12-1994

Toward Scalable Parallel Software: An Active Object Model and Library to Support von Neumann Languages

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Toward Scalable Parallel Software
An Active Object Model and Library to Support von Neumann Languages

George K. Thiruvathukal
Illinois Institute of Technology
High Performance Languages and Systems Group
Department of Computer Science
Chicago, Illinois 60616
gkt@iit.edu

Abstract

Scalable parallel processing has been proposed as the technology scientists and engineers can use today to solve the problems of tomorrow. Many computational Grand Challenge problems require between two and three orders of magnitude than can be provided with the scalable parallel hardware of the early nineteen-nineties. While hardware continues to become more scalable and cheaper, software is not advancing at the same pace and remains a very expensive part of systems development.

A great deal of emphasis on software technology to support scalable parallel processing is placed on von Neumann languages. One of two approaches is common: (a) augment the von Neumann language with explicit parallel constructs or (b) write super-optimizing compilers to "find" the parallelism in a von Neumann program. These two approaches appear to be useful at some level; however, this paper argues that software constructed using these approaches is not likely to scale very well, because an appropriate level of abstraction is not being used to solve the problem.

We propose a simple layered architecture for doing parallel processing. The outer layer is the composition layer. This layer is used from a von Neumann language to encode algorithms using standard building blocks (objects). The middle layer uses objects. These objects exhibit high potential for parallelism. In our application, we focus on multidimensional arrays. At the lowest level, Itinerant Actors is used. Itinerant Actors is an object model developed by Christopher and Thiruvathukal at IIT to support asynchronous message-passing between active objects with a number of other useful ideas.

Themes: Parallel Programming Systems, Distributed Computing, Scalable Parallel Algorithms and Implementations, Parallel Matrix Computation

Other Themes: Program Composition, Layered Architecture, Dataflow with Macro-Operations

1 Introduction

The principle goal of high performance computing research is to radically improve the performance of a given algorithm or program in proportion to the number of processors available at one's disposal. A desirable attribute, therefore, is that the performance of software scales up, given more processors.

Scalable parallel processing should be easy to achieve. After all, the available hardware continues to improve. A scalable parallel processing system can be purchased for under a million dollars. Once the purchase order is written, it should be academic to get the software up-and-running in parallel--no problem.

It ought to be simple to design and implement scalable parallel software, but it remains a task whose difficulty eludes even the best computational scientists. To a large extent, the challenge of implementing parallel software appears to be due to artificial complexity in systems software architecture. We now explore this point further.

One manifestation of artificial complexity is due to a lack of abstraction to solve the problem. The abstraction problem has a severe repercussion in that the mathematics involved in solving the problem is often lost in our implementation. In other words, we have code, but it is difficult or impossible to see the underlying algorithm or mathematics that looked so good while on paper. How is it possible to lose the mathematics?

The mathematics in most of our parallel programs is lost, because of our reliance on mechanisms over methods. The computational scientist is encumbered by the need to learn a lot of unnecessary details about programming languages and mechanisms. Instead of programming with objects such as vectors, matrices, tensors, and lattice structures, the computational scientist is preoccupied with the need to learn about compiler pragmas, granularity calculations, message-passing libraries, array layout algorithms, and compiler optimization theory.

Object-oriented methodology has been proposed as essential to the design and implementation of reusable software components. Despite the potential for reuse through objects, most applications of object technology remain trivial. Example applications include graphical interface libraries, fundamental structures libraries, and numerical class libraries; however, these applications are all well-understood. Prior to the existence of an object model, it was well-understood how to package such subroutines in modules. Despite the limited applications of object technology to solve real problems, we argue in this paper that objects can be vastly instrumental in managing artificial complexity. Apparently, object technology can help us to cope with the abstraction problem.

While the abstraction problem in its own right is a very serious problem, another problem is that we are in a constant struggle with the von Neumann model of computation. In the 1979 Turing Lecture, John Backus discusses the von Neumann bottleneck and its potential impact (negative) on parallel processing. We argue in this paper that it is possible to use von Neumann units of computation in a non-von Neumann style, but we argue that there are certain aspects of computing for which von Neumann languages are not well-suited. Augmenting von Neumann languages with compiler pragmas, explicit parallel constructs, message-passing "primitives" and relying upon heuristics (compiler optimizations) is not an appropriate level for exploiting task-level, medium-grain, and coarse-grain parallelism.

