

Automatic Approach Towards Actor-Oriented Programming

Peter Bertels and Dirk Stroobandt
Ghent University, ELIS
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
peter.bertels@ugent.be

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, but programming multi-threaded applications is a tedious task. Therefore we are working on a framework and tool support to alleviate the burden of threads, synchronisation and locking, based on process networks. This paper describes our initial ideas for this new programming model.

Index Terms—Java, profiling, application partitioning, actor-oriented models, parallel processing

I. INTRODUCTION

According to Moore’s law the number of transistors on a single die is doubling every 18 months. 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.

This new era of multi-core computing challenges software designers to efficiently exploit the parallelism that is now massively available on these multi-core architectures. Programmers have to exchange the conventional sequential programming paradigm for parallel programming. As Edward Lee correctly pointed out [6], using threads as a model of computation, makes this parallelisation process a tedious task.

A promising approach is the use of actor-oriented models for the specification of the parallel application. Because of the explicit modelling of inter-task communication, actor-oriented models are easier to reason about than threaded applications. Moreover, these models allow for scalable and efficient execution on multi-processing platforms [10], [3].

In this paper we show how an actor-oriented model can be extracted from a conventional sequential program. Our two-step approach consists of profiling and a partitioning step. First, the application is profiled leading to a detailed view on the communication streams between several methods in the program. Second these communication profiles are used for communication-aware partitioning minimising inter-task communication.

Our approach enables the conversion of conventional Java programs to actor-oriented models which can be executed by the DataRush framework [3]. This fully automatic approach results in considerable performance improvements.

II. PARALLEL PROGRAMMING AND PARALLELISATION

A. 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, parallelisation is straightforward. 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. Several papers [5], [7] 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.

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 architecture 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.

B. 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 on a multi-core architecture with 4 cores as well 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?

III. CAN JAVA SOLVE THE PROBLEM?

The Java programming language has built-in thread support. Therefore, it seems a very suitable language for parallel programming. However, as Edward Lee pointed out, there are some problems with threads [6]. Threads are fundamentally flawed as a computation model because they are wildly non-deterministic. Executing the same program twice can lead to different results. Programmer's are expected to prune this non-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 non-determinism 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 [2] and the DataRush framework developed by Pervasive Software [3]. We will reveal the differences between these approaches with an example: an image filter operation.

A. Example: filter operation

As a running example for this section we use a simple image manipulation: filtering. The pixels in the original image represented by matrix a are averaged over a sliding window of 3 by 3 pixels and the filtered image is stored in a matrix b . This simple algorithm is clearly suitable for parallel computation: the values of every pixel in matrix b can be computed independent from each other.

B. The zJava project

The zJava project, by Chan and Abdelrahman [2], investigates automatic parallelisation of Java programs. 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 system 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 system uses compile-time information about memory accesses to determine when a thread may execute and which dependencies need to be enforced. Chan and Abdelrahman report a scalable speedup when using 1, 2, 3 or 4 cores of a Sun quad core multiprocessor. For some benchmarks the ideal speedup, equal to the number of processors, has been achieved.

This approach overcomes the non-determinism problem in the sense that the original program is fully sequential and that the run-time system 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-A: 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.

If we refactor our example program, so that the actual filtering is placed in a separate method, we can use zJava to parallelise it. The i and j loops will create new threads for each pixel in matrix b . For each filter operation the matrix a has to be locked in order to avoid other threads of writing to a during filtering.

C. Pervasive DataRush

The DataRush framework [3] 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 system uses this information to efficiently map the application on the given platform. The number of available processor cores determines the amount of parallel decomposition.

Horizontal parallel decomposition can be used to parallelise our filter operation. However, we have to drastically change our source code if we want to make a DataRush operator from this averaging filter. DataRush prevents us from communicating Java objects — shared data on the heap — through the FIFO buffers. This means we have to communicate the 9 pixels from the sliding window directly to our filter. This way DataRush gets around problems as locking and data races.

The non-determinism 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.

