Multicore Programming
in Object-Oriented Languages

Author:
Behrooz Nobakht
bnobakht@liacs.nl

Supervisor:
Prof. Frank S. de Boer
F.S.de.Boer@cwi.nl

Leiden Institute of Advanced Computer Science
Leiden University

January, 2011

Contents

List of Tables

List of Figures

Abstract

In this report, we studied different languages and libraries that brought multicore programming methods into object-oriented languages. The methods included Actor model, software transactional memory, and data flow programming. The languages included Java, C++, Python, and C#; yet not all of them were thoroughly studied. Some hybrid solutions were also discussed in this report that show the diverse approaches in this line of research. We conclude by discussion on what the future of this research will be. This report is also accompanied with a presentation I did in this regards and is attached to this report.

Preface

The current report is the deliverable of my "research project" course as part of the curriculum for masters of computer science in LIACS with a credit of 17 EC's. The research is generally about how to deploy concurrent objects onto multicore technology and this report tries to present an overview of the past few years research on the topic. The research and project is supervised by Professor Frank S. de Boer at CWI. I am also honored to continue my master thesis under his supervision in this research.

Chapter 1

Introduction

A multicore processor is composed of two or more independent cores. One can describe it as an integrated circuit which has two or more individual processors (called cores in this sense) [15]. Designers may couple cores in a multicore device together tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods [15], [33, ch. 1].

1.1 History

Essentially, the problem starts when there comes two objects that are executing together and need to share data and synchronize behavior while each also may introduce behavior that can be executed simultaneously on another processing unit. Some classical approaches to tackle this problem include:

Lock-based programming in which there is always an object that acts as *lock* or a *mutual exclusion detector* and helps objects synchronize the act of entering a shared code or data section.

Multi-threaded programming that separates the line of execution and control from internal state of the object. There may be a number of objects in a thread that need to synchronize and share. In this approach, lock-based approach is also used.

Data-flow programming proposes a divide and conquer approach in which the programmer decomposes the data to be processed into parts through a function that she provides. After the processing is done, there is an aggregation of results again with a function that the programmer provides.

Philippsen in his survey on concurrent object-oriented languages (COOL) [42] states that the essential problem of multicore era is to bring together the widespread object-oriented paradigm with the power of multi-processor performance and functionality.

According to [42], the fundamental problems in COOLs are:

Parallel Performance that is the original reason for COOLs and is divided into categories:

- *Fan-out*: It is ideal to be able to create and spawn activities in a parallel mode in a programming language. The higher fan-out, the better the language.
- *Intra-object concurrency*: In a true concurrent language, there can be several methods in an object that are executing concurrently.
- *Locality*: that mainly talks about distribution and how to manage remote objects not to lose performance.

Broken Encapsulation. It is discussed that coordination is done either through *callee-side coordination* (*boundary coordination*) or *activity-centered coordination*. In both cases, it is argued that the fundamental concept of *encapsulation* is broken because of consideration of concurrency issues. The violation roots in the fact there is always a need to be aware of the internal state of the object to decide on the concurrency situation.

Inheritance Anomalies. There is a deep dependence interwoven in methods and instance variables in an object in which case when the object is required to be *reused*, it is inevitable to change super properties for coordination purposes. In this regards, there are two approaches: *centralized coordination style* and *separable coordination code*.

Expressive Coordination Constraints. Callee-side coordination requires mechanisms to express *proceed criteria*: *state* or *history*. State proceed criteria depends on the internal state of the object whereas history proceed criteria is based on the previous method calls to decide to continue.

Languages vs. Libraries. It is argued based on previous research that in general library extensions can not be implemented correctly. The basic insight is that the *compiler* does not know about the semantics of the added libraries and may have side effects in the correctness of the executing code.

The discussion in the survey [42] follows by comparing different languages and libraries to the date of the survey in the way how they allow **initiating concurrency** including automatic parallelization, fork/join, cobegin, forall, and autonomous code.

Then the survey discusses the approaches towards **coordinating concurrency** consistent of activity-centered coordination and boundary coordination. The boundary coordination in turn is possible with three different approaches: implicit control, handshake control, and reflective control.

At the end, the survey discusses about the concept of **locality** in concurrency and the fact that many researches and works have totally ignored this problem

for simplicity or prototyping purposes. The locality problem also leads to distribution and networked processing that is currently very important to the topic. The survey discusses that there are approaches to locality problem: meta-level locality, external locality, internal locality, virtual topology, and group locality.

In the survey [42], Philippsen concludes that there are some difficulties rising from the fact that parallelism and object-orientation have some contradictory issues. He also proposes to run research that would provide empirical data and experiments to be able to provide quantitative analyses over the topic.

There is a list of concurrent programming languages listed at [49] that seemingly have received more attention among others.

