Distributed Execution of Actor Programs

Gul Agha, Chris Houck and Rajendra Panwar

Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: { agha | houck | panwar}@cs.uiuc.edu

Abstract

A number of programming language models, including actors, provide inherent concurrency. We are developing high-level language constructs using actors and studying their implementation on multiprocessor architectures. This report describes our experience with programming in actors by means of a specific example of scientific computation. We also discuss work in progress on language annotations and compilation technology for efficient program execution on multiprocessors.

1 Introduction

Concurrent language models, such as concurrent logic programming, functional programming and actors, provide inherent concurrency in the evaluation of expressions. However, unlike other models, actors allow state to be directly expressed and manipulated. Our experience suggests that this enables us to write programs which not only avoid unnecessary sequencing of actions but which are also easily understandable. We discuss the structure of actor languages and issues related to their implementation on distributed memory architectures.

This work has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

This paper was submitted to the Forth Workshop on Languages and Compilers for Parallel Computing, August 1991.

Inherently concurrent languages suffer from an embarrassing amount of concurrency. Current compiler technology is not sufficiently developed to optimize placement and migration of objects in order to provide sufficient execution efficiency on distributed memory architectures. Furthermore, in some cases, the structure of a problem is *crystalline* and well-understood by the programmer. In these cases, the use of high-level annotations to guide the runtime system has been suggested. The use of annotations provides modularity by separating the logic of an algorithm from its implementation. Thus we believe annotations are preferable to explicit placement.

The organization of this paper is as follows. In Section 2 we give a brief overview of the actor model. Section 3 outlines annotations to provide efficiency directives for multiprocessors. Section 4 illustrates the concepts by an example, it describes the use of actors to express an algorithm for the Cholesky Decomposition of an SPD matrix. In Section 5 we discuss the representation of actors on multiprocessors. The final section presents some conclusions.

2 The Actor Model

Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. Each actor has a conceptual location, its *mail address*, and a *behavior*. An actor's *acquaintances* are all of the actors whose mail addresses it knows. In order to abstract over processor speeds and allow adaptive routing, preservation of message order is not guaranteed. However, messages sent are guaranteed to be received with an unbounded but finite delay.

State change in actors is specified using replacement behaviors. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. The replacement behavior for a purely functional actor is identical to the original behavior; in general it may change. The change in the behavior of an actor may represent a simple change of state variables, such as change in the balance of an account, or it may represent changes in the operations (methods) which are carried out in response to messages. For example, suppose a bank account actor accepts a withdrawal request. In response, it will compute a new balance which will be used to process the next message.

Replacement is a serialization mechanism which supports a trivial pipelining of the replacement actions: the aggregation of changes allows an easy determination of when we have finished computing the state of an actor and are ready to take the next action [3]. For example, as soon as the bank account actor has computed the new balance in the account, it is free to process the next request – even if other actions implied by the withdrawal request are still being carried out. This allows concurrent execution of actions specified within the body of the actor.

The concept of actors was originally proposed by Hewitt in [9]. The actor model was formally characterized by means of power domain semantics in [6], by a transition system in [1], and by Colored Petri Nets in [13]. Complexity measures for actor programs have been studied in [5]. The model has also been proposed as a basis for multiparadigm programming in [2] and has been used as a programming model for multicomputers in [4] and [7].

Rosette

Our work uses Rosette, an actor language developed at MCC in collaboration with one of the authors [16]. The following code-fragment gives a flavor of the Rosette language. It defines a class of actors, `Add-Counter` which accept two types of messages and has an acquaintance (cf. local variable) called `count`, initially set to zero. Upon receipt of an `add` message with parameters `a` and `b` an actor of this type returns the sum of `a` and `b` and specifies its replacement behavior using the same behaviour definition and local count incremented. If the actor receives a `(die bookkeeper)` message it sends a message `[total count]` to `bookkeeper` and becomes a `sink`, an actor which ignores all future messages that it receives.

```
(defActor Add-Counter (slots& count 0)
   (method (add a b)
      (become Add-Counter (+ count 1))
      (+ a b))
   (method (die bookkeeper)
      (total bookkeeper count)
      (become sink)))
```

Notice that the `(become Add-Counter (+ count 1))` and the `(+ a b)` statements do not affect each other and can therefore be executed concurrently.

