A Heterogeneous Parallel Programming Language and Compiler Architecture

A Part III Project Proposal

N. S. Tomlinson (*nst25*), Robinson College
18th November 2012

Project Supervisor: Prof Alan Mycroft

Abstract

Today's computers are becoming increasingly parallel in nature, owing to limitations imposed by limits on instruction-level parallelism and design team size preventing the creation of ever higher performance superscalar processors. If processors become more parallel, then programmers are required to write parallel programs. Unfortunately, today's programming languages do not allow programmers to write parallel programs in an intuitive way while permitting the compiler to perform optimisations on the parallelism itself.

This project aims to by answer the following questions: How might a programming language be designed that allows the programmer to use their intuition about how programs should be written to write parallel programs? How might we design a compiler to compile and optimise that language to several different targets that interact with each other?

1 Introduction

The design of processors is being pushed towards designs that exploit data-level parallelism, and task-level parallelism in order to significantly improve performance. This requires programmers to write parallel code, but at present the most intuitive programming paradigms do not permit this to be done easily. Most programming languages that allow parallelism to be expressed are not well suited as an intuitive general purpose programming language that produces efficient parallel code without hand optimisation.

One class of programming languages that permit expression of parallelism is functional programming languages. Functional programming languages typically have no mutable state, and so the compiler is able to perform more trivial static analysis on the program in order to determine what may be calculated in parallel. However, many programmers find imperative programming more intuitive, and there are many applications such as system controlling that poorly fit into the computation model provided by functional programming language.

Commonly used programming languages such as C can provide primitives for working with threads – such as the C library `pthreads` that provides methods for creating threads, mutexes, and so on. The sequential part of the code is written in a way that is intuitive to the programmer. Unfortunately, the parallelism must be micro-managed and there is no opportunity for the compiler to optimise. In this way, these languages are akin to assembly in that there is very little abstraction of the parallel primitives exposed by the operating system. There are several consequences of hard-coding parallelism in this way. Firstly, the implementation of parallelism is error prone. Secondly, it is not possible for the compiler to perform optimisation on the parallelism. Thirdly, this approach leads to code that cannot

be ported – it is not possible to adapt a program written for a multi-core CPU to run on a GPGPU without significant work.

Languages such as OpenCL require the programmer to separate the parallel code from the sequential code. They support data types that are appropriate for SIMD operation, and they present a computing model that is intuitive to the programmer. In such languages, the parallel code is represented as a kernel that is executed many times, differing only by a global ID (which may be used for example, to calculate an index into a large array). This is a much higher-level representation of parallelism, and permits portability between different systems. Unfortunately, this separates out the parallel part of the code from the sequential to too great an extent. It becomes difficult to write programs that share code and ideas between the sequential and parallel component. Arguments to kernels are represented in the sequential code as library calls that add a single argument to the end of a list of arguments. This makes it very difficult for the compiler to check this for correctness in any case, and impossible in the general case. The it is hard for the compiler to perform optimisations that require changes of both the parallel code and the sequential code. For example, AMD's compiler does not optimise executions of a kernel on a single data item into an equivalent SIMD or loop based kernel, as this would require changing the sequential code to create fewer work items, and changing the kernel to operate on more work items. Hence the programmer is forced to hand-optimise this aspect of the code in order to prevent the performance from being inferior to that of a sequential implementation.

Languages such as OpenMP work by extending an existing language, in this case C, C++, or Fortran. OpenMP allows the programmer to annotate the code with `pragmas` that alter the semantics of some of the structures to include parallelism. Like OpenCL, OpenMP fits a programmer's intuition well, but it is hard to perform optimisations on the parallelism. Additionally, since OpenMP is an extension of an existing language, it is limited to semantics compatible with the original. OpenMP also lacks a more powerful set of concepts that would allow for easy use of GPGPUs where there is no global shared memory.

Cilk is an extension of the C programming language, adding five new keywords: `cilk`, `spawn`, `sync`, `inlet`, and `abort`. Cilk presents a thread based model of parallelism – the `spawn` keyword instructs the compiler to generate a new thread. The `sync` keyword instructs the compiler to wait for each thread spawned in the function to finish. Wool is a C library that provides for a similar programming model, but as a C library rather than a language in its own right. These concepts are indeed intuitive and very powerful, but they do not well express data-level parallelism well enough for GPGPU applications, nor support for disjoint memory to allow the use of both CPU and GPGPU.

X10 is a language that is designed for a similar computing environment to that envisaged by us – a cluster-like environment where processors may or may not have access to shared memory. It has the concept of places. These can be thought of as a single machine in a cluster of machines – with its own memory, and its own processor cores. Items of data belong to a single place, where this data may be accessed in sequential code. Items in a place other than that on which the current thread is executing may only be accessed asynchronously. X10 supports spawning and management of new threads with `asynch` and `finish` concepts that are similar in nature to Cilk's `spawn` and `sync`. X10 also supports `atomic`, which prevents other threads from executing at the same time as the atomic thread. These concepts are powerful, however X10's concept of places forces the programmer to make decisions about the location of data within the computer cluster, rather than allowing the compiler to optimise this.

2 Approach & Outcomes

"How might a programming language be designed that allows the programmer to use their intuition about how programs should be written to write parallel programs?" I intend to design an imperative-like programming language that has support for block structures with parallel semantics, appropriate data types, and support for an environment with no globally shared memory.

"How might we design a compiler to compile that language to several different targets that interact with each other?" I intend to produce a compiler for the designed language that can target several different parallel computing devices, including use of these devices together in a heterogeneous computing cluster. The design of the compiler will inform the design of the language, in addition to vice versa. I would like to design the compiler in a way that allows variants of similar systems (such as GPGPUs with different amounts of memory and ALU types) to be targeted. This should extend to clusters of machines that consist of several different systems that are specified in this manner. The compiler will be required to perform optimisations on the parallelism itself rather than on the sequential code (which is usually handled by target compilers such as LLVM). Such compiler optimisation should enable the programmer to write code that expresses a great deal of parallelism, while the compiler decides how to exploit it efficiently.