Parallel Design Pattern
Part 2 - Report

B232688

We developed a distributed actor version of the traffic simulation code that supports multiple actors per MPI process. To achieve this, we created the **Parallel Actor Model** (`PAM`) framework, implemented in C++ with MPI. Section 1 details the design and implementation of the framework, as well as instructions for using its API. In Section 2, we explain how the `PAM` framework is applied to implement the actor-based version of the traffic simulation program, along with the results from performance and strong scaling experiments.

Contents

1 Actor Model Framework

The **Actor Model** is a way of managing concurrency by structuring computation around independent entities called **actors**. An actor has a *state* and a *behavior*, where the state represents its internal data, and the behavior defines the task it executes. It communicates with other actors through message passing only, ensuring that its state remains unshared. This section describes the **Parallel Actor Model** (`PAM`) framework, a simple C++ implementation of the actor model built on the MPI standard, allowing the user to write an application that targets a single core machine or an HPC system. In Section 1.1, we discuss the design and implementation of `PAM`, covering its core components, actor life cycle, messaging system, workload distribution, scheduling strategies, and limitations. In Section 1.2, we explain how to use the framework, presenting it as an API along with a sample application.

1.1 Design and Implementation

1.1.1 Core Components

The PAM framework consists of two core components: Actor and Mailbox. An Actor operates autonomously, executing its own tasks independently of other actors. Some of these tasks may involve communication with other actors. Each actor has a dedicated Mailbox, uniquely identified by an Address. Actors interact by sending messages to one another's mailboxes using these addresses.

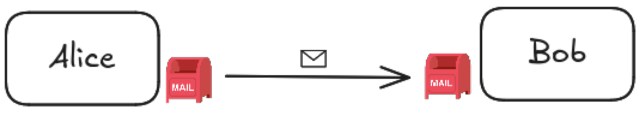


Figure 1: Alice sends a message to Bob via the Mailbox API

The PAM framework, built on the MPI standard, implements actors as entities executed by MPI processes. **It supports managing and executing multiple actors within each MPI process**. An actor managed by a specific MPI process is uniquely identified by the tuple (rank, tag), where rank denotes the rank of the MPI process, and tag is an integer. In particular, this tuple is the Address of the actor's Mailbox.

1.1.2 Actor Life Cycle

Figure 2 shows the life cycle of an actor. It begins with an initialization phase consisting of three steps. First, it calls its pre_barrier_init method to initialize the actor's state. Next, it waits for all other actors to finish executing their respective pre_barrier_init methods. Finally, it calls its post_barrier_init method to complete any remaining initialization tasks such as setting up time-sensitive variables.

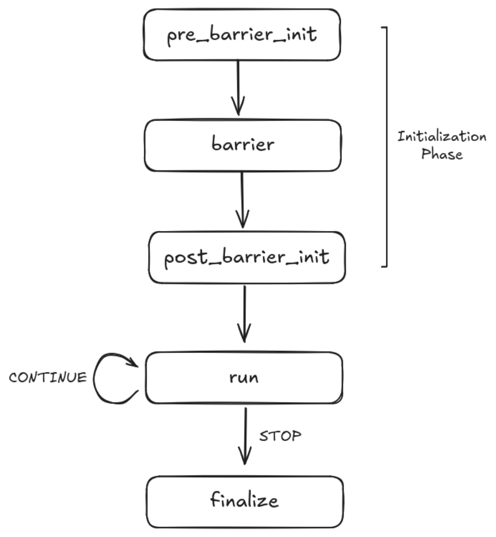


Figure 2: Actor's life cycle showing its initialization phase and run phase.

The actor then proceeds to execute its tasks by calling its run method. Since each MPI process manages multiple actors, it calls the run method of each actor it oversees once per iteration, in sequence. The actor's run method can return either a CONTINUE or STOP constant. As long as the run method returns CONTINUE, the actor remains in the execution cycle and its run method is called again in the next iteration. If the method returns STOP, the actor is removed from the execution cycle.

1.1.3 Messaging System

Actors communicate by sending messages to one another's Mailbox. The Mailbox provides three methods to enable message passing:

- `send`: Sends a `Message` object to an actor identified by a specific ID.
- `receive`: A blocking operation to retrieve a `Message` object.
- `hasMessage`: Checks whether a `Message` is available for retrieval.

