Increasing Resilience of Distributed Actor-Based Programming System by Supporting Complex Acyclic, Cyclic Task Dependencies

Youssef Elmougy
School of Computer science,
Georgia Institute of Technology
Atlanta, Georgia, USA
yelmougy3@gatech.edu

***Abstract*—The automatic communication termination protocol proposed in [1] transfers the burden of termination detection and related communication bookkeeping from the programmer to the selector runtime. However, this actor-based programming system is only built to support acyclic linearly task dependent applications. We focus on implementing support for complex acyclic and cyclic task dependencies to increase the resiliency of this distributed actor-based programming system.**

***Index Terms*—Actors, Selectors, C++, Communication Aggregation, Termination Detection, Task Dependency Graph.**

I. Introduction

With the increased need of computation parallelism, the Actor Model [2], which is a model of concurrent computation in distributed systems, was created. Since the publishing of the novel actor model, there have been subsequent models that were created with the goal of improving productivity and scalability. A new actor-based programming system for PGAS applications was introduced in [1]. This new Actor-based system utilizes fine-grained asynchronous actor messages to express point-to-point remote operations in an effort to alleviate programmers of message aggregation and termination detection. The termination graph support implemented in this project will focus on the actor-based system as described in [1].

The actor/selector runtime in this model was created by extending the Habanero C/C++ library ("HClib", a C/C++ asynchronous many-task (AMT) runtime library) [3]. Importantly, it employs a lightweight work-stealing scheduler in order to schedule tasks. Within the application, the programmer can create as many computation tasks as required and offload all remote access computation tasks to a respective communication task (referred to as "mailbox") in an asynchronous manner.

This model has proven to be very efficient and scalable through evaluations on acyclic linearly task dependent applications such as index gather and triangle counting. Although, this model does not currently support all types of task dependency graphs, that including complex acyclic task dependencies (non-linear cases) as well as complex cyclic task dependencies (both linear and non-linear cases). Such task dependencies are apparent in many extensively used applications such as quicksort, breadth-first search, and matrix decomposition, which are gaining popularity within High Performance Computing (HPC) systems.

In an effort to increase resiliency of this distributed actor-based programming system, the project aims to focus on implementing support for acyclic non-linear, cyclic linear, and cyclic non-linear task dependencies.

II. Background

A. Actor-Based Programming System

The Actor model was initially introduced by Carl Hewitt et al. in [4]. It is an asynchronous message-based concurrency model that treats actors as primitives of computation, where actors are inherently isolated from one another and have a no-shared mutable state. Though there might be multiple actors within the system that execute concurrently, it is important to note that an actor will only process messages sequentially within its queue. This property allows for the avoidance of data races and synchronization, since there is no concurrent local execution there will be no concurrent contention for access to local data.

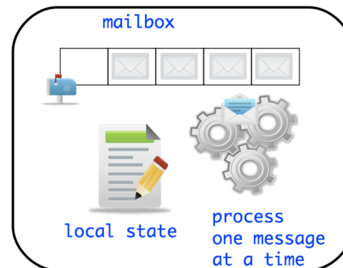

Fig. 1: Actor with a mailbox, a local state, and ability to process one message at a time.

Actors communicate by sending asynchronous messages to one another, hence each actor has a mailbox that stores incoming messages while it processes other messages as shown in Figure 1. At creation, each actor initializes a local state which is only updated through messages it receives and the

intermediate results it computes while processing the message during execution.

There has been work to extend the actor model, as introduced in [5]. Imam et al. proposed the extension Selectors, which preserves the characteristics of actors while providing an abstraction to support synchronization and coordination mechanisms. Selectors gives actors the ability to have multiple guarded mailboxes as seen in Figure 2, where messages can be received by a specific mailbox of the selector (in which case the sender specifies the target mailbox of each message). It's important to notice that an actor can be seen as a selector with one mailbox.

