# Multicore Diversity: A Software Developer's Nightmare

David A. Penry
Department of Electrical and Computer Engineering
Brigham Young University
dpenry@ee.byu.edu

## Abstract

Commodity microprocessors with tens to hundreds of processor cores will require the widespread deployment of parallel programs. This deployment will be hindered by the architectural and environmental diversity introduced by multicore processors. To overcome diversity, the operating system must change its interactions with the program runtime and parallel runtime systems must be developed that can automatically adapt programs to the architecture and usage environment.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Run-time environments; D.4.1 [*Operating Systems*]: Process Management—Scheduling

***General Terms*** Performance, Algorithms, Design

***Keywords*** multicore, runtime parallel optimization, parallel adaptation, packaging, multicore diversity

## 1. The Problem of Diversity

Future microprocessor generations will be distinguished by increases in the number of cores per die; within ten years mainstream microprocessors will have tens to hundreds of processor cores. As with all previous processor generations, users will expect application performance to increase and new applications such as recognition, mining, and synthesis (RMS)[2] to be enabled when they purchase these new *multicore* chips. If users' expectations are to be met, applications must come to take advantage of multiple processor cores.

For the software developer, taking advantage of multiple processor cores is not easy. Effective parallelization requires the developer to both *discover* the potential parallelism and locality inherent in an application and then *package* that parallelism and locality into multiple threads of execution such that both parallelism and locality are exploited as much as possible.[7] This process is highly architecture-dependent. A parallel program optimized for one particular system will not be efficient on more than a few highly similar systems.

Unfortunately, multicore systems will not be highly similar to one another. Instead, they will be both more diverse and more widely deployed than past parallel systems. This diversity will take two forms:

- **Architectural diversity** Multicore systems will have differing numbers of cores, core complexity (e.g. in-order vs. out-of-order), cache hierarchies, and memory bandwidth. Intentional heterogeneity among the cores, both in their performance and their capabilities, is extremely likely.
- **Environmental diversity** Mainstream multicore systems will operate in a desktop computing environment. Desktop computing is a highly dynamic environment; there are often multiple applications running simultaneously and the mix of applications changes frequently. It will not be acceptable for an application to wait until a fixed set of required resources are available. Instead, applications must adapt continuously to whatever resources the OS makes available at the moment.

Parallel application performance is highly sensitive to how well an application's parallelism and locality packaging is tuned for an architecture and environment.[8] This can be seen easily by considering the simplest architectural parameter: the number of cores. If an application uses fewer threads than there are cores available to it, some performance potential could be lost, as more threads may perform better. On the other hand, if an application uses more threads than there are cores, context switching ensues and application performance can fall dramatically.

Architectural and environmental diversity will make it impossible to determine *a priori* how an application should be packaged. Thus distribution of a highly-optimized multicore program will not feasible. *Overcoming diversity will be a nightmare for developers who wish to deploy efficient parallel programs on multicore platforms.*

## 2. Overcoming diversity

As mentioned before, parallelization consists of two activities: discovery and packaging. Discovery is achieved by determining what parallelism and locality is present in an application. Packaging is performed by mapping and scheduling the application's work onto threads and laying out its data. The application may be packaged into more threads or less threads than there are resources. Packaging can be performed both statically and dynamically and can operate on both coarse-grained and fine-grained tasks.

Examination of the characteristics of the discovery and packaging activities reveals that:

1. Discovery finds properties inherent in an application's logic and programming model that are independent of the architecture and usage environment.
2. Packaging uses the properties found through the discovery in an architecture- and environment-dependent fashion.

Thus discovery and packaging interact with diversity in very different ways, suggesting a solution to the problem of diversity: perform discovery at compile time, but delay packaging until both the architecture and the usage environment are known: i.e., until runtime.

Runtime packaging could be performed by an application itself; e.g., the inspector-executor model of execution[6] is a special case of application-driven runtime packaging. However, doing so in every application would place a significant burden upon developers. A better solution is to provide a parallel runtime system that can perform packaging. Figure 1 is a block diagram of such a runtime system, which operates in the following manner:

