A Uniform Object Model for Parallel Programming

Lawrence A. Crowl
University of Rochester
Computer Science Department
Rochester, New York 14627
716-275-9499 crowl@cs.rochester.edu

23 September 1988

1 Introduction

One of the problems with parallel programming is that the languages are often adapted from sequential languages rather than designed specifically for the task. Current imperative languages for parallel programming limit the range of programming styles, force premature binding of parallelism, and limit the class of architectures for which they are effective. This abstract describes Matroshka (Матрёшка)[1], a uniform object model for parallel programming. The Matroshka model is based on a small number of concepts and enables a wide range of programming styles, late binding of parallelism, and effective use of a wide range of architectures. Late binding of parallelism means that programmers may decide which abstractions execute concurrently late in program development. The model admits a wide range of languages by leaving many language issues unspecified such as syntax, inheritance, static or dynamic typing, *etc.* Natasha (Натáша, or Natáša) is a simple, statically typed language based on the Matroshka model. An implementation of Natasha on the BBN Butterfly Parallel Processor [BBN, 1985a] is in progress.

Most current parallel programming languages support only a narrow range of programming styles, such as message passing (PLITS [Feldman, 1979]), rendezvous (Ada [U. S. DoD, 1983], Lynx [Scott, 1987]), and tightly-coupled shared memory (MultiLisp [Halstead, 1986], Uniform System [BBN, 1985c]). However, many applications contain computational tasks that are best expressed in different programming styles. For example, subgraph isomorphism has both tree-search parallelism and vector parallelism. With the available parallel programming systems, programmers must either use a single style or use multiple programming systems. The interface between multiple programming systems often consumes a considerable portion of the coding effort [Atkinson *et al.*, 1983]. A language model for imperative parallel programming should support a wide range of programming styles, from shared memory to message passing, and a granularity of sharing ranging from integers to large data structures, while providing a single conceptual framework. Some languages addressing portions of this problem include Orient84/K [Ishikawa and Tokoro, 1986], SR [Andrews *et al.*, 1986], and StarMod [Cook, 1980]. In constrast to previous work, we use a single mechanism to achieve multiple styles rather than provide many mechanisms.

Current imperative languages for parallel programming tend to provide separate facilities for parallel programming on top of a base sequential model of programming. These facilities include a

[1] Matroshka (often transliterated as Matrëška) are the wooden Russian dolls where the smallest nests within the next smallest and so on. They are shaped somwhat like squat bowling pins and are painted to depict Russian peasant women.

distinct notion of process, monitor constructs, and messages, *e.g.* Argus [Liskov, 1982], NIL [Strom and Yemini, 1983], SR [Andrews *et al.*, 1986]. This approach presents programmers with two qualitatively different models of interaction between program components, which forces them to bind parallelism early in program development, often with the higher-cost, more general mechanism [Greif *et al.*, 1986]. Unfortunately, the appropriate binding of parallelism is often not known until experimental performance analysis is undertaken late in program development. A programming model that enables programmers to delay the binding of actual parallelism must be based on abstraction mechanisms that apply to both sequential and parallel execution. Some languages addressing portions of this problem include ABCL/1 [Yonezawa *et al.*, 1986], Actors [Agha, 1986], and Emerald [Black *et al.*, 1986a]. In contrast to earlier work, the Matroshka model provides uniformity in control as well as abstraction.

A consequence of binding parallelism early with special language constructs is that often a parallel programming system works well on only one architecture because its underlying model assumes a specific architecture. These languages represent a means for programming a machine, not a description of a parallel solution. To provide efficient use of a wide range of architectures, a parallel programming language must enable description of parallelism at all granularities within a program, and enable tuning a program to an architecture without affecting the semantics of the source. The programmer is still responsible for adapting the program to the architecture, but such concerns are separate from program construction.

2 Description

This section describes the Matroshka model of parallel programming, not a programming language. The model has a wide range of possible instantiations as programming languages. Many issues in language design are not specified in the model, such as inheritance, static or dynamic typing, *etc.* The main characteristics of the model are: uniformity base on objects for both encapsulation and control, concurrent execution, and a copy model of variables and parameters.