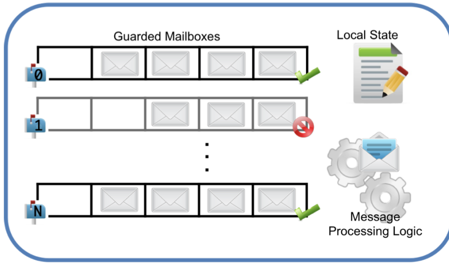

Fig. 2: Selector with multiple guarded mailboxes, a local state, and message processing logic.

In the actor/selector's life cycle, it can be in one of the following defined states:

New An instance of the actor/selector (including its mailboxes) has been created; however, the actor/selector is not yet ready to process messages from its mailboxes. This is shown on Line 26 in Listing 1.

Started An actor/selector moves to this state from the new state when it has been started using the `start` operation, as shown on Line 28 in Listing 1. It can now receive asynchronous messages and process messages from any active mailbox one at a time. While processing a message, the actor/selector should continually receive any messages sent to it without blocking the sender. Messages are sent using the `send` operation, as shown on Line 29 in Listing 1, where the sender specifies the target mailbox as an argument. After processing a message, a message can be selected from any active mailbox to be processed next, without a restriction on the order of message execution.

Terminated The actor/selector moves to this state from the started state when it has been terminated and will not process any messages in its mailbox or new messages sent to it. An actor/selector signals termination by using the `done` operation, as shown on Line 30 in Listing 1, on itself while processing some message.

Listing 1: An example program with a Selector `DepSelector` defined with 2 mailboxes (`A, B`).

```
1 enum MailBoxType {A, B};
2
3 class DepSelector: public hclib::Selector<2, Pkt> {
4 public:
5     DepSelector() : {
6         mb[A].process = [this] (DepPkt pkt, int
              sender_rank) {
7             this->a_process(pkt, sender_rank);
8         };
9         mb[B].process = [this] (DepPkt pkt, int
              sender_rank) {
10            this->b_process(pkt, sender_rank);
11        };
12    }
13
14 private:
15    void a_process(DepPkt pkg, int sender_rank) {
16        send(B, pkg, sender_rank); // user code
17    }
18
19    void b_process(DepPkt pkg, int sender_rank) {
20        // user code
21    }
22 };
23
24 int main(int argc, char* argv[]) {
25    hclib::launch([=] {
26        DepSelector* depSelector = new DepSelector();
27        hclib::finish([=]() {
28            depSelector->start();
29            depSelector->send(A, pkg, pe); // user code
30            depSelector->done(A);
31        });
32    });
33 }
```

Within the started state, actors/selectors can process an incoming message in one of the following ways:

1) It can create one or more new actors/selectors;
2) It can asynchronously send one or more messages to other actors/selectors, given that their target mailbox ID is known;
3) It can change its local state and define how the state will look for the next message it receives.

B. Termination Graphs

As described previously, the `done` operation is used to signal termination for an actor/selector. In the actor model described by Paul et al. [1], the `done` operation is implemented to associate with sending of messages to a mailbox, while letting the runtime track and drain all messages sent to it. This was implemented due to the following two key observations:

1) Sending messages is considered an active part of communication due to the need of explicitly invoking `send`.
2) Receiving messages is considered a passive part of communication because the arrival of a message is not directly under the user's control.

It is important to notice that in Listing 1, the `done` operation is performed only for mailbox `A` and not for mailbox `B`. This is possible since mailbox `B` depends on mailbox `A` - i.e., a message is only sent from mailbox `A` to mailbox `B` in Line 16.

The concept of Termination Graph explains how this is possible. Let us first define that mailbox `Y` *depends* on mailbox `X` if a message is sent to mailbox `Y` in the process function of mailbox `X`.

Based on dependency relations, a directed graph between mailboxes within a selector can be created. Paul et al. [1] gave the assumption that this graph is acyclic, which only deals with a subset of irregular applications. They assume an imaginary

Outside mailbox, which is a virtual mailbox that does not depend on any mailboxes within the selector. A dependency on the Outside mailbox implies a message is received from a non-actor/selector or a different actor/selector from the current distributed one.

