Actors: a Unifying Model for Parallel and Distributed Computing

Gul A. Agha
Open Systems Laboratory
Department of Computer Science
Univ. of Illinois, Urbana-Champaign
Urbana, IL 61801, USA
Email: agha@cs.uiuc.edu

Wooyoung Kim [*]
Information and Computer Science
Univ. of California, Irvine
Irvine, CA 92717, USA
Email: wooyoung@ics.uci.edu

Abstract

Parallel computing and distributed computing have traditionally evolved as two separate research disciplines. Parallel computing has addressed problems of communication-intensive computation on tightly-coupled processors while distributed computing has been concerned with coordination, availability, timeliness, etc., of more loosely coupled computations. Current trends, such as parallel computing on networks of conventional processors and Internet computing, suggest the advantages of unifying these two disciplines. Actors provide a flexible model of computation which supports both parallel and distributed computing. One may evaluate the utility of a programming paradigm in terms of four criteria: expressiveness, portability, efficiency, and performance predictability. We discuss how the Actor model and programming methods based on it support these goals. In particular, we provide an overview of the state of the art in Actor languages and their implementation. Finally, we place this work in the context of recent developments in middleware, the Java language, and agents.

---

[*]The work has been done while this auther was staying at the Open System Lab. at the University of Illinois at Urbana-Chapmaign.

1

1 Introduction

Parallel computing and distributed computing share the same basic computation model: physically distributed processes that operate concurrently and interact with each other in order to accomplish a task as a whole. However, parallel computing and distributed computing have evolved as separate research areas: the differences between the two areas are a consequence of assumptions that each makes about the execution environment of programs.

In parallel computing, processes are assumed to be placed "closer" to each other and to communicate frequently – hence the computation/communication ratio of parallel applications is usually much smaller than that in distributed applications. Moreover, models of parallel programming typically assume that the processors and communication links are reliable and trustworthy. On the other hand, distributed computing focuses on processes that are "dispersed" in a wide area – i.e., communication between processes is assumed to be more costly than in parallel computing. For example, client/server applications typically involve limited communication between a client which sends a request and a server which does much of the processing. Moreover, research in distributed computing addresses issues such as unreliable and faulty hardware, security, and timing constraints. For example, mission critical applications in distributed computing employ fault tolerance mechanisms and real-time scheduling.

A number of recent trends point to a convergence of research in parallel and distributed computing. Perhaps the most significant of these trends is architectural. Three architectural trends may be noted. First, increased communication bandwidth and reduced latency make geographical distribution of processing nodes less of a barrier to concurrent computing. Second, the development of architecture neutral programming languages, such as Java, provides a virtual

computation environment in which nodes appear to be homogeneous. Finally, server machines in client/server computing are increasingly adopting multiprocessor architectures, often multiple processors with a shared memory in a single workstation and symmetric multiprocessors (SMP). While such architectures are less scalable than networks of computers, some concurrent programs with high communication traffic may execute on them more efficiently.

The second important trend which points to a convergence of parallel and distributed computing is the potential of Internet computing. With improvements in network technology and communication middleware, one can view the Internet as a huge parallel and distributed computer. Because connectivity on the Internet can be intermittent and the bandwidth variable, the ability of processes as well as data to migrate becomes critical. In turn, this requires a satisfactory treatment of mobility.

We believe that working with a programming model that addresses parallel, distributed and mobile computing uniformly is important to help address the challenge of developing software for real-world systems. Such a universal programming model must address at least four important concerns:

Expressiveness: not only development and reasoning about relatively complex programs needs to be simplified, but also their modifiability.

Portability: architecture-dependent specifics, such as the number or kinds of processors, the network topology, latency, bandwidth, etc., should be abstracted away from the code.

Efficiency: expressing algorithms in the programming model should not result in an unduly large execution overhead.
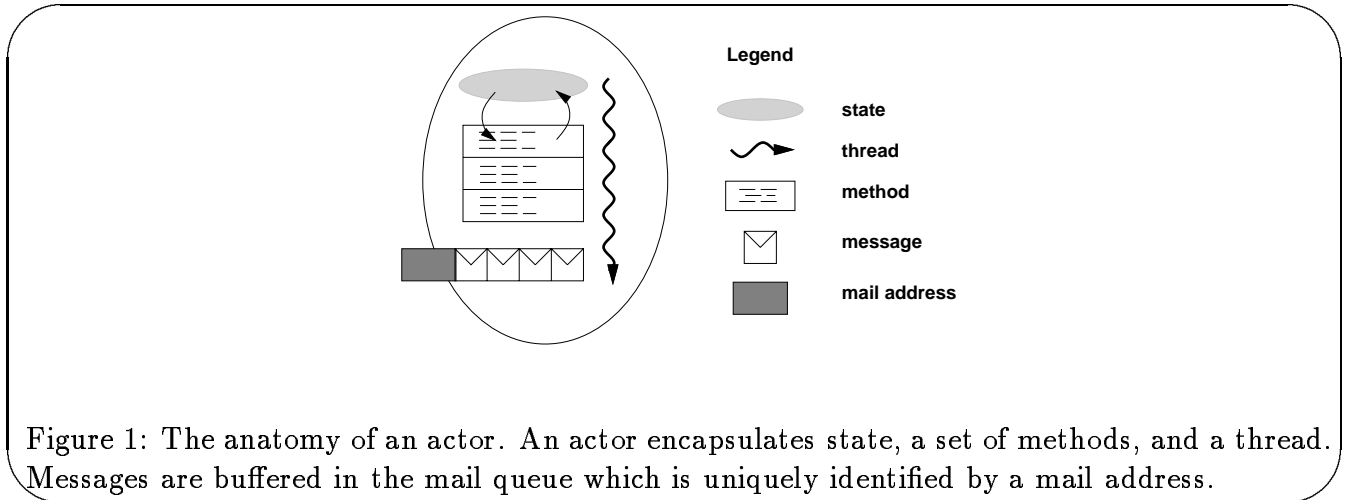
Performance predictability: programmers should be able to predict the relative performance of algorithms using a suitable parallel complexity analysis model.

