DOI:10.1145/2742908

To view the accompanying paper, visit doi.acm.org/10.1145/2742910

Technical Perspective
Programming Multicore Computers

By James Larus

WHAT IS THE best way to program a parallel computer? Common answers are to have a compiler transform a sequential program into a parallel one or to write a parallel program using a parallel language or library.

In the early days of parallel computers, parallelizing compilers offered the tantalizing promise of running unmodified "dusty deck" sequential FORTRAN programs on the emerging parallel computers. Although research on these compilers led to many program analysis and representation innovations used in modern compilers, the resulting tools were not successful at parallelizing most applications, and developers turned instead to libraries such as pthreads and MPI.

In this approach, programs use parallel constructs; either explicitly parallel operations such as fork-join or implicitly parallel operations such as map and reduce. These abstractions in theory should encourage developers to think "parallel" and write parallel programs, but in practice, even with them, parallel programming is challenging because of new types of errors such as data races and the diversity of parallel machines (for example, message passing, shared memory, and SIMD).

So, what can a developer do to improve the performance of his or her code on a modern, parallel microprocessor with multiple cores and vector processing units? The following paper advocates an appealing division of labor between a developer and a compiler, with the human restructuring code and data structures and forcing parallel execution of some loops, thereby increasing the opportunities for the compiler to generate and optimize parallel machine code.

The results in this paper are quite striking. For 11 computationally intensive kernels, code developed in this manner performed within an average of 30% of the best hand-optimized code and did not require the developer to use low-level programming constructs or to understand a machine's architecture and instruction set.

But why is this division of labor necessary? Why are compilers unable to parallelize and vectorize these (relatively simple) programs? The authors allude to "difficult issues such as dependency analysis, memory alias analysis, and control-flow analysis." In practice, compilers employ a large repertoire of local optimizations, each of which incrementally improves a small region of code. Large, pervasive restructurings that change how a program computes its result are outside of the purview of a traditional compiler. Until recent work on program synthesis, there has been little research on efficient techniques for exploring large spaces of possible transformations. Moreover, even for local optimizations, compilers are hamstrung by conservative program analysis, which at best only approximates a program's potential behavior[a] and must disallow optimizations that might adversely affect a program's result.

The following paper argues the restructurings and annotations, when performed by developers, should be part of every programmer's repertoire for modern computers.

This paper argues the restructurings and annotations, when performed by developers, are not difficult and should be part of every programmer's repertoire for modern computers. These changes include transforming an array of structures into a structure of arrays, blocking loops to increase data reuse, annotating parallel loops, and adopting more parallel algorithms. Conceptually, none of these changes is difficult to understand—although finding a new algorithm may be challenging. However, these modifications can introduce errors into a program and can be complex to apply to a large application, where a data structure may be shared by many routines.

Of course, program optimization in general can have similarly pernicious effects on program structure and readability, so these concerns are not limited to parallel programs. Balanced against the challenge of directly writing a correct, high-performing parallel program, restructuring and annotation appear to be a reasonable methodology that produces maintainable programs. However, this approach would have little value if the resulting programs do not run significantly faster.

The paper's principal contribution is to demonstrate this division of labor between human and compiler achieves its goal of effectively using hardware parallelism to improve performance. Mature, modern compilers—aided by restructuring and annotation—can produce extremely efficient parallel code. Neither compilers nor people are very good at achieving this goal on their own. C

[a] Many program analyses, if fully precise, would allow solution of the Turing halting problem.

James Larus is a professor and dean of computer and communications sciences at EPFL, Lausanne, Switzerland.