A Model for Parallel Programming

Lawrence A. Crowl

Abstract

Current parallel programming languages support only a narrow range of programming styles; force programmers to bind the specification of parallelism too early in program development; and limit the class of architectures for which they are effective. This paper presents a model for parallel programming based on uniform encapsulation and control, concurrent operations on objects, synchronous invocation with early reply, references to code in an environment, and a copy model of parameters and variables. This simple, coherent model enables programmers to use different styles of programming and to bind parallelism late in program development. In addition, the model may be efficiently implemented on a wide range of multiprocessor architectures.

1 Introduction

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 typically include a distinct notion of process, monitor constructs, and messages. The dichotomy between the constructs for sequential and parallel programming limits the effectiveness of programmers in exploiting multiprocessor hardware because parallelism is not available at all levels within a program based on the dual model. In addition, the choice of mechanisms tends to limit the style of parallel programming.

The remainder of this introduction distinguishes between parallel and distributed programming and describes three problems with current parallel programming systems. Section Two presents and justifies a model of programming that unifies the sequential and parallel aspects and is both simple and powerful. Section Three shows how the model addresses the problems with current systems. Section Four summarizes the paper.

1.1 Parallel Versus Distributed Programming

Distributed programs assume an environment in which the parallelism available does not exceed the number of relatively independent functions to be performed, *e.g.* file managers, transaction managers, screen managers, *etc.* In such an environment, programs can achieve sufficient parallelism by decomposition along functional lines. Thus, the main task of a distributed program language is managing concurrency, not exploiting it. On the other hand, parallel programs tend to assume an environment in which there is a small collection of highly coordinated functions to be performed on a large collection of data. In this environment, decomposition along functional lines does not provide enough parallelism. Parallel programming languages must be able to readily exploit significantly more of the parallelism available within a problem. In particular, they must be able to parallelize over the data available (*data parallelism*), in addition to possibly parallelizing along functional lines. This paper addresses parallel programming, as opposed to distributed programming. No aspect of the proposed model excludes its further development in support of distributed programming. Indeed, the model very naturally extends into distributed programming. However, this paper will not address the aspects of distribution.

1.2 Limited Programming Styles

Many parallel programming systems are available, supporting programming styles ranging from very loosely coupled messages (PLITS [Feldman, 1979]) to tightly-coupled shared memory (the Uniform System [BBN, 1985]). Most of these systems support only a single programming style. Different programming problems are often more naturally expressed using very different programming styles. These different problems often appear within a single application. With the available parallel programming systems, programmers must either use a single style even when awkward for the task, or use multiple programming systems.

The use of a single style unsuited to the problem at hand makes the program difficult to produce. For instance, the programmer may be forced to implement a stream filtering mechanism in a shared-memory model when it is most naturally expressed in a message-passing model. The multiple programming systems approach means that programmers must deal with several programming models, all of which have different mechanisms and constraints. This represents a considerable mental load because programmers must learn and remember the subtle interactions between different models. More importantly, managing the interface between different mechanisms in an application represents a considerable portion of the coding effort [Atkinson *et al.*, 1983].

An ideal 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, simple conceptual framework.

1.3 Early Binding of Parallelism

Many current programming systems provide separate mechanisms for the parallel and sequential portions of a parallel program. This presents programmers with two qualitatively different models of interaction between program components forcing them to bind the location and granularity of parallelism early in program development.

In a distributed system, communication between different processors costs typically two orders of magnitude more than communication within a processor. Many distributed programming languages, such as PLITS [Feldman, 1979], make distributed objects visible within the language under the assumption that programmers will manage visible costs more effectively than invisible costs. Programmers tend to make distributed objects large so as to minimize interactions between them. However, in multiprocessors, the difference in local and remote communication costs is much lower, which encourages more frequent communication within programs. The proper distribution and level of implementation for objects is not obvious before doing performance experiments on completed code. Choosing the granularity and location of parallelism should be more a part of the optimization effort rather than the algorithmic effort.