According to linearizability property of actions in threads, the set of all actions in a multi-threaded environment is linearizable; i.e. that there are no two actions that are running at the same exact time. However, in a multicore environment, there can be at least two actions running each in a different core (the maximum is the number of cores). It is also notable that in the past few years, the term "concurrency" has been used to represent multi-threading concepts; it is gradually replaced to introduce the concurrency that is introduced by multicore.

1.2 Parallel Programming

Parallel programming techniques can benefit from multiple cores directly that is the ideal approach to constructing software to utilize multiple cores. In this paradigm, there are four principle in essence to write a program to manage concurrency [15]:

Partitioning The focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.

Communication The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the communication phase of a design.

Agglomeration This phase of design is practically an optimization over the tasks already ready to be executed. It is the decision of the developer to recognize the level with which the tasks should be grouped or individualized for execution. The output of this phase may introduce clusters of tasks for execution.

Mapping In the fourth and final stage of the design of parallel algorithms, the developers specify where each task is to execute. This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling.

In another orthogonal perspective, parallel computing is categorized in four different levels [17]:

Bit-level parallelism On the single-core technology, increasing "processor word size" has been a major technique to improve the computation power. With the advent of multicore, the increment has been replaced with parallel computation. This level of parallelism is mostly discussed and utilized at the hardware applications of the problem.

Instruction-level parallelism In a program execution, there are a set of instructions that should be run. The instruction may be grouped in a way that re-ordering of the instructions in subgroups do not finally affect the program execution. Originally known as instruction-level parallelism, with the rise of modern processors, this technique has also improved with newer algorithms to adapt to multi-stage pipelines.

Data parallelism discusses mainly how data can be distributed across different computation units that run in parallel.

Task parallelism While data parallelism focuses on the distribution of data, task parallelism focuses on the distribution of execution in which the programmer tries to distribute different computing tasks across different units.

1.3 Theoretical Principles

This section tries to overview the principles discussed in [33] on the topic of programming with multiprocessors. The book emphasizes the discussions are for shared-memory multiprocessors [33, ch. 1]. It gives a well-presented flow of discussions on different theoretical aspects of multicore programming challenges. Briefly, we just mention one or two of the more related discussions.

1.3.1 Mutual Exclusion

Lock-based solutions

The standard way to approach the mutual exclusion problem is through a `Lock` object. A good `Lock` algorithm should satisfy the following properties [33, ch. 2, p. 24].

1. **Mutual Exclusion**: Critical sections of different threads do not overlap.
2. **Freedom from Deadlock**: If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock.
3. **Freedom from Starvation**: Every thread that attempts to acquire the lock eventually succeeds.

	LockOne	LockTwo	Peterson
Mutual Exclusion	✓	✓	✓
Deadlock-freedom	×	×	✓
Starvation-freedom	×	×	✓

Table 1.1: 2-Thread Lock-based solution comparison

Based on these properties, first 2-Thread [33, ch. 2.3] solutions are presented including LockOne, LockTwo, and Peterson Lock summarized in Table 1.1.

Filter lock is aims at a general n-thread solution [33, ch. 2.4]. It is a generalization of Peterson lock for n threads. Lamport's Bakery algorithm is discussed in [33, ch. 2.6] that also takes into consideration of *fairness* concept.

Fairness

Although starvation-freedom property ensures that every thread acquires the lock before entering the critical section, it makes no guarantee how long it may take to acquire a lock. In an ideal situation, if A tries to acquire the lock before B, then A should actually acquire the lock before B. Acquiring lock can be divided in two sections:

Doorway section with execution interval D_A consists of a bounded number of steps.

Waiting section with execution interval W_A may take an unbounded number of steps.

With this definition, the doorway section is ensured to finish in a bounded number of steps; called *bounded wait-free* property. Accordingly, the concept of fairness is defined as the requirement that if A finishes its doorways section before B starts its doorway section, then A cannot be taken over by B [33, ch. 2.5].

1.3.2 Concurrent Objects

The authors discuss the notions of "correctness" and "progress" in [33, ch. 3].

The first property that is observed from concurrent objects is that it will be easier to reason about their behavior and correctness if we can somehow map their concurrent executions to sequential ones and limit the reasoning to these sequential instructions. This is why method-level locking makes the reasoning easier while not desired with respect to more fine-grained approaches that provide more specific locking techniques [33, ch. 3, p. 47].

The gist idea is that sequential specification based on preconditions and postconditions is useful when a single thread manipulates a collection of objects. However, in the case that multiple threads can concurrently execute several methods of one object, the sequential order of behavior cannot be specified [33,

	Quiescent	Sequential	Linearizability
Nonblocking	✓	✓	✓
Compositional	✓	×	✓
Multicore	×	×	✓

Table 1.2: Consistency Properties

ch. 3, p.49]. With this assumption, a method may encounter an incomplete state of the object during its execution since another method is being concurrently executed through another thread.

