Actor Oriented Programming in Chapel

Amin Shali
Department of Computer Science, University of Texas at Austin

Spring 2010
CS380P, Prof. Calvin Lin

Abstract

Programming for scalable multi-cores with more focus on locality has increased the interest in message passing models of programming. One of the well known theoretical and practical models of message passing is the *actor* model. There are a multitude of actor oriented libraries and languages, each of which implement some variant of actor semantics. Among the languages and the libraries based on the actor semantics we can name Erlang, SALSA, E language and Axum. Chapel is a global view language and have no notion of actor and message passing. However, it tends to be a general language for parallel and concurrent programming and wants to provide simple means for the programmer to implement a parallel program. In this work we want to study and analyze some questions regarding actors in the Chapel language. In particular we want to answer the following questions:

- Why do we need actor semantics in Chapel? In particular we want to see if we really need something like actors in Chapel.
- What are the advantages and disadvantages of having actor semantics implemented in Chapel? By answering this question we will see what is the profit of having actors for programmer. Also we try to see what is the cost of having the actor semantics implemented in chapel.
- Can we do actor oriented programming with current Chapel language constructs? This question tries to reveal how well or poorly the current constructs in Chapel support actor style programming.
- What changes are needed to support true actor semantics in Chapel? The answer to this question helps us to realize the changes required to the compiler and run-time in order to have a true actor semantics in Chapel.

This study would be valuable to the Chapel community because it provides a fair amount of knowledge on the cost and benefits of having actors in Chapel and can help them with starting points for implementing actor semantics in the language.

1 Introduction

The advent of multi-core CPUs demanded a bigger need for parallel and concurrent programming. The program that was running fast on my desktop machine yesterday is not running as fast with my new quad core machine today [1]. The sequential programming models of yesterday are not suitable to take advantage of the power of multi-cores. Therefore from a programming language perspective we need a shift in the way we program these new machines.

Parallel programming is hard and is not like that of sequential programming. Programmers have more concerns when it comes to taming parallelism. Parallel programs are usually bigger than equivalent sequential ones. Models of parallel programming languages are different from familiar and popular sequential languages. Even libraries supporting parallel programming have different and strange concepts. This makes the parallel languages a bit unattractive [2]. Also parallel machines are different architecturally and programmers are concerned with new issues such as communication and synchronization.

Parallel and concurrent programming has been around since almost the birth of digital computers. Therefore many of the issues regarding parallel programming, mainly such as *Correctness*, *Performance*, *Scalability* and *Portability* [3], has been discussed and dealt with in different scales. However, recently these issues has gained more attention and that is because of the abundance of multi-core CPUs.

In this between, programming models are of great importance in the performance and productivity of parallel and concurrent programs. There has been two models of parallel programming from a memory point

of view; *Shared memory* and *Distributed memory*. In the shared memory model all the threads or processes see the memory as one big continuous chunk which can be accessed by all of them. This memory model is what we have in sequential programming and therefore it is easier for the programmer to deal with. On the other hand, in the distributed memory model, different processes see different parts of the address space. In this model processes usually communicate using message passing or an abstraction of message passing.

For instance, *Pthread* is a library for parallel programming which is based on shared memory model. *OpenMP* is another example from this model. *MPI* is an example of a parallel programming library based on distributed memory model. Moreover, the *Actor programming model* which is based on message passing assumes a distributed memory model.

We cannot really say which of these two models is superior over the other one. Each one has its own advantages and disadvantages with respect to the application. The shared memory model is familiar. But it has the disadvantage that manipulating shared data requires synchronization. This synchronization can be very costly if one does not do it right. Besides, programs written in this model are very hard to debug and reason about.

The distributed memory model has this advantage that it exploits locality very well. In addition, since there is no shared memory the assumption is that we do not have the problem of synchronization and that is a big relief. However, this model has significant communication overhead and that is the source of performance inefficiency.

In order to solve the issues of these two parallel programming models and take advantage of their strengths, a new programming model has been introduced [2]. This new model which is the combination of distributed and shared memory model is called *Partitioned Global Address Space* (PGAS). The PGAS model is similar to shared memory model in this respect that it provides a single contiguous memory view for all the processes. However, memory is partitioned into parts and different threads have different affinities with various parts of memory. The threads in a parallel program see a global address space. However, the physical memory might be distributed across different nodes. In order to exploit locality each thread or process has affinity with a part of memory and accessing the memory of other parts comes at a cost. The *Chapel* programming language is designed and implemented based on PGAS memory model. We will talk about Chapel language later.

The PGAS memory model allows us to build different programming models on top of that. For example, we can build a message passing programming model based on PGAS infrastructure. But the question is why we need such a model on top of PGAS? The answer to this question is two fold. First comes performance. Some algorithms are designed with the assumption that the memory is distributed (which is a realistic assumption) and therefore the algorithm designer take into consideration the cost of communication. On the other hand, there are algorithms which are designed with the idea of shared memory model and they perform really well on true shared memory machines. However, they don't perform very well on simulated shared memory models (e.g. PGAS) because they have not considered the cost of communication in algorithm design. Therefore, it makes sense to have message passing programming model on PGAS because then we can optimize the program for performance by taking advantage of the locality introduced by message passing model.

