# Parallel Design Patterns in Ptolemy II using Higher-order Actors
# EE290N Project Report

Chang-Seo Park
parkcs@cs.berkeley.edu

Christos Stergiou
chster@cs.berkeley.edu

## Abstract

*The ubiquity of multicore processors nowadays allows for increased performance in applications that are parallelizable. We focus on achieving high performance for a class of data parallel applications which have a static schedule of actor firings. These applications can be run efficiently in parallel and correctly if the actors are stateless. We present a new execution model based on multiplexing receivers on SDF models. We have implemented a new director, receivers, domain specific actors in the Ptolemy II framework. We show that using this model, data parallel applications can be executed with near linear speedup. The receiver multiplexing paradigm can be extended to efficiently execute actor recursion as well.*

## 1 Introduction

The ParLab's vision for the future of parallel programming includes two layers of programmers. The *efficiency layer* programmers are experts of parallel programming and their goal is to provide frameworks and libraries to the *productivity layer* programmers who are usually domain experts who wish to utilize the processing capability of multicore processors to solve problems without worrying about the low-level details. As noted in the Berkeley View paper [1], there are recurring patterns [6] in parallel programs and if these are provided to the productivity layer programmers as frameworks and libraries, it would help them write correct, efficient, and scalable parallel programs easily.

Ptolemy II has a graphical framework that can help domain experts not proficient in textual programming languages. Using diagrams that consist of blocks and arrows along with a rich set of library actors can be much more intuitive and productive. Also, these programmers should not be concerned with low level thread spawning and synchronization to achieve higher performance on multicore processors. In contrast, Ptolemy II computation models, such as synchronous dataflow, impose restrictions that make automatic parallelization and optimization more feasible. The framework allows for extensions that can transparently encapsulate parallelism using custom directors and library actors.

Patterns such as Pipe-and-Filter seem "built-in" to visual languages, as the blocks are the filters with arrows as pipes. Parallelization of such programs in already done in Ptolemy II when using the Process Network [5] director, by assigning one thread to each actor and letting them run concurrently. However, more management needs to take place behind the scenes if we wish to scale linearly with the number of processors. For example, this strategy limits the parallelism to the number of actors. It could also be problematic for a design with too many actors causing unnecessary threading overhead.

We attempt to tackle this problem by separating task parallelism with data parallelism. Task parallelism can be exploited by efficiently distributing threads to independent tasks. In the project we focus on data parallelism, where the same computation is performed on different data. We use the synchronous dataflow model of computation, which has some restrictions on expressiveness, but allows for easier parallelization.

We have made the following contributions in this project:

- Extended the SDF domain by implementing a new director and receivers designed for running dataflow actors on multicore processors
- Implemented two composite actors corresponding to parallel design patterns (A parallel fork-join actor and a recursion actor)
- Evaluated the scalability of the patterns on the new execution model under varying numbers of cores

In the following section we discuss some related work on the parallelization of synchronous dataflow models. In section 3, we describe the implementation details on how we parallelize SDF models. Section 4 presents our experiments with some case studies and their results. We then give an outline for future work in section 5 and conclude the report in section 6.

## 2 Related Work

Synchronous dataflow is a model of computation that restricts all signals to be synchronous with each other. Operationally, this means that all inputs to an actor must be present together, and the output will be present at the same time. This implies that each actor consumes and produces fixed amounts of tokens on each firing, and there cannot be any absent values. This results in the ability to statically compute a schedule of firing sequences. In our work, we make an additional assumption that actors do not have any state (i.e. actors are functional), such that parallelization will not affect the semantics of the actor.

There have been several efforts in the past that attempted to parallelize schedules of the SDF domain. One example is the work on the distributed SDF director in [2]. In that project, concurrency was exposed in the firings of actors that belong in the same topological level of an SDF schedule and in pipelining successive executions of the schedule. In contrast, we targeted multicore processors instead of distributed networks. We run multiple copies of the whole schedule and synchronize when all the spawned schedules are complete.

There has also been some work in automatically parallelizing StreamIt programs. StreamIt is a stream language that includes actors with fixed input and output rates and synchronous dataflow was the starting point of the work. The target of the StreamIt optimizations is multicore architectures as in our case. [4] talks about extracting task, data and pipeline parallelism while we focused on parallel patterns such as fork-join and recursion.

## 3 Implementation

### 3.1 Parallel Fork-Join

The `MultiInstanceComposite` actor creates multiple clones of a model to perform the same work on different data. Using a process network director, these clones can run in parallel. However, the cloning amount and data distribution needs to be explicit. If there are not enough clones, then a multicore processor can be under utilized. If there are too many clones, thread scheduling overhead can hinder the advantages of parallel execution.