When parallel and sequential mechanisms are distinct, programmers must decide upon whether a given object will be implemented in a parallel manner or sequential manner early in program development. However, programmers may not know ahead of time which manner is appropriate for a given object. Under this situation, programmers may choose the higher-cost, more general mechanism [Greif *et al.*, 1986]. If a choice of a mechanism is incorrect, the programmer must recode the object and reintegrate it into the environment. General purpose abstractions must often be provided in both mechanisms in order to provide programmers with the incentive to use them. In addition, programmers using an object often must know in which mechanism the object is implemented. Because the use of an object may be distributed throughout the program, a change in mechanism could involve rewriting much of the program. Programmers will only perform such changes under extreme circumstances.

A programming model that enables the programmer to delay the binding of actual parallelism must be based on abstraction mechanisms that apply to both sequential and parallel execution.

Abstraction mechanisms are more important in parallel programming because abstraction affects performance.

1.4 Limited Architectures

One of the consequences of programming systems that bind parallelism with special language constructs is that many parallel programming systems work well on only one architecture because the underlying model assumes a specific, limited hardware configuration. The appropriate architecture is embedded in the level at which the language presents parallelism. In other words, these languages represent a means for programming a machine, not a description of a parallel solution.

A parallel programming language that provides for high portability of programs must enable description of parallelism at all granularities within a program. In order to use efficiently a wide variety of parallel architectures, programmers must be able to tune a program to an architecture without affecting the semantics of the source.

2 Model Description

This section describes the proposed model of parallel programming. This is a description of a *model*, not a description of a language. The model has a wide range of possible instantiations as programming languages. There are still many issues in language design that this model leaves unspecified, such as static or dynamic typing.

2.1 Uniform Encapsulation and Operation

The proposed programming model provides mechanisms for encapsulation and operation that apply uniformly to both parallel and sequential portions of a program.

Objects Provide Encapsulation

Many parallel programming systems have two mechanisms for encapsulation, one for parallel and distributed objects and one for local and sequential objects. This dual mechanism approach splits the programming environment into two qualitatively different models of interaction, introducing an artificial granularity in the program's parallelism.

Parallel programming systems need encapsulation mechanisms which apply uniformly to both parallel and sequential objects. The proposed programming model addresses these problems by providing a single encapsulation mechanism, the *object*. The object model provides natural encapsulation of abstract data types and scales effectively from simple integers to distributed databases.

Objects provide natural units for distribution. Distributed systems and non-uniform-memory-access multiprocessors have substantial performance differences depending on whether or not communication occurs within a processor or between different processors. To use such hardware effectively, program systems must limit the amount of communication between processors. Objects provide a natural destination for communication, and hence aid the programming system in reducing communication.

Synchronous Operation Invocation

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*. Several programming systems, *e.g.* Starmod [Cook, 1980], allow both synchronous communication (procedures) and asynchronous communication (messages), but usually require both sender and receiver to agree on the form. These systems have non-uniform communication.

Most computations initiate communication with the intent of receiving a reply. Message-based communication requires the programmer to filter incoming messages in search of the reply value. To support this, message-based programming languages, *e.g.* PLITS [Feldman, 1979], usually provide complex filtering mechanisms. In addition, message-based systems usually provide synchronous invocation for primitive operations, such as waiting on a message. Thus, message-based systems tend to be non-uniform. The complexity and non-uniformity of most message-based languages has lead to a greater concentration on synchronous, procedure-based communication. The evolution of the asynchronous, message-based ECLU [Liskov, 1979] into the synchronous, atomic-transaction-based Argus [Liskov and Scheifler, 1983] provides evidence for this trend.

Given these considerations, the proposed model provides synchronous communication in the form of operation invocation on objects as the sole communication mechanism. 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. The model does not provide asynchronous communication directly because it has a straight-forward implementation with other concepts in the model.

2.2 Copy Model of Variables and Parameters

There are two major models of variables and parameters in imperative languages. In the conventional value (or copy) model, variables contain values. Fortran, Algol, and its derivatives use this model. In the reference model, variables refer to objects. Smalltalk [Goldberg and Robson, 1983] and CLU [Liskov *et al.*, 1977] use this model. The desire for a uniform encapsulation mechanism implies that a parallel language must choose one model and stick with it. This includes variables for parallel abstractions, parameters, and variables local to an abstraction. Most imperative parallel languages provide a reference model for concurrent abstractions, but a value model for parameters and local variables.

