Adapt or Become Extinct!

The Case for a Unified Framework for Deployment-Time Optimization (Position Paper)

Georgios Goumas
National Technical University of Athens
Athens
Greece

Sally A. McKee
Magnus Själander
Chalmers University of Technology
Gothenburg
Sweden

Thomas R. Gross
ETH Zürich
Zürich
Switzerland

Sven Karlsson
Christian W. Probst
Technical University of Denmark
Kongens Lyngby
Denmark

Lixin Zhang
National Research Center of High Performance Computers
Beijing
China

ABSTRACT

The High-Performance Computing ecosystem consists of a large variety of execution platforms that demonstrate a wide diversity in hardware characteristics such as CPU architecture, memory organization, interconnection network, accelerators, etc. This environment also presents a number of hard boundaries (walls) for applications which limit software development (parallel programming wall), performance (memory wall, communication wall) and viability (power wall). The only way to survive in such a demanding environment is by adaptation. In this paper we discuss how dynamic information collected during the execution of an application can be utilized to adapt the execution context and may lead to performance gains beyond those provided by static information and compile-time adaptation. We consider specialization based on dynamic information like user input, architectural characteristics such as the memory hierarchy organization, and the execution profile of the application as obtained from the execution platform's performance monitoring units. One of the challenges of future execution platforms is to allow the seamless integration of these various kinds of information with information obtained from static analysis (either during ahead-of-time or just-in-time) compilation. We extend the notion of information-driven adaptation and outline the architecture of an infrastructure designed to enable information flow and adaptation throughout the life-cycle of an application.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EXADAPT '11 June 5, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0708-6/11/06 ...$10.00.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Design studies

General Terms

Design, Experimentation, Performance

Keywords

High-efficiency computing, runtime adaptation, specialization

1. MOTIVATION

The proliferation of multicore technology and the commoditization of High-Performance Computing (HPC) systems have changed the parallel computing landscape. Multiple cores within a node exacerbate the problem of utilizing resources efficiently. For instance, executing threads aggressively compete for common system resources such as memory and I/O bandwidth. Memory and interconnect technologies cannot keep pace with processor technology, and thus many types of applications have hit a *memory* [27] or *communication wall*. Numerous current HPC codes fail to scale beyond a few hundred cores, yet exascale systems will contain over a million cores.

One approach to deal with this situation might exclusively focus on the target system and attempt to develop codes that are "perfect" for a given target system. This approach requires a complete understanding of the low-level hardware and architectural features (often not completely disclosed by processor or system vendors) and then would produce codes that utilize a system's potential while at the same time avoiding pitfalls and hotspots. But such an approach would be viable only in a world dominated by one (or at most few) stable HPC architecture(s). However, this world does not exist and is unlikely to appear. Today's execution

platforms exhibit a wide diversity in hardware characteristics with different CPU architectures, memory organizations, interconnection networks, accelerators, and other features. Most likely, future exascale systems will share this characteristic with today's systems. Developing and optimizing codes for each hardware singularity would require an enormous programming effort by application domain experts, an approach that would hit the *parallel programming wall*. Even if the programming aspects of exascale systems are stable and identical, there is another reason to look beyond an approach that takes a static view of the target system. As exascale systems scale to millions of processors, the energy and cooling costs will pose a *power wall* (or power ceiling), and future software systems must include strategies to manage the power/energy budget.

These metaphorical walls represent physical limitations with respect to money, energy, and time (in terms of both human effort and machine occupancy). Traditional software development and execution practices are reaching a hard limit, and even existing petascale systems fail to deliver sustained petascale performance to a large number of applications. We argue that we need a radical change in the way we develop, execute and maintain software to effectively utilize existing systems and, more importantly, proceed to the exascale era.

2. THE (EXECUTION) ENVIRONMENT

The need for cost-effective supercomputers drives the commoditization of parallel systems, and growing research communities and industrial markets now have access to the "heavy metal" hardware that was once the exclusive property of the scientific HPC community. Parallel execution platforms proliferate within small research labs, SMEs, hospitals, schools, and universities.

The TOP500 list of supercomputers [22] includes diverse systems ranging from custom-made, finely tuned platforms to large clusters of cheap, off-the-shelf components. These systems span large design spaces with respect to internal CPU architecture, number of cores, memory subsystem, interconnection network technology and topology, and hardware acceleration support. A closer look at the list reveals several interesting issues:

- There are 33 different processor types (although the list is dominated by Xeon-based systems);
- There are more than 20 different interconnection networks, some of them custom-made;
- The number of cores per system ranges from a few thousands to a quarter of a million;
- 4 out of the 7 petaflop systems base their high performance on hardware acceleration;
- More than half of the systems rely on x86-based nodes that communicate via inexpensive Gbit Ethernet.