Our take on this problem is to run clones on multiple threads for speedup, but controlling the number of spawned threads depending on the environment. We achieve this by creating a new director, which we call MSDF, short for Multicore Synchronous Dataflow. This is the default director for our convenience actor called `ThreadedMultiInstanceComposite`. The execution model is quite different from the original `MultiInstanceComposite`, as we do not explicitly create any clones of the actors in the model. Instead we clone the receivers, calling this the *context* of the cloned model, and execute the model on multiple threads with individual contexts.

### 3.2 Multicore Synchronous Dataflow

The MSDF director is a straightforward extension of the SDF director, so we can statically compute the schedule and token consumption / production rates of each port. To run the model on multiple threads, we need input tokens for each clone of the model. Therefore, we inflate the token consumption and production rate for each port of the composite by the number of worker threads assigned to the director. On each firing of the director (which is triggered by the firing of the composite), we enqueue the statically computed schedule on each of the worker threads. Each worker will iterate through the schedule with its own context.

The context of a worker is implicitly given, by means of multiplexing the receiver between actors under the control of a MSDF director. We implemented a special `MSDFReceiver` for this purpose. An `MSDFReceiver` encapsulates a number of `SDFReceiver`s. Each worker thread can index into its own independent `SDFReceiver`. Since the model is not cloned, every worker will be firing the same actors which are connected to the same `MSDFReceiver`s. We override the `get` and `put` methods of the `MSDFReceiver`, so that each worker accesses its own receiver without any interference from other workers or the need for synchronization. The receivers connected to the input and output ports of the composite need to be treated specially, because we need to distribute and collect tokens in a deterministic order. We differentiate between calls from worker threads and "outside" threads to the `get` and `put` methods, so that we can handle them accordingly.

### 3.3 Recursion Actor

We observe that multiplexing receivers allows to run actors with multiple contexts. This can also be extended to implement recursion more efficiently. Thus we created an actor called `Recurse` that works under the `MSDF` director and acts as a recursive reference to the composite actor that contains it.

The implementation of `Recurse` solves some of the problems of the `ActorRecursion` actor but also enforces some functional limitations on the model that contains it. `Recurse` works only under an MSDF director which, as SDF, is a static scheduling director. Thus it is necessary for the model that contains `Recurse` to execute under the same static schedule (and the same token consumption and production rates) for both the base and the recursive case of the function being modeled. In other words, a

model that fires a different set of actors when some boolean condition holds than when it does not hold cannot be given a static schedule. To overcome that difficulty, we encoded the base case and the corresponding condition as parameters of the `Recurse` actor. The user needs to provide the base case of the recursive function and the boolean guard that causes the actor to recurse or not, as expressions that depend on the input of the actor.

In the initialization of `Recurse`, we create as many input and output ports as the input and outputs ports of the composite that contains the actor. In addition we create a parameter `guard` and a set of parameters that provide the default value of the recursive function in the base case. The number of the default parameters is equal to the number of the outputs of the composite.

When a `Recurse` actor fires, the following steps are taken. First, the input tokens are read and the guard parameter is evaluated with given inputs. If the guard is true, the actor will recurse, otherwise, the default parameters are evaluated and then passed to the output of the actor. In the case of recursion, the inputs of the actor are transferred as inputs to the ports of the composite that contains `Recurse` and the worker starts executing a fresh schedule. To enable recursive execution, each worker is associated with a recursion depth and each MSDF receiver encapsulates multiple stacks of SDF receivers. When a worker makes a recursive call, its depth is increased by one and when it returns from that call the depth is decreased. Furthermore, the `get` and `put` functions of MSDF receivers are modified to be aware of worker depth and to return the corresponding tokens from the stack of SDF receivers.

## 4 Experimental Results

We experimented with our parallel actors on a dual socket quad-core Intel Core 2 Duo 2.0GHz (total 8 cores) with 8GB of RAM. All measurements are reported as an average of 5 runs. For the models using `ActorRecursion`, the initial run-times were much higher than subsequent runs due to actor cloning, rewiring, type checking, etc. In this case, we report the run-times separately.

