For Extreme Parallelism, Your OS Is Sooooo Last-Millennium *

Rob Knauerhase
Intel Labs

Romain Cledat
Intel Labs

Justin Teller
Intel Labs

Abstract

High-performance computing has been on an inexorable march from gigascale to tera- and petascale, with many researchers now actively contemplating exascale (10^{18}, or a *million trillion* operations per second) systems. This progression is being accelerated by the rapid increase in multi- and many-core processors, which allow even greater opportunities for parallelism. Such densities, though, give rise to a new cohort of challenges; for example, containing system software overhead, dealing with large numbers of schedulable entities, and maintaining energy efficiency.

We are studying software and processor-architectural features that will allow us to achieve these goals. We believe that exascale operation will require significant "out of the box" thinking, specifically in terms of the role of operating systems and system software. We describe some of our research into how these goals can be achieved.

* This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
Government Purpose Rights
Purchase Order Number: N/A
Agreement No.: HR001 10 3 0007
Contractor Name: Intel Corporation
Contractor Address: 2111 NE 25th Ave M/S JF2-60, Hillsboro, OR 97124
Expiration Date: None
The Government's rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by paragraphs B (1),(3) and (6) of Article VIII as incorporated within the above purchase order and Agreement. No restrictions apply after the expiration date shown above. Any reproduction of the material or portions thereof marked with this legend must also reproduce the markings. The following entities, their respective successors and assigns, shall possess the right to exercise said property rights, as if they were the Government, on behalf of the Government: Intel Corporation (www.intel.com); ET International (www.etinternational.com); Reservoir Labs (www.reservoir.com); University of California San Diego (www.ucsd.edu); University of Delaware (www.udel.edu); University of Illinois at Urbana-Champaign (www.uiuc.edu).

1 Introduction

1.1 Motivation

Historic parallelism has come from banding processors together on a task, either in a large distributed system (whether in a grid, cloud, or other HPC/supercomputer configurations), a smaller cluster of server nodes, or a number of sockets on a motherboard. In each case, there are interesting problems for division of labor, interconnect trade-offs (e.g. latency, bandwidth), and so forth. Concurrently, the microprocessor industry is trending toward many-core processors, uniting an increasing number of CPUs inside one processor; indeed, prototype designs currently include 80 cores [17] or more.

Processor fabrication technology, meanwhile, has enabled cores to run at ever-lower voltages. This has worked out very well for some high-core count uses (throughput computing, graphics, etc.). However, even including optimistic public estimates of energy-efficiency improvements over time, one can extrapolate that an "exascale computer" would require at least 500 MW of power, or roughly the output of a small nuclear power plant.

Despite the advances in many-core processor capabilities, we anticipate that exascale operation will require a staggering amount of computational resources. Indeed, one of our exascale hardware efforts (a complete description of which is outside the scope of this paper) is exploring the combination of a small number of "control engine" (CE) processors which supervise a very large number of lightweight "execution engine" (XE) cores.

Exascale systems will entail an unprecedented amount of complexity in managing the effective coordination and use of resources. Our research has been exploring how notions of system software — current and old ideas, along with some new ideas — should change in order

to effectively run an exascale machine.

1.2 Philosophy

In March 2010, DARPA generated a solicitation for innovative research joint-work proposals [7] on extreme-scale systems. The DARPA challenges, in addition to performance, included aggressive energy efficiency (both in terms of performance/watt and minimizing energy spent on intra-system communication), dynamic adaptability (to changes in workload, hardware, or external goals), and programmability. Our philosophies, software prototypes, and experiments have been strongly influenced by this program.

Our research hypothesis for exascale system software is that *a fine-grained, event-driven execution model with sophisticated observation and adaptation capabilities* will be key to exascale system software. To this end, we break applications down into dataflow-inspired [8] "codelets" [26]. These small routines are invoked based on satisfaction of data and control dependencies, which are specified either by the programmer or by a high-level compilation system. Codelets run to completion (or abort) on an XE, and because they are not multitasked, they contain no context other than a set of inputs and outputs.

