A Parallelization Framework for Multi-Agent Reinforcement Learning Environments

Sai Meher Karthik Duddu*
sduddu@andrew.cmu.edu

Venkat Srinivasan*
venkatks@andrew.cmu.edu

1 Summary

Multi-agent learning scenarios for reinforcement learning (RL) are becoming increasingly common[1] due to their applicability to domains such as collective intelligence and population studies[2]. Popular frameworks for dealing with RL environments have been restricted to single-agent frameworks[3], or have limited support for multi-agent environments[3], or require setting up (and understanding) non-trivial parallel architectures[4]. In this project, we design a flexible and robust framework for expressing grid-based environments, and provide transparent parallelization of games built using the framework through OpenMP and OpenMPI, thereby allowing the user to focus on defining the environment and rules of the game. The framework is designed considering data dependencies present in Markovian games (games in which an agent acts based on current state information), and will allow the simulation to run in parallel. We analyze the framework along three dimensions: language of implementation, parallelization methods, and complexity of the game. For the current discussion, we consider two relevant metrics: efficiency (as measured by speedup over sequential execution), and ease-of-use for the programmer. In order to perform this analyses, we implement two grid-based simulation environments with differing complexity, and conduct a performance analysis using different types of parallelism technologies and under different loads. Finally, we discuss scenarios where our framework is beneficial, and the hyperparameters that allow it to offer the best speedup/convenience tradeoff.

2 Background

2.1 Expository Background

Developing reinforcement learning (RL) algorithms requires access to a simulator. For example, the OpenAI Gym offers a plethora of simulation environments for RL algorithm engineers, such as LunarLander or Atari or GridWorld-based environments. Reinforcement learning models trained to interact with the simulator decide on an action for an agent, and request the simulator to 'step' using that action. In this step function, the new state of the simulator is computed, on the basis of changes introduced by the agents within the environment. While the simpler games can easily be implemented sequentially and can offer very good performance by virtue of the lack of computational complexity, larger and more involved simulation environments with more agents might require a significant amount of time per simulation

step. This is a problem since a lot of RL algorithms require a significant amount of interactions with the simulator to properly learn, and the amount of time spent computing information for the simulator itself takes away from compute dedicated to the RL policy network. Therefore, we want to spend as little time as possible within the simulator step function, necessitating parallelism.

2.2 Technical Background

There are 2 axes of parallelism which are exposed by multi-agent environments: a) parallelization of agent simulation and interactions, and b) environment state updates on the basis of agent actions. The correctness of applying parallelization is dependent on the nature of the environment and agent interactions - for the purposes of this project, we shall restrict ourselves to games in which agents act simultaneously based on environment state information (Markovian games[5]). For such games, we can think of each agent as an "instruction stream" or a "thread", as each agent will be able to "undertake" independent actions that directly impact the shared space.

We design a framework that allows reinforcement learning (RL) engineers to easily parallelize their gridworld-based simulators without having to write any parallel-programming code. We have written OpenMP and OpenMPI code within our abstraction layers that reside in the core of our system that provides the users powerful API abstractions that can easily allow them to parallelize across their agents with just a few lines of code.

2.2.1 Key Structures

While there are several abstractions we offer the RL engineers, the most salient ones are the **AbstractAgent** abstraction, the **AbstractEnv** abstraction, and the **AbstractMap** abstraction. Together, these maintain the state required for parallelizing the underlying implementation of the game. This design allows us to untangle dependencies relatively easily, as data dependencies are broken down into parallelizable components, while also allowing us to offer a simple interface.

2.2.2 Key Operations

The primary operation that we are focused on parallelizing is *Step()* within the AbstractEnv. This operation is the heart of the simulation environment, as it is designed to take the agents' inputs and actuate them in the simulated map, modifying the map state in a germane way to represent the impact of those actions. This is the area where the majority of the computation time is spent, and therefore, this is where we focused our parallelism efforts on. In the sequential version, the algorithm will iterate over the agents and map and collate their actions, and once again will iterate over the agents and actuate their actions by modifying the world. In our parallel versions, we divide the list of agents over the execution contexts and then recombine them at the end. In addition, there is also the *doAction* and *doActionCollate* operations present within the **AbstractAgent** layer. These operations help convert the agent's actions into an ac-

tionable map update. The former helps us convert the individual agent actions into actions that also affect other agents. These functions are user-defined, and can be specified arbitrarily, and the algorithm will use the user specifications to disentangle agent-state-based dependencies so that operations per agent can be parallelized.

