Ubiquity Symposium

The Multicore Transformation

Waiting for Godot? The *Right* Language Abstractions for Parallel Programming Should be Here Soon

by Todd Mytkowicz and Wolfram Schulte

Editor's Introduction

"Waiting for Godot" is a famous play by Samuel Beckett, in which two men occupy their time while waiting, indefinitely, for the arrival of their friend, Godot. As the play progresses we learn while both men claim Godot as an acquaintance, they really do not know him and, further, would not recognize him even if they were to see him.

As a discipline, we have been discussing parallel programming for years. After all these years, do we know the right language abstractions for parallel programming? Would we recognize the right abstractions if we were to see them? In this article, Todd Mytkowicz and Wolfram Schulte, both from Microsoft Research, ask: Have we been simply biding our time, waiting for our Godot?

Ubiquity Symposium

The Multicore Transformation

Waiting for Godot? The *Right* Language Abstractions for Parallel Programming Should be Here Soon

by Todd Mytkowicz and Wolfram Schulte

A good programming language should enable programmers to be productive. It should allow programmers to quickly express their ideas as programs while at the same time allow a compiler or runtime system to optimize their program's execution. In other words, a programming language and its implementation balance programmer productivity with a system's efficiency.

The mechanism by which programming languages provide this balance is "abstraction." An abstraction allows the programmer to say what to do in order to accomplish a task not *how* to do it. The "how to do it"—or the implementation of the abstraction—is left to the compiler or runtime system. A good abstraction allows programmers to say what to do, while at the same time not overly constraining a compiler or runtime system from providing good performance. Good abstractions increase programmer productivity by allowing developers to reason about the semantics of their programs and ignore the low-level details of how that abstraction is implemented.

For example, the popularity of managed languages (e.g. Java and C#) is in part due to the fact that those languages—through the abstraction of automatic memory management—remove a certain class memory related bugs, thus making programmers more productive. In these languages, programmers need not say how to manage memory, only what memory to manage. It is the job of the runtime to allocate and free memory when a program needs it.

In this paper we ask whether popular abstractions for parallel programming on consumer devices—typically implemented by multiple threads—running on multicore, allow programmers to balance productivity with performance. To make our point, consider the following post to an MSDN forum on the C# Thread Parallel Library (TPL)—one of C#'s language abstractions for parallel programming [1]:

> I am just trying TPL with some of the example code and I run into an issue with Parallel.Do function. I took a simple quicksort and used Parallel.Do and ran it on a dual core machine and saw significant performance degrade in the parallel version compared to serial version. I think I must have done some silly mistake.

Getting good performance from parallel programs is non-trivial; the programmer here has not made a ”silly” mistake. Indeed, we ran the example quicksort code that ships with the TPL over a large, 9M element array, on our four-core desktop and saw a 2X slowdown over a sequential version.

A bar in Figure 1 (x, y) compares the runtime in seconds (y-axis) with different implementations of quicksort (x-axis) on our four-core desktop machine. We used the example code from the TPL to build a sequential, parallel, and hand-optimized version of quicksort. In order to demonstrate these results generalized across different languages, we ran this experiment with two languages: C# and C++. To parallelize the C++ implementation instead of the TPL we used *Cilk* [2], which is a similar parallel language abstraction. Lastly, above each bar, we plotted the speedup over the sequential versions (C# and C++, respectively).

