Chapter 1: Introduction

In 1965 Moore [3] predicted that the transistor density on processors would grow exponentially with time, and the manufacturing cost would fall. The smaller transistors are, the faster they can switch (given adequate power), and therefore manufacturers can ship faster processors. The industry celebrated this trend, calling it Moore's Law. However as faster processors require more power, they create more heat which must be dissipated. Without novel power saving techniques (such as Bohr at al [1]), this limits increases of processors' clock speeds.

Around 2005 it became clear that significant improvements in performance would not come from increased clock speeds but from multicore parallelism [4]. Manufacturers now build processors with multiple processing cores, which can be placed in the same package, and usually on the same die. Individual cores work separately, communicating through the memory hierarchy.

Other methods of improving performance without increasing clock speed have also been tried.

- Modern processors perform super-scalar execution: processor instructions form a pipeline, with several instructions at different stages of execution at once, and by adding extra circuitry, several instructions may be at the same stage of execution. However, we have just about reached the limits of what super-scalar execution can offer.
- Manufacturers have also added Single Instruction Multiple Data (SIMD) instructions to their processors; this allows programmers to perform the same operation on multiple pieces of data. In practice however, SIMD is useful only in some specific circumstances.
- Multicore computing has the potential to be useful in many circumstances, and does not appear to have limitations that cannot be overcome. Cache coherency could be a limitation for processors with many cores. However there are solutions to this such as directory based memory coherency; there are also research opportunities such as making compilers responsible for cache management.

Multicore computing is a special case of multiprocessing. Most multiprocessing systems are symmetric multiprocessing (SMP) systems. An SMP system consists of several homogeneous processors and some memory connected together. Usually all processors are equally-distant from all memory location. Most multicore systems are SMP systems; they may have more than one CPU each with any number of cores. Some multiprocessing systems use a non-uniform memory architecture (NUMA). Usually this means that each processor has fast access to some local memory and slower access to the other processors' memories. A new type of architecture uses graphics programming units (GPUs) to perform general purpose computing, they are called GPGPU architectures. However they are not as general purpose as their name suggests: they work well for large regular data-parallel and compute-intensive workloads, but do not work well for more general symbolic processing. GPGPUs give programs access to small amounts of different types of memory of fixed sizes, however most symbolic programs rely on dynamic allocation of unpredictable amounts of memory. Additionally, symbolic programs often include code with many conditional branches; this type of code does not perform well on GPGPUs. GPGPU architectures are not as general purpose as SMP systems and SMP systems are vastly more common than NUMA systems. Therefore, in this dissertation we are only concerned with SMP systems as they are both more general and more common, making them more desirable targets for most programmers. Our approach will work with NUMA systems, but not optimally.

To use a multicore system, or multiprocessing in general, programmers must parallelise their software. This is normally done by dividing the software into multiple threads of execution which execute in parallel with one another. This is very difficult in imperative languages as the programmer is responsible for coordinating the threads [5]. Few programmers have the skills necessary to accomplish this, and those that do, still make expensive mistakes as threaded programming is inherently error prone. Bugs such as data corruption, deadlocks and race conditions can be extremely tedious to find and fix. These bugs increase the costs of software development. Software companies who want their software to out-perform their competitors will usually take on the costs of multicore programming. We will explain the problems with parallelism in imperative languages in section 1.2.

In contrast to imperative languages, it is trivial to express parallelism in pure declarative languages. Expressing this parallelism creates two strongly-related problems. First, one must overcome the costs of parallel execution. For example, it may take hundreds of instructions to make a task available for execution on another processor. However, if that task only takes a few instructions to execute, then there is no benefit to executing it in parallel. Even if the task creates hundreds of instructions to execute, parallel execution is probably not worthwhile. Most easy-to-exploit parallelism is *fine grained* such as this. Second, an abundance of coarse grained parallelism can also be a problem. Whilst the amount of parallelism the machine can exploit cannot increase beyond the number of processors, the more parallel tasks a program