The last remark is what we call the *TOP500 paradox*: although the list includes the 500 most powerful HPC systems worldwide, it is dominated by a system architecture incapable of delivering peak performance to large classes of HPC applications sensitive to communication and thus bound by network bandwidth and especially latency.

The execution environment is not pleasant for many applications at the small scale of multicore nodes either. Again, here one can meet a large variety of processor architectures and memory hierarchy organizations. However, no configuration has been successful in providing good scalability to data-intensive applications that are bound by memory bandwidth and/or latency.

This is clearly an unfriendly environment for large classes of applications that require high performance but are not capable of obtaining it. Quite importantly though, this is not only a matter of hard performance bounds. Suboptimal codes that naively request all available system resources may end up with poor (even negative) scalability and, at the same time, waste energy. Regardless of the many differences across systems, one unifying limitation characterizes platforms ranging from supercomputers down to battery-powered smartphones: energy costs and distribution/dissipation considerations increasingly constrain the problems we can solve. These trends point to a growing need to move from HPC to *HEC: High-Efficiency Computing*.

3. ADAPT OR BECOME EXTINCT

In a diverse environment with many boundaries and strong requirements, applications need to execute efficiently. A nature-inspired way to meet this challenge is by adaptation: applications need to be able to adapt both to current and future execution platforms. Unfortunately, effective adaptation is not a straightforward process. We need to substantially change all stages in the life-cycle of an application, including its design, implementation, compilation, execution, maintenance, etc.

However, the good news is that research over the last decade has identified several opportunities for adaptation, i.e. several application features that can be parameterized and tuned, leading to significant performance gains or energy savings. We can find adaptation opportunities at the highest application level when selecting among available algorithms [3, 23], in application-specific parameters [15], in the parallelization stage (e.g., topology of processing elements) [11], in system software (e.g. communication libraries) [16], in loop or compiler optimizations (e.g., block size, unroll factor, optimization flags) [1], in the operating system (e.g., scheduling policies) [6], in the run-time system (e.g., concurrency level) [8, 21], and in hardware (e.g., cache coherence protocol, operation frequency) [25].

Autotuning libraries have been a success story for adaptation. Researchers and library developers have established that performing a *compile or installation time* search over the possible combinations of algorithms and data structures for a given kernel can be used, along with heuristics, to find (near) optimal codes. Although the cost of tuning can be expensive, once chosen, a kernel may be used thousands of times. The class of frameworks that support such a search are known as autotuners. Autotuners provide a portable and effective method for tuning over the plethora of optimizations available on today's multicore architectures. This is the case of ATLAS [26] for dense linear algebra, OSKI [24] for sparse linear algebra, FFTW [10] for fast Fourier transforms and Spiral [18] for digital image processing that are widely used by the scientific and engineering communities.

The effectiveness of adaptation is closely coupled to the quality of the information used. Compile-time adaptation has been successful since it has a central –though partial–

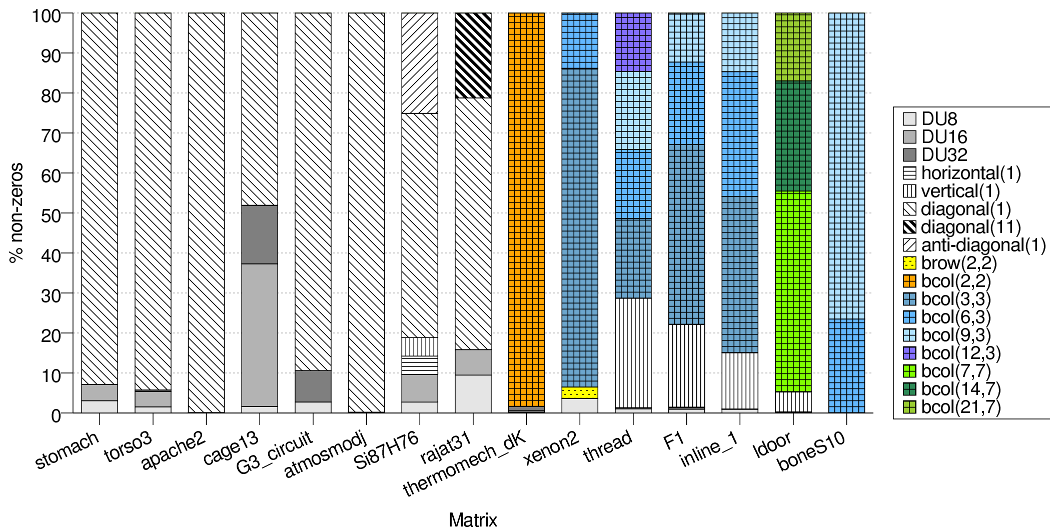

Figure 1: Distribution of substructures in 15 matrices. DUx represents a structure with non-zero elements close enough, so that their distance can be represented with x bits. Numbers in parentheses correspond to constant distances between elements in the relevant direction. brow/bco are two-dimensional dense blocks of the shape shown in the parenthesis.