For instance, the user can specify that *Agent* α issuing a *KILL Agent* β command will be converted into a *DIE* command for Agent β. This allows us to disentangle the dependency, and issue the map updates for Agents α and β in parallel, before actually actuating those updates serially. The latter converts an agent's *move* operation into a map *del agent old_pos* and *put agent new_pos* actions. It is also worthwhile to note that all of these specifications can be arbitrary, and therefore can be used to define any kind of actions and map updates. The system will only use these specifications for disentangling the dependencies for parallel/sequential processing.

2.3 Algorithm IO

The fundamental parallelism we implemented exists within the AbstractEnv framework. Everything else is used as user-provided specifications that allow them to arbitrarily define the environment and the way to disentangle dependencies. Therefore, the following can be considered inputs to our framework:

1. **Agent Definition:** It is necessary to specify what the agents are capable of doing in terms of their concrete action space, as well as how the action of one agent will affect another agent. This will allow the system to determine the dependencies between agents and disentangle them. This can be specified by inheriting from the *AbstractAgent* class and implementing the pure virtual functions.

2. **Agent Registration:** Once agents are created, it is necessary to register them with the framework so that the framework knows what agents exist and can be parallelized.

3. **World Definition:** While it is true that the framework only supports a gridworld-based map, it is required for the user to define the space of the world and what positions are valid or invalid, and how to actuate specific updates within the map state. This takes the form of inherting from the *AbstractMap*, *AbstractPosition*, and *AbstractDistance* framework and implementing the pure virtual functions. This allows the algorithm to understand what the space looks like, what distances are defined as, and what locations are invalid.

4. **Rewards Definition:** This is irrelevant for the parallelization aspect of our program, but since we are trying to address a Reinforcement Learning problem, the environment needs to know how to return rewards once the *Step()* operation is called. This specification is used to address that.

In total, all of these specifications can be defined in exactly 4 functions: *doAction* in Agent, *doAction-Collate* in Agent, *doTaggedMapUpdate* in Map, and *ComputeReward* in Env. This dataflow is expressed

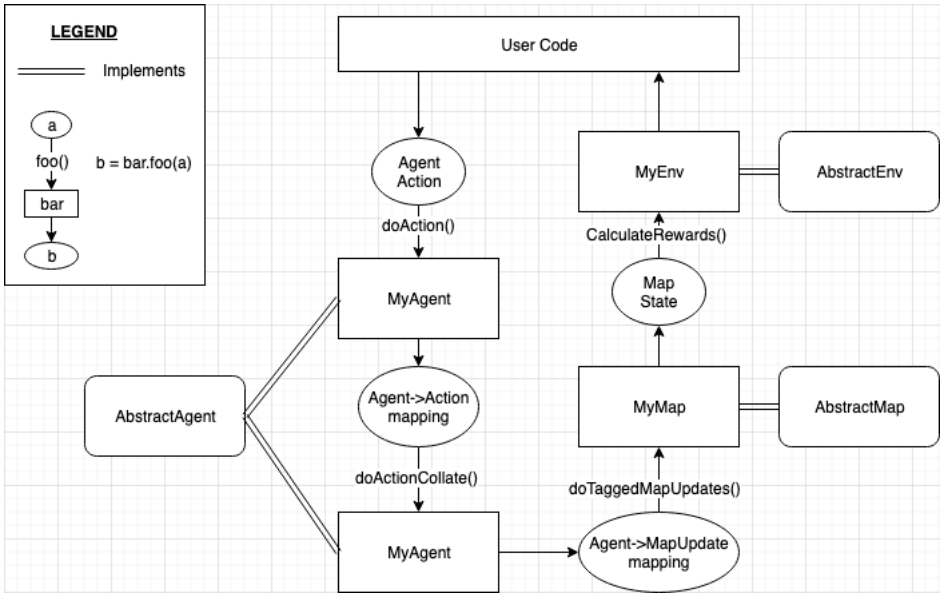

Figure 1: The overall dataflow of our framework implementation.

in Figure 1. The output of this system is a correctly parallelized framework that experiences significant speed-up over the default sequential implementation current popular frameworks utilize.

2.4 Workload Focus

As briefly mentioned in Section 2.2.2, the primary area of workload is within the *Step()* function, which was illustrated in Figure 1. Particularly, it is worthwhile to note that the parts of the workload that operate across agents tend to have the most workload, as the amount of work needed scales with the number of agents spawned. Therefore, that is the focus of our parallelism efforts within our abstraction. The way we parallelized our model can be observed in Figure 2. Figure 2 will be discussed in far more detail in Section 3.

