On Concurrent Execution of Object–Oriented Programs

Technical Report THD-BS-1993-02

Thilo Kielmann

April 1993

Department of Computer Science (FB 20)
University of Darmstadt
Alexanderstr. 10
D–6100 Darmstadt
Germany

Tel.: (49)–6151–163606
Fax: (49)–6151–165410
Email: kielmann@isa.informatik.th-darmstadt.de

Abstract

In this paper, we introduce object–oriented programs as sets of communicating objects. We investigate possibilities for their concurrent execution, starting with a review of existing concurrent, object–oriented systems. Most of these systems introduce new programming–language features, forcing a programmer to control concurrency by hand. In contrast, we prefer automatic parallelization of programs. For this purpose, we introduce a concurrent, object–based execution model.

1 Introduction

The aim of this work is to take programs written in an object–oriented language and execute them in parallel on a set of processors. The object–oriented language will thus play the role of an "executable specification" without any implication of an execution mode (sequential or parallel).

As it is widely accepted, significant improvements of computing power will, in the near future, only be achieved by the use of multi–processor architectures in which numerous processing elements share the applications' work cooperatively.

It is as well accepted that object–orientation plays the role of the leading technology in programming [11] as well as software engineering [15]. As it will be shown later, object–orientation is also a well–suited paradigm for the construction of programs to be executed concurrently.

1.1 Basic terminology

We will now introduce the terminology in which concurrent execution of object–oriented programs shall be treated later.

1. An *object–oriented program* consists of a collection of *classes*.
2. A *class* describes the behaviour common to its instances, the *objects*.
3. Every *object* holds its private memory in which the objects' state is stored.
4. Objects communicate by exchanging *messages*.
5. Every *message* consists of the identification of a routine to be executed upon receipt and an optional list of parameters. Every routine to be executed must be chosen from the *class interface*.
6. The *class interface* provides an exactly defined signature, realizing an *abstract data type* [11].

An object–oriented program can thus be seen as a set of communicating objects. In the case of sequential execution, there is a fixed execution order for all messages to be exchanged, which is directly derived from the statement order in the program source. So there is exactly *one* thread of control in which objects communicate. All communication is performed synchronously while the thread of control is passed together with each message sent from one object to another.

This situation bears a large amount of potential for concurrency which may be exploited through asynchronous communication. In this case, messages will be sent without the accompanying thread of control. As a result, the sender may continue his computations concurrently with the receiver of the message. This leads to a model in which a program will be executed by several concurrent threads of control.

2 Current state of concurrent object–oriented systems

We will now give a brief overview of currently existing concurrent object–oriented systems. In contrast to the goal of this work, all these systems use explicit programming language elements forcing the programmer to express parallelism manually. The reason behind this design decision is the interest in minimizing the application execution times. Protagonists of explicit programming language elements expect the human programmer to be the best code optimizer in the sense of exploiting as much as possible of the parallelism available in an applications' source.

2.1 Object–oriented languages with explicit parallelism

We will now briefly discuss the basic concepts for programming language elements expressing parallelism. We also mention the most prominent representatives which implement these concepts.

2.1.1 Control–Parallelism

One approach to formulate concurrency in a program is to have specific instructions to be executed. Depending on the modelling of the underlying hardware architecture[1], there are two different flavours.

Distributed–memory model In this model, there are several processors each of which has its own (private) memory. Communication between processors may only be performed by exchanging messages.

As a consequence, languages supporting this model offer constructs for sending and receiving messages. Variations occur in synchronous vs. asynchronous messages and in different priorities. Well-known languages supporting this model are ABCL/1 [18] and POOL-T [2].

Shared–memory model Here, the cooperating processors may (or may not) have private memory — communication is performed via a shared (or "global") memory area. Because the shared–memory model comes very close to the single memory of a uniprocessor system, it is much easier to program than the distributed–memory model. There are at least two ways to implement a shared memory model in an object–oriented language:

[1]Because shared–memory architectures may be simulated by distributed–memory machines and vice versa, it is sufficient to discuss only the logical view which is presented by a language.

