Parallel Reinforcement Learning

Sam DePaolo, Michael Kielstra, Manqing Liu, and Xiaohan Wu

Harvard University

May 5, 2022

Abstract

In this project, we develop both a single reinforcement learning agent and multiple reinforcement learning agents capable of interacting and learning from the same environment in parallel. We run multiple agents at once using both OpenMP alone and hybrid (OpenMP + MPI), evaluating the performance gained by comparing wall clock time to achieve a certain cumulative average reward per action taken for single agent and multiple agents cases. In addition, we perform roofline and strong and weak scaling analysis to analyze the parallel speedup and efficiency of our code. The optimal synchronization frequency between agents is also evaluated.

1 Background and Significance

Reinforcement Learning (RL) is concerned with how agents ought to take actions in an environment in order to maximize the cumulative reward [1]. At each time t, the agent is told that the environment is in a state S_t. It then selects some action A_t, receiving reward R_t for the state-action choice. The action choice moves the agent to a new state S_{t+1} in the environment, and the process repeats. The goal of the agent is to learn a policy π to maximize the expected cumulative reward $\pi(a, s) = Pr(A_t = a|S_t = s)$ [1, 5]. See figure 1.

Instead of learning the policy with only one agent, one can investigate if the learning process can be accelerated if multiple agents learn the policy in parallel, sharing information with each other during the learning process. Several authors focus on parallelizing *Deep Reinforcement Learning* [6, 3], which combines RL algorithms with neural network technologies, but there is currently only a very small body of research on pure parallel RL. Our main reference [5] did some thorough work on parallel RL and compared the performance of RL for single agent vs. parallel agents. However, this paper did not perform speed-up analysis with parallel agents. In addition, the possible relationship between synchronization frequency between agents and performance of the group as a whole was not explored. It is also not clear how the author hid the latency when sharing information between agents.

1.1 The Mathematics of Reinforcement Learning

The information in this section is taken from [5].

1.1.1 The Sequential Case

Reinforcement Learning, or RL, attempts to solve one fundamental problem: how can we determine the optimal course of action in a specific situation without knowing enough about the environment to predict

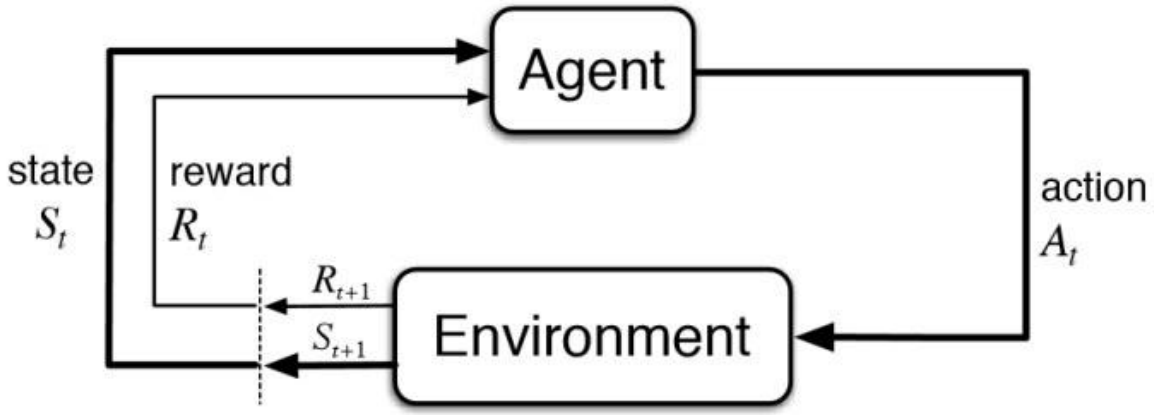


Figure 1: The core RL loop [2].

outcomes ahead of time in an entirely accurate manner? The answer comes down to trial and error, plus assumptions about the nature of the environment that allow us to relate the rewards from different trials to each other and thereby make a better guess at what might be a good action to take next time we find ourselves in a similar situation.

Model-based RL, in which the RL agent attempts to build an internal model of the environment, is a common tactic. In particular, we focus on model-based RL in the case of a *bandit*, an environment with fixed actions and no mutable state. Therefore, any action should give the agent information about what reward to expect if it ever takes that action again.