The actor's `Mailbox` is accessible within its `run` method, enabling the actor to communicate with other actors while executing its tasks. This mechanism alone is sufficient to enable message passing between any pair of actors. For added convenience, the framework offers a mode called **ingress**, which, when activated, modifies the actor's run phase to include a dedicated stage for processing incoming messages. As shown in Figure 3, the actor's run phase alternates between the `ingress` method and the `run` method. If a message is available, it is forwarded to the `ingress` method as a parameter for processing. When no messages are present, control returns to the `run` method. Additionally, the framework includes a configurable parameter that limits the number of times the `ingress` method can be called before control is returned to the `run` method, ensuring the `run` method is not starved of execution time.

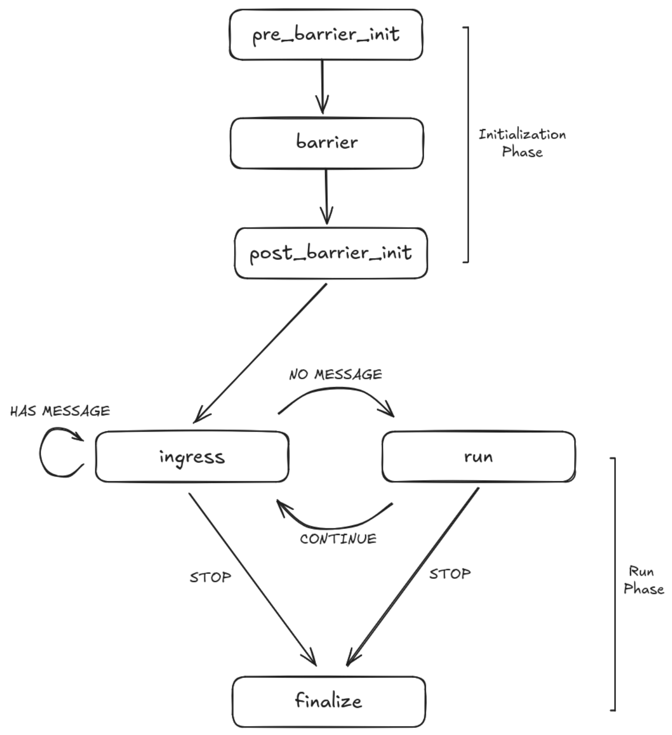


Figure 3: Actor's life cycle when ingress mode is activated.

1.1.4 Workload Distribution

Each MPI process can manage the execution of multiple actors. The framework differentiates between two types of actors when assigning them to an MPI process: *grouped actors* and *isolated actors*. An MPI process managing a group of actors is said to be handling grouped actors, which are added to the framework using the `addActor` method. Conversely, an MPI process managing a single actor is said to be handling an isolated actor, which is added to the framework using the `addIsolatedActor` method. When the workload distribution is uneven across actors, some may require more intensive computations. In such cases, consider adding these actors to the framework as isolated actors.

1.1.5 Summary of Features and Limitations

In summary, the `PAM` framework supports the following features:

- Multiple actors per MPI process.

- An MPI process can manage exactly one actor.
- Message passing between actors via the `Mailbox` API.
- Ingress mode separating the concerns of message processing and task computation.
- Actor initialization divided into stages with built-in synchronisation.

A key limitation of the framework is that actors cannot spawn new actors, meaning the number of actors is fixed and determined at the start.

1.2 How to Use

1.2.1 Driver Code

To use the `PAM` framework, write a driver code that carries out the following steps:

(1) Call `MPI_Init`

(2) Create an instance of `ParallelActorModel`

(3) Add actors via the `addActor` or `addIsolatedActor` methods

(4) Add message types via the `addType` method (optional)

(5) Start the execution cycle by calling the `start` method

(6) Call `MPI_Finalize`

Each MPI process runs the same driver code. That is, every MPI process creates its own instance of `ParallelActorModel` and adds the same set of actors. However, only a subset of these actors will be retained in the framework's internal state for each process. When actors are added using the `addActor` method, they are assigned to MPI processes through bin-packing scheduling, starting from MPI rank 0 and proceeding sequentially. For instance, consider the following calls to `addActor`: `addActor(Alice)`, `addActor(Bob)`, `addActor(Charlie)`, `addActor(Dave)`, `addActor(Eve)`, and `addActor(Frank)`, in this order. In a framework configured to support four actors per MPI process, process 0 would manage `(Alice, Bob, Charlie, Dave)`, while process 1 would manage `(Eve, Frank)`.