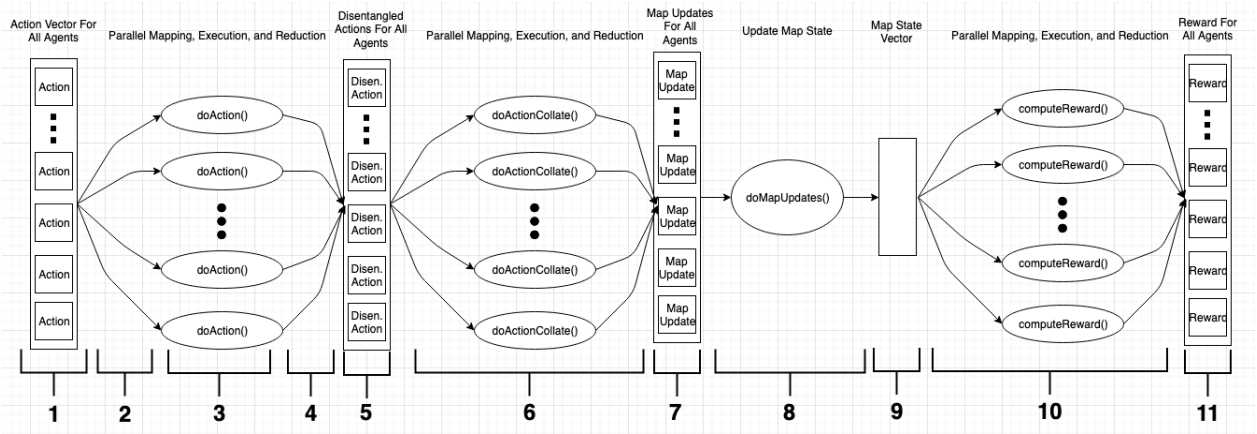

Figure 2: Parallelism Methodology.

We note that the user-supplied components only touch upon aspects of the game which would require

implementation even in a sequential environment. Within the space of a grid-oriented Markovian game, the framework is thus able to provide parallelization "for free" to the developer.

2.5 Workload Breakdown

The dependencies of the problem lie largely in the interactions between the agents, as does the inherent workload within the program. There are two axis of scaling: the number of agents, and the complexity of the agent actions. Were the number of agents to increase, the amount of total computational burden will also increase. Likewise, were the complexity of the agent actions to also increase (in that, there are far more dependencies between the agent's actions on the other agents; for instance, if an agent has to scan the grid to seek out another agent before attacking), then the simulation will also experience a slowdown. The workload here is largely data-parallel, and is rather amenable to SIMD execution. However, it is worthwhile to note that the operations performed in the data-parallel manner do tend to diverge a lot, and thus might not make full use of the infrastructure capabilities were vector instructions to be used for execution.

3 Approach

3.1 Technologies

We were aiming for a wider applicability of our contributions, so we opted to implement our framework in both C++ and Python. In C++, we implemented the generalized parallelism using OpenMP, and MPI + OpenMP. The user can pass in a command-line option when running the binary to choose an underlying parallelism framework. We primarily targeted machines with a large number of core counts for our OpenMP implementation, and utilized clusters for our MPI+OpenMP implementation.

3.2 Task Mapping

3.2.1 Sequential

The sequential implementation was mapped in a very straightforward manner. Once we were given the input action vectors, we simply assigned the entire workload to a single processor, which then iterated over all the agents to perform the action for them via the *doAction* interface to generate the disentangled actions, and then again iterated over all the agents to convert the disentangled actions to actionable map updates, and actuated the map updates serially, before finally again iterating over the agents to generate the reward values for each agent.

It is worthwhile to note that the reason for this staged sequential approach is because we utilized the same parallel framework for the sequential implementation, as well. Were this to be only implemented sequentially, there would be no need to generate the disentangled actions, or any of the intermediate staged data since we would simply iterate over each agent and actuate their actions onto the map. However, we

believe that the additional overhead due to the parallelism framework does not add too much overhead, and these sequential numbers do represent the inherent amount of time any sequential algorithm would take for this problem.

3.2.2 OpenMP

A high level view of our implementation can be seen in Figure 2. Please refer to that figure for the continuing discussion. In our OpenMP implementation, we first receive the inputs per agent, as shown in *Stage 1*. We then figure out the avaliable execution contexts on the host machine, and seperate the agent actions into equivalent groups and map each to the appropriate core, as shown in *Section 2*. The agents execute their action in parallel, as shown in *Stage 3*. To allow for a minimization of contention, we do not utilize locks to recombine the actions, and instead opt to utilize a reduction stage afterwards, as shown in *Stage 4*. This generates a list of disentangled actions that can now be run in parallel, as the dependencies between each has been removed during the aforementioned parallel and reduction stages. We now have a vector of disentangled actions, as shown in *Stage 5*. These actions are then remapped onto the execution contexts and are then used to generate MapUpdate rules, as shown in *Stage 6*. This puts us at *Stage 7* with a vector of MapUpdates. These MapUpdates are then sequentially actuated by the algorithm in *Stage 8*, since it is hard to estimate how exactly it can be disentangled in a generalized parallel way, but this can be considered a future work. This leaves us with a map stage at *Stage 9*, from which we can now map to execution contexts to retrieve per-agent rewards, as shown in *Stage 10*. Finally, at the end at *Stage 11*, we are left with a per-agent reward vector, which is then returned to the user.