The Actor model address these important concerns. In the Actor model, we think of parallel

and distributed systems as a composition of distributed, asynchronous components that are open to interaction with each other and their environment. A component may be software that is executing, or a physical device. We think of components as composed of a collection of autonomous entities called *actors*. For example, the processing elements (PEs) of a parallel computer may be modeled as actors.

The Actor model provides sufficient generality for representing a wide variety of computations. Specifically, actors are a natural way to integrate concurrency with objects. Beyond data abstraction, the Actor model provides a small number of primitive, atomic operators which simplify synchronization, name space management, scheduling, and memory management. In this paper, we review recent developments in research based on the Actor model and explore how it is used as a unified model for parallel, distributed, and mobile computing.

The outline of the rest of the sections is as follows. In Section 2, we define basic concepts of the Actor model and discuss typical language constructs for synchronization and communication that are used in high-level actor languages. In Section 3, we briefly sketch a parallel complexity model for actors. In Section 4, we describe some implementation techniques which allow high level actor languages to be efficiently implemented. Section 4 also briefly shows the results of experiments using the implementation techniques in an compiler and runtime system for an Actor language. In Section 5, we discuss the usefulness of the Actor model for distributed and mobile computing. In particular, Section 5 discusses the use of meta-architectures for customizing applications to address requirements such as heterogeneity and portability. We conclude the paper with a summary and discussion of current research in Actors.

Figure 1: The anatomy of an actor. An actor encapsulates state, a set of methods, and a thread. Messages are buffered in the mail queue which is uniquely identified by a mail address.

2   Actors as a unifying model

Actors are autonomous objects: they are objects in that they encapsulate data, methods, and an interface; and they are autonomous in that they encapsulate a thread of control (Figure 1). Actors interact with their external environment (represented by other actors) by sending and receiving messages. Thus, program execution follows the dynamic data flow implied by the messages – without imposing unnecessary synchronization overhead. Message reception initiates execution of the specified method with the message arguments. The execution of a method is atomic: once initiated a method executes without interruption by other messages to the actor. Note that atomic method execution does not preclude multiple active threads in an actor, as long as the execution of such threads is serializable. In response to a message, an actor may send messages, create actors, and make changes to its local data [1, 2] (Figure 2).

On the one hand, actors are similar to sequential objects in that they encapsulate data and procedures. On the other hand, actors differ from conventional concurrent objects in languages such as Java in two important ways:

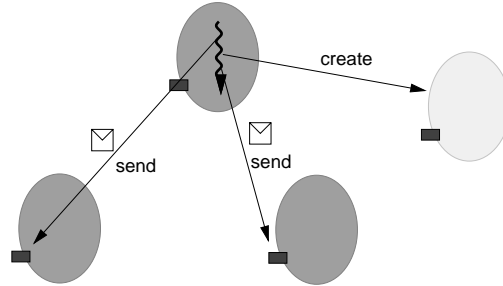- Although several threads may be simultaneously active in the actor, threads are prop-

5

Figure 2: The primitive operators in the Actor model. Actors may send messages, create new actors, and change its local state.

erly contained in an actor. By contrast, conventional concurrent objects are separated from threads and only serve as surrogates for communication and synchronization between threads. Such separation necessitates additional synchronization to maintain consistent object states.

- By using class variables, conventional objects may share a part of their state with other objects. Such data sharing further complicates distribution and synchronization. Unlike conventional objects, an actor may not share its state with any other actors.

The encapsulation of state and threads of control within an actor facilitates both the management of concurrency and location transparent distribution. The potential parallelism in an actor program is bounded by the number of actors in the system at any given time, the actual parallelism is the number of actors that are active.

2.1   Basic Naming, Communication and Synchronization

When an actor is created, it is given a globally unique reference called a *mail address*. The mail address abstractly represents the actor's whereabouts in a logical computation space. Messages

are sent using mail addresses and an actor may send a message only to those actors that it knows the mail addresses of. Thus, communication topology of actors is deterministic. However, mail addresses may be communicated through messages – thus causing the communication topology of actors to change dynamically.

Message passing between actors is asynchronous and non-blocking, and message receiving events are unordered except when they are causally connected. The nondeterminism inherent in actor programs necessitates coordination of interactions between potentially large numbers of actors. The need for such coordination introduces considerable programming complexity in concurrent systems. In order to simplify the task of writing concurrent programs, high-level programming constructs that abstract common patterns of computation and communication are needed. In fact, almost all actor-based concurrent languages provide additional communication and synchronization constructs to improve programmability [40, 11, 20]. We describe some of the basic constructs below.

2.2 Synchronization Mechanisms

Although asynchronous, unordered message passing is efficient, it does not always provide sufficient structure for geographically distributed, independently executing actors. For example, consider the parallel execution of a Gaussian elimination problem. Given that computations in processors advance at different rates and that messages are dynamically routed, messages from different rows may arrive at a node (or an actor) in an incorrect order. Enforcing a logically correct message reception order is one of the simplest examples of a coordination problem.

A message execution order may be enforced on actor computation by using *synchronization*

constraints. A synchronization constraint for a method specifies some property that must be satisfied by the local states of a group of actors when an actor accepts a message that invokes that method [13, 14]. In general, synchronization constraints are expensive, requiring complex forms of coordination that are enforced using meta-level actors [15]. The simplest and most common form of synchronization constraints is local synchronization constraints, where message acceptance is predicated only on the receiver's local state. Local synchronization constraints are relatively efficient to implement [21].