1.3 Unsuitability of OS functions for exascale

Traditionally, the role of the operating system (OS) is to provide a variety of services so as to expose a common programming interface to programmers and applications, irrespective of the underlying hardware. An OS therefore *actively* tries to abstract away and hide the exact hardware. This approach has proven to be very successful and enabled the programmer to focus on the essence of his application without worrying about the nitty-gritty of managing hardware devices, memory, or threads and tasks. It also enabled programmers to write once and run unmodified on generations of processors. However, our research has led us to believe that many of the traditional OS functions are in the best case suboptimal, and in the worst case directly counter to the energy and performance goals we are seeking. To that end, we reject the notion of a traditional "operating system" and have focused on either omitting (avoiding the need for) or simplifying much of the functionality one imagines in an OS. Instead, our research is exploring the development of a sophisticated yet lighter-weight *runtime environment* to manage an exascale machine.

The remainder of the paper describes how our work has led us to be iconoclastic about the role of OS software at extreme scale; Sections 2 to 4 explore the services we remove or modify in an OS. Section 6 describes related work and we conclude in Section 7.

2 Hardware management

An important function of the OS is to manage access from user code to hardware resources. Traditionally, a kernel runs in a privileged execution mode while user code runs with fewer privileges and has specific channels (system calls) to interact with the hardware. We similarly separate *control code* and *execution code* but do so in a way that enables us to get rid of many expensive OS functionalities.

2.1 Separation of control and execution duties

Our current hardware architectural prototypes (simulated in software) include two types of processor cores:

- **"Control engines" (CEs)**: cores which execute our distributed runtime environment, including support for peripherals, but not direct user code;
- **"Execution engines" (XEs)**: simple, plentiful, very low-power cores optimized for HPC application needs, potentially heterogeneous among themselves when disparate types of fixed-function logic or accelerators are useful. The XEs run the application (user) code.

This division of duties allows specialization of each core type for power and performance as desired. For example, CEs do not need accelerator hardware for advanced math, but may however benefit from special low-latency interconnects or particular atomic instructions specialized for queue processing to help with the distribution of our control software. Likewise, depending on the assumed needs of applications, XEs can contain variably-sized floating point hardware, optimized implementations of arithmetic and transcendental functions, or other hardware (e.g. encryption logic) as needed.

2.2 Defining away hardware management overhead

Our design of separating control-code on the CEs and user-code on the XEs and our use of codelets allows us to do without many expensive OS functions.

Kernel versus user mode Implicit in traditional operating systems is the necessity to switch between user and kernel mode. This incurs overhead which in some cases can be very expensive (hundreds of cycles [23]). For complex exascale machines, such overhead (in terms of time, as well as power) is likely to be unsupportable.

Our system disregards the notion of ring boundaries entirely, choosing instead to separate privileges by space rather than by time. As mentioned, application code is only run on XEs, and our control software runs on CEs. This division obviates the need for traditional processor "modes" and their associated overhead. Several types of security concerns are likewise alleviated; combined with

hardware features (e.g. configurable range-checked access control and "locking" to render portions of memories immutable) and associated compiler support, our system obtains much of the security benefit of user/kernel division. Our simulations include hardware support to implement fast and power-efficient communication between the XEs and their controlling CE to replace the functionality traditionally provided by system calls.

Absence of device drivers Since only system code runs on the CEs, there is no need for device drivers in the usual sense of the term. If a CE does not directly connect to a peripheral, it can forward its request to a CE that does. The system runtime for CEs that do directly support devices includes device functions (e.g. to manage device state, or to accommodate special instructions or interfaces) as appropriate.

Unlike a traditional OS, our system does not support a wide array of devices. However, in the timeframe of our research, exascale machines are unlikely to be off-the-shelf commodity designs, and their owners are likely to have programmers or contracts to support hardware upgrades. User-code requests that entail peripheral devices are treated as data dependencies by the scheduler, and satisfied through runtime code on the CEs (in a manner akin to RPC).

