Parallel Programming and Complexity Analysis using Actors

Gul Agha and WooYoung Kim *
Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{agha | wooyoung}@cs.uiuc.edu

Abstract

We describe Actors, a flexible, scalable and efficient model of computation, and develop a framework for analyzing the parallel complexity of programs written in it. Actors are asynchronous, autonomous objects which interact by message-passing. The data and process decomposition inherent in Actors simplifies modeling real-world systems. High-level concurrent programming abstractions have been developed to simplify program development using actors; such abstractions do not compromise an efficient and portable implementation. In this paper, we define a parallel complexity model for actors. The model we develop gives an accurate measure of performance on realistic architectures. We illustrate its use by analyzing a number of examples.

1. Introduction

A parallel programming model must address three important concerns: portability, efficiency, and performance predictability. Portability means that the model should, as far as feasible, abstract away architecture-dependent specifics such as the number of processors, network topology, etc. Efficiency means that expressing algorithms in the programming model should not result in unnecessary overhead on program execution. These two criteria imply that programs in the model can be implemented efficiently without making assumptions about specific architectural support. Finally, a parallel complexity analysis model is required to help programmers predict performance.

*This work was made possible in part by support from the National Science Foundation under contracts NSF CCR-9523253 and NSF CCR-9619522; by support from the Air Force Officie of Science Research, under contract AF DC 5-36128.

The commonly used formal models of parallel complexity are unsatisfactory for predicting the performance of massively parallel programs. The PRAM-based models unrealistically assume that processors operate in synchrony and access a global shared memory in unit time. The BSP model assumes periodic global synchronization, introducing unnecessary inefficiency and providing an inaccurate performance model. Models based on hierarchical memory can provide accurate performance analysis, but they make it difficult to write portable programs.

In this paper, we describe how an actor-based model for programming can be used to address these concerns. Actors are autonomous computing agents which interact with each other using asynchronous, buffered communication. Actors are a general model for parallel and distributed systems. For example, processing elements (PEs) of a parallel computer may be modeled as actors; each PE is autonomous and injects packets asynchronously which are buffered in the network interface of their destination PE.

Thinking of parallel programming in terms of actors has inspired a number of projects in architecture and software. Mosaic [3] and J-machine [8], are two examples of early fine-grain parallel computers which provide architectural support for programming actors. A number of actor-based, concurrent programming languages have also been implemented on general purpose architectures [26, 4, 23, 17].

The Actor model provides only a few primitive, atomic operators which simplify synchronization, name space management, scheduling, and memory allocation. However, considerable programming complexity is introduced by the involved interaction and the nondeterminism inherent in parallel and distributed programs. To simplify the task of parallel program development, high-level abstractions are needed to represent common patterns of computation and commu-

nication. In fact, almost all actor-based concurrent languages have additional communication and synchronization abstractions to improve programmability. We discuss some of these constructs in Section 3.

To be effective as a parallel programming paradigm, actor implementations need to be as efficient as other low-level programming models. From an implementation point of view, this can be factored into two concerns. First, actor primitives should be efficiently implemented in a runtime system. Second, high-level programming abstractions should be translated by the compiler into efficient code which is executed on the runtime system. We describe some efficient implementation techniques which address these concerns in Section 4.

It is generally a challenging task to accurately analyze the complexity of parallel applications given the asynchrony in communication. The difficulty partly accounts for why so little work has been done to develop a realistic parallel complexity model for actors. To develop a performance model with manageable complexity, an actor's inherently asynchronous execution can be conservatively approximated with some synchrony in internal operations. However, the impact of communication latency needs to be accurately analyzed. We briefly sketch a parallel complexity model for actors in Section 5 and illustrate it using a number of examples. A comparison of Actors with the other models is given in Section 6.

2. The Actor Model of Computation

In some ways, actors are similar to sequential objects: actors encapsulate a state and a set of methods that manipulate the state. Unlike traditional sequential objects, which allow shared class variables, an actor's state is not shared with other actors. The strict encapsulation of state in actors simplifies maintaining consistency between distributed objects without the use of expensive locking protocols or special architectural support. In addition, each actor defines a process with its own thread of control (Figure 1). The autonomy of actors offers logical distribution and concurrent execution.

Computation in actors is message-driven; in response to a message, an actor executes the specified method and subsequently sends zero or more messages, creates zero or more new actors, and changes its own local state (Figure 2). The receiver of a message is uniquely identified by a *mail address*; thus, an actor represents a unique point in a computation space.

Because actors are logically distributed, the natural form of communication between them is asynchronous;

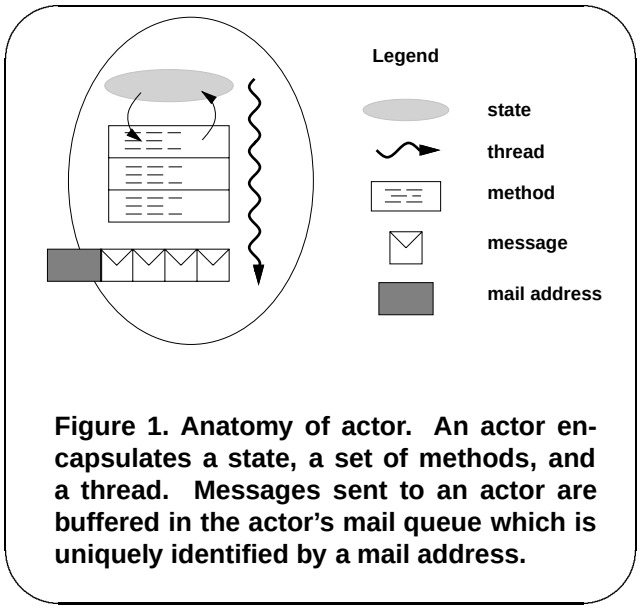

Figure 1. Anatomy of actor. An actor encapsulates a state, a set of methods, and a thread. Messages sent to an actor are buffered in the actor's mail queue which is uniquely identified by a mail address.