Correctness

Regarding this concurrent behavior of concurrent objects, the following concepts are discussed:

Quiescent Consistency informally states that when an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed calls [33, ch. 3.3].

Sequential Consistency requires that method calls act as if they occurred in a sequential order consistent with program order. That is, in any concurrent execution, there is a way to order the method calls sequentially so that they (1) are consistent with program order, and (2) meet the object's sequential specification.

Linearizability . Each method call should appear to take effect instantaneously at some moment between its invocation and response (Principle 3.5.1); i.e. the real-time behavior of method calls must be preserved. This property is called linearizability. Every linearizable execution is sequentially consistent, but not vice versa.

Table 1.2 summarizes the discussed consistency properties in terms of "blocking" and "compositionality".

Progress

In this part of the discussion in [33, ch. 3.7], the authors only provide some progress condition definitions:

Wait-free method is some method that guarantees that every call finishes its execution in a finite number of steps. It is *bounded wait-free* if there is a bound on the number of steps a method call can take. An object is wait-free if its methods are wait-free. This is a progress property [33, ch. 3.7].

Population-oblivious method is a wait-free one whose performance does not depend on the number of active threads [33, ch. 3.7].

Lock-free method is one that guarantees that infinitely often *some* method finishes in a finite number of steps. Any wait-free method is also lock-free, but not vice versa [33, ch. 3.7].

Dependent progress condition. Progress occurs only if the underlying platform provides certain guarantees. Deadlock-free and starvation-free properties are examples.

A method call executes *in isolation* if no other threads take steps [33, ch. 3.7].

Obstruction-free method is one that from any point after which it executes in isolation, it finishes in a finite number of steps. This is a nonblocking progress condition [33, ch. 3.7]. A lock-free algorithm is obstruction-free, but not vice versa.

Chapter 2

Concurrent Programming Paradigms

2.1 Actor Model

Agha, in [22], introduces "Actors" as an inherently concurrent programming model. According to [36], in the Actor model, systems comprise of concurrent and autonomous entities called *actors* and *messages*. Actors communicate by sending asynchronous messages to other actors for which they should be aware of the destination actor *name* (mailbox). Each actor in response to the receiving messages can display different behavior as [22] proposes:

1. Send a finite set of messages to other known actors;
2. Create a a finite set of new actors; and
3. Define how it will behave in relation to the next incoming messages

To realize the actor model characteristics, [22, 36] propose that each actor semantics should provide:

Encapsulation In this model, there are basically two concerns:

1. State Encapsulation: An actor cannot directly access the internal state of another actor. An actor may affect other actors' internal state through messages that it sends to them [22, 36].
2. Safe Messaging: As [36] mentions, there is no shared state between actors. Therefore, message passing should be done through call-by-value semantics.

Fair Scheduling [36] proposes the notion of *fairness* in actor model meaning that each message is eventually delivered to its destination actor. This infers that no actor can be starved for ever.

Location Transparency As described in [22, 36], actors communicate through their mail box address. Thus the actor's name should not be dependent on its physical address. This also affects the mobility of actors in case of location transparency.

Transparent Migrations This property falls into two types of mobility: strong and weak [22]. Strong mobility refers to the possibility of migrating both execution state and code while weak mobility refers to the movement of the actor code.

In actor model, there is an extensive usage of "pattern matching" for messages that are entrant to the mail box of an actor. This is also important in case of an internal representation of actor. When a programmer is writing an actor, she needs to specify the messages to which the actor will respond. Thus, actor model supporters also tend to take much advantage of "functional programming" concepts [26, 23, 48, 30].

[36] provides a good comparison of the frameworks implemented for JVM platform.

2.2 Software Transactional Memory

2.2.1 Overview

Shavit and Touitou [46] introduce software transactional memory as a novel method for supporting transactional programming of synchronized operations. They use STM to provide a general concurrent method for translating sequential object implementations to non-blocking ones. The most important property of STM is its *non-blocking* nature as opposed to lock-based programming models.

According to [46], the challenge of STM is to eliminate the deadlocks for transactions for which they propose the "helping" methodology. In helping methodology, each transaction tries to help the owner transaction complete its works to release the location others are waiting for.

Technically, a *transaction* is a finite sequence of local and shared memory machine instructions [46]. Transactions are either *read-transactional* or *write-transactional*; in the former, there is a read operation from a shared location while in the latter there is a write operation into a shared location [46]. Each transaction may either *fail* or *success*; in the case of success, the changes are atomically visible to other processes. Additionally, transactions are *isolated* and *atomic*; the former means that each transactions runs as if others are suspended while it runs and the latter means that they are all-or-nothing operations [21]. Thus, they give the programmer the illusion of a serial execution.

