PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems

Siegfried Benkner
Sabri Pllana
Jesper Larsson Träff
University of Vienna

Philippas Tsigas
Chalmers University of Technology

Uwe Dolinsky
Codeplay Software

Cédric Augonnet
INRIA Bordeaux

Beverly Bachmayer
Intel GmbH

Christoph Kessler
Linköping University

David Moloney
Movidius

Vitaly Osipov
Karlsruhe Institute of Technology

PEPPHER, A THREE-YEAR EUROPEAN FP7 PROJECT, ADDRESSES EFFICIENT UTILIZATION OF HYBRID (HETEROGENEOUS) COMPUTER SYSTEMS CONSISTING OF MULTICORE CPUS WITH GPU-TYPE ACCELERATORS. THIS ARTICLE OUTLINES THE PEPPHER PERFORMANCE-AWARE COMPONENT MODEL, PERFORMANCE PREDICTION MEANS, RUNTIME SYSTEM, AND OTHER ASPECTS OF THE PROJECT. A LARGER EXAMPLE DEMONSTRATES PERFORMANCE PORTABILITY WITH THE PEPPHER APPROACH ACROSS HYBRID SYSTEMS WITH ONE TO FOUR GPUS.

• • • • • • Recent years have seen a proliferation of radically different computer architectures—including many-core CPUs, embedded CPUs, single-instruction, multiple-data (SIMD) instruction sets, the Cell Broadband Engine Architecture, and Intel's Many Integrated Core (MIC) and Single-chip Cloud Computer (SCC) architectures—and will see a fusion of different such architectures into hybrid (heterogeneous) systems. Also driven by the rapid succession of architecture generations (such as Nvidia GPUs) ensuring both reasonable performance as well as functional and performance portability between such hybrid systems are already pressing challenges to computer science research and engineering. Because of the large architectural parameter space, ensuring programmability and portability for such systems cannot be done manually, but must be handled or assisted automatically.[1] Many research projects are currently addressing these problems, including the European FP7 project PEPPHER (Performance Portability and Programmability for Heterogeneous Many-Core Architectures). For more information on PEPPHER, see the "PEPPHER Project Facts" sidebar. For information on related projects, see the "PEPPHER: Related Work and Projects" sidebar.

PEPPHER attacks the problems of performance portability and efficient use of heterogeneous systems at several levels. PEPPHER proposes solutions that involve a combination of static and dynamic scheduling, automatic adaptation of (library) components, compilation and transformation techniques, and a resource-aware runtime aided by performance information with feedback monitoring for gathering empirical performance information. Some of the PEPPHER contributions that we will discuss include:

- making (legacy) code written in existing programming languages and parallel APIs performance aware and portable through stepwise componentization;
- a component model with metadata for performance awareness and adaptivity;
- algorithms and data structures for hybrid parallel architectures, library-based performance portability through

PEPPHER Project Facts

PEPPHER (pronounced ''pep-fer'') is a European Union—funded (FP7) project that began in January 2010 and will run through 2012. The PEPPHER partners are the Universities of Vienna, Chalmers (Sweden), Linköping (Sweden), and Karlsruhe (Germany); INRIA Bordeaux; European small and medium enterprises Codeplay (United Kingdom) and Movidius (Ireland); and Intel Labs Europe (Germany). For more information, see www.peppher.eu.

PEPPHER: Related Work and Projects

Many current projects are concerned with aspects of multicore programmability. In contrast to other European projects—such as HyVM (Hybrid Virtual Machines), SARC (Scalable Computer Architecture), and AppleCore—PEPPHER does not focus on providing a common programming model or virtual-machine-type portability layer. In PEPPHER, the application programmer provides performance information by annotating components and describing characteristics of the actual environment and architecture, using the most convenient API for implementation variants that are tailored to different CPUs, GPUs, and other types of cores. Likewise, PEPPHER is not concerned with automatic parallelization per se. PEPPHER is not an auto-tuning project, but it enables use of auto-tuning techniques by exposing tunable parameters of both components and parameterized, adaptive library algorithms. PEPPHER takes a general-purpose approach in contrast to implicit parallelization and performance portability via domain-specific languages (as in work by H. Chafi et al., for example[1]).

Many other projects also employ implementation variants of functions, methods, or components tailored to different architectures as their basic premise for addressing performance and performance portability issues. Three recent such projects are PetaBricks,[2] Merge,[3] and elastic computing.[4] PetaBricks is an auto-tuning project that addresses performance portability mostly across homogeneous multicore architectures by focusing on auto-tuning methods for different optimization criteria.[2] Parallelism is implicit. Merge also provides variants, but relies on MapReduce[5] as a unified, high-level programming model.[3] Elastic computing provides large numbers of variants—so-called elastic functions—among which the best combination is composed mostly by static means.[4] Performance profiles and models guide selection. The STAPL project has extensively studied the use of algorithmic variants.[6] Other work that influenced PEPPHER is the Sequoia programming model for heterogeneous architectures based on a tree abstraction of the memory system.[7]

Unlike some of these works, PEPPHER takes a holistic approach, attacking performance portability at multiple layers from high-level component-based programming, compilation, library and runtime support, to hardware mechanisms for performance monitoring and feedback. Ultimately, PEPPHER aims to become language and parallelization agnostic and support different implementation languages and parallelization interfaces. Thus, the programmer can choose the most convenient language and API for implementing the components.

References

1. H. Chafi et al., ''A Domain-Specific Approach to Heterogeneous Parallelism,'' *Proc. 16th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* ACM Press, 2011, pp. 35-46.
2. J. Ansel et al., ''PetaBricks: A Language and Compiler for Algorithmic Choice,'' *Proc. 2009 ACM SIGPLAN Conf. Programming Language Design and Implementation,* ACM Press, 2009, pp. 38-49.
3. M.D. Linderman et al., ''Merge: A Programming Model for Heterogeneous Multi-core Systems,'' *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 2008, pp. 287-296.
4. J.R. Wernsing and G. Stitt, ''Elastic Computing: A Framework for Transparent, Portable, and Adaptive Multi-core Heterogeneous Computing,'' *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems,* ACM Press, 2010, pp. 115-124.
5. J. Dean and S. Ghemawat, ''MapReduce: Simplified Data Processing on Large Clusters,'' *Comm. ACM,* vol. 51, no. 1, 2008, pp. 107-113.
6. N. Thomas et al., ''A Framework for Adaptive Algorithm Selection in STAPL,'' *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* ACM Press, 2005, pp. 277-288.
7. K. Fatahalian et al., ''Sequoia: Programming the Memory Hierarchy,'' *Proc. ACM/IEEE Conf. Supercomputing,* ACM Press, 2006, article 83, doi:10.1145/1188455.1188543.

algorithmic auto-tuning, and compositional techniques;

- performance-aware, heterogeneous runtime scheduling;
- hardware feedback mechanisms through a tailored simulator; and
- source-to-source transformation and OpenCL compilation.

