DotPar
A simple, lightweight, implicitly parallelized language

Logan Donovan - lrd2127@columbia.edu
Justin Hines - jph2149@columbia.edu
Andrew Hitti - aah2147@columbia.edu
Nathan Hwang - nyh2105@columbia.edu
Sid Nair - ssn2114@columbia.edu

April 15, 2012

1 Motivation

Moores Law, the trend that the number of transistors that can fit on a chip doubles every 2 years, has held true for half a century, driving innovations in all areas of technology. But recently chip manufacturers have moved away from ever-increasing clock speeds, the usual manifestation of Moores law, and instead increase the number of cores in CPUs to manage power dissipation.

As consumer and business demands for speed grow, programs must now be written to take advantage of multiple cores in order to utilize the power of CPUs. However, the imperative programming paradigm that has prevailed in the programming community isnt conducive to safe, parallel programming. Parallel programming is difficult and synchronization bugs are notoriously difficult to fix.

Many solutions to this problem of parallelization have emerged, but each has its drawbacks. We will briefly survey a few of the most prominent existing solutions.

1.1 Locking

Locking controls access to memory accessible by several threads of execution. However, this requires that the programmer manually lock regions of code, which is tedious and error prone. Historically, it can lead to deadlock and race conditions, both of which are extremely hard to debug.

1.2 Task-based Concurrency and the Actor Model

Task-based concurrency, used in languages like Ada and X10, doesnt expose threads to the programmer. Simple computations, not threads are the unit of parallelization. The actor model, present in SmallTalk and Scala, treats different threads as actors that communicate with each other via message passing. Since the actors dont share state, this is much less error prone than locking since race conditions can be avoided entirely. These models can be elegant, but they still require manual division of labor on the part of the programmer, which introduces errors and overhead, especially when refactoring code. We seek to avoid imposing this burden on programmers.

1.3 Functional Programming

The functional paradigm minimizes side effects in favor of deterministic functions. Under this model, functions could easily be parallelized because they execute independently. Functional languages like Haskell boast compilers that produce a great degree of parallelization.

However, the functional paradigm is an unfamiliar way of coding for many programmers and having state is often a preferable way of modeling a program. In particular, object-oriented approaches have proven useful in building large applications. These factors have limited the adoption of purely functional languages.

Additionally, the benefits of parallelizing a purely functional programming languages is limited because objects must be frequently copied because they are immutable. This means that cache hit rates drop significantly, main memory bandwidth becomes a bottleneck, and garbage collection becomes a pain[1].

1.4 Implicit Parallelization

Some efforts to write implicitly parallel, imperative languages have been made[2]. However, state- of-the-art compilers are not yet capable of good implicit parallelization. While some cases of potential parallelization are easy to recognize, many of them difficult or impossible to find with static analysis[3].

With this in mind, implicit parallelization may sound daunting, but it is important to remember that the compiler can miss many potential opportunities for parallelization without sacrificing performance. Even with a modest ability to find parallelization opportunities, the performance of parallel programs is still limited by the number of processors, not the number of threads that run simultaneously. In fact, if too many threads are created, the overhead of thread creation and execution will adversely affect performance.

[1] http://bit.ly/dotpar1

[2] See http://bit.ly/dotpar2 and http://bit.ly/dotpar3 for two examples.

[3] http://bit.ly/dotpar4

2 Introducing DotPar

With familiar C-like syntax, DotPar is a multi-paradigm language that implicitly parallelizes code with minimal programming effort. By abstracting away parallelization, DotPar enables programmers to focus on designing and and architecting systems instead of on micromanaging performance. Additionally, it provides the expressive power of functional languages and includes some imperative paradigms derived from C to ease the job of the programmer.

2.1 Automatic Parallelization

Based on static analysis of code and user annotations, DotPar can detect a large class of parallelization opportunities, speeding up execution considerably.

2.2 Static and Strong Typing

Keeping DotPar strongly and statically typed provides compile-time check for common mistakes that should never make it to production. This also enhances the compilers ability to parallelize the code.

2.3 First-class Functions and Lexical Closures

By providing first-class functions and closures, DotPar provides a great degree of expressiveness not possible in a language like Java. However, it is not as restrictive as a purely functional language like Haskell because it provides some imperative constructs as well.

2.4 Robustness and Security

Through implicit parallelization, DotPar removes the risk of race conditions and deadlock. And, since it runs on the JVM, the code is highly robust and portable. Furthermore, DotPar raises informative errors that allow the user to debug code efficiently and quickly, reducing development time.

2.5 Memory Management and Garbage Collection

Since DotPar runs on the JVM, garbage collection and memory management are handled automatically, reducing programmer overhead.