Remote procedure call
Here, classical procedure calls are simulated while hiding the different processors from the programmer. Remote procedure calls may be performed synchronously (the caller waits for the answer) or asynchronously (without waiting). An example is the HYBRID language [13].

Monitors
The monitor concept, primarily introduced for operating–system purposes [7] may also be used to synchronize concurrent access to shared (object) memory by gathering requests in a queue and allowing only one at a time to be executed. An object–oriented language using monitors is pSATHER [10].

2.1.2 Data–Parallelism

An alternative way to introduce parallelism is based on the notion of performing the same operations on a large set of data. Because computations may be performed effectively when data can be accessed in local momory, data–parallel languages provide declaration–like constructs to distribute the data to be processed over the available processors. Therefore, the compiler has to insert program code to implement the inter–processor communication. pSATHER [10], for example, provides such language constructs.

2.2 Introducing parallelism into existing (sequential) object–oriented languages

Instead of inventing completely new programming languages, one may also extend existing (sequential) object–oriented languages by features realizing parallelism. One main advantage of this approach is the familiarity with the laguage's syntax which avoids time- and cost-expensive learning phases for the programmers.

2.2.1 Language extensions

One possibility is to introduce new language features into an existing language. It is hard to decide if this approach leads to completely new languages like the ones discussed in the previous section or if it simply leads to derivatives of the existing languages. pSATHER, as discussed above, DISTRIBUTED EIFFEL [6], CONCURRENT SMALLTALK [17], and μC++ [4] are examples for this approach.

2.2.2 Runtime support

Another possibility is to provide a set of specialized classes the use of which allow concurrent execution of their code. There are a lot of runtime libraries, supporting several of the flavours mentioned above. Examples may be found in [14] (control parallelism support for C++) and in [8] (data parallelism support for Eiffel).

The major drawback of this approach is its lack of generality. The application programmer still has to write code which explicitly uses the libraries' classes. Moreover, there is no solution for problems which cannot be solved by the classes of the library. Finally, these libraries are typically hand–coded, so their implementation is usually optimized for a specific hardware architecture.

3 Automatic parallelization approaches

Automatic parallelization is the (fully automated) transformation of programs from sequential into parallel versions. Initially, automatic parallelization was intended to reuse old "dusty deck" programs for new, parallel computers. But, as it is seen today, automatic parallelization becomes much more important.

As discussed in the previous sections, hand-written parallel programs suffer from their lack of generality and portability. Additionally, the programmers carry the burden of understanding and exploiting specific hardware architectures. Thus, automatic parallelization seems to become the most elegant method of parallel program development. As also stated in [5], this technique provides portability of parallel programs as well as a sound programming methodology for computer architectures of the future. This arises from the separation of concerns between application–specific code and inter–processor communication. The programmer's task remains the solution of the application's problems, while the compiler handles the technical issues of envolving the processing ressources. Similar to the traditional (sequential) compiler technology, all optimizations may thus be performed by machine–specific "compiler backends", in which all machine–dependent information shall be concentrated.

3.1 Existing, parallelizing systems

Until today, there are a lot of compiler systems which allow parallel execution of formerly sequential programs. Besides systems which are specialized on functional or declarative programming languages, there are two efforts worth mentioning, because they may help in parallelizing object–oriented programs.

3.1.1 Parallelizing compilers for imperative languages

Primarily, large amounts of computational power has been needed for numerical problems. Therefore, most efforts on automatic parallelization has been spent on FORTRAN dialects. Well known examples are the PTRAN sytem [1] and Vienna Fortran [20].

Such parallelizing compilers are dealing with dependencies between variables involved in different loop iterations. The problem is to describe the sequential program using a data–dependency relation: A pair (I_1, I_2) of instructions belongs to this relation *iff* I_2 *must be executed after* I_1. A restructured statement order (by parallel execution) will be called to be *safe* if the data–dependency relation will be left intact. A classification of these dependencies may be found in [9]. An in–depth description of compiler techniques for parallelization based on data–dependency analysis can be found in [21].