Given a termination graph, removing an edge, for instance from X to Y, implies that no more messages will be sent to mailbox Y in the process function of mailbox X. Therefore, the done operation on a specific mailbox corresponds to removing an incoming edge to that mailbox. Using this edge deletion notion, termination of a selector was formulated as follows in [1]:

> *Given a graph whose nodes are mailboxes of a selector and edges represent dependencies between those mailboxes, termination of the selector corresponds to the removal of all edges from this graph.*

Within the program, it is the responsibility of the user to invoke the done operation for those mailboxes that depend on the Outside mailbox. Using the dependency graph, the runtime can then figure out when to perform done on the other dependent mailboxes.

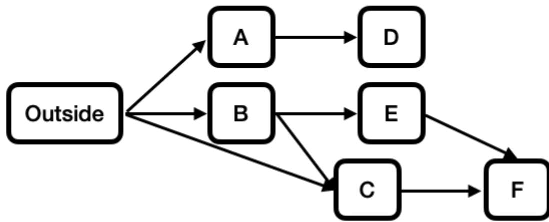

Fig. 3: Sample Termination graph with nodes representing mailboxes and edges representing dependencies.

Figure 3 shows a sample mailbox dependency graph in which an arrow from B to E implies that mailbox E depends on mailbox B. In the given case, the user is required to invoke the done operation only for mailboxes A, B, and C. The runtime then deduces when done should subsequently be invoked on mailboxes D, E, and F.

Once the user performs done(A) on a partition, no more messages can be *sent* from mailbox A from that partition. However, it can still continue to *receive* and *process* messages. This implies that messages can be sent from the process method on any partition of mailbox A to mailbox D. Therefore, the runtime needs to ensure that done(A) is invoked on all partitions and wait for all messages to be drained from all partitions of mailbox A. At this stage, no more messages can be sent to mailbox D since it was dependent on mailbox A. Hence, the runtime now invokes done(D). If a mailbox has multiple predecessors, such as mailbox F, the runtime waits for the termination of all the predecessors before terminating the mailbox.

III. Literature Review

It is important to dive into detail regarding different types of task dependencies, as well as applications that are Representative of each.

Task dependencies have two characteristics: 1) the number of predecessors and successors of each node (1 or multiple), and 2) whether dependencies are acyclic or cyclic. In order to increase the resilience of the distributed actor-based system, support for each of these dependencies is required. The following looks at possible types of dependency graphs within the acyclic and cyclic cases, as well as provide representative applications of the type:

A. Acyclic Task Dependencies

Within the acyclic case, we can have the following different types according to the first characteristic of task dependencies:

1) Nodes with 1 predecessor and 1 successor: In this case, each mailbox can be dependent on only one other mailbox and can invoke a dependency on only one other mailbox as shown in Figure 4. In the actor model defined in [1], this was the only pattern that was considered, producing a linear graph as a dependence pattern.

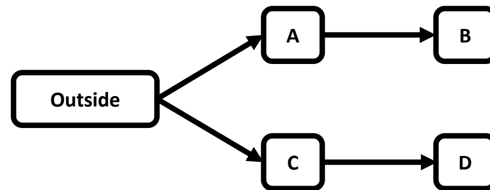

Fig. 4: Acyclic Termination graph where nodes have only one predecessor and one successor.

The real world applications that fall within this dependency pattern include histogram, index gather, topological sort, triangle counting, triangle centrality, jaccard index, page rank, and some other applications from the Bale Kernels [6] that were used to test the correctness of the actor model in [1].

2) Nodes with 1 predecessor and multiple successors: In this case, each mailbox can be dependent on only one other mailbox but can invoke dependencies on multiple other mailboxes as shown in Figure 5.