view of both the application and the execution platform: it can collect information for the application through static analysis and can work with an approximate model of the execution platform as well. However, since a lot of information about the actual execution environment is not known at compile time (actually, sometimes only the ISA is known), we argue that there is a lot more to do as soon as this information becomes available. We illustrate this view in the next section, where we show examples of effective adaptation when utilizing information on the user input, the architectural features, and the execution profile of the application.

4. ADAPTING AT RUNTIME

In this section we discuss effective approaches of runtime adaptation. We show how we can utilize three valuable sources of information that become available only after static analysis is done: *user input*, *architectural parameters*, and information collected from *performance monitoring units (PMU)* of the execution platform.

4.1 Specialization on User Input

An important and ubiquitous computational kernel with streaming memory access pattern is the Sparse Matrix-Vector multiplication (SpMV). It is used in a large variety of applications in scientific computing and engineering. For example, it is the basic operation of iterative solvers such as Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES), which are extensively used to solve sparse linear systems resulting from the simulation of physical processes described by partial differential equations [20]. Furthermore, SpMV is a member of one of the "seven dwarfs" [4].

The distinguishing characteristic of sparse matrices is that they are populated by a large number of zeros, making it highly inefficient to perform operations using dense array structures. Special storage schemes are used instead, which target both the storage space costs of the matrix as well as the efficient execution of various operations by performing only the necessary computations. Thus, the common approach is to store only the non-zero values of the matrix and employ additional indexing information representing the position of these values. The most commonly used storage format for sparse matrices is the Compressed Sparse Row (CSR) format [5, 20]. CSR uses three arrays to store the non-zero arithmetic values, indexes to the columns of the original matrix for each value, and the number of non-zero elements per row.

Past work [12, 2] has identified the memory subsystem as the main performance bottleneck of the SpMV kernel. Obviously, this problem becomes more severe in a multithreaded environment, where multiple cores access the main memory. An approach for alleviating this problem is the reduction of the data volume accessed during the execution of the kernel. A target for data reduction is the indexing structure of the sparse matrix. Sparse matrices typically arise from regular computational domains in simulation problems and thus include several regularities in their structure. Such regularities can be one-dimensional dense sub-blocks (horizontal, vertical, diagonal), two-dimensional dense sub-blocks, areas of non-zeros with constant distances, or areas of non-zeros where elements are "quite close together". Figure 1 illustrates this situation in 15 matrices taken from Tim Davis' collection [9]. There exist a lot of different substructures in this matrix set, and quite interestingly one matrix (rajat31) has 20% of its non-zero elements in diagonals with distance 11.

CSR is a structure-agnostic storage format consuming one index per non-zero element. More efficient storage formats have been proposed that keep one index per substructure: BCSR [19] used in OSKI keeps one index per 2-dimensional block of constant size $r \times c$, and CSR-DU [13] exploits substructures along the same row. CSX [14] is the most aggressive storage scheme capable of mining all the substructures discussed in the previous paragraph. Figure 2 shows the speedups (over serial SpMV with CSR) attained by these four storage formats on an 8-core platform with 2 Intel Xeon

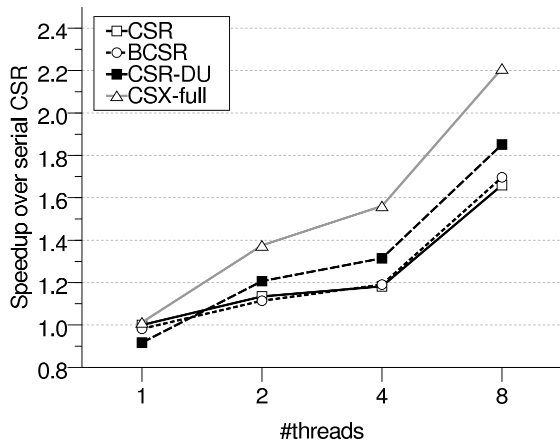

Figure 2: Average speedup of SpMV for various storage formats.

E5335 processors. We can see that with the elaborate analysis of user input performed by CSX, the SpMV kernel can be adapted to achieve on average 33% better performance compared to the standard approach (CSR).