Since there are already efficient algorithms designed for the message passing model, it is easier to have the ability to express them in any memory model we have. And this brings us to the second reason which is expressiveness. We need an easy way to express programs which are based on message passing. Providing programmers with facilities and language constructs to write those programs increase the power of the parallel language. In addition, there are strong tools for correctness validation and verification of message passing programs.

One of the famous and important programming models based on message passing is the *Actor* model [4]. It is an inherently concurrent model merely based on asynchronous message passing. The actor semantics comprises of a number of important concepts including encapsulation, fair scheduling, location transparency, locality of references, and transparent migration [5]. All these make the actor model a suitable programming model for distributed settings. There are a multitude of languages, libraries and platforms developed based on the actor model [5]. Languages include *Erlang* [6], *E* language[7], *SALSA* [8] and *Axum* [9]. There are also many libraries from which we can name *ActorFoundry* [10], *ActorsGuild* [11], *Retlang* [12], *Jetlang* [13], *Jasab* [14] and *JavAct* [15]. There are also platforms such as ActorNet [16] based on actor model for programming wireless sensor networks.

No one really knows what would be the appropriate architecture and programming model of the future generation of mutil-cores with order of thousands of cores. However, actor model is definitely a good candidate. The argument in favor is that memory bandwidth and consistency are major drawbacks for

performance. Therefore architectures tend to be more distributed and programs try to exploit locality. Since actor model is basically designed as a programming model for distributed systems with no sharing, it sounds like a good choice.

As we discussed earlier, even in global view languages based on PGAS model, it is good and convenient to support message passing model. That said, we need to choose an appropriate and simple enough message passing model to support. Actor model with a strong semantics and theory suggests a simple and yet powerful model. We do not want to adopt something like MPI message passing model for two reasons. First, it is a rather complicated model with so many different kinds of sends and receives. Second, it does not have a strong semantics and theory behind it.

The rest of this paper is organized as follows: In the next section we discuss the concepts of the Actor model and why it is important. In section 3 we review the basics of the Chapel language and study the language constructs that we need for our message passing model in Chapel. Section 4 discusses the design of the actor model in Chapel and how we actually adopt the actor model in it. Furthermore, in section 5, we explain the compiler support for actor programming in Chapel and finally section 6 will conclude the paper.

2 Actors

Actor model is a model of concurrent computation for distributed systems [4]. An actor system consists of a number of concurrent and autonomous entities called *actors* [5]. An actor has its own thread of control and state and is the unit of concurrency in the actor model. The state of actor is encapsulated and is not shared. Each actor updates its own local state by processing messages that it receives in its mailbox.

2.1 Messages

Actors communicate using asynchronous message passing. The sender actor sends a message to the other actor (receiver) and continue without being blocked. The receiving actor has a mailbox to receive all the messages from which it processes messages one at a time in a single atomic step [5]. The single atomic step execution of a message in an actor enables the macro-step semantics [17] which is crucial in reasoning about actor systems.

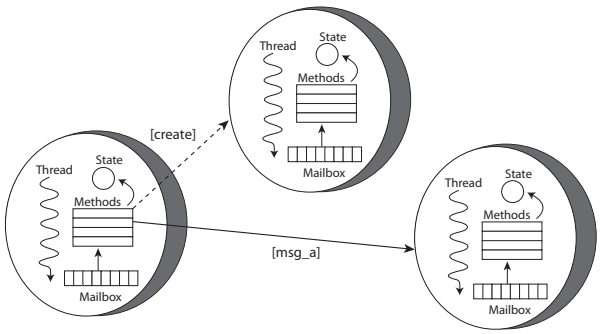

Figure 1: Actors communicate merely using asynchronous message passing. (Figure from [5])

As illustrated in Figure 1, actors do not share state. Each actor picks a message from mailbox and process the message and updates its state. Each message resembles a *task*. We use the terms task and message interchangeably. A task, as defined in [4], is a tuple of :

- A tag which distinguishes the task from all other tasks,
- A target which is the name of the actor to receive the task,
- A communication which is basically information required to do the task. It is a tuple of values.

The target of the task is the address of the receiver actor. This address can be a mailbox address. In some systems based on actor model it is a unique name for the actor which would have the same functionality of the mailbox address. The name should not be something that can be guessed. An actor can send messages to other actors that it knows. It can receive the address of other actors through the received messages or upon its creation through the constructor.

2.2 Actor Structure

An actor has a mailbox or a queue of messages. In some languages and libraries this queue is exposed to programmer and some systems hide the queue. For example in Kilim [18] library the mailbox is exposed. However, in ActorFoundry framework it is hidden from programmer. The process of delivering a message M consists of finding the appropriate message handler for it and give the message handler the communication information for that message to process the message. While delivering a message in an actor no other activity is being done inside it. In other words message delivery is serialized inside an actor.