In this article, we explain PEPPHER's vision and basic premises and highlight some concrete results that substantiate our approach. In particular, we show how a numerical kernel algorithm built from available component variants can achieve a linear performance improvement with each extra GPU added to a hybrid CPU–GPU system, while also efficiently exploiting the CPU. The example demonstrates performance portability across hybrid systems with different numbers of attached GPUs. Best overall performance is achieved not by fixed, static offloading of tasks to the GPUs but by dynamically scheduling component tasks on either CPUs or GPUs according to resource availability and relative efficiency of the component variants.

The PEPPHER approach

PEPPHER's fundamental premise for enabling performance portability is to provide performance-critical parts of the applications in multiple variants suitable for different types of cores, usage contexts, and performance criteria. To this end, PEPPHER provides a flexible and powerful component model. Component composition techniques statically preselect and specialize component variants for a given heterogeneous architecture as far as possible, while a resource-aware runtime system handles final selection of the most appropriate variants. Runtime selection is based on optimization objectives, resource availability, data availability and placement, and available performance information for the component variants, while respecting data dependencies between components. Component variants can be generated in part by the PEPPHER framework's compilation and auto-tuning mechanisms, or supplied directly by the "expert programmer" (for instance, as part of a performance-portable library of algorithms and data structures). Component variants can themselves be parallel, in which case they make explicit requirements for specific, parallel resources. Preselection, composition, and runtime selection are guided by performance information that the application developer provides incrementally and that is evaluated in the context of an explicit platform model. Thus, PEPPHER's framework and methodology lets us gradually make an application more efficient for a given heterogeneous parallel system and more performance portable across heterogeneous systems by progressively supplying more suitable and efficient component variants and componentizing more of the application.

Concretely, PEPPHER introduces a flexible and extensible component model for encapsulating and annotating the application's performance-critical parts. Components become *performance aware* by association of performance models or regressions based on performance history for predicting a desired performance aspect (such as execution time or power consumption). Performance aspects are parameterized and evaluated relative to an abstract platform description.[2] The component model also lets us specify resource constraints and requirements, as well as other nonfunctional component properties that are nevertheless needed for efficient execution. It is essential for the PEPPHER framework to maintain *component implementation variants* for different platforms, input types, and data placements, and for different optimization objectives. PEPPHER can automatically generate variants via compilation to different platforms and via auto-tuning techniques. For the latter, the component model lets us expose tunable component parameters for which suitable auto-tuning tools can find good ("optimal") values. However, such auto-tuning tools are not specifically developed in PEPPHER. The expert programmer can also supply implementation variants manually (for example, targeted to different platforms). PEPPHER components might have been parallelized already using conventional parallel models and languages (such as OpenMP, Intel Threading Building Blocks (TBB), OpenCL, and Pthreads).

At runtime, the components form a directed acyclic graph (DAG) of component tasks. A component task variant can be scheduled when all data dependencies are resolved. Performance information, input information, optimization criteria, resource requirements and availability, and data placement in the system (for example, in the main CPU or in GPU memory) all determine which of the ready component-task variants are scheduled on which part of the system.

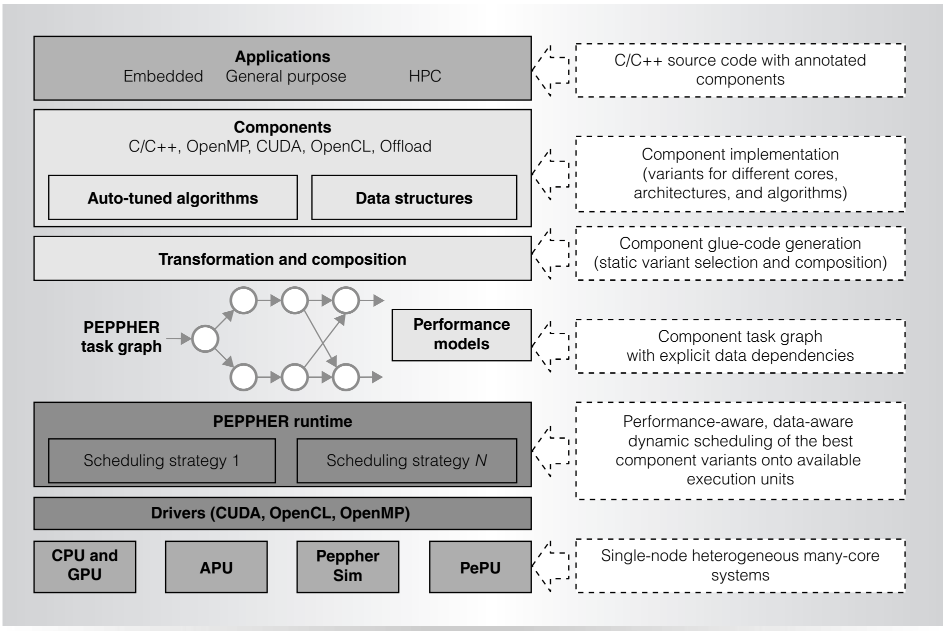
Applications
Embedded
General purpose
HPC
C/C++ source code with annotated components
Components
C/C++, OpenMP, CUDA, OpenCL, Offload
Auto-tuned algorithms
Data structures
Component implementation (variants for different cores, architectures, and algorithms)
Transformation and composition
Component glue-code generation (static variant selection and composition)
PEPPHER task graph
Performance models
Component task graph with explicit data dependencies
PEPPHER runtime
Scheduling strategy 1
Scheduling strategy N
Performance-aware, data-aware dynamic scheduling of the best component variants onto available execution units
Drivers (CUDA, OpenCL, OpenMP)
CPU and GPU
APU
Peppher Sim
PePU
Single-node heterogeneous many-core systems

Figure 1. The PEPPHER architecture and software stack. PEPPHER applications written in C++ are made performance aware and portable by turning critical parts into components with implementation variants for different hardware, input characteristics, and optimization criteria. Variants can themselves be parallelized in the most suitable framework. Variants can be generated automatically or supplied by "expert" programmers as part of libraries. Transformation and compilation techniques support the generation of variants. Scheduling and dynamic variant selection for the available resources is done by the PEPPHER runtime. Hardware mechanisms for performance monitoring and scheduling can be explored via the PeppherSim simulator. (APU: accelerated processing unit; HPC: high-performance computing; PePU: PEPPHER processing unit, may be new hardware or simulation based.)