2.3   Communication and Naming Abstractions

Asynchronous point-to-point communication is a rudimentary form of communication; other communication abstractions, such as *remote procedure calls* (RPC), and *call/return communication* [22] may be implemented on top of asynchronous communication. Moreover, with high level naming abstractions, asynchronous communication may be used to implement a range of group communication abstractions. For example, a number of logically related actors may be grouped together and referred to as a unit; messages then sent to the group may be delivered to all members of the group (*broadcast*) [21, 20]. More generally, by abstracting patterns over names or attributes of group members, messages may be sent only to some subset of the group (*multicast*) [10]. Alternately, messages may be delivered to an arbitrary member of the group (*one-to-one-out-of-many*) [11].

Actor programming languages often provide such collective communication abstractions. The collective communication abstractions are implemented using asynchronous point-to-point message passing. The flexible communication and naming model in high-level actor languages makes

it easy for software developers to use common parallel programming paradigms such as data parallel and fork-join parallel executions.

3   Analysis of Parallel Complexity for Actors

A programming model is defined not only by specifying the operations that may be performed and how they may be combined, but also by a complexity model which tells us something about the relative performance of a program. Because the performance of an algorithm is strongly dependent on the architecture on which it is implemented as well as the placement and scheduling strategies used, accurately analyzing the complexity of parallel algorithms requires fixing the architecture as well as the placement of actors (data and processes). Thus realistically analyzing the complexity of a parallel algorithm. can involve an inordinate amount of detail. These difficulties partly account for why much of the analysis of parallel algorithms been done using unrealistic models of parallel computation.

In this section, we briefly sketch a method for estimating the computational cost of an actor program. In order to make such an estimate, we conservatively approximate the inherently asynchronous execution of actors with the assumption that the internal operations of actors are of synchronous. Such an assumption facilitates analyzing the complexity. Given that a large number of operations are typically performed on relatively homogeneous processors, synchrony in counting operations is a reasonable abstraction. However, we do not assume that the operations are synchronized in any way – rather we explicitly model the impact of communication latency.

A simple model for complexity analysis of actor programs would represent computation cost in terms of the number of messages communicated. Unfortunately, such an analysis model

does not accurately capture the performance difference of two implementations of an algorithm. Specifically, it does not take into account two important factors in message passing, namely, the overlap in message transmissions, and the cost difference between local and remote message transmissions. These two factors can result in a significant difference in complexity measures.

We extend the message-based complexity model with the following parameters to account for the two factors mentioned above: latency (L), bandwidth $(1/B)$, communication overhead (O) for message transmission, the number of processing nodes (P), the number of actors in the system (A), and a location function $\mathcal{L} : \mathcal{A} \to \mathcal{P}$ for actor placement [4]. In the rest of this section, we sketch the complexity model using an example. (For more detailed discussion on the model, please refer to [4]).

To illustrate our model, consider an implementation of a Cholesky decomposition algorithm based on outer product updates [16], where each element of the lower triangle of a matrix is represented as an actor. Figure 3 illustrates four computation phases (and their communication patterns) in the k-th iteration of the implementation. For simplicity, we assume all method executions take a unit time or *cycle*. This suffices because method executions in actor languages are usually fine-grained and take roughly a number of operations similar to that of a message delivery. Message delivery is counted as a part of a method's execution. For simplicity, we measure all parameters as multiples of a cycle and we also assume that all messages are one word long. Both these assumptions can be easily generalized by introducing finer-grained parameters for the times of various operations. This will change the resulting complexity by a constant factor.

Assume that we have as many processors as the number of actors (i.e., $N(N+1)/2$) and assign one actor to each processor using a location function $\mathcal{L}:A(i,j) \to \frac{i(i-1)}{2}+j-1$. The general analysis

(1) Iteration k (k=2). A(k,k) updates its value by the square root of A(k,k).

(2) A(k,k) sends its value to the actors below it.

(3) Each of A(j,k),j=k+1:N sends A(j,k)A(j,k) to A(j,j) for subtraction. Also, A(j,k) sends its value to the actors below it.