Accordingly, [46] proposes that a software transactional memory is *shared object* that behaves like a memory that supports multiple changes to its addresses by means of transactions. A transaction is a thread of control that applies a finite sequence of primitive operations to memory. This is why software transactional memory is believed to bring in the concept of database transactions on

data into object-oriented paradigm. In this model, each object provides a set of *primitive operations* only through which the underlying object (shared memory) can be manipulated. [46] introduces *wait-free*, *non-blocking*, and *swap-tolerant* transactions as types of STM.

To reason about STM, [46] introduces "real-time order" of processes in a system; i.e. operation `A` precedes operation `B` if `A`'s response occurs before `B`'s response. With this definition, two operations are *concurrent* if they are unrelated according to real-time order. A sequence of invocations and responses is called *history*. A *sequential history* is a history in which each invocation is followed by its corresponding response. The correctness requirement of STM is based on *linearizability* of a concurrent history being equivalent to some legal sequential history that is consistent to some real-time order induced by the concurrent history.

2.2.2 Pros and Cons

In an evaluation, in [24], they propose that the promise of STM may likely be undermined by its overheads and workload applications. They believe that STM introduces new issues into programming models:

Interaction with non-transactional codes that includes accessing data that is outside the transaction.

Exceptions and serializability that is a question on how to handle exceptions and propagate them within the context of transactions to outside.

Interaction with code that cannot be transactionalized that can be a requirement in specific circumstances.

Livelock that is a property to ensure that all transactions are making progress through time.

Based on these issues, they believe that STM has not yet matured and will not by itself solve the challenges of parallel programming, however, they are useful in the context of *concurrent data structures*. They also argue that STM introduces nontrivial drawbacks with respect to performance and programming semantics:

Overheads. STM approach inherently introduces overhead that is in direct opposition with performance.

Semantics that falls into:

1. Weak Atomicity that occurs when there is no detection of conflicts based on transactional and non-transactional regions.
2. Privatization: Being in the context of a transaction or being used in private by the underlying object can make the design decision more complex.

3. Memory Reclamation: Some designs restrict the use of memory accessing constructs directly that becomes a burden.

Legacy Binaries. STMs that take advantage of code instrumentation on the original code face difficulties to be used with legacy programs for which the original code is not available.

They conclude that there are a lot of challenges in STM field including lowering the overheads of STMs and the elimination of unnecessary read and write operations. Finally, they believe that TM programming model introduces complexities that limit the expected productivity gains and reduces the tendency towards its usage.

In another research, [21], they provide some implementation and evaluation in favor of STM. They start by proposing the concept and requirement for languages to introduce new constructs to support software transactional memory. They reason that unlike coarse-grained locking, STM provides fine-grained locking mechanisms and boost scalability through:

1. Allowing concurrent read operations on the same variable since basic mutual exclusion locks do not permit concurrent readers.

2. Allowing concurrent read and write operations on *disjoint* variables that may comprise two or more active threads of control.

However, they also mention that in modular software engineering fine-grained locks are not feasible when modules are composed together. They conclude that yet STM is not the best solution in parallel programming but with the help of other technologies such as task decomposition or mapping, it has taken concrete step in making parallel programming easier.

In another research, [39], they provide a good overview of design and implementation issues in STM and they introduce RSTM. They propose that major *design issues* include:

Metadata Organization that is mainly about how to maintain information about acquired objects in a transactional system that is referred to as *transactional metadata*. They compare two approaches that are *per-object metadata* as used in DSTM [32] and *per-transaction metadata* as used in OSTM [29]. RSTM uses per-object metadata.

Conflict Detection. There are two approaches that existing STMs use for conflict detection. In *eager* approach, the objects are acquired at the soonest open time (DSTM). However, in *lazy* approach it is delayed until the commit time of the object (OSTM). RSTM supports both. Each approach has its own advantages and disadvantages. With eager detection there could be prevention of unnecessary future operations in close future while also taking the chance to ruin related work that is already performed. And, obviously, lazy detection may have the opposite properties.

Contention Management introduces the concept of competing objects and their circular dependencies that may lead to deadlocks. As proposed in [46], *helping* is used, however, it may also result in heavy interconnect contention and high cache miss ratio. Thus, some use the concept of obstruction freedom [31] that prevents livelock and starvation; RSTM is obstruction free.

Validating Readers talks about the inconsistency that may be caused in lazy approaches when some private data objects are created to be used later in write operations and in case of abortion they may still expose inconsistent data to other transactions.

Memory Management. There is always need for memory reclamation in software transactional memory. A general purpose garbage collector helps, however, in languages such as C++ in which direct access to memory is present the reclamation policy should be decided by the programmer.

Also, they summarize that what could be the source of *overhead* in STM implementations:

Bookkeeping talks about the maintenance of the objects to facilitate the fetch and write by objects.