The execution model is parallel at multiple levels: ready component tasks can execute in parallel on different parts of the system, and component tasks can themselves be parallel (such as OpenCL or CUDA variants for the GPU and multicore parallel variants for the CPU).

Architecture and software stack

Figure 1 gives an overview of PEPPHER's architecture and software stack, indicating how the approach's elements fit together. The figure illustrates the PEPPHER approach for developing and generating efficient performance-portable applications for heterogeneous many-core systems.

Performance guidelines and portability

Performance portability is an elusive notion. An application can be considered performance portable if it executes with the same efficiency (fraction of theoretical peak performance) across different heterogeneous multicore systems. This is a strong, absolute requirement. A different conception of performance portability is that application restructuring is not necessary for efficiency when porting code between architectures. In a library-based approach to performance portability, implementations of the library routines would target different types of architectures, input configurations, and performance criteria. At each call, the library would invoke the best implementation for the given situation using an efficient lookup and selection procedure. In PEPPHER, the application's components, rather than fixed-library functionalities only, are the units that enable performance portability. Performance portability is supported and enforced by guidelines and requirements that application components and library routines must fulfill.

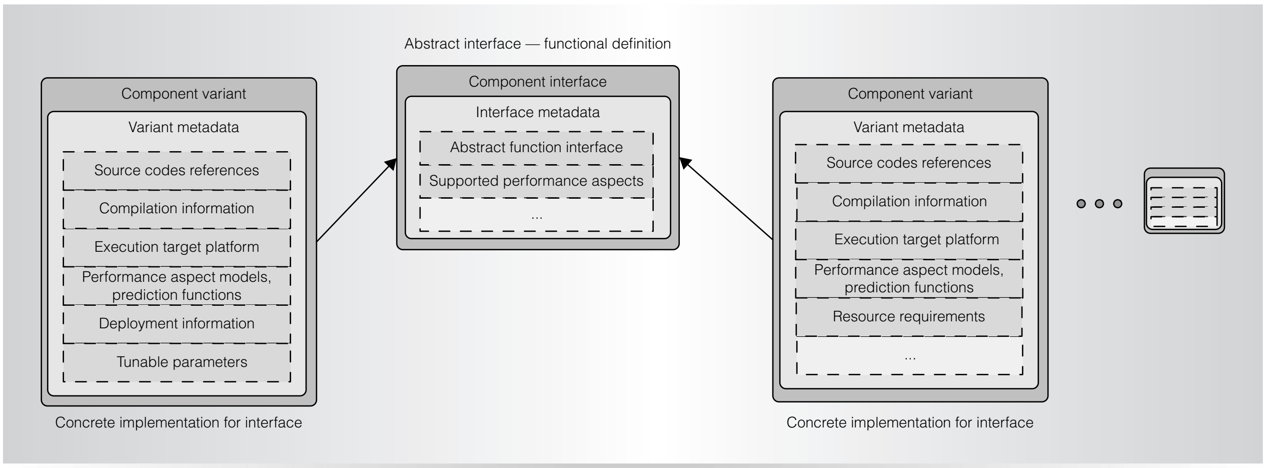
Abstract interface — functional definition
Component interface
Interface metadata
Abstract function interface
Supported performance aspects
...
Component variant
Variant metadata
Source codes references
Compilation information
Execution target platform
Performance aspect models, prediction functions
Deployment information
Tunable parameters
Concrete implementation for interface
Component variant
Variant metadata
Source codes references
Compilation information
Execution target platform
Performance aspect models, prediction functions
Resource requirements
...
Concrete implementation for interface

Figure 2. A PEPPHER component consists of an abstract interface describing the component's functionality and several implementation variants targeted for different architectures and with different performance characteristics. XML documents describe the interface and variant metadata.

An example guideline would prescribe that no library functionality could be improved just by expressing the same functionality using other related or specialized library components. Such a guideline would ensure that no performance gain on a different system would be possible by reimplementing a library or component functionality in terms of other related library functionality. For the PEPPHER framework, this is an obligation to the library implementation that efficient implementation variants for different architectures are in place, and that general library functionalities select the most efficient special-case implementations as appropriate. The guideline would further imply that the PEPPHER framework makes the best possible selection among a library component's available implementations. This is a nontrivial requirement in a heterogeneous and dynamic setting. Enforcing such requirements would relieve users from the temptation to write the selection code themselves. Other rules govern component annotations' use: the more information provided, the better the PEPPHER framework can select the most suited component variant. If no component information is provided, the PEPPHER framework produces default code. In this way, PEPPHER provides an incremental approach to making applications performance portable.

PEPPHER's technical aspects

The PEPPHER framework's technical ingredients include component model, runtime system, tunable algorithms and data structures, compilation techniques, hardware support, and application benchmarks.

Component model

PEPPHER provides a comprehensive, flexible component model for specification of performance-aware components and implementation variants, and exposing tunable component parameters. Currently, PEPPHER applications have to be written in C++. PEPPHER components are identified by language-external means (such as pragmas) rather than language extensions, and they use XML schemata for their detailed specification. Thus, the PEPPHER framework is largely nonintrusive and requires little direct application modification, for instance in legacy source code.

PEPPHER components are annotated software modules that implement specific functionalities declared in PEPPHER interfaces. A PEPPHER interface is defined by an interface descriptor, which is an XML document specifying the name, parameter types, and access types (read, write, or both) of the function to be implemented, and the performance aspects (such as mean execution time or energy consumption) that the component implementations' prediction functions must provide. Figure 2 illustrates the component model. Interfaces can be generic in static entities such as parameter types or code; genericity is resolved statically by expansion similar to C++ templates. From the interface descriptor, PEPPHER can generate header files in the various component implementations' source languages.