The process of finding the appropriate message handler is a pattern matching process. Each message is matched against a number of patterns. If it matches then the expressions or statements associated with that match will be executed. Thereafter, the actor starts over to receive and deliver another message if there is any. If there is no message to process, actor will wait until it receives a message. Figure 2 shows the abstract structure of receiving and delivering messages in an actor.

```
loop {
  receive M
  match M with
    P1 : S1
    P2 : S2
    .
    .
    .
    Pn : Sn
}
```

Figure 2: The abstract structure of receiving and delivering messages in an actor

As a real example, we can see in Figure 3 an actor which is written in Scala language [19]. Aside from

```
class Pong extends Actor {
  def act() {
    var pongCount = 0
    while (true) {
      receive {
        case Ping =>
          if (pongCount % 1000 == 0)
            Console.println("Pong: ping "+pongCount)
          sender ! Pong
          pongCount = pongCount + 1
        case Stop =>
          Console.println("Pong: stop")
          exit()
      }
    }
  }
}
```

Figure 3: An actor in Scala. Notice that the parts for receiving and matching the messages are highlighted. [19]

the details of definging the actor, we can see that there is a never ending loop (`while(true)`) in which there is a `receive` statement for receiving messages. Inside the `receive` statement there are two match statement starting with `case`. Every time a messages matches any of them it will execute the statements after the case.

Not all the actor languages or libraries has this way of message handling mechanism. For example, in the ActorFoundry [10] message handlers are defined as usual Java methods. The framework then would handle the message delivery by dispatching the message in the queue to one of the methods that matches that. The process of dispatching is basically invoking the methods with the actual parameters provided in the message using *Java Reflection*. Figure 4 shows the Pong actor defined in the ActorFoundry.

Scala has the language level support for pattern matching and thus it makes it suitable for that style of message delivery. Erlang is pretty much the same as Scala in this regard. Java, on the other hand, does not have such thing and then actor libraries usually incorporate the Java reflection to accomplish the purpose.

```
class Pong extends Actor {
  private int pongCount = 0
  @message
  public void Ping(ActorName sender) {
    if (pongCount % 1000 == 0)
      System.out.println("Pong: ping "+pongCount);
    send(sender, "Pong");
    pongCount = pongCount + 1;
  }
  @message
  public void Stop() {
    System.out.println("Pong: stop");
  }
}
```

Figure 4: An actor in ActorFoundry

We cannot really say which way is superior to the other. It depends on the language and the expressiveness of the approach in that language.

In addition to message delivery part, an actor maintains a state. The state representation depends on the language. In imperative languages like Java or Scala, fields of the objects represent the state. In functional actor languages such as Erlang the state can be passed as a value tuple in the loop.

So far we have seen what an actor is all about and what its structure is. It has mainly two parts: *state* representation and *behavior* representation. Now we want to see what are the requirements for a minimal actor language. In other words we want to investigate the structure of an Actor program.

2.3 Actor Program Structure

A program in an actor language or a library which support actor programming consists of:

- *actor definition*, which is usually defined by a class that has the state variables and the part for handling messages as we saw in the previous part.
- *new expressions*, in order to create new actors.
- *send commands*, in order to send messages (creating tasks).

In Scala and ActorFoundry an actor can be created by using the keyword **new**. In Erlang an actor can be created using the keyword **spawn**. Both Erlang and Scala use the bang operator, **!**, to declare a message send. The syntax for sending a message in these languages looks like this:

receiver ! Message

In Erlang, the "Message" is declared by a tuple using braces. In Scala, though, it is a tuple or a single value. In ActorFoundry the message sending is done using the send function which is provided in the library. The parameters to the send function are: 1. The actor name we wish to send the message to. 2. The name of the message which is a string literal. 3. The actual parameters, as many as needed.

send(receiver, "message name", parameters...)

So far we saw the overview of the actor model from syntax and semantics point of view. In the next part we want to discuss some important semantics properties of actor systems. This discussion is important for our purpose in this work since we want to see to what extent we can support these properties in Chapel.

2.4 Actor Properties

There are four important properties for actor systems [5]: encapsulation, fairness, location transparency and mobility. We are going to introduce these properties in this part and later on we will see how we can implement these properties in Chapel.

2.4.1 Encapsulation

Encapsulation is by no means one of the most important parts of the object oriented programming. It is also important in actor systems. Encapsulation provides locality and safety for actors. As a result we will have data race free and memory safe actor programs. Encapsulation is two-fold [5]: state safety and messaging safety.

State safety means that no actor can access the state variables of another actor [5]. The only way to access and manipulate the state of an actor is through its messages. Therefore the language or library should provide some sort of mechanism for hiding data and making sure that data can only be accessed and altered using message passing.

Messaging safety is concerned with what is being passed inside messages. In order to guarantee that there is no shared state between actors we need to enforce the *call-by-value* semantics for message passing [5]. Call-by-value semantics requires making a copy of the actual data and put it in the message even in shared-memory systems. In other words we need to make sure that there is no pointer or reference to data that is being passed in the messages. Some languages encourage programmers to use immutable objects inside messages or explicitly make a copy of data upon sending the data. Other languages let the programmers decides which data should be copied and which data can be sent by reference. These languages provide some tags to annotate parameters. Thereafter, if the copying is necessary it will be done by the language run-time.

2.4.2 Fairness

In actor semantics it is stated that as a fairness condition every single message should be delivered finally. By delivering a message we mean successfully processing the message and remove it from the mailbox. There is no strict way of guaranteeing fairness defined in actor semantics. One can implement the fairness using a FIFO queue of messages or any other mechanism. Whatever the mechanism is, we want to make sure that no message will remain in the mailbox indefinitely.