There is a close relation between imperative and object–oriented languages because of which their parallelization efforts become relevant for object–oriented programs: Usually, each routine provided by a class's interface will be interpreted and executed like a routine or "procedure" of a program written in an imperative language. This allows the execution of an object–oriented program to be viewed as a series of nested calls to procedures, each of which is executed in an imperative manner.

3.1.2 Parallelizing compilers for object–oriented languages

Currently, there is one known system which automatically parallelizes programs written in a subset of the C++ language [16]. This approach restricts programs written in C++ to a subset without dynamical object creation. Such programs with a number of objects known at compile time are then translated into a set of programs, one for each processor involved. This is done by analyzing and numbering all routine calls and data transports of the sequential program, which are all known at compile time.

Although this approach draws significant restrictions on the language in order to achieve static program analysis, its basic ideas of representing a progam in a communication graph from which the parallel code will be derived, bears some potential to be exploited in a dynamical model. The next sections will outline such an approach on parallelizing object–oriented programs in their usual manner in which a dynamically growing and shrinking set of objects communicate in order to solve an application's task.

4 A concurrent, object–based execution model

We will now describe how a program written in an object–oriented language may be executed concurrently. The basic notion needed for this execution model is the concept of atomic actions.

The concept of atomic actions has been derived from the transaction model, which has primarily been introduced for maintaining consistency in databases. Recently, atomic actions, as well as transactions, became of interest as program structuring methods for distributed systems [12].

In principle, transactions are atomic units of execution which either succeed or fail without having any effect. Failing ("aborting") transactions are of minor interest in this execution model. So, we focus on atomic actions as units to be scheduled.

In our execution model, an object–oriented program is no longer seen as a set of nested procedure calls. Instead, it becomes a set of nested atomic actions. In the sequential case, message passing between objects is implemented by procedure calls. Here, it becomes a "real" exchange of messages which will be stored by the appropriate receiver. Hence a message becomes a request for execution of a distinct action. Assigned to each object is some kind of manager which receives incoming messages for this object and determines how their requests may be served. Actions ready for execution will be collected in a queue out of which the processors may fetch and execute them.

This execution model leaves two important problems unsolved:

1. It is not clear how the object managers shall decide when to schedule which actions without changing the program's semantics.

2. Important features for efficient parallel execution are the distributions of data and workload between the processors involved. In spite of this importance, this paper will not focus on that topic.

4.1 Synchronizing atomic actions on objects

We will now show how to solve the first of the two problems stated above. In order to do so, we rely on [3]. In this paper, the authors deal with the synchronization of transactions. Because recovery features are of minor interest in our execution model, we use their work while replacing transactions by simple atomic actions.

The authors of [3] introduce the notion of *commutativity* of transactions. They define that two transactions commute if they may be executed in arbitrary order (with respect to each other) without changing their semantics. As a result, commuting transactions may be executed concurrently.

For exploiting as much as possible of the available concurrency, objects will be viewed using so-called granularity graphs. Here, vertices denote objects (data items) and edges describe how objects are composed out of each other.

The transactions will be analyzed in order to collect the sets of edges and vertices which are affected by the transaction's execution (the so-called *affected sets*). The authors of [3] prove that two transactions commute iff their affected sets are not intersecting. As a result, a transaction may be scheduled for execution if its affected set does not intersect with any of the affected sets of all transactions currently beeing active.

4.2 Modelling concurrent execution by commuting atomic actions

The work described in the previous section bears the solution for the core problem. But for the purposes of automatic parallelization of object–oriented programs, it has to be augmented by the following topics:

- The objects in [3] consist of plain data items. Nested objects are solely treated as a special case. Instead, nested objects have to be the standard case, conforming to object–oriented programming. Thus, additional developments are necessary for synchronizing atomic actions on sets of possibly cyclically nested objects.
- The object managers in [3] are quite obscurely defined. In our environment, we need to localize them clearly: Because object managers must be generated as a result of the parallelizing compiler's activity, they shall become part of the parallelized application code. An obvious way to achieve this is to build a completely new set of classes out of the original source code. Each of the new classes joins the original code with the manager's code for this class. So, on runtime, every object becomes its own manager.