Actor programs have a fine-grained concurrent structure which is quite similar to that in functional and concurrent logic languages. Actors can be used to represent purely functional programs as well as programs that require objects with history sensitive behavior. There are two sources of concurrency in actor programs. First, the actions on different actors can be executed in parallel, thereby allowing us to write parallel programs which require expression of state changes in history sensitive behaviors in objects. Second, actions carried out within an actor are executed concurrently. The efficiency of an actor program on a distributed memory architecture depends on where different actors are placed and the communication traffic between them. Thus the placement and migration of actors can drastically affect the overall efficiency.

Execution of actor programs on parallel architectures requires a translator which takes an actor program as input and generates executable code for a given parallel machine. Ideally, the system executing actor programs should be able to decide efficiently where to place actors and when to migrate them. We provide some simple schemes for providing annotations to actor programs in order to help the system decide on placement issues. The next section discusses in greater details, how the execution efficiency of actor programs on parallel machines can be ensured by using annotations.

3 Efficiency Increasing Directives using Actors

When a new actor is created, a choice must be made regarding the processor on which the actor should be placed, for example, locally or on some specific remote processor. The problem is complicated due to the ability of actors to create more actors and dynamically change the topology of the system. For a general actor program, it may be difficult to give a scheme which optimally decides where a given actor should be created and when and where it should be migrated. In general, such a method will need to be adaptive and may require expensive bookkeeping of the load on different processors and the communication patterns between different actors. However, for many numerical algorithms and other crystalline problems, it is possible to give simple schemes that can efficiently decide the placement of actors and migration. In our current work, we provide the programmer with the ability to abstractly annotate actor programs to govern actor placement and their migration. This is similar to the notation used in the language Logo and later incorporated in other languages such as *ParAlfl* [10].

The syntax for providing annotations to actor programs is: <*expr*1> @<*expr*2> where <*expr*1> represents a message send (or function application) or an expression that creates a new actor, and <*expr*2> evaluates to a processor number PID. The above annotation indicates that if <*expr*1> is an expression representing a message sent to an actor, evaluate the function on the processor PID. Alternately, if < *expr*1 > is an expression creating a new actor, create the actor on the processor PID. Thus `(new some-behavior) @(pe-expr)` creates an actor on the processor whose id returned by `pe-expr`. The expression `(mult scalar vector)` sends the `(mult vector)` message to `scalar`. Here `vector` is a tuple and `start-pe` evaluates to the number of the processor containing the first element of `vector`. This causes unnecesary degree of data movement. However with a `@(start-pe vector)` annotation, `scalar` will be sent to the processor containing `vector` and operation `mult` carried out there.

As an application of the above annotation, consider the following program for computing the sum of elements of an array A:

```
(defActor SUM (slots& A)
  (method (partial-sum)
    (if (= (size A) 1) (head A)
       (+ (partial-sum (new SUM (first-half A))
                              @(start-pe (first-half A)))
          (partial-sum (new SUM (second-half A))
                              @(start-pe (second-half A)))))))
```

The calling actor creates a new actor `SUM` with acquaintance A (the original array), sends it the message `partial-sum` and waits for the final answer. Assume that array A is distributed between p processors so that there are n elements on each processor. The creation of actors for computing the partial sums of array elements on each processor is

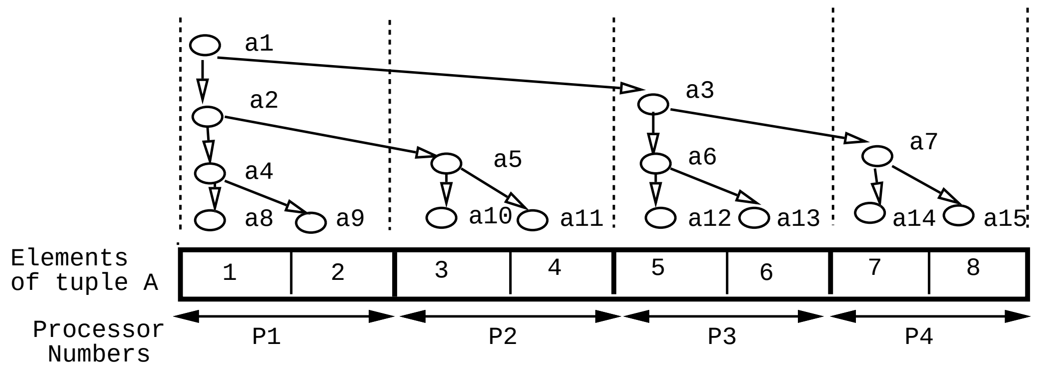

Figure 1: The creation of actors on processors for addition of the elements of array A.

shown in Figure 1. The `SUM` actor, $a1$, creates two new `SUM` actors, $a2$ to compute the partial sum of the first half of the array and $a3$ to compute the partial sum of the second half of the array. The actor $a2$ is created on the processor containing the first element of the first half of A. The actor $a3$ is created in the processor containing the first element of the second half of the array A. Once $a4 - a7$ are created, there is at least one actor in each processor and subsequent actors are created locally on each processor. The final value of the sum of the array elements is returned to $a1$ because of the implicit continuations provided by the function call mechanism [1].