The reference model has some attractive properties. Unfortunately, it has some serious disadvantages in the context of parallel and distributed programming.

aliasing: The reference model naturally provides for extensive aliasing among objects. It is not generally possible to determine *a priori* when two variables will refer to the same object. The result is that the programmer and language system must assume that the variables may refer to the same object. This inhibits parallelism. In contrast, the copy model ensures that each variable refers to a different object. Thus the programmer and the implementation are free to operate on two different variables concurrently.

contention: In multiprocessor systems, mutable objects may be a source of contention. Many programs will require access to the current states of objects, but not to their potential future state. To reduce contention for these objects in the reference model, the programmer has two options, to make the objects immutable or to explicitly make copies of the objects. Both cases increase the amount of dynamic allocation in comparison to the copy model. More importantly, these options represent an explicit coding of the copy model, but with higher run-time and programmer costs. An implicit copy model often requires less run-time support, and hence costs less.

remote arguments: The reference model implies heavy communication between machines as operations traverse back and forth across machine boundaries to reach objects referenced by the

objects passed as parameters. This potential inefficiency on distributed systems is such that Argus, which has in internal reference model based on CLU, uses an external value model. On the other hand, the Emerald system [Black *et al.*, 1986] uses the reference model over a local area network. Emerald achieves some measure of efficiency by moving parameter objects, and the objects they refer to, from the calling machine to the called machine. This approach fails under contention for the argument object. In contrast, the copy model moves all necessary information at the point of call. Communication occurs exactly twice, once sending the parameters, and once sending the results.

parameter optimizations: Parameters passed "by value" are more amenable to optimization than parameters passed "by reference". This flexibility is more important on multiprocessors because they tend to provide a much larger set of tradeoffs between passing pointers and doing physical copying. The reference model discourages "by value" parameters, and hence provides more limitations on parameter passing mechanisms. These limitations become important when porting programs among different multiprocessors.

garbage collection: Under the reference model, references to an object may spread freely. Because the existence of reference to an object implies the existence of the object, it is generally not possible to determine the lifetime of an object statically. The undeterminable lifetime of objects implies dynamic heap allocation and system-wide garbage collection. To perform garbage collection, the system must examine the entire set of references of the system to insure that no references to an object exist before deleting the object. This is possible on multiprocessors, but on very large or widely distributed systems, the number of possible locations for a reference is too large to be effectively searched. This issue is important because institutions invest in parallel programming for speed. They are willing to purchase performance with engineering effort. The reference model inhibits performance. On the other hand, the variables under the copy model contain objects. This enables static determination of the lifetime of an object, because the lifetime of the variable containing it is known, which in turn allows more efficient storage management.

The proposed programming 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 proposed 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.

2.3 Uniform Control

Parallel programming distinguishes itself from distributed programming by utilizing data parallelism as well as functional parallelism. To support this, a model for parallel programming must provide mechanisms to control the partitioning of work. This means that the programmer must be able to reference code in addition to data. The desire for late binding of parallelism coupled with the need to reference work, 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. The issue of programmable control abstractions is not unique to parallel systems, it is common to programming in general.

Single Argument and Result

Control abstractions will often need to delay the actual invocation of an operation, or apply an operation to a number of objects. In addition, control abstractions may need to combine the results of several computations. In order to describe general-purpose abstractions to perform such tasks, an intermediary must be able to handle arguments and results of operations as single units. Message based systems naturally provide this capability by referring to messages as a whole. RPC based systems do not provide a mechanism to refer to the set of arguments. (Suitable changes would enable RPC systems to refer to the set of arguments.)

The proposed model provides a handle for arguments and results by allowing exactly one parameter and one result for each operation. The effect of multiple parameters is achieved by passing a record as the single argument. If a language based on the proposed model provides *record constructors*, this approach can be as notationally concise as a list of parameters. Those methods that do not need an argument, or have no useful result, accept or return an empty record. Control abstractions need only handle the case of one argument and one result.