5 Future Work

With the execution model developed so far, we are able to design a first concept for a parallelizing tool for object–oriented programs. Besides the augmentations mentioned above, we have to clarify several notions mentioned only briefly in the preceeding sections. Additionally, we shall investigate the problems arising from programming language features not mentioned so far, namely inheritance and exception handling.

For validation purposes, we will have to direct our work to a particular programming language, which covers all features of object–oriented programming. Also, this language shall provide a sound and well–defined basis on which further work may be performed.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An Overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1st International Conference on Supercomputing*, number 297 in Lecture Notes in Computer Sciences, pages 194–211, 1987.

[2] Pierre America. POOL-T, a Parallel Object–Oriented Language. In Yonezawa and Tokoro [19], pages 199 – 220.

[3] B. R. Badrinath and Krithi Ramamritham. Synchronizing Transactions on Objects. *IEEE Transactions on Computers*, 37(5):541 – 547, 1988.

[4] Peter A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software: Practice and Experience*, 22(2):137–172, 1992.

[5] Paul Feautrier. Automatic Parallelization Techniques. In *Tutorial notes from Second Joint International Conference on Vector and Parallel Processing (CONPAR 92 – VAPP V)*, Lyon, France, 1992.

[6] L. Gunaseelan and Jr. Richard J. LeBlanc. Distributed Eiffel: A Language for Programming Multi-Granular Distributed Objects on the Clouds Operating System. Technical Report GIT-CC-91/50, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, 1991.

[7] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549 – 557, 1974.

[8] J.-M. Jézéquel, F. Bergheul, and F. André. Programming Massively Parallel Architectures with Sequential Object Oriented Languages. In D. Etiemble and J.-C. Syre, editors, *PARLE '92, Parallel Architectures and Languages Europe*, number 605 in Lecture Notes in Computer Science, pages 329 – 344, Paris, France, 1992. Springer.

[9] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, 1978.

[10] Chu-Cheow Lim, Jerome A. Feldman, and Stephan Murer. Unifying Control- and Data-parallelism in an Object-Oriented Language. In *Joint Symposium on Parallel Processing 1993, Tokyo, Japan*, May 17 - 19 1993.

[11] Bertrand Meyer. *Object–oriented Software Construction.* Prentice Hall, New York, 1988.

[12] Sape J. Mullender. *Distributed Systems.* ACM Press Frontier Series. ACM Press, 1989.

[13] O. M. Nierstrasz. Hybrid – A Language for Programming with Active Objects. In D. C. Tsichritzis, editor, *Objects and Things*, pages 15 – 42. Centre Universitaire d'Informatique, University of Geneva, 1987.

[14] Alan Tully. Distributed Programming on Transputer Networks – An Object Oriented Model for Concurrent Processing. In Tariq S. Durrani, William A.

Sandham, and John J. Soraghan, editors, *Applications of Transputers 3*, pages 602–607, Glasgow, UK, 1991. IOS Press.

[15] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object–Oriented Software.* Prentice Hall, New York, 1990.

[16] Meng-Lai Yin, Lubomir Bic, and Theo Ungerer. Parallel C++ Programming on the Intel iPSC/2 Hypercube. In *Proceedings of the Fourh Annual Parallel Processing Symposium*, pages 380–394, 1990.

[17] Yasuhiko Yokote and Mario Tokoro. Concurrent Programming in Concurrent-Smalltalk. In Yonezawa and Tokoro [19], pages 129 – 158.

[18] Akinori Yonezawa, editor. *ABCL, An Object–Oriented Concurrent System.* The MIT Press, Cambridge, Mass., 1990.

[19] Akinori Yonezawa and Mario Tokoro, editors. *Object–Oriented Concurrent Programming.* The MIT Press, Cambridge, Mass., 1987.

[20] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification, Version 1.1. Technical Report Series ACPC/TR 92-4, Austrian Center for Parallel Computation, University of Vienna, Austria, 1992.

[21] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. ACM Press, 1990.