Object-Oriented Programming and Parallelism

A. A. Radenski

Department of Computer Science, Winston-Salem State University

P. O. Box 13069, Winston-Salem, NC 27110, U.S.A.

E-mail: radenski@uncecs.edu

Initially, object-orientation and parallelism originated and developed as separate and relatively independent areas. During the last decade, however, more and more researchers were attracted by the benefits from a potential marriage of the two powerful paradigms. Numerous research projects and an increasing number of practical applications were aimed at different forms of amalgamation of parallelism with object-orientation. It has been realized that parallelism is a inherently needed enhancement for the traditional object-oriented programming (OOP) paradigm, and that object-orientation can add significant flexibility to the parallel programming paradigm.

Why add parallelism to OOP? Primary OOP concepts such as objects, classes, inheritance, and dynamic typing were first introduced in the Simula language and were initially intended to serve specific needs of real-world modelling and simulation. Object-orientation developed further as an independent general-purpose paradigm which strives to analyze, design and implement computer applications through *modelling* of real-world objects. From a programming perspective, object-orientation originated as a specific method for *modelling through programming* but evolved to a general approach to *programming through modelling*. Many real-world objects perform concurrently with other objects, often forming distributed systems. Because modelling of real-world objects is the backbone of the object-oriented paradigm and because real-world objects are often parallel, this paradigm needs to be extended with appropriate forms of parallelism.

Why add object-orientation to parallel programming? The high cost of specialized high-performance SIMD and MIMD machines is still an obstacle for some potential users. However, the rapid development of ATM networks and other fast connections opens opportunities to integrate existing workstations into relatively cheap distributed computing resources. Nowadays, diverse parallel processing platforms become increasingly available to application programmers. The expanding parallel applications cover not only traditional areas, such as scientific computations, but also new domains, such as, for example, multimedia. It has been widely recognized, however, that parallel languages and language environments are behind the needs of parallel programmers and users. For example, parallel software reuse and portability are particularly important areas that need improvement. Reuse of existing parallel software is very important because of the significant diversity of new parallel platform and the short lifetime of existing ones. Because parallel languages and compilers are architecture-oriented, parallel applications are difficult to port. Researchers believe that parallel programming can benefit from object-orientation in the same way as traditional serial programming does. For example, object-oriented languages can provide better reuse of parallel software through the mechanisms of inheritance and delegation. Object-orientation can also help with portability of parallel applications because OOP languages support high level of abstraction through separation of services form implementation. Parallel applications can be consistently and naturally developed through object-oriented analysis, design, and programming.

Current State of Parallel OOP

Very active research has been conducted in the last decade in the area of parallel OOP. A significant number of parallel OOP languages have been designed, implemented, and tested in diverse parallel applications. Despite of the progress in the area of parallel OOP, many difficulties and open issues still exist. Existing parallel OOP languages are often compromised in important areas, such as inheritance capability, ease of use, efficiency, heterogeneity, automatic memory management, and debugging.

Inheritance capability. Many of the proposed languages fail to provide inheritance for objects which can be distributed in a network and which can accept and handle messages concurrently. Even languages that permit some amalgamation of parallelism with inheritance tend to support only single-class inheritance. Most languages are weak in providing inheritance for the synchronization code of parallel objects.

Efficiency and ease of use. The largest group of experimental languages for parallel OOP consists of C++ extensions. Such extensions can be very large and complex, and therefore, not easy to learn, use, or implement efficiently. An alternative group includes interpretation-oriented languages (i.e. Smalltalk, Self, actor-based languages, Java) which do not provide high run-time efficiency and lack the reliable static-type checking.

Heterogeneity. Nowadays computing environments are becoming more and more heterogeneous. Users typically have access to a variety of platforms (such as workstations, PCs, specialized high-performance computers) which are networked locally or are geographically distributed. Most parallel OOP languages, however, are targeted at specific high-performance platforms, or implemented for homogeneous networks. There are no compilation-oriented languages that can convert heterogeneous networks into a single high-performance object-oriented computing environment in which the peculiarities of the specific architectures are transparent for the user.

Researchers have proposed diverse approaches to the problematic points of parallel OOP. For example, the following alternatives have been advocated: Enhancing popular serial OOP languages with parallelism versus designing entirely new parallel OOP languages, explicit parallelism versus parallelizing compilers, parallel OOP languages versus parallel OOP libraries, and so on.