Traditionally, bandits are designed such that rewards for each action follow some well-known distribution. In order to more accurately simulate real-world complexity, we begin by choosing each reward value from a normal distribution with mean depending on the action and variance 1, but we then *mangle* it, passing it through a complicated increasing function. This changes the characteristics of the distribution but should not affect the effectiveness of the RL agents.

We define a *Q-function* $Q(a)$, which gives the expected value of the reward on taking action a, given the past actions the agent has taken and the rewards it has received. Say that, at time i (that is, after taking i actions), we have taken action a k_a times and our estimated Q-function is $Q_i(a)$. Say that we select action α at this time and receive reward r. Then we define, in the sequential case, the following update rule:

$$Q_{i+1}(a) = \begin{cases} \frac{r+k_\alpha Q_i(\alpha)}{1+k_\alpha} & a = \alpha \\ Q_i(a) & a \neq \alpha \end{cases}$$

We use an *epsilon-greedy* algorithm: fixing some small ϵ—we usually use 0.1—our agent takes a uniformly random action with probability ϵ and the action that maximizes $Q(a)$ with probability $1 - \epsilon$. This ensures both that the agent will receive a high overall reward and that it will not get stuck if an action that usually gives a low reward gives a high one by random chance early on.

1.1.2 The Parallel Case

In the parallel case, multiple agents are acting on the same bandit and would like to synchronize their internal states to learn more effectively. This is complicated by the fact that, due to the randomness of the problem, different agents might have had different experiences. If we naively try to handle this by

simply concatenating all agents' histories of actions and rewards, we end up with a problem where two agents can carry out only a few trials, continually swap information, and end up believing that they have, in aggregate, carried out hundreds or thousands of actions. Certainty gained by talking about the work becomes much more important than certainty gained by actually doing it. We are not in the business of making heavy-handed political metaphors, so we considered a different approach.

Rather than incorporating data from other agents into our own Q-function, we instead keep two separate sets of knowledge: one from our own experimentation, and one from all other agents. Let $\hat{Q}$ be the Q-function calculated from a naive average of all other agents' histories, and let $\hat{k}_a$ be the number of times all other agents have taken action a. Let $\tilde{Q}$ be our own Q-function, independent of other agents, and let $\tilde{k}_a$ be the number of times we have taken action a. This is updated after every action in the same way as for the parallel case. Then we define

$$Q(a) = \frac{\tilde{k}_a \tilde{Q}(a) + \hat{k}_a \hat{Q}(a)}{\tilde{k}_a + \hat{k}_a},$$

which we can use to determine the optimal action.

2 Scientific Goals and Objectives

We have 3 main goals in this project:

- Replicate the work by [5]
- Find an optimal synchronization frequency between agents
- Quantify the speed-up and parallel overhead via roofline analysis and weak scaling analysis

3 Algorithms and Code Parallelization

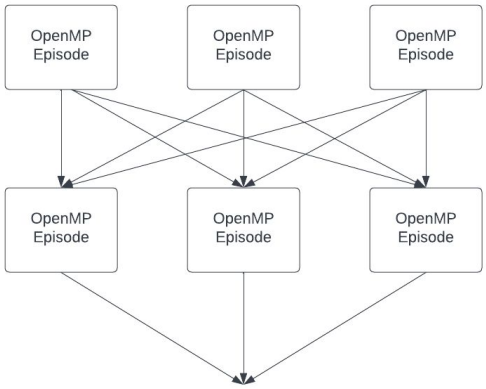
3.1 History synchronization

Whenever an agent takes an action, it must immediately record both that action and the reward it receives. Therefore, parallelizing the core act-record-learn loop of an agent would require prohibitively many atomic operations. Instead, we adopt the philosophy that each thread is its own agent, leaving us with "only" the problem of synchronizing data between them.

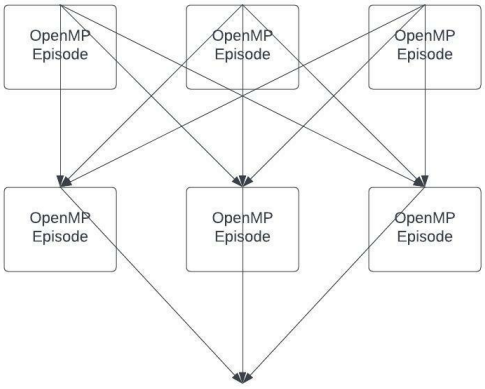
We wrote two variants of parallel code. One runs on only one CPU and manages threads with OpenMP. The other runs on multiple CPUs, internally using OpenMP and communicating between CPUs via MPI. This requires a somewhat complex synchronization structure.