We think, however, that it is important to give a practical and theoretical treatment to the issue of von Neumann computing. The pragmatics are that von Neumann languages are here to stay. Computational scientists (and computer scientists) are fluent in the von Neumann languages. Von Neumann languages generally deliver better performance than their functional counterparts, particularly in the solution of numerical problems. FORTRAN 9x is proof that von Neumann programming will continue for many, many years to come.

It is possible to bring non-von Neumann capabilities to von Neumann programming environments. For a number of years, the author and others (mentioned in related work) have been investigating active objects and dataflow as methods of supporting high performance computing; however, the majority of these works have centered around new language designs, which does not provide an evolution path for real applications. A few approaches have been based on existing von Neumann languages, but migrating existing codes to these new languages remained an unsolved problem.

The issue of programming languages raises significant human factors issues. The failure to address these issues usually results in resistance to trying new programming languages and the near-term extinction of a programming language. A major human factors issue is religion. Religion influences our choice of a programming language more than any other issue. Language bashing has long been fashionable, even between people who use languages that have many of the same useful capabilities (e.g. FORTRAN and C). Usually, the differences are found in syntax.

A second human factors issue is that most people would rather use a familiar, proven language than an unfamiliar one. Our experience with languages has shown that it is better to propose language mechanisms, initially, through a library (class library) and at a later point introduce a set of language mechanisms as extensions to an existing language or a new language. If you have a user community, it is much easier to justify the use of a new programming language. Too much language work exists in a theoretical vacuum, and these languages usually become extinct before being used.

Another human factors issue seems very important in the parallel world. There is a warm feeling experienced by a software developer when the algorithm works without modification when moving from the sequential to the parallel universe. In the ideal case, it should be possible to develop an algorithm sequentially and run it either sequentially or in parallel and get the same results. This mandates the availability of a clean semantics model, regardless of whether a library or a language is provided. Sloppiness in the semantics model results in non-portable applications. Additionally, it results in software that cannot be easily verified, which we believe is a requirement for parallel software, because debugging and comprehension of algorithms is presently a serious problem.

Having motivated the issue of scalable parallel processing, the problems in achieving it, and the human factors that play a role in our often futile attempts to deal with the problems, what do we propose to do to effectively address scalable parallel processing?

In this paper, we will be discussing an approach for integrating von Neumann and non-von Neumann environments. An Active Object Model and Library are proposed to support object-oriented capabilities from ordinary von Neumann languages. The present implementation is designed to be used from C++, an object-oriented language which does not support parallel, or active, objects. A future implementation will use the object-brokering principle (and an object-brokering standard, such as CORBA) to allow procedural and functional languages to define and manipulate parallel active objects.

2 Organization of Paper

As a roadmap to reading and understanding this paper, this is an overview of the remaining sections.

Section 3, Scalable Parallel Software, attempts to bring some definition to the notion of scalable parallel software. The definition is more or less a collection of ideas. We discuss what ideas support scalable parallel software development and what ideas impede the same. A brief discussion of why our ideas are a framework for scalable parallel software is presented.

Section 4, Related Work, summarizes related work on coordination languages, message driven execution, dataflow, and abstracting the heterogeneous/homogeneous computing environment. This work has influenced a number of ideas in this paper.

Section 5, Active Objects Library/Language, discusses the process coordination and active object library (language) and the importance of dataflow. We will discuss how our library is capable of directly supporting dataflow computations from a von Neumann language.

Conclusions, future directions, and references will be presented toward the end of the paper.

3 Scalable Parallel Software

To understand why software is not scaling up well, it pays to discuss the trends toward scalable parallel hardware. Parallel hardware today is much better suited to scalable solutions than parallel hardware of the past for a number of reasons. First, parallel machines are being built from off-the-shelf

components (commodity parts) as opposed to proprietary components. This allows a better software development environment to be provided, since many operating systems and development tools can run on the new parallel machines. Second, hardware has always been modular in nature. Now it is more modular. A parallel system once represented a huge purchasing commitment, but now a parallel system can be purchased inexpensively and grow, as the needs of an organization grow.