2.4.3 Location Transparency

In actor semantics the location of the actor does not interfere with its computation and also does not affect others' computation. This means that it does not matter if an actor is running on a GPU or another node in the network or in the local CPU. The location of the actor only affects the latency and not the semantics of the system. Therefore we cannot make any assumption based on the location of actors. Location transparency enables seamless migration of actors which we are going to describe next.

2.4.4 Mobility

It is important that we can move data and computation from one processing node to the other one. This is specially important with today's heterogeneous architecture in which we want to create a balanced work load and we want to exploit different kinds of computation powers such as GPUs, DSPs etc. In order to have mobility we need to be able to move the data, program and the state of the computation to another place.

There are two kinds of mobility [20]; weak and strong. Weak mobility is the ability to move the code and data and strong mobility is the ability to move code, data and the execution state. Strong mobility is more difficult to support and requires preemption of the computation at any points or at some designated points in the code. Weak mobility, on the other hand, is easier to support and it only requires an idle actor (one that is not processing any message).

2.5 Programming Abstractions

There are two useful programming abstractions that we are going to discuss here. [5].

2.5.1 Request-Reply Messaging

The *Request-Reply Messaging* or the synchronous message passing scheme is one of the most common patterns of messaging. In this scheme the sender, after sending the message, will wait until the receiver processes the message completely. Meanwhile, the sender can receive messages from other actors in it mailbox. If the

fairness is guaranteed the message will eventually be processed and the sender will proceed. This pattern can be used for synchronization purposes.

2.5.2 Local Synchronization Constraints

The order of processing messages in an actor, though fair, is non-deterministic. But sometimes an actor needs to process the messages in a specific order or has to defer the processing of a message because it has to meet a certain condition to process the message. For example, consider a bounded buffer. The state machine for the bounded buffer is shown in Figure 5.

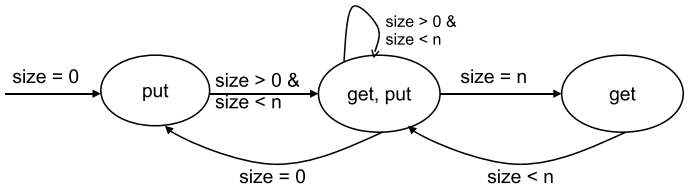

Figure 5: Bounded Buffer State Diagram

As can be seen in the figure, when the buffer is in the **put** state it cannot accept any **get** message. Nevertheless it might receive **get** messages. In this case, it has to defer the processing of the **get** message until it has something in the buffer to offer. Local synchronization constraints enable us to encode these states and define the circumstances under which a message can be processed. These constraints are usually a boolean expression on the state of the actor and the parameters passed with the message. Deferred messages are put in a queue for further processing. Whenever a message is processed successfully we should process all the deferred messages.

Not all the actor languages and libraries support all these properties. The reason that not all of them support all these features is basically the cost of implementation and the performance penalty they have to pay to support all of them. In case the actor implementation does not support a feature, it is the responsibility of programmer to implement them. For instance, in Scala, in order to assure the safe messaging one should make a full copy of the data being passed or hand in the ownership of data to the recipient. However, some features like fair scheduling requires some support in the run-time system and cannot be implemented by the programmer. Although in these cases programmer can write a better code. For instance, programmer can write a cooperative code that does not take down the thread for a long time. In any case, the burden is on programmer, which is not always good and is prone to error.

2.6 Why Actor programming is important?

Actor model is an inherently concurrent model of programming. So concurrency is in the heart of the model. It is very easy to create concurrent/parallel programs in an actor based language or library. Programmer does not have to deal with complexities of thread creation and synchronization. For example in GUI application where we need a lot of responsiveness and concurrency, actor model is definitely a good solution.

The non-shared memory architecture of the actor model is a huge gain in correctness and simplicity of the reasoning about the programs. Since there is no shared state, there would be no race. In addition, because there is no explicit locking required, programmer will not run into problems resulted from locking. Since the locking for message processing is being done by the actor framework or library it can be implemented in an efficient way to increase the performance.

The means of communication between actors, the message passing, and the lack of locking suggest less contention in the program. This helps improving the performance. In addition to this, programs written with the idea of non-shared memory architecture will perform better on non-shared memory machines. In contrast, programs written with the idea of a shared memory will not perform very well when we run them on a simulated shared memory machine which is in fact a non-shared one.

In addition to all these the simplicity of actor model makes it easy for an efficient implementation and portability. It is a model and has no restriction on the kind of underlying hardware. So, we can theoretically run actors on CPU, GPU, embedded devices and all other sorts of hardwares provided that we can communicate with them. There are currently implementation of languages and libraries such as ActorNet [16] to operate in wireless sensor networks.

3 Chapel

In this section we are going to have a brief overview of the Chapel language. We mainly focus on the parts of the language that is important in our implementation of actors.

Chapel[2] is a global view language based on the PGAS memory model. However, in contrast to other SPMD languages based on the PGAS model, Chapel is an MPMD one. Chapel has been designed to fulfill four goals:

1. Programmability
2. Performance
3. Portability
4. Robustness

In fact Chapel tends to be a general programming language for parallel computing. It targets the desktop multi-cores and clusters. Chapel is a multi resolution language. It provides language constructs for data parallelism, task parallelism and distribution of data. In addition, there is this notion of locales in Chapel. A locale is a unit of computation in the system. It could be a CPU, GPU, DSP or whatever computing device that exists in the system. Chapel allows programmer to choose which locale to run a computation. It also enables programmer to say where to reside the data.