(3 cont'd) As A(i,k) receives A(j,k), k<j<i, it sends A(i,k)A(j,k) to A(i,j) for subtraction.
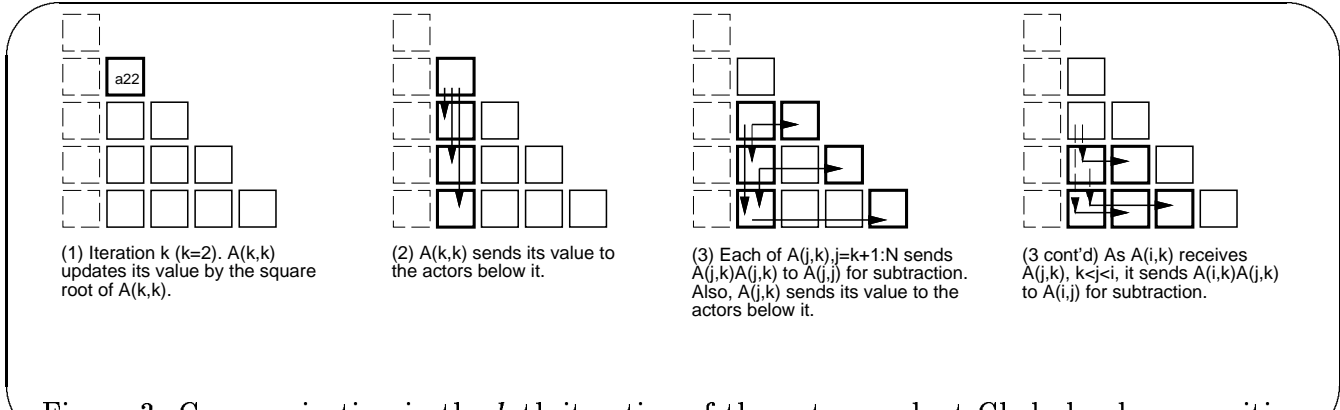
Figure 3: Communication in the k-th iteration of the outer product Cholesky decomposition.
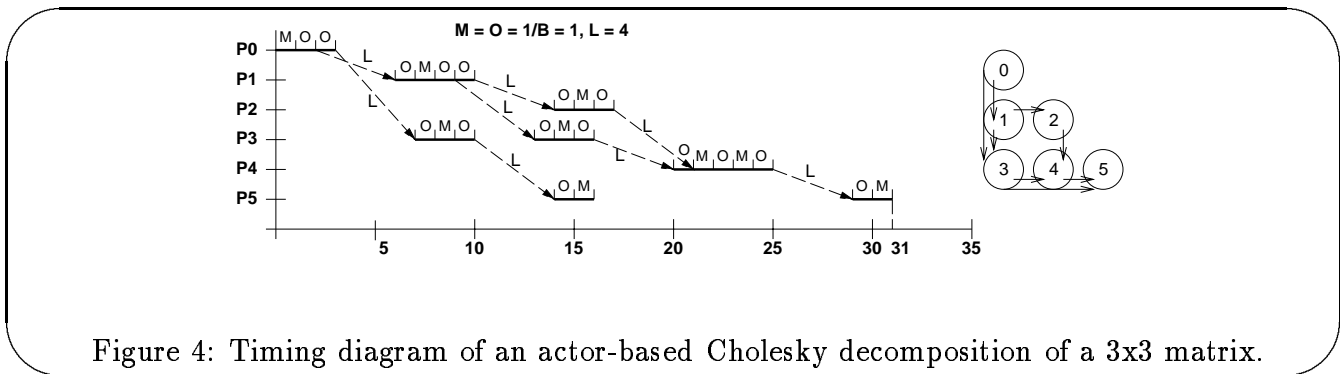

Figure 4: Timing diagram of an actor-based Cholesky decomposition of a 3x3 matrix.

of the implementation is quite involved, even given the simplification of global synchronization; we will show only an illustrative example below. Figure 4 is the timing diagram of the decomposition of a 3×3 matrix when messages are processed in first-come-first-served order and no global synchronization is assumed. By using local synchronization constraints, synchronization requirements may be implemented on a per-actor basis. The execution takes 31 cycles when $O=\frac{1}{B}=1$ and $L=4$.

On the other hand, assuming all the actors are synchronized after each iteration (global barrier synchronization) simplifies the analysis. Since we charge a cycle for method execution, step 1 takes one cycle for each iteration (refer to Figure 3). Step 2 for the k-th iteration takes $(N-k)O+L+O+1$ except the last iteration. The cost of step 3 is further divided into two components. First, each of $A(j,k)$, $j=k+1:N$, sends $A(j,k)A(j,k)$ to $A(j,j)$ (O cycles) and then

11

sends $A(j,k)$ to all the actors below it ($(N-(k+1))O+L+O$ cycles), except for the last two iterations. The costs are 0 and $O+L+O$ for the last and the second last iterations, respectively. The second component is the cost for an actor $A(i,k)$, $k+2 \leq i \leq N$ to compute $A(i,k)A(j,k)$ in response to a message $A(j,k)$, $k+1 \leq j \leq i-1$ (1 cycle) and to send it to $A(i,j)$ $(O+L+O$ cycles). Note that when m messages arrive at a node simultaneously, one of them experiences the longest delay of $(m-1)(2O+1)$ cycles before its reception. The longest delay for the k-th iteration is $(N-k-2)(2O+1)$ for $k=1:N-3$ and 0 for $k=N-2:N$. Thus, the execution time for a 3×3 matrix is 38 cycles when $O = \frac{1}{B} = 1$ and $L = 4$. Notice that the analysis with the implicit assumption of iteration synchronization results in larger estimation than that without such an assumption.

The above example illustrates how, by ignoring the possibility of overlapping communication and computation, a complexity model requiring a global barrier synchronization can result either in inefficiencies in execution or misleading relative complexity estimates. In fact, as the performance ratio of computation and communication continues to grow in network architectures, the global barrier synchronization model becomes more misleading.

4   Implementation Issues

The main advantage of using Actor languages is that they help programmers build systems at a higher level of abstraction: programmers use an abstract view of parallel execution embodied in a virtual architecture which hides unnecessary architectural details. The disparities between the virtual architecture and an underlying concurrent computer are resolved by a machine-specific runtime kernel. In addition to implementing Actor primitives, the runtime kernel provides the services that are necessary for actor execution; such services include name space management

and name translation, location-transparent message delivery, and scheduling. The rest of this section shows how architecture independence and efficiency are realized in implementations of Actor languages.

The runtime kernel should realize complete connectivity between actors. In most concurrent computers, complete connectivity between physical processors is provided by interconnection networks and their communication software. Thus the runtime kernel needs to implement only two additional services: first, the ability to locate a receiver using its mail address, and second, the ability to retrieve messages from the network and deliver them to their respective receivers. It is important that message delivery be done in such a way that references to actors are location transparent in programs.

The non-blocking, asynchronous communication model in Actors has three important consequences for the kernel implementation:

1. Messages need to be buffered at their destination because their arrival time is not predictable.

2. Actor communication is "non-blocking" – after sending a message, a sender need not wait for an acknowledgement or reply. Thus, the sender may execute concurrently with the delays in the processing of the messages that it has sent – often masking communication latency by overlapping computation and communication.

3. By scheduling multiple actors on a single processor, during the synchronization wait of one actor, other actors may be executed to optimize processor utilization.