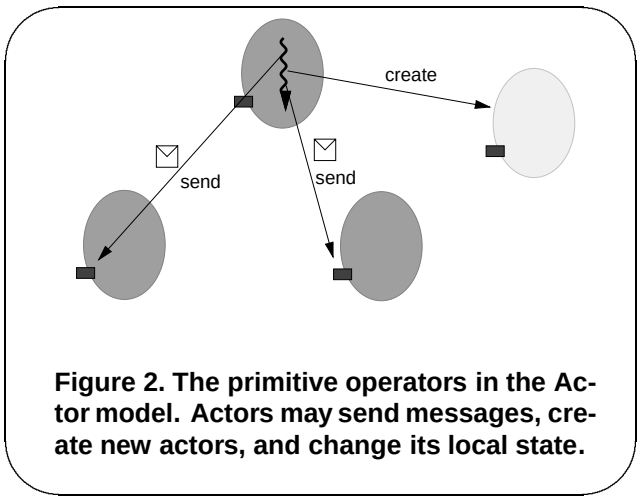

Figure 2. The primitive operators in the Actor model. Actors may send messages, create new actors, and change its local state.

upon sending a message, an actor does not wait for the receiver to be ready. Furthermore, the model does not enforce a specific order on message delivery, thus allowing dynamic routing. However, messages are guaranteed to be eventually delivered to their destination. In practice, the messages are buffered in their receiver's mail queue because the receiver may not be ready to process the messages when they arrive. The messages are then processed, one at a time, in the order of their arrival.

The semantics of eventual delivery is a form of fairness which can be used to reason about the liveness property of a system. Fairness combined with the uniqueness of mail addresses implies location transparent communication between actors.

Method execution in actors is atomic; once started,

a method executes to completion. Program execution follows the dynamic data flow – constrained by the messages in a program – without imposing an unnecessary synchronization burden. By contrast, some programming models may impose an unnecessary synchronization overhead because of the potential uncertainty in determining actual dependencies in concurrent program threads.

Actors allow programmers to express computation using communication between autonomous objects. Actors can thus use remote computing resources and *delegate* parts of a computation – sending messages to the place where computation is to be done. Typically, an actor performs some local computation and delegates parts of its continuation to other actors by sending those actors messages. By contrast, many parallel programming models are built exclusively around the "owner compute" rule, a legacy from the von Neumann model of sequential computing. A processor/process sends messages to acquire remote data and the peer processors are merely regarded as memory controllers.

Unidirectional, asynchronous communication in actors offers an effective mechanism to mask latency in many cases. Moreover, by partitioning computation cycles allocated by a processor to multiple actors, communication may be overlapped with local computation. There are two important considerations here. First, because asynchronous communication implies no-wait on message sending, an actor continues its method execution concurrently while messages are in transit. Second, because a consistent execution history of an actor is always kept in its state throughout program execution, the scheduler need not be concerned with saving execution context.

From a programming perspective, Actors offer a number of advantages. The object style encapsulation is useful for modeling real-world systems, while the autonomy of actors frees the programmer from the burden of implementing low-level synchronization primitives such as semaphores, monitors, etc. Moreover, asynchronous communication and encapsulation make it easy for software developers to use actors to emulate many well-exercised parallel programming practices, such as data parallel and fork-join parallel executions.

3. Higher-level Programming Abstractions

Although actors are useful to explicitly express parallelism available in applications, primitive actor operations are low-level. Thus, implementing common interaction patterns can be time-consuming and error-prone. Adding high-level abstractions for frequently used interaction patterns simplifies the development of parallel applications, improves their readability, and provides more opportunities for optimization [24, 15, 18]. We describe some of these programming abstractions.

3.1. Synchronization abstraction

Consider a problem of solving the Laplace equation $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = 0$ using the five-point stencil. Given initial approximations for $u^0_{i,j}$, at each iteration, the Jacobi iterative method computes the average of each point's four neighboring points from the previous iteration [22]. For simplicity, suppose each point is represented as an actor.

In an implementation of the porblem using only asynchronous communication, two methods, `push` and `compute`, may comprise an iteration (Figure 3). By executing the `push` method, each point sends its value to the four neighbors. Each message triggers the execution of the `compute` method; a point computes its new value when all the values from the four neighbors are available. It then starts the next iteration by sending itself a `push` message.

Not shown in the implementation is a requirement that messages from an iteration are processed before messages from the subsequent iterations. Such synchronization requirements are critical for correct execution, yet the basic Actor model does not provide any synchronization primitives beyond asynchronous communication and atomic method execution.