Several component variants can implement a PEPPHER interface's functionality (for instance, by different algorithms or for different execution platforms); in addition, further component implementation variants can be generated automatically from a common source module—for example, by special compiler transformations or instantiation of tunable parameters. The latter would be the task of auto-tuning tools. Variants could have different resource requirements and performance behavior and thus become alternative choices for composition or runtime selection whenever the interface function is called. To prepare and guide variant selection, component implementations must expose their relevant properties explicitly to the composition tool and runtime. Each PEPPHER component implementation variant thus provides its own component descriptor, an XML document containing information (metadata) about the following properties:

- the implemented PEPPHER interface;
- the required PEPPHER interfaces (that is, other component functionality called from this component, if any);
- the component implementation's source files;
- deployment information such as compilation commands and options and required software modules;
- reference to the platform consisting of the target architecture and the programming model and language used for the component implementation;
- the type and extent of resources required for execution on the given platform;
- reference to a performance prediction function, whose parameters are given by a context descriptor data structure;
- the context descriptor's structure, consisting of call parameter context and resource availability information;
- the component implementation's tunable parameters, such as buffer sizes, loop blocking factors, and cut-off values; and
- additional component selectability constraints, such as parameter ranges.

The component developer usually supplies the performance prediction functions. These can be purely analytical, using performance data tables determined by microbenchmarking for the target platform, or based on historical performance data. The latter reside in a performance data repository. The PEPPHER runtime can automatically generate performance models from historical data.

The actual platform properties are defined separately in another XML document and aim to describe the heterogeneous target environment's hardware and software characteristics. Such platform descriptions are used at multiple PEPPHER framework levels.[2] Lookup of specific platform properties is performed by the composition tool, the runtime, or the component developers themselves.

At runtime, component invocations result in tasks that are managed and executed nonpreemptively by the PEPPHER runtime system. PEPPHER components and tasks are stateless, and by default so are input and output parameters that determine the component tasks' dependencies. The runtime system tracks parameter data placement in CPU or GPU memory and takes this into account during variant scheduling. However, parameters can themselves have state, which is implemented by wrapping into Standard Template Library (STL)-like container data structures, and can track, for example, which memory modules hold which parts of the data. The container state then becomes part of the call context information because it is now relevant for performance prediction. The PEPPHER framework tracks the component implementation variants by storing descriptors in repositories that the composition tool and runtime system can explore.

Composition preselects a subset of appropriate implementation variants on the basis of statically available parameter and resource information. Parameter information can be either concrete or abstract values given by constraints. In the special case where sufficient metadata for performance prediction is available for all selectable component variants, composition can be done completely statically and co-optimized with the required resource allocation, as shown in work by C.W. Kessler and W. Löwe.[3] In general, however, several component variants will be available for the runtime system to select between, and this is PEPPHER's default mechanism. The composition tool can also generate part of the variant selection code. Finally, executable component variants are generated by compilation for the platforms as described in the component-variant descriptions. The runtime system makes the final selection among the remaining component variants.

To enhance programmability and provide for more variant selection and auto-tuning possibilities, we are currently extending the framework with higher-level coordination strategies for expressing computations in terms of structured execution of PEPPHER components, such as pipeline pattern, wave-front pattern, task farming, and various standard skeletons. We have taken first steps in SkePU,[4] an auto-tunable C++ template library of data-parallel generic skeleton components such as map, reduce, or scan, each with multiple implementation variants including CUDA, OpenCL, and OpenMP, for multi-GPU-based systems. Using microbenchmarking, we can automatically tune SkePU to select, depending on the call context, the expected fastest back end and values for tunable parameters, such as number of GPU threads and thread block size.

Runtime system

A flexible performance- and resource-aware heterogeneous runtime system executes a PEPPHER application that consists of compiled component variants together with the parts of the application that have not been componentized. The executable forms a DAG of component tasks with data dependencies, where each task is a set of one or more variants together with performance and other information to trigger selection. On the basis of system availability, resource requirements, estimated performance, execution history, and input availability, the runtime system schedules the most promising component variant on the best available resource. The further development of such a performance-, resource-, architecture-, and memory-aware runtime system, which can also schedule parallelized component variants over different parts of the heterogeneous system, is an important aspect of PEPPHER.[5]

For CPU- and GPU-based systems with separate memory spaces, the runtime system implements a directory-based virtual shared-memory system. The application registers component input and output data with the system. Scheduling of the ready component tasks occurs in a centralized fashion from a CPU. At runtime, the actual placement of data in either main CPU memory or GPU device memory is used together with the component performance models to decide where to launch the next component variant that uses these data. To this end, the runtime system uses a data-transfer cost model together with the cost estimation for the component execution. The runtime automatically handles the actual data transfers between different memory modules. Through so-called filters, the runtime can handle both nonconsecutive and block-distributed data.

PEPPHER users can modify the runtime system's scheduling policy. The current default strategy is HEFT (Heterogeneous Earliest Finish Time),[6] but we are also developing alternate, less-centralized scheduling policies that would use hints on processor resource requirements to execute parallel tasks on the multicore CPU.[7]

Together, the multilevel parallel framework that enables concurrency between component variants and intracomponent-variant parallelism; the static, resource-, and architecture-aware compositional techniques; and the dynamic, flexible runtime scheduling enable the PEPPHER framework to both efficiently utilize given, heterogeneous resources and to provide performance portability (by recompilation, recomposition, and retuning relative to a different platform description) to entirely different architectures.

Tunable algorithms and data structures for parallel architectures

The static compositional- and dynamic-runtime-supported approach to performance portability is complemented by expert-written auto-tuned architecture and context-adaptive algorithmic components for further enhancing performance portability. Algorithms written by specialists with a detailed understanding of the underlying architecture and its range of tunable architecture-dependent parameters can more flexibly and efficiently adapt to architectural changes and provide for more detailed performance control. Through libraries, application programmers can access such highly performance-portable algorithms as components. The highly nontrivial, auto-tunable GPU sorting algorithm developed by N. Leischner et al. is an example of the level of adaptable, portable performance that algorithm engineering experts can achieve.[8]

Table 1 summarizes some of the Nvidia Tesla and Fermi architectures' basic, performance-determining parameters. Based on these, the (sorting) algorithm developer infers tunable, algorithmic parameters related to these architectural features (see Table 2). These parameters are partially interrelated through the algorithm's analytical performance model, and they must partly be determined experimentally or through auto-tuning. By finding the right settings for the free parameters, we achieve superior performance of the algorithm on both architectures. Figure 3 compares the original version of the

Table 1. Recommended parameter settings for two current GPU architectures: Nvidia's Tesla and Fermi.

