UPSCALE

Upscale will develop a new programming language based on concurrent objects which integrates in the compilation process extended static analysis techniques targeted towards optimizing the deployment of concurrent objects unto multicore architectures.

AT A GLANCE

Project title:
From Inherent Concurrency to Massive Parallelism through Type-based Optimizations.

Project number:
612985

Project coordinator:
Prof.dr. F.S. de Boer, CWI. Email: F.S.de.Boer@cwi.nl

Partners:
CWI (Centrum Wiskunde & Informatica), Amsterdam, Netherlands, Imperial College, London, UK, University of Oslo , Norway, Uppsala University , Sweden

Duration:
February 2014 – February 2017.

Total cost:
€ 1.7 M

Programme:
FP7-ICT-2013-X

Website:
http://www.upscale-project.eu

Social Network:
twitter:@UpscaleEUF7

Abstract :

In a radical paradigm shift, manufacturers are now moving from multicore chips to so-called manycore chips with up to a million independent processors on the same silicon real estate. However, software cannot benefit from the revolutionary potential power increase, unless the design and code is polluted by an unprecedented amount of low-level, fine-grained concurrency detail. Concurrency in mainstream object-oriented languages is based on multithreading. Due to the complexity of balancing work evenly across cores, the thread model is of little benefit for efficient processor use or horizontal scalability. Problems are exacerbated in languages with shared mutable state and a stable notion of identity-the very foundations of object-orientation. The advent of manycore chips threatens to make not only the object-oriented model obsolete, but also the accumulated know-how of a generation of programmers. Our vision is to provide the means for industry to efficiently develop applications that seamlessly scale to the available parallelism of manycore chips without abandoning the object-oriented paradigm and the associated software engineering methodologies. We will realise this vision by a breakthrough in how parallelism and concurrency are integrated into programming languages, substantiated by a complete inversion of the current canonical language design: constructs facilitating concurrent computation will be default while constructs facilitating synchronised and sequential computation will be explicitly expressed. UpScale will exploit this inversion for a novel agile development methodology based on incremental type-based program annotations specifying ever-richer deployment-related information, and for innovative type-based deployment optimisations both at compile time and at run-time in the run-time system devised in **UpScale** for massively parallel execution. The targeted breakthrough will profoundly impact software development for the manycore chips of the future.

Background:

A radical paradigm shift is currently taking place in the computer industry: chip manufacturers are moving from single-processor chips to new architectures that utilize the same silicon real estate for a conglomerate of multiple independent processors, so-called multicores. Future predictions move beyond multicores into manycores: Ericsson foresees chips with more than a hundred cores readily available in the next years, and the Intel road-map mentions one million cores on a chip in the reasonably near future. The potential computing power delivered by such manycore chips is far beyond that of its single processor counterpart. Realising this potential, however, requires software developers to introduce an unprecedented amount of fine-grained parallelism considerations in the design of their applications.

The massively parallel execution of code enabled by manycore processors requires software abstractions for the coordination of interactions among parallel processes, as well as between processing and storage units. While concurrency has a long history in Computer Science, the classical context for the study of concurrency primitives differs from that provided by parallel manycore chips because the scale of parallelism offered by the latter is orders of magnitude beyond what contemporary concurrent programming models can comprehensibly express.

Object-orientation is the dominant programming paradigm in industry today; in fact, the TIOBE index shows that 4 of the world's 5 most popular programming languages are object-oriented (Dec, 2012). As a consequence, industry has heavily invested in object-oriented software engineering practices and know how.

The prevalent concurrency model in all of the top 5 TIOBE programming languages is multithreading. Even though support for multithreading requires only small syntactic additions to a sequential programming language, the development of efficient and correct concurrent programs for

multicore processors is very demanding. Naive use of concurrency does not necessarily pay o; for instance, loop-level parallelisation hardly improves performance if threads need to be created and destroyed, and application throughput is easily destroyed by serial bottlenecks unless parallelism saturates the program.

To address these issues, increasingly advanced language extensions, concurrency libraries, and program analysis techniques are currently being developed to explicitly control thread concurrency and synchronisation. Pessimistic" approaches are based on various locking constructs, chords, fork/join parallelism, atomic sections and lock inference, while optimistic" approaches comprise lock-free programming patterns and software transactional memory. Despite these advances in programming support, concurrency is still a difficult task: only the most capable programmers can explicitly control concurrency, and eciently make use of the relatively small number of cores readily available today.