2.1 Uniformity in Encapsulation

Most imperative parallel programming systems have two mechanisms for encapsulation, one for parallel and distributed objects and one for local, sequential objects. This dual mechanism approach splits the programming environment into two qualitatively different models of interaction, introducing an artificial granularity in the programmer's specification of parallelism. Parallel programming systems need uniform encapsulation mechanisms that apply to both parallel and sequential objects. The Matroshka model provides a single, uniform encapsulation mechanism, the *object*. The object model provides natural encapsulation of abstract data types from simple integers to distributed databases.

Not only must the encapsulation mechanism apply uniformly to parallel and sequential objects, the means for communicating with them must also apply uniformly. The originating object must be able to initiate communication without knowing how the receiving object will handle it, and vice-versa. Otherwise, the cost of providing complete flexibility is combinatorial in the number of communication mechanisms. The Matroshka model provides uniform communication in the form of synchronous operation invocation on objects. The invoker of an operation waits upon receipt of the reply value. Reply values are implicitly returned to the caller and do not require explicit addressing. This is the sole communication mechanism between objects.

2.2 Uniformity in Control

Parallel programming systems that support data parallelism must provide a mechanism for partitioning work among the data, without requiring a priori knowledge of the work to be performed. This means that the programmer must be able to reference code, as well as data, indirectly; which coupled with the desire for late binding of parallelism, implies that control constructs apply uniformly to parallel control as well as to sequential control. Since the appropriate parallelization of control is generally dependent on user data structures, a programmer must be able to define new control structures, as well as use primitive control structures (*e.g.* iteration over a list). The issue of programmable control abstractions is not unique to parallel systems, it is common to programming in general.

The Matroshka model simplifies programming of control abstractions by allowing exactly one parameter and one result for each operation. This means that control abstractions need only handle one argument and result combination. The effect of multiple parameters is achieved by passing a record as the single argument. If a language based on the Matroshka model provides *record constructors* (*e.g.* Mesa [Xerox, 1984]), this approach can be as notationally concise as a list of parameters. Those operations that do not need an argument, or have no useful result, accept or return an empty record.

The Matroshka model provides for some measure of communication independence with *ports*. A port is a first-class language entity binding an operation to an object reference. An operation is invoked when an argument object is applied to a port. The user of a port may need to know the type of the argument and the type of the result, but does not need to know the operation name or the type of the port's object. This allows programmers to connect objects into rich networks, where the exact type of objects is not known in advance, even in the context of statically typed languages.

The need for defining control abstractions is greater in parallel languages than in sequential languages, because otherwise each programmer must build *ad hoc* mechanisms for creating parallelism. The Matroshka model provides for the effective implementation of control abstractions with *blocks*, which are similar to Smalltalk's blocks [Goldberg and Robson, 1983] and Lisp's closures [Steele, 1984]. The contexts of executing operations are *activation objects* (or activation records). Blocks are anonymous operations on activation objects. As such, blocks are implicitly an object reference and operation pair, so the specification of a block defines a port. This port is exactly the same mechanism introduced earlier, and may be passed outside the enclosing scope to be executed by other objects. Blocks maintain access to the variables defined within the scope of the block. The ability to pass blocks as ports provides the basis for control flow within the model.

A model that does not provide an indirect handle for operations (one that has no mechanism for non-literal reference to operations) prevents the programmer from building general-purpose control abstractions that can control the ordering of operations. There are two approaches to providing operations as data. The first approach is to provide variables whose values are operation names. This approach forces run-time type checking, so is only appropriate for dynamically typed languages. For statically typed languages, one may bind an operation and an object type into an *action*. An action is similar to a port, but requires an object upon which to operate, in addition to an argument object. The choice between these approaches depends upon the nature of the language in which the model is implemented.