Our first strategy in the OpenMP case, presented here as motivation for our final one, was simply to use a buffer. After completing an episode with a set number of actions, agents would one by one enter a critical section wherein they would write out their histories into a shared array. Then, in parallel, they would read the shared data back into their own private histories, ready to start learning independently again. By sharing total rewards rather than average, we reduce the writing process to a summation over all threads, which lends itself well to such things as MPI reductions.

However, a naive buffer strategy falls victim to the problem presented above: there is no distinction between events that have already been synchronized and those that have not, meaning that all synchronizations treat all events from all agents as new and therefore end up keeping multiple copies of earlier ones. To remedy this, we instituted a two-buffer system as in [5]. All agents would maintain a separation between "my experience" (the results of the action that the agent had taken) and "their experience" (the



(a) An optimistic view of synchronization, where there is sufficiently little latency that the operation can be blocking. This is how we synchronize OpenMP threads.



(b) A more realistic view of synchronization, in which latency must be hidden. This is how we synchronize MPI instances.

Figure 2: Synchronization strategies for agents.

actions and rewards that the agent had only been told about by other agents). To synchronize, all agents would add their "my experience" buffers together, and then, independently, subtract off their own "my experience" again and store the result in "their experience." In the notation of the earlier section, "my experience" stores $\tilde{k}$ and $\tilde{Q}$, while "their experience" stores $\hat{k}$ and $\hat{Q}$.

Having two separate buffers has the advantage of directly applying the mathematics of [5], but has two disadvantages. First, there is no way to do it elegantly. The add-all-then-subtract-your-own approach is quick, but uses superfluous operations, while trying to avoid it would require further buffers or some kind of complicated thread-to-thread communication. More importantly, though, it does not scale as we add more stages of synchronization. Since shared-memory access from within threads is quick compared to MPI communication, we can carry out blocking synchronization between threads on the same CPU in a way that we could not justify when communicating between different CPUs. Instead, we use non-blocking communication, hiding the latency by carrying out new episodes while the old ones synchronize (see figure 2). Doing this with buffers would require storing not only "my experience" and "their experience," but also "MPI-synchronized experience" and "non-MPI-synchronized experience." Such an implementation quickly becomes unwieldy and prone to bugs.

We resolve both of these problems with a *levelled history*. Internally, this consists of an array of size L, each element of which is a naive history called a *level*. Recording new actions and rewards is done by directly modifying level 0.

Levelled histories are not synchronized all at once. Rather, the process begins by fixing a *sync level* $0 \leq l < L$. Then, all relevant threads enter a critical section in which they add their level-l naive histories into a shared buffer. Finally, all histories, in parallel, add the contents of the buffer into their level-$(l+1)$ naive histories and zero out their level l.

Our implementation uses 3-levelled histories. Individual agents write their new actions and rewards into level 0, and then, at the end of the episode, carry out a blocking synchronization that combines all level-0 histories and adds them to each agent's level 1. We think of level 0 as for unsynchronized data and level 1 for locally synchronized. (Note that, since we zero out level 0 as part of this synchronization, we do not need to subtract it away from the contents of the buffer before we add that contents to level 1. The synchronization process moves each agent's level-0 data to level 1 at the same time.)

One agent from each thread can then begin a non-blocking MPI AllReduce operation to collect the contents of each CPU's synchronized level-1 histories into an MPI buffer. While waiting for the request to complete, all agents can carry out further learning, since that changes only level 0. At the apposite time, all threads can add the MPI buffer's data to their level-2 histories and zero out their level 1, leaving it ready to receive the data from the next local synchronization. We think of level 2 as containing globally synchronized data.

We store the levelled history as one array of size $2NL$ where N is the number of possible actions. Recall that L is the number of levels. We assume that L is fairly small, so any function that fixes an action a and iterates over the values for a in each level is going to be fast. The bottleneck therefore arises in functions which iterate over all actions for either a fixed level or all levels at once. To best exploit caching here, we use a row-major order: the first N elements store the action counts for level 0, the next N the total rewards for level 0, the next N the action counts for level 1, and so on. Our implementation of levelled histories comes with functions for recording action-reward pairs, calculating Q-function values, and performing synchronization steps that make this structure completely transparent to the agent.