The need for actors to coordinate messages from different sources may be directly addressed by providing *local synchronization constraints*. Synchronization constraint is an abstraction which specifies under what conditions a message can be processed. A synchronization constraint is local when the conditions are specified with local state and message arguments only. For example, by maintaining information on the current iteration (i.e., `iter`) and passing the information in messages, the synchronization constraint

```
restrict push(v,i) when (i != iter);
```

can effectively enforce the synchronization required between messages in the implementation of the five-point stencil Jacobi method.

Observe that local synchronization constraints are easily implemented by adding a pending message queue to each actor and re-scheduling suspended messages whenever an actor changes its state. While the Actor model is flexible enough to represent such constraints, providing a programming abstraction to do

```
class Point {
  Point   north, east, west, south ;
  double  myvalue, v[3] ;
  int     n;
  ...
  method push () {
    north <- compute (myvalue);
    east  <- compute (myvalue);
    west  <- compute (myvalue);
    south <- compute (myvalue);
  }
  method compute (double v_in) {
    if (n == 3) {
      myvalue=(v[0]+v[1]+v[2]+v_in)/4;
      n = 0;
      self <- push ();
    } else {
      v[n] = v_in;
      n = n + 1;
    }
  }
  ...
}
```

Figure 3. An asynchronous implementation of the five-point stencil Jacobi method. Left arrow means asynchronous message sending: on the right hand side is the message and on the left hand side is the receiver. The message is composed of a method name and zero or more arguments.

so simplifies the programmer's task of expressing the constraints.

3.2. Call/return communication

Although the asynchronous implementation of the five-point stencil Jacobi method is efficient, it is somewhat difficult to understand. An easier-to-write implementation would be based on the owner-compute rule in which a point requests each of the four neighbors to return a value and then computes a new value. The interaction pattern in the program is succinctly expressed by using call/return communication (or remote procedure call) (Figure 4.a).

Call/return communication causes the sender to block until a reply is received. This blocking semantics implicitly serializes execution of a call/return communication and its continuation. Without the call/return communication abstraction, the compute method can only be implemented by explicitly passing a continuation to the neighbors (Figure 4.b). Note that both implementations also need to synchronize messages from different iterations.

```
class Point {
  ...
  method compute () {
    myvalue = ((north.value()+east.value()
               +west.value()+south.value())/4;
    self <- compute ();
  }
}
```

(a) with call/return communication

```
class Point {
  ...
  method send_requests () {
    north <- value (self);
    east  <- value (self);
    west  <- value (self);
    south <- value (self);
  }
  method value (Point to) {
    to <- compute (myvalue);
  }
  method compute (double v_in) {
    if (n == 3) {
      myvalue=(v[0]+v[1]+v[2]+v_in)/4;
      n = 0;
      self <- send_requests ();
    } else {
      v[n] = v_in;
      n = n + 1;
    }
  }
}
```

(b) without call/return communication

Figure 4. Two implementations of the five-point stencil Jacobi method with the owner-compute semantics. In (a), dot represents call/return communication.

3.3. Group abstractions

Data parallel execution is one of the most exercised programming paradigms in parallel computing. Data parallel programs may be represented in actors by creating a number of homogeneous actors and broadcasting messages. However, it is tedious to emulate the interaction patterns in data parallel execution using only actor creation and point-to-point message passing. Abstracting common interaction patterns in data

parallel computation and providing them as primitives offers a seamless way to integrate data parallel execution with task parallel execution, while at the same time providing an opportunity for optimization.

Group abstractions are of two kinds: group creation and group communication. Group creation abstractions take the type and the number of member actors to be created together with their common initialization arguments, and create member actors according to a specified placement policy. If a placement policy is not specified, a default one is used. For example,

```
MemberClass.grpnew [10] ();
```

will create 10 actors of the type `MemberClass`, place the actors using the default placement, and return a value (i.e., *group name*) that refers to the actors collectively. Group communication abstractions enable collective communication (i.e., *broadcast*) which uses group name, in addition to point-to-point communication between two member actors. Individual member actors are named by indexing their group name with a number.

3.4. Placement abstractions

Efficiency and scalability of a parallel implementation are determined by how a computation is partitioned and distributed. The optimal partition and placement strategy for an application is architecture-dependent as well as application-specific, as such strategies usually have to trade-off locality and load balance. Moreover, automatic detection of optimal strategies is generally a computationally intractable problem. However, in some cases, a programmer may have a good idea of what partitioning and distribution strategy works better for a particular problem. Thus, to obtain the best performance, it should be possible for a programmer to specify such strategies.

In actor systems, partitioning and distribution strategies are determined by actor placement. In general, actor placement may be specified statically or dynamically. In the static case, location information is passed when an actor is created (i.e., *remote* creation is possible). However, note that the creation of actors may be dynamic. Moreover, an actor may be moved during program execution.

4. Implementation Issues

The use of the Actor model helps application developers write architecture-independent programs by taking an abstract view of parallel execution while hiding unnecessary architectural details. One way to think of this is that programmers write their programs on a virtual machine using an unbounded number of processors, i.e., *actors*, which are fully connected. The virtual machine is then embedded on an underlying parallel machine by application-specific placement before execution.

