Software Engineering for Parallel Systems: The Role of the Programming Language in the Prototyping Phase

Position Paper for the
ICSE-17 Workshop on research issues in the intersection of
Software Engineering and Programming Languages
Seattle, April 1995

W. Hasselbring
Dept. of Computer Science, University of Dortmund
Informatik 10 (Software Technology), D-44221 Dortmund, Germany
Telephone: 49-(231)-755-4712, Fax: 49-(231)-755-2061
email: willi@ls10.informatik.uni-dortmund.de

Abstract

Engineering parallel software systems is still in its infancy. At present, there is a wide gap between formal approaches to algorithm specification and practical approaches to algorithm implementation. In this contribution, we emphasize the step from specification to implementation in parallel programming and discuss applications experience with a prototyping approach, which exploits the strength of a set-oriented language to program design through rapid prototyping of parallel algorithms.

1 Introduction

It is a well-known fact that the cost to correct an error in a computer system increases dramatically as the system life cycle progresses [4]. The cost of correcting an error increases by orders of magnitude as the system moves from the development stages of analysis and design, to become most expensive during the maintenance and operation phase. Formal specification and prototyping help to eliminate many of these errors in the very early stages of a project before any production-level code has been written.

Furthermore, parallel programming is conceptually harder to undertake and to understand than sequential programming, because a programmer often has to focus on more than one process at a time. Additionally, on most of today's parallel machines, programmers are forced to program at a low level to obtain performance — ease of use is sacrificed for efficiency. Consequently, developing parallel software systems is in general considered as an awkward undertaking. A high-level language designed for prototyping parallel algorithms helps to manage engineering parallel software systems.

2 Prototyping

First, a prototype helps the specification writer to evaluate the specification. It also helps a potential user to explore the capabilities of the system. It is often only through this type of experience that the

necessary functional requirements can be discovered. Furthermore, it is better to have the user discover needs early in the production process, and not after the system has been completely implemented and delivered. The prototype provides the user with a vehicle which can be exercised to see if it meets the (sometimes fuzzy) requirements. Users are involved in the system development process which supports the communication between users and developers.

Prototypes should be built in very high level languages to make them rapidly available. To be useful, prototypes must be *built* rapidly, and designed in such a way that they can be *modified* rapidly. Consequently, a prototype is usually not a very efficient program since the language should offer constructs which are semantically on a very high level, and the runtime system has a heavy burden for executing these highly expressive constructs. Note that a prototype is a model, and that this model taken as a program has to be executable so that at least part of the functionality of the desired end product may be demonstrated on a computer [6].

3 Parallel Programming related to Prototyping

Most current programming environments for distributed memory architectures, which are usually based on some kind of message passing, provide inadequate support for mapping applications to the machine. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level, since it is complicated to simulate shared memory. This greatly increases the complexity of programs, and also fixes algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. This makes programming such machines very difficult, since the programmer has to explicitly encode all the low-level details required to implement the algorithm. The resulting programs are complex and inflexible.

Meanwhile, the techniques of parallel programming are evolving slowly. Initially, parallel programs were a collection of sequential programs written in traditional sequential languages. These languages were extended with communication instructions to enable message-passing communication. Such synchronizing communication is feasible in problems with completely regular patterns of execution, but it is difficult to control in areas where the execution pattern is strongly data-dependent and irregular.

Therefore, it is clear that changes in parallel programming languages are needed — in particular it should be possible to dynamically create computations, to synchronize them and to allow information exchange between them. Several languages have been proposed with this goal in mind: for instance C.A.R. Hoare proposed an influential model called CSP (Communication Sequential Processes) allowing the definition, activation and synchronization of communicating processes [18]. However, it is very difficult to build parallel programs with this approach. This is mainly due to the fact that the programmer has to mentally manage several threads of control simultaneously instead of one at the time.

During the past years several *high-level* languages with mechanisms for parallel programming have been developed to alleviate parallel programming. Even though such high-level languages were designed to express parallelism, they are mostly intended to be implemented and used on ordinary computers with only one central processing unit. Parallel programming is then used as a convenient way of expressing logical relationships between different parts of a complex program, and only secondary for increasing the execution speed.