The net effect is that parallel hardware has become more accommodating from the standpoint of software development. Parallel computers are of much more general applicability than in the past. As an example, many companies are looking at parallel machines to support better database throughput for multimedia applications. Relational databases, on a commercial scale, are available for some of the emerging parallel computers and boast support distributed data and transactions processing. In short, this trend means parallel hardware is able to support general applications and is no longer exclusively catering to computational science applications.

The availability of scalable hardware emphasizes the need to continue research and development of solutions for scalable parallel software. The notion of scalable parallel software continues to be elusory, because the parallel software solutions of today are lacking in a number of areas. Parallel hardware is elegant, by comparison to software in any event, and has modular structure. While some software exists, as discussed under related research, which supports modular software design, to a large extent parallel software lacks modular structure.

What is scalable software then? This paper proposes that objects can be helpful for managing task-level parallelism and process-level parallelism. This loosely-coupled architecture allows a unit of work to be executed on any available processor. The ability to define such objects from a von Neumann language allows a parallelizing compiler to uncover hidden parallelism in a unit of work to be performed.

Another attribute of scalable parallel software appears to be a decupling from message-passing libraries, compiler pragmas, and explicit parallel constructs. All of these items, while being appropriate at some level of abstraction, interfere with comprehension of the underlying algorithms. The work proposed here provides a level of abstraction above, but not beyond, these ideas to improve performance for applications. Application developers and end-users should not have to understand how to pack and unpack data structures to be passed in a message. These capabilities can be supported by a class library, such as TLC.

Finally, scalable parallel software can be defined in one of two ways. The first definition involves the addition of processing elements. As processing elements are added, scalable parallel software is able to exploit the additional processing power and achieve improvement with little or no loss of efficiency. This definition is somewhat liberal but fair. Many applications that benefit from parallel processing do not have high efficiency, but the problem is solved in dramatically less time than it would take sequentially; however, the software cannot be argued to be scalable if the efficiency is dropping off dramatically as we add more processors.

Another way of defining scalable parallel software is the partial or total replacement of a system. What if the processor speed doubles? What if the network speed triples? What if the memory bandwidth is improved? A scalable software architecture is able to adapt to such changes in performance parameters. If performance cannot be tuned, the software cannot be argued to be scalable.

4 Related Work

The work described earlier is based on a solid foundation of computer science literature. The relevant literature pertains to coordination languages, message driven execution, dataflow computation, and abstraction of the heterogeneous computing environment. These four areas have some degree of overlap, but the aim here is to classify a related research area according to the principle goals of the stated research.

Coordination Languages and Generative Computing

The seminal research in coordination models and constructs is CSP [11] (Communicating Sequential Processes) and Linda [8]. In both of these models, support is provided to synchronize processes. The essence is to provide a more dignified interface for coordinating processes than a low-level message-passing library and to support classical synchronization mechanisms. The Linda model is a major advancement for coordinating processes in its introduction of a tuple space which provides the application with a conceptually shared memory. Distributed Memo [14][16] is proposed as an alternative to Linda, which preserves the elegance and simplicity of the Linda model, and provides a clarification of the shared memory abstraction as merely a distributed table of unordered process queues. Additionally, the simplification is enhanced with better support for mapping table entries (a major struggle reported in actual Linda implementation). Much of our work in Memo is reused here to provide support for coordinating processes.

Message Driven Execution

Message driven computing and execution are proposed as another model of computation. All of the major works in this area center around variations on the actors model of computation. The essentials of actors are discussed in detail in [1][12]. An actor is an object in the purest sense. It has a mailbox, into which messages are deposited, and a name. An actor executes a script whenever it receives a message. The script is useful for modifying the local state of the actor and for issuing communications with other actors. The message-passing support provided by actors systems is asynchronous.