The disparity between virtual processors (i.e., actors) and actual processors is resolved by machine-specific runtime support called a *node manager*. A copy of the node manager is placed on each actual processor; the node manager services requests from local and remote actors. In addition to implementing the actor operators, such as actor creation and message send, the node manager provides three important services for actor execution:

1. To support location independent message passing the node manager provides information on the whereabouts of actors (i.e., *name service*). Note that keeping track of the up-to-date location of each actor in the presence of migration is quite involved.

2. The node manager receives a message from the network and delivers the message to its receiver. The message delivery is done transparently even if the receiver is relocated before the delivery. (An efficient mechanism for this has been developed in [18].)

3. The node manager implements a scheduling policy. A typical actor program creates more actors than available processors. Consequently, multiple actors are assigned to a processor and these actors compete for limited computation resources. To allow "balanced" progress in co-located actors, the node manager needs to implement a fair dispatcher (i.e., scheduler). The fair dispatcher keeps a processor from being monopolized and thus guarantees that any delivered message is eventually processed.

Figure 5 shows the architecture of a node manager that is designed as a three-tier system. The top tier is a uniform application program interface common to all platforms. Architecture-specific services, such as message transmission and file I/O, are placed together in the bottom tier. All other core services, such as scheduling, resource management, and naming service, are sandwiched between them. This tiered organization insulates implementation of runtime primitives from disparities between different parallel architectures, thereby offering portability.

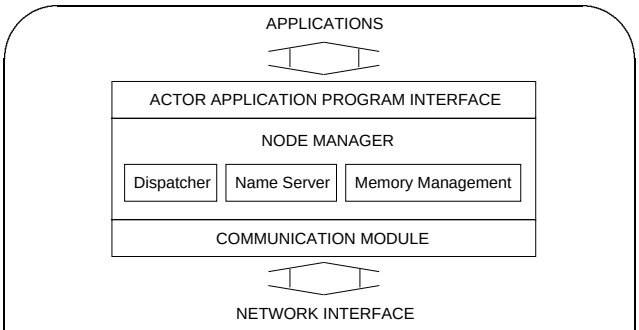

Figure 5. Architecture of a node manager.

The efficiency of executing actor programs is dependent on the implementation of the node manager. To support efficient execution, the node manager needs to be designed to interact with a compiler closely [15, 18]. First of all, the node manager provides a range of primitives with varying cost characteristics; the compiler must use the cheapest one to implement a construct in a given circumstance. Moreover, runtime primitives implemented in the node manager exploit information inferred by the compiler to reduce unnecessary or redundant computation. Finally, the compiler and the node manager collaborate to transform high-level abstractions to semantically equivalent efficient implementations [19].

For example, the compiler performs a data dependence analysis and transforms mutually independent call/return communications to one-sided message sends, which overlap the executions of different call/return communications. The replies from the receivers are redirected to a join continuation which is efficiently implemented by the node manager. The compiler also performs a data flow analysis to determine which methods may be executed through function invocation. At execution time, the locality of the receiver of a message which will invoke such a method is examined by using a facility provided by the node manager; if the receiver is local, the method is executed through function invocation [17].

5. A Parallel Complexity Model for Actors

In order to write efficient implementations on parallel computers, programmers need an accurate, easy-to-use method for analyzing the computational cost of a program before execution. We present a parallel complexity model that provides a method for estimating the performance of actor programs on distributed

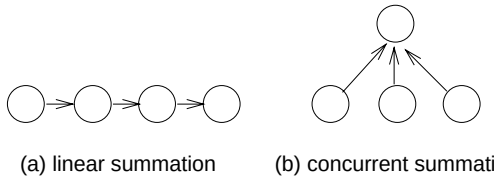

Figure 6. Two summation examples.

memory architectures. Then, we sketch how to adapt the model to shared memory architectures. Note that we will focus exclusively on time taken by computations rather than their space requirements (which are simpler to compute and usually less critical).

Our discussion considers only distributed memory computers with a finite number of processor/memory modules (or nodes) connected by a network. Each node allows a potentially unbounded number of actors to be executed (by time-slicing).

5.1. Parallel Complexity Model

A simple way to analyze the complexity of an actor application is to express the cost in terms of the number of messages communicated. However, this metric cannot differentiate between two implementations of the same logical algorithm in terms of efficiency: for example, it does not take into account possible overlap in message transmissions. Even more basically, the metric is not able to distinguish between local communication and remote communication, although remote communication introduces significantly higher delays due to communication latency.

We illustrate these limitations by means of a simple example. Consider two implementations of the summation of n numbers, each represented as an actor. We will call the two implementations linear summation and concurrent summation. Assume that the numbers are in some order and distributed over several processors. In linear summation (Figure 6.a), each actor receives a partial sum from its predecessor and sends its successor the result of adding itself to the partial sum (with the exception of the first and the last actors). If we charge unit time for a message, the parallel complexity of linear summation is $O(n)$. Concurrent summation designates one of the actors as an accumulator and lets the others send their number to the accumulator (Figure 6.b). Although the message transmissions may overlap so that the communication latencies are not additive, parallel complexity, as measured by the number of messages, is still $O(n)$.

To take into account both the overlap in message

transmissions and the cost difference between local communication and remote communication, we treat actors as if they were processors and parameterize the message-based complexity model using the following parameters:

L is an upper bound on the *latency* incurred in sending a message from a node to another.