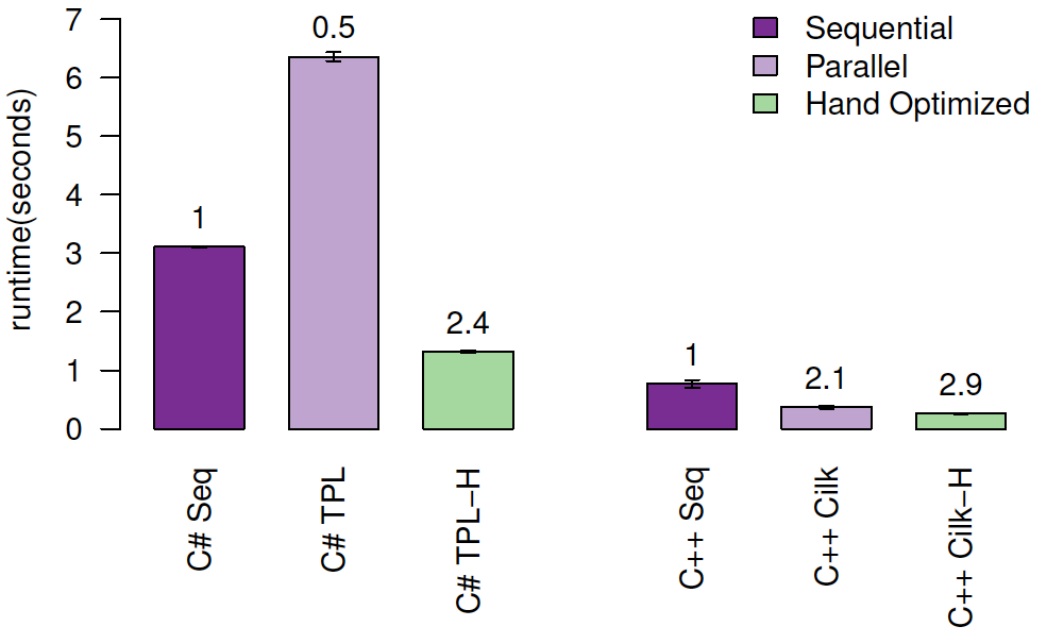

Figure 1. Peak parallel performance is non-trivial, even on simple programs.

Both the TPL and *Cilk* enable a programmer to succinctly parallelize quicksort: Each requires a single line change to fork a new task at the recursive decomposition of quicksort. From a programmer's standpoint this is great; with little effort they can parallelize their code.

However, the language abstraction that enables this single line change is "leak" [3]. With a good abstraction, a programmer only has to say what to do and not how to do it. In contrast, a leaky abstraction is one in which a programmer is required to understand how the abstraction is implemented in order to get good performance. In other words, with a leaky abstraction, certain low-level details that the abstraction is supposed to hide must nevertheless be dealt with by the programmer. The overhead of creating, scheduling, and balancing threads has a significant impact on quicksort's performance; and the language abstractions in both the TPL and *Cilk* do not sufficiently hide this low-level implementation detail. In particular, to get good performance, programmers must explicitly reason about the granularity of thread creation and change their code to only fork tasks when there is enough work per thread to amortize the cost of thread creation. In short, a leaky abstraction forces a programmer to intertwine both algorithm (i.e., quicksort) and implementation details (i.e., task granularity) in the same program.

To fix this leaky abstraction, we hand-optimized the TPL and *Cilk* implementations to control the granularity of thread creation (see green bars of Figure 1). In particular, our hand-optimized versions fall back to sequential quicksort when the number of elements in the input is less than 4,096. This simple change had a significant effect: The speedup for the TPL is 2.4X (e.g. vs 0.5X without this optimization) and 2.9X (e.g. vs 2.1X) for *Cilk*.

The right parallel programming abstraction should not require a programmer to understand these low-level implementation details in order to get good performance. However, in reality, to get good performance, software developers must learn these details. In the case of the TPL and *Cilk* its task creation overhead, its scheduling strategy, and its task placement all have a substantial impact on scalability and performance. We posit there are at least three possible solutions to this problem:

- First, we may simply have to accept the fact that parallel programming is hard, and the solution is to better educate the programmer. However, we don't believe this is the correct approach. In the case of the TPL and *Cilk*, requiring a programmer to manage scheduling, task placement, and granularity puts too much on the shoulders of a programmer, which slows their software development. Further, it requires programmers to reason about the performance characteristics of all of the different hardware on which their program will run. Clearly, this is not desirable.

- Second, it may be that our parallel programming abstractions, in theory, adequately balance programmer productivity with a system's efficiency. However, current compilers and runtime systems require more research and time to ensure good performance. This is common in computer science; for example, until compilers had

good register allocators, in order to get good performance, programmers had to manually perform register allocation by decorating a variable with the register keyword.

- Third, we can lift the burden of managing these details (e.g., costs of task creation, scheduling, and placement) to automation; we can build automated solutions that are able to implicitly manage these low-level implementation details. We would like to believe a runtime (e.g., the TPL) can automatically figure out how to balance the overheads of creating and scheduling threads for a variety of architectures, however, this is a difficult and program dependent problem. And in effect, this approach ensures our abstractions are the right ones, balancing programmer productivity with a system's efficiency.

In the text that follows we discuss the difficulty of this problem in greater detail by examining why current parallel abstractions leak.