3.2 Individual agent implementation

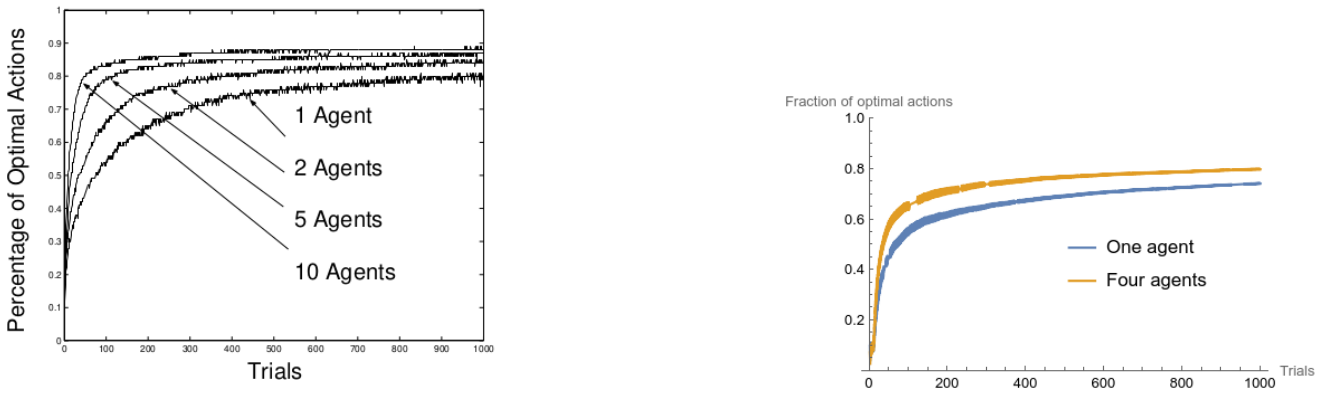
With synchronization out of the way, we can focus on the implementation of the individual agent. As before, to avoid the overhead of a huge number of atomic operations, this is mostly a standard sequential implementation. The only difficulty is the generation of random numbers. Every agent must have an independent PRNG (pseudo-random number generator) in order for their individual experiences to be meaningful to each other, and, while this is easily accomplished across CPUs using MPI (although we do seed the PRNG slightly differently depending on MPI rank, just to be safe), it is more difficult within OpenMP, as the standard C++ PRNG is not thread-safe [4].

To remedy this, we give each thread its own random buffer and use functions from the drand48_r family, which use an explicitly-passed store rather than a hidden global one. The other major random element under the agents' direct control, the initial generation of the bandit, does not require such care, as it is done exactly once. (In fact, we even use an MPI broadcast to ensure that all CPUs' bandits start with the same internal state.)

We are the main developers of all our code, and use no external libraries beyond OpenMP, MPI, and various C/C++ standard libraries.

3.3 Validation, Verification

[5] provides data giving the expected behavior for parallel RL agents in the case where they synchronize after every action. In particular, we duplicate the figure from that paper which gives the fraction of agents which took the optimal action at every timestep. [5] uses one, two, five, and ten agents; in order to generate meaningfully new data while also verifying that our code works, we use one and four. Whatever the number of agents, the procedure for generating the chart remains the same: run n agents for 1000 actions each, synchronizing states after each action, and then determine the fraction of those n which took the correct action, averaged over 1000 trials. (See figure 3.)



(a) Results from [5].

(b) Results from our code.

Figure 3: A comparison of the behavior of our agents to [5]'s in terms of fraction of actions taken that were optimal.