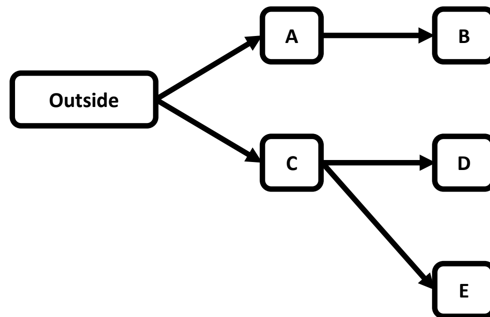

Fig. 5: Acyclic Termination graph where nodes have only one predecessor but multiple successors.

The real world applications that fall within this dependency pattern include quicksort, Fibonacci, binary tree, BFS, DFS, and strongly connected components.

3) Nodes with multiple predecessors and multiple successors: This is the complete case where each mailbox can be dependent on multiple other mailboxes and can invoke dependencies on multiple other mailboxes as shown in Figure 6.

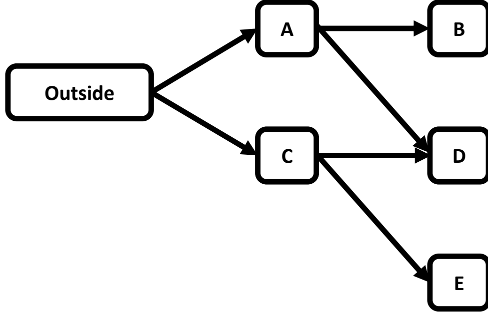

Fig. 6: Acyclic Termination graph where nodes have multiple predecessors and multiple successors.

The real world applications that fall within this dependency pattern include Fast Fourier Transform (FFT).

B. Cyclic Task Dependencies

Cyclic task dependencies are expected to be explored within Milestone 3's work, this section will be updated accordingly.

IV. Related Work

From my research, there is no prior work related to task dependency graph terminations specific to actor-based programming systems. Although, there have been several studies done related to task dependencies in parallel systems. Slaughter et al. [7] designed Task Bench, a parameterized benchmark to evaluate parallel runtime performance. These benchmarks explore the performance of distributed systems and allows for a wide variety of benchmark scenarios that are evident in real world applications. They evaluate several task-based applications such as Stencil and FFT.

Lin et al. [8] proposed an efficient work-stealing scheduler for task dependency graphs. This is not directly related to our study, although it provided insights into task dependencies within graph algorithms as well as an evaluation of scalability analysis that was utilized.

V. Implementation

In this section, we discuss the current termination graph implementation within the actor system described in [1], our extended acyclic termination graph implementation, and our extended cyclic termination graph implementation.

A. Current Termination Graph Implementation

In [1], only the acyclic termination graph shown in Figure 4 is implemented. In that case, we only deal with nodes that have one predecessor and one successor. Listing 2 shows the implementation of the `done` operator within the system, which takes the mailbox ID, `mb_id`, as a parameter to the function. When a user invokes `done(mb_id)`, the following occurs:

1) The mailbox with ID `mb_id` begins its termination phase by calling `done()` on itself on Line 2.
2) The worker thread asynchronously waits upon the completion of the loop on Lines 3 and 12, and within the loop the following occurs:
   a) The variable holding the number of mailboxes terminated (`num_work_loop_end`) is incremented on Line 4.
   b) If we have terminated all `N` mailboxes within the system, then we go into Lines 8-11 in the if statement, and send out a "promise" (signal of completion) to the main worker loop indicating that the termination process for all mailboxes has been completed and that the main program can now terminate.
   c) If we have not yet terminated all `N` mailboxes within the system, then we go into Line 6 and call `done()` on the next mailbox within the sequence (the mailbox with ID `mb_id+1`).