A potential concern with the approach above is that it might appear memory-inefficient for each MPI process to instantiate the same set of actors in the driver code. However, it is important to note that these actors remain uninitialized at this stage. Initialization occurs only after all actors have been assigned to MPI processes and the `start` method is invoked. Users are responsible for ensuring that actors have a relatively small memory footprint in their uninitialized state. Finally, we end this subsection by showing the signature of the framework's constructor:

```
ParallelActorModel(
    int num_actors_per_procs,
    bool ingress_mode = false,
    bool log_debug = false,
    int max_num_message_per_iteration = MAX_NUM_MESSAGE_PER_ITERATION);
```

The parameter `num_actors_per_procs` denotes the number of actors per MPI process. To activate ingress mode, set its value to true. When `log_debug` is set to true, additional debug statements are printed on the console. Lastly, the `max_num_message_per_iteration` indicates the maximum number of messages the `ingress` method handles before handling control back to the `run` method.

1.2.2 Creating Actors

To define an actor, you need to define a class that implements the abstract `Actor` class. The minimum requirements for sub-classing the `Actor` class are:

- Defining a constructor that calls the `Actor(int id)` constructor. This was made mandatory by deleting the default constructor of `Actor`, ensuring that each subclass of `Actor` is instantiated with an id.
- Overriding the `run` method in the following snippet of the `Actor` class declaration:

  ```
  class Actor {
  public:
      actor::id id;
      mail::Mailbox mailbox;
      Actor() = delete;
      explicit Actor(int id);
      virtual bool pre_barrier_init();
      virtual bool post_barrier_init();
      virtual next_step ingress(mail::Message &message);
      virtual next_step run() = 0;
      virtual ~Actor();
      void finalize();
  };
  ```

  Overriding the other virtual methods is optional.

After actors are defined and instantiated, they can be added to the framework using the `addActor` method for a grouped actor, or `addIsolatedActor` for an isolated actor. For a discussion on workload distribution and the different categories of actors, refer to section 1.1.4.

1.2.3 Using the Mailbox API

Actors communicate by sending a `Message` to another actor's `Mailbox`. To send data to another actor, the data must be encapsulated in a `Message` object, which includes the following three mandatory fields:

- `MPI_Datatype mpi_datatype`: The type of a single element of the data.
- `int count`: The total number of elements in the data.
- `void* data`: A pointer to the message payload.

For the framework to recognize the `MPI_Datatype` of the data used by actors in a `Message` object, these types must be registered. A message type can be registered by invoking the `addType` method. The sample code in Section 1.2.4 demonstrates how this is done. After data has been wrapped in a `Message` object, it can be sent or received via the `Mailbox` API.

- `send`: Sends a `Message` object to an actor identified by a specific ID.
- `receive`: A blocking operation to retrieve a `Message` object.
- `hasMessage`: Checks whether a `Message` is available for retrieval.

As discussed in section 1.1.3, the `Mailbox` API is accessible within the actor's `run` method. Alternatively, section 1.1.3 introduces a messaging mode supported by the framework called **ingress mode**. When enabled, message processing is managed by a separate method called `ingress`, which the user can define. A detailed example of how ingress mode is utilized can be found in the source code of the traffic simulation program.

1.2.4 Sample Application - Hello World

Let's create a "Hello World!" program using the `PAM` framework. The application will consist of two actors: `Bob` and `Alice`. In this program, `Bob` sends the string "Hello World!" to `Alice`, who then prints it to the console.

We start by defining the Bob class by implementing the abstract Actor class. Listing 1 shows the constructor of Bob, which calls the Actor(int id) constructor to assign the actor's ID to the specified value. Additionally, it demonstrates the actor's task in the definition of the run method: sending a Message with the string "Hello World!" as the data payload to an actor with ID 2.

```
class Bob : public actor::Actor {
public:

    Bob(int id) : Actor(id) {};

    actor::next_step run() override {

        auto payload = "Hello␣World!";

        // Create message
        mail::Message message;
        message.count = strlen(payload) + 1;
        message.mpi_datatype = MPI_CHAR;
        message.data = (void *) payload;

        // Send message
        auto receiver_id = 2;
        mailbox.send(message, receiver_id);

        return actor::STOP;
    }
};
```

Listing 1: Class Bob implments abstract class Actor.