Chapel is similar to C++ from a syntactic point of view. Its variable and function definitions are similar to that of Perl and Python. We assume that reader has some basic knowledge about the Chapel language. Therefore we are not going to spend much time on explaining the syntactic details.

The three important aspect of the language which are of interest to us are *task creation*, *task coordination* and *encapsulation* mechanisms. For task creation programmer can use the `begin` keyword before a single statement or a block of statements. This will cause that single statement or that block of statements to be run concurrently with the rest of the program. Figure 6 shows an example of task creation in Chapel. As illustrated in this figure, `task1` will run concurrently with the rest of the program. In other words, the program will not block for the `task1` to complete.

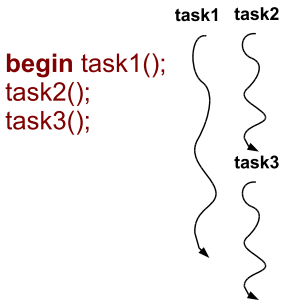

Figure 6: Task creation in Chapel

Task coordination is done through *sync* type variables. Sync type variables in addition to a value maintain a full/empty state. Whenever someone writes to the variable, it will become full and whenever someone reads the content of the variable it will become empty. The default behavior of write and read operations is that they will block for the variable to become empty and full respectively. This behavior then can be used to create a lock. A write operation (acquiring the lock) to the sync variable will not proceed until variable becomes empty (someone release the lock) and a read operation will not proceed unless the variable has been already written into. Figure 7 shows how we can define a sync variable.

Chapel supports object-oriented features in order to leverage the productivity boost in a parallel setting [2]. This is important for us because we want to have encapsulation and the object oriented features will provide that for us. Currently Chapel does not support any access control on the class fields. Chapel team is planning to support that in the future.

```
var lock$: sync bool;
...
lock$ = true;   // block until lock$ is empty
…
lock$;          // block until lock$ is full
```

Figure 7: Sync variables in Chapel

3.1 Why we need actor programming in Chapel?

Chapel is a global view language. It has no notion and means of message passing. Message passing is something that we usually see in local view languages and libraries like Erlang or MPI. So the question is why we might want to do actor programming in Chapel? The motivation behind this work is two-fold.

First motivation is programmability. Chapel is (or wants to be) a general programming language. It wants to provide ordinary programmers to easily program for multi-cores and clusters. In order to do so, it should have enough and strong language constructs to support different known concurrent/parallel programming models. Actor model is a well-known concurrent programming model. It has a nice abstraction which makes concurrent programming easier and less prone to error. Therefore, even if Chapel is a global view language, it still makes sense to support local view constructs and abstractions. This support will increase the usability and programmability of the language and it will boost the productivity.

The second motivation behind actor programming in Chapel is performance. Actor boundaries highlight the cost of communications. This is very important because aside from the sequential computation that is going on inside an actor, the only costly operations are the message passing between actors. Therefore, this fact can be used to create an elegant performance model based on the pattern and the number of messages. In addition as we mentioned before, programs written with the idea of non-shared memory perform and scale better compared to programs written with the idea of a single shared memory.

Besides, actor model has a strong theory behind it which makes the reasoning about actor-based programs easier. There are many tools to model check and verify the systems developed based on actor model.

4 Chapel+Actor

In this section we want to see how we can marry Chapel and Actor model. Before we proceed, let us have a look at the anatomy of an actor. An actor as illustrated in Figure 8 has a mailbox. This mailbox is basically a queue of messages. A single active thread inside the actor will continually get a message from this mailbox and process it atomically. The state of the actor might change as a result of processing the message. Now that we know enough about an actor's anatomy let us see how we can implement this anatomy in Chapel.

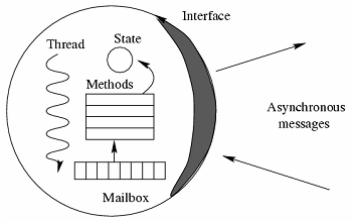

Figure 8: An Actor's Anatomy [5]

There are a number of challenges we are facing here:

1. How to model the mailbox?
2. How to guarantee atomicity in message processing?
3. How we do the asynchronous message passing?
4. How we support locality?
5. How we support programming abstractions such as request-reply and local synchronization constraint?
6. How about location transparency and migration?

4.1 Modeling Mailbox

As we saw previously, Chapel does not support any sort of message passing. We need to somehow simulate message passing. Unfortunately, Chapel lacks a good mechanism for pattern matching of values and parameters. Besides, it lacks flexible data structures like queue and linked list. We first tried to simulate the mailbox using our own defined queue. However, it failed because Chapel arrays are not flexible to support heterogeneous tuples (like that of Scala).

Nonetheless we designed an easy solution to solve both mailbox and atomicity together. The solution is to use a sync variable. There is an implicit queue associated with every sync variable. Whenever a thread is trying to write into that, it will be put into that queue. In other words, the write operations on the sync variables are serialized. Figure 9 shows an illustration of this queue for the sync variable `lock$`.