D. DataRush versus zJava

Both zJava and Pervasive DataRush present solutions for parallel programming in Java. The zJava approach is automatic parallelisation: it performs quite good but its use is limited to applications with an analysable control flow which is not or at least not heavily data dependent. Due to the limited amount of parallelism in general purpose applications the automatic parallelisation provided by zJava is less suitable for multi-core platforms with lots of cores.

DataRush on the other hand provides a framework for explicitly describing parallel programs with a combination of Java and XML dataflow graphs. This approach is highly scalable, but limited to relatively easy and straightforward parallel applications: applications without loops and with a limited interaction between the tasks.

IV. OUR APPROACH

The previous sections motivate our quest for a new approach which could be described as in-between the aforementioned techniques. Currently we are working on a framework which will enable programmers to explicitly describe interaction between processes described in Java, but which on the other hand would support communication via shared memory and the associated locking problems.

The DataRush framework is a good starting point to build a more extended framework upon. The next subsections describe the basic idea and the problems we have to solve in our extension of the DataRush framework.

A. Actor and object orientation

Our basic idea is to combine an object-oriented programming environment, as provided by standard Java, with an actor-oriented design, as in the process networks used by the DataRush environment. All software designers are familiar with object-orientated programming: it provides among others flexibility, modularity and maintainability to the programmer. The support for inheritance, polymorphism and encapsulation allows one to define a clear hierarchical data model with separation of concerns in the definition of data objects.

Actor-oriented design focuses on concurrency and communication between components. It provides a similar clear hierarchical model with well-defined interfaces for the functionality of the application, just as object-orientation provides these properties for the data. This way, actor-orientation and object-orientation are complementary. The actor-oriented approach allows the programmer to specify in a strict and well-defined manner the communication and interaction between components and therefore counters the non-determinism which occurs when using the alternative approach with threads.

We propose to use *process networks* as a model of computation in the actor-oriented approach. Kahn [4] first proposed the idea of using process networks to describe parallel processing and parallel algorithms. His Kahn Process Networks represent operator nodes which can communicate via FIFO buffers. These original networks were strictly meant as a theoretical model of computation and therefore they have unbound FIFO

buffers on the communication edges. Practical implementations, general process networks, however use finite buffers which can lead to deadlock. In Section IV-C we discuss this problem in more detail.

The combination of object-orientation and actor-orientation clearly provides our approach with some interesting advantages: the process network approach enables the composability of the components in the network and possible reuse of other components in a new context. Moreover we can reuse design patterns provided by standard parallel decomposition blocks for horizontal, vertical and pipelined parallelisation¹ which are available to some extent in DataRush. Just like the DataRush environment, our framework also allows rigorous specification of communication by means of process networks.

Due to the use of Java in our approach, legacy Java code can easily be ported into this framework. Our approach facilitates the description and implementation of parallel concepts which enables the programmer to truly benefit from parallelisation without the burden of manually creating threads and solving synchronisation issues.

B. Handling shared memory

In the process network model of computation, the FIFO channels carry data objects which are used to trigger the actual computation in the actor nodes. Data has thus a dual function within the model: representing information and driving the computation. Therefore process networks do not use shared memory: objects only exist on the communication paths between operators.

Adding shared memory to a process network is not trivial: appropriate locking is required in order to avoid multiple actors modifying the same object simultaneously. As one of the problems with parallel programming is the need for explicit locking, we are focussing our research on the strict definition of a subset of the shared memory paradigm which fits nicely in our extended process network model of computation. This subset has to be limited enough to guarantee that it does not violate the determinism of process networks but nevertheless extensive enough to be practically useful. This would ease the adoption of the framework by conventional software developers.

The DataRush framework allows operators to access objects in the Java heap memory. The programmer has to properly synchronise these memory accesses himself as the DataRush framework does not provide any solution for this. Therefore the non-determinism we had with the threaded approach remains unsolved. Within DataRush it is impossible to communicate Java objects through FIFO buffers.