Memory Management for metadata and private date objects that cooperate in the implementation of STM principles.

Conflict Resolution. There should be some speculation over the avoidance and resolution along with the operations required for helping concept.

Validation. As discussed before validating reader may introduce new and costly operations for the STM.

Copying. When to-be-written data object are created the copying process may not be so costly, however, in the case of large objects that only require a small change this could become a heavy cost.

They conclude with strengthening features of RSTM over other implementations, however, as this is an optimization approach, it also implies that STM has much of overhead and challenges that should be faced with.

2.3 Data Flow Programming

Data flow programming [3], that can be considered a branch of flow-based programming [5], focuses on facilitating parallel computation through division, conquerence and merging of a program based on the data it processes. The division occurs such that parallel flows of computation can be managed and directed after which the results are again merged to yield the final output of the program. Much of the decisions in the phases are upon the programmer but not the synchronization and parallelization parts.

2.3.1 MapReduce

Based on data flow programming, in [27], they propose MapReduce that is basically based on one *map* and one *reduce* function. It is originally proposed and extensively used at Google.

The user writes two function. "Map" takes an input pair and produces a set of intermediate key/value pairs. When the computation is done, the library groups together all intermediate values associated with the same intermediate key and passes them to reduce function. "Reduce" function accepts an intermediate key and all the grouped associated values and it is supposed to merge the values to possibly form a smaller set of values. For instance, in [25], they use MapReduce in Machine Learning purposes.

[43] introduces Phoenix as an implementation of MapReduce on shared memory. They do an evaluation of MapReduce feasibility on multicore and multiprocessor systems. They start by arguing that the main feature of MapReduce approach is its *simplicity*. The programmer thinks about the functionality rather than parallelization concerns. However, they propose the question that "how widely applicable is the MapReduce model" is not studied in the research. They conclude that MapReduce is a useful programming and concurrency management approach for shared-memory systems that are heavily data-centric.

Chapter 3

Languages and Libraries

3.1 Actor Model

A number of languages have been developed that either specifically or as-a-part support actor programming model:

3.1.1 Languages

Erlang [23] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [26]. Erlang has a process-based model of concurrency. Concurrency is explicit and the user can precisely control which computations are performed sequentially and which are performed in parallel. Message passing between processes is asynchronous, that is, the sending process continues as soon as a message has been sent.

SALSA [48] is an actor-based language for mobile and Internet computing that provides three significant mechanisms based on actor model: token-passing continuations, join continuations, and first-class continuations. Essentially, a *continuation* is an executable behavior that is used when an actor completes its asynchronous response to a message.

In token-passing continuation, a concept of "customer" actor is defined that is part of a message sent to an actor. When the actor completes processing the message, it will pass by the token to the customer actor that will continue the processing the result.

In join continuation, a customer actor receives an array with the tokens returned by multiple actors once they have all finished processing their messages.

First-class continuation is a smart way to delegate computation to a third-party independent of the message processing context. In this type of

continuation, a continuation is assigned to an object to be used when some processing is finished.

E Language [4] is a lambda-language such as Smalltalk that mainly comprises of the language and ELib; ELib provides is a pure Java library that provides distributed programming concepts. It provides inter-process messaging with security and encryption, event-loop concurrency and deadlock-free distributed object computing. E runs on JVM.

Ptolemy [37] is an actor-oriented open architecture and platform that is used to design, model and simulate embedded software. Their approach is hardware software co-design. It provides a platform framework along with a set of tools.

Axum [40] is a language that builds upon the architecture of the Web and principles of isolation, actors, and message-passing to increase application safety, responsiveness, scalability, and developer productivity.

3.1.2 Libraries

Another stream of effort has been into developing libraries and frameworks for existing languages:

Java: Scala Actors Library [30, 19] : Scala is a hybrid object-oriented and functional programming language inspired by Java in which a famous Actors library exists that mimics the Erlang implementation. The most important concept introduced in [30, 19] is that Scala Actors unifies *thread-based* and *event-based* programming model to fill the gap for concurrency programming. In this model, an actor is a thread that can also react to events that come from other actors; i.e. it provides both "receive" and "react" features. "React" is a compliment to "receive" that is proposed in the original actor model. Listing 1 shows a sample how to compute Fibonacci value with actors in Scala.

Java: Kilim [47] is a framework used to create robust and massively concurrent actor systems in Java. It uses a bytecode post-processor called Weaver [47]. Kilim takes advantage of code annotations on bytecode to provide the facility.

Java: ActorFoundry [36] is a Java framework that brings the actor model implementation to the developer through the use of code annotations. It provides fair scheduling, actor mobility, and safe messaging out of the actor semantics.

