Objective Linda: A Coordination Model for Object-Oriented Parallel Programming

Ph.D. Dissertation, University of Siegen, Germany, 1997

Thilo Kielmann

1 Abstract

The rapid growth of interconnected high performance computing resources has fostered various efforts to develop the software infrastructure required for effectively utilizing the offered computational power. Although different types of software systems are already available to take advantage of parallel and distributed computer architectures, the exchange of information across networks is based on three primary programming paradigms: *message passing*, *remote procedure calls* and *distributed shared memory*.

Message passing is used to explicitly communicate information between simultaneously executing components of an application residing on different processors. Several message passing systems with different functionalities have been developed. They all rely on the basic capability to send and receive messages in an architecture-independent manner and elaborate on their functionality by providing variations on the sends (blocking, non-blocking, multicast, synchronous) and the receives (blocking, non-blocking, polling). More sophisticated constructs such as barriers, reduction operations, and process groups provide added value, but the most common systems operate in similar ways and are used for similar purposes.

Remote procedure calls allow the distribution of an application over a network and are therefore primarily intended for being used as a programming tool for client-server applications in distributed computing environments. Since with remote procedure calls a single thread of execution is passing across the network to another system, there is no inherent parallelism in their use; a threads package or particular system calls must be used in conjunction to provide parallelism.

Distributed shared memory is an abstraction for supporting the concept of real shared memory in a network environment without physically shared memory. In contrast to message passing and remote procedure calls, in a distributed shared memory system a process which stores a value does not need to know the existence or location of other processes which may fetch it. But distributed shared memory systems are difficult to implement because the inherent communication delays are significant and the shared memory paradigm requires all cooperating processes to have identical images of the shared address space.

A particular type of distributed shared memory system based on the notion of *generative communication* is the Linda system. Programming with this type of system is intended to be very simple and only requires the use of a small number of calls to put, examine or retrieve

data in the distributed shared memory, referred to as a *tuple space*. There are no explicit calls to transfer data to different processors or other ways to pass information between specific machines. The operation *out* puts data into the tuple space and the data is then available through calls to *in* and *rd*, which read the data with *in* removing it from the space. New processes that are capable of accessing the tuple space are spawned with the *eval* operation, supporting multiple threads of execution. Linda relies on computations to reduce the need for communications and is intended to perform at a level comparable to explicit message passing systems.

No matter which basic programming paradigm is used in a parallel programming language or application development toolkit, the present programming methodology for parallel computing environments does in general not provide suitable abstractions and software engineering methods for structured application design, in contrast to sequential programming where *object–oriented* techniques are by now well established for designing and implementing large application systems. The central notions exploited by object–oriented programming are objects, classes, and inheritance as means to structure applications and libraries of reusable software components. In the traditional object model, objects as the building blocks are defined as abstract data types which encapsulate their internal state through well-defined interfaces and thus simply represent passive data containers.

A simple approach to utilize the properties of object-oriented programming in a parallel environment is to add the notion of processes (usually in the form of lightweight processes, or "threads") to a given object–oriented language. In this way, all three basic programming paradigms described above can be realized: message passing is achieved by giving the method invocations the semantics of messages being exchanged, remote procedure calls are modelled by treating method invocations as procedure calls and allowing objects being called to be "remote", and shared memory programming is possible by adding monitor semantics to the objects in order to enable synchronization on the basis of the methods of the objects involved. In any case, the objects are responsible for keeping their own states consistent, such that the operations concurrently executing on a given object must be somehow synchronized, which in turn leads to interferences between concurrency control mechanisms and notions of reuse based on inheritance. This problem, known as the *inheritance anomaly*, is the main obstacle prohibiting the widespread use of this style of object–oriented concurrent programming.

A more promising approach to exploit object–oriented technology in parallel systems is (a) to keep the idea of encapsulated entities as suitable means for composing and refining predesigned plug–compatible software components and (b) unify the notions of (passive) objects and processes by introducing *active objects*, each with its own thread of control still being protected by its interface. This approach eliminates the consistency problems mentioned above, because there is exactly one thread operating in an active object. Of course, this is only true as long as the *coordination model* in charge for the communication between active objects is properly designed and does not introduce new consistency problems.