$1/B$ is the smallest interval between consecutive message transmissions or message receptions. B is the available per-processor communication *bandwidth*.

O is the *overhead* defined as the length of time that a node is engaged in message transmission or message reception to/from the network.

P is the number of processing nodes.

A is the number of actors in the system.

$\mathcal{L} : \mathcal{A} \rightarrow \mathcal{P}$ is a location function which takes a mail address to a node. $\mathcal{A}$ is the set of mail addresses and $\mathcal{P} \subset N$ is the set of the node identifiers.

For simplicity, we will assume a unit time for method execution cost and call it a *cycle*. This suffices in our examples because method executions in each example are fine-grained and take roughly the same number of operations. We will measure all parameters as multiples of the cycle. Also for simplicity, we assume that all messages are one word long. Both of these assumptions can be easily generalized. Note that we include in the cost of method execution the time for a node manager to put a message into the mail queue of the receiver (message delivery overhead).

The refined model shows the cost difference between the two summation implementations. The configuration of the system is $P{=}A{=}n$. Using the refined model, the cost of the linear summation is $(n{-}1)(O{+}L{+}O{+}1)$ because only one message is in the network at any time. The cost of concurrent implementation, however, is $O{+}L{+}(n{-}1)(O{+}1)$ when $\frac{1}{B}{<}O$ because all the messages are sent concurrently.

The model also accounts more accurately for the cost difference between local communication and remote communication. Suppose we change the configuration such that $2P{=}A{=}n$ and use $\mathcal{L}{:}i{\rightarrow}i/2$ as the location function to assign two consecutive actors on the same node. The cost of the linear summation drops by approximately one half (i.e., $(\frac{n}{2}{-}1)(O{+}L{+}O){+}(n{-}1)$). In contrast, the cost of the second implementation reduces only slightly to $O{+}L{+}(n{-}2)(O{+}1)$ when $\frac{1}{B}{<}O$ because only one of the remote messages is converted to a local one.

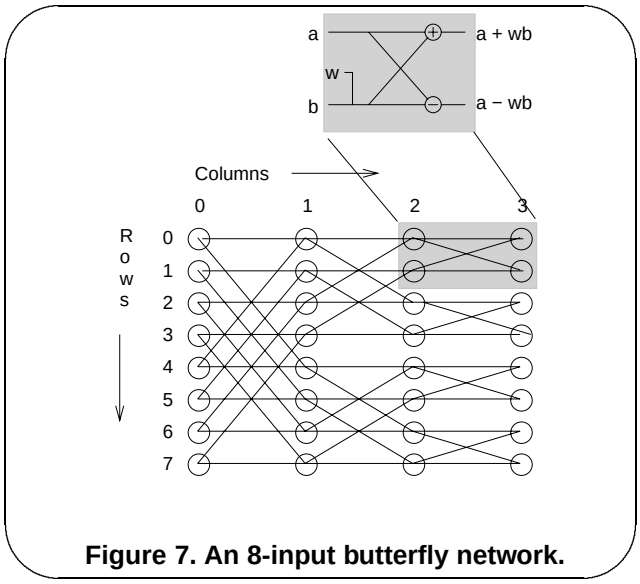

Figure 7. An 8-input butterfly network.

We further illustrate the parallel complexity model by analyzing actor implementations of a parallel fast Fourier transform (FFT) algorithm and the Cholesky decomposition. Complete analyses of the parallel complexity of these examples are fairly complex, particularly when we consider the overlap between computation and communication. Because of space limitations, we present only illustrative cases here.

5.2. Fast Fourier transform

Consider the "butterfly" algorithm [6] for the discrete FFT problem – so called because the algorithm has the same computation graph as the butterfly network. An n-input (n a power of 2) butterfly network is a directed acyclic graph consisting of $\log n$ stages. Each stage consists of $n/2$ butterflies executed in parallel. For $0{\leq}r{<}n$, and $0{\leq}c{<}\log n$, the node (r,c) has directed edges to nodes $(r,c{+}1)$ and $(\overline{r},c{+}1)$ where $\overline{r}$ is obtained by complementing the $(c{+}1)$-th most significant bit in the binary representation of r. Figure 7 shows an 8-input butterfly network. Inputs are given to the nodes in the leftmost column and outputs come out of the rightmost column. Each non-input node represents a complex multiplication and an addition (or subtraction); we charge a cycle for the two operations.

Suppose we have an n-input butterfly network for FFT which is to be realized on a P node parallel computer. For simplicity, we assume that $n{=}kP{=}A$, $k{\in}N$ and each row of the butterfly network is assigned to an actor. A natural placement strategy is a *cyclic* layout – the first actor is assigned to processor 0, the second actor to processor 1, so on. With this layout, the first $\log \frac{n}{P}$ columns of the butterfly network