Hardware parameter	Tesla value	Fermi value
Thread overload factor, o	5	5
Hardware limit on threads per block, T	512	1,024
Number of streaming multiprocessors, S	15	30
Size of shared memory, E	16 Kbytes	48 Kbytes

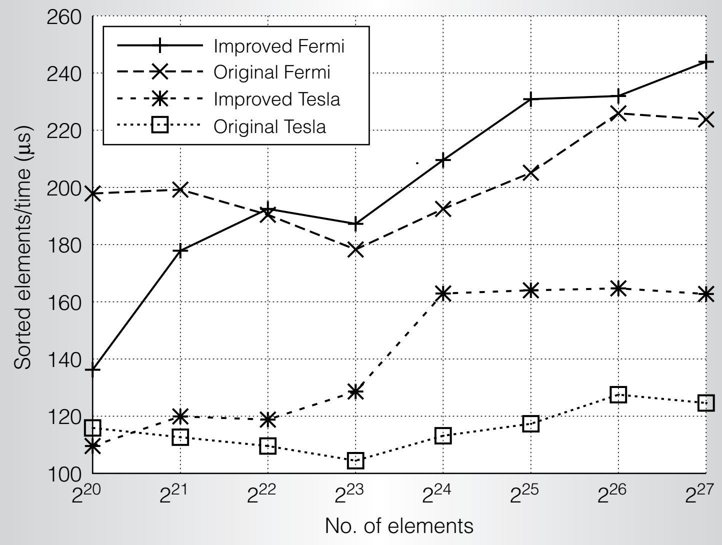

Figure 3. Comparison of original and improved GPU sorting algorithm on Tesla and Fermi GPUs for uniformly distributed random sequences. The figure shows the number of sorted elements per time unit (µs) as a function of the number of elements to be sorted. By finding the right settings for the free parameters, we achieve superior performance on both architectures.

Table 2. Tuning parameters for the two algorithm variants of the GPU sample sort on Nvidia's Tesla and Fermi GPUs.

		Original values		Improved values	
Parameter	**Constraint**	**Tesla**	**Fermi**	**Tesla**	**Fermi**
k	Distribution degree, $k \leq E/(ro)$	128	128	Variable	Variable
t	Threads per block, $t \leq T$	256	256	512	1,024
l	Elements per thread, $l := n/(t \cdot p)$	8	8	$n/(t \cdot p)$	$n/(t \cdot p)$
p	Thread blocks, $p := oS$	$n/(t \cdot l)$	$n/(t \cdot l)$	150	75
M	Fallback to small case sorter, determined experimentally	2^{17}	2^{17}	2^{16}	2^{16}
M''	Fallback to odd-even merge sort, $M'' \leq E/o$, $M'' \in 2^N$	2^{10}	2^{10}	2^{10}	2^{11}
r	Histogram replication factor, determined experimentally	8	8	8	8