Java: Jetlang [44] provides a high performance Java threading library. The library is based upon Retlang [45] for C#. The library is a complement to the `java.util.concurrent` package introduced in 1.5 and should be used for message based concurrency similar to event based actors in Scala.

The library does not provide remote messaging capabilities. It is designed specifically for high performance in-memory messaging.

Java: JavAct [35] platform is based on the actor model and open implementation principles. With JavaAct, users write high-level Java standard code without considering low-level mechanisms such as threads, synchronization, RMI, Corba, etc. JavaAct has been designed in order to be minimal in terms of code and so maintainable at low cost, portable, and easy-to-use for Java junior programmers who know a little of actors. It does not need any preprocessing and can be used with any standard Java toolkit.[1]

Java: AJ [50] is a software system for writing distributed programs in Java, and is based on the actor model. In AJ, an actor is an extension of an object: where objects communicate by calling each others methods, actors communicate by sending asynchronous messages to each other. AJ provides the messaging layers that allow actors to communicate with each other, no matter if the actors are on the same computer, or scattered across a network. AJ design goal has been experimentation in terms of clarity and modularity rather than performance.

Java: Jsasb. The project homepage has been removed.

C++: Act++ [38] is a class library for concurrent programming in C++ using actors model. Theron [18] is a lightweight, portable C++ class library for developing parallel applications. It implements a simple service-oriented model of concurrent processing based on the Actor Model.

Listing 1: Simple Fibonacci in Scala

```
1  import scala.actors._
2  import Actor._
3  object FibonacciActorObject {
4
5    def fibonacci(n : Int) : Long = {
6      if (n == 0 || n == 1)
7        return 1
8      var f1 = fibonacci(n - 1)
9      var f2 = fibonacci(n - 2)
10     f1 + f2
11   }
12
13   def main (args: Array[String]) = {
14     val fibComputer = actor {
15       var continue = true
16       while (continue) {
17         receive {
18           case (caller: Actor, n : Int) => actor {
19             caller ! (n, fibonacci(n))
20           }
21           case "quit" => continue = false
22         }
23       }
24     }
25
```

[1] The documentation seems to be only in French.

```
26      fibComputer ! (self, 40)
27      fibComputer ! (self, 40)
28
29      for (i <- 1 to 2) {
30        receive {
31          case (n, result) => println(result)
32        }
33      }
34
35      fibComputer ! "quit"
36    }
37
38  }
```

Other implementations include Broadway, and Thal for C/C++, Stackless for Python, MS Asynchronous Agents Library and Retlang [45] for .NET, Stage for Ruby, and Actalk for Smalltalk.

3.2 Software Transtactional Memory

Mutiverse [16] is a Java based STM implementation that aims at seamless integration in the language and language independence in the form a framework. Listing 2 displays a sample script how to control the modifications of a bank account object using Multiverse.

Clojure [34, 2] is a dynamic programming language as a dialect of LISP that target the Java Virtual Machine. Clojure provides concurrent programming constructs based on STM concepts.

JVSTM [13] , another Java based STM library, introduces two core concepts as "transactions" and "versioned boxes". The goal is to allow transaction programming at the programming language level, independent of an external transaction manager.

Intel C++ STM Compiler [10] is a C++ platform that provides STM concepts of isolation and atomic executions in a C++ compiler. Transactional Locking II (TL2) [28] is an STM algorithm based on a combination of commit-time locking and a novel global version-clock based validation technique.

Listing 2: Mutilverse Annotations

```
1  import org.multiverse.annotations.TransactionalMethod;
2  import org.multiverse.annotations.TransactionalObject;
3
4  @TransactionalObject
5  public class Account{
6
7      ...
8
9      @TransactionalMethod
10     public static void transfer(Account from, Account to, int amount) {
11         from.setBalance(from.getBalance()-amount);
12         to.setBalance(to.getBalance()+amount);
```

	Java	C++	Python	C#
Actors	Scala, Erlang, SALSA, Axum, Ptolemy, Kilim, ActorFoundry, Jetlang, JavAct, AJ	Theron, Act++	Stackless, V3.2+	Retlang
STM	Multiverse, Clojure, DSTM2 (Fortress), Deuce, JVSTM	TL2 (Sun), Intel C++ STM	Durus	SXM
Data Flow	Java 7+ JSR 166y Fork/Join, Hadoop	OpenMP, TBoost.STM, Hadoop Streaming	Disco (Nokia)	Dryad, Hadoop Streaming

Figure 3.1: Summary of different libraries and languages for different paradigms

```
13      }
14 }
```

3.3 Data Flow Programming

Java 7 [11] is supposed to provide language-level features for data flow programming as proposed in JSR 166y [12] including fork/join.

Apache Hadoop project [7] develops software for reliable, scalable, distributed computing proposing several frameworks among which is MapReduce [8] that is a framework for distributed processing of large data sets on computer clusters.