Coordination as the key concept for handling all the issues associated with communication has many facets and therefore many definitions depending on the individual viewpoint. An intuitive definition is the following: *"A coordination model is the glue that binds separate activities into an ensemble"*. In other words, a coordination model provides a framework in which the interaction of individual, active entities (called *agents*) can be expressed. In

this framework, all aspects of the creation and destruction of agents, the communication among agents, their spatial distribution, as well as the synchronization and distribution of actions performed by agents over time are subsumed. Coordination models consist of the following components: *coordination entities* (the agents which are to be coordinated), *coordination media* (the means enabling communication between agents), and *coordination laws* (the rules of how agents are coordinated making use of the given coordination media).

A natural way to realize a coordination model is to embody it in a *coordination language*, which ideally should be designed as an orthogonal combination of a coordination model (for the inter–agent actions) and a (sequential) computation model (for the intra–agent actions). The presumably most famous example of a coordination model is the Linda tuple space approach based on uncoupled generative communication, for which several linguistic embodiments like C–Linda or FORTRAN–Linda were developed, both for workstation networks and massively parallel architectures. Another prominent example are the *interaction abstract machines* which have been realized with implementations of the *linear objects* language *LO*.

In this thesis, *Objective Linda* is presented, a coordination model designed for the needs of parallel programming environments by rigorously combining object orientation with uncoupled generative communication. It is based on the the general Linda philosophy, but allows the benefits of object-oriented programming to be utilized for parallel application development. The functionality of Objective Linda is described, a formal semantics of its operations is presented, a prototype implementation is discussed, and its usefulness is demonstrated by several parallel programming examples.

The thesis is organized as follows:

In Chapter 2, current approaches to parallel programming with a particular focus on workstation clusters are presented. In general, there is a large gap between the low-level abstractions in which parallelism is typically expressed and the highly elaborated concepts of object orientation reflecting the state–of–the–art of programming sequential applications. The proposals for bridging this gap include concurrent and parallel object–oriented programming languages, as well as class libraries for use with existing object–oriented programming languages. It is argued that by focusing on the modelling of interaction itself rather than on hardware–related communication mechanisms or on properties of sequential programming languages, coordination models and languages represent highly promising approaches to parallel programming.

In Chapter 3, the new coordination model Objective Linda is presented. Objective Linda improves the original Linda model by replacing the notion of tuples by objects, and tuple spaces by object spaces. Objective Linda's operations treat objects as encapsulated entities, such that object matching (the process of identifying objects satisfying a request) is based on the object types plus the predicates defined by type interfaces. In addition to introducing object–orientation to the Linda model, Objective Linda also enhances the capabilities of the operations on object spaces. These enhancements include (a) the possibility to store and retrieve multisets of objects instead of single objects only, (b) the introduction of a timeout parameter to all operations which allows to smoothly adapt communication behaviour between immediately returning, as with Linda's inp and rdp, and infinitely blocking, as with Linda's in and rd, (c) a mechanism for dynamic composition based on generative communication which allows agents to dynamically attach to object spaces by identifying them using

objects of any type, hence using abstractions from the application level, and (d) a mechanism for migrating objects between distributed hosts (address spaces) based on type interfaces.

In Chapter 4, a formal presentation of Objective Linda's concepts and mechanisms is given. Objective Linda's notion of types and the corresponding subtyping relation will be specified, and a formal semantics of Objective Linda's operations on object spaces based on high–level Petri nets is presented. The formal presentation is completed by a type system for *agent types* that provides means for specifying the *observable behaviour* of Objective Linda's active objects.

In Chapter 5, the properties of a prototype implementation are discussed. This implementation allows Objective Linda to be used in conjunction to the C++ language due to its code efficiency. The implementation relies on the ACE toolkit, which provides a very powerful set of C++ classes for implementing multi–threaded systems of communicating processes.

In Chapter 6, Objective Linda's benefits are demonstrated using a small set of parallel example applications. These include a parallel computation of the Mandelbrot set, a parallel knapsack problem solution, and a parallel program for solving a colour quantization task. It is shown that Objective Linda's performance in terms of speedup is comparable to (or even better than) equivalent message passing solutions based on PVM.

In Chapter 7, Objective Linda's expressive potential for the modelling of other application domains than parallel computing is discussed. These include examples from the areas of multi–agent systems and collaborative distributed applications.

Chapter 8 concludes the thesis and outlines areas for further research.