3 Memory management

Memory management is another area where the OS has traditionally abstracted hardware with different memory configurations, through the use of virtual memory and heap management. We anticipate that exascale systems will contain more complex hierarchies to accommodate and make explicit "distances" between cores and memories (for both latency and energy consumption) as well as performance characteristics of different memory technologies. As part of our codesign process, we are exploring trade-offs in the use of different memory hierarchies and how their implicit costs are likely to be detrimental to energy and performance in extreme-scale machines of the future.

3.1 Virtual memory

Modern OSes rely on virtual memory to provide each individual process with its own address space thereby hiding the details of the backing physical storage medium (whether it is RAM or disk) from the programmer and allowing him to ignore the issue of code overlay. Virtual memory, however, also comes with two major costs: **a)** the hardware and energy costs of doing the translation between virtual and physical addresses and **b)** the loss of visibility into the varying characteristics of the physical memory (distance, energy costs, etc.).

Even if the energy costs of translation can be brought down [11], the latter problem is exacerbated in exascale systems where physical constraints make it necessary to have deep memory hierarchies to be able to simultaneously provide fast memory for data actively used by computations as well as very large ones for input and output data. The loss of visibility into the memory hierarchy can have drastic energy consequences. For example, without considering the energy required for the memory controller, it takes about 75 pJ per bit to move data from DRAM while it only takes about 0.5 pJ per bit to move data on-chip. This double-order of magnitude difference means that applications and their data must be very carefully positioned so that energy may be spent on computation rather than communication.

Why not make virtual memory smarter? Virtual memory in its current implementations is arguably ill-suited for exascale systems; it could be improved by making it more aware of energy considerations; one such effort is described by Lee et al. in [22]. In that work, the authors describe a scheduling method that is aware of the mapping between memory pages and caches so as to improve the efficiency of cache usage. However, the management granularity of virtual memory (a page) is not well-suited to multiple levels of memory hierarchy which will, by necessity, be of different sizes as the closer levels are smaller (faster and fewer transistors available). While [22] proposes making the virtual memory system more aware of locality and cache implications, we believe that for exascale systems, we need to go further and make explicit the placement of *objects* in the memory hierarchy.

Fundamentally, without giving the programmer more control over what data go together and the access characteristics of that data, an OS would be hard pressed to make good placement decisions with reasonable overhead costs.

3.2 Heap management

Similar to virtual memory, the OS's support for memory allocation is also agnostic to the characteristics of the underlying memory. Making the allocation system aware of the memory hierarchy, for example by adding a "hint" about the access pattern of the to-be allocated block of memory, could partially address the problem but would fail to capture the *time-varying nature* of access patterns: a particular block of memory could be frequently accessed when it is first allocated, then not used while the program accesses other data and then used intensely towards the end of the program. Ideally, this data should first be placed in "close-by" memory (to minimize power spent on communication rather than computation), then transparently moved farther away when not used and brought back in when used again. Allocation systems provided by the OS do not allow this as memory generally cannot be "moved" once allocated.

3.3 Solution: make data objects first class citizens

The solution we are pursuing is to do away with all the memory "support" mechanisms an OS provides and allow the programmer to directly manipulate *data-blocks* as first-class entities. A data-block is simply a contiguous chunk of memory with metadata annotations to allow the system to move it throughout the memory hierarchy. At a high level, data-blocks could seem akin to memory pages as pages are also contiguous chunks of memory that can be moved (mapped) to various parts of the physical memory. However, data-blocks have a key difference from memory pages: they are not agnostic to the computation. In other words, memory pages contain data that may or may not be related: i.e. a single page may contain data pertinent to different and unrelated sections of the program. Data-blocks, on the other hand, are explicitly created by the programmer and will more accurately reflect the control and data flows of the program. This enables a more optimal placement of data-blocks as the granularity of memory pages raises problems similar to "false sharing": two unrelated computations in the program may want the page to be close to them because they are accessing distinct parts of the page.