Figure 3.1 summarizes some of the most current works for languages Java, C++, Python, and C# in the three discussed paradigms.

3.4 All-in-one solutions

There are also some interesting languages that have tried to provide all the different approaches in one library or language including Haskell and GPars.

3.4.1 Haskell

Haskell [9] is an advanced purely functional programming language which, among others, seems to have implemented more than one approach towards providing *built-in* multicore programming features. These , according to [20, 41], include a primitive type `MVar` that is a primitive in the language to provide *asynchronous channels*, an ability to spawn new concurrent threads via `forkIO` primitive, and

software transactional memory through `TVars` and `retry` and `orElse` primitives. Additionally, it provides an actor model implementation that is internally dependent on Haskell's STM implementation.

3.4.2 GPars

GPars [6] is a Groovy/Java library that aims to provide easier techniques to programmers to handle concurrency tasks. The main areas include concurrent collection processing and fork/join abstraction, asynchronous operations, Actor model, data flow concurrency data structures, agent-oriented programming primitives and a planned[2] STM implementation.

3.5 Hybrid solutions

In another perspective, some have started to create a blend of different approaches to provide solutions and techniques for multicore programming.

3.5.1 Akka

Actor model is based on asynchronous message passing. That is why Scala, for instance, favors the use of *immutable objects* in message passing mechanisms. However, immutable objects make it so hard to actually have a *shared state* so that multiple threads and objects can manipulate.

Software transactional memory (STM) makes it simple to synchronize shared state manipulation with object transactions. It also has the advantage that transactions are *composable*. However, asynchronous communication is complicated with STM.

Akka [1] is a platform that provides a simple way to develop concurrent and fault-tolerant applications using a mixture of Actors and STM. It proposes the notion of **Transactors** that are Actors which support STM in their behavior, so, they provide *transactional*, *compositional*, *asynchronous*, *event-based* actors. Akka already takes advantage of another library Multiverse [16] that is an STM implementation for Java, Groovy, and Scala. The platform is still under development but working.

3.5.2 MPI

The MPI [14] standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++ and Fortran are defined. This standard has an assumption that it will be used on

[2] As of the date of this report

a *distributed memory* architecture. On the contrary, the multicore architecture is commonly assumed to be based on a *shared memory* architecture. The difference should be noticed in this line for the applications.

Chapter 4

Future works

In this report, we tried to gather a collection of related work in the domain of multicore programming and object-oriented languages. The efforts usually break down into three paradigms of Actor model, software transactional memory and data flow programming. Some created a new language or a new flavor of some other language to propose new features. Others built libraries on top of a language with the same goal. We studied several languages and libraries in different languages such as Java, C++, Python, and C#. The amount of work gone through to provide multicore programming techniques in object-oriented languages implies the challenges and interesting problems that are yet to be solved in this field.

As a general vision, considering the notion of concurrent objects, we are interested in methods to deploy concurrent objects onto multicore technology. In this regard, we will use Creol as an active object-oriented language for the modeling layer. We will convert a Creol program to its equivalent Java program that will take advantage of the new concurrency features in Java 1.5+. The translation will be performed in a way that will simulate the asynchronous message passing nature of Creol in Java using the notion of *future* values. Additionally, we will try to use other locking mechanisms in Java to provide the notion of *processor release points* in Creol. We are looking to create a new approach towards how to mix and merge different levels of abstraction to deploy concurrent objects onto multicore. This will ease the task of programming in the sense that still the most important job of a programmer will be how to code correctly towards functionality; and, meanwhile others can take care of aligning the functionality onto multicore. Future reports and works will include results and more elaboration on this project.

Appendix A

Terminology

Computability is the concept of figuring out what can be computed in a an asynchronous concurrent environment [33, ch 1., p. 2].

Program Correctness for which to verify we need to be able to specify what a program does [33, ch. 1, p 2].

Safety property states that some bad thing never happens [33, ch. 1, p 2].

Liveness property that a particular good thing will happen [33, ch. 1, p 2].

Thread A *thread* is a state machine and its state transitions are called *events*. Events are instantaneous [33, ch. 2, p 22].

Event Precedence One event a precedes another event b, written as $a \rightarrow b$, if a occurs earlier than b. Event precedence is a *total order* on events [33, ch. 2, p 22].

Event Interval If events a_0 and a_1 exist such that $a_0 \rightarrow a_1$, $interval(a_0, a_1)$ is the time between [33, ch. 2, p 22]. Having $I_A = (a_0, a_1)$ and $I_B = (b_0, b_1)$, then:

$$I_A \rightarrow I_B \Longleftrightarrow a_1 \rightarrow b_0$$

Concurrent Events The $\rightarrow$ relation is a partial order on intervals. Intervals that are unrelated by $\rightarrow$ are said to be concurrent [33, ch. 2, p 22].