Achieving speedup through parallelism is a common motivation for executing an application program on a parallel computer system. The main motivation for integrating explicit parallelism into a prototyping language is to provide means for modeling inherently parallel applications. Consider, for instance, distributed systems such as air-traffic-control and airline-reservation applications, which must respond to many external stimuli and which are therefore inherently parallel. To deal with nondeterminism and to reduce their complexity, such applications are usually structured as independent parallel processes. Similarly, a company with multiple offices and factories may need a computing

system which enables people and machines at different sites to communicate with each other. Such a system has to run on distributed hardware and, thus has to be programmed in a parallel way.

Programmers who can express their ideas in a parallel way sometimes invent entirely new ways of solving problems. In order to embody their inventions in working programs they need languages that allow parallelism to be expressed explicitly — languages based on parallel software models.

3.1 Possible Approaches to Prototyping Parallel Algorithms

Parallel computers are traditionally divided into two broad subcategories: tightly coupled and loosely coupled systems. In a tightly coupled system at least part of the primary memory is *shared*. All processors have direct access to this shared memory. In a loosely coupled (distributed) system, processors only have access to their own local memories; processors can communicate by sending messages over some kind of communication channel. Tightly coupled systems have the advantage of fast communication through shared memory. Distributed systems, on the other hand, are much easier to build, especially if a large number of processors is required. Initially, programming language and operating system designers strictly followed the above classification, resulting in two parallel programming paradigms: shared variables (for tightly coupled systems) and message passing (for distributed systems).

Many languages for parallel programming have evolved during the last years, making the choice of the most suitable language for prototyping parallel algorithms a difficult one. More important, the underlying *models* of the languages differ widely.

Message Passing The basic model of *message passing* is that of a group of sequential processes running in parallel and communicating through passing messages. This model directly reflects the distributed memory architecture, consisting of processors connected through a communications network. Many variations of message passing have been proposed [3].

There exist some approaches to develop prototypes for message-passing programs on the basis of Petri-nets or data-flow diagrams. Petri-Nets and data-flow diagrams can be used as graphical representations of message-passing systems. Therefore, they are often used to build prototypes for message-passing programs. For instance, prototypes for occam programs are developed with Petri-nets in [5], and with data-flow diagrams in [19].

For some applications, the model of message passing may be just what is needed. This is, for example, the case for an electronic mail system. For other applications, however, this basic model may be too low-level and inflexible. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level, since it is complicated to simulate shared memory. Refer to [2] for an extensive discussion of the shortcomings of the message-passing model. Most of the problems which arise with message passing exist similarly for graphical approaches based on Petri-nets or data-flow diagrams, because such graphs are just graphical representations of message-passing systems.

In contrast to the message-passing model, the shared-memory model allows application programs to use shared memory as they use normal local memory. The primary advantage of shared memory over message passing is the simpler abstraction provided to the application programmer, an abstraction the programmer already understands well. This allows a more natural transition from sequential to parallel programming.

Data Parallelism Data parallelism extends conventional programming languages so that some operations can be performed simultaneously on many pieces of data. All the elements in a list or in an array can be updated at the same time, for example, or all items in a data base are scanned simultaneously to see if they match some criterion. For an account to data parallel algorithms see [17] and for an account to data-parallel programming see [21]. Data-parallel operations appear to be done *simultaneously* on all affected data elements. This kind of parallelism is opposed to *control parallelism* that is achieved through multiple threads of control, operating independently. According

to Flynn's taxonomy of computer architectures, the data-parallel programming model is based on the *single-instruction-stream/multiple-data-stream* (SIMD) model as opposed to the *multiple-instruction-stream/multiple-data-stream* (MIMD) model [11]. The SIMD programming model is *synchronous* because all active processing elements execute the same operation simultaneously.

Data parallelism is opposed to *control parallelism* which is achieved through multiple threads of control, operating independently. The data parallel approach lets programmers replace iteration (repeated execution of the same set of instructions with different data) with parallel execution. It does not address a more general case, however: performing many interrelated but *different* operations at the same time. This ability is essential in developing complex application programs. We are searching for appropriate means that enable the expression of *new* parallel algorithms for implementing inherently parallel systems, and not primarily to increase the execution performance of prototypes.

Parallel Object-Oriented Programming An approach to imperative programming which has gained widespread popularity is that of object-oriented programming [20]. In this approach, an object is used to integrate both data and the means of manipulating that data. Objects interact exclusively through message passing and the data contained in an object is visible only within this object itself. There are several possibilities for the introduction of parallelism into object-oriented languages [25].