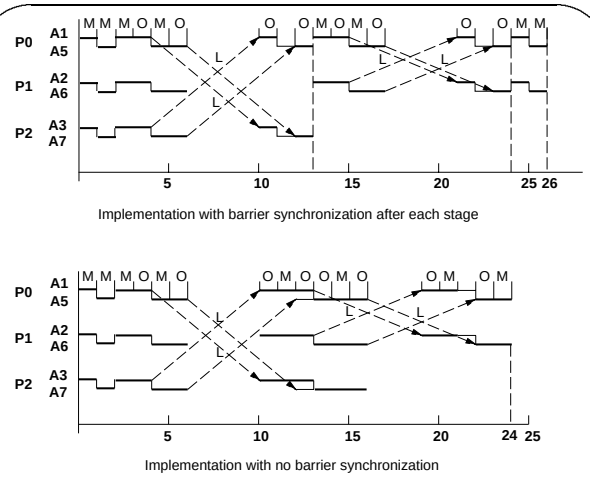

Figure 8. Timing comparison of two implementations of FFT when M=1, O=1, 1/B=2, L=6. M stands for method execution time, O overhead, B bandwidth, and L communication latency. Shown are only the relevant parts to P0.

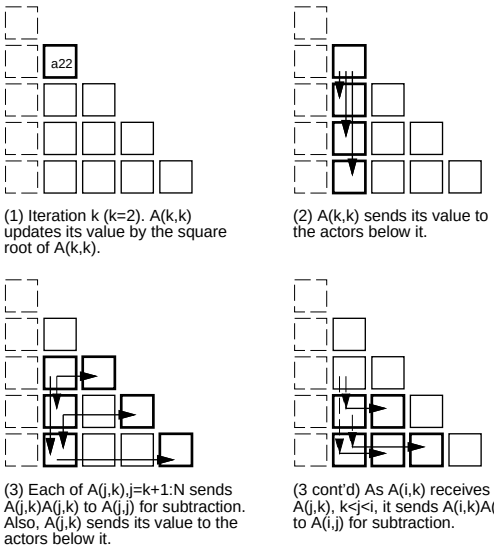

Figure 9. Communication in the k-th iteration of the outer product Cholesky decomposition.

require no remote communication, while the remaining $\log P$ columns need remote data for computation. Thus, the network spends $(n/P)\log n$ time in computing and $(O+(\frac{n}{P}-1)/B+L+O)\log P \leq (\frac{n}{BP}+L)\log P$ time in communicating, assuming $\frac{1}{B} \geq 2O$.

Note that the total execution time is less than the sum of the computation and communication time. For example, assuming O=1, $1/B$=2, L=6, the sum is $(8/4)\log 8+((8/4)2+6)\log 4=26$ cycles. However, because of the overlap between local computation and communication, the actual parallel time is 24 cycles (Figure 8).

5.3. Cholesky decomposition

Our next example uses a Cholesky decomposition algorithm based on outer product updates [12]. An iteration of the decomposition consists of three steps.

```
   for k = 1 : n
(1) A(k,k) = sqrt(A(k,k))
(2) A(k+1 : n, k) = A(k+1 : n, k) / A(k,k)
      for j = k+1 : n
(3)     A(j : n, j) = A(j : n, j) - A(j : n, k) A(j, k)
      end
   end
```

In the first step of the k-th iteration, the diagonal element $A(k,k)$ is square-rooted. Then, elements $A(i,k)$, $k+1 \leq i \leq n$ are divided by $A(k,k)$. In the last step, elements in the triangular matrix to the right of the column k are subtracted by the outer product $A(k+1{:}n,k)A(k+1{:}n,k)^T$.

Suppose each element of the lower triangular of A is represented as an actor. Figure 9 illustrates the communication in the k-th iteration of the implementation. Note that because of the asynchrony in communication, messages from different steps and iterations may coexist during the decomposition. Thus:

1. Each element should process messages from each step of an iteration before messages from subsequent step(s). For example, $A(4,2)$ should process the message $A(2,2)$ before the message $A(3,2)$.

2. Each element should process messages from one iteration before messages from subsequent iterations. For example, $A(4,2)$ should process a message from $A(4,1)$ before a message from $A(2,2)$.

The synchronization and scheduling requirements in the Cholesky decomposition algorithm greatly complicate performance analysis of an actor implementation of the algorithm: the implementations may exhibit different performance characteristics depending on synchronization and scheduling mechanisms they employ.

For example, consider the Cholesky decomposition of an $N \times N$ matrix. Assume that we have as many

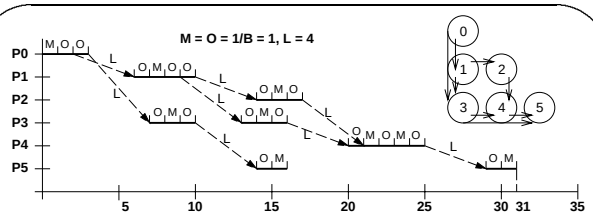

Figure 10. Timing diagram of an actor-based Cholesky decomposition of a 3x3 matrix.

processors as the number of actors (i.e., $N(N+1)/2$) and assign one actor to each processor using a location function $\mathcal{L}$:$A(i,j) \rightarrow \frac{i(i-1)}{2}+j-1$. The complete analysis of an implementation without assuming global synchronization is quite involved. Thus, we only show an illustrative example. Figure 10 is the timing diagram of the decomposition of a 3×3 matrix when messages are processed in the first-come-first-served order and no global synchronization is assumed. The necessary synchronization requirements may be implemented on a per-actor basis by using local synchronization constraints. The execution takes 31 cycles when $O=\frac{1}{B}=1$ and $L=4$.