A number of actors systems are provided. We apologize for any omissions in this survey. Act [12] and ABCL/1 [22] are extensions to the functional language model to support actors. In these implementations, the mailbox concept is tightly coupled with the actor. Our approach is to decuple these principles. We will show how deadpanning the mailbox and actors principles give us a lot of flexibility for supporting process coordination (synchronous operations) and object coordination (asynchronous operations).

Mentat is proposed as an extension to the object-oriented model of computation aimed at providing easy-to-use parallelism for people who are not computer science experts [9]. Specifically, it is implemented atop C++. The major contribution of the Mentat research is to allow objects in an object-oriented sequential language (such as C++) to run in parallel model with the support of a translator to support dynamic dataflow graph elaboration [10]. Grimshaw alludes to a principle called macro-dataflow. This is a variant of dataflow to support coarse-grained dataflow (dataflow on arbitrary objects). Some difficulties in this research include the inability to map structures and to synchronize processes effectively. Additionally, the software developer must understand dataflow to exploit the capabilities of Mentat. In the literature on Charm, a criticism is levied against Mentat in that the cost of procedure calls is somewhat unclear. Additionally, the implementation of Mentat includes a large dataflow simulator, which for many applications presents performance and portability problems.

Charm and CHARM++ are proposed as enhancements to the actors model of computation [18]. The unit of computation in Charm is a chare. A chare is an actor which corresponds to a chare, an old English term for a chore, or a unit of work. In this model of computation, a chare is defined with a number of named entry points, similar to actors. When one of these named entry points is called, the semantics are similar to the basic actors model. Charm and Mentat provide extensive support for shared, or global, variables, but the necessity of these features is somewhat questionable. Linda and Memo, as discussed earlier, support the notion of a global shared memory adequately within the bounds of the model of computation. Supporting this notion in the model has the immediate benefit of not imposing ad hoc protection mechanisms on the global shared memory from the end-user and application developer's point of view.

Christopher has proposed Message Driven Computing as a model of computation based on actors [5]. A major contribution of this work is in isolating the performance issues of actors languages and applications employing the actors model to solve problems. MDC provides a lightweight execution model with locations (similar to mailboxes), behaviors (similar to scripts), and generalized message pattern-matching. The generalized pattern matching principle allows full support of the various forms of dataflow, and it does so efficiently. The other languages discussed here do not support general-purpose pattern matching, which in general requires an extensive currying of operators. The foundations for an object-oriented version of Message Driven Computing (so-called OO/MDC) are presented by Christopher in [6]. In OO/MDC, the objects have local state; however, the objects cannot be used as in other object-oriented languages (such as Smalltalk and C++). Furthermore, inheritance is not supported, which many argue as an essential ingredient in object language design.

Dataflow Model and Pattern Matching

Dataflow is a model of computation which has been the most inspirational research area for us. Many of the design decisions we have made in our computational model specifically address dataflow; however, we are more interested in dataflow as part of a suite of techniques available in one's programming arsenal. Work in progress has shown that many computations (iterative and irregular problems) perform extraordinarily well when expressed as a dataflow algorithm. Given appropriate computational granularity, a dataflow solution to such computations we have found to perform with near linear speedup.

Research on static dataflow [7] and dynamic dataflow [3] architecture exposed a number of problems implementing dataflow architecture effectively in hardware. In particular, the implementation of matching hardware and data structures imposes serious overhead in practice at the architecture level. At the hardware level, it is important to keep things "light." The model proposed in our paper addresses these problems, by imposing these responsibilities on our actor and mailbox abstractions, and includes the numerous value-adding aspects of dataflow: futures, incremental structures [2], and parallel loop operations. These details are the subject of an unpublished paper, but we will address each briefly when discussing the model.

Abstraction Levels

Talk about Snyder's XYZ levels. Aside from terminology, we are thinking of the same ideas.

The highest level is composition. Snyder discusses phase composition at the Z, or problem, level. In our scheme, a single process creates specifies a computation by brokering the services of existing objects. These objects are concurrent/parallel in nature.