### 4.1 Mandelbrot Set

Visualizing the Mandelbrot set is a fairly computation-intensive operation. The Mandelbrot set $M$ is formally defined to be the subset of the complex plane where

$$M = \left\{ c \in \mathbb{C} : \sup_{n \in \mathbb{N}} |P_c^n(0)| < \infty \right\}$$

and

$$P_c(z) = z^2 + c\,.$$

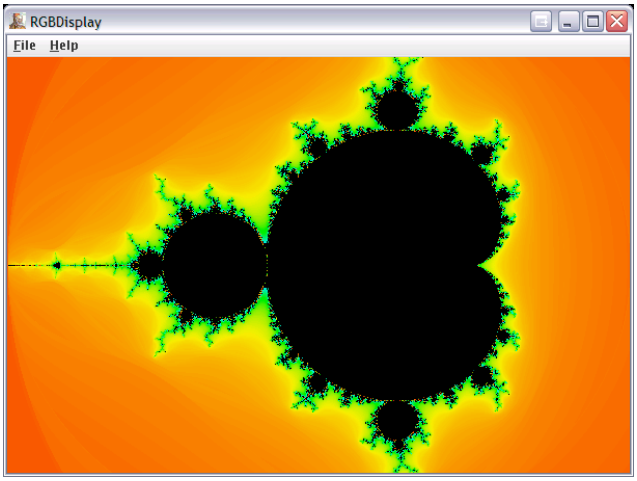


**Figure 1. A sample visualization of the Mandelbrot set where $\Re(c) \in [-2.0, 1.0], \Im(c) \in [-1.0, 1.0]$. The black region corresponds to the complex numbers $c \in M$.**

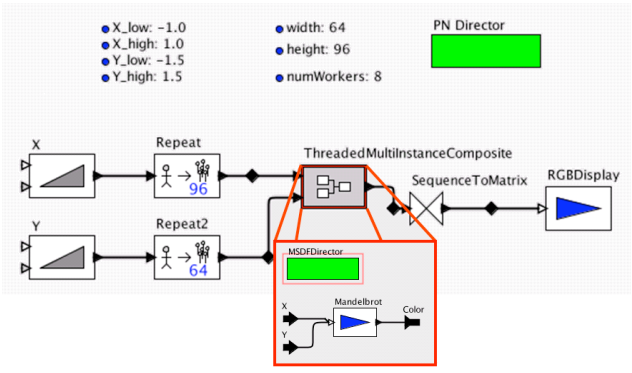


**Figure 2. Model that computes the Mandelbrot set**

The boundary of $M$ is a fractal, and we can approximately calculate whether a number $c$ is in $M$ or not by iteratively calculating $z_{i+1} = P_c(z_i)$ for a finite number of times and using the well-known fact that once $|z_i| > 2$, it will diverge to infinity ("escape") and therefore $c \notin M$. Beautiful visualizations can be made with this set (for example, Figure 1) where a color is assigned to each point in the complex plane whether it is in the set (black) or how fast it escapes.

The computation required at each point is *embarrassingly parallel*. This kind of computation fits well into our parallel fork-join pattern, so we have implemented a model (Figure 2) using our `ThreadedMultiInstanceComposite` actor under the MSDF director. The Mandelbrot actor is a Java

| | p = 3.0 | | | p = 4.0 |
|---|---|---|---|---|
| $n_w$ | $v_f$=1 | 2 | 4 | 2 |
| 1 | 12.5 | 12.3 | 12.1 | 35.4 |
| 2 | 6.9 | 7.1 | 7.4 | 19.2 |
| 4 | 4.2 | 4.3 | 5.1 | 10.6 |
| 8 | 3.5 | 3.4 | 4.2 | 7.2 |

**Table 1. Average run time (s) for calculating the Mandelbrot set for varying parameters. $p$ is the exponent for the Mandelbrot criterion function, $n_w$ is the number of worker threads, and $v_f$ is the vectorization factor.**

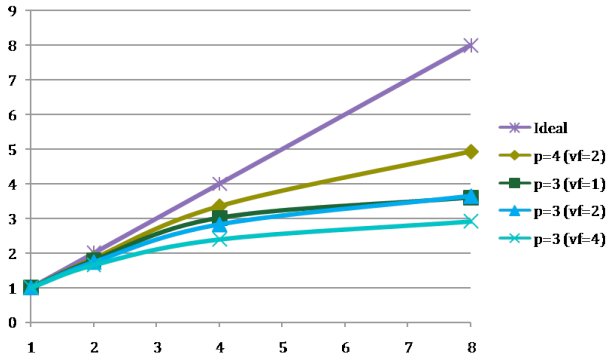


**Figure 3. Speedup obtained by computing the Mandelbrot set on multiple cores**

actor that computes the color for each coordinate.