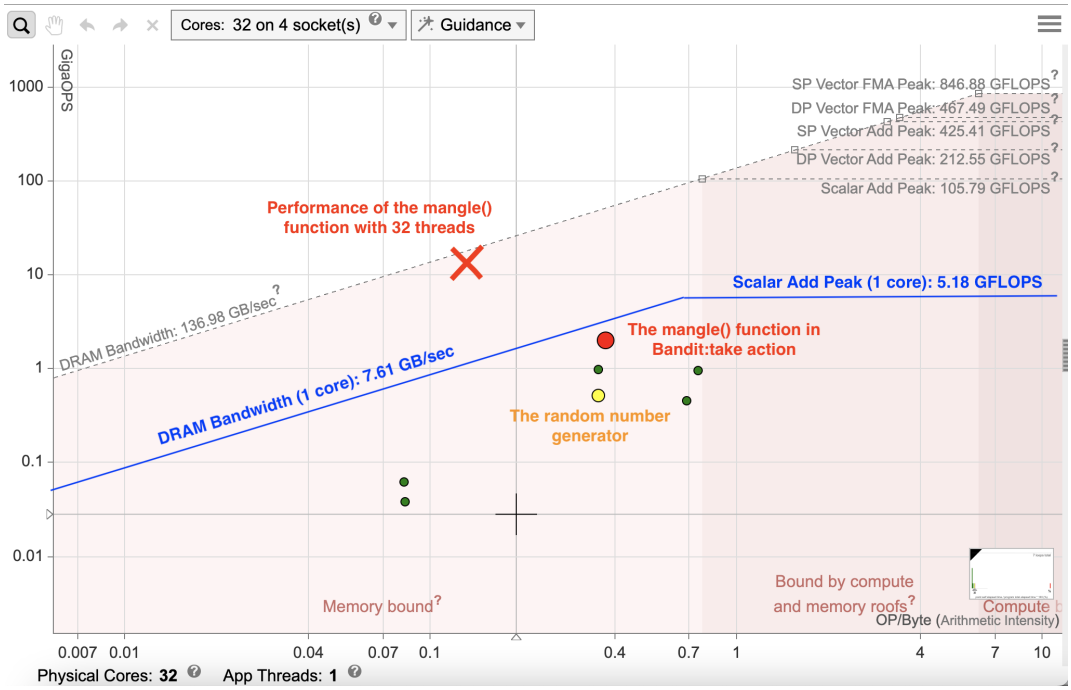
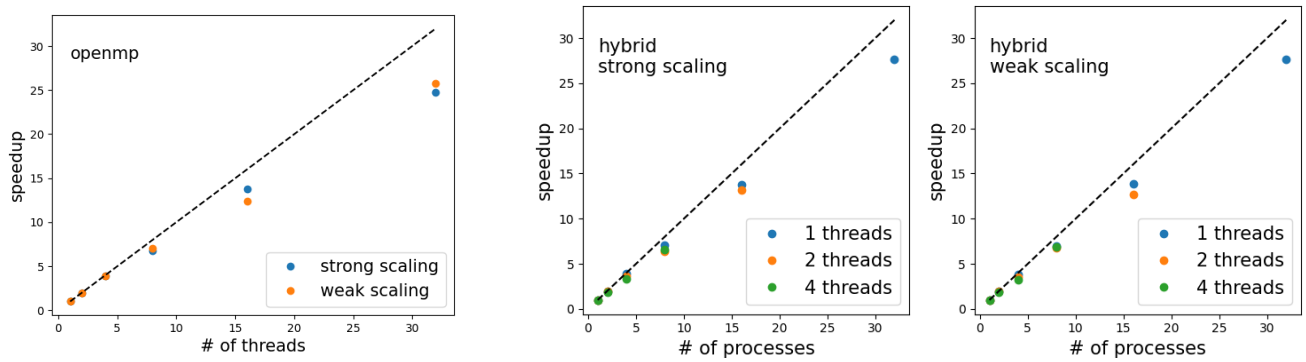


Figure 4: Roofline analysis run with Intel Advisor on our sequential code. The performance analyzed here considered both float and integer operations. Each dot represents a function or loop in our code. The bigger the dots are, the larger the fraction of the total execution time is spent on the corresponding function. The big red dot represents the mangle() function in Bandit::take_action(), and the big yellow dot represents the random number generator we used. The plus represents the overall performance of our code. The blue line shows the roofline of 1 core, while the dashed lines show the rooflines if using 32 cores. The performance of the mangle() function when using 32 OpenMP threads is illustrated by the red cross.



(a) OpenMP

(b) Hybrid OpenMP/MPI

Figure 5: Scaling results for OpenMP and hybrid MPI-OpenMP, performed on Broadwell nodes. Black dashed lines show the one-to-one relation. We varied both the number of OpenMP threads (represented by different colors) and number of MPI processes, but kept their product ≤ 32.