The next level is concurrent evaluation. The objects brokered by the composition level cause the creation of itinerant actors. Itinerant actors are used to do one of the following:

- process coordination
- object coordination
- macro-dataflow (with virtual pattern matching)
- distributed and shared data structures (all objects)

Snyder speaks of a Y level, wherein a phase composes process units to achieve a parallel computation. In our model, we use the term process somewhat differently. Our processes are lightweight at this level and partially ordered as in dataflow. This allows a much higher degree of parallelism to be achieved, since we do not rely on operating system mechanisms to manage lightweight processes (in other words, we do not use threads either).

Snyder speaks of the X level, wherein a process composes sequential program units in a single address space. We are consistent in this interpretation at our lowest level. While we do not address this in detail in this paper, there is nothing to stop you from exploiting parallelism at this level. This is the level where we believe optimizing compiler technology is particularly applicable.

5 Active Objects Library/Language

5.1 Goals and Guiding Principles

To support the middle layer of the conceptual model, the concurrent execution layer, we have developed (and are continuing to develop) an actors model which incorporates many of the coordination mechanisms of the Memo model. Addtionally, we have added support to specifically address dataflow.

Rather than define a language at this point, we have chosen to define a class library which provides classes and an application programming interface for introducing and executing actors from a von Neumann programming language, such as C or C++. The API will be discussed shortly. The idea of borrowing object-oriented capabilities from another language (not necessarily object-oriented) is called object-brokering.

We believe this approach to providing parallel object-oriented capabilities for von Neumann languages (and implemented in von Neumann languages) allows course and fine grain parallelism to be exploited. Our emphasis is on course-grain parallelism (in the ability of the model to support macro-dataflow, job jars, distributed and logically shared data structures), but nothing precludes the possibility of exploiting fine-grain parallelism. Since our model is easily implemented in von Neumann languages, objects and behaviors which modify their local state can be optimized and parallelized very effectively. This is not just a lofty claim. As an object has a local state, and behaviors directly modify only the local state, optimizations can be much more effective.

5.2 Language Elements

This section presents the essential language elements of the actors model we have evolved to support concurrent active objects. The basis for this model is called Itinerant Actors, developed by Christopher and expanded to support the object model by Thiruvathukal.

Itinerant Actors is based on the Actors model, described earlier, and the Memo model. Memo is a model designed to support process coordination. Processes deposit and retrieve memos to/from mailboxs with varying degrees of blocking supported. This model is particularly useful for supporting a number of ideas, collectively called generative computing. It also has an efficient implementation, since the mailbox name can be hashed to map the mailbox onto a given processor. When memos are exchanged between processors, it is clear as to the cost of performing a given communication.

5.2.1 Generalized Naming and Structure Mapping

In Itinerant Actors, the folder is referred to as a mailbox. The mailbox is an abstract class that defines a protocol to be followed by all subclasses:

- hash - produce a hash value for yourself, which will be taken modulo the number of processing elements to compute a destination processor id.
- compare - compare yourself to another mailbox. Comparison is essential to manage collisions which may occur (two mailboxes produce a same hash value).

We really cannot resist the opportunity to talk about the benefits of structured naming here.

5.2.2 Itinerant Actors and Rock-Bottom Actors

Two additional abstract classes are provided in Itinerant Actors: Message and Actor. An Actor is also a Message. The essential operations[1] of Message include:

- encode - encode the message into a network representation.
- decode - decode the message from a network representation.

An Actor has only one essential operation in addition to the protocol defined by Message:

- script - a block of code to be executed when the Actor receives a message from a mailbox.

The Itinerant Actors framework defines rock-bottom actors as a basis for further discussion.

5.2.3 Evolving Pattern-Matching and Dataflow

The rock-bottom actors defined by the Itinerant Actors framework is useful to support a variety of data structures (Section 5.2.6, Distributed Data Structures) and synchronization mechanisms (Section 5.2.7, Synchronization Mechanisms).

The variety of programming techniques that can be supported by this model is an improvement over the basic actors model and Linda coordination models. We believe, however, that both of these models are inadequate to support dataflow in general.

MDC, Message Driven Computing, introduced pattern-matching of named messages. We have expanded the Itinerant Actors framework to support a the matching capabilities needed to support dataflow.