Listing 2: Algorithm associated with the `done` operation.

```
1  void done(int mb_id) {
2     mb[mb_id].done();
3     hclib::async_await_at([=]() {
4        num_work_loop_end++;
5        if(num_work_loop_end < N) {
6           done((mb_id+1)%N);
7        }
8        else {
9           assert(num_work_loop_end == N);
10          end_prom.put(1);
11       }
12    }, mb[mb_id].get_worker_loop_finish(), nic);
13 }
```

We can use Figure 4 as an example to show this process. Let's assume the user invokes `done(A)` and then `done(C)`, the following will occur:

1) **`done(A)` is called**. Line 2 will begin the termination phase for mailbox `A`. In Line 4, we increment `num_work_loop_end` to 1. Since `num_work_loop_end < N`, **`done(B)` is called** (since it is the next mailbox in the linear sequence). When `done(B)` is called, Line 2 will begin the termination phase for mailbox `B`. In Line 4, we increment `num_work_loop_end` to 2. Since `num_work_loop_end < N` and there are no more mailboxes within the sequence, we wait until all mailboxes are terminated.
2) **`done(C)` is called**. Line 2 will begin the termination phase for mailbox `C`. In Line 4, we increment `num_work_loop_end` to 3. Since `num_work_loop_end < N`, **`done(D)` is called** (since it is the next mailbox in the linear sequence). When `done(D)` is called, Line 2 will begin the termination phase for mailbox `D`. In Line 4, we increment `num_work_loop_end` to 4. Since `num_work_loop_end == N`, we know that we have terminated all mailboxes and hence we send the "promise" signal on Line 10 and exit from the loop.

B. Acyclic Termination Graph Implementation

In this project, we built upon the current implementation and added support for the acyclic task dependencies as seen in Figure 5 (nodes with one predecessor and multiple successors) and Figure 6 (nodes with multiple predecessors and multiple successors). It is clear that if support is implemented for the latter case, then it will cover the former case as well. Listing 3 shows the implementation of the `done_extended` operator within the system, which takes the mailbox ID, `mb_id`, as a parameter to the function.

Listing 3: Algorithm associated with the `done_extended` operation.

```
1 void done_extended(int mb_id) {
2    mb[mb_id].done();
3    hclib::async_await_at([=]() {
4       num_work_loop_end++;
5       vector<int> successor_mailboxes = mb[mb_id].
            get_dep_mailboxes();
6       if (successor_mailboxes.size() > 0) {
7          for (int const& i : successor_mailboxes) {
8             int num_predecessors = mb[i].
                  get_predecessor_count();
9             if (num_predecessors == 1) {
10               done_extended(i);
11            } else {
12               mb[i].dec_predecessor_count();
13            }
14         }
15      }
16      else if (num_work_loop_end == N) {
17         end_prom.put(1);
18      }
19   }, mb[mb_id].get_worker_loop_finish(), nic);
20 }
```

The assumption made in this approach is that the user provides a list of predecessors and a list of successors for each mailbox initialized within the definition of the `Selector` class. This can be seen on Lines 9, 14, 19, 24, and 29 in Listing 4 (a sample program), where the utility function `add_dep_mailboxes()` is utilized (function implementation can be seen in Listing 5). These two lists are local to each mailbox. When a user invokes `done(mb_id)`, the following occurs:

1) The mailbox with ID `mb_id` begins its termination phase by calling `done()` on itself on Line 2.
2) The worker thread asynchronously waits upon the completion of the loop on Lines 3 and 19, and within the loop the following occurs:
   a) The variable holding the number of mailboxes terminated (`num_work_loop_end`) is incremented on Line 4.
   b) All the successor mailboxes of mailbox `mb_id` are grabbed and stored in a vector on Line 5.
   c) If the current mailbox has no successor mailboxes and we have terminated all `N` mailboxes within the system, then we go into Lines 16-18 in the if statement and send out a "promise" (signal of completion) to the main worker loop indicating that the termination process for all mailboxes has been completed and that the main program can now terminate.
   d) If the current mailbox has successor mailboxes, then we go into Lines 7-14 and loop through each successor mailbox in Line 7.
      i) The number of predecessors for mailbox `mb_id` is grabbed on Line 8.
      ii) If the number of predecessors is 1, then we know that this is the last predecessor mailbox, hence we call `done_extended` on the successor mailbox on Line 10.
      iii) If the number of predecessors is not 1, then we know that the successor mailbox has other predecessors that have not completed execution, hence we cannot terminate the successor mailbox, and instead we decrement the number of pending predecessors on Line 12.