2.3 Copy Model of Variables and Parameters

There are two models of variables and parameters in imperative languages. In the conventional model, variables contain values (*e.g.* the Algol family of languages). We call this the *copy* model. In the *reference* model, variables refer to objects (CLU [Liskov *et al.*, 1977], Smalltalk [Goldberg and Robson, 1983]). Most imperative parallel languages use the reference model for concurrent abstractions, but use the copy model for parameters and local variables. Systems with uniform encapsulation (*e.g.* Actors [Agha, 1986] and Emerald [Black *et al.*, 1986b]) use the reference model.

The reference model has some attractive properties. However, the copy model has several advantages in the context of parallel and distributed programming. These include:

- controlled aliasing, which aids the exploitation of parallelism
- decreased contention for argument objects, because they are copied rather than referenced
- decreased inter-processor communication for accessing arguments on other processors
- more opportunities for optimizing parameter passing
- no need for garbage collection, because lifetime can be associated with scope

The Matroshka model adopts the copy model of variables and parameters. Variables contain objects, they do not refer to objects. This means that operation arguments are objects, not object references. Because a strict copy model of computation can only represent hierarchical structures, the Matroshka model provides an explicit *object reference* capability. The type of a reference depends on the type of the referent. Real programs must deal with values as well as references, so the choice of the copy model in a language represents a bias, and not an absolute choice. The bias of the copy model is more appropriate for parallel programming than the reference model.

2.4 Concurrent Execution

Because the appropriate boundaries between the parallel and sequential portions of a program may not be known until late in program development, programmers must have the ability to specify concurrency at all points within their program, and to change the implementation of an abstraction to increase or decrease the utilization of available concurrency while not affecting other abstractions. Any limitation on the potential concurrency within the encapsulation mechanism will magnify itself at each level of abstraction.

In a system with uniform encapsulation, many abstractions will be large. To prevent unnecessary synchronization, the Matroshka model provides *concurrent operation execution* within a single object. Each object invocation provides a new, concurrent thread of control within the object. The operation invocation itself implicitly dispatches the thread of control for execution. Thus, while invocations are synchronous with respect to their callers, they are asynchronous with respect to other invocations on the object. The model itself does not supply synchronization. If the programmer needs to synchronize invocations, the programmer must explicitly program such synchronization using language or implementation defined object types.

The Matroshka programming model provides for asynchronous processing between a server and its clients with an *early reply* on operation invocation. After the reply, the callee may continue processing for an arbitrary time. The use of asynchronous processing is transparent to the caller, allowing the implementation of an object to change according to the system's need for concurrency. Since the invoker may continue after receiving the reply, this early reply provides a source of parallelism. *This is the sole source of parallelism provided by the model.* It is a sufficient mechanism for implementing other forms of parallelism. Early reply maintains the local state of the operation after reply, but without the cost of forced concurrent access to activation variables.

Not only must a parallel programming model enable handling requests in parallel, it must enable making requests in parallel and receive the results at a later time in order to fully exploit available parallelism. This capability is *asynchronous invocation*. The programming model could introduce additional mechanisms for asynchronous invocation, but early reply, coupled with generic definitions, is sufficient to enable straight-forward implementation of asynchronous invocation. Therefore, the model does not provide asynchronous invocation directly. Such an approach encourages the implementation of different kinds of asynchronous invocations, each tuned to the needs at hand.

3 Discussion

This section describes how the Matroshka model provides multiple programming styles, late binding of parallelism, and effective use of multiple architectures.

3.1 Multiple Programming Styles

We provide evidence for multiple programming styles by showing the implementation of both shared-memory and message-passing styles within the Matroshka model.

The shared memory style is characterized by processes calling procedures concurrently on shared data structures. This style is easily programmed under the Matroshka model by representing the data structure as an object, allowing operations to proceed concurrently, and delaying reply of an operation until the operation has finished.

The message passing style is characterized by processes asynchronously exchanging messages and then handling such messages one at a time. This style is slightly more complicated. Objects represent processes, the invocation of an operation represents the sending of a message. Each operation replies immediately (which provides asynchrony with the caller), then synchronizes with other possible invocations via a semaphore within the object, and finally performs the necessary work. The early reply provides asynchrony and the semaphore provides exclusive processing.