Object Reference and Operation Pair

One of the advantages of message passing systems is that they may be connected into rich networks where the nature of intermediaries is not necessarily known in advance. Passing references to neighboring objects allows effective construction of networks. However, if the operations must be directly named, each sender in a network must know the name of the operation of the recipient. This requires the sender and receiver to agree on an operation name *a priori*. This implies that programmers may have to place in the communication path many intermediary objects whose sole purpose is to translate operation names. Similar problems arise with static typing of message recipients. Network construction under these circumstances will be an *ad hoc* task.

The proposed model provides for object type and operation name independence with *ports*. A port is a first-class entity binding an operation to an object reference. An operation is invoked when an argument object is applied to a port, in direct analogy to sending a message. 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.

Blocks

In the implementation of a control abstraction, the abstraction must be able to accept a unit of work to perform. This work is usually defined in terms of the context of a procedure which called the control abstraction. For instance, the body of a for-loop is the unit of work passed to the for-loop abstraction. The local variables of the procedure in which the loop is embedded provide the context for the loop body. The most effective support of work within a context is represented by Smalltalk's *blocks* [Goldberg and Robson, 1983], or Lisp's *closures* [Steele, 1984]. Blocks are portions of code that have access to an operation's local variables, but may be executed by another object. This is similar to passing nested procedures in Pascal, but with the added notational convenience of being defined in-line. Blocks provide Smalltalk with substantial power. For instance, in Smalltalk the programmer may define an operation on trees that accepts a block and then executes the block for each element in a tree.

The need for blocks is even greater for parallel languages because of the need to provide many abstractions for generating parallelism. If there is no such control abstraction, each programmer must build *ad hoc* mechanisms for creating parallelism. 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. For instance,

without a handle for work within a context, users cannot define general purpose control abstractions, such as a parallel for-loop, without explicitly collecting references to the relevant portion of the environment.

The proposed model provides for work within a context with blocks. 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 provides 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 existence of the environment is identical to the existence of the object representing the environment. Any mechanisms for determining the existence of a normal data object also apply to environments.) The ability to pass blocks as ports provides the basis for control flow within the language. For example, boolean objects have an '`if`' operation which accepts a port (usually a block) to invoke if the boolean is true.

Operations as Data

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 timing and ordering of operations. For instance, consider an abstraction to apply an operation to all the elements in an array. If operations must be directly named at the point of call, it is not possible to build a general-purpose abstraction for distributing an operation over an array. *In such abstractions the operation to be performed becomes data.* Without mechanisms to support operations as data, programmers must construct such abstractions on an *ad hoc* basis, which in turn discourages the use of such constructs. Such constructs are essential to enable effective exploitation of data parallelism.

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 a 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.

2.4 Concurrency Management

Parallel programming necessitates maximum exploitation of the available concurrency. Because the appropriate boundaries between parallel and sequential implementations 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. Programmers can change the implementation of general-purpose abstractions by selecting a different implementation from a library.

Concurrent Operations

In a system with uniform encapsulation, many abstractions will necessarily be large. The programming language must enable extensive concurrency within a single abstraction and between an abstraction and its callers. Any limitation on the potential concurrency within the encapsulation mechanism will magnify itself at each level of abstraction.

Many parallel programming systems, *e.g.* Distributed Processes [Brinch Hansen, 1978], make processes the unit of distributed encapsulation. Since the encapsulation provides only one active thread of control, these systems will force programmers to abandon encapsulation in the higher levels of their systems in order to maintain an acceptable level of concurrency. The lack of dynamic process creation can inhibit effective programming of client/server relationships in those languages with synchronous communication mechanisms [Liskov *et al.*, 1984]. In addition, these systems cannot represent true concurrent data structures. The ability to support concurrent data structures is an important feature of shared-memory multiprocessors. Object-based systems, *e.g.* ConcurrentSmalltalk [Yokote and Tokoro, 1986], often avoid the pitfall of making processes the unit of encapsulation, but sometimes implicitly enforce mutual exclusion between operations on an object. Such systems will not take full advantage of the capabilities of shared-memory multiprocessors.