3.2.3 MPI + OpenMP

The general implementation for MPI is very similar to OpenMP. The primary difference is that have an initial distribution where we divide the agents into groups and map each group to a cluster. The algorithm utilizes OpenMP to properly parallelize within each cluster across the agents in the group.

3.3 Algorithm Changes and Choice of Baseline

The entire framework was designed from the ground-up for the parallel task. Therefore, it is the case that we are using the staged parallel framework for our sequential testing.

3.4 Iterations

The algorithm utilized many iterations to achieve our optimal state. These iterative changes will be detailed in this section.

3.4.1 Python to CPP Switch

We initially wanted to use the OpenAI GYM Framework for our implementation and only implement the parallelization part of our project. However, the issue with this is that GYM is built on Python only,

and Python's Global Interpreter Lock essentially makes multithreading worthless. While there can be tangible benefits observed were the work to be IO-bound (something like saxpy), for our situation, where most of the simulation environments are mostly CPU-bound, the speed-up due to the GIL is virtually nil. We then explored possibly using the multiprocessing capacity of Python by parallelizing the initial stage of the pipeline in Python, which utilizes parallel processes with shared memory addresses. However, due the fact that Python multiprocessing pickles to the filesystem and unpickes the data that is to be sent to the child processes, we failed to observe any significant speedup, and even observed a slow-down as we scaled the number of processeses generated. This prompted us to put the breaks on our Python development and switch to CPP, instead. However, as OpenAI was not present in CPP, we had to write the design and implement the entire framework from scratch, before writing the parallelization code in CPP.

3.4.2 CPP Pure MPI

One of the approaches we also tried was simply implementing the entire framework using MPI. However, we noticed that the amount of communication that this approach generated was enormous, and we observed negligible speedup, as the entire run was dominated by the communication more than the actual underlying computation. This took us back to the drawing board, where we decided to take a hybrid approach of combining our existing OpenMP implementation with MPI, whereby we decrease the number of groups we spawn and map to nodes, but instead just parallelize within each node using OpenMP. This decreased the amount of communication we faced between nodes, and allowed us to actually observe a useful speedup.

3.5 Initial Code

While we were initially hoping to utilize OpenAI's GYM framework, we dropped this once we realized that we cannot effectively parallelize in Python. Instead, we wrote the entire framework from scratch and did not use any preexisting code for our actual CPP deliverables. We only utilized the OpenMP and MPI libraries for parallelism.

4 Results

4.1 Definition of Metrics

We measure our speedups in terms of wall-time. We have defined a `Recorder` class within our CPP and Python modules that allow us to calculate a stage-wise execution time for each implementation. The difference in this metric is what we are focused on discussing within this report.

4.2 Experimental Setup

4.2.1 Language-wise Experimentation

Our experiments spanned two languages, Python and C++. For Python, we experimented with parallelizing the first stage of the pipeline only (*Stage 2, 3, 4* in Figure 2). This is because the lack of any significant improvements due to the Global Interpreter Lock and the requirements to pickle/unpickle when assigning work in multiprocessing. For CPP, we parallelized the entire pipeline, as shown in Figure 2. This will be detailed when presenting the graphs.

4.2.2 Technology-wise Experimentation

We experiment with OMP and MPI. For OMP, we vary across the number of threads, and for MPI, we fix the number of threads (the optimal that we determined from the OMP experiments) and vary the number of nodes used. All of these experiments are run in C++.

4.2.3 Environment-wise Experimentation

For some of the experiments, we also scale across the number of agents present in the environment. We expect to see a bigger contribution from parallelism as we increase the number of agents present in the environment. Likewise, we also scale across the complexity of the agent interactions. We have implemented two distinct environments, the `Exploration` and `Battle` scenarios, to account for this. The `Exploration` environment has very little dependencies between the agents, while the `Battle` scenario has heavy dependencies. We expect to see greater improvements from parallelism as the complexity of the agents also increase.