Multiple actors that are scheduled on a single node compete for processing resources on that node. Semantic correctness, and in many cases efficiency, requires the runtime kernel to imple-

ment a *fair scheduling* mechanism – i.e., one that keeps some actors on a processor from starving out other actors on the processor by preventing them from being scheduled. Fair scheduling is also necessary to guarantee that messages destined for an actor will eventually be processed by that actor (assuming the actor's behavior allows it).

Different placement mechanisms typically generate different reference patterns and load distributions and finding a good compromise between reference locality and load balance is important to ensure efficient execution. The dynamic nature of actor creation and message-passing means that load may be distributed unevenly over the processors in ways that cannot be statically predicted. To allow imbalances in the load to be rectified, a runtime kernel should provide not only efficient remote actor creation but also actor migration facilities.

Our experience supports the conjecture that greater efficiency in program execution can be achieved by integrating the design of a runtime kernel with that of a compiler. The benefits of an integrated design approach have been argued in a number of studies [19, 21]. In one direction, a runtime kernel provides a range of implementations with varying cost characteristics, and if the compiler is aware of these implementations, it may produce code that selects the most efficient primitives at runtime. In the the other direction, if a runtime uses information that has been inferred by a compiler, the execution may avoid unnecessary or redundant computation.

4.1   The THAL Kernel Implementation

A number of runtime kernels have been implemented to support Actor computations [39, 33, 19, 23, 21]. Although these implementations provide essentially similar services, they differ in how much flexibility they provide. We describe a specific implementation that we developed for

```
                        APPLICATIONS

            ACTOR APPLICATION PROGRAM INTERFACE

                        NODE MANAGER

        Scheduler   Name Server   Memory Management

                    COMMUNICATION MODULE

                      NETWORK INTERFACE
```
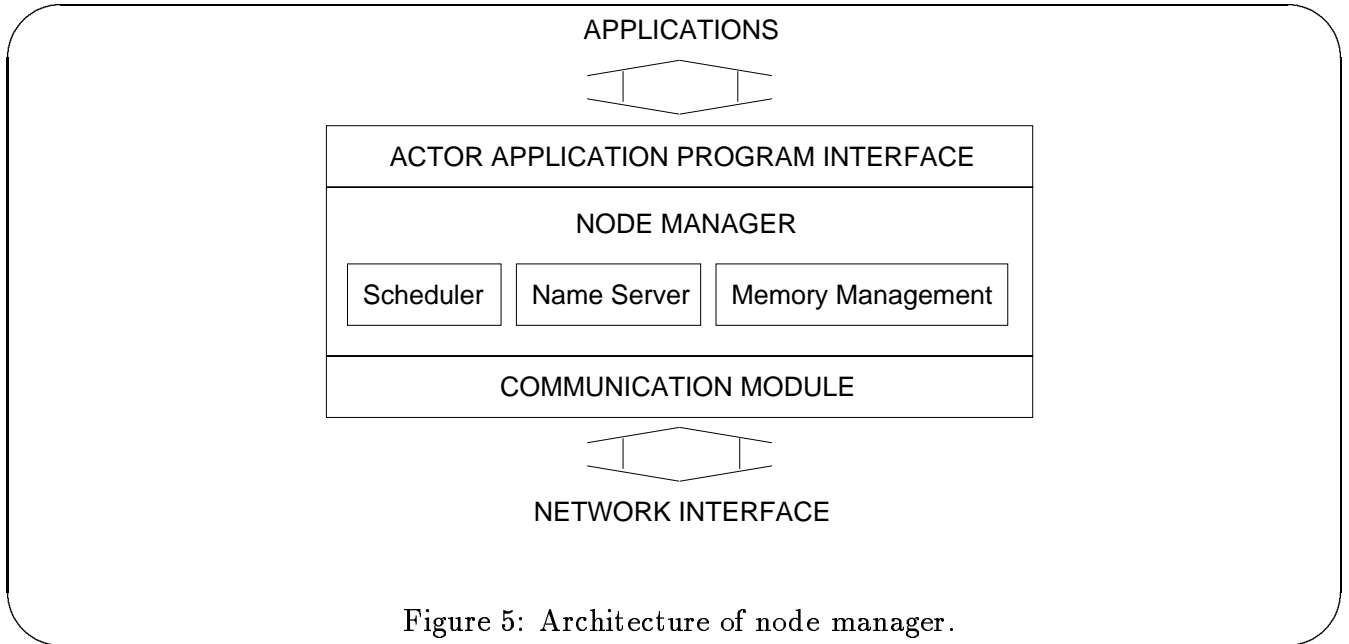
Figure 5: Architecture of node manager.

THAL, an actor-based concurrent language. We show how the implementation ensures architecture independence of application programs. The design of the THAL system also makes it easy to port the system to different architectures.

Figure 5 shows the architecture of the THAL runtime kernel. The kernel is designed as a three-layered system. The top layer provides the compiler with a well-defined interface. The interface is architecture independent and thus supports application portability. Architecture-dependent aspects of the implementation, such as message send/receive and input/output, are placed in the bottom tier. All other services, such as scheduling, memory management, and naming service, are implemented in the middle. Besides providing a uniform actor interface, the layered organization insulates the implementation of actor services from architectural specifics, making the runtime kernel portable.

The Node Manager module is responsible for handling remote requests, such as delivery of messages from remote nodes, remote actor creation, etc., while all the local services are performed in the execution context of the requesting actor. In addition to handling remote requests, node

15

managers communicate with each other to maintain a consistent state of the system and to enable dynamic load balancing.