Parallel object-oriented languages tend to use either message passing or remote procedure calls for inter-process communication: the object space (the collection of all objects in the program) is not the communication medium, and does not constitute a *shared* object memory. Therefore, most of the problems with programming parallel applications which arise with message passing exist similarly for parallel object-oriented languages.

Additionally, probably the most difficult aspect of integrating parallelism into object-oriented languages is that inheritance greatly complicates synchronization. When a subclass inherits from a base class, programs must sometimes redefine the synchronization constraints of the inherited method. If a single centralized class explicitly controls message reception, all subclasses must rewrite this part each time a new operation is added to the class. The subclass cannot simply inherit the synchronization code, because the highest-level class cannot invoke the new operation. The parallel object-oriented languages resolve these synchronization problems in different ways. Refer to [1] for a discussion of the resulting problems and various solutions.

Parallel Functional Programming A functional program comprises a set of equations describing functions and data structures which a user wishes to compute. The application of a function to its arguments is the only control structure in pure functional languages. Functions are regarded in the mathematical sense in that they do not allow side effects. As a consequence a value of a function is determined solely by the values of its arguments, a property which is referred to as *referential transparency*. Therefore, functional programs are inherently parallel. Because they are free of side effects, each function invocation can evaluate all of its arguments and possibly the function body in parallel. The only delay may occur when a function must wait on a result being produced by another function. The real problem is not discovering parallelism but reducing it so as to keep the overhead on an acceptable level. Parallel functional languages address this problem by allowing the programmer to insert annotations which specify when to create new threads of control. Refer to [23] for a collection of papers on several parallel functional languages.

However, pure functional languages are not suitable for programming cooperating processes: they are deterministic and they do not have variables. Therefore, processes described as functions cannot include choices of alternative actions and they cannot remember their states from one action to another. Nondeterminism would destroy referential transparency in functional programming languages.

Processes sometimes cooperate in a way that cannot be predicted. It is impossible, for instance, to predict from which terminal of a multi-user computing system the next request for a particular service might come. Moreover, the system behavior necessarily depends on previous requests. Both nondeterminism of events and dependence on the process history are strong arguments for an imperative

rather than applicative programming model for cooperating processes. This is due to the determinism and the lack of variables which make pure functional languages impractical for programming parallel systems. Even the parallel functional languages do not support nondeterminism.

Parallel Logic Programming Parallel logic languages usually use *shared logical variables* as a communication medium. In [22], it is described how several communication patterns can be expressed using shared logical variables despite the single-assignment property of such variables. On the other hand, the shared logical variable also has its problems. Although it is possible to implement shared data structures like streams and queues using shared logical variables, only a single process can add elements to such data structures. Some problems with shared logical variables in parallel logic languages are discussed in [8].

Coordination Languages A *coordination language* provides means for process creation and inter-process communication which may be combined with sequential *computation languages* to create parallel programming languages [7]. Linda is a coordination language concept for explicitly parallel programming in an architecture independent way [12]. Communication in Linda is based on the concept of *tuple space*, i.e., a virtual common data space accessed by an associative addressing scheme. The parallel processes are decoupled in time and space in a simple way: processes do not have to execute at the same time and do not need to know each other's addresses. Process communication and synchronization in Linda is called *generative communication*, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly.

Reading access to tuples in tuple space is associative and not based on physical addresses. It is based on their expected content described in so-called *templates*. This method is similar to the selection of entries from a data base. Each component of a tuple or template is either an *actual*, i.e., holding a value of a given type, or a *formal*, i.e., a declared placeholder for such a value. Tuples in tuple space are selected by *matching*, where a tuple and a template are defined to match, iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields.

The uncoupled and anonymous inter-process communication in Linda is in general not directly supported by the target machines. However, a *high-level* language must be able to reflect a particular top-down approach to building software, and not a particular machine architecture. This is also important to support portability across different machine architectures. Implementations of Linda have been performed on a wide variety of parallel architectures: on shared-memory multi-processors as well as on distributed memory architectures [24].

4 Experience with an Approach to Prototyping Parallel Algorithms in a Set-Oriented Language

To support prototyping of parallel algorithms, a prototyping language must provide simple and powerful means for dynamic creation and coordination of parallel processes. ProSet is a set-oriented prototyping language [9]. In ProSet-Linda, the concept for process creation via Multilisp's futures [13] is adapted to set-oriented programming and combined with Linda's concept for synchronization and communication [14]. Synchronization and communication in ProSet-Linda are carried out through addition, removal, reading, and atomic updates of individual tuples in tuple space. Refer to [15] for an account to prototyping parallel algorithms in a set-oriented language.