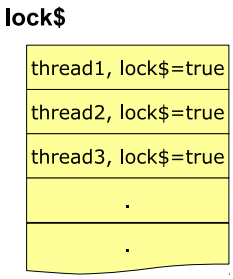

Figure 9: The queue associated with sync variable `lock$`.

Therefore, using a sync variable we can easily simulate the mailbox and messages. What we need to do is to try to write into the sync variable before processing the message. Writing into the sync variable will make it full. As a result, any consequent calls for writing into that will be blocked. Afterwards, we need to read from the sync variable to make it empty. When the sync variable becomes empty another writer (thread) can come in and write into it and process the message.

4.2 Atomicity

In order to ensure the atomicity across the actor we need to acquire the lock (write into that) and release the lock (read from it) before and after the processing of each message respectively. Considering that we represent an actor with a class in Chapel and every message is a function defined in that class, we need to have a class-wide sync variable. Let us call this variable **lock$**. What we need to do next is to alter each function definition to add a write statement and a read statement at the beginning and end of the function block.

```
class actor_name {
  var lock$ : sync bool;
  def msg1() {
    lock$ = true;
    ...  // the message body
    lock$;
  }
  ...
  def msgN() {
    lock$ = true;
    ...  // the message body
    lock$;
  }
}
```

Figure 10: Message bodies are altered with two statements to acquire and release the lock variable `lock$`

From now on, in order to have a concrete example, we are going to have a running example. This is the famous example program used by almost every programming language; the "Hello World!" example. In our actorized version of Hello World we have two actors; the Hello actor and the World actor. The hello actor has a message *hello*. A Hello actor upon receiving the hello message will print the `"Hello "` string on the

console and then it creates a World actor and send a *world* message to that. The World actor upon receiving the world message will print the `"World!"` string on the console.

4.3 Asynchronous and Synchronous Messaging

In order to solve the issue of asynchrony we need to resort to the **begin** keyword to create a new task. As we saw before the tasks created this way run in parallel with the rest of the program. Therefore, every time we want to send an asynchronous message to an actor we simply call the appropriate function from the actor class preceded by a the **begin** keyword.

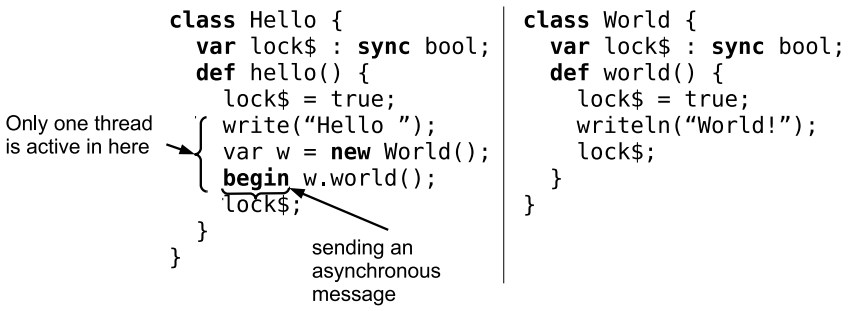

Figure 11: Hello World actors in Chapel

4.3.1 Synchronous Messaging and Return Values

For synchronous or the request-reply style of message passing we simply do not use the `begin` keyword before the call to the appropriate function. Therefore the sender of the message will be blocked until the receiver processes the message. Messages do not have a return value. This is an important thing to consider when writing programs in the message passing style. However, we can simulate a regular function call using the synchronous message passing and a memory reference.

We already know how we can do synchronous message passing. But how do we get the return value? In ordinary programs we have the stack of calls. The callee will put the value onto the stack and the caller will grab it from there. In message passing we do not have this stack but we have memory references. Based on the underlying system and framework, there are a number of ways to implement this. In fact this is just an implementation issue. In Chapel we can take advantage of this fact that the language is global view. Therefore we can use memory references to solve this problem.

By passing a memory reference to the function and let the function set the value of that memory before return we can get the return value from a message. In Chapel there are a couple of *intent* keywords which comes before the formal parameters. One of these keywords is *out*. The effect of out keyword is that the value assigned to the parameter will be set upon the returning from the function. So, we do not pass anything to the function, but the function pass something back to us in that parameter.

```
def mult(a : int, b : int, out x : int) {
  x = a * b;
}
```

Figure 12: Using out intent to get the return value from a message

Figure 12 shows a multiplication function (which could be a message defined in an actor for mathematical operations). As you can see the `x` parameter has intent out. Upon returning from the function the value `a * b` will be assigned to `x`. If we combine this with synchronous messaging, then we can get the request-reply effect. Thanks to Chapel memory model, the implementation of this is very easy.

4.4 Locality

Locality is another issue that we have to deal with. The computation of the messages should be done on the locale of the actor. Actors can be created and placed on different locales using the *on-clause*. In addition

we can use the on clause to run a computation on a specific locale. There is a locale associated with every instance of a class. The clause **on this** select the locale of the current instance represented by `this`. Figure 13 shows a code snippet with an on-clause around it. The code snippet is in fact the body of the hello message from Hello actor. Therefore as we saw, using the on-clause we can achieve the locality.