Several programming models, *e.g.* Eden [Almes *et al.*, 1985], allow concurrent operations within an object, but only after explicit dispatching. In explicit dispatching, the receiver explicitly indicates when the processing of a request begins. This explicit dispatching introduces a weak form of serialization in that the resources devoted to dispatching processes are limited. This serialization for dispatching limits the amount of potential parallelism, increases the latency of operations, and introduces unnecessary costs in many cases.

The proposed model provides concurrent execution of operations on objects. 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 the language or implementation defined object types that provide synchronization.

Asynchronous Processing

Many abstractions can solve the information needs of their clients long before all of the associated computations are complete. A mechanism for processing an operation asynchronously to the caller reduces the non-essential synchronization between processes. Allowing the programmer of an object to minimize synchronization with the external world increases the potential concurrency within a program.

The proposed programming model provides for asynchronous processing with an *early reply*. 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.* Early reply maintains the local state of the operation after reply, but without the cost of forced concurrent access to activation variables. Early reply also implies an independence between the invoker and the operation. The termination of the invoker does not imply the termination of the invokee.

Asynchronous Invocation

Not only must a parallel programming model handle requests in parallel, it must be able to make parallel requests and receive the result at a later time in order to fully exploit available parallelism. This capability is known as *asynchronous invocation*. The caller's use of asynchronous invocation must be transparent to the callee. Otherwise, the programmer must provide and name two different forms for each operation.

The programming model could introduce additional mechanisms for asynchronous invocation, but early reply, coupled with generic definitions, is sufficient to enable straight-forward definition of asynchronous invocation. We demonstrate this by using the proposed model to outline (in pseudo-code) the implementation of a *future*, which is an object that asynchronously executes an operation and allows the result to be retrieved later. A future is a generic definition, requiring the type of the argument and the type of the result.

```
    begin object 'future(argument,result)'
        'hold' is a variable of type result
        'ready' is a binary semaphore initially locked

        begin operation 'do', parameter 'request' is of type
                record 'action' is of type port(argument,result)
                       'data' is of type argument
                end record
            reply with an empty record -- this allows caller to continue
            assign result of 'request.data' applied to 'request.action'
                to the variable 'hold'
            signal 'ready' -- 'hold' now contains the result
        end operation

        begin operation 'wait', parameter is of type empty record
            wait for 'ready'
            reply with value of 'hold' -- return result of request
        end operation
    end object
```

The use of a future is straightforward.

```
    'async' is a variable of type future(integer,integer)
    apply record action is '3.add'
                 data is '4'
          end record to 'async.do'
    ... do other work ...
apply empty record to 'async.wait' and use the integer result
```

Often, many invocations can start simultaneously. For example, each object in an array may be initialized in parallel. This capability is implied by asynchronous invocation because programmers may use an array of futures and loop through the array starting invocations.

While programmers need asynchronous invocation, a sufficiently powerful programming model need not support asynchronous invocation as a primitive operation. The future abstraction, which provides for asynchronous invocation, has a simple implementation within the proposed model. In addition, such an approach encourages the implementation of different types of futures, each tuned to the needs at hand. For instance, possible future types are single asynchronous reply, first of several asynchronous replies (including timeouts), and priority ordering of replies as they arrive.

3 Problem Solutions

This section shows how the proposed model addresses the three difficulties with current parallel programming systems discussed in the introduction.

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 proposed model.

Shared memory systems are characterized by processes calling procedures concurrently on shared objects. This style is easily programmed under the proposed model by delaying the reply of an operation until the end of the operation.

```
begin object 'shared-memory-abstraction'
    ... object's variables ...

    begin operation 'procedure', parameter 'whatever'
        ... operation's local variables ...

        ... do work ...
        reply with return value
    end operation

end object
```

The implementation of message passing styles is slightly less convenient, but still easy. For this example, messages are processed serially. Each object replies immediately upon receipt of an activation, synchronizes the activation with other activations, and completes the computation.

```
begin object 'sample'
    ready is a binary semaphore which is initially unlocked

    begin operation 'send', parameter 'message'
        reply with empty record -- this lets the caller continue
        wait for 'ready'        -- this ensures mutual exclusion
        ... do work ...
        signal 'ready'          -- allow next invocation (message)
    end operation
end object
```

Invoking the `send` operation with a `message` argument on an instance of `sample`, is equivalent to sending a message.