Next, we define the Alice class, as illustrated in Listing 2. Each time the run method is invoked, it checks for the presence of a message in the mailbox. If a message is available, it is received and printed to the screen. If no message is found, the method returns the CONTINUE constant, allowing it to be invoked again in the next iteration of the execution cycle.

```
class Alice : public actor::Actor {
public:

    Alice(int id) : Actor(id) {};

    actor::next_step run() override {

        if (mailbox.hasMessage()) {
            auto message = mailbox.receive();
            printf("%s\n", (char *) message.data);
            return actor::STOP;
        } else {
            // Invoke run method again
            return actor::CONTINUE;
        }
    }
};
```

Listing 2: Class Alice implments abstract class Actor.

With all actors defined, we instantiate each one and use the framework to execute them, as demonstrated in Listing 3.

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    // Create instance of actor model framework
    auto framework = ParallelActorModel(5);

    // Create actors
    auto bob = new Bob(1);
    auto alice = new Alice(2);

    // Register actors
    framework.addActor(bob);
    framework.addActor(alice);

    // Register any MPI_Datatype used by the actors
    framework.addType(mail::Type{sizeof(char), MPI_CHAR});

    // Initialize and run actors
    framework.start();

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Listing 3: Driver code for "Hello World" program.

Although straightforward in purpose, the "Hello World!" program illustrates the fundamental concepts of using the PAM framework. A more advance sample application in `user/sum_reduction` is included as part of the code submission. For an application that utilizes all of the features of the `PAM` framework, refer to the source code of the traffic simulation program.

2 Traffic Simulation

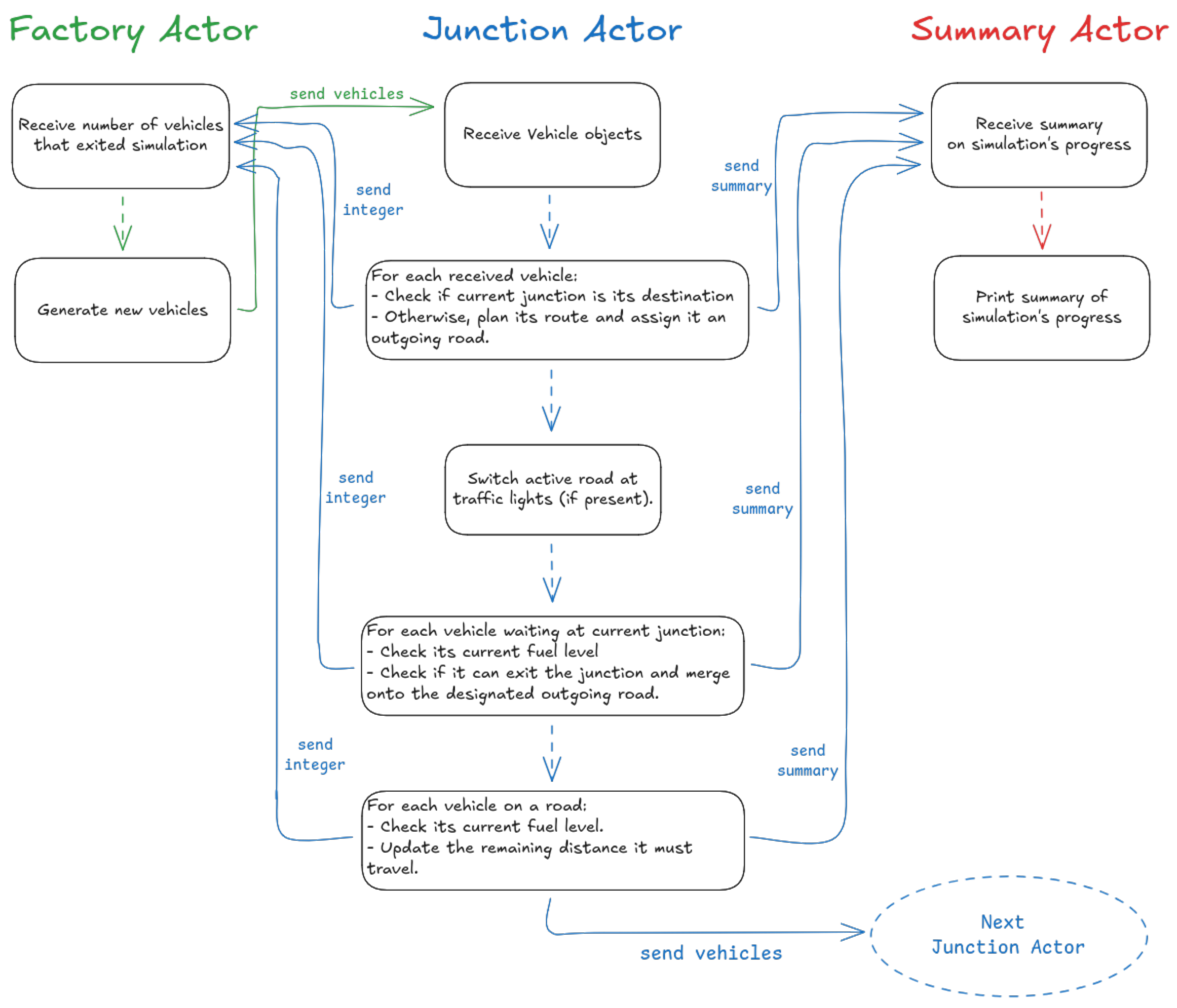
Using the `PAM` framework, we implement a distributed actor version of the traffic simulation code that supports multiple actors per MPI process. Section 2.1 outlines the three types of actors used in the application. Section 2.2 explains how the features of the `PAM` framework are leveraged to implement the traffic simulation program. In Section 2.3, we present and analyze the results of running the simulation on road networks with various configurations and sizes. Lastly, we briefly discuss the optimizations applied to the original serial code in Section 2.4.