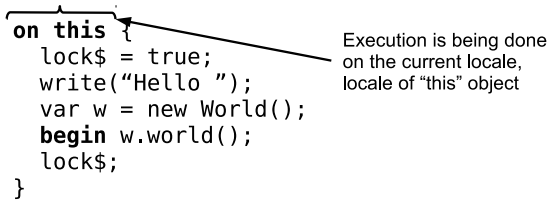

Figure 13: On-clause around the body of the message

Note that the cost of messages passed between actors that are on the same locale is far less than the messages passed between actors on separate locales. Therefore, careful placing of actors on locales could have a dramatic effect on performance improvement.

4.5 Local Synchronization Constraints

In 2.5.2 we discussed what the local synchronization constraints are. The implementation of local synchronization constraints is not easy in Chapel since there is no explicit queue or mailbox to manipulate. As a result we have to do this using only sync variables and locks. The implementation details are in appendix A. Here we are just going to give a general idea behind the implementation.

We saw previously that for local synchronization constraints we need a second queue in addition to the main queue in which we keep the arrived messages. In this second queue we keep the deferred messages. We simulate this queue by using another sync variable. Let us call this second variable lsc$. Each time a message is deferred it will be blocked on this variable. Upon successfully processing of a message without deferring, we need to see if we can process any of the deferred messages. Therefore, the task which has successfully processed the last message is responsible for waking up the deferred messages one by one. A deferred message then will test the condition to see if it can proceed. If it can, it will go and process the deferred message. Otherwise it will go back to the second queue again. We also need a callback mechanism to inform the initiator task when a deferred message is done or is back to the waiting queue.

4.6 Location Transparency and Mobility

Location of actors in Chapel does not affect the computation at all. It only affects the latency. Since Chapel is based on a global address space, all the objects including the actors are part of the same address space. It does not matter on which locale they reside, we can always call their methods and send them message as long as we have a reference to them. This reference is indeed valid on all the locales because of the global view to memory.

Mobility in the context of Chapel means that we should be able to move actors across locales. Right now there is no construct in the language to do this. It is not clear if the programmer can move object across locales by copying their content and create a copy of the object on the destination locale. The question is what is going to happen to all the references to the old object? Who is going to update them? Currently we support no automatic way of moving actors. We need to answer some basic question in order to be able to do that. This is left as future work.

5 Compiler Support for Actors in Chapel

In this section we are going to explain very briefly our compiler support for actor oriented programming in Chapel. What is the role of compiler in actor programming? As we saw in the previous section Chapel has necessary language constructs for actor programming. However, writing all those code for locking, atomicity, asynchrony, locality and local synchronization constraints is a tedious job and is prone to error. For example, just consider writing the local synchronization constraints for all the messages in an actor. It is a lot of work which can be automated. Therefore, in order or alleviate the programmer's job we need some compiler

support so that programmer can easily write an actor program. We have extended Chapel compiler with a syntax for actor definition, message passing and local synchronization constraints. Henceforth, first we are going to show our proposed syntax and then we explain a little bit how we extended the Chapel compiler.

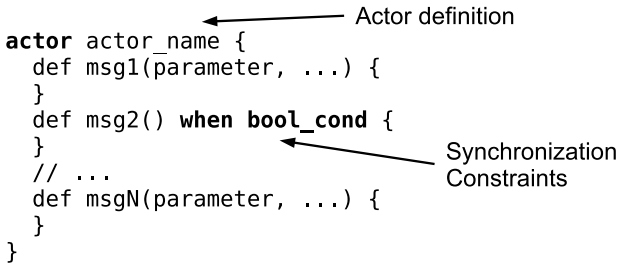

Figure 14: An actor definition in Chapel

Figure 14 shows the syntax for actor definition. It is basically the class definition in Chapel where instead of using the keyword `class` we should use keyword `actor`. The rest is all the same as class definition. There are some restrictions applied to function (message) definitions. No message can have a return value. Because they are regarded as messages and as we said before, messages cannot have return values.

Local synchronization constraint is a condition defined on a message. This condition is a boolean expression defined on the state of the actor and the message parameters. In our proposed syntax we overloaded the **when** keyword (already present in Chapel) to define the constraint. Programmer can write a boolean expression from the parameters and the actor's variables preceded by the when keyword and right after the method signature. The meaning of this boolean expression is that, whenever the expression evaluates to **true**, the message is enabled and can be processed. Otherwise we have to defer its processing. In Figure 14 we can see the syntax for this purpose.

Also, in order to make the message passing easier and more verbose, we have the syntax shown in Figure 15 for asynchronous and synchronous message passing. For asynchronous message passing we use the '@' right before the message name and synchronous message passing is just like usual function call with '.' before the message name.

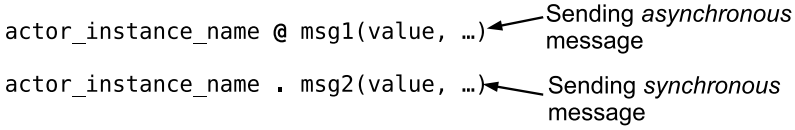

Figure 15: The syntax for sending asynchronous and synchronous messages

The extended compiler does the job of generating the appropriate code for every aspect of the actor program. We check the actor program right after the construction of abstract syntax tree of the program. Then, for each actor defined in the program we generate the necessary Chapel codes. It is basically a source to source transformation.