We cannot expect the approaches developed to deal with multicore to scale to the manycore setting. The development of efficient programs that exploit the hundreds and even millions of cores expected by the transition from multicore to manycore systems, presents a quantum leap in complexity when compared to today's multicore computing. Because of the complexity required to express and balance the work evenly across such an amount of cores, the massive parallelism available on manycore chips goes far beyond what the available explicit concurrency abstractions based on multithreading can effectively exploit. As a consequence, the advent of manycore chips threatens to make object-orientation obsolete as a programming model, and thereby jeopardise the benets from the last 20 years of development in industrial software engineering and the accumulated knowledge of a generation of programmers.

Main Objectives:

Analysability By Design:
Core language elements will be designed for analysability relying on a programming model where properties which may potentially inhibit parallelism (such as synchronous communication and shared mutable state) must be explicitly declared. As a consequence of our design properties such as thread-locality and immutability, which facilitate reasoning about correctness and optimisation, will arise naturally in the programming. By making asynchronous communication the default, programmers must explicitly handle lost messages in their programming in places where the application logic requires it, but frees the system from meeting expensive delivery guarantees where they are not needed.

Extended Static Checking:
Our language design forces programmers to explicate where stronger guarantees (e.g., synchronous communication and delivery guarantees) are needed, or otherwise deviate (e.g., shared mutable state). We will leverage programmer annotations for specifying compile-time checked properties which will drive analysis of possible safe parallel execution, facilitated by the underlying formal semantics. For example, strong updates due to operations on message or statement levels must be identified to reason about their possible parallel application.

Incremental Application:
We will support incremental adoption along two different dimensions. First, we will support partial program annotation. This allows programmers to concentrate on the most relevant or cost-effective subparts of a program, or ignore performance-oriented annotations during prototyping or before the software is sufficiently understood. Second, we will allow programmers to specify properties at increasing levels of detail. This allows fast unlocking of coarse-grained parallelism between objects and subsequent unlocking of message-level parallelism inside objects as their internal semantics

stabilise. Finally, is allows (mostly implicit) fine-grained parallelism on the statement-level based on programmer-specified annotations of memory access patterns, where cost-effective.

Parametrised Deployment Configuration:
The actor model is a natural fit for concurrent as well as distributed computing. These different deployment scenarios however have fundamentally different cost models, for example, the overhead of sending messages to objects on the same core is orders of magnitudes cheaper than sending messages across a wire. Distributed systems and multicore systems provide different approaches to scalability and fault-tolerance, with different merit. In similar ways, scheduling policies of an object can vary with different QoS requirements of applications. By untangling deployment aspects from code, code can focused on functional requirements. The factored-out deployment information can then be weaved in by a compiler to optimise high-level constructs such as message sends, name binding and resource allocation and message scheduling.

Approach:

The approach taken in UpScale will consider the following aspects mentioned below:

- **Design**: leveraging deployment concerns in the programming language design.
- **Analysis**: Extended static checking using ownership and session types, and lightweight assertions to provide usable information to better address deployment concerns, both statically and dynamically.
- **Implementation**: development of a prototype compiler and run-time system for the programming language.
- **Application**: social simulation suite.

Impact:

The main breakthrough targeted by UpScale is a novel programming technology to enable the efficient utilization of massively parallel manycore chips by object-oriented programs. The long-term vision of UpScale is to develop the wherewithal for industry to be able to eciently develop applications that seamlessly scale to the available parallelism of manycore chips without abandoning the object-oriented paradigm and the associated software engineering methodologies. The fulfillment of this long-term vision requires a breakthrough in programming technology that enables software developers to master the complexity of balancing computations across massively parallel architectures. The realization of the breakthrough targeted by UpScale will be based on a fundamentally novel approach to how parallelism and concurrency abstractions are integrated into programming languages. Our approach involves a complete inversion of the current canonical language design so that language constructs facilitating concurrent computation are implicit and language constructs facilitating synchronised and sequential computation need to be explicitly expressed. UpScale will exploit this inversion for the incremental annotation of the code specifying ever-richer deployment-related information, so as to facilitate optimal deployment upon an advanced run-time system for massively parallel execution.