creates, the more the overheads of parallel execution will have an effect on performance. In these situations, there is no benefit in parallelising many of the tasks, and yet the overheads of their parallel executions will still have an effect. This often cripples the performance of such programs. This is known as an *embarrassingly parallel*workload. Programs with either of these problems almost always perform more *slowly* than their sequential equivalents. Programmers must therefore find the parts of their program where parallel execution is profitable and parallelise those parts of their program *only*, whilst being careful to avoid embarrassing parallelism. This means that a programmer must have a strong understanding of their program's computations' costs, how much parallelism they are making available, and how many processors may be available at any point in the program's execution. Programmers are not good at identifying the hotspots in their programs or in many cases understanding the costs of computations, consequently programmers are not good at manually parallelising programs. Programmers are encouraged to use profilers to help them identify the hotspots in their programs and speed them up; this also applies to parallelisation.

Automatic parallelism aims to make it easy to introduce parallelism. Software companies will not need to spend as much effort on parallelising their software. Better yet, it will be easier for programmers to take advantage of the extra cores on newer processors. Furthermore, as a parallel program is modified its performance characteristics will change, and some changes may affect the benefit of the parallelism that has already been exploited. In these cases automatic parallelism will make it easy to *re-parallelise* the program, saving programmers a lot of time.

1.1 General approach

Unfortunately automatic parallelisation technology is yet to be developed to the point where it is generally useable. Our aim is that automatic parallelisation will be easy to use and will parallelise programs more effectively that most programmers can by hand. Most significantly, automatic parallelism will be very simple to use compared with the difficulty of manual parallelisation. Furthermore as programs change, costs of computations within them will change, and this may make manual parallelisations (using explicit parallelism) less effective. An automatic parallelisation system will therefore make it easier to maintain programs as the automatic parallelisation analysis can simply be redone to re-parallelise the programs. We are looking forward to a future where programmers think about parallelism no more than they currently think about traditional compiler optimisations.

In this dissertation we have solved several of the critical issues with automatic parallelism. Our work is targeted towards Mercury. We choose to use Mercury because it already supports explicit parallelism of dependent conjunctions, and it provides the most powerful profiling tool of any declarative language, which generates data for our profile feedback analyses. In some ways our work

can be used with other programming languages, but most other languages have significant barriers. In particular automatic parallelism can only work reliably with declaratively pure languages, the language should also use a strict evaluation strategy to make it easy to reason about parallel performance, and in the case of a logic language, a strict and precise mode system is required to determine when variables are assigned their values. Mercury's support for parallel execution and the previous auto-parallelisation system [2] is described in chapter 2. In this dissertation we make a number of improvements to Mercury's runtime system that improve the performance of parallel Mercury programs (chapter 3). In chapter 4 we describe our automatic parallelism analysis tool and its algorithms, and show how it can speedup several programs. In chapter 5 we introduce a new transformation that improves the performance of some types of recursive code and achieve almost perfect linear speedups on several benchmarks. The transformation also allows recursive code within parallel conjunctions to take advantage of tail recursion optimisation. Chapter chapter 6 describes a proposal to add support for Mercury to the ThreadScope parallel profile visualisation tool. We expect that the proposed features will very useful for programmers and researchers alike. Finally in chapter 7 we conclude the dissertation, tieing together the various contributions. We believe that our work could also be adapted for other systems; this will be easier in similar languages and more difficult in less similar languages.

1.2 Concurrency

Chapter 2: Background

Chapter 3: Runtime system

Chapter 4: Overlap

Chapter 5: Loop control

Chapter 6: ThreadScope

Chapter 7: Conclusion

References

[1] Mark T. Bohr, Robert S. Chau, Tahir Chani, and Kaizad Mistry. The high-k solution. *IEEE Spectrum*, 44(10):29--35, October 2007.

[2] Paul Bone. Calculating likely parallelism within dependant conjunctions for logic programs. Honours thesis, University of Melbourne, Melbourne, Australia, October 2008.

[3] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[4] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr Dobb's Journel*, 30(3), March 2005.

[5] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54--62, September 2005.