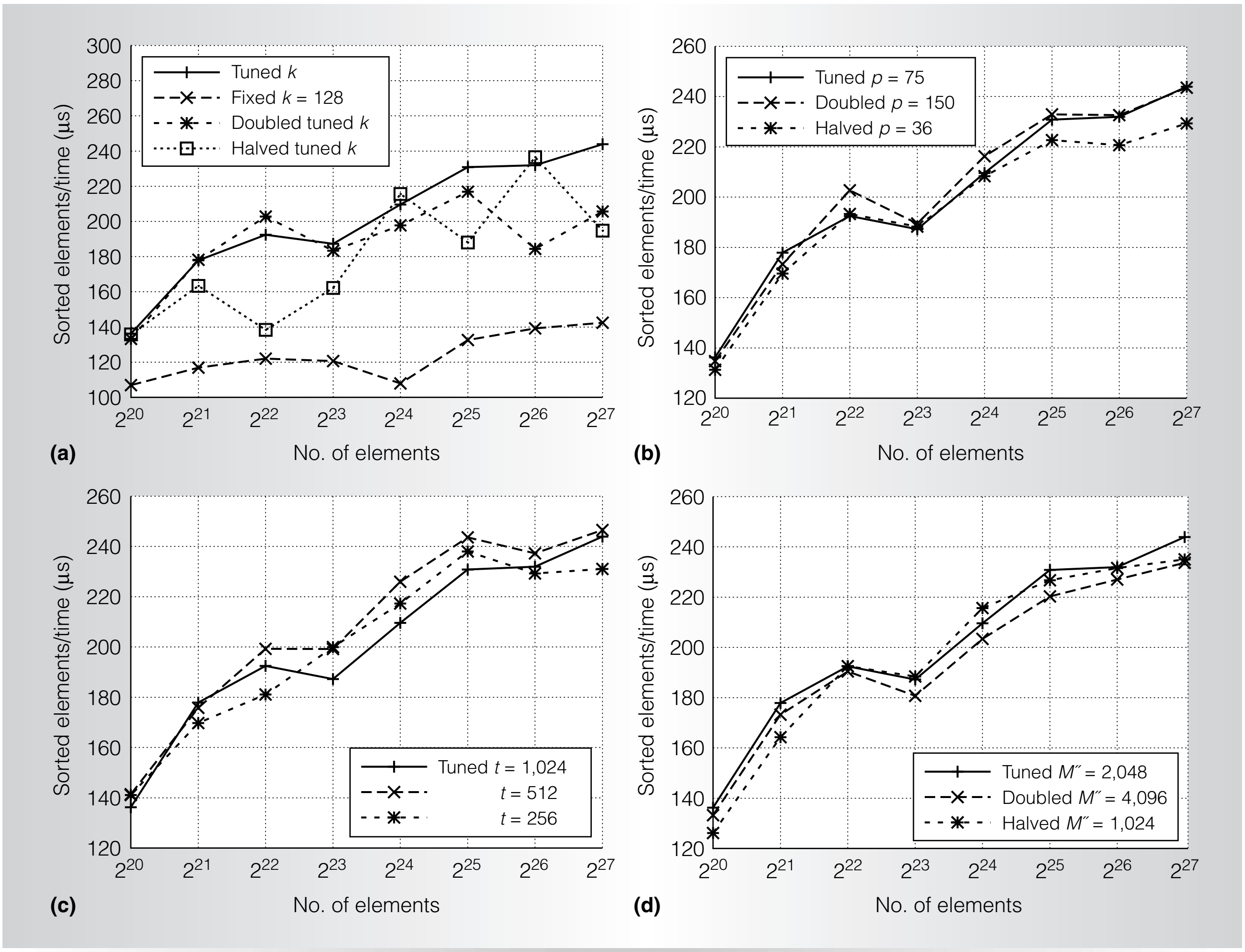

Figure 4. Comparison of alternative parameter values' effects versus the best, auto-tuned values for the Fermi GPU, uniformly distributed random sequences. We compare the changes to the parameters k (a), p (b), t (c), and M'' (d), as identified in Table 2.

algorithm to an improved version.[8] We take the best possible values of the tunable parameters from Table 2, and Figure 4 shows the impact of changes to the four basic parameters k, t, p, and M''. Some of these parameters are determined analytically; others have been found via auto-tuning experiments. In PEPPHER, concrete values for architectural parameters are looked up in the PEPPHER platform descriptors. At application deployment time, PEPPHER can generate Tesla and Fermi sample-sort variants, partly through auto-tuning for determining best values for the nondeterminate tuning parameters, and add them to the PEPPHER component repository. The algorithm has been implemented in both CUDA and OpenCL, but we only show the CUDA results here. We note that the CUDA and OpenCL interfaces by themselves ensure code portability but do not provide performance portability: this is achieved by the parameterized algorithm.

A larger selection of similarly tunable algorithms for many-core CPU architectures is found in the Multicore STL library (MCSTL)[9] and in the Thrust GPU library. Also, here algorithms are parameterized with typical architectural features, and thus exhibit a high degree of performance portability via auto-tuning for their target architectures class. Not mentioned here, PEPPHER needs and is developing algorithms and data structures for lock-free programming on CPUs and GPUs, both as application programmer components and for supporting the runtime system's implementation (see, for example, work by A. Gidenstam et al.[10]).

Table 3. PEPPHER benchmarks.

Application	x86 multicore CPU	Nvidia GPU	Cell	PeppherSim
	Enterprise and general purpose			
Suffix array construction	X	X	—	—
Games physics simulation	X	X	X	—
	High-performance computing			
Gromacs	X	X	—	—
	Embedded and multimedia			
BZIP2	X	X	—	—
Computational photography	X	X	X	—
	(Numerical) kernels			
Magma/Plasma	X	X	—	X
Rodinia	X	X	—	X
Fastest Fourier Transform in the West (FFTW)	X	X	—	X
Standard Template Library (STL) algorithms	X	X	—	—

Choosing between CPU and GPU variants for such library components is done by specialized glue-code that considers the resource availability—that is, by the library framework itself—or is delegated to the PEPPHER runtime. Thus, variant selection in PEPPHER is always external to the variant's implementation.

Compilation techniques

OpenCL is a possible portability layer for current heterogeneous systems, especially those that employ GPUs, but of a low abstraction level and not in itself solving the performance portability problem. Therefore, efficient compilation from C++ to OpenCL can support both the component model and the runtime system. Specifically, PEPPHER develops a C++ extension termed OffloadCL that allows for explicit compilation and offloading to GPUs but also to Cell synergistic processing elements (SPEs) with the compiler handling the necessary call graph duplication, functional duplication, and host data replication. The Offload compiler interfaces directly with the PEPPHER runtime system by encapsulating offloadable code as PEPPHER component tasks.[11]

Hardware support and feedback

To provide a larger spectrum of heterogeneous target architectures and to investigate possibilities for hardware support for performance monitoring and portability, PEPPHER has developed a highly configurable hardware simulator called PeppherSim. The simulator lets us run benchmarks and kernels on architecture configurations that are not physically available and experiment in a controlled way with the performance portability that the PEPPHER framework can provide. The simulator enables what-if experiments to determine performance bottlenecks and investigate new mechanisms for more detailed performance monitoring and feedback, especially energy consumption feedback. Also, we can investigate new synchronization primitives (for example, work by P.H. Ha et al.[12]) and other architecture support for algorithms and runtime with the simulator.

Application benchmarks

We selected a small set of larger application benchmarks to experiment with and validate the performance portability that the PEPPHER framework can provide. The benchmarks cover relevant application areas ranging from embedded to server, enterprise, and general purpose to high-performance computing domains. Additionally, important numerical kernels have been included as manageable test cases, which will also be useful as library components. Table 3

Table 4. The QR application's kernel components. The table shows the kernels' performance on CPUs and GPUs.

BLAS kernel	CPU performance (Gflops)	GPU performance (Gflops)	Speedup ratio
SGEQRT	9	30	3
STSQRT	12	37	3
SORMQR	8.5	227	27
SSSMQR	10	285	28

```
for (step = 0; step < min(MT, NT); step++){
        for (p = proot; p < P; p++) {
            SGEQRT(step, step, ...);
            for (j = step+1; j < NT; j++)
                SORMQR(j, step, ...);
            for (i = i_beg+1; i < MT; i++){
                STSQRT(i, step, ...);
                for (j = step+1; j < NT; j++)
                    SSSMQR(i, j, step, ...);
            }
        }
}
```

Figure 5. A QR factorization code fragment for matrices of size NT × MT using basic linear algebra subroutine (BLAS) routines. The kernels become PEPPHER components via suitable XML interface specifications.

summarizes the benchmarks and kernels and the target architectures for which component variants already exist.

A larger example

A final example illustrates how PEPPHER can provide efficient utilization of heterogeneous compute resources and performance portability across systems with varying numbers of GPU-type accelerators.

