Java and the Power of Multi-Core Processing

Peter Bertels

Supervisor: Dirk Stroobandt

Abstract—**The new era of multi-core processing challenges software designers to efficiently exploit the parallelism that is now massively available. Programmers have to exchange the conventional sequential programming paradigm for parallel programming: single-threaded designs must be decomposed into dependent, interacting tasks.**

The Java programming language has built-in thread support and is therefore suitable for the development of parallel software. We are working on a framework and tool support to assist the programmer with the tedious task of parallel programming.

Keywords—**Java; multi-core; parallel programming**

I. Introduction

According to Moore's law the number of transistors on a single die is doubling every two years. During the last decades this evolution has lead to an exponential performance increase because processor clock speeds also doubled at the same rate. Due to power limitations this clock speed doubling came to an end. Computer architects came up with the idea of multi-core computing: large and complex processors are replaced by simpler and slimmer cores working together.

In this new era of multi-core computing, Moore's law can be applied to the number of cores. Current multi-core architectures consist of only a few cores, but this number is expected to increase as exemplified by the announcement of an 80 core architecture by Intel.

Software designers are challenged to efficiently exploit the massive parallelism available on these future multi-core architectures. In Section II we discuss parallelisation of sequential code as a first attempt to meet the multi-core challenge and we explain why this cannot completely solve the problems we are faced with. We feel that software designers have to exchange the conventional sequential programming paradigm for parallel programming. This transition requires a fundamentally different way of thinking about algorithms and programming problems. We elaborate on the challenges imposed by this transition in Section III.

Parallel programming languages and programming models are needed to help programmers with this 'parallel thinking'. Multi-threaded programming languages can be used to describe parallel behaviour, e.g. Java. We feel that Java alone is not sufficient, because multi-threaded programming is cumbersome. In Section IV we explain how Java can be extended to make it better suited for multi-core programming and we discuss related work and similar approaches.

II. Why parallelisation cannot solve the problem

After more than 40 years of research in the field of parallelising compilers, the results are still limited. For fairly simple and regular code, loop nests in particular, parallelising is straight forward. But automatic parallelisation of irregular code with lots of data dependencies is still not completely solved.

Control dependencies severely limit the available parallelisation. The control flow in useful and real-world applications is often complex and highly data dependent. Lam et al. propose to relax the constraints imposed by control flow and they suggest several techniques: speculative execution, control dependence analysis and following multiple flows of control [1].

Although these techniques for automatic parallelisation increase the possible parallelism in applications, these efforts are still too limited to effectively use 80 cores on a multi-core architectures for general purpose applications. We can conclude that automatic parallelisation is useful for very regular applications but that other applications require a more specific, manual approach. Software designers have to adopt to the parallel programming paradigm and have to come up with new, inherently parallel, software solutions.

III. Challenges of parallel programming

The most important challenge for parallel programmers is the *decomposition* of a single application into several, dependent and interacting tasks. The efficiency of the parallel program is highly dependent on this decomposition step: it determines the synchronisation and communication overhead.

Other challenges are *synchronisation* and *communication* between parallel threads. Synchronising parallel threads is a tedious task: synchronising too often leads to inefficient program execution but not enough synchronisation can lead to incorrect results due to data races or conditional hazards. Faulty synchronisation can lead to deadlocks.

We identify *load balancing* as our fourth important challenge. If we can achieve an appropriate decomposition of the problem in several tasks that can be run on multiple cores, we have to think about executing these tasks efficiently in parallel. How can we divide the work load equally among all these cores? Can we avoid that some cores remain idle? A related challenge is *scheduling* of all these parallel threads and tasks.

Finally we want to add *scalability* to our list of challenges. If we want to execute the same application as well as on a multi-core architecture with 4 cores as on an architecture with 80 cores, we have to think about scalability. Can we describe our parallel algorithms in such a way that we can exploit the available parallelism on an 80 core machine and nevertheless execute the same parallel program efficiently on a quad core?

IV. Can Java be the solution?

The Java programming language has built-in thread support. Therefore, it seems a good candidate to become the ultimate language for parallel programming. However, as Edward Lee

Peter Bertels is with the Department of Electronics and Information Systems (ELIS), Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium. E-mail: peter.bertels@ugent.be.

pointed out, there are some problems with threads. Threads are fundamentally flawed as a computation model because they are wildly nondeterministic [2]. Executing the same program twice can lead to different results. Programmer's are expected to prune this determinism by adding synchronisation, semaphores, monitors etc. This is a tedious task and synchronisation errors are almost impossible to avoid. Moreover synchronisation problems often remain undetected: the nondeterminism makes it difficult, if not impossible, to write test cases that cover all these possible faults.