2.1 Actors

Our traffic simulation program is composed of three types of actors: `Junction`, `Factory`, and `Summary`. The `Junction` actor represents a junction within the road network and its associated outgoing roads. The `Factory` actor periodically generates new vehicles to maintain the maximum number of active vehicles during the simulation. The `Summary` actor periodically outputs a summary of the simulation's progress to *stdio* and, at the end of the simulation, writes a file with detailed statistics per junction and road. Figure 4 illustrates the behavior of the actors and their interaction during the simulation.

2.1.1 Junction Actor

The `Junction` actor models a junction in the road network along with its connected outgoing roads. It maintains a list of *Vehicle* objects, either waiting at the junction's traffic light (if one is present) or traveling on one of its outgoing roads. When a vehicle reaches the end of a road, its data is forwarded to the next `Junction` actor. It is important to note that if the junction is equipped with a traffic light, it periodically switches the active road at the light.



Factory Actor
Junction Actor
Summary Actor
Receive number of vehicles that exited simulation
send vehicles
Receive Vehicle objects
send summary
Receive summary on simulation's progress
send integer
Generate new vehicles
For each received vehicle:
- Check if current junction is its destination
- Otherwise, plan its route and assign it an outgoing road.
Print summary of simulation's progress
send integer
Switch active road at traffic lights (if present).
send summary
For each vehicle waiting at current junction:
- Check its current fuel level
- Check if it can exit the junction and merge onto the designated outgoing road.
send integer
send summary
For each vehicle on a road:
- Check its current fuel level.
- Update the remaining distance it must travel.
send vehicles
Next Junction Actor

Figure 4: Behavior of actors and interaction between them.

Figure 4 provides a high-level overview of the steps taken by a `Junction` actor during the simulation. It receives `Vehicle` objects from the `Factory` or other `Junction` actors and adds them to the list of vehicles waiting at the junction. If any of these vehicles have the current junction as their destination, they are removed from the simulation, and both the `Factory` and `Summary` actors are notified. Specifically, the number of passengers who have reached their destination is reported to the `Summary` actor. For the remaining vehicles, route planning assigns each one to one of the junction's outgoing roads.

The actor then updates the state of the vehicles, which fall into two categories: those waiting at the current junction and those traveling on a road.