The application is a standard Tile QR factorization algorithm based on basic linear algebra subroutine (BLAS) components implemented on top of the PEPPHER runtime.[13] We constructed the application from BLAS kernel functions (see Figure 5), and we turn these kernels into PEPPHER components by giving suitable XML interface specifications. Expert implementation variants of the BLAS kernels for CPUs and GPUs are available through existing GPU and CPU libraries, namely Plasma[14] and Magma,[15] and are added as component variants to PEPPHER. Figure 6 shows the results of running the QR factorization application on a CPU–GPU system with one to four GPUs. The test platform comprises four quadcore AMD Opteron 8358 SE CPU cores (16 cores total) running at 2.4 GHz with 32 Gbytes of memory divided into four Non-Uniform Memory Access (NUMA) nodes. It is accelerated with four Nvidia Tesla C1060 GPUs of 240 cores each (960 GPU cores total) running at 1.3 GHz with 4 Gbytes of Graphics Double Data Rate 3 (GDDR3) memory (102 Gbytes/second) per GPU. One CPU is reserved for each GPU, leaving 12 CPUs available for computational work. Table 4 shows the performance of the QR application's kernels, along with the speedup achievable on the GPU relative to a single CPU core. The SORMQR and SSSMQR kernels are particularly efficient on the GPU, and the PEPPHER runtime is therefore likely to select the GPU variants.

The plot in Figure 6 shows that the PEPPHER runtime can exploit the GPUs efficiently, in that each additional GPU brings a linear performance increase of about 200 Gflops. Additionally, the runtime can exploit the remaining 12 CPUs for execution of the most suitable BLAS components. The latter is interesting; by automatically avoiding wasting GPU processing power with GPU-unfriendly BLAS kernels (which are scheduled on CPUs instead), we can achieve higher performance from the 12 CPUs (about 200 Gflops) than would have been possible running the whole QR application on those 12 CPUs (only 150 Gflops) alone. In the experiment, about 20 percent of the SGEQRT tasks were scheduled on the GPUs, whereas more than 90 percent of the SSSMQR tasks executed on the GPUs (see Table 4's speedup ratios).

The experiment demonstrates this particular application's efficiency and performance portability across systems with one to four GPUs. Note that it relied on existing expert-written components and required no extra implementation effort.

This article outlined the PEPPHER approach for achieving improved programmability and performance portability for current heterogeneous systems consisting of many-core CPUs with attached

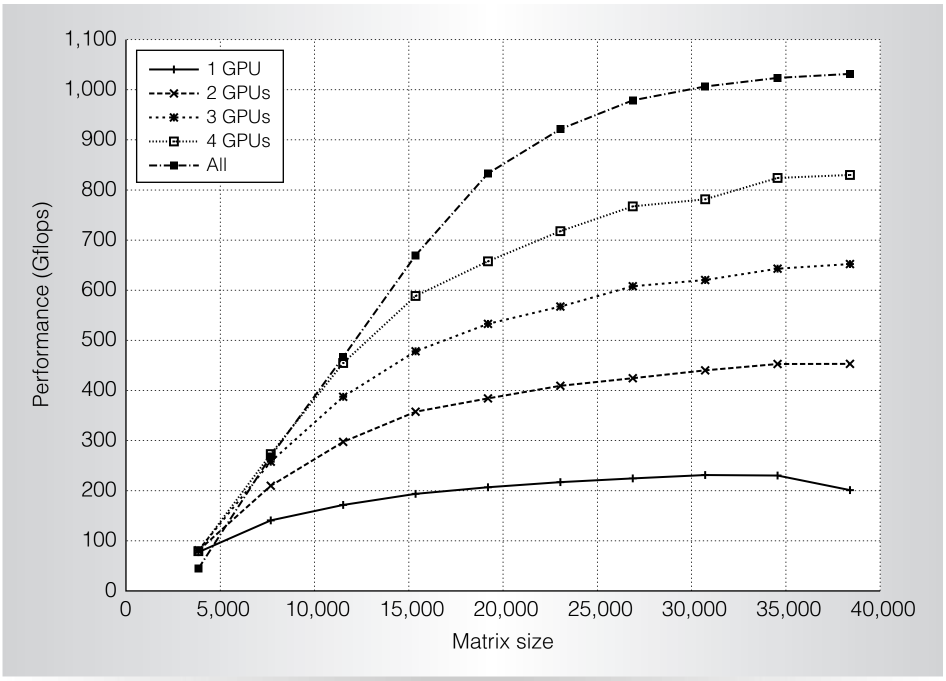

Figure 6. Achieved performance (Gflops) of the PEPPHER QR factorization implementation on a 16-core AMD system with up to four Nvidia Tesla GPUs (C1060). The cores are running at 2.4 GHz with 32 Gbytes of memory. The GPUs have 240 cores each (960 total) running at 1.3 GHz with 4 Gbytes of Graphics Double Data Rate 3 (GDDR3) memory (102 Gbyte/sec.) per GPU.

GPU- (or other) type accelerators. A main feature of the approach is the variant-based model of performance-aware and possibly parallel components. This is coupled with static composition techniques and a resource- and performance-aware runtime system. The project is complemented by a toolbox of adaptive algorithms and data-structures, OpenCL compilation techniques, and a highly configurable simulator for investigating feedback mechanisms for performance monitoring and data structure on a wider range of possible, heterogeneous architectures. Ongoing and future work will consolidate the framework, especially component model and composition techniques, and apply it to the selected larger benchmark codes, as well as expand on the algorithmic toolbox and runtime support. By this we aim to demonstrate the PEPPHER ingredients as key for achieving performance portability across architecturally different, heterogeneous multicore systems. MICRO

Acknowledgments

The EU supported this work as part of the FP7 Project PEPPHER under grant 248481. We gratefully acknowledge the contributions of all those involved in the PEPPHER project. We also thank the anonymous reviewers for constructive feedback.

References

1. M.W. Hall, Y. Gil, and R.F. Lucas, ''Self-Configuring Applications for Heterogeneous Systems: Program Composition and Optimization Using Cognitive Techniques,'' *Proc. IEEE,* vol. 96, no. 5, 2008, pp. 849-862.
2. M. Sandrieser, S. Benkner, and S. Pllana, ''Explicit Platform Descriptions for Heterogeneous Many-Core Architectures,'' *Proc. 16th Int'l Workshop High-Level Parallel Programming Models and Supportive Environments, Int'l Parallel and Distributed Processing Symp.,* IEEE Press, 2011, p. 42.
3. C.W. Kessler and W. Löwe, ''A Framework for Performance-Aware Composition of

Explicitly Parallel Components,'' *Parallel Computing: Architectures, Algorithms, and Applications,* IOS Press, 2007, pp. 227-234.

