, ACM: New York, 2007; 1–12.
7. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proc. 2008 ACM/IEEE Conf. on Supercomputing*, ACM: New York, 2008; 1–12.
8. Williams S, Carter J, Oliker L, Shalf J, Yelick K. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Proc. Int. Parallel and Distributed Processing Symposium*, 2008.
9. Van Amesfoort A, Varbanescu A, Sips H, Van Nieuwpoort R. Evaluating multi-core platforms for HPC data-intensive kernels. In *CF'09: Proc. 6th ACM Conf. on Computing Frontiers*, ACM: New York, 2009; 207–216.
10. Shalf J. The new landscape of parallel computer architecture. *Journal of Physics: Conf. Series* 2007; **78**.

11. Asanovic K et al. The Landscape of Parallel Computing Research: A View from Berkeley. *EECS technical report* 2006. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html` [Dec 20 2009]
12. Baskaran MM, Bondhugula U, and Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* 2008, 1–10.
13. Volkov V, Demmel J. LU, QR and Cholesky factorizations using vector capabilities of GPUs. *EECS technical report* 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html [December 20 2009]
14. Kang D, Liu C, Gaudiot JL. The impact of speculative execution on SMT processors. *Int. J. Parallel Program.* 2008; **36**(4):361–385.
15. Cleary AJ, Demmel J, Dhillon IS, Dongarra J, Hammarling S, Petitet A, Ren H, Stanley K, Whaley RC. Practical experience in the dangers of heterogeneous computing. In *PARA*, 1996; 57–64.
16. Hill MD, Marty MR. Amdahls law in the multicore era. *IEEE COMPUTER* 2008.
17. Dagum L, Menon R. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering* 1998; **05**(1):46–55.
18. *MPI: A Message-Passing Interface Standard (Version 2.2).* `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf` [December 20 2009]
19. Richardson, H. High Performance Fortran – history, overview and current developments. *TMC-261* 1996. `http://citeseer.ist.psu.edu/rd/0,174206,1,0.25,` [December 20 2009]
20. Carlson B, El-Ghazawi T, Numrich R, Yelick K. Programming in the Partitioned Global Address Space Model. *Supercomputing 2003.* `http://upc.gwu.edu/tutorials/tutorials_sc2003.pdf`
21. Gschwind M, Hofstee HP, Flachs B, Hopkins M, Watanabe Y, Yamazaki T. Synergistic processing in Cell's multicore architecture. *IEEE Micro* 2006; **26**(2):10–24.
22. Halfhill T. Parallel processing with CUDA. *Microprocessor Report* 01/28/08-01, 2008.
23. RapidMind Multi-Core Development Platform. `http://www.rapidmind.net/` [December 20, 2009]
24. Ghuloum A, Sprangle E, Fang J, Wu G, Zhou X. Ct: a flexible parallel programming model for tera-scale architectures. `http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf` [December 20 2009]
25. Wang P, Collins J, Chinya G, Jiang H, Tian X, Girkar M, Yang N, Lue GY, Wang H. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. *SIGPLAN Not.* 2007; **42**(6):156–166.
26. Linderman M, Collins J, Wang H, Meng T. Merge: a programming model for heterogeneous multi-core systems. *ASPLOS XIII* 2008; 287–296.
27. Huang S, Hormati A, Bacon D, Rabbah R. Liquid Metal: object-oriented programming across the hardware/software boundary. *ECOOP 2008.*
28. Buchty R, Kramer D, Kicherer M, Karl W. A light-weight approach to dynamical run-time linking supporting heterogenous, parallel, and reconfigurable architectures. *ARCS 2009.* LNCS 5467: 60–71.
29. Buchty R, Kicherer M, Kramer D, Karl W. An embrace-and-extend approach to managing the complexity of future heterogeneous systems. *SAMOS IX* 2009; LNCS 5657:226–235.

30. Nowak F, Kicherer M, Buchty R, Karl W. Delivering guidance information in heterogeneous systems. *PASA* 2010, to appear.
31. Automatically Tuned Linear Algebra Software (ATLAS). `http://math-atlas.sourceforge.net/` [December 20 2009]
32. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, ACM: New York, 2009; **52**(4):65–76.
33. Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K. Scientific computing kernels on the Cell processor. *Int. J. Parallel Program.* 2007; **35**(3):263–198.
34. Kamil S, Datta K, Williams S, Oliker L, Shalf J, Yelick K. Implicit and explicit optimizations for stencil computations. In *Proc. 2006 workshop on Memory System Performance and Correctness*, ACM: New York, 2006; 51–60.
35. Mohiyuddin M, Hoemmen M, Demmel J, Yelick K. Minimizing communication in sparse matrix solvers. In *Proc. Supercomputing Conf.*. ACM: New York, 2009; 1–11.
36. Strzodka R, Göddeke D. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines* 2006; 259–270.
37. Augustin W, Heuveline V, Weiss JP. Optimized stencil computation using in-place calculation on modern multicore systems. In *Proc. 15th Int. EuroPar Conf.* 2009; 772–784.
38. White BS, McKee SA, de Supinski BR, Miller B, Quinlan D, Schulz M. Improving the computational intensity of unstructured mesh applications. In *ICS '05: Proc. 19th annual int. conference on Supercomputing*. ACM: New York, 2005; 341–350.
39. Balay S, Gropp WD, McInnes LC, Smith BF. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern software tools for scientific computing*, Birkhauser: Boston, 1997; 163–202.
40. Bergen B, Gradl T, Hülsemann F, Rüde U. A massively parallel multigrid method for finite elements. *Computing in Science and Eng.* 2006; **8**(6):56–62.
41. Becker C, Buijssen SHM, Turek S. FEAST: Development of HPC technologies for FEM applications. *High Performance Computing in Science and Engineering 2007*. Transact. HLRS: Berlin, 2007; 503–516.
42. Göddeke D, Buijssen SHM, Wobker H, Turek Sa. GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. In *High Performance Computing & Simulation 2009*. Logos: Berlin, 2009; 12–21.
43. Göddeke D, Wobker H, Strzodka R, Mohd-Yusof J, McCormick P, Turek S. Co-processor acceleration of an unmodified parallel solid mechanics code with FEAST-GPU. *International Journal of Computational Science and Engineering*. 2009; **4**(4): 254–269.
44. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 2009; **180**.
45. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ACM: New York, 2009; 1–11.

Preprint Series of the Engineering Mathematics and Computing Lab

recent issues

No. 2009-01 Vincent Heuveline, Björn Rocker, Staffan Ronnas: Numerical Simulation on the SiCortex Supercomputer Platform: a Preliminary Evaluation

The responsibility for the contents of the working papers rests with the authors, not the Institute. Since working papers are of a preliminary nature, it may be useful to contact the authors of a particular working paper about results or caveats before referring to, or quoting, a paper. Any comments on working papers should be sent directly to the authors.