We add the possibility to communicate Java objects through the FIFO queues by means of reference passing. All objects are protected by our framework in the sense that these objects can be accessed by only one operator at a time. But we will not prevent multiple references to an object from being

¹We use the term horizontal parallelisation to indicate a form of parallelism where the same operation is performed on multiple data simultaneously. Vertical parallelism in this context means that the same data object is simultaneously processed by multiple operators.

passed around on several communication links in the process network. Whenever an object actually enters an actor in the network, it gets locked and other operators cannot access it anymore. This strict procedure avoids any non-determinism or strange behaviour and is completely transparent for the software programmer.

However, sometimes this is too stringent: in order to execute our averaging filter example in parallel, several individual filter operators have to access matrix a simultaneously. But as we can see, these operators will only be reading, not modifying the data. Therefore this action can be allowed. We use a *contract-based* approach which allows operators that can guarantee a read-only behaviour to process objects on their read-only FIFO ports even if these objects are locked by other actors. This can be combined with for example a horizontal parallelising operator which reads tokens from an input buffer and then duplicates them to send them to multiple copies of a parallelisable actor: e.g. in our filter example we can duplicate the incoming reference to matrix a and communicate this reference to several instantiations of the filter actor.

Objects which are declared *final* and therefore cannot be modified, are obviously allowed to be read by all operators.

C. Deadlock detection

The DataRush framework prevents programmers to build cyclic dataflow graphs. Therefore it can guarantee, by construction, a deadlock free behaviour of the application. However, this also makes DataRush unsuitable for a broad range of general-purpose applications. Video encoders for example are fairly regular applications which are also highly parallel, but are unsuitable for DataRush because their dataflow graph is not acyclic.

If we extend the framework and enable cyclic graphs, we have to solve the deadlock problem. Parks [8] proposed an algorithm to do this: first a deadlock will be detected, then the blocking write operation causing the deadlock will be determined, finally one of the bounded FIFO queues will be enlarged in order to relieve the deadlock.

An efficient implementation of Parks method will be implemented in our framework.

D. Some related approaches

In 1998, Randall proposed the Cilk programming model [9] for efficient multithreaded computing. Cilk is an extension of GNU C with a few specific keywords for parallelism. The programmer has to expose the parallelism explicitly in the code. This solves the problems we mentioned in Section II-A. Randall also developed the Cilk scheduler which divides the execution of Cilk procedures among multiple processors in an efficient way, based on the Cilk parallel language constructs.

Another approach worth mentioning is the HPJava (high-performance Java) project [1] at Indiana University. Carpenter and Fox combine in HPJava ideas from earlier work on high performance fortran with ideas from distributed computing.

Very interesting in their work is the notion of distributed memory in the language, in the form of multidimensional arrays which can be distributed among concurrent processes in some of their dimensions.

V. CONCLUSIONS AND FUTURE WORK

Multi-core architectures have recently become the name of the game in computer industry. These and future platforms provide massive parallelism to software designers who are trained to write inherently sequential code. In this paper we summarised the challenges these programmers are facing when they make the transition from the conventional programming paradigm to parallel programming. New programming models and tools can support this transition. We have 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 and the DataRush framework — to execute our parallel programs.

ACKNOWLEDGEMENTS

Peter Bertels is supported by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). This research is also morally supported by the FlexWare project, grant 060086 of the same institution.

REFERENCES

- [1] Bryan Carpenter and Geoffrey Fox. Hjava: A data parallel programming alternative. *Computing in Science and Engineering*, 5(3):60–64, 2003.
- [2] Bryan Chan and Tarek S. Abdelrahman. Run-time support for the automatic parallelization of Java programs. *Journal of Supercomputing*, 28:91–117, 2004.
- [3] Pervasive DataRush. A java framework for dataflow applications: unleash the power of multi-core. <http://www.pervasivedatarush.com>.
- [4] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 1974*, North-Holland, Amsterdam, 1974.
- [5] 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*, pages 46–57, New York, NY, USA, 1992. ACM Press.
- [6] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- [7] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. pages 228–237, 1999.
- [8] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, 1995.
- [9] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.
- [10] Gang Zhou, Man-Kot Leung, and Edward A. Lee. A code generation framework for actor-oriented models with partial evaluation. *Lecture Notes in Computer Science*, 4523/2007:193–206, 2007.