The results for running the model on different numbers of cores are shown in Table 1 and Figure 3. Since our fork-join actor enforces a deterministic order by synchronizing at each firing, we varied the vectorization factor to measure the overhead of synchronization (a higher vectorization factor makes a worker go through the schedule multiple times before synchronizing). It seems beneficial to reduce synchronizing for the case of one and eight workers, but in the other cases, it actually had an adverse effect.

We also extended the Mandelbrot set criterion function such that $P_c(z) = z^p + c$ for added complexity in calculation to give an effect of a larger problem size. Linear scaling seems to taper off as the number of cores increases, but if we increase the problem size, we can achieve higher scalability.

## 4.2 Fibonacci Sequence

Recursion is not the most efficient method to compute the Fibonacci sequence, but we have implemented two recursive versions using the `Recurse` actor to compare

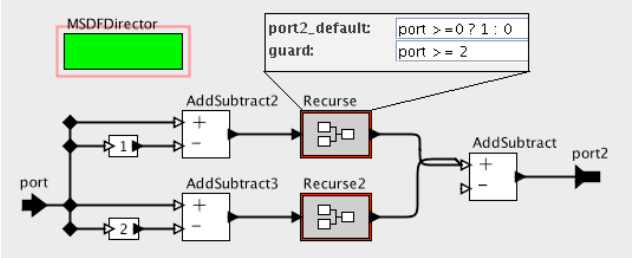


**Figure 4. Model that computes the n-th Fibonacci number recursively**

| | `Recurse` | | `ActorRecursion` | |
|---|---|---|---|---|
| n | $\text{fib}_1(n)$ | $\text{fib}_2(n)$ | $\text{fib}_1(n)$ | $\text{fib}_2(n)$ |
| 10 | 32 | 22 | 909 (12,922) | 62 (542) |
| 20 | 2,627 | 26 | - (>10min) | 101 (1,065) |
| 40 | >3min | 29 | - (-) | 217 (2,633) |

**Table 2. Average run time (ms) for calculating the $n$th Fibonacci number. For `ActorRecursion`, the value in parentheses is the initial run time.**

against the existing `ActorRecursion` implementation. The first version is the most simple and straightforward implementation which calculates in time $O(2^n)$ (shown in Figure 4):

$$\text{fib}_1(n) = \text{if } (n > 1) \quad \text{then } \text{fib}_1(n-1) + \text{fib}_1(n-2) \quad \text{else } n$$

and the second version is more efficient ($O(n)$) by having only one recursive call and being tail-recursive:

$$\text{f}(n, i_1, i_2) = \text{if } (n >= 1) \text{ then } \text{f}(n-1, i_1 + i_2, i_1) \text{ else } i_1$$
$$\text{fib}_2(n) = \text{f}(n, 0, 1)$$

Both versions perform significantly faster than the existing `ActorRecursion` implementation. The results are in Table 2. The exponential version of `fib` obviously does not finish in reasonable time if the input in too high. However, `Recurse` was able to calculate `fib`(20) in 2.6 seconds while `ActorRecursion` could not complete even after 10 minutes. For the linear version, `ActorRecursion` had a much higher initial run time and presumably a much higher constant in the asymptotic time complexity. We conclude from the experiments that our receiver multiplexing strategy is advantageous over explicit actor cloning.

## 5 Future Work

In the current implementation, work is distributed equally to a static number of workers. However, dynamic load balancing techniques can be used to achieve higher performance gains. Also, if multiple parts of the program are parallelizable, a good strategy is needed to distribute the available workers among those parts. SEDA [7] is an architecture framework for servers that dynamically assigns threads to different parts of the workflow called stages. These stages are connected by queues whose lengths are used to measure utilization. If the input queue length of a stage becomes too big, more threads are assigned to the stage. We can adopt a similar strategy to distribute workers, in case multiple `ThreadedMultiInstanceComposite` actors exist in a model.

Using receiver multiplexing for recursion, the original schedule which has a recursive call and the new schedule that is created by recursion can run concurrently. Thus we can apply a work stealing algorithm [3] to execute models which contain `Recurse` actors on multicore processors more efficiently.

## 6 Conclusion

In this project, we explored the use of visual programming to increase the productivity of writing parallel programs. To support parallel execution, we developed an experimental execution model for synchronous dataflow to achieve scalability on multicore processors. Our strategy is to multiplex the receivers between actors and fire the actors in parallel on worker threads. We observed that linear speedup can be achieved for computationally intensive applications. The same strategy was also found helpful for executing recursive models efficiently.

## References


[1] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. *EECS, UC Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

[2] D. L. Cuadrado. *Automated Distribution Simulation in Ptolemy II*. PhD thesis, Aalborg University, April 2008.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[4] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[5] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[6] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[7] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.