4 Cholesky Decomposition of an SPD Matrix

We now discuss an example, the Cholesky Decomposition (CD) of a Symmetric Positive Definite (SPD) matrix, to illustrate how the actor model enables us to represent different ways of solving the problem in parallel. Assume A is a symmetric positive definite matrix of size $n \times n$. The following algorithm computes a lower triangular matrix G, of size $n \times n$ such that $A = GG^T$ [8]. Since A is a symmetric matrix, it can be stored as a lower triangular matrix. The elements of G overwrite the corresponding elements of A.

```
for j := 1 : n
   if(j > 1) A[j:n,j] = A[j:n,j] - A[j:n, 1:j-1] * A[j, 1:j-1]$^T$
   A[j:n, j] = A[j:n, j] / sqrt(A[j,j])
end
```

The above algorithm is suitable for solving dense systems of SPD equations. Most of the problems involving large systems of equations have very few non-zero elements in the matrices and require methods for solving sparse linear systems. But some of the methods used for solving sparse systems of equations (e.g. [15]) require solution of a dense block of equations as an intermediate step. Depending on the size and structure of the original systems of equations, this dense block may be huge and sequential solution of such a block may reduce the overall speedup significantly. We discuss ways of computing CD of dense matrices in parallel.

Figure 2 illustrates the maximal parallelism available in the CD. Note that the communication requirements are language independent — they are characteristics of the algorithm at hand.

With a two dimensional mesh of n^2 processors [8], the algorithm takes $O(n^2)$ time if all the steps of one iteration are completed before starting the next one. Pipelining the execution of different iterations gives an $O(n)$ time parallel implementation. In general the number of processors available is fewer than the number of elements in the matrix and several elements are assigned to each processor. For example, if n processors are available, one row can be assigned per processor to get an $O(n^2)$ time parallel implementation. We now show details of two representations of the CD algorithm using actors. The first representation is purely functional whereas the second uses destructive updates and pipelines the execution of different iterations.

A Functional Implementation of CD

We define an actor `Mat-ops` with a method `cholesky`. A message `cholesky` can be sent to `Mat-ops` to compute the CD of a given matrix. The arguments sent with the message `cholesky` are the matrix itself (which is essentially the location of the first element of the matrix) and the size of the matrix. The matrix is represented as a tuple of rows, where each row itself is a tuple of floating point numbers. The matrix is stored as a lower triangular matrix and the tuple representing the i^{th} row is of size i.

```
(defActor Mat-ops
```

Figure 2: Parallelism available in CD algorithm

```
(method (cholesky Matrix num-rows)
        ((iteration 1 Matrix num-rows) @(start-pe Matrix))))
```

The function *iteration* organizes the execution of the n iterations of the algorithm. The function `(updated-row Row)` returns a row obtained by updating the diagonal element of row k for starting the k^{th} iteration as shown in Figure 2. Similarly, `(updated-matrix Row Matrix)` returns a matrix obtained by updating its elements as required in iteration k.

```
(defProc (iteration iter Matrix n
    (if (> k n)
      []
      (let [[ X ((updated-row (head Matrix)) @(start-pe (head Matrix)))]]
        (concat X (iteration (inc iter)
            ((updated-matrix X (tail Matrix)) @(start-pe (tail Matrix)))))))
```

The input matrix is distributed between different processors, and the execution of function `updated-matrix` can result in large data movements if not done carefully. The following function uses annotations for efficiently executing `updated-matrix` function.

```
(defProc (updated-matrix Row Matrix k)
    (if (= (size Mat) 1)
      ((updated-row Row (head Matrix)) @(start-pe (head Matrix)))
      (concat
        ((updated-matrix Row (first-half Matrix))
                           @(start-pe (first-half Matrix)))
        ((updated-matrix Row (second-half Matrix) k)
                           @(start-pe (second-half Matrix))))))
```