The Name Server supports location transparent message delivery. Specifically, the module translates a logical mail address to an actual location. A number of naming and translation schemes have been proposed to allow different sorts of flexibility and efficiency characteristics. For example, the naming scheme in [33] uses physical attributes of an actor's implementation; this offers very efficient name translation but limits an actor's ability to relocate, making it difficult to do memory compaction after garbage collection and to load balance dynamically. A different naming scheme defines the mail addresses independently of Actor implementation. This sort of a scheme provides more flexibility but at the cost of name translation time [19]. THAL adopts a hybrid scheme where a mail address is defined using physical attributes of a *locality descriptor* which stores an actor's whereabouts. THAL's indirection in naming allows the runtime kernel to retain flexibility while keeping name translation efficient [21, 20].

The Scheduler module is implemented by taking advantage of two semantic properties of actors:

- All computation is message driven.
- The delivery order of messages is unconstrained.

Instead of implementing individual mail queues separately, the scheduler provides a shared message queue on each physical node. Each message now contains a reference to its receiver. All the messages sent to the node are scheduled in the message queue. By scheduling messages rather than actors, scheduling overhead is significantly reduced.

Furthermore, asynchronous unordered message delivery semantics allows us to schedule local messages differently from remote ones. The compiler generates a local and a remote version of the implementation of a message send. Using locality check provided by the runtime kernel, code produced by the compiler uses the appropriate version of the message send. Moreover, depending on the type information of the receiver expression, the compiler may implement a local message send with a function invocation, or even by inlining the call.

4.2   Performance measurements

The THAL kernel has been implemented on two different architectures: Thinking Machines's CM5 and a network of workstations (NOW). TMC's CM5 is a multicomputer (distributed memory multiprocessor) whose processing elements contain a 33 MHz Sparc processor. The processing nodes are connected with a fat-tree interconnection network [34]. THAL's communication module on the CM5 is implemented using the Active Message layer (CMAM) [38]. NOW is a network of Sun Ultra 2's, each of which hosts two 200 MHz UltraSPARC-I processors, connected by 10 MB/s Ethernet. The communication module on NOW is implemented using UDP. Table 1 shows performance figures for the primitive operations of the THAL runtime kernel on these two platforms. The results on CM5 are comparable to those of other systems, such as ABCL/onAP1000 [33] and Concert [19]. The large execution time of remote operations on NOW may be attributed to the communication overhead in the UDP layer and the large latency of Ethernet. We expect sizable improvement when the workstations are connected by a low-latency, high-bandwidth interconnection networks, such as Myrinet [27] or ATM [37].

Message scheduling accounts for a significant portion of overall overhead in actor execution.

		Local Creation	Remote Creation	Locality Check	Lsend & Dispatch	Rsend & Dispatch
TMC CM5	μsec	8.04	5.83(20.83)[†]	1.00	0.45/5.67[‡]	9.91
(33MHz)	cycles	265	192(687)	33	15/187	327
NOW	μsec	2.3	490	0.17	0.86	250
(200MHz)	cycles	460	98000	24	172	50000

Table 1: Performance of THAL run-time primitives. Local send and dispatch time does not include the time for locality check. Times obtained for the THAL runtime system are measured by repeatedly sending a null message. [†]The local execution of remote actor creation takes 5.83 μsec while the actual latency is 20.83 μsec. Thus, the actual latency is masked. [‡] Time using function call vs. time using local message scheduling.

We illustrate the message scheduling overhead by describing the results of computing the 33rd Fibonacci number. The THAL implementation of the Fibonacci computation that we used is very fine-grained: each method invocation executes only two message sends, one addition, and one reply. Thus, it serves as a good yardstick to measure the overhead. Executing Fib(33) on a single node of a CM5 took 60.7 seconds. An optimized C version on the same Sparc processor took 8.49 seconds. This is expected because we trade off flexibility with efficiency; for example, using message scheduling we may easily implement dynamic load balancing. However, our implementation is as efficient as other, less flexible systems. For example, evaluating Fib(33) using the Cilk system [8] takes 73.16 seconds on the same Sparc processor.

We have implemented a number of applications to test the efficiency of computation in the THAL system. To illustrate our results, we show the performance of one application, a systolic matrix multiplication known as Cannon's algorithm, on a CM5. The results are compared with those obtained from a Split-C implementation; Split-C is a C-based parallel programming system [12]. The performance results show that as the grain size is increased, the actor scheduling overhead is masked and performance improves (Table 2). Note that the use of local synchronization

	256x256	512x512	1024x1024
THAL	279	389	434
Split-C	305	390	413

Table 2: Performance comparison of THAL and Split-C using Cannon's matrix multiplication, a coarse-grain application on a 64-node TMC CM5 (unit: MFlops).

constraints reduces synchronization overhead significantly.

5 Actors in Distributed Computing

The emergence of low-latency, high bandwidth communication networks has made LAN-based cluster computing attractive [6, 37, 27]. The rapid growth in the World Wide Web and the availability of architecture-neutral languages supporting remote execution, in particular Java, have made WAN-based Internet computing feasible. Moreover, many corporations are deploying their own "Intranets" for internal client/server applications. In this section, we explore how the Actor model is relevant to traditional distributed computing applications.

5.1 Client/Server Computing

One of the latest developments in client/server computing is the emergence of component-based 3-tier architecture. In the 3-tier architecture, clients and servers run on different processors connected by a software middle layer, written in an interconnection language such as CORBA [25] or DCOM [24]. The traditional 2-tier architecture, in which the applications are executed directly on the local operating systems simply does not address the problems of heterogeneity in large enterprise environments. Many server applications, such as complex database management or business logic software, run on dedicated high-performance hardware. In other cases, they

may run on a symmetric multiprocessor (SMP), on a network of SMPs, or on a massively parallel computer.

The asynchronous and distributed semantics of the Actor model makes it a promising implementation technology for component-based distributed client/server computing. There are at least three advantages of using actors as described below. These advantages are improved performance, customization, and scalability.