In this section we elaborate on several approaches to overcome the problem with Java threads: the zJava project developed at the University of Toronto [3] and the DataRush framework developed by Pervasive Software [4].

A. The zJava project

The zJava project investigates automatic parallelisation of Java programs [3]. The basic idea of this project is to combine a compile-time and a run-time method to exploit parallelism among methods in Java applications. The zJava run-time systems starts a new independent thread for each method invocation. This thread executes asynchronously with the main thread which consists of the main method of the application. The run-time systems uses compile-time information about memory accesses to determine when a thread may execute and which dependencies need to be enforced.

Chan and Abdelrahman have evaluated their approach on a Sun quad core multiprocessor. They report a scalable speedup when using 1, 2, 3 or 4 cores. For some benchmarks the ideal speedup, equal to the number of processors, was achieved.

This approach overcomes the nondeterminism problem in the sense that the original program is fully sequential and that the run-time systems which creates the threads is supposed to do this in a correct-by-construction manner. The zJava project is essentially an automatic parallelisation tool as explained in Section II: it helps the programmer to make use of the parallelism available in the sequential program, but it does not increase the parallelism. This seems to be a viable approach for speeding up applications on multi-core architectures with a few cores, depending on the inherent parallelism in the sequential program.

B. Pervasive DataRush

The DataRush framework [4] developed by Pervasive Software enables the programmer to describe parallel programs as a directed dataflow graph in an XML based DataRush process composition language, DFXML. The basic operators in this dataflow graph are written in standard Java. DFXML also allows to specify hierarchical dataflow graphs. A run-time system can execute these applications efficiently on a multi-core platform.

DataRush provides special operators which can be used to wrap collections of processes and other operators together. These operators can be used to express several forms of parallel decomposition: horizontal decomposition, i.e. one process can be duplicated on several cores to process multiple data objects in parallel, vertical decomposition, i.e. several processes can process the same data object in parallel and pipelining. The run-time systems uses this information to efficiently map the application on the given platform. The number of available processor cores determines the amount of parallel decomposition.

The nondeterminism problem with threads is solved because the programmer has to specify interactions and dependencies between threads explicitly. This framework is also scalable from multi-core platforms with only a few cores to much larger multiprocessors. The only drawback is that the DataRush framework somewhat limits the possibilities, e.g. cyclic dataflow graphs cannot be represented — this is an easy way to guarantee deadlock freedom — and processes can only communicate via input and output streams whereas shared variables would be an interesting feature, especially if we want to transform legacy code for single processing into code for multi-core platforms.

C. The best of both worlds

We propose something which could be described as a combination of the aforementioned techniques. Currently we are working on a theoretical basis for a new framework which will, as the DataRush framework, enable programmers to explicitly describe interaction between processes described in Java, but which on the other hand would support communication via shared memory. Our research is currently focussed on the definition of a subset of the shared memory paradigm which is limited enough to guarantee that it would fit in such a framework but which is nevertheless practically useful.

V. Conclusions and Future work

Multi-core architecture have recently become the name of the game in computer industry. These platforms provide massive parallelism to software designer who are trained to write inherently sequential code. In this article we summarised the challenges these programmers are facing when they make the transition from the conventional programming paradigm into the new era of parallel programming. New programming models and tools can support this transition. We described recent approaches to extending the Java language to make it more suitable for programming multi-core architectures and propose some new ideas and extensions to alleviate the burden of parallel programming.

In order for our new programming model to be useful we plan to implement a run-time environment — built on top of the Java virtual machine — to execute our parallel programs.

Acknowledgments

Peter Bertels is supported by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). His research is also partially funded by the IWT grant 060086.

References

[1] Monica S. Lam and Robert P. Wilson, "Limits of control flow on parallelism," in *ISCA 1992: Proceedings of the 19th annual international symposium on Computer architecture*. 1992, pp. 46–57, ACM Press.

[2] Edward A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[3] Bryan Chan and Tarek S. Abdelrahman, "Run-time support for the automatic parallelization of Java programs," *Journal of Supercomputing*, vol. 28, pp. 91–117, 2004.

[4] Pervasive DataRush, "A java framework for dataflow applications: unleash the power of multi-core," http://www.pervasivedatarush.com/.