A More Concurrent Implementation of CD

The program illustrated above is written in a purely functional style with a call-by-value semantics. This results in two problems. First, it sequentializes the execution of different iterations by forcing completion of one iteration before starting the next one, and second, its straight-forward implementation is inefficient in space utilization. The first issue can be handled by observing the fact that the $(k+1)^{th}$ iteration can be started as soon as row $(k+1)$ has been updated for the k^{th} iteration. A call-by-name semantics with implicit futures would be one way to address this problem. The second issue can be handled without changing the functional specification by allowing destructive updates in the implementation. However, a concurrent implementation of lazy functional languages with implicit futures and side-effects in the implementation has to address the effects of the asynchrony of components and the indeterminacy in message order. Interestingly, because the implementation itself involves a nondeterministic merge, it cannot be expressed functionally. For this reason, we believe that it is better to provide a more powerful linguistic model. We find actors a suitable alternate for two reasons. First, the use of actors with local states makes the structure of the algorithm more perspicuous. Second, it allows us to write efficient programs whose execution has not been optimized by the compiler.

As before we assume the matrix is available as an acquaintance of a given `master` actor. This actor creates several `Row-act` actors, one for each row. Each `Row-act` actor has three acquaintances, the row of the matrix associated with the `Row-act`, the next `Row-act` actor (`nextRow-act`) and the number of its row `Row-num`. Since messages corresponding to several iterations exist in the system at the same time and messages can reach out of order, it is necessary for `Row-act` to impose some order on the messages it processes. For example, an actor on processor P1 can send a message to `Row-act` R_m to update its row for the iteration k and P2 can send a similar message to R_m for iteration $(k+1)$. Even if, the message from P2 is sent after the message from P1 the two messages may follow different paths and reach R_m in an unpredictable order. Since, processing messages for two different iterations in a wrong order leads to incorrect results, R_m needs some mechanism to decide which message should be picked from the mailbox for processing next. We use a mechanism called *enabled sets* in Rosette for imposing order on the processing of messages [14]. The following code uses the `block` construct which packages a set of expressions which are evaluated concurrently. The `next` construct specifies the next enabled-set of message that the actor will accept.

```
(method (update-mat iter inpRow)
    (block
        (update-row iter diag-value)
        (if (present nextRow-act)
            (update-mat nextRow-act iter inpRow)))
        (cond (< iter (dec Row-num))
                (next [[update-mat (inc iter)]])
              (= iter (dec Row-num))
                (next [[start-iter]])))
(method (start-iter)
   (let [[x (new-diagonal (head Matrix))]]
(block
    (update-diagonal x)
    (if (present  nextRow-act)
        (block
           (update-mat nextRow-act Row-num x)))
           (start-iter nextRow-act)))
    (next [[return-ans]]))))
```

When `Row-act` R_m receives an `update-mat` message with value of `iter` equal to k, it updates its row and enables the next set of messages which is a singleton set containing messages of type `update-mat` with value of `iter` equal to one more than the current value of `iter`. Thus only the update corresponding to $(k+1)^{th}$ iteration will be carried out after iteration k. Once the value of `iter` reaches $(m-1)$ (i.e. one less than the row number of R_m), the next set of messages enabled is the singleton set containing the message of type `start-iter`. Finally, an actor can receive the resulting matrix by sending the `return-ans` message to the first `Row-act` actor. The complete result is returned when the last `Row-act` finishes processing.

5 Representation of Actors

Sequential processes with sends and receives form a low level language support available on several distributed memory architectures. We are currently developing a translator to transform the Rosette specification of an actor language to code for a dialect of C requiring explicit sends and receives. There are a number of design issues which result from the static nature of C. We discuss one of these issues, namely the internal representation of an actor.

An actor's *behavior* specifies new tasks to create and communications to send as a function of its local state and the message being processed. Upon receipt of a message, an actor may replace its current behavior with a new behavior that is used to process all subsequent messages [1]. In Rosette, this is done with the `become` primitive. It is assumed that if an actor does not specify a replacement behavior with a `become`, its behavior is the same as its current behavior. In the following example, the actor `Foo` turns itself into a `sink` (i.e. it stops processing further messages) upon receipt of a `die` message:

```
(defActor Foo
  (method (die)
    (become sink)))
```

In a statically specified language, such as C, behavior replacement is a troublesome trait to mimic; an actor may choose its future behavior definition completely dynamically. For example, the behavior definition may be specified in a message: as an address of a different actor whose behavior the recipient should adopt. One possibility is to use static analysis to determine the set of behavior definitions that an actor may use and encode them into a single behavior. While this technique will work for many actor programs, it is insufficient in general. For example, static analysis cannot support *reflection* where the runtime system may be reconfigurable by an executing application. Furthermore, the method may create source code on the order of the product of the number of possible actor classes by the number of actor definitions in the user's program.

