

Reasoning about Parallelism

Wayne A. Kelly

Microsoft QUT eResearch Centre, Australia

If general purpose applications are to run faster in the future they will have to exploit parallelism. Currently there are two approaches to parallelising a program - 1) the programmer explicitly expresses the parallelism. 2) a compiler automatically detects parallelism. Option 1 is too difficult to expect everyday programmers to do well. Option 2 is clearly preferable but is well beyond the state of the art for arbitrary general purpose applications. So, if the problem is too hard, rather than giving up, we should *change/simplify* the problem. Whether it is a programmer or a compiler trying to parallelize an application, the fundamental problem is the same – it is difficult to *reason* about the inherent parallelism in imperative programming languages. This should come as no surprise as they were never designed for this purpose. With the mainstreaming of many-core processors, traditional programming languages face a clear and imminent challenge.

What is it that makes reasoning about parallelism in imperative programs difficult? Clearly it is has to do with the presence of mutable state, as pure functional programs are much easier to reason about. I do not believe, however, that the solution to the many-core challenge is to expect all programs in the future to be coded in a pure functional style. Forcing functional programming on all programmers would constitute a gigantic disruptive change. There are also other drawbacks to functional programming. While they are Turing equivalent in terms of expressibility, many programmers find it more nature to express certain algorithms in an imperative fashion. And because the underlying machines that we are programming are inherently state machines, imperative algorithms often execute more efficiently than equivalent functional algorithms.

Just because unrestrained use of mutable state can make it difficult to reason about parallelism does not mean that we should necessarily banish it completely. Fire, also is an extremely dangerous element, but when used in a carefully controlled way, becomes an invaluable resource. What we need to do is learn how to *appropriately* control and restrict our use of mutable state. It is illustrative to note that all of the major break throughs in programming language and paradigm design have been in essence a self imposed restriction on constructs that had previously been used in an uncontrolled manner. Consider for example, structured control flow as heralded by Dijkstra's famous "Go-to Considered Harmful" paper. By limiting the way in which code branches we obtained great advances in comprehensibility, maintainability and in our ability to reason about program semantics. Subsequent advances such as object-oriented programming, component-oriented programming, aspect-oriented programming and even functional programming similarly

represent a disciplined use of various constructs. The need to exploit new parallel hardware will I believe herald a new more disciplined use of mutable state.

At present we have abstractions such as functions and methods for managing control flow complexity. Control flow within each method is structured and so can be reasoned about. We similarly have data structure abstractions for managing data complexity. So, while both code and data is structured, the relationship between code and data is not sufficiently structured. If we take a code abstraction such as a method, in general it is not easy to determine which data an invocation of that method might read and write. It is therefore difficult to determine whether two such methods could be executed in parallel without interfering with one another.

The data that a method reads and writes is easy to determine empirically at runtime – but it cannot easily be determined at compile time and so cannot be reasoned about statically. In a component oriented world, we need to be able to reason about behaviour based solely on information provided in interface definitions. At present most type systems inform us about the input and output types of each method, but they say little about what data that method might read and write. C++'s `const` declaration is an example of type system allows us to control the writing of data in certain methods and so is a step in the right direction.

Advanced type systems have been proposed that allow the effects of methods to be captured. However most such type systems are extremely complex – to the point where getting the types correct becomes more difficult than writing the actual code. Clearly, we need to be pragmatic if we are to create a new programming paradigm that we expect the vast majority of programmers to adopt. We do not necessarily need a fully general effect type system, we just need a minimalist change to current object-oriented type systems that allow us to reason about the common forms of parallelism that arise in everyday programs. Our reasoning need only provide conditions that are sufficient to imply parallelism – they need not find all possible parallelism. Simplicity must be kept foremost to achieve widespread adoption.