In the exploration game, each agent explores the environment by moving around and receives a reward in accordance with its distance from the center of the grid. In the battle game, agents are placed in a "free-for-all" environment with a certain amount of predetermined health, and can move around, scan the map, and impact other agents by "pushing" them (which causes a decrease in health for the latter agent and an increase in health of the former).

4.3 Results Graphs

In the following section, unless otherwise stated, all results have been discussed for the "exploration" game, in which each agent explores the environment by moving around, and receives reward in accordance to its distance from the center of the grid.

4.3.1 Overall Performance Profiles

In Figure 3, we compare the amount of time taken for the *Step()* function to execute as we scale the number of threads present; therefore, a lower number is better. Our initial hypothesis specified in Section 4.2.3 is correct, as we see larger improvements from parallelism as we scale up the number of agents

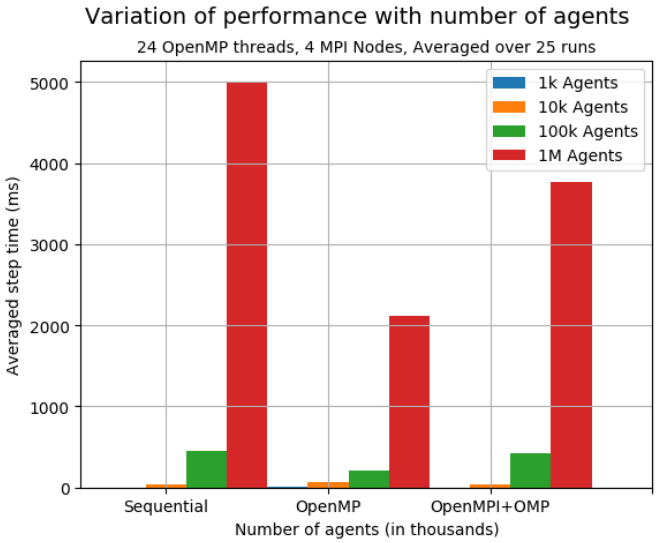

Figure 3: Raw Comparision of Parallelism Methodologies.

present in the environment. The sequential baseline is what was specified in Section 3.2.1. The OpenMP is what was specified in Section 3.2.2 and the OpenMPI+OMP was specified in 3.2.3. We see an immediate speedup in implementing OMP, but we see a slowdown between OMP and MPI due to the overhead of communication for such a large amount of agents.

4.3.2 OMP C++ Performance

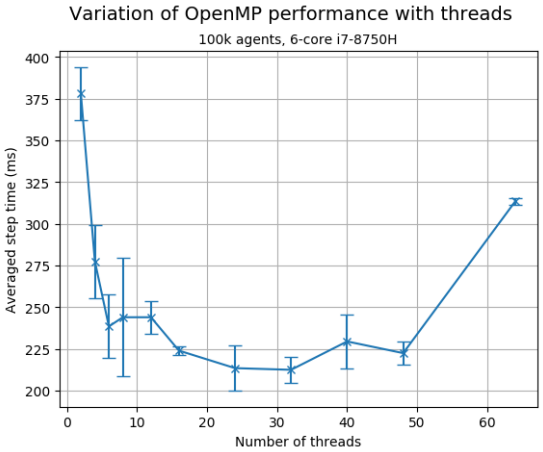

Figure 4: Performance of OMP as we scale the number of threads, after fixing the number of agents.

A view into the performance of OMP can be seen in Figure 4. In Figure 4, we see the impact of scaling the number of threads dedicated to parallelizing the agents. We can observe how we see massive improvements initially as we scale up the number of threads up to the number of cores. This was run in a machine with *8* cores, and thus we see rapid improvements as we scale up to 8 threads. Once we hit there, we still see some improvements as we scale beyond that, due to the hardware being able to hide the

memory latency by swapping out blocked threads. But as we increase it, the hardware starts thrashing and the overhead of scheduling starts to overwhelm any improvements, and we start seeing runtimes increase. This allowed us to determine the correct number of threads to spawn for our OMP runs.

4.3.3 MPI C++ Performance

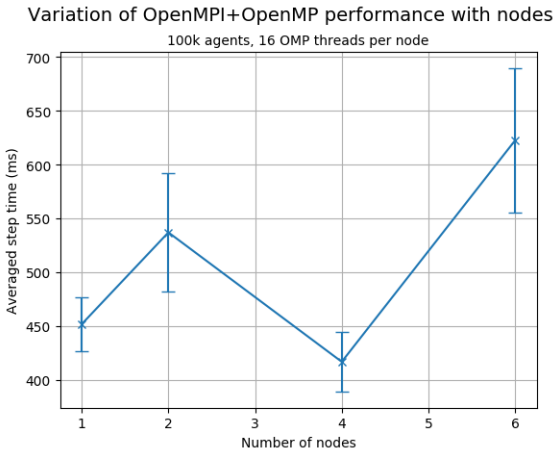

Figure 5: Performance of MPI as we scale the number of nodes, after fixing the number of agents and per-node thread count.