Forwarding Pointers. An attractive, and fairly simple idea, is to mimic each actor with a separate process which executes its behavior. When an actor changes its behavior definition, we simply create a new process which executes all future messages that get sent to the original actor. We should first note that it will be necessary to keep the original actor around, since its acquaintances may only know of it by the original mail address. So when messages get delivered to the original mail address, the actor simply forwards them to the new process that it created. While there are typically few behavior redefinitions in practice, this method can entail a significant performance loss if the network gets filled with messages getting forwarded from previous behaviors to new ones. Furthermore, it is not clear how to implement enabled-sets and inheritance under such a scheme [14].

Actors as Records. An alternative scheme which addresses these problems represents each actor as a record containing a local variable bound to the address of its current behavior and the actor's current local variables. This representation is used in actor languages such as Acore [12]. Each behavior type is represented by a different process. The execution of an actor system is controlled by *drivers* which maintain lists of local actors and the messages to them.

All messages to an individual actor are sent to the *driver* which is responsible for that actor. If the actor is currently enabled to execute a message of that type, the driver packages up the actor's record and the incoming actor and sends them in a message to the process which corresponds to the actor's current behavior. Sending an actor record out to a separate process (rather than have the driver itself execute the actor's behavior) recovers much of the concurrency in the program since all of the processes can execute in parallel.

Once the actor's replacement is determined, the process that is executing it sends its record back to the driver which places it back in the queue of actors ready to execute. If the actor changes its behavior, it simply means that the process which is executing the actor's behavior changes the local-status variable before it sends its record back to the driver.

This is more efficient in terms of source code length. However, if a driver is implemented on a single processor it can only execute a single message at a time. This creates the possibility of a bottleneck at the driver. The problem can be addressed by dynamically splitting a driver to maintain load balance.

The main problem with representing actors as records is that we *always* suffer an indirection penalty: messages have to be first sent to a driver and then to the process to execute the behavior. However, no matter how many times an actor changes its behavior, messages to it only suffer a constant amount of indirection, unlike the message forwarding scheme. Furthermore, a driver can keep track of what type of messages an actor is enabled to receive and check them against the incoming messages, so we get enabled sets for free. It appears that inheritance will be easier to implement under such a representation.

6 Conclusions

The language support currently provided on a number of distributed memory architectures consists of sequential procedures which can send and receive data across processors. The intel iPSC also provides the ability to choose specific (tagged) messages out of the mail box [11], thus allowing a user to handle arrival order non-determinism efficiently. However, programmers have to specify explicit processor addresses with every message that is sent across nodes. In particular, this means that they have to keep track of the mapping of the original data on the processors and compute the number of processor where certain data that is needed may be available or where the next message to start some computation should be sent. Although the CD algorithm discussed above allows a fully static mapping of data to processors, more generally optimal execution may require that objects be moved dynamically to different processors. It then becomes increasingly difficult for the programmer to keep track of the object to processor mapping. We feel that the actor model enables a programmer to have a logical view of the problem being solved, thereby hiding some architecture dependent details. Our work also suggests that it is possible to easily represent algorithms using actors and that such representations can be efficiently executed on distributed memory architectures.

References

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

[2] G. Agha. Supporting multiparadigm programming on actor architectur es. In *Proceedings of Parallel Architectures and Langua ges Europe, Vol. II: Parallel Languages (PARLE '89)*, pages 1–19. Espirit, Springer-Verlag, 1989. LNCS 366.

[3] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–23, August 1988.

[5] F. Baude and G Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science*, 1991. LNCS 480.

[6] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[7] W. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Press, 1986.

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

[9] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[10] P. Hudak. Para-functional programming. *IEEE Computer*, pages 60–70, August 1986.

[11] Intel Corporation, Beaverton, Oregon. *iPSC/2 C Programmers reference manual*, 1988. Order Number: 311017-002.

[12] Carl Manning. Acore: The design of a core actor language and its compil er. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[13] Y. Sami and G. Vidal-Naquet. Formalisation of the behaviour of actors by colored petri nets and some applications. In *Proceedings of Parallel Architectures and Languages Europe, (PARLE '91)*, 1991.

[14] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA Proceedings*, 1989.

[15] P. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Technical report, University of Illinois Department of Computer Science, In Preparation.

[16] C. Tomlinson W. Kim M. Schevel V. Singh B. Will and G. Agha. Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.