The numerous proposals to integrate objects and parallelism typically follow two distinct scenarios: a) design a *new* parallel OOP language with built-in parallelism, or b) use an *existing* OOP language and enhance it with parallelism mechanisms. Most recent proposals follow the second approach. Its proponents take the object-oriented paradigm as given (on the basis of its contribution to the production of quality software) and investigate how to enhance it so that it covers parallel applications as well [4].

The transition from a sequential language to its parallel enhancement would be easier and less frustrating than the transition to a new language designed from scratch. The problem is not so much that of *learning* a new language as it is of *rewriting* a 100,000-line program [2]. For entirely practical reasons like the above one, the parallel extension of an existing language may have better utility than a new language designed from scratch.

Some researchers assume that the programmer is interested in and capable of specifying parallelism in *explicit* form. Because OOP means programming by modelling, and because real-world objects may exist and do things concurrently, OOP languages may have to provide explicit support to modelling parallelism. Other researchers adhere to the idea that parallelism should be *transparent* to the programmer, and that a *parallelizing compiler* should take the burden of finding and exploiting potential parallelism. It seems possible to combine both approaches in certain beneficial proportions.

A sequential language can be enhanced with explicit parallelism by means of a special *library* of low-level parallelism primitives, such as fork and join for example. Alternatively, the language can be extended with additional *linguistic constructs* that implement higher-level parallelism abstractions, such as monitors. The main motivation for the library approach is that the underlying sequential language does not need to be changed. External library primitives can provide flexible but uncontrolled access to data and hardware resources; unfortunately, such access can result in unreliable programs. Some authors overcome this difficulty through encapsulating the library services in special classes. Parallel class libraries are extended by end-users and adapted to their specific needs.

Parallel OOP has proven to be, or is expected to be, a beneficial implementation approach in a number of application areas, such as scientific computing, management information systems, telecommunications, and banking. New developments in the domain of parallel OOP occur in close interaction with other computer science fields, such as database management and artificial intelligence. More research is needed to further improve the design and implementation of parallel OOP platforms, and to increase their applicability.

In This Issue

The goal of this special issue is to provide a snapshot of some on-going projects on parallel OOP. Initial versions of these papers were presented in September 1995 at an invited session on Object-Oriented Programming, Concurrency, and Distribution at the Joint Conference on Information Sciences, Wrightstville Beach, North Carolina. The five articles included in the issue cover various aspects of the design, implementation, and applications of parallel object-oriented languages and systems.

The first article, "Object-Oriented Parallel Processing with Mentat", submitted by Andrew Grimshaw, represents a major project developed in the University of Virginia. This paper outlines the Mentat programming language, its implementation, and several of its applications. The Mentat programming language is a parallel extension of C++ which separates the responsibility for process generation between the programmer and the compiler. The programmer specifies classes of active objects which are to be instantiated with their own threads of execution, while the compiler automatically provides proper process synchronization and communication. Because part of the programmer's burden is taken by the compiler, Mentat is easier to use than systems in which process coordination is explicitly specified by the programmer. The paper describes four of the numerous successful applications of Mentat which show that Mentat is not only easy-to-use parallel OOP environment, but also that its performance is high. One of the applications, DNA and protein sequence comparison, is characterized with high data-parallelism and relatively simple master-worker coordination. For this type of problem the higher-level Mentat solution with automatically generated coordination has reached 90-95 percent of the efficiency of a hand-coded process coordination. The other three Mentat applications presented in the paper prove that Mentat can produce a considerable speed-up with different types of computationally intensive problems using diverse parallel platforms; these applications are a matrix algebra library, a library of stencil algorithms, and an implementation of the finite element method. The paper ends with an outline of the Legion project which is evolving from Mentat at the University of Virginia. Legion is aiming at converting available fast networks of eventually geographically distributed heterogeneous resources in a large virtual parallel computing recourse.