These are very encouraging results towards adapting data-intensive applications for multicore platforms. Of course, the kernel remains memory-bound and scalability is not impressive after CSX either. Note, however, that all aforementioned optimization approaches target only the indexing structure of the sparse matrix, which is typically approximately 1/3 of the kernel's working set. In that respect, the performance improvement attained by CSX can be considered an important achievement. From the adaptation point of view, we see that user data carry a lot of valuable information that cannot be utilized until load time or later. So if we can pay a cost of analyzing and possibly pre-processing the user data as soon as this information becomes available, then we can achieve important benefits.

4.2 Utilizing Architectural Parameters and PMUs

As the number of processors per node is increased and the number of cores per processor grows, architects resort to non-uniform memory access (NUMA) designs that are able to provide a higher aggregate memory bandwidth than can be obtained in simpler memory systems. NUMA-based multicore processors integrate one (or more) memory controller(s) with each processor, and the physical memory space is divided between processors. Each processor has direct access to its local share of the memory space via its on-chip memory controller and has access to non-local (remote) memory via a cross-chip interconnect. Such multiprocessor configurations have non-uniform access times since remote requests are subject to various overheads. Remote memory accesses (in current small-scale multiprocessors with point-to-point connections between nodes) may cost 1.5 to 2 times more than local memory accesses. Large-scale multiprocessors that employ more sophisticated interconnect structures are likely to experience a higher cost. Consequently, it is highly desirable to enforce data locality, e.g., to place processes close to their data.

The different access times to memory are one aspect that complicates mapping of applications onto a system. In addition, modern architectures exhibit another key characteristic: the cores of a processor share one or more levels of the cache hierarchy. Resource contention for shared caches can be a source of severe performance degradation [7]. Thus, optimizing only for data locality can counteract the benefits of cache contention avoidance and vice versa. Since operating system schedulers might balance only the CPU load and do not account for data locality or cache contention, a process might be mapped to one node while the data for this process resides in the memory connected to another node. Consequently, the process will use the memory system in very inefficient ways. Thus, operating system scheduling is a good place to alleviate resource contention and increase data locality, targeting a tradeoff between them.

To enforce a cache-conscious scheduling policy for NUMAs, we need information about architectural parameters such as cache sharing and a characterization of the applications. The latter can be extracted from the PMU of modern microprocessors. In particular, we need to know if an application is memory or CPU-bound and to have an estimate of its cost when accessing its data remotely.

Majo and Gross [17] discuss two NUMA-aware scheduling approaches: the first one (*maximum-local*) targets only locality increases, by prioritizing the allocation of processes that suffer from accessing remote memory so that these processes are close to their data. The second one called *N-MASS (NUMA-Multicore-Aware Scheduling Scheme)* adapts the scheduling of maximum-local when this strategy fails to cope with cache contention. For example, N-MASS may decide to separate one or more CPU-bound processes from their data to leave space at the (previously) shared cache for memory-bound processes. N-MASS improves performance up to 32%, and 7% on average, over the default setup in current Linux implementations.

5. TOWARDS A HOLISTIC ADAPTATION APPROACH

There is a lot of room for more innovation towards adaptive parallel software. The key observation towards more effective solutions is that more *information* must be made available throughout an application's life-cycle to enable the collaboration between the application, the run-time system, and the hardware. The objective of this collaboration is to seamlessly manage the resources to deliver consistent, predictable, energy-efficient performance across the increasingly diverse computing platforms on which we rely. Valuable information can flow from the compiler by static program analysis, the runtime system by collecting profiling information, the hardware by exploiting its performance monitoring facilities, and, of course, by the domain experts. Interestingly, this information becomes available at different stages in the life-cycle of an application.

Current approaches do not exploit this information in combined ways, separating its utilization at the stages of design/implementation, compilation and runtime. For example, the successful approaches discussed in the previous section do not utilize information about the algorithm at the higher level, and thus they are not able to dynamically adapt the control of the application. Interesting approaches [28, 3] work towards this direction.

However, plain information is useless if we are unable to transform this information to *knowledge*, i.e. clear un-

derstanding of the application's execution behavior. We need to know hotspots of performance degradation or energy wastage (e.g. if the application is consuming all available memory bandwidth), and "cool spots", i.e. resources of the system that remain underutilized. This knowledge can drive the system to take *decisions* towards an execution context with higher efficiency. Finally, the system must have the power to take *actions*, i.e. adapt the running application to meet the required objectives.

