Predictive Maintenance using MDP on a Distributed GPU-Cluster

Naman Sharma

***Abstract*— Predictive Maintenance aims to maximize the time between consecutive maintenance while also minimizing the number of unforeseen failures. Intense research has been conducted in improving the accuracy and capabilities of predictive maintenance algorithms. In this work, we use the notion of Markov Decision Processes (MDP) to tackle the problem of predictive maintenance. This thesis first attempts to fit machine telemetry data to an MDP model by using machine learning models such as Decision Trees, Random Forests and SVM. We then analyze the results obtained by solving the modeled MDP. This thesis also aims to develop a framework to allow large MDP problems to be solved in a distributed GPU cluster. Finally, this thesis analyzes the feasibility of such a distributed algorithm. The results of the analysis show that such a framework is indeed useful for solving large scale MDP problems more efficiently. Moreover, such a distributed algorithm can also be used in a single GPU to improve the concurrency in the code.**

I. Introduction

Maintenance costs are always a significant part of any industry. These costs make up 40-50% of all operational budget in any industry [1]. However, 1/3rd of all this cost is wasted because of improper or unnecessary maintenance.

Predictive maintenance can offer much greater benefits over reactive maintenance. According to Intel's 2016-17 Annual Performance Report [2], the company was able to save $656 million a year due to predictive analytics.

The aim of this paper is to use MDP models to solve the predictive maintenance problem in a distributed manner. Further, the paper aims to use the computational capabilities of GPU-enabled edge devices to provide both efficiency in terms of time for computation and the ability to use multiple GPUs to solve the same MDP problem.

II. Literature Review

A. Predictive Maintenance with Markov Models

Utilizing Markov models for optimizing maintenance is not a novel approach: Dawid et al. [3] performed a survey of Markov models used in maintenance in the context of Offshore wind. They discuss other papers using the different types of Markov