First, implementing clients and servers as actor groups improves performance. Asynchronous communication between clients and servers and the software bus reduces synchronization overhead and network traffic, increasing system throughput. Because actors are self-contained and location independent, dynamically tuning performance by relocating server applications becomes easier. For example, the load of server computers may be dynamically balanced by remote creation, replication, or migration of applications [21, 20]. Alternatively, a system administrator can "drag-and-drop" the applications to different nodes. For example, applications may be migrated so that they are closer to clients.

Second, the customization required by Quality of Service (QoS) specifications can be facilitated by enforcing actor semantics on services provided by the middleware. Without disrupting services, policies for fault tolerance, reliability, security, etc., may then be dynamically added or removed using middleware [32, 7]. Requirements for timeliness of response, or other QoS parameters, may also be specified [29, 30].

Finally, the overall scalability improves. Adding or removing client actors results in only a slight impact on performance because message buffering essentially provides a funneling effect and fair scheduling guarantees graceful degradation of performance and response time (horizontal scaling). At the same time, architecture independence of actors makes migration of server

applications to larger and faster server machines or multiservers transparent (vertical scaling).

5.2 Mobile computing

Recall that the recipient (destination) of a message is specified using a mail address. The mail address is a location-independent, logical name that is unique to each actor. This makes actor migration relatively efficient for two reasons:

- Because actors refer to each other by location independent mail addresses, migrating an actor does not require updating the local state of all potential senders – which may be a large number of actors. Instead, name tables can be efficiently maintained to control the mapping of mail addresses to physical addresses.

- Because an actor does not share its state with other actors, consistency between the states of different actors need not be directly maintained. This simplifies migration to different, physically distributed locations. The ease with which actors can be migrated makes them a natural model for mobile agents.

Because the semantics of actors simplifies migration of both the state of an actor and the process it encapsulates, a growing application of Actors is in the area of agents. In fact, most agent systems are actually implementations of the Actor model which provide support for mobility [3], even though the implementors are often not aware of this. Agents (mobile actors) enable increased possibilities for large-scale coordination. Consider a couple of examples. As a system evolves, it may realize the need to use geographically distributed resources – either because the information (data) is available only on some particular sites, or because the system needs more computational resources. Mobile actors can localize the processing of queries – extracting the

21

information they need, or computing on a remote site and then, either proceeding to other sites using this information, or returning with it. Conversely, agents can be used to upgrade software at remote sites automatically by investigating current versions, requirements, etc., and applying what they learn from upgrading one site to the task of upgrading another related one.

5.3   Customizing Systems through Meta-Architecture

Developers of distributed applications must deal with diverse requirements, such as heterogeneity, scalability, timing, synchronization, and fault-tolerance. Such non-functional requirements are implemented by enforcing appropriate interaction policies, such as atomicity and first-in first-out scheduling. Addressing non-functional requirements not only complicates the development of distributed software, it also results in a greater dependence of distributed applications on the underlying hardware. Specifically, the correct or efficient software solutions to synchronization, coordination, and fault-tolerance problems often depend on the particular network and processor characteristics, as well as the semantics of the software components that are governed by the interaction policy.

The implementation of interaction policies can be quite complex. Existing techniques for developing distributed software require developers to intermix the code implementing the functionality in components with the code implementing the interaction of the components. Such intermixing significantly complicates the development of distributed software. First, such intermixing effectively prevents the testing and debugging of interaction policies governing the components independently from the components themselves. Second, it prevents reuse of the code implementing an interaction policy; because the code is intermixed, it cannot be used to

22

constrain the behavior of different components.

Our approach is to allow a meta-level specification and implementation of a component's interaction policies. Protocols are described as meta-level entities which can customize the components by changing the communication between them, by scheduling them differently, or by recording or restoring their local state. In response to communication events, protocols may add or remove messages, record or replace the state of an actor, or halt the processing of a group of actors. Using our approach, orthogonal design requirements may be implemented by independent collections of meta-level actors. Such meta-level customization is a generalization of standard middleware architectures which only provide flexible naming and communication.

An important reason why the use of meta-architectures integrates well with the Actor model is that actors provide a uniform message-passing semantics; the semantics is easy to generalize and may be used to model the interactions between actors as well as interactions between actors and the underlying system. We have studied applications of the use of meta-architectures to address problems of coordination [15], actor placement and migration for efficient parallel execution [28], resource management [36], interaction protocols [32], and real-time constraints [29].

6   Conclusion

The Actor model offers a number of advantages from a programming perspective. The object style encapsulation is useful for modeling real-world systems, while the autonomy of actors frees a programmer from the burden of explicitly managing threads and synchronizing them. The Actor message-passing semantics ensures serializability of actor invocations, eliminating the need to reimplement low-level synchronization primitives such as semaphores and monitors. Mobility

is facilitated by the autonomy of actors and their location independence. Moreover, the Actor model is sufficiently concrete to allow the estimation of a realistic measure of complexity of an Actor program on a given architecture.

An important recent development is the demonstration that actors may be efficiently executed on a variety of architectures. In particular, some of the techniques (e.g., statically transforming local message-sends to function calls) improve efficiency of local execution, while others allow actors to be implemented efficiently on distributed multicomputers [21, 20]. Other researchers who have developed systems for the efficient execution of Actor programs have obtained similar results (e.g. [33, 19, 18]).

Actors are a general model of concurrent computation. Actors can be used to provide an abstraction for communication, synchronization, and concurrency management between programs written in different languages and running on networks of heterogeneous computers. The programs are wrapped in actors and their inputs and outputs are treated as actor messages. An effective industrial application of an Actor language has been the Carnot project which used the Actor language *Rosette* for heterogeneous interoperability in enterprise integration [35].