4 Performance Benchmarks and Scaling Analysis

4.1 Roofline Analysis

We performed our roofline analysis using Intel Advisor[1]. This software analyzes the peak arithmetic performance and memory bandwidth of the hardware and examines the fraction of execution time that is spent on each function or loop in an application. Figure 4 shows the roofline analysis results performed on our sequential code. This snapshot considers both float-point operations and integer operations. The plus sign indicates the overall performance of our code. The dots represent the performance of individual kernels. The bigger the dots are, the larger the fraction of the total execution time spent on the corresponding function. The big red dot represents the mangle() function in Bandit::take_action(), and its execution takes up 99.7% of the total time, according to Intel Advisor. The big yellow dot represents the random number generator we used. The blue line shows the roofline of 1 core, while the dashed lines show the rooflines if using 32 cores. The performance of the mangle() function when using 32 OpenMP threads is illustrated by the red cross.

Since the main compute kernel of our sequential code is already close to the 1-core roofline, we are justified in focusing on parallelization in our efforts to speed it up.

4.2 Scaling

We consider first the scaling for OpenMP only, shown in figure 5a. For strong scaling, we kept the total number of episodes fixed as the number of threads grew: each of the n agents carried out T/n episodes for fixed T. For weak scaling, meanwhile, we allowed the number of threads to grow linearly with the agents. We see an almost linear scaling, which is to be expected, since most of the work of the code is done inside agents. It is especially notable that the strong and weak speedups are consistent with each other. Again, this is to be expected, since the amount of work required by each episode is independent of the amount of work required by any other.

Figure 5b shows the strong and weak scaling results (left and right panels respectively) of our hybrid MPI-OpenMP code. We varied both the number of OpenMP threads (represented by different colors)

[1] https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html#gs.zdj9s1

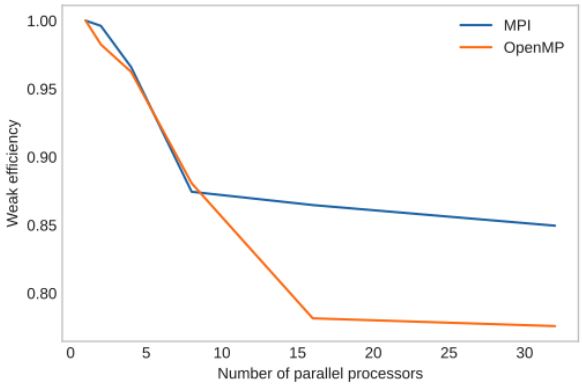


Figure 6: Weak efficiency for OpenMP and MPI, executed on Broadwell nodes using 1, 2, 4, 8, 16, 32 cores.

and the number of MPI processes, but kept their product ≤ 32. For each fixed number of threads, we calculate the speedup by comparing the execution time of using p processes to that using 1 process. For strong scaling, the total number of episodes are the same regardless of the number of threads or ranks. For weak scaling, the total number of episodes scales with the product of the number of threads and the number of ranks. Overall, our hybrid MPI-OpenMP code exhibits very good strong and weak scaling.

Weak Efficiency for OpenMP and MPI (our hybrid code using 1 OpenMP thread) is shown in Figure 6. Efficiency E_p was calculated as time spent using 1 process (T_s) divided by time spent using p processes (T_p). Efficiency for both codes are good, indicating that parallel overhead is hidden well in both cases. In addition, efficiency improves for the MPI code. This makes sense since we did less blocking synchronization and more non-blocking in MPI code. Synchronization between OpenMP threads is blocking, whereas MPI synchronization can be done in a non-blocking way.

4.3 Vital Statistics

All statistics in this table were calculated by running 5000 episodes of 100 actions each. Peak memory use and time were measured with the GNU time utility.

	OpenMP only	Hybrid OpenMP and MPI
Typical wall clock time (seconds)	10	6
Typical job size (nodes)	1	4
Memory per node (KB)	5232	5428
Library used for I/O	None (prints to standard output)	

Table 1: Workflow parameters of the two test cases used during project development.

4.4 Optimal synchronization frequency

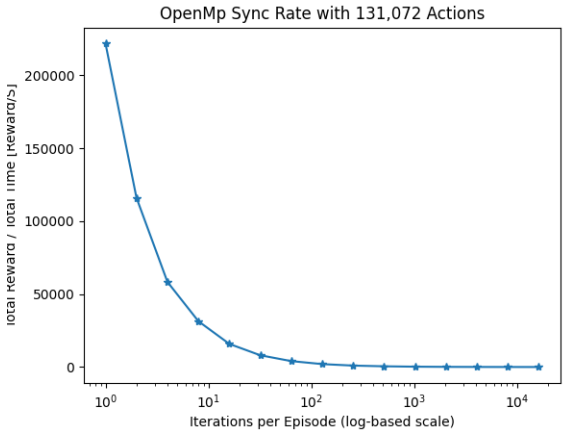
Independently of our investigations into scaling, we looked into the optimal synchronization frequency for both the OpenMP and the MPI implementations. More frequent synchronization leads to greater rewards overall since the agents learn more effectively, while less frequent synchronization saves time due to less