After determining the best number of threads per node, we wanted to calculate the optimal number of groups to divide the workload between, to minimize the amount of communication to computation ratio. The chart in Figure 5 details the speedup we experienced. We notice that a good "middle-of-the-road" approach works the best, as having too few nodes causes the communication to dominate, as does having too many nodes. Having a good balance between the number of nodes present and the overall work assigned to each node (around 4 nodes per 100k agents) seems to work the best.

4.3.4 Timing Per Stage

It is also worthwhile to look at the amount of time each technology takes at every stage of the step function. In Figure 6, we observe that the biggest contributions from OMP comes in the first stage, when the agents are interacting with the environment. We also see that the reward computation stage also massively benefits from parallelization stage, as there are few dependencies between the agents at that stage. However, we have not been able to parallelize the map update stage due to the intricacies of maintaining generalizability of the framework for all gridworld-based environments. However, similar to the other stages, we believe that given more time, it is possible to introduce parallelization in a generalizable way for this stage as well. We also see that the parts with significant dependencies cause MPI to underperform significantly, as in the doAction stage (where there is a lot of dependencies that need to be untangled). However, it is able to make up for this in the reward computation stage (that's more straightforward in

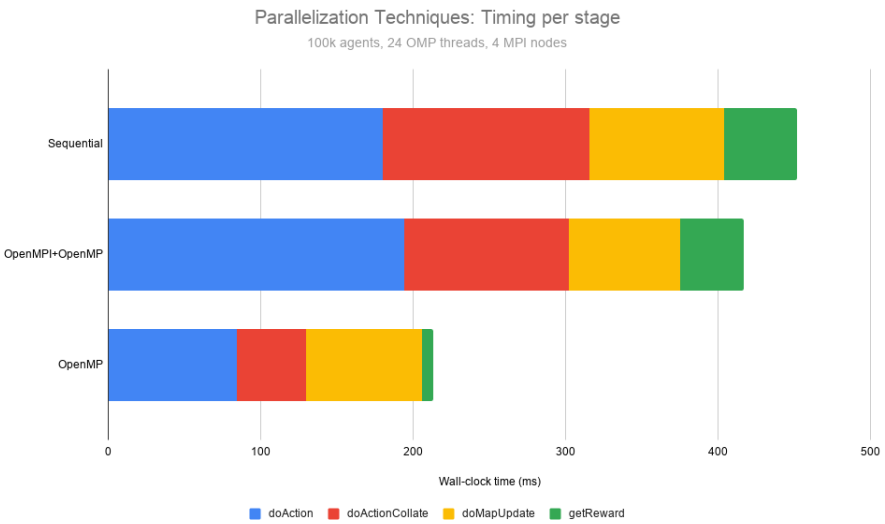

Figure 6: The time taken by each technology at each stage.

terms of dependencies), thus informing us that MPI is limited by the communication overhead.

4.4 Varying By Agents

The crux of our contribution is effective and quick computation for environments with a lot of agents. Therefore, this is one of the venues that we wanted to experiment with. The speedup observed as we scale the agents can be seen in Figure 7. We observe how our framework performs extremely well

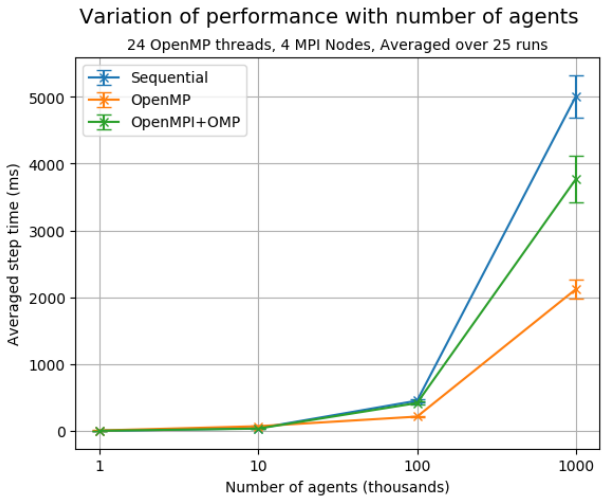

Figure 7: The time taken by each technology as we scale the number of agents present in the environment.

compared to the sequential baseline, when using OMP. This curve is what we were hoping to achieve, as the amount of time we save increases rapidly as the number of agents present in the environment is scaled up. We observe that as we scale the number of agents to 1M+, the slowdown experienced by MPI due to the overhead of communication also disappears, and we start seeing massive improvements from

parallelization. We believe that the disentangling of dependencies in the action stage allows us to achieve good speed up in that stage, that allows us to get these numbers.