3.2 Late Binding of Parallelism

The model supports late binding of parallelism through mechanisms that allow the disciplined use of data and control abstractions with multiple implementations. Each implementation exploits a different level of concurrency, at a different cost. The multiple implementations strategy applies to both user-defined and primitive abstractions.

For example, there are no dependencies between distinct iterations of the loops to implement matrix addition. In choosing an abstraction that provides no guarantees on the relative execution of iterations, programmers may delay the choice of sequential or parallel implementations of the loops. On uniprocessors, the most efficient approach is to use no parallelism. On a uniprocessor with vector hardware, the most efficient approach may be to parallelize the inner loop, but maintain sequentiality in the outer loop. On a shared-memory multiprocessor, the most efficient approach may be just the opposite, to parallelize the outer loop, but maintain sequentiality in the inner loop. The choice of implementation can be made without affecting the semantics of the source.

3.3 Multiple Architectures

The Matroshka model enables efficient use of multiple architectures by associating communication with abstraction. Since programmers will use layers of abstractions, the implementation can communicate across processor boundaries at any layer of abstraction. On a distributed system, object invocation at higher levels of the program's abstractions would be implemented by messages across the communication network, and invocation at lower levels would be implemented by procedure calls.

On a shared-memory multiprocessor, object invocation at all levels of abstraction would be implemented by procedure calls, except at the lowest level, where processors read and write individual words in shared memory.

Note that not all programs that execute efficiently on a shared-memory multiprocessor will execute efficiently on a distributed system. In particular, if a program has a great deal of small, communication intensive objects with no intervening layers of abstraction, the communication graph may be too fine-grained for efficient implementation on a distributed system.

4 Implementation

The Natasha programming language is a prototype instance of the Matroshka model. The language is being implemented on the BBN Butterfly Parallel Processor [BBN, 1985a] under the Chrysalis Operating System [BBN, 1985b]. The prototype emphasizes aspects of the implementation relevant to parallel processing and ignores optimizations commonly employed in sequential languages.

A strict interpretation of the semantics of early reply suggests that operations be implemented via independently schedulable threads of control. However, in many cases the reply can be delayed until after completion of the operation without affecting the program. This allows the compiler to implement the operation within the thread of control of the caller. The compiler will, at the direction of a pragma, delay the reply and implement the operation with a less costly procedure call.

Programs under the Matroshka model are likely to generate a large number of schedulable threads of control. The runtime scheduler for Natasha implements a LIFO algorithm, which corresponds to a depth-first evaluation of the tree of potential threads ($O(\log n)$ simultaneous activations). This approach is preferable to the normal FIFO scheduling, which corresponds to a breadth-first evaluation ($O(n)$ simultaneous activations).

The Butterfly architecture can effectively emulate a wide range of architectures, including conventional uniprocessors, small bus-based shared-memory multiprocessors, large network-based shared-memory multiprocessors, large message-based multiprocessors, and distributed systems. These emulations may be implemented as a different set of language runtime procedures. By varying the implementation of operations, programs can be tuned to any of the various emulations. The implementation of Natasha on the Butterfly aids in the evaluation of the Matroshka model under various architectures and in the exploration of the limits to portability of programs based on the model.

5 Summary

Most current parallel programming systems support a limited range of programming styles, which limits the effectiveness of programmers in implementing their applications. This paper presented a programming model based on *concurrent operations* on objects with an *early reply*, which allows programmers to use multiple styles of programming within a single computational framework.

Programming languages that provide parallelism with separate and distinct mechanisms for encapsulation force the programmer to bind parallelism too early, which inhibits the exploitation of possible concurrency. The Matroshka model is based on *uniform encapsulation*, which provides abstraction independently of parallelism. Likewise, separate mechanisms for sequential and parallel control flow also force early binding of parallelism. The model has *uniform control*, in which control constructs apply uniformly to both sequential and parallel control flow, and which enables powerful user-defined control abstractions. Uniform encapsulation and control enables programmers to delay the specification of actual parallelism within a program.