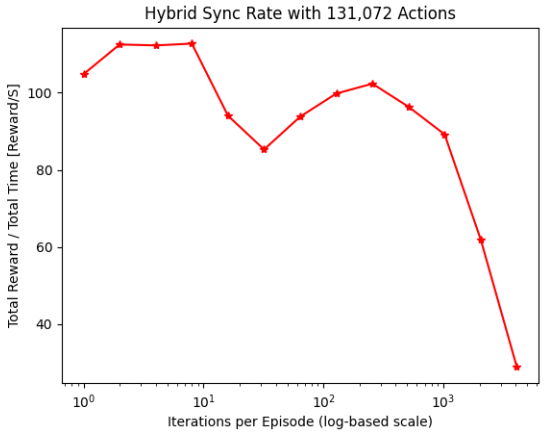
overhead. We therefore attempted to optimize overall total reward per second, holding the number of episodes fixed and varying their lengths

We ran trials where the number of actions (number of episodes $\times$ length of episode) is set to the constant 131072 and the length of episode and number of episodes parameters correspond to powers of 2. For the OpenMP run, we used 8 threads and 8 cores, and for the hybrid version we used 4 nodes and 8 threads and cores per node. From the results, we can see that the latency of synchronization tends to outweigh the added overhead of communicating between processes and threads. In the case of OpenMP, an episode length of 1 yields a significantly larger reward per second than runs with longer episodes. Similarly, our hybrid version peaks with episode lengths of 2, 4, and 8, where communication is frequent.

In cases where the total action count inputted is larger, it is possible that these trends will not completely hold. A larger number of total actions means that a run where the episode length is 1 has processes and threads communicating frequently, and this latency cost may begin to outweigh the learning benefits. However, for the OpenMP version of our code, we did test for a case where episode length was 1 and number of episodes 16777216 as well as the case where episode length is 2 and number of episodes 8388608. Much like our run of the 131072-action case, it seems that the more frequent communication with episode length 1 yields a larger reward per second than an episode length of 2.

Furthermore, overall time must also be a consideration. It took the OpenMP version with 131072 actions to run in 40-50 seconds time, and the hybrid version 10-20 seconds. However, it took the two trials of 16777216 roughly 6200 seconds each, which is a relatively long time. Thus, increasing total actions means increasing the overall time that the program must run, and, at a certain point, the reward/time metric will mean nothing when compared to the overwhelming amount of time that the program requires to run. Thus, in application, we can see that frequent synchronization yields the highest rewards per second.





5 Resource Justification

Up until now, we have had $N = 10$ arms on our bandit. While attractive for benchmarking, this does not always represent the real world, in which there are often many more than ten actions available in any situation. Our proposal, therefore, centers on increasing to $N = 10000$, which will not in and of itself greatly affect the speed of the code since it will still take only one action at a time.

However, learning on bandits with more arms is more difficult, and for this reason we plan to also multiply the total number of actions by a factor of 1000. Carrying this out in the hybrid OpenMP and MPI model, on four cores, would take an estimate of 4 nodes $\times$ 6 seconds $\times$ 1000 = 24000 seconds, or six hours and forty minutes. Due to the randomness inherent in the problem, it would be impossible to

properly explore the same problem without running it a large number of times, so we wish to run it 300 times over.

We therefore request 2000 node hours, as calculated in table 2.

Runs	300
Node hours per run	$6\frac{2}{3}$
Total node hours	2000

Table 2: Justification of the resource request

References

[1] Reinforcement learning. https://en.wikipedia.org/wiki/Reinforcement_learning. Accessed: 2022-04-24.

[2] S. Bhatt. 5 things you need to know about reinforcement learning.

[3] A. V. Clemente, H. N. Castejón, and A. Chandra. Efficient parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1705.04862*, 2017.

[4] cplusplus.com. rand - C++ reference.

[5] R. M. Kretchmar. Parallel reinforcement learning. *The 6th World Conference on Systemics, Cybernetics, and Informatics*, 2002.

[6] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.