- For vehicles waiting at the junction, the actor determines if they can exit the junction and enter their assigned outgoing road. If the junction has traffic lights, vehicles must wait until their designated road becomes the active road. Otherwise, they may leave the junction and enter their designated road, albeit with a potential risk of crashing. In the event of a crash, both the `Factory` and `Summary` actors are notified.
- For vehicles traveling on a road, the actor updates the remaining distance they need to travel. Once a vehicle reaches the end of the road, it is forwarded to the next `Junction` actor.

Additionally, any vehicle at the junction or on the road that runs out of fuel is removed from the simulation, with notifications sent to both the `Factory` and `Summary` actors. Finally, when the `Junction` actor receives the termination message from the `Summary` actor, it sends a detailed report of the junction's progress, including details of each of its outgoing roads.

2.1.2 Factory Actor

The `Factory` actor periodically generates new vehicles to ensure the maximum number of active vehicles is maintained throughout the simulation. It sends `Vehicle` objects to `Junction` actors and receives the number of vehicles that have exited the simulation from them. Finally, it stop when it receives the termination message from the `Summary` actor.

2.1.3 Summary Actor

The `Summary` actor periodically prints a summary of the simulation's progress to `stdio` and, upon completion of the simulation, generates a file containing detailed statistics for each junction and road. It receives progress data from both the `Factory` and `Junction` actors related to the simulation. When the maximum simulation time is reached, it sends a termination message to all other actors and then waits to receive a final summary from all `Junction` actors.

2.2 Using the Actor Model Framework

The following provides a brief overview of how the features of the PAM framework are utilized to develop the distributed actor version of the traffic simulation code.

- The `Junction` actors are added as **grouped actors**. For instance, we successfully executed the program with 100 `Junction` actors per MPI process on a road configuration containing a total of 100,000 junctions.
- Due to their time-sensitive operations (e.g., the `Factory` actor periodically generates and sends new vehicles, and the `Summary` actor periodically prints the simulation's progress), the `Factory` and `Summary` actors are added as **isolated actors**.
- The synchronization barrier in the actors' initialization phase ensures that all actors simultaneously begin the computation phase of the traffic simulation. This is especially crucial for road networks containing 100,000 junctions.
- We created user-defined MPI datatypes to be used in the data payload of messages between actors. These datatypes are shown in Figure 5.

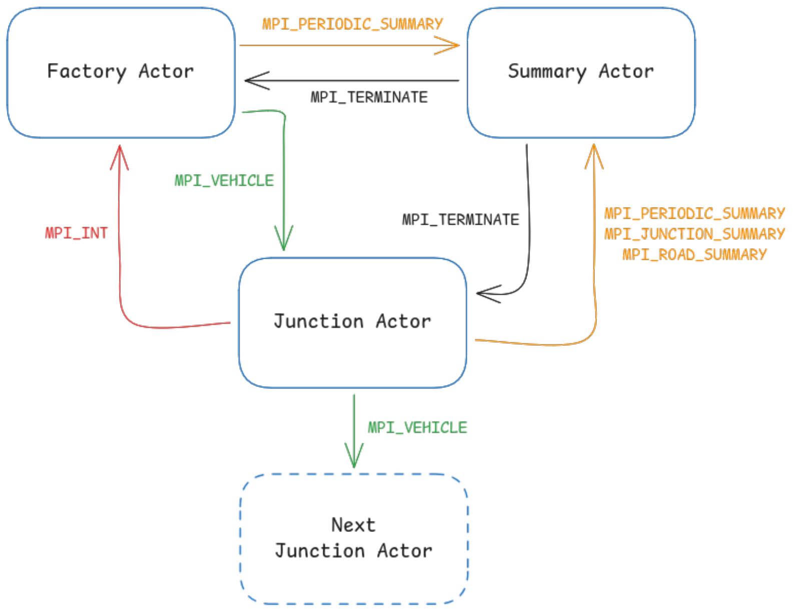


Figure 5: User-defined MPI datatypes used in the messages between the actors.

2.3 Performance and Scaling Experiments

2.3.1 Supported Road Network Configurations

The traffic simulation program can handle problem sizes listed in Table 1, with road network configurations ranging from as few as 1,000 junctions to as many as 100,000 junctions.