The compiler can be downloaded from the following URL:
`http://www.cs.utexas.edu/~amshali/chaptor-1.0.zip`
There are a number of examples in the `examples/actor` folder.

6 Conclusion and Future Work

In this work we studied actor model of programming, a highly concurrent model based on message passing. Actor model is important because, first, it is an inherently concurrent model which abstracts the use of threads. In addition, messages inside an actor are processed atomically and no one can access the data encapsulated in the actor. Therefore, programmer does not have to deal with thread creation, scheduling and locking. This is a great deal and makes programming and reasoning about the actor programs easier. On the other hand, Chapel tries to be a general parallel/concurrent programming language. However, it lacks a good message passing model. Our main contribution in this work was to marry the actor model with the Chapel programming language. We designed a complete system for defining actors and sending

messages between actors. We also came up with a design for doing useful programming abstractions such as request-reply messaging and local synchronization constraints in this system. We then extended the Chapel compiler to support a proposed syntax for actor definition.

We believe that doing message passing style programming even in a global view language like Chapel is still beneficial for three reasons:

- Increased programmability: There are programs and algorithms which are inherently designed based on message passing model. Being able to express them in a succinct way is an important matter. Therefore bringing the notion of message passing to the Chapel would increase the capabilities of the language.
- Better performance: Most architectures are going towards a non-shared memory style because of the issue of cache and memory consistency. Programs written in the message passing style with the ability to exploit locality (as in actor model) are able to perform better in non-shared memory architectures.
- Easier performance modeling: The volume of messages passed between actors could be used for modeling the performance of the system based on actors. Everything else is happening locally in each actor and we can use regular sequential program performance modeling for them. As opposed to an arbitrary parallel program which could expose a complicated behavior, actor programs are much easier for performance modeling.

For the future work we are going to study the issue of mobility in Chapel. In particular we need to see what are the difficulties of moving an actor from a locale to another locale. Does simple copying of fields and state of the actor suffice for moving the actor? What is going to happen if actor receive a message in the middle of movement? These questions builds the foundation of our research for future studies in this area.

References

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. February 2005.

[2] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[3] Larry Snyder Calvin Lin. *Principles of Parallel Programming*. Addison-Wesley, 2009.

[4] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, New York, NY, USA, 2009. ACM.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] The e language. *http://www.erights.org/elang*, 2000.

[8] Carlos A. Varela, Gul Agha, Wei-Jen Wang, Travis Desell, Kaoutar El Maghraoui, Jason LaPorte, and Abe Stephens. *THE SALSA PROGRAMMING LANGUAGE 1.1.2 RELEASE TUTORIAL*. Rensselaer Polytechnic Institute, Troy, New York, February 2007.

[9] Microsoft Corporation. Axum programming language. *http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx*, 2008-09.

[10] Mark Astley. *The Actor Foundry: A Java-based Actor Programming Environment*. Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99.

[11] Tim Jansen. Actors guild. *http://actorsguildframework.org/*, 2009.

[12] Mike Rettig. Retlang. *http://code.google.com/p/retlang/*, 2007-09.

[13] Mike Rettig. Jetlang. *http://code.google.com/p/jetlang/*, 2008-09.

[14] Rex Young. Jsasb. *https://jsasb.dev.java.net/*, 2008-09.

[15] S. Rougemaille J.-P. Arcangeli, F. Migeon. Javact : a java middleware for mobile adaptive agents, February 2008.

[16] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: An actor platform for wireless sensor networks. In *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pages 1297-1300*, 2006.

[17] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[18] S. Srinivasan and A. Mycroft. Kilim: Isolation typed actors for Java. In *Procedings if the European Conference on Object Oriented Programming (ECOOP)*, 2008.

[19] Scala actors: A short tutorial. *http://www.scala-lang.org/node/242*, 2010.

[20] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

A Local Synchronization Constraints Implementation

```
def message() {
  on this {
    def lsc() : bool {
      return evaluate_condition;
    }
    var were_blocked = false : bool;
    lock$ = true;
    while (true) {
      // if local synch constraint is true then go on
      if (lsc()) {
        break;
      }
      else {
        // if this is the first time this message is being
        // blocked then add it to the blocked queue and release
        // the lock
        if (!were_blocked) {
          blocked = blocked + 1;
          were_blocked = true;
          lock$;
        }
        // if process already has been blocked before
        // and is going to be blocked again then remove
        // it from the toprocess queue and wake up the
        // initiator.
        else {
          toprocess = toprocess - 1;
          callback$ = true;
        }
        // wait for the next change in actor
        lsc$;
      }
    }

    //// message body
```

```
    ...
    ////
    // if message never blocked then some blocked message
    // might be able to be processed.
    if (!were_blocked) {
      begin {
        // go over all blocked messages and deliver them one by one
        toprocess = blocked;
        while (toprocess > 0) {
          // release one blocked message
          lsc$ = true;
          // wait until the block message callback
          callback$;
        }
        lock$;
      }
    }
    // if message was in the blocked queue then
    // just remove itself from blocked queue and inform
    // the initiator
    else {
      blocked = blocked - 1;
      toprocess = blocked;
      callback$ = true;
    }
  }
}
```