The Matroshka model provides communication via *operation invocation*. This, coupled with a *copy model* of variables and parameters, enables communication to scale with abstraction. Uniform encapsulation and scalable communication enables the programmer and language processor to exploit parallelism at many levels within a program. Because parallelism may be exploited at many levels, and be rebound among these levels easily, some programs may execute efficiently on many different architectures.

References

[Agha, 1986] Gul A. Agha, "An Overview of Actor Languages," *ACM SIGPLAN Notices*, 21(10):58–67, October 1986.

[Andrews *et al.*, 1986] Gregory R. Andrews, Ronald A. Olsson, Michael H. Coffin, Irving J. P. Elshoff, Kelvin Nilsen, and Titus Purdin, "An Overview of the SR Language and Implementation," Technical Report 86–6a, Department of Computer Science, University of Arizona, Tuscon, Arizona, 85721, June 1986.

[Atkinson *et al.*, 1983] M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chisholm, and R. Morrison, "An Approach to Persistent Programming," *The Computer Journal*, 26(4), 1983.

[BBN, 1985a] BBN Laboratories, Cambridge, Massachusetts, *Butterfly Parallel Processor Overview*, June 1985.

[BBN, 1985b] BBN Laboratories, Cambridge, Massachusetts, *Chrysalis Programmer's Manual*, June 1985.

[BBN, 1985c] BBN Laboratories, Cambridge, Massachusetts, *The Uniform System Approach To Programming the Butterfly Parallel Processor*, October 1985.

[Black *et al.*, 1986b] Andrew P. Black, Norman Hutchinson, Eric Jul, and Henry Levy, "Object Structure in the Emerald System," In *Proceedings of 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, September 1986, reprinted in ACM SIGPLAN Notices 21(11), November 1986.

[Black *et al.*, 1986a] Andrew P. Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, December 1986.

[Cook, 1980] Robert P. Cook, "*MOD—A Language for Distributed Programming," *IEEE Transactions on Software Engineering*, SE-6(6):563–571, November 1980.

[Feldman, 1979] Jerome A. Feldman, "High Level Programming for Distributed Computing," *Communications of the ACM*, 22(6):353–368, June 1979.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson, *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Greif *et al.*, 1986] I. Greif, R. Seliger, and W. Weihl, "Atomic Data Abstractions in Distributed Collaborative Editing Systems," In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, January 1986.

[Halstead, 1986] Robert H. Halstead, Jr., "Parallel Symbolic Computing," *Computer*, 19(8):35–43, August 1986.

[Ishikawa and Tokoro, 1986] Yutaka Ishikawa and Mario Tokoro, "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its features and Implementation," *ACM SIGPLAN Notices*, 21(11):232–241, November 1986, OOPSLA'86.

[Liskov, 1982] Barbara H. Liskov, "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering*, SE-8(3):203–210, May 1982.

[Liskov *et al.*, 1977] Barbara H. Liskov, Alan Snyder, R. R. Atkinson, and J. C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, 20(8):564–576, August 1977.

[Scott, 1987] Michael L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[Steele, 1984] Guy L. Steele, Jr., *Common Lisp: The Language*, Digital Press, 1984.

[Strom and Yemini, 1983] Robert E. Strom and Shaula Alexander Yemini, "NIL: An Integrated Language and System for Distributed Programming," Research Report RC 9949, International Business Machines Corporation Research Division, April 1983.

[U. S. DoD, 1983] United States Department of Defense, Washington D. C., *Reference Manual for the Ada Programming Language*, June 1983, ANSI/MIL-STD-1815A.

[Xerox, 1984] Xerox Corporation, Palo Alto, California, *Mesa Language Manual*, November 1984.

[Yonezawa *et al.*, 1986] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama, "Object-Oriented Concurrent Programming in ABCL/1," *ACM SIGPLAN Notices*, 21(11):258–268, November 1986, OOPSLA'86.