Table 1: Problem sizes

Road Network	Junctions	Roads	Initial Vehicles	Maximum Active Vehicles
Small	1000	2516	100	1000
Medium	20000	55472	100	1000
Large	50000	142166	200	2000
Largest	100000	277590	200	2000

The serial code included preprocessor constants that controlled how the simulation components were updated. To achieve meaningful results after running for 30 simulated minutes (especially for bigger problem sizes), the following constants and road network properties were adjusted:

- The minimum and maximum fuel levels for each vehicle type were increased by a factor of 5. This adjustment allowed vehicles to travel farther, increasing the likelihood of reaching their destinations, and reduced the number of vehicles running out of fuel.
- The length of each road was reduced by a factor of 10, with a minimum resulting road length of 50. This resulted in vehicles reaching their destination in fewer simulated minutes than originally required.

With these adjustments and allocating 100 `Junction` actors per MPI process, we ran the traffic simulation on ARCHER2 for 30 simulated minutes using various road network configurations, and the results are presented in Table 2. As expected, larger problem sizes demand more execution time for simulation, as updating the state of the actors in a large road network becomes more computationally intensive. For instance, running Dijkstra's algorithm on a large road network is more demanding than on a smaller network. Additionally, as the problem size increases, the number of delivered passengers decreases, while the number of stranded passengers rises.

Table 2: Running the simulation for 30 simulated minutes across all road network configurations.

Road Network	Total Vehicles Modelled	Passengers Delivered	Passengers Stranded	Vehicles Crashed	Vehicles Exhausted Fuel	Execution Time (s)
Small	2336	14832	544	0	314	59.19
Medium	1423	1564	546	2	312	61.85
Large	2473	611	793	0	469	66.45
Largest	2930	421	797	0	435	76.20

2.3.2 Strong Scaling Experiment

Next, we evaluate how well the application utilize additional computing resources for the same workload by performing a strong scaling experiment. Applying the same adjustments described in Setion 2.3.1, we perform 30 simulated-minute runs on the largest configuration for various number of `Junction` actors per MPI process, and present the results in Table 3. A smaller number of `Junction` actors per MPI process corresponds to a greater number of cores being utilized. For instance, with 25 `Junction` actors per MPI process, 4000 MPI processes manage grouped actors, while two additional MPI processes handle the isolated `Factory` and `Summary` actors.

In order to understand the results in Table 3, we break down the execution of the traffic simulation in to three phases:

- **Initialization phase**: During this phase, the `PAM` framework initializes all `Junction`, `Factory` and `Summary` actors. Recall that a synchronization barrier ensures the execution does not proceed to the compute phase until all actors have successfully completed their initialization.

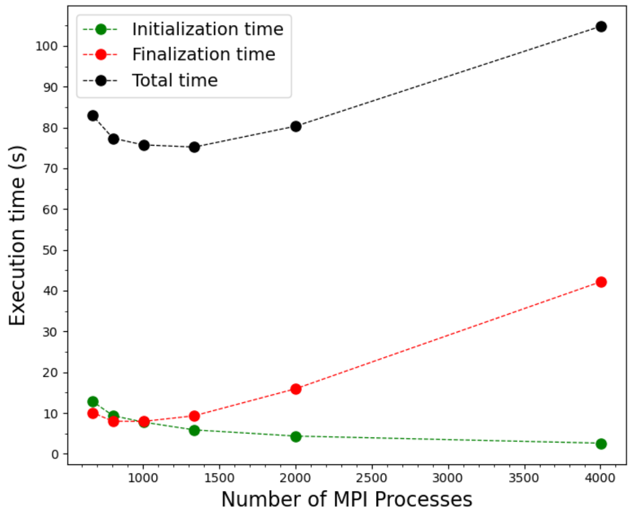
- **Compute phase**: The actors carry out the 30 simulated-minute run. The application is configured to run 2 real-time seconds for every simulated minute, ensuring that this phase consistently takes approximately 1 minute of wall clock time.
- **Finalization phase**: The `Summary` actors sends the termination message to all other actors. Then each `Junction` actor sends a detailed summary, which the `Summary` actor receives and writes on a file.