In this position paper we can only sketch some applications experience. In our contribution we would discuss experiences with two example applications: developing algorithms with ProSet-Linda for parallel interpretation-tree model matching and cooperative planning of independent agents.

4.1 Parallel Interpretation-Tree Model Matching

In [16], we discuss the development of algorithms for parallel interpretation-tree model matching for 3-D computer vision applications such as object recognition.

The classical control algorithm for symbolic data/model matching in computer vision is the *Interpretation Tree* search algorithm. This algorithm has a high computational complexity when applied to matching problems with large numbers of features. We examine parallel variations of this algorithm. Parallel execution can increase the execution performance of model matching, but also make feasible entirely new ways of solving matching problems. The expected improvements attained by the parallel algorithmic variations for interpretation-tree search are analyzed.

4.2 Cooperative Planning of Independent Agents

In [10], we discuss the development of algorithms for cooperative planning of independent agents by means of an example application.

Cooperative planning of independent agents is a problem which requires careful study. For concentrating on the essential aspects (plan generation, conflict resolution) we propose a prototypical approach. Finding a clear and intelligible solution to plan generation and conflict resolution is certainly more important than obtaining directly a very efficient program — once a solution is found through exploration, it may be used as an executable specification for an efficient implementation. Consequently, we concentrate on conceptual aspects and implement our solution in a prototyping language.

This is what prototyping is about: experimenting with ideas for algorithms and evaluating them. Purely theoretic evaluations are often not possible in practice. A very high-level language for parallel programming is needed to make this approach feasible.

References

[1] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[2] H.E. Bal. *Programming Distributed Systems*. Silicon Press, 1990.

[3] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[4] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[5] F. Bréant and E. Pavoit-Adet. Occam prototyping from hierarchical Petri nets. Technical Report MASI 92.08, University of Paris 6, Institut Blaise Pascal, Paris, France, February 1992.

[6] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Züllighoven. *Prototyping — An Approach to Evolutionary System Development*. Springer-Verlag, 1992.

[7] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[8] P. Ciancarini. Parallel programming with logic languages: a survey. *Computer Languages*, 17(4):213–240, April 1992.

[9] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992. IEEE Computer Society Press.

[10] E.-E. Doberkat, W. Hasselbring, and C. Pahl. Investigating strategies for cooperative planning of independent agents through prototype evaluation. In P. Ciancarini and C. Hankin, editors, *Proc. First International Conference on Coordination Languages and Models (COORDINATION '96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 416–419, Cesena, Italy, April 1996. Springer-Verlag.

[11] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[12] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[13] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[14] W. Hasselbring. Prototyping parallel algorithms with ProSet-Linda. In J. Volkert, editor, *Parallel Computation*, volume 734 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, October 1993.

[15] W. Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language*. PhD thesis, Department of Computer Science, University of Dortmund, 1994. (Published by Verlag Dr. Kovač, Hamburg).

[16] W. Hasselbring and R.B. Fisher. Investigating parallel interpretation-tree model matching algorithms with ProSet-Linda. DAI Research Paper No. 722, University of Edinburgh, Dept. of Artificial Intelligence, Edinburgh, UK, December 1994. (also available as Software-Technologie Memo Nr. 77, University of Dortmund).

[17] W.D. Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[18] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[19] D.G. Jones, S.J. Dowdeswell, and T. Hintz. A rapid prototyping method for parallel programs. In T. Bossomaier, T. Hintz, and J. Hulskamp, editors, *The Transputer in Australia (ATOUG-3)*, pages 121–128, Sydney, Australia, June 1990. IOS Press.

[20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[21] M.J. Quinn and P.J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, September 1990.

[22] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[23] B.K. Szymanski, editor. *Parallel functional languages and compilers*. Addison-Wesley, 1991.

[24] G. Wilson, editor. *Proc. Workshop on Linda-Like Systems and Their Implementation*. Edinburgh Parallel Computing Centre TR91-13, June 1991.

[25] B.B. Wyatt, K. Kavi, and S. Hufnagel. Parallelism in object-oriented languages: A survey. *IEEE Software*, 9(6):56–66, November 1992.