Similar to pages in virtual memory however, data-blocks introduce an added level of indirection (since their base address in memory can change) but this can be efficiently dealt with in hardware and with compiler support. Inspite of this additional indirection, the advantages provided by using data-blocks in exascale systems are very appealing:

- Each data object can be precisely placed in memory.
- As access patterns change, the objects can be moved either by the programmer, or automatically by the runtime. This also enables multiple cores to pass the same data object to each other (as in a pipeline) and have it optimally placed at each stage of the pipeline.
- Runtime observation systems, with the assistance of hardware, can track accesses to data objects thereby enabling optimizations based on access patterns.

Note that the increased control given to the programmer does not necessarily mean that he has to manually manage everything[1] and deal with all the complexities of fine-grained memory management. We envision a system where a runtime, aided by programmer hints or sophisticated observation via hardware performance-monitoring units, would optimally move data-blocks so as to reduce the energy required to access data.

[1] A "hero" programmer could of course laboriously micro-manage placements to eke out maximum energy savings.

4 Scheduling management

The threading metaphor is the only parallel abstraction a modern OS exposes to the programmer. However, the scheduling granularity of threads is ill-suited for new programming models (such as Cilk [19, 12], TBB [16], Habanero [4], X10 [5], Chapel [3] and Fortress [25]).

4.1 Thread parallelism for exascale

A traditional OS is responsible for mapping threads to hardware resources and context switching them as needed to ensure that they all get equal access to computing resources. While this approach eliminates the need to know the number of underlying parallel resources — in line with the OS's objective to abstract away the hardware — emerging parallel programming models rely on being able to *precisely* schedule much smaller grained tasks. These tasks require their own runtime to manage dependencies and the mapping from the tasks to the OS provided thread interface. The mismatch between the parallelism exposed by the programming model and that exposed by the OS leads to kludges in the runtime to ensure that the OS does not perform "optimizations" that harm performance. These hacks include **a)** using processor affinity to prevent thread migration, **b)** carefully matching the number of OS threads to the number of hardware supported threads to avoid context switching among others. Essentially, the runtime and the programmer work very hard to avoid many of the services the OS provides[2] as the goals of the OS (fairness in execution, responsiveness, etc.) are not shared by exascale programming.

Context switching Specifically, avoiding context switching is paramount in exascale systems as the cost of context switching in terms of energy (as well as time) is non trivial. Furthermore, to reduce the energy needed to access data, massively parallel systems will increase the amount of hardware-provided thread-local storage [9] such as registers and private scratchpad memories. In this situation, avoiding context switching becomes all the more critical.

Our program decomposition into small codelets [26] allows us to run a single codelet per XE without overhead from preserving ephemeral state (swapping registers, managing scratchpads, etc.). Since XEs are plentiful, the scheduling system can narrow or widen its scheduling of parallel code according to system state, workload, and overall power/performance trade-off policies. Additionally, as the runtime understands that a finishing task's context is no longer needed, it can put that core into a very-low-energy (destructive to memory) sleep state.

[2] This problem is not unique to exascale; DBMS implementations have also found benefit in bypassing OS functionality.

4.2 Required threading notions

The scheduling of codelets really only requires two notions: **a)** affinity and **b)** dependencies.

Notion of affinity A modern OS will provide hooks for threads with the affinity interface expressing links between threads and hardware resources. Conversely, our runtime provides an interface to define affinity between and among tasks. This approach allows the runtime to better understand the relationships among all tasks, such as which tasks are related and therefore share data. With this information, the runtime can schedule tasks onto the hardware resources in such a way to increase locality, or to optimize for other system operation goals. Furthermore, since the entire programming system and runtime are aware of the presence of possibly specialized XE hardware, the programmer is able to create specialized tasks to run on that specialized hardware. Both the compiler and programmer are also able to generate multiple equivalent versions of a task to run more efficiently on a heterogeneous computing substrate.