3.2 Late Binding of Parallelism

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

For example, consider the following implementation of matrix addition.

```
for i in 1..n do
    for j in 1..n do
        a[i,j] := b[i,j] + c[i,j]
```

There are no dependencies between distinct iterations of any of these loops. In choosing an abstraction which 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.

```
for i in 1..n $sequential do
    for j in 1..n $sequential do
        a[i,j] := b[i,j] + c[i,j]
```

On a uniprocessor with vector hardware, the most efficient approach may be to parallelize the inner loop, but maintain sequentiality in the outer loop.

```
for i in 1..n $sequential do
    for j in 1..n $parallel do
        a[i,j] := b[i,j] + c[i,j]
```

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.

```
for i in 1..n $parallel do
    for j in 1..n $sequential do
        a[i,j] := b[i,j] + c[i,j]
```

Late binding requires that the programmers use abstractions that meet their sequentiality needs, but that do not greatly exceed their needs. This is because the possible implementations of an abstraction are constrained by the sequentiality it guarantees.

3.3 Multiple Architectures

The proposed model supports 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. Because the binding of actual processor-to-processor communication to object invocation can occur very late in program development, the model allows the programmer to tune a program to an architecture without affecting the integrity of the algorithm.

For example, consider executing a program on both a distributed, message-based system and on a shared-memory multiprocessor. On the distributed system, object invocation at higher levels of the programs abstractions would be implemented by messages across the communication network, and invocation at lower levels would be implemented by procedure calls. Expensive message traffic is reduced by using messages only at the highest levels of program. On the shared-memory multiprocessor, object invocation at all levels of abstraction would be implemented by procedure calls, except at the lowest level were the processor reads and writes individual machine words. Processor communication occurs with reads and writes, and avoids the expense of constructing messages.

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 Summary

A single program may have several programming problems, each suited to a different programming style. Current imperative programming languages tend to support only one programming style.

Given a task with problems appropriate to different styles, a programmer must chose inappropriate styles for some problems, or attempt to mix programming systems. Neither is an attractive solution. 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.

Abstraction mechanisms are more important in parallel programming because abstraction and parallelism interact. Abstraction affects performance, so the abstraction mechanisms must not inhibit parallelism. Programming languages that provide parallelism with separate and distinct mechanisms for encapsulation force the programmer to bind parallelism too early, which inhibits parallelism. The proposed 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 proposed 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, some programs may execute efficiently on many different architectures by rebinding the actual parallelism to different levels within the program.

Acknowledgments

Special thanks to John M. Mellor-Crummey and César A. Quiroz for their many helpful comments in the development of this paper.

This work was supported by United States Army Engineering Topographic Laboratories research contract number DACA76-85-C-0001 and National Science Foundation Coordinated Experimental Research grant number DCR-8320136.

References

[Almes *et al.*, 1985] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe, "The Eden System: A Technical Review," *IEEE Transactions on Software Engineering*, SE-11:43–58, January 1985.

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

[Black *et al.*, 1986] 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.

[Brinch Hansen, 1978] Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, 21(11):934–941, November 1978.

[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.

[Liskov, 1979] Barbara H. Liskov, "Primitives for Distributed Computing," In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 33–42. Association for Computing Machinery, December 1979.

[Liskov *et al.*, 1984] Barbara H. Liskov, Maurice P. Herlihy, and Lucy Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," Programming Methodology Group Memo 41-1, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1984.

[Liskov and Scheifler, 1983] Barbara H. Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[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.

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

[Yokote and Tokoro, 1986] Yasuhiko Yokote and Mario Tokoro, "The Design and Implementation of ConcurrentSmalltalk," *ACM SIGPLAN Notices*, 21(11):331–340, November 1986, OOPSLA'86.