The Problem: Leaky Abstractions

Asymptotic complexity is not enough. When we teach the fundamentals of computer performance we often discuss asymptotic complexity—or the characterization of a program's performance as a function of the size of its input. With asymptotic complexity, a programmer can claim that one algorithm is, in the limit, better than another. For example, a programmer would prefer a sorting algorithm with a $O(n\ log(n))$ complexity over one with $O(n^2)$.

However, asymptotic complexity may be misleading; when the size of the input is small, a program's performance is dominated by constants rather than asymptotic behavior. In other words, when the size of the input is small, the low-level details of an algorithm's implementation have a disproportionate impact on the algorithm's runtime. In effect, the low-level details of an algorithm leak through the abstraction, which complicates a programmer's task of choosing an efficient sorting routine. For instance, insertion sort, an $O(n^2)$ algorithm, compares very well with "faster" algorithms, like quicksort, when the input array is small. The reason being that insertion sort better utilizes the low-level hardware's cache and thus has higher performance for small n. In effect, these overheads cause asymptotic complexity to leak—programmers are required to know how each sort routine is implemented in order to optimally choose the best one when all they want to do is say what their program should do (i.e., sort).

We have asymptotic models for describing the behavior of parallel programs too; a work efficient algorithm is one that does, asymptotically, no more work than a sequential version. However, asymptotic models for parallel programs suffer the same fault as sequential ones: A parallel algorithm's asymptotic complexity is defined in terms of the number of processors in a system, which is often small. Thus, as in the sequential case, constants have a big impact on a

parallel algorithm's performance and can thus mislead a programmer. But where do these overheads for parallelism come from?

Overhead of language abstractions. In this paper we focus on explicit language abstractions, like threads, where a programmer has to explicitly control parallelism (e.g. TPL, OpenMP, *Cilk*, TBB, MPI, etc.).

The complexity of introducing multi-threading for a user, who wants to parallelize a sequential program, can be substantial. First, the program must be modified to add calls to create threads that can then perform independent work. Often additional code is needed to control and coordinate threads, like waiting for child threads to finish. Finally some synchronization code might be needed, in particular when two or more threads need to assess the same memory location, and if at least one of them is looking to update the location.

These calls to threading libraries can add substantially to the runtime overhead of the parallel execution, since they were not in the serial code. In addition, threading introduces constant scheduling overhead; for example, a scheduler might ensure that threads are assigned to all processors so that no one processor sits idle. Finally an often overlooked and completely unexpected cost in parallel programs is poor cache behavior. Most modern processors have special hardware that allows them to predict which memory location will be needed next by an executing processor. The processor can fetch these locations before a thread accesses them. However the management of shared caches in multicore processors is much more complicated and cache behavior can dramatically impact the program's performance, increasing computation time in the extreme case by 8X [4]. For example, if two threads on two processors access independent but close memory locations, each processor might predict the next access is to the other's location, resulting in continuously thrashing the cache. So in addition to all the above mentioned problems, multicore programmers must be aware of hardware features that are not even visible in the processor instruction set architecture. One could argue this problem is a constant and thus once a developer figures it out, their work is done. However, these constants change with each parallel platform—Intel's MIC, for example, has a very different set of constants than their normal Core architecture.

Experts of course know all about this, and decide on the right task and data decomposition. For example, Mark Harris, an expert NVIDIA developer, demonstrates order of magnitude performance gains by explicitly managing task and data decomposition for simply computing a running total of an array of numbers in CUDA [5]. But do normal developers get this right with existing languages? Do common language abstractions for parallel programming hide and correctly optimize for these low-level details?

How Do Current Language Abstractions Perform?

To study this, we picked a set of exemplary programs that professional developers have developed and see how they use abstractions and what performance they achieved.

Benchmarks: parallel dwarfs. Rather than use a suite of embarrassingly parallel benchmark programs, which are trivial for runtime systems to get good performance, we investigated the performance of a set of "anti-benchmarks." The parallel dwarfs are 13 kernels that focus on patterns of communication and computation found in modern applications [6]. The Parallel Dwarf project [7], provides an implementation of these programs in two languages: C# and C++. The C# implementation has a sequential implementation, one based on shared memory via the TPL, and one based on message passing via MPI (message passing abstractions) [8]. Likewise, the C++ implementation has a sequential and shared memory implementation using OpenMP. Professional developers, including some Microsoft personnel, have developed these 13 kernels.