Notion of dependencies All dependency information is also expressed within the runtime's tasking interface. This concept is not novel among runtime interfaces (the TBB task graph [16] is one example). However, our runtime makes this dependency information available to the lowest levels of the system software. One particularly exciting area we are exploring is how this interfaces with the memory management portion of the runtime. As the memory subsystem can see how tasks are related and vice versa, the runtime can make better scheduling decisions. For instance, tasks can be scheduled to minimize data movement by either relocating data or moving a task to execute closer to its data. As all the dependencies are known to the runtime, related tasks can be moved at the same time, leading to an overall more efficient system. This data-directed scheduling research is an area from which we anticipate significant results from our work and that of the broader community.

5 Programming model

We discuss the programming model in the context of a single application taking over the entire exascale machine; as such, there is no notion of fairness between applications and correctly allocating resources between them. This could however be extended by considering different applications simply as "bunches" of codelets. While this would not ensure fairness, it would ensure that all applications are eventually run to completion. Both for performance and programmability issues, our runtime supports some novel programming model aspects:

- The utilization of *stateless* tasks (codelets) to ensure that once a codelet finishes, its state can be safely destroyed thereby conserving energy;
- Direct access to hardware memory features such as DMA and inter-core networks;
- Fine grained power and clock control to allow the runtime to turn off unused parts of the chip;
- Explicit description of the dependencies between tasks and between tasks and data to allow for better co-location of related tasks/data;
- Explicit placement of code and data for fine-grained control in particular in a heterogeneous environment to select the exact core type for a particular task.

The above programing model is heavily inspired by the codelet ideas recently described by Zuckerman et al. [26]. The programmer writes code to target a codelet framework such as the one we are developing at Intel, the SWARM runtime [21, 10] or the nascent Open Community Runtime [24]. Codelets may also be automatically derived from higher-level representations, such as Concurrent Collections [18].

The programming model was created to empower "hero" programmers by exposing more of the hardware features and details while still allowing the "average Joe" programmer to write correct code.

A key objective of our programming model is to reduce energy consumption by allowing the co-location of data and computation, thereby devoting a larger part of the energy available to actual computation (as opposed to merely shuffling data around). Furthermore, explicit data and computation placement will allow high-level tools, such as compilers and performance analysis toolkits, to provide enormous insight into the performance and energy usage of a program thereby enabling the programmer to hone in on "energy bottlenecks".

5.1 Leveraging the compiler

Our research is also investigating means to pass more information, or metadata, from the programmer and compiler through the compiler and into the runtime. In a traditional compilation system, much of the non-code data available to the compiler is not made available to the OS, being largely discarded when the binary is created[3]. One example is the trade-off between code space and loop overhead when a loop is unrolled: under different conditions, different versions of the code will have higher efficiency. Our research indicates that a runtime system that is aware of trade-offs like this could choose a task's implementation based on runtime information, increasing performance and saving energy. The compiler could therefore generate various versions of the code and the

[3] With the notable exception of debugging symbols which are not used/loaded during normal program execution

runtime could pick, based on environmental conditions and system goals, which version to run.

Other metadata will also prove crucial in a heterogeneous system as the codelets are able to expose their requirements or preferences in terms of hardware. The runtime scheduler then appropriately schedules these codelets trying to optimize for their preferences, available resources, environmental constraints (such as temperature in various parts of the chip) and data locality.

6 Related work

Exascale computing is an active domain of research. Most notably, nVidia Corp., another participant in DARPA's UHPC program, has identified [6] similar concerns for performance and energy efficiency. Previous, non-UHPC work, also identifies some of the issues related to OS bottlenecks as core counts increase. In [2], Boyd-Wickizer et al. show how performance of various parallel benchmarks can be greatly improved when applications are given the option to control the sharing of certain OS data-structures (file descriptor tables for example). Baumann et al. also argue in [1] that current OSes are not suitable for large-scale machines as they are more and more "networked" and should therefore be programmed as such. The Barrelfish OS described in [1] aims to distribute the OS across processors and replace some of the shared OS state with replicated state maintained through message passing. Therefore, [2, 1] both confirm our intuition that current OS principles are unsuited for exascale and need to be rethought.