Abstract class Message is expanded to include a unique name, or ID, which can be used to do fast pattern matching. This ID can be combined with a matching operator to support a pattern-matching operation in an Actor in true dataflow style.

Abstract class Actor is expanded to include an additional essential operation, match, which enables an actor whenever a pattern of messages exists. There are two possibilities for

1. These essential operations are so-called pure virtual functions in C++. What this means: a class derived from this class must provide a definition for this function.

supporting matching:

- the actor is enabled and given one of each of the messages that caused the pattern to match (provided the match operator demands the message)
- the actor is enabled as above, but its execution is mutually exclusive to other actors that might be enabled in the same mailbox. This is important in dataflow, so that an enabled dataflow node can leave messages to suppress executions at the same mailbox (or cause totally different execution).

The support for generalized pattern-matching allows for a very efficient implementation of dataflow, particularly dataflow with macro-operations. In our implementation, a bit vector (bit set) can be used to match a pattern in constant order time, since our messages are indexed. We discuss the performance of dataflow in the context of a simp

5.2.4 Essential System Architecture

Put a figure here, maybe.

- end-user perspective - application is composed of objects. These objects may represent tasks, loosely-coupled objects, or domain-specific objects (with an actors implementation)
- application - a SPMD program. The architecture of the application is assumed to be fully distributed. The application may or may not be object-oriented. The application, however, is bound to an execution environment, which is a lightweight actors run-time package.
- actors execution environment - this environment is responsible for supporting the coordination and message-driven execution semantics described earlier. The environment is extremely lightweight. Maintains a queue of actors which are ready to execute. These actors, once ready, may execute on any processor. Presently, we execute them locally.

5.2.5 Application Programming Interface

The following Application Programming Interface (API) is provide to use the capabilities of the Active Object Model. This describes the core Itinerant Actors functions; the pattern-matching discussed earlier (see section 5.2.3) is not presented, due to space considerations.

send(Mailbox* mailbox, Message* message)

Deposit message into mailbox. If an actor is present in the mailbox structure, it can be scheduled for execution with message passed as a parameter to its script upon execution. If an actor posted a receiveCopy function, the actor is executed with a copy of message; message remains in mailbox on the message queue in this case or when no actor is present.

receive(Mailbox* mailbox, Actor* actor)

Deposit actor into mailbox. If a message is present on the message queue, the actor is scheduled for execution with the found message passed as a parameter to its script upon execution. If no message is present, the actor remains in the mailbox on the actors queue.

receiveCopy(Mailbox* mailbox, Actor* actor)

Identical to receive with the exception that the actor, when scheduled, will receive a copy of the message. Receive copy is convenient for the implementation of read-only, shared variables.

create(Mailbox* mailbox, Actor* actor)

Identical to receive but intended to make it clear a new actor is being introduced to the execution environment.

Message* get(Mailbox* mailbox)

Get a message from mailbox. If no message is available at the time the call is made, return NULL. This is a non-blocking function.

Message* getWait(Mailbox* mailbox)

Get a message from the mailbox. If a message is not available, this function will block until one becomes available. Used in conjunction with send, this allows the Linda and Memo computation models to be employed to do process-level coordination.

5.2.6 Distributed Data Structures

This section presents examples of distributed data structures that can be implemented using the Active Object Library discussed. Similar examples have been presented in a paper describing the Memo [14] programming model.

Named Objects

A mailbox that holds at most one actor can represent a dynamically allocated object on the heap. Instead of pointers to objects, we use a mailbox name object to refer to the actor.

Arrays

Arrays of shared objects may be created similarly. The element a[i,j] can be stored in a mailbox whose name is constructed as:

```
Mailbox* m = new Array2DMailbox("A",i,j);
Actor* a = new SomeActor;

actorEnv->create(m, a);
```

Recall from the earlier discussion of Mailbox and Actor as abstract classes. Array2DMailbox and SomeActor are classes derived from Mailbox and Actor, respectively.

The above example illustrates using the key name to construct a 2-dimensional array abstraction. The key name

will support up to 3-dimensional array abstractions.

Unordered Queues

A mailbox structure contains an unordered queue of actors and messages, so if order is not vitally important, processes and actors can communicate simply by passing messages through a mailbox.