Listing 4: A program with a Selector `DepSelector` defined with 5 mailboxes (`A, B, C, D, E`) that creates the task dependency graph in Figure 6.

```
1 enum MailBoxType {A, B, C, D, E};
2
3 class DepSelector: public hclib::Selector<5, Pkt> {
4 public:
5    DepSelector() : {
6       mb[A].process = [this] (DepPkt pkt, int
            sender_rank) {
7          this->a_process(pkt, sender_rank);
8       };
9       mb[A].add_dep_mailboxes({}, {B,D});
10
11      mb[B].process = [this] (DepPkt pkt, int
            sender_rank) {
12         this->b_process(pkt, sender_rank);
13      };
14      mb[B].add_dep_mailboxes({A}, {});
15
16      mb[C].process = [this] (DepPkt pkt, int
            sender_rank) {
17         this->c_process(pkt, sender_rank);
18      };
19      mb[C].add_dep_mailboxes({}, {D,E});
20
21      mb[D].process = [this] (DepPkt pkt, int
            sender_rank) {
22         this->d_process(pkt, sender_rank);
23      };
24      mb[D].add_dep_mailboxes({A,C}, {});
25
26      mb[E].process = [this] (DepPkt pkt, int
            sender_rank) {
27         this->e_process(pkt, sender_rank);
28      };
29      mb[E].add_dep_mailboxes({C}, {});
30   }
31
32 private:
33   void a_process(DepPkt pkg, int sender_rank) {
34      send(B, pkg, sender_rank); // user code
35      send(D, pkg, sender_rank); // user code
36   }
37
38   void b_process(DepPkt pkg, int sender_rank) {
39      // user code
40   }
41
42   void c_process(DepPkt pkg, int sender_rank) {
43      send(D, pkg, sender_rank); // user code
44      send(E, pkg, sender_rank); // user code
45   }
46
47   void d_process(DepPkt pkg, int sender_rank) {
48      // user code
49   }
50
51   void e_process(DepPkt pkg, int sender_rank) {
52      // user code
53   }
54 };
55
```

```
56 int main(int argc, char* argv[]) {
57   hclib::launch([=] {
58     DepSelector* depSelector = new DepSelector();
59     hclib::finish([=]() {
60       depSelector->start();
61       depSelector->send(A, pkg, pe); // user code
62       depSelector->send(C, pkg, pe); // user code
63       depSelector->done_extended(A); // user code
64       depSelector->done_extended(A); // user code
65     });
66   });
67 }
```

We can use Figure 6 as an example to show this process. Let's assume the user invokes `done_extended(A)` and then `done_extended(C)` (Line 63 and 64 in Listing 4), the following will occur:

1) **`done_extended(A)` is called**. Line 2 will begin the termination phase for mailbox `A`. In Line 4, we increment `num_work_loop_end` to 1. We then grab {`B, D`} as the successor mailboxes in Line 5. Since we have successors, we loop through each on Line 7.
   a) For mailbox `B`, the number of predecessors is 1, so **`done_extended(B)` is called**.
      i) When `done_extended(B)` is called, Line 2 will begin the termination phase for mailbox `B`. In Line 4, we increment `num_work_loop_end` to 2. We then grab {} as the successor mailboxes in Line 5. Since we do not have successors and `num_work_loop_end < N`, we wait until all mailboxes are terminated.
   b) For mailbox `D`, the number of predecessors is 2, so we decrement the number of waiting predecessors on Line 12.
2) **`done_extended(C)` is called**. Line 2 will begin the termination phase for mailbox `C`. In Line 4, we increment `num_work_loop_end` to 3. We then grab {`D,E`} as the successor mailboxes in Line 5. Since we have successors, we loop through each on Line 7.
   a) For mailbox `D`, the number of predecessors is 1, so **`done_extended(D)` is called**.
      i) When `done_extended(D)` is called, Line 2 will begin the termination phase for mailbox `D`. In Line 4, we increment `num_work_loop_end` to 4. We then grab {} as the successor mailboxes in Line 5. Since we do not have successors and `num_work_loop_end < N`, we wait until all mailboxes are terminated.
   b) For mailbox `E`, the number of predecessors is 1, so **`done_extended(E)` is called**.
      i) When `done_extended(E)` is called, Line 2 will begin the termination phase for mailbox `D`. In Line 4, we increment `num_work_loop_end` to 5. We then grab {} as the successor mailboxes in Line 5. Since we do not have successors and `num_work_loop_end == N`, we know that we have terminated all mailboxes and hence we send the "promise" signal on Line 17 and exit from the loop.

Listing 5: Algorithm associated with the `add_dep_mailboxes` operation.

```
1 void add_dep_mailboxes(list<int>
      predecessor_mailboxes, list<int>
      successor_mailboxes) {
2   // deal with predecessors
3   predecessor_count = predecessor_mailboxes.size();
4   // deal with successors
5   for (int const& mb_id : successor_mailboxes)
        dependency_mailboxes.push_back(mb_id);
6 }
```

C. Cyclic Termination Graph Implementation

Cyclic task dependencies are expected to be explored within Milestone 3's work, this section will be updated accordingly.

VI. Next Steps: Milestone 2 and Final Submission

Milestone 2 consisted of adding functionality for the acyclic cases, although I am ahead of schedule and have implemented that. Therefore, for the next milestone I will have implemented a real world application to test out the dependencies. (SUBMISSION DATE: 11/4)

For the final milestone, I am still on track to add functionality for the cyclic cases, although from discussion and the literature review it seems as this case will prove to be a lot more difficult. (SUBMISSION DATE: 12/9)

For the Final submission, the plan was to write a full report on the project. In this milestone, I wrote the parts of the report regarding the introduction, background, literature review, related work, and the acyclic implementation. For the final submission I will build upon these sections and complete the cyclic implementation part. (SUBMISSION DATE: 12/9)

Artifact

Repository: https://github.com/youssefelmougy/hclib/tree/bale_actor

Termination Graph Implementation: https://github.com/youssefelmougy/hclib/blob/bale_actor/modules/bale_actor/inc/selector.h

Test Benchmarks: https://github.com/youssefelmougy/hclib/blob/bale_actor/modules/bale_actor/benchmarks/dependancy_selector.cpp

References

[1] S. R. Paul, A. Hayashi, K. Chen, and V. Sarkar, "A productive and scalable actor-based programming system for pgas applications," in *International Conference on Computational Science*. Springer, 2022, pp. 233–247.

[2] G. Agha, *Actors: a model of concurrent computation in distributed systems*. MIT press, 1986.

[3] M. Grossman, V. Kumar, N. Vrvilo, Z. Budimlic, and V. Sarkar, "A pluggable framework for composable hpc scheduling libraries," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 723–732.

[4] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3. Stanford Research Institute Menlo Park, CA, 1973, p. 235.

[5] S. M. Imam and V. Sarkar, "Selectors: Actors with multiple guarded mailboxes," ser. AGERE! '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2687357.2687360

[6] F. M. Maley and J. G. DeVinney, “A collection of buffered communication libraries and some mini-applications.” https://github.com/jdevinney/bale, 2020.

[7] E. Slaughter, W. Wu, Y. Fu, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney *et al.*, “Task bench: A parameterized benchmark for evaluating parallel runtime performance,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[8] C.-X. Lin, T.-W. Huang, and M. D. Wong, “An efficient work-stealing scheduler for task dependency graph,” in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2020, pp. 64–71.