Other efforts, such as Kitten [20] by Sandia National Labs and IBM's Blue Gene CNK [14] also aim to support extreme scale systems. These OS projects aim to provide a light-weight scalable kernel while still maintaining the traditional boundaries between kernel and user code.

In the realm of concurrent serial processes, there are also efforts to scale existing HPC technologies (e.g. MPI [13, 15] and OpenMP) into the extreme. Such work remains highly dependent on existing operating system concepts.

7 Conclusion and future work

We believe that extreme-scale systems are likely to break many of the known and beloved design wisdoms of both hardware and software. Existing challenges for performance, power efficiency, adaptability, and programmability will be greatly exacerbated when going to exascale operation. Our research therefore is pursuing some fairly radical concepts with respect to traditional definitions of OS functionality. We are taking advantage of hardware and software co-design to experiment with separating control and application duties across heterogeneous cores, as well as resurrecting prior data-flow inspired techniques in our codelet execution model.

While *much* work remains to be done, results to date are encouraging. Our system reduces much of the overhead that a traditional OS would incur, especially in terms of context-switching and memory management. These results will be discussed after the completion of the UHPC project later this year. Leveraging prior work, we plan to introduce dynamic observation features in our runtime and hope to further optimize both performance and power by automatically migrating data closer to code, and/or code closer to data. Similarly, as our hardware prototypes mature, we will be able to run more sophisticated runtime codes and workloads, and thus obtain more experimental data about the pros and cons of our system software assumptions.

Acknowledgments

We are indebted to our project's Principal Investigator, Shekhar Borkar, for championing this research throughout the exascale community. As well, our work relies heavily on the hardware research of colleagues on our team (Dave Dunning, Josh Fryman, and Nick Carter, among others). We hope our software lessons are as informative as their hardware lessons have been throughout the co-design process.

We would also like to thank the reviewers whose invaluable comments helped improve the final version of this paper.

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[2] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[3] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS04)*, pages 52–60, 2004.

[4] V. Cave, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old X10. In *9th International Conference on the Principles and*

Practice of Programming in Java (PPPJ), August 2011.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[6] B. Dally. The challenges of exascale computing (IPDPS keynote). `http://techtalks.tv/talks/54110/`, 2011.

[7] DARPA. UHPC BAA. `http://tinyurl.com/38zvgm4`, Jan. 2012.

[8] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, Supercomputing '88, pages 368–373, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[9] A. E. Eichenberger, K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine^{tm} architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[10] ETI. Eti website. `http://www.etinternational.com/`.

[11] D. Fan, Z. Tang, H. Huang, and G. Gao. An energy efficient tlb design methodology. In *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pages 351 – 356, aug. 2005.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[13] A. Geist. Mpi must evolve or die. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 5–5, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[15] W. Gropp. Mpi at exascale: Challenges for data structures and algorithms. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 3–3, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Intel. Intel threading building blocks. `http://www.threadingbuildingblocks.org`.

[17] Intel. Teraflops research chip. `http://techresearch.intel.com/ProjectDetails.aspx?Id=151`, 2007.

[18] Intel. Intel concurrent collections for c++ 0.7 for windows and linux. `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`, 2011.

[19] C. F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, jan 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[20] S. N. Labs. Kitten lightweight kernel. `https://software.sandia.gov/trac/kitten`, 2012.

[21] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[22] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 451–462, New York, NY, USA, 2012. ACM.

[23] J. Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, 1995.

[24] Sarkar, Wheeler, Teller, and Knauerhase. Ocr googlecode repository. `http://code.google.com/p/opencommunityruntime/`, 2011.

[25] G. Steele. Parallel programming and parallel abstractions in fortress. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 157–, Washington, DC, USA, 2005. IEEE Computer Society.

[26] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.