Job Jars

An important use of an unordered queue is a job jar. The actors (and/or messages) in the job jar indicate tasks to perform. When ever a process creates more work to do, it drops actors (and/or messages) in the job jar.

It is often convenient to have one job jar for each process and one common jar for all. The individual job jars are used for operations that must be performed by a particular process.

Futures and I-Structures

A future ia an assign-once variable used to communicate between a producer and a consumer. Both the producer and consumer may run in parallel, with the consumer only being delayed if it attempts to fetch from a variable before is has been assigned.

An I-structure (an "incremental structure") is a collection, or array, of futures. I-structures were invented for dataflow [2]. In the Active Objects Model, any mailbox that will have only one memo ever placed in it may correspond to a future. The consumer executing a *receive* or *receiveCopy*, fetching from the mailbox will be delayed until the value has been produced.

5.2.7 Synchronization Mechanisms

In addition to distributed, logically-shared data structures, the Active Objects Model supports process synchronization mechanisms.

Locks and Shared Records

Shared records are accessed by getting them from their mailboxes, examining them and updating them, and putting them back. While the record is being updated, it's mailbox is empty, and any other process trying to access it will have to wait; the records are implicitly locked.

```
Mailbox* m = new NamedMailbox("Object");
Message* anObject;

object = actorEnv->getWait(m);
/* modifications to object */
actorEnv->send(m,object);
```

There is more than one way to implement a shared record. It could easily be done with actors, as well.

Semaphores

The simplest implementation of a counting semaphore is identical to a lock, except that to initialize the semaphore, a process places as many memos in the semaphore's mailbox as required.

Dataflow

Dataflow programming triggers execution of code when it's operands become available. This is supported by the pattern-matching of messages in a mailbox by an actor. An actor which performs a pattern-matching operation has exclusive access to the mailbox while doing the pattern-match. When the actor is enabled, it may or may not have exclusive access to the mailbox, depending on whether it needs to suppress further matches by other actors in that mailbox.

In addition to pattern-matching of messages, the model may be expanded to include an operator similar to the put_delayed operator in the Memo programming model. The put_delayed operator facilitates the implementation of basic dataflow, where general pattern-matching is not needed. Presently, the functionality of put_delayed can be supported by introducing an actor whose script performs a forwarding operation to another mailbox.

6 Implementation Status

A porting implementation of the Active Objects Library is available now which runs on any platform supporting C++. This implementation has a fully object-oriented design and ships with a C++ class library developed by the author called TLC (tools, libraries, and catalogues). The current implementation is sequential and can be used to design and test parallel algorithms today. A parallel implementation is being developed at the Argonne High Performance Computing Facility and is expected to be pre-released by the end of year.

A significant effort is underway in the High Performance Languages and Systems research group in the Department of Computer Science at the Illinois Institute of Technology to release a version of the Distributed Memo system. This release is built atop a class library called DOPPLER, which is intended to support the implementation of parallel systems by providing a number of classes to abstract the heterogeneous computing environment. An implementation of the Active Objects Library is expected to be released concurrently with Distributed Memo toward the end of year.

7 World Wide Web and Anonymous-FTP

A site has been established for the retrieval of products, papers, and other information related to the High Performance Languages and Systems Group.

The Anonymous-FTP site is ftp.rice.iit.edu. Check the directory /pub/research/parallel. There are subdirectories which contain packages (Active Object Library, D-Memo, MDC, and TLC).

The World Wide Web site is www.rice.iit.edu. The home page for the HPLS group is:

http://www.rice.iit.edu/hpls/hpls.html

8 Acknowledgements

This work would not be possible without the inspiration of others.

Thomas Christopher has been a constant source of inspiration and idea generation since 1988, when we first began working with compilers and parallel computating systems. His work on actors and message-passing environments is a living part of the present work. His insistence on exploring "crazy ideas" has given identity to this work and to the work of the High Performance Languages and Systems research group.

William O'Connell, a hard worker, has been a major source of inspiration in the area of distributed systems implementation. His class library, DOPPLER, will prove invaluable in the deployment of the Active Objects Library on a grand scale.