4.5 Varying By Agent Complexity

As stated in Section 4.2.3, we also believe that we will see massive improvements from parallelization for games that are more complex and have more interactions between the agents. Therefore, we implemented a new game that has more salient and more complex agent interactions. The stage-wise breakdown of this game can be seen in Figure 8. In this complex battleground game, we can see how the sequential

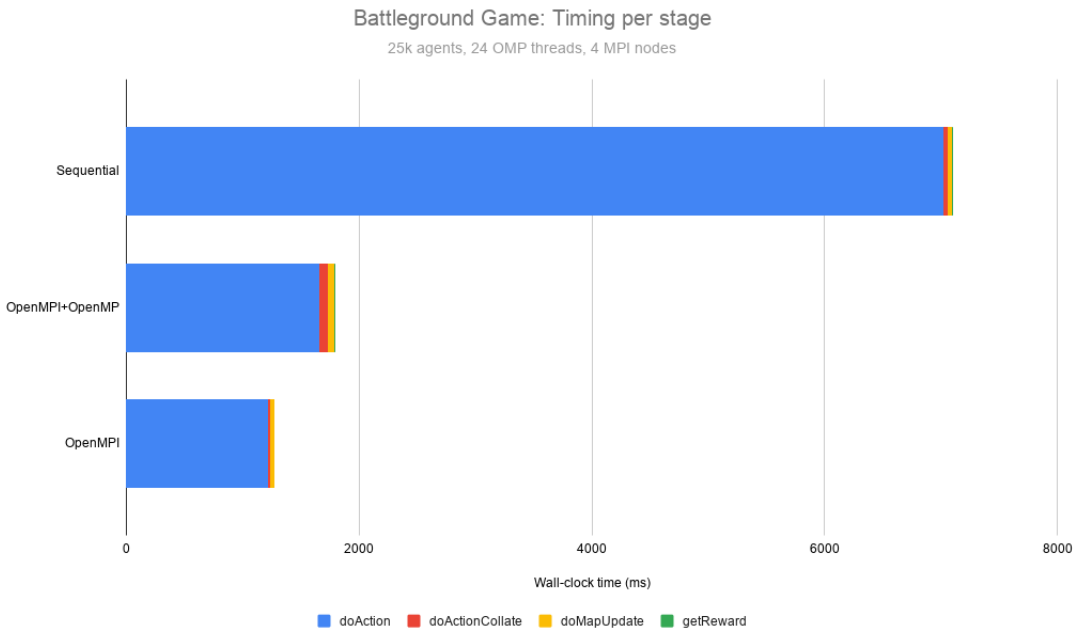

Figure 8: Complex Environment Stage-Wise Time.

computation is dominated by the stage that has a lot of interactions between the agents. Untangling the dependencies and parallelizing the environment allows us to experience *massive* improvements, thus giving credence to our hypothesis specified before. It is true that as we increase the complexity of the environment and the number of agents, our framework starts to shine and yield real dividends for RL algorithm engineers. It is also worthwhile to reiterate here that our framework is flexible, and we expect to see similar speedups for any arbitrary game with a similar similar level of agent interaction complexity. It is also likely the case that for more complex and elaborate games, the speedup offered by our framework will only increase. As stated in Section 4.2.3, we also believe that we will see massive improvements from parallelization for games that are more complex and have more interactions between the agents. Therefore, we implemented a new game that has more salient and more complex agent interactions. The stage-wise breakdown of this game can be seen in Figure 8.

4.5.1 Potential bottlenecks affecting speedup

One aspect of our slowdown is in the fact that we are IO-Bound in some places. Even when running on machines with 8 cores and 2 hardware threads per core, oversubscribing the number of threads (as shown in Figure 4) still shows a speedup. This indicates that some of our computation is actually IO bound, and isn't strictly CPU-bound. In addition, the fact that we haven't parallelized the map-update stage inherently limits the extent of speedup which can be achieved (by Amdahl's Law) as shown in Figure 6 (we see a better ratio for almost every stage but the overall ratio stays low due to the map update stage).