Critical Section is a block of code that be executed by only one thread at a time [33, ch. 2, p 22]. This property is called **mutual exclusion**.

Object is a container for data that is usually created from a class [33, ch. 3, p.48]. Each object has a state at any time that is an indication what internal values are.

Precondition denotes the state that an object has before invoking some method.

Postcondition denotes the state that an object has after invoking some method.

Method Call is the interval that starts with an *invocation event* and ends with a *response event*.

Pending Method is a method whose invocation event occurred but not its response event.

Quiescent Object has no pending method calls [33, ch. 3.3].

Program order is the order in which a single thread issues method calls [33, ch. 3.4]. Method calls by different threads are unrelated by program order.

Bibliography

[1] Akka Project. http://akkasource.org/.

[2] Clojure concurrent programming. http://clojure.org/concurrent_programming.

[3] *Data Flow Programming.* http://en.wikipedia.org/wiki/Dataflow_programming.

[4] *The E Language.* http://erights.org/.

[5] *Flow-based Programming.* http://en.wikipedia.org/wiki/Flow-based_programming.

[6] GPars. http://gpars.codehaus.org/.

[7] Hadoop. http://hadoop.apache.org/.

[8] Hadoop MapReduce. http://hadoop.apache.org/mapreduce/.

[9] Haskell. http://www.haskell.org/.

[10] Intel C++ STM Compiler. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.

[11] Java 7. https://jdk7.dev.java.net/.

[12] JSR 166: Java concurrency utilities. http://www.jcp.org/jsr/detail/166.jsp.

[13] JVSTM. http://web.ist.utl.pt/~joao.cachopo/jvstm/.

[14] MPI. http://www.mpi-forum.org/.

[15] Multi-core processor - wikipedia. http://en.wikipedia.org/wiki/Multi-core_processor.

[16] Multiverse: Software Transactional Memory for Java. http://multiverse.codehaus.org/overview.html.

[17] Parallel computing - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Parallel_programming.

[18] Theron. `http://theron.ashtonmason.net/index.php`.

[19] *Coordination Models and Languages*, chapter Actors That Unify Threads and Events. Springer Berlin / Heidelberg, 2007.

[20] *Multicore Haskell Now!*, DEFUN 2009. `http://donsbot.wordpress.com/2009/09/05/defun-2009-multicore-programming-in-haskell-now/`.

[21] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4:24–33, December 2006.

[22] Gul A. Agha. *Actors: a model of concurrent computation in distributed systems*. PhD thesis, MIT, 1986.

[23] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[24] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51:40–46, November 2008.

[25] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288. MIT Press, 2006.

[26] Fábio Corrêa. Actors in a new "highly parallel" world. In *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, WUP '09, pages 21–24. ACM, 2009.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[28] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin / Heidelberg, 2006.

[29] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[30] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202 – 220, 2009.

[31] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Distributed Computing Systems*, pages 522–529. IEEE, May 2003.

[32] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101. ACM, 2003.

[33] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[34] Rich Hicky. Clojure. `http://clojure.org/`.

[35] S. Rougemaille J.-P. Arcangeli, F. Migeon. *JavaAct*. `http://www.javact.org/JavAct.html`.

[36] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[37] Edward A. Lee. Overview of the Ptolemy Project. Technical report, University of California, Berkeley, 2003.

[38] Loch M., Mukherji M., and Lavendar G. Act++ 2.0: A class library for concurrent programming in C++ using actors. *Journal of Object-Oriented Programming*, 1993.

[39] Vriendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Esienstat, William N. Scherer III, and Michael L. Scot. Lowering the overhead of nonblocking software transactional memory. In *Proceedings ACM SIGPLAN*, TRANSACT '01. ACM, 2006.

[40] Microsoft Corporation. *Axum Programming Language*. `http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx`.

[41] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell*, chapter (24) Concurrent and multicore programming. O'Reilly, 2008.

[42] Michael Philippsen. A survey of concurrent object-oriented languages. In *Concurrency: Practice and Experience*, pages 917–980. John Wiley, 2000.

[43] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[44] Mike Rettig. *Jetlang*. `http://code.google.com/p/jetlang/`.

[45] Mike Rettig and Graham Nash. *Retlang*. `http://code.google.com/p/retlang/`.

[46] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997.

[47] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP 2008 Object-Oriented Programming*, volume 5142, pages 104–128. Springer Berlin / Heidelberg, 2008.

[48] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36:20–34, December 2001.

[49] Wikipedia. Concurrent programming languages. `http://en.wikipedia.org/wiki/Concurrent_programming#Concurrent_programming_languages`.

[50] Wililla Zwicky. *AJ: A system for building actors with Java*. PhD thesis, University of Illinois at Urbana-Champaign, 2008.