Scott Danielson has been a participant in the design, implementation, and configuration of the TLC library. His contributions in the area of configuration have greatly enhanced the potential to release a portable parallel and distributed object environment in the near future.

Ufuk Verün has participated in numerous discussions to expand the scope of the actors model. Many of his suggestions will be part of a future version of the model.

The Argonne National Laboratory Math and Computer Science (MCS) Division High Performance Computing Research Facility (HPCRF) has provided access to one of the best parallel computing environments available today, the IBM SP-X. This IBM SP-X is being used for the development of the first production version of the Active Object Library.

A special thanks to George Kutty, William O'Connell, and Ben Wong, who reviewed the drafts of this paper (under somewhat high pressure circumstances) and provided insightful suggestions for improvement.

9 References

[1] G. Agha and C. Hewitt. Concurrent Programming Using Actors. In A. Yonezawa and M. Tokoro, Object-Oriented Concurrent Programming. MIT Press. 1987.

[2] Arvind. "I-structures: An Efficient Data Type for Functional Languages", Technical Report LCS/TM-178. MIT. 1980.

[3] Arvind. "Dataflow Architecture", *Annual Reviews in Computer Science*. Volume 1, pages 225-253. 1986.

[4] J. Backus. Can Programming be Liberated from the von Neumann Style? 1979 Turing Award Lecture. Communications of the ACM.

[5] T. W. Christopher, "Message Driven Computing and its Relationship to Actors", *Proc. of the ACM Sigplan Workshop on Object-Based Concurrent Programming*" San Diego, CA. 1988

[6] Christopher, T. W. Early Experience with Object-Oriented Message Driven Computing. In Proceedings of the3rd Symposium on Frontiers of Massively Parallel Computing, October 1990.

[7] J. B. Dennis "Data Flow Architectures",*IEEE Computer*, November 1980, pp. 48-56.

[8] D. Gelernter. "Generative Communication in Linda", *ACM Transactions on Parallel Languages and Systems*, Vol. 7, No 1, Jan. 1985, Pages 80-112.

[9] A. Grimshaw "Easy-to-Use object oriented Parallel Processing with Mentat", *IEEE Computer*, May 1993

[10] A. Grimshaw, W. Strayer, P. Narayan "Dynamic, Object-Oriented Parallel Processing", I*EEE Parallel & Distributed Tech., Sys. & Apps.*, May 1993

[11] Hoare, C. A. R. Communicating Sequential Processes. Communications of the ACM. Volume 21, Number 8 (August 1978), pp. 666-677.

[12] Lieberman. Concurrent Object-Oriented Programming in Act I. In A. Yonezawa and M. Tokoro, Object-Oriented Concurrent Programming. MIT Press. 1987.

[13] Microsoft. OLE Specification. Where was it published?

[14] W. T. O'Connell, G. K. Thiruvathukal, and T. W. Christopher. Distributed Memo: A Heterogeneously Parallel and Distributed Programming Environment. In Proceedings of the 23rd International Conference on Parallel Processing, August 1994.

[15] O'Connell, Thiruvathukal, and Christopher. A Generic Modelling Framework for Building Heterogeneous Distributed Systems. In Proceedings of the 10th International Conference on Advanced Science at Technology, May 1994.

[16] O'Connell, Thiruvathukal and Christopher. The Memo Programming Language. To appear Proceedings of the International Conference on Parallel and Distributed Computing Systems, October 1994.

[17] Object Management Group. CORBA Sepcification Document

[18] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object-Oriented System Based on C++. In Proceedings of OOPSLA '93.

[19] L. Snyder et al. The XYZ Levels of Abstraction. Where was it published?

[20] Thiruvathukal and Christopher. Supporting Macrodataflow in MDC. Technical Report HPLS-91-001. Illinois Institute of Technology.

[21] U. Verün and T. Elrad, "Integrating Decision Controls in Concurrent Programming Languages." Submitted to ACM SIGPLAN '95 for consideration.

[22] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In A. Yonezawa and M. Tokoro, Object-Oriented Concurrent Programming. MIT Press. 1987.