4.5.2 Was The Machine Choice Sound?

We believe that we could have shown even better results had we focused our efforts on also porting the parallel algorithm to the GPU. We can observe that our framework performs very well in shared memory paradigms with a lot of compute contexts, as we can show strong performance numbers for OMP. We believe that porting this to the GPU would have allowed us to deliver even more impressive performance, even after accounting for the possibility of divergence between agents assigned to the same warp. In this paradigm, we would have ideally assigned each cuda thread to a single agent and broken the overall agents into groups of 1024, similar to what was done when assigning agents to MPI clusters. However, we observed that using CUDA as a platform would require the game designer/developer to write custom kernels for every environment, which we believe would limit the generalizability of our framework. However, with some custom interfaces, this might be a possibility in the future.

5 Conclusion

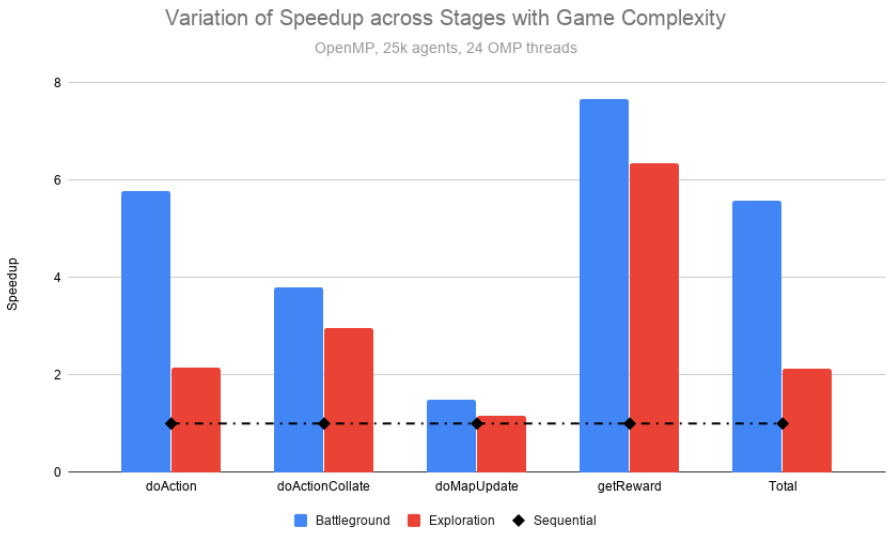

Figure 9: Conclusion: An Overall View Of Performance Metrics.

Figure 9 provides a conclusive view of the overall metrics we discussed earlier. We can observe that

as the complexity increases, especially in the agent action phase, we start observing better speedups. For instance, in the complex dependency laden battleground game, we see nearly 6x speedup, as opposed to the 2x in the simple exploratory game. Overall, we do consistently well in parallelizing the rewards, but it is worthwhile to note that this could be an artifact of our implementation as we have very little dependencies in our reward computation structure. Introducing more complex dependencies in this stage could make it start acting more alike the earlier action stages that do have complex dependencies. Map updates are currently done in serial due to a lack of time, but they can be easily serialized by introducing a few more abstractions that can be used to disentangle the dependencies and parallelize the map updates. Overall, we see close to a 2x speedup in both scenarios, and across all parameters, and we see OMP perform better overall. Increasing the number of agents with MPI also shows a slowdown, meaning that OMP is likely the best technology for this, as MPI with a lot of agents and for more complex might lead to communication overhead dominating. In conclusion, we provide a robust, highly generalizable framework that offers significant speedup that Reinforcement Learning engineers can use to parallelize their simulators without needing to write any parallel code.

6 References

Hernandez-Leal, Pablo, Bilal Kartal, and Matthew E. Taylor.“A surveyand critique of multiagent deep reinforcement learning.”AutonomousAgents and Multi-Agent Systems (2019): 1-48.

Yang, Yaodong, et al.“A Study of AI Population Dynamics with Million-agent Reinforcement Learning.”Proceedings of the 17th InternationalConference on Autonomous Agents and MultiAgent Systems. InternationalFoundation for Autonomous Agents and Multiagent Systems, 2018.

Brockman, Greg, et al.“Openai gym.”arXiv preprint arXiv:1606.01540(2016).

Suarez, Joseph, et al.“Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents.”arXiv preprintarXiv:1903.00784 (2019).

Littman, Michael L.“Markov games as a framework for multi-agent rein-forcement learning.”Machine learning proceedings 1994. Morgan Kauf-mann, 1994. 157-163.

7 List of Work

The work was split 50-50 between the two contributors of this project, with both partners working on each segment of the project. Both Venkat and Karthik contributed to the development of the framework code for both Python and CPP, and for the parallelization of the framework code in both CPP and in Python. Both partners also contributed to developing the various implementation examples that we used to test the underlying frameworks. Equal contribution was also provided for writing the reports and generating the posters for all the submissions in this semester.