1. The developer writes applications in any of a variety of new or legacy programming languages and models, making no assumptions about the target system.
2. The compilers for these languages perform parallelism and locality discovery using user annotations, code analysis, or inference from the language semantics and produce binaries augmented with information about application parallelism and locality (abbreviated as *PLI*). PLI is found during compilation, when time for analysis is readily available; however, the compiler may find that some PLI is data-dependent and indicate that the PLI must be analyzed further at runtime using application data.
3. The runtime system uses the PLI, runtime application data, and online performance feedback to package the application for the architecture and available resources. This packaging uses very fine-grained units of work not restricted to high-level language constructs such as loop iterations, thereby increasing scheduling flexibility and potential parallelism above that of coarser-grained parallel adaptation such as [3, 8]. The resulting schedule may contain fewer threads than there are resources when additional threads would have low efficiency; to prevent context switching, the schedule never contains more threads.
4. The application threads execute using the mapping and schedule produced by the packaging step. To amortize the cost of producing the schedule, the schedule is reused when possible. For example, in an iterative solver, the parallel schedule for an individual iteration would be reused in each iteration. The use and reuse of a computed schedule distinguishes this style of runtime packaging from work-stealing or dynamic task scheduling.
5. When the available resources change, the runtime system repackages the application. Feedback from performance monitoring may also trigger re-packaging.

## 3. Changes to Operating Systems

Runtime packaging will affect the architecture of operating systems. The most important effect will be to relieve the operating system of the responsibility to schedule the individual threads of applications. Instead, the operating system's scheduler becomes responsible solely for determining what "mix" of applications should be currently running and what resources they should be allowed to use. In making this decision, the scheduler will take into account application priorities, overall system throughput, fairness, and how efficiently applications can use the resources. The resulting scheduler architecture is similar to Exokernel[4] or Corey[1].

The second important effect will be that applications and operating systems will need to negotiate for resources in more detail than they now do. The application runtime system will request resources and inform the operating system about how efficiently it can use those resources. The operating system will decide what resources to allocate and inform the application as the allocation changes, allowing the application to repackage for the new resource allocation. There will need to be mechanisms to allow the application a "grace period" in which to perform its repackaging, as well as mechanisms for validating an application's efficiency claims.

## 4. A Plan of Action

Diversity will be a major hindrance to practical parallel program deployment for multicore systems. A parallel runtime that performs packaging coupled with the changes to the operating system mentioned in the previous section has the potential to overcome this diversity. Development of such a runtime will require significant improvements in multiprocessor parallelism and locality packaging algorithms that overcome the following challenges:

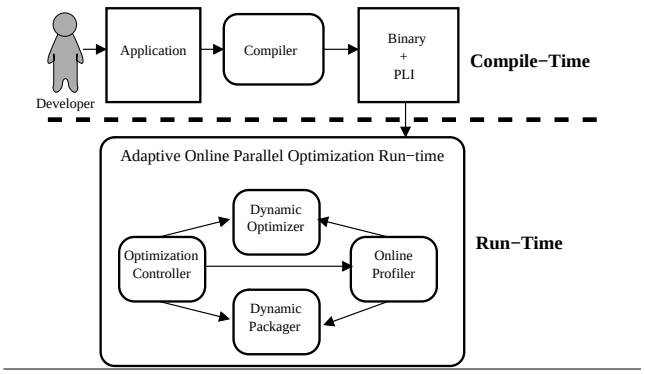


**Figure 1.** A parallel runtime packaging system

- **Packaging algorithm performance.** The performance overhead of fine-grained runtime packagers must be reduced. Achieving this reduction requires two advances: packagers that can exploit any regularity present in the application's task structure (while still working when there is no such regularity) and packagers that can themselves be parallelized.
- **Trading off locality with parallelism.** Runtime packagers must simultaneously improve the application's temporal, spatial, and thread locality while maintaining good load balance and parallel efficiency. Locality will be particularly important for commodity systems with limited memory bandwidth.

We are currently developing such a parallel runtime. The runtime is built upon the LLVM compiler framework[5], which provides both static and JIT compilation capabilities. We are evaluating new packaging algorithms applied to the kernels of RMS applications, with a particular emphasis on improving locality.

## References


[1] 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 Symposium on Operating Systems Design and Implementation*, 2008.

[2] Y.-K. Chen, J. Chhugani, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, and M. Smelyanskiy. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5):790–807, May 2008.

[3] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. A helper thread based EDP reduction scheme for adapting application execution in CMPs. In *Proceedings of the International Parallel and Distributed Processing Symposium 2008*, pages 1–5, 2008.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251–266, 1995.

[5] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[6] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[7] B. Smith. Reinventing computing. In *Manycore Computing Workshop*, June 2007.

[8] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the 13th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 277–286, 2008.