Figure 3 provides an overview of a system designed to perform holistic adaptation for HEC. Domain experts can convey extra semantic information about a program via annotations and language extensions (e.g., alternative algorithms, or a domain-specific language to describe data-structure properties and usage characteristics). The compiler toolset applies optimizations and transformations based on the static source code, any additional semantic information, architecture-specific parameters of the target platform, and user inputs for a specific execution instantiation. The runtime system tracks additional information via a lightweight monitoring infrastructure and directs data to dynamic tuning modules (decision maker) to adapt the execution context. In a larger context, the system can tune multiple "knobs" synergistically: applications can be co-scheduled to minimize contention for resources, to control energy dissipation or thermal properties of the system, or to balance clock frequency with checkpointing behavior to deliver efficiency and high performance with acceptable reliability.

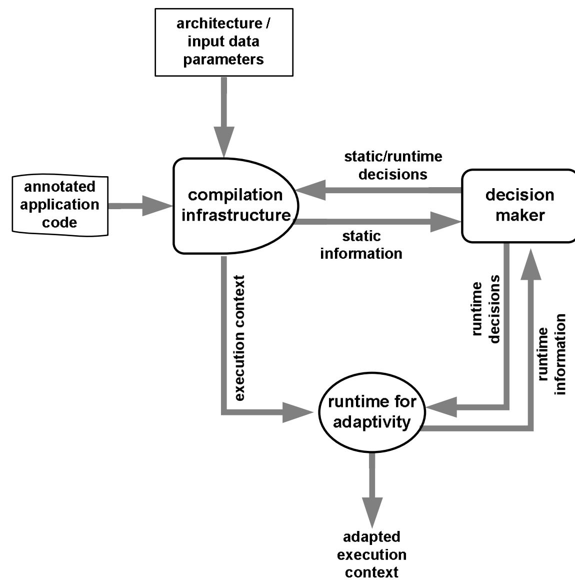

Figure 3: Overview of an information-driven adaptation infrastructure

Our view is that if we break the bounds between the sources of available information, we will be able to exploit numerous opportunities for adaptation, extending the target architectures, scales, parameters and metrics. Tuning activities will include simultaneously choosing the best instance among multiple code versions, selecting appropriate numbers and placements of threads, restructuring data (and the loops that operate on them) to increase performance and reduce energy dissipation.

6. CONCLUSIONS

Execution environments are becoming increasingly diverse and pose severe limitations to applications. Adaptation is a viable approach for software to meet the challenges of High-Efficiency Computing. The effectiveness of adaptation is closely coupled to the quality of the information used, thus it is important to devise a unified framework so that static program analysis, properties of the user input, and PMU observations can contribute to application optimization. We illustrate this approach by presenting two examples of runtime adaptation that utilize information on user input, architectural features, and the execution behavior of the application. The benefits of a unified framework are beyond what static transformations can obtain and show a clear path to information-driven adaptation. We expect that future software infrastructures that enable information flow and adaptation throughout the life-cycle of applications will provide a key element in the development of exascale software.

7. REFERENCES

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, 2006.

[2] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh cfd application. In *SC '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 69, 1999.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *SIGPLAN Not.*, 44:38–49, June 2009.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, December 18 2006.

[5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[6] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, 2010.

[7] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[8] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 19:1396–1410, October 2008.

[9] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97(23):7, 1997.

[10] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] G. Goumas, N. Drosinos, and N. Koziris. Communication-aware supernode shape. *IEEE Trans. Parallel Distrib. Syst.*, 20:498–511, April 2009.

[12] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*.

[13] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2008. ACM.

[14] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. CSX: an extended compression format for spmv on shared memory systems. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 247–256, 2011.

[15] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, 2007.

[16] S. Lee and R. Eigenmann. Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 195–204, 2008.

[17] Z. Majo and T. R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the 2011 ACM International Symposium on Memory Management*, 2011.

[18] M. Püschel, B. Singer, M. M. Veloso, and J. M. F. Moura. Fast automatic generation of dsp algorithms. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 97–106, 2001.

[19] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[20] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2003.

[21] K. Singh, M. Curtis-Maury, S. A. McKee, F. Blagojević, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Comparing scalability prediction strategies on an smp of cmps. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, 2010.

[22] Top500 supercomputing sites. http://www.top500.org, May, 2011.

[23] O. Trachsel and T. R. Gross. Variant-based competitive parallel execution of sequential programs. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 197–206, 2010.

[24] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005.

[25] P. E. West, Y. Peress, G. S. Tyson, and S. A. McKee. Core monitors: monitoring performance in multicore processors. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 31–40, 2009.

[26] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, 1998.

[27] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23, March 1995.

[28] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. *Parallel and Distributed Processing Symposium, International*, 0:447, 2007.