4. U. Dastgeer, J. Enmyren, and C. Kessler, ''Auto-Tuning SkePU: A Multi-backend Skeleton Programming Framework for Multi-GPU Systems,'' *Proc. 4th Int'l Workshop Multicore Software Eng.,* ACM Press, 2011, pp. 25-32.
5. C. Augonnet et al., ''StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures,'' *Concurrency and Computation: Practice and Experience,* vol. 23, no. 2, 2011, pp. 187-198.
6. H. Topcuoglu, S. Hariri, and M.-Y. Wu, ''Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing,'' *IEEE Trans. Parallel and Distributed Systems,* vol. 13, no. 3, 2002, pp. 260-274.
7. M. Wimmer and J.L. Träff, ''Work-Stealing for Mixed-Mode Parallelism by Deterministic Team-Building, *Proc. 23rd ACM Symp. Parallelism in Algorithms and Architectures,* ACM Press, 2011, pp. 105-115.
8. N. Leischner, V. Osipov, and P. Sanders, ''GPU Sample Sort,'' *Proc. 24th IEEE Int'l Parallel and Distributed Processing Symp.,* IEEE CS Press, 2010, doi:10.1109/IPDPS.2010.5470444.
9. J. Singler, P. Sanders, and F. Putze, ''MCSTL: The Multi-core Standard Template Library,'' *Proc. 13th Int'l Euro-Par Conf. Parallel Processing,* LNCS 4641, Springer, 2007, pp. 682-694.
10. A. Gidenstam, H. Sundell, and P. Tsigas, ''Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency,'' *Proc. 14th Int'l Conf. Principles of Distributed Systems,* LNCS 6490, Springer, 2010, pp. 302-317.
11. P. Cooper et al., ''Offload: Automating Code Migration to Heterogeneous Multicore Systems,'' *Proc. 5th Int'l Conf. High Performance Embedded Architectures and Compilers,* LNCS 5952, Springer, 2010, pp. 337-352.
12. P.H. Ha, P. Tsigas, and O.J. Anshus, ''NB-FEB: A Universal Scalable Easy-to-Use Synchronization Primitive for Many-Core Architectures,'' *Proc. 13th Int'l Conf. Principles of Distributed Systems,* LNCS 5923, Springer, 2009, pp. 189-203.
13. E. Agullo et al., ''QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators,'' *Proc. 25th IEEE Int'l Parallel and Distributed Processing Symp.,* IEEE Press, 2011, p. 32.
14. A. Buttari et al., ''A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures,'' *Parallel Computing,* vol. 35, no. 1, 2009, pp. 38-53.
15. S. Tomov et al., ''Dense Linear Algebra Solvers for Multicore with GPU Accelerators,'' *Proc. 15th Int'l Workshop High-Level Parallel Programming Models and Supportive Environments, Int'l Parallel Distributed Processing Symp.,* IEEE Press, 2010, pp. 1-8.

Siegfried Benkner is a full professor in the Faculty of Computer Science at the University of Vienna, where he heads the research group for Scientific Computing. His research interests include languages, compilers, and runtime systems for parallel and distributed systems, service-oriented software architectures, and grid and cloud computing. Benkner has a PhD in computer science from the Vienna University of Technology. He's a member of the ACM and IEEE.

Sabri Pllana is a senior research scientist in the research group for Scientific Computing at the University of Vienna. His research interests include performance-oriented software engineering for parallel and distributed systems. Pllana has a PhD in computer science from the Vienna University of Technology. He's a member of IEEE, the HiPEAC network of excellence, and the PlanetHPC network.

Jesper Larsson Träff is a professor in the research group for Scientific Computing at the University of Vienna. His research interests include interfaces, algorithms, and architectures for parallel computing. Larsson Träff has a PhD in computer science and a DSc from the University of Copenhagen.

Philippas Tsigas is a professor in the Department of Computing Science at Chalmers University of Technology. His research interests include concurrent data structures for multiprocessor systems, communication, and coordination in parallel systems, fault-tolerant computing, mobile computing, and

information visualization. Tsigas has a PhD in computer engineering and informatics from the University of Patras.

Uwe Dolinsky is CTO at Codeplay Software, where he is in charge of researching and developing novel compiler and software optimization technologies (such as Offload C++) for multicore processors. His research interests include programming models and compilation techniques for parallel processing. Dolinsky has a PhD in engineering and computer science from John Moores University Liverpool.

Cédric Augonnet is part of the Runtime team at INRIA Bordeaux and a PhD student in computer science at the University of Bordeaux. His research interests include task scheduling and hybrid accelerator-based machines. Augonnet has an MSc in computer science from the University of Bordeaux.

Beverly Bachmayer is a technical consulting engineer in the Software and Solutions Group at Intel GmbH. Her research interests include performance analysis and optimization of software on new computer architectures. Bachmayer has a BS in computer science from the University of Oregon and an MBA from Portland State University. She's a member of IEEE, the ACM, and the European Professional Women's Association.

Christoph Kessler is a professor in the Department of Computer and Information Science at Linköping University, where he leads the Programming Environments Laboratory's research group on compiler technology and parallel computing. His research interests include parallel programming, compilers, and software composition. Kessler has a PhD in computer science from Saarbrücken University. He's a member of the ACM and the IEEE Computer Society.

David Moloney is the cofounder and CTO of Movidius, a fabless semiconductor company focused on the design of software-programmable multimedia accelerator systems on chips. His research interests include computer architecture and digital signal processing. Moloney has a PhD in engineering from Trinity College Dublin. He's a member of IEEE.

Vitaly Osipov is a PhD student in computer science at the Karlsruhe Institute of Technology. His research interests include graph algorithms for modern architectures such as external memory algorithms and parallel algorithms. Osipov has an MSc in mathematics from Ural State University and an MSc in computer science from Saarland University.

Direct questions and comments about this article to Jesper Larsson Träff, University of Vienna, Faculty of Computer Science, Research Group of Scientific Computing, Nordbergstrasse 15/3C, 1090 Vienna, Austria; traff@par.univie.ac.at.