A Uniform Object Model for Parallel Programming

Lawrence A. Crowl
University of Rochester, Computer Science Department, Rochester NY 14627
crowl@cs.rochester.edu

This position statement presents a uniform object model of parallel programming that enables use of multiple programming styles, binding parallelism late, and porting programs among a wide variety of architectures. It then argues for concurrent operations on objects, as opposed to mutually exclusive operations on objects, based on orthogonality between encapsulation and concurrency. The final section argues for its relatively novel insistence on uniform control, based on the flexibility it provides the programmer.

The Matroshka Model

A single parallel application may have several subproblems, each suited to a different programming style. Current imperative programming languages tend to support only one of three programming styles based on loosely coupled messages, remote procedure calls, or shared memory. Given a task with subproblems appropriate to different styles, a programmer must choose inappropriate styles for some subproblems, which implies awkward, error-prone code, or attempt to mix programming systems, which represents a substantial interface problem. A few programming languages support multiple styles, but do so with separate mechanisms. This separation increases the complexity of the language and forces the programmer to manage the boundary between mechanisms.

Current approaches to parallel computation tend to provide separate facilities for parallel programming on top of a base sequential model. These facilities are typically a distinct notion of process, monitor constructs, messages, etc. This dichotomy between sequential and parallel aspects of programming forces programmers to bind the granularity of parallelism at program construction time. For instance, only specially defined program objects may execute concurrently with others. However, the best binding is often not known until after performance experiments performed on a working program. Altering the program for a different binding of parallelism entails a substantial amount of work and limits the effectiveness of programmers in exploiting multiprocessor hardware.

This paper presents Matroshka (Матрёшка)[1] a model of parallel programming that enables programmers to use several styles of programming within the same simple framework and enables programmers to bind the granularity of parallelism late in program development. A consequence of the late binding is that programs may be effectively ported among a wide range of multiprocessor architectures. The model is a synthesis of a few concepts and is minimal in that the loss of a single concept severely degrades its expressive power. See [2] for a more complete description. The model may serve as a base for a wide variety of possible programming languages.

Matroshka *objects* provide encapsulation at all levels, from simple integers to databases. The model uses a *copy model* of variables and parameters, as opposed to the reference model of Smalltalk [3]; objects contain other objects[2], not references to other objects. This approach reduces contention for objects and localizes communication with programs, which in turn enables effective implementation on shared-memory and message-passing systems. *Synchronous operation invocation* on objects is the sole communication mechanism; the caller waits for a reply before continuing execution. Operations may *reply early*, which allows the caller to continue concurrently with the callee; this is the sole mechanism for concurrency. Two invocations *execute concurrently* unless explicitly synchronized; the

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

[2] Just as the dolls nest within other dolls, Matroshka objects nest within other objects.

model relies on a language or implementation defined library of synchronization abstractions. The model provides sequential execution and first-class *closures*, which represent code to execute within existing variable environments. As in Smalltalk, the Matroshka model relies on a library of control abstractions to implement more sophisticated control. To aid in the definition of control abstractions, all operations take a *single parameter* and return a *single result*; records provide the effect of multiple arguments and return values. Programmers may communicate and store *operations as data* and *bind objects and operations* to form a port. Under Matroshka, closures are operations on other operation activations (environments), and so they also define ports. These features provide a measure of type independence in communication.

The Matroshka model enables a wide range of parallel programming styles. Programmers achieve a shared-memory style of programming by representing shared data structures as objects and relying on procedure-based, concurrent execution of operations to provide concurrent access to the structure. The remote procedure call style is similar, but the system implements operation invocation as a message send-and-receive pair. Programmers achieve message-passing styles by representing processes as objects, relying on operation invocation and the copy model of parameters to communicate the "message", and early reply to provide asynchrony. Operations synchronize within the object to provide exclusive message processing. Programmers achieve asynchronous invocation styles by creating a *future* object, using a port to describe the operation to be performed, and using early reply to delegate waiting for a result to the future object. A semaphore within the future object provides synchronization for retrieving the result. See the accompanying argument "Concurrent Data Structures and Actor Programming under the Matroshka Model" for more complete examples.