Although designing and implementing a programming language allows greater control over optimizations for fine-grained concurrency, it is possible to develop an actor library in any language. Such a library abstracts the scheduling and message passing functionality of actors. Some early examples of such actor implementations are in Smalltalk [23, 9] and C++ [17]. More recently, Broadway [31] implemented in C++ provides support for not only the basic actor primitives, but also support for a sophisticated meta-architecture for customization. The *ActorFoundry*, is a platform supporting actors through a Java library [26].

Although this paper has focused on Actors as a prescriptive model, where the model is used

to guide the development of a flexible concurrent programming language, Actors may also be used descriptively. In this case, the model may be used to reason about arbitrary distributed systems by modelling the behavior of such systems as a collection of actors. The formal methods developed in the study of Actor systems are based on notions of data encapsulation, process identity, asynchronous operation, and asynchronous communication, notions that are common in distributed computing. Thus constructs in arbitrary distributed languages may be translated into an Actor language to understand their operational semantics.

The Actor formalism is relatively well-developed – a number of results about program equivalence and transformations have been derived to simplify reasoning about Actor systems [5]. Research into extensions of this work to model mobility, explicitly taking into account locality and resource use, are currently on-going. Related practical issues for mobile systems include intermittent disconnects, security against malicious incoming actors, and optimizing resource use in the presence of mobility. By providing a formal framework, the Actor model can contribute to an understanding of mobile computing.

Acknowledgments

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha and N. Jamali. Concurrent Programming for Distributed Artificial Intelligence. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, Introduction to Distributed Artificial Intelligence, chapter 12. MIT Press, 1998. To appear.

[4] G. Agha and W. Kim. Parallel Programming and Complexity Analysis using Actors. In *Proceedings of the third International Working Conference on Massively Parallel Programming Models (MPPM '97)*. IEEE Computer Society, 1998. (to appear).

[5] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 1996.

[6] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[7] M. Astley and G. Agha. Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management. In *Sixth International Symposium on the Foundations of Software Engineering*. ACM SIGSOFT, 1998.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, 1994.

[9] J.-P. Briot. Modèlisation et Classification de Langages de Programmation Concurrente á Objets: l'Expèrience Actalk. In *Proceedings of the Colloquium on Langages et Modèles á Objets*, Grenoble, France, October 1994. (Also published as LITP research report, No 94-59, Paris, October 1994.) In French.

[10] C. J. Callsen and G. A. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[11] A. A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. MIT Press, 1993.

[12] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, 1993.

[13] S. Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In O. Lehrmann Madsen, editor, *ECOOP'92 European Conference on Object-Oriented Programming*, pages 185–196. Springer-Verlag, June 1992. Lecture Notes in Computer Science 615.

[14] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, July 1993. LNCS 627.

[15] Svend Frølund. *oordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[16] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[17] D. Kafura, M. Mukherji, and G. Lavender. ACT++ 2.0: A Class Library for Concurrent Programming in C++ Using Actors. *Journal of Object-Oriented Programming*, 6(6):47–62, October 1993.

[18] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In Andreas Paepcke, editor, *Proceedings of OOPSLA 93'*. ACM Press, October 1993. ACM SIGPLAN Notices 28(10).

[19] V. Karamcheti and A. A. Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Supercomputing '93*, 1993.

[20] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997. http://www-osl.cs.uiuc.edu/.

[21] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Supercomputing '95*, 1995.

[22] W. Kim, R. Panwar, and G. Agha. Efficient Compilation of Call/Return Communication for Actor-Based Programming Languages. In *High Performance Computing '96*, pages 62–67, 1996.

[23] L. Lescaudron, J.-P. Briot, and M. Bouabsa. Prototyping Programming Environments for Object-Oriented Concurrent Languages: a Smalltalk-Based Experience. In *Proceedings of the 5th Conference on the Technology of Object-Oriented Languages and Systems (Tools Usa'91)*, pages 449–462. Prentice-Hall, August 1991.

[24] Microsoft Corporation. *Distributed Component Object Model*. http://www.microsoft.com/com/dcom.asp.

[25] Object Management Group. *Common Object Request Broker Architecture*. http://www.omg.org/.

[26] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment. Available for download at http://osl.cs.uiuc.edu/foundry.

[27] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processor. *IEEE Concurrency*, 5(2):60–73, April 1997.

[28] R. Panwar and G. Agha. A Methodology for Programming Scalable Architectures. *Journal of Parallel and Distributed Computing*, 22(3):479–487, September 1994.

[29] S. Ren, G. Agha, and M. Saito. A Modular Approach for Programming Distributed Real-Time Systems. *Journal of Parallel and Distributed Computing*, 36(1), July 1996.

[30] S. Ren, N. Venkatasubramanian, and G. Agha. Formalizing QoS Constraints Using Actors. In *Proceedings of Second IFIP International Conference on Formal Methods for Open Object Based Distributed Systems,FMOODS'97*, pages 139–157, July 1997.

[31] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996. TR UIUCDCS-R-96-1950.

[32] D. Sturman and G. Agha. A Protocol Description Language for Customizing Failure Semantics. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Computing*, pages 148–157. IEEE Computer Society Press, October 1994.

[33] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme of Concurrent Object-Orient ed Languages on Stock Multicomputers. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, pages 218–228, May 1993.

[34] Thinking Machine Corporation. *Connection Machine CM-5 Technical Summary*, revised edition, November 1992.

[35] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in carnot. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2), May 1993.

[36] N. Venkatasubramanian. *An Adaptive Resource Management Architecture for Global Distributed Computing*. PhD thesis, University of Illinois, Urbana-Champaign, 1998.

[37] T. von Eicken, A. Basu, and V. Buch. Low-Latency Communication over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–53, February 1995.

[38] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of International Symposium of Computer Architectures*, pages 256–266, 1992.

[39] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A New Software/Hardware Architecture for Object-Oriented Concurrent Computing on an Extended Dataflow Supercomputer. In *ICS '92*, pages 93–103, 1992.

[40] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.