On the other hand, if we assume that all the actors are synchronized after each iteration, we may simplify the analysis. Since we charge a cycle for method execution, step 1 takes one cycle for each iteration. 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 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 taken by an actor $A(i,k)$, $k+2\leq i\leq N$, in response to a message $A(j,k)$, $k+1\leq j\leq i-1$, to compute $A(i,k)A(j,k)$ (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 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 $L=4$.

If one assumes global synchronization, the task of estimating the cost of an actor program can be simplified. However, as we have illustrated in the example above, such a model is very misleading because it ignores the possibility of overlapping communication and computation. In fact, the model becomes more misleading as the gap between the speeds of computation and communication continues to grow.

5.4. Shared Memory Architectures

Although the complexity model is stated for the distributed memory multicomputer, it may be adapted to other architectures as well by adjusting a few parameters. In particular, we may apply the model to shared memory architectures by using $\mathcal{L}_{sm}$:$\mathcal{A}\rightarrow0$ as the location function. Because all actors reside in shared memory, there is no distinction between local and remote messages. Let O, B, and L denote the overhead to send a memory access request, the bandwidth, and the memory access latency, respectively. Assuming $\frac{1}{B}\leq0$, sending a message costs O cycles. Because a message must be brought into a processor before it is processed, processing a message takes additional $2L+2O$ time, assuming no overhead in the shared memory. Similarly, we may adapt the model to a hybrid architecture where a number of shared memory multiprocessors (or clusters) are connected by an interconnection network. By using a location function which takes a mail address to a pair of a cluster id and a processor id, we may distinguish between intra-cluster communication and inter-cluster communication.

6. Other Models

A number of parallel programming models have been proposed in the literature. We compare some of them with the actor-based parallel programming model.

6.1. PRAM models

Parallel Random Access Memory (PRAM) [9] is one of the simplest models proposed for representing parallel algorithms and analyzing their complexity. PRAM was also the first model for analyzing parallel complexity and has remained popular with theoreticians for a long time. The PRAM model assumes a single shared memory; each processor can access any memory cell in the shared memory in unit time. Different variants of PRAM are based on how memory contention (i.e., concurrent reads or writes by different processors) is handled. By contrast, the state of an actor is private and interaction between actors is through explicit asynchronous message passing.

Though the PRAM model has been useful in designing parallel algorithms, it is based on at least three

unrealistic assumptions. First, the model assumes that interprocessor communication is free, i.e., communication network has infinite bandwidth, zero latency, and zero overhead. The inability to recognize the cost difference between local communication and remote communication encourages algorithm design with unnecessary interprocessor communication and hinders efficient exploitation of data locality. Second, PRAM assumes each memory cell is independently accessible and ignores memory contention at the level of memory module. Finally, it assumes that the processors operate in complete synchrony, an assumption that is reasonable for the purpose of simply counting the number of operations but is unreasonable since the PRAM is assuming the possibility of synchronizing through shared memory after each local operation.

These unrealistic assumptions have given rise to a number of extensions which address one or more of the problems. For example, to address the memory contention issue at the memory module level, several extensions [16, 20] divide the shared memory into a limited number of modules, each of which can respond to only one access request at a time. Other extensions [11, 5] introduce asynchrony to the PRAM model to avoid the unrealistic assumption of lock-step execution. There are also extensions that account for communication latency [21, 1] and that model communication bandwidth [2]. However, all of the models invariably assume that the architecture has a common memory shared by all the processors, which can introduce considerable inefficiency.

6.2. Hierarchical memory model

As the difference between processor cycle time and memory access time grows, a number of parallel models that acknowledge the cost difference in accessing different memory modules have been proposed. Among them is the Parallel Memory Hierarchy (PMH) model; in the PMH, memory is represented as a hierarchy of memory modules. A multiprocessor is represented by a tree of modules with a global memory at the root and processors at the leaves. Another memory-centric model is the MPI-2 remote memory model [10].

The hierarchical memory model is based on the observation that the techniques for tuning code for the memory hierarchy are similar to those for developing parallel algorithms. Although such models may accurately analyze performance and thus assist high-performance implementations [13], they make programming tedious: the programmers need to explicitly control all the architectural details. Moreover, the programs may not be portable across architectures.

By contrast, the use of high-level actor abstractions hides architectural details while the use of efficient compilation and runtime support retains efficiency. High-level actor programs represent the logical order of operations and decomposition of the data. Obtaining efficient execution of an actor program on a particular architecture may be facilitated by combining separately specified placement with the program. Moreover, we argue that it is generally impossible to determine complexity without fixing a parallel architecture and placement strategy.

6.3. Bulk synchronous parallel model

In the Bulk Synchronous Parallel (BSP) model, a parallel machine is described in terms of three attributes: processor/memory modules, an interconnection network, and a barrier synchronization facility. A computation consists of a sequence of *supersteps* of a regular interval L in which a processor performs local computation and sends and receives messages. The local computation may depend only on data locally available at the beginning of the superstep, and a processor can only send at most h messages and receive at most h messages in a superstep (called h-relation). At the end of every superstep, all processors are synchronized; the parallel machine proceeds to the next superstep only after all the processors have completed the current superstep. Messages are for memory reads and writes and the interconnection network implements storage accesses between distinct components. The BSP model is shown to optimally simulate the PRAM model given sufficient *parallel slackness* [25].