The second paper, "Parallel Object-Oriented Programming for Parallel Simulations", describes the work of Francoise Baude, Fabrice Belloncle, Denis Caromel, Nathalie Furmento, and Yves Roudier in the University of Nice of Sophia Antipolis, France, and contributions of Philippe Mussi and Gunther Siegel in INRIA Sophia Antipolis. The authors have developed the Sloop system which is a parallel object-oriented environment intended to be used for distributed discrete event simulation. The Sloop system consists of three layers. The main layer comprises C++//, an extension of C++ with a library of parallel programming classes. The computational model of C++// is based on a meta-object protocol which transforms method calls into regular objects which then can be sent to distributed processes, stored in queues, and in general, handled as first-class citizens. The lower layer of the Sloop system serves the needs of C++// for run-time support. This layer is an object-oriented abstraction of a communication network which is built on top of a widespread structured programming communication library, such as PVM. Finally, the upper layer is application-oriented and is intended to serve as a challenging testbed for C++//. This layer uses C++// to define classes for distributed discrete event simulation that users can customize for their specific modelling tasks. An event simulation consists of active objects which can execute services for other objects in the simulate time.

From a language design perspective, Mentat and Sloop are representatives of the wide-spread C/C++ culture. The alternative Pascal-Modula culture has been continued in the recent years by Oberon, an OOP language and a programming environment developed in the Swiss Institute of Technology in Zurich by Niklaus Wirth and Jurg Gutknecht. Oberon is small and efficient language in which classes are emulated by means of a minor enhancement of the traditional concept of a record type. The paper "Oberon, Gadgets, and Some Archetypal Aspects of Persistent Objects" by Jurg Gutknecht outlines the module hierarchy and the type hierarchy in the original Oberon environment, and then specifies extensions that provide persistency, more convenient visual interfaces, and parallelism. The paper proposes to specify separate threads of execution as bodies that are attached to parallel, or active objects. In contrast, passive objects are just records without such bodies and therefore, without separate threads of execution.

One more project in the Pascal-Modula tradition is outlined in the paper "Concurrent Object-Oriented Programming for Distributed Real-Time Systems". The author, Katsumi Maryuama, worked for the Japanese NTT Network Service Systems Laboratories and was guided by the needs for real-time OOP in the area of telecommunications. A typical telecommunication program, such as a switching system, needs to be highly efficient in processing a potentially large number of concurrent calls. Telecommunication applications are expected to allow easy modifications and extensions, such as adding new functions or modifying existing ones. The paper suggest to satisfy the demanding requirements of such applications through a careful integration of parallelism and object-orientation on a programming language level. Some language design principles are first advocated, then used for the design of a particular language, ACOOL. The proposed language can be viewed as an enhancement of Modula-2 with passive objects and with transparently distributed active objects. The paper also outlines a switching system application framework in which individual calls are handled by specially created active objects, "call agents".

Reasoning about concurrent object-oriented programs is a challenging activity for which program visualization can be beneficial. Program visualization is expected to help users cope with the high complexity of such tasks as, for example, debugging, performance tuning, or just studying parallel object-oriented programs. Visualizing parallel OOP programs is difficult because the high complexity of the underlying parallel object-oriented mechanisms. This task is attacked in the last paper, "A Visualization Model for Concurrent Systems", submitted by Mark Astley and Gul Agha from the University of Illinois at Urbana-Champaign. The authors take the Actor model of parallelism as given, and put on top of it a set of visualization primitives that can be used to specify visualization events and corresponding visualization actions. Linguistically, the visualization primitives are integrated in program units called visualizers. The purpose of a visualizer is to filter interactions between concurrent programming components and trigger relevant visualization activities. A careful choice of visualization activities is expected to enhance the programmer's capability to reason about asynchronous parallel programs without significantly deteriorating their performance.

Readers who are interested in learning more on parallel OOP may want to read the surveys [5, 1] and look in the collection [3]. Besides, the bibliographies of papers included in this issue contain examples of current projects on parallel OOP.

Acknowledgment. This special issue would not be possible without the understanding and the support of the editor-in-chief, Paul Wang. Thanks are also due to authors for their readiness to write good quality paper, revise them and promptly prepare them for publication.

References

1. Agha G., Concurrent Object-Oriented Programming, *Communications of the ACM*, 33, No 9 (Sep.), 1990, 125-141.
2. Almasi, G., A. Gotlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Co., Inc., 1994.
3. Agha A., P. Wegner, A. Yonezawa, *Research Directions in Concurrent Object-Oriented Programming*, The MIT Press, 1993.
4. Meyer B., Systematic Concurrent Object-Oriented Programming, *Communications of the ACM*, 36, No 9 (Sep.), 1993, 56-80.
5. Wyatt B., K. Kavi, S. Hufnagel, Parallelism in Object-Oriented Languages: A Survey, *IEEE Software*, 9, No 6 (Nov.), 1992, 56-66.