The best total execution time in Table 3 is achieved with 75 `Junction` actors per MPI process, which corresponds to using 1336 MPI processes. Using fewer MPI processes increases initialization time, as each MPI process must initialize a larger number of `Junction` actors, leading to longer total execution times. On the other hand, using more MPI processes increases communication overhead during the finalization phase, as a greater number of processes across more nodes are involved when each `Junction` actor sends a detailed summary to the `Summary` actor. To summarize, initialization overhead decreases with increasing number of MPI processes; however, communication overhead during the finalization phase rises with the number of MPI processes. These observations are plotted in Figure 6.

Table 3: Strong scaling experiment on the largest configuration for 30 simulated minutes per run.

Number of Junction Actors per MPI process	Number of MPI processes	Initialize (s)	Compute (s)	Finalize (s)	Total (s)	Total Number of Vehicles Modelled
150	667+2	12.83	60	10.07	82.91	2946
125	800+2	9.36	60	7.99	77.36	2952
100	1000+2	7.77	60	7.95	75.72	2930
75	1334+2	5.86	60	9.33	75.19	2910
50	2000+2	4.35	60	15.96	80.32	2857
25	4000+2	2.62	60	42.16	104.78	2478

Figure 6: Plot for the execution times in Table 3.



Thus, the strong scaling experiment reveals that our application does not scale efficiently with a higher number of MPI processes, due to the increase in communication overhead as the number of processes grows. One potential solution to address this issue is to modify when the detailed summaries are sent to the Summary actor. Instead of having each Junction actor send its summary at the end of the computation phase, the Junction actors could send them during the computation phase. Currently, each Junction actor already sends periodic updates containing vehicle statistics to the Summary actor, allowing it to print the simulation progress to stdio. In addition, each Junction actor could also send junction and road statistics to the Summary actor as they are updated during the computation phase, rather than waiting until the finalization phase. With this improvement, the finalization phase would then only involve writing the detailed summary to a file.

2.4 Optimizations

The serial code underwent significant refactoring, with two key optimizations worth highlighting:

1. **Improve Implementation of Dijkstra's Algorithm**
   The serial code's implementation of Dijkstra's algorithm iterates through all nodes to find the next node with the smallest tentative distance, leading a run time of $\mathcal{O}(N^2)$, where N is the number of nodes. In contrast, using a priority queue for this task reduces the runtime to $\mathcal{O}((N+M)\log N)$, where M is the number of edges. This improvement is essential for large road configurations, as the `Factory` actor executes this algorithm whenever new vehicles are generated, and the `Junction` actors run it whenever a vehicle requires road assignment, which occurs each time a vehicle enters a junction.

2. **Improve Selection of Potential Source and Destination Junctions**
   The serial code assigns source and destination junctions to a vehicle by randomly selecting two junctions and verifying their connectivity using Dijkstra's algorithm. If a valid path cannot be established, it generates a new potential destination junction while keeping the source junction unchanged and reruns Dijkstra's algorithm to verify connectivity. This process continues until valid source and destination junctions are successfully selected.

   The process described above is a slow process, especially for large road configurations. Table 4 shows the number of connected components for each road network configuration. For instance, the graph of the medium configuration consists of 224 connected components, 61 of which has a size of one node. To select a valid pair of source and destination junctions, the process must randomly select two nodes within the same connected component, which itself does not guarantee that there is path between them, hence it must be verified by Dijkstra's algorithm. Moreover, if the source junction belongs to a single-node connected component, the process will never terminate, as it only regenerates the destination junction while leaving the source junction unchanged.

   We improve the process above as follows. First, we randomly select a source junction. If this junction belongs to a single-node connected component, then we regenerate the source junction. Second, we randomly select a destination junction from the pool of nodes belonging to the same connected component as the source junction. Third, we verify that there is a valid path between the source and destination junctions by running Dijkstra's algorithm. If no valid path is found, the process is repeated. We leverage a pre-computed disjoint set of the graph making the first and second steps run in $\mathcal{O}(1)$. For more details, refer to the comments in the `assign_source_and_destination` and `generate_random_destination` methods in `Factory.cpp`.

Table 4: Number of connected components of supported road network configurations.

Road Network	Nodes	Total Connected Components	Single-Node Connected Components
Small	1000	50	1
Medium	20000	224	61
Large	50000	30	3
Largest	100000	29	1