All control structures under the Matroshka model apply to both sequential and parallel control, enabling programmers to implement a given control abstraction with different levels of parallelism. Programmers choose a control abstraction during program development, then choose an implementation during performance analysis. For example, matrix addition has two nested iterations that require no synchronization. The programmer chooses an iterative control abstraction that makes no statement about whether iterations execute concurrently or not. Cheap sequential implementations for both iterations is appropriate on a uniprocessor. On a vector processor, the implementation may choose to parallelize only the inner iteration, while on a multiprocessor the implementation may choose to parallelize only the outer iteration. The model does not define the mechanism for choosing the implementation of an abstraction; it could be done by the programmer, a static analysis program, or the run-time system.

Concurrent Operations

One view of object-based concurrent programming is that operations on objects should have exclusive access to their object's data during their execution. The main reason for this view is that concurrent programming is "hard" and programmers should not have to, or even be allowed to, deal with concurrency. However, this approach couples encapsulation and concurrency, which means that encapsulating a portion of a program's data also implies synchronizing on that data, which in turn affects potential parallelism and therefore performance. With exclusive operations, programmers must consider how implicit synchronization affects performance when designing program abstractions. With concurrent operations, programmers must provide sufficient synchronization to ensure operation integrity. The latter approach is more flexible.

Exclusive operations violate Parnas's concept of transparency [4] for shared-memory multiprocessors, and prevent the implementation of generally useful abstractions such as wait-free queues, *etc.* Exclusive operations effectively make each object a monitor, which introduces the problems of nested monitors, such as deadlock due to nesting and the tricky semantics associated with non-blocking versus blocking waiting.

An object-based model for concurrent programming should support concurrent operations on a single object. This implies encapsulation is independent of concurrency because encapsulation does not imply synchronization. Note that this unrestrictive model does not prevent programming languages, tools, *etc.* from using a more restricted, "safer" style of programming.

Uniform Control

The case for uniform encapsulation has been stated before [1]. However, uniform control is just as important for parallel processing. Uniform control enables late binding of parallelism and general data parallelism.

Late binding of parallelism aids in porting parallel programs among different architectures and in improving performance of a program on a given architecture. For example, a program to find subgraph isomorphisms has both tree-search parallelism and vector parallelism. The appropriate point at which to place the actual parallelism is not clear without performance experiments. In addition, the appropriate binding point changes as the program moves from a distributed-memory multiprocessor to a vector processor. Providing efficient programs is considerably simpler if one can bind the actual parallelism late in program development.

Parallel programming usually distinguishes itself from distributed programming by utilizing data parallelism as well as functional parallelism. To support this, a parallel programming model must provide mechanisms to control the partitioning of work. This means that the programmer must be able to reference work indirectly, in addition to data. The desire for a uniform encapsulation coupled with the need to reference work implies that control constructs uniformly apply to parallel control as well as sequential control. For instance, an abstraction for iteration over a range of iterations could leave undefined whether it is executed sequentially or in parallel. The implementation may then choose most efficient form very late in program development. 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. However, 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, and parallel control is significantly harder than sequential control. This increases the cost of developing highly parallel programs and markedly increases the cost of changing the method of parallelism. This in turn inhibits the specification of extensive parallelism and especially data parallelism.

Summary

A model of parallel programming consisting of uniform encapsulation and control, concurrent operations on objects, synchronous invocation with early reply, closures, and a copy model of parameters and variables enables the programmer to use multiple styles of programming to specify extensive concurrency, and to bind the actual parallelism late in program development. It also enables efficient implementation of programs on a wide range of architectures.

References

[1] Andrew P. Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986. OOPSLA'86.

[2] Lawrence A. Crowl. A uniform object model for parallel programming. Distributed at the 1988 Workshop on Object Based Concurrent Programming and available from the University of Rochester Computer Science Department, September 1988.

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

[4] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, July 1975.