Experimental methodology. We run these programs unmodified on an unloaded four-core Intel workstation (Q9650 processor) with 8GB of RAM. In order to produce statistically significant results, we run each program 10 times and record the time it takes to execute the entire program. In all graphs, we plot the mean and 95 percent confidence interval of the mean (difficult to see as they are all very tight).

Writing parallel programs is easy. A good parallel programming abstraction should allow a programmer to write sequential code and then easily add in parallel constructs [9]. We posit that with today's parallel programming abstractions, writing parallel programs is easy—language abstractions like the TPL or OpenMP make writing parallel code a relatively simple task. To make this point, consider the average extra lines of code to implement the parallel dwarfs is 18 for the TPL and 26 for OpenMP. In effect, a sequential program is made parallel by adding a few constructs to fork tasks and managing resource contention (e.g. locks). This is in contrast to MPI, an older parallel programming abstraction, which requires almost 200 extra lines of code.

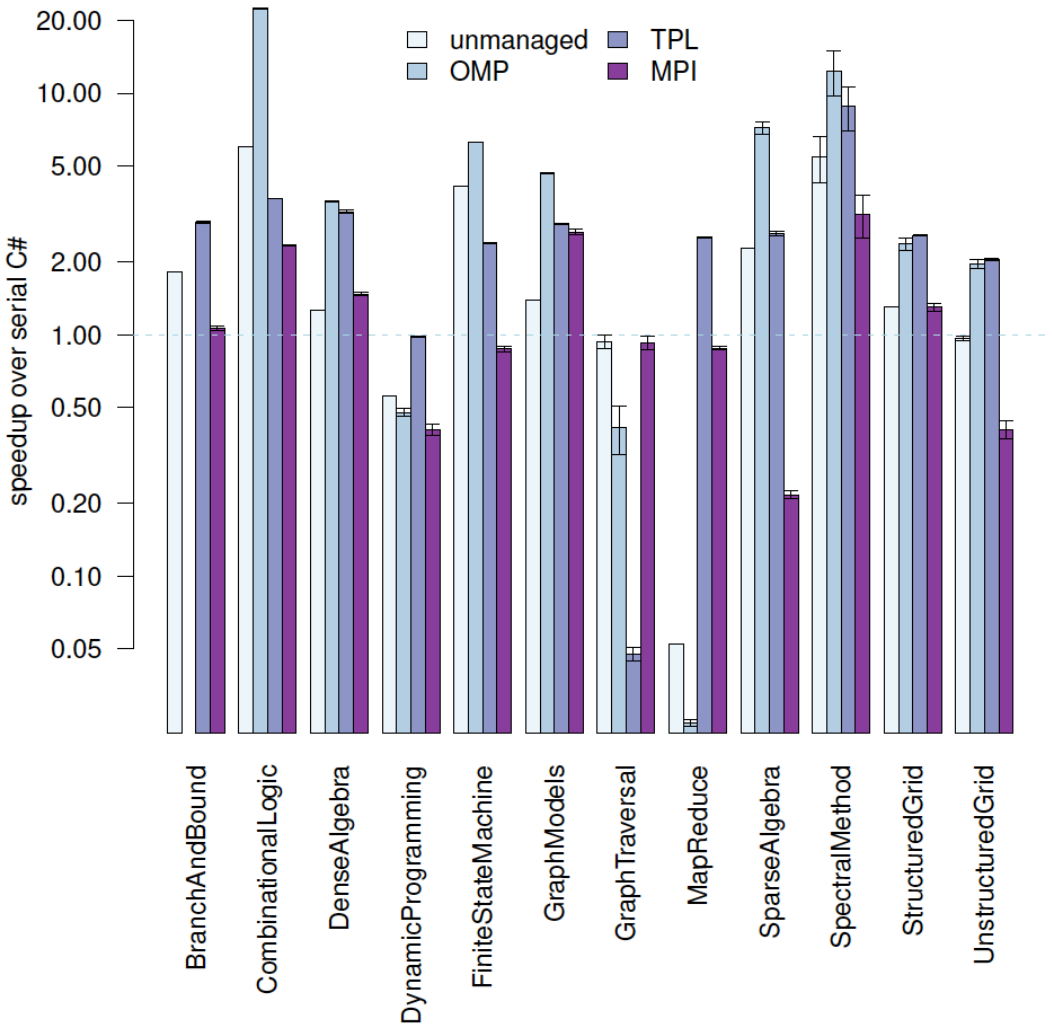

Figure 2. Speedup for Parallel Dwarfs.

Writing performant parallel programs is hard. While writing parallel programs may be easy and only require a few extra lines of code, writing parallel programs that get peak—or at least better than sequential—performance is difficult.

To make our point, consider Figure 2. A bar on this graph (x, y) gives the speedup (y), or time it takes to run a serial C# version of a Dwarf by the time it takes to run a parallel (or unmanaged) version of that same Dwarf, as a function of each Dwarf (x).[1] For each Dwarf, we show the speedup of (i) writing the program in unmanaged (or C++) code, (ii) shared memory abstractions (TPL and OpenMP, respectively), and (iii) message passing abstractions (MPI).

[1] The OMP implementation of BranchAndBound failed thus we do not plot a bar.

The geometric mean of shared memory abstractions (TPL and OpenMP) for the 12 Dwarfs is 2.0X and 2.2X, respectively on our four-core desktop machine. The mean, however, hides the details of Figure 2; some Dwarfs show a significant degradation in runtime performance. For example, the GraphTraversal Dwarf is 0.4X slower than the sequential implementation when using the TPL. In other words, if the sequential GraphTraversal Dwarf takes 100 seconds, the parallel TPL version takes 250 seconds. Likewise, the MapReduce Dwarf is 0.02X slower when using OpenMP.

The reason for these slowdowns is due to the cost of communication; each Dwarf does a small amount of independent computation in parallel and then needs to communicate those results to other threads via locks on shared memory. The result is contention and thus a significantly slower program. While easy to write, these programs are not performant.

Are We Still Waiting for the Right Abstraction

It is our conjecture that current general-purpose, thread-level abstractions focus on making writing multi-threaded code easy, but they do not enable programmers to write efficient code without knowing the abstractions' implementation. Too many low-level details (e.g. overhead of thread control, synchronization, and impact of data layout) make it difficult to get portable and consistent performance.

To address this problem, idealists believe one should lift the abstraction even higher. While one can easily point out that it is beneficial for programmer productivity, it is often detrimental for performance. Consider pLINQ, a query language integrated into .NET, which automatically parallelizes sequential LINQ expressions [10]. While beautiful in its simplicity, going from LINQ to pLINQ often does not improve performance; it is an Herculean effort for the pLINQ runtime to (re-)construct the right thread granularity and "best" communication pattern between threads that maximizes performance.

Instead we are moving toward a model where our parallel language abstractions should be restricted to a particular domain, but in doing so work great within that domain. Compilers and runtimes can then exploit domain knowledge to define the right thread granularity, optimal memory layout, minimize synchronization overhead, and thus control the constants in an algorithms complexity. A case in point is the recent work on parallelizing finite state machines. These seemingly sequential algorithms for finite automata can effectively take advantage of vector instructions and multicore and, as a consequence, provide multi-factor performance improvements over sequential implementations on a whole range of different hardware by building a compiler and runtime that specializes only for finite automata [4]. By specializing for a domain, the authors of a compiler/runtime can: (i) provide high-level abstractions that allow programmers to say what to do, and (ii) focus on building efficient runtimes that do not require a programmer know anything about how the abstraction is implemented to get good performance.

Similarly Berkeley addresses leaky abstractions with Selective Embedded Just-In-Time Specialization (SEJITS) [11]. For productivity, programmers embed a domain specific language (DSL) into an existing language (e.g. Python) so programmers can use the DSL as well as the features of the host language. At runtime the DSL is compiled for the target hardware, exploiting all its capabilities (like number of cores, vector instructions, even GPUs) often resulting in superb performance for kernels, while the rest of the program is still being executed sequentially.

We should point out both of these approaches also have issues. For example, while we can expect that libraries provide common parallel primitives, applications contain computations that are not available in a library, and are thus sequential. The problems with DSLs are similar, however, DSLs have the additional drawback that their costs are high, both for the programmer and for the supporter of the DSL.

Of course, thread-level abstractions for parallel programming are not the only abstraction our community has developed over the years. Dataflow based parallel programming abstractions model program execution as a directed graph of data flowing between operations. Languages and runtimes built on top of this abstraction may provide good abstractions for parallel programming. However, dataflow abstractions have yet to gain traction as stand-alone programming models for consumer devices. Instead, they have become popular library based abstractions embedded into other languages for ”big data” (i.e., Google’s MapReduce [12] or Microsoft’s DryadLINQ [13]). However, it is yet unclear if these libraries can scale down and provide good performance on modern desktops, laptops and tablets, and mobile phones.

Recognizing Godot

This paper investigated the performance of the TPL and *Cilk* on multicore desktops although we also expect our comments are applicable to popular programming abstractions for GPUs [5], too. We found it difficult to get peak performance with these abstractions albeit easier with *Cilk* than the TPL. It is yet unclear if these abstractions are the right ones and it is just that the implementation has not yet caught up with the abstraction. To be fair, it is entirely possible in time and with more research, the implementation will be able to automatically manage low-level details of the abstraction (e.g., thread scheduling, task granularity, etc.), thus relieving the burden from programmers and also at the same time ensuring good performance on a wide variety of machines. However, as of yet, this burden is on the programmers’ shoulders.

The two main characters in “Waiting for Godot” never find their friend, Godot. The play ends with the two men discussing, full of hope, that Godot will probably arrive tomorrow. The audience sees the folly in their waiting; however, the two men do not. Will we ever identify our Godot? In Beckett’s play, the two men are lost as they don’t even know how to recognize Godot, even if they were able to see him.

While it may be difficult to describe the right language abstractions for parallel programming, our situation is not as dire—a good parallel programming abstraction is sufficient. We posit a good abstraction should enable a programmer to succinctly express their ideas as code and not bother themselves with the details of how the abstraction is implemented. At the same time, the abstraction should not overly constrain the system from optimizing its execution.

In other words, a good abstraction is one, which programmers use often but rarely complain about using!

About the Authors

Todd Mytkowicz works in the RiSE group at Microsoft Research and works on parallelism and probabilistic programming. His thinks hard about abstractions that help programmers easily express complex problems in these domains yet are sufficiently constrained such that we can efficiently implement them. He holds a Ph.D. from the University of Colorado, Boulder.

Wolfram Schulte is a partner engineering manager at Microsoft's Cloud and Enterprise division, Redmond, USA. He is currently leading the Tools for Software Engineers team, which develops and operates developer services for all of Microsoft's software engineers. He has contributed to *software development tools*, ranging from build, via automatic test to deployment, *software engineering analytics,* ranging from collecting data to prediction, to *programming languages*, ranging from language design to runtimes.

References

[1] Leijen, D., Schulte, W., and Burckhardt, S. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIG- PLAN conference on Object oriented programming systems languages and applications, OOPSLA '09.* ACM Press, New York, 2009, 227–242.

[2] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y. Cilk: An efficient multi-threaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '95*. ACM Press, New York, 1995, 207–216.

[3] Spolsky, J. *Joel on Software*. aPress, 2004.

[4] Mytkowicz, T. M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '1*4. ACM Press, New York, 2014, pages 529–542.

[5] Harris, M. Parallel Prefix Sum (Scan) With CUDA. In *GPU Gems 3*. Ed. Hubert Nguyen., Addison-Wesley Professional, 2007.

[6] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The Landscape of Parallel Computing Research: A view from Berkeley. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec 2006.

[7] Safonov, V. Parallel Dwarfs. CodePlex. 2009.

[8] Gregor, D. and Lumsdaine, A. Design and implementation of a high-performance MPI for C and the common language infrastructure. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008*.

[9] Blelloch, G. Is Parallel Programming Hard? Intel Developer Zone. 2009.

[10] Duffy, J. and Essey, E. Parallel LINQ: Running queries on multi-core processors. *msdn Magazine* (Oct. 2007).

[11] Catanzaro, B., Kamil, S., Lee, Y., Asanovi, K., Demmel, J., Keutzer, K., Shalf, J., Yelick, K., and Fox, A. SEJITS: Getting productivity and performance with selective embedded JIT specialization. Technical Report UCB/EECS-2010-23. EECS Department, University of California, Berkeley, Mar. 2010.

[12] Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113.

[13] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U´., Gunda, K., and Currey, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *In OSDI'08 Proceedings of the Eighth Symposium on Operating System Design and Implementation.* USENIX Association, Berkeley, CA, 2008, 1–14.

DOI: 10.1145/2618395