Because the model requires that all processors synchronize at every superstep, the efficiency of using the model is open to question. In particular:

- Because a superstep must be sufficiently large to accommodate an arbitrary h-relation, the *bulk synchronization* may waste cycles of some processors.
- The results of the requests sent during a superstep can only be used in the subsequent supersteps, even if the interval L is longer than communication latency. The requests should be scheduled well before the results are used, which may disrupt the logical flow of program execution and complicate programming and compilation.
- Barrier synchronization is generally quite expensive. Although special hardware support is possible for bulk synchronization, it is not likely to be available on many parallel machines. Moreover, such hardware is expensive to scale.

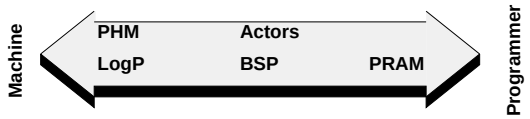

Figure 11. Relation between models for massively parallel programming

The Actor model is inherently asynchronous and concurrent, and synchronization among actors is specified on a per-actor basis only when necessary. Actor messages encapsulate data and specify remote computation so that actors capitalize on computation power of remote resources through delegation.

6.4. The LogP model

The LogP model [7] is a model that was motivated by technological trends in the early 90's when massively parallel machines were built by connecting processor/memory modules with interconnection networks. It is a low-level programming model for distributed memory computers and resembles the CSP model [14] by assuming the configuration of one process per processor. The model accurately predicts performance of a range of implementations by incorporating communication latency, overhead, and network bandwidth and inspired the development of our performance analysis framework for actors in this paper.

Although the LogP model brought up important issues of data placement and balanced communication, it deliberately neglects the issue of potential latency hiding by overlapping communication with local computation. Our performance analysis framework extends the LogP model with a location function and thus enables us to model multiple actors on a single processor and to take into account the overlap between local computation and communication. As our examples and practical experience show, such an overlap can result in a large reduction in parallel complexity.

As a summary, we may plot the models on a horizontal line between programmers and machines (Figure 11). As we move closer to programmers, we get programmability but lose efficiency and accuracy. Conversely, as we move closer to machines, we are rewarded with efficiency and accuracy at the expense of programmability.

7. Summary

Correctness and performance are two aspects of any program. In parallel programming, inferring correctness is complicated by the inherent nondeterminism of programs. Assessing performance is complicated by the impact that architecture-specific aspects, such as placement, scheduling, and message routing, have on parallel complexity. It is a fact of life that a number of different parallel architectures have been proposed and built and there is no convergence on a single architectural model. Human intervention is indispensable for achieving efficient execution of programs on these diverse parallel architectures. An important goal of a parallel programming model is to facilitate such intervention.

We have developed a model for parallel programming based on actors. We believe that the asynchronous, concurrent semantics makes actors suitable for scalable parallel computing on a range of parallel computers. We described a number of high-level abstractions for actors which enhance programmability and readability, and discussed how actors are implemented in parallel computers. Further development of programming abstractions and implementation techniques will facilitate high-level parallel computing. We have also proposed a general parallel complexity analysis framework that can be applied to accurately predict the performance of programs on a variety of parallel architectures. While analyzing program complexity is more complicated in our model than in many of the other proposed models, it is also more accurate. We believe the extra work is worth the effort.

It is clear that any particular parallel programming model involves a number of trade-offs. We have provided the motivation for the design decisions we have made to balance the various concerns. However, it is also clear that considerable research remains to be done in this area before scalable parallel programming can converge on a single approach or methodology.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989. PRAM with latency; block PRAM.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. In *Theoretical Computer Science*, pages 3–28, 1990. PRAM with bandwidth.

[3] W. Athas and C. Seitz. Multicomputers: Message-

Passing Concurrent Computers. *IEEE Computer*, pages 9–23, Aug. 1988.

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

[5] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989. asynchronous pram model.

[6] J. M. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.*, 39(6):297–301, June 1965.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[8] W. Dally. *The J-Machine: System Support for Actors*, chapter 16, pages 369–408. M.I.T. Press, Cambridge, Mass., 1990.

[9] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978. original proposal of the PRAM model.

[10] T. M. Forum. *MPI-2: Extensions to the message-passing interface.*

[11] P. B. Gibbons. More Practical PRAM Model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. ACM, 1989. PRAM with asynchrony.

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

[13] W. Gropp. Performance driven Programming Models. In *Proceedings of the 3rd International Working Conference on Massively Parallel Programming Models (MPPM '97)*, 1998.

[14] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[16] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Proceedings of the 24th Annual ACM Symposium of the Theory of Computing*, pages 318–326, May 1992. module parallel computers 1.

[17] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

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

[19] 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.

[20] K. Mehlhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984. module parallel computers.

[21] C. H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the 20th Annual ACM Symposium of Theory of Computing*, pages 510–513, 1988. PRAM with delay.

[22] D. Reed and R. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing.* The MIT Press, 1987.

[23] K. Taura. Design and Implementation of Concurrent Object-Oriented Programming Languages on Stock Multicomputers. Master's thesis, The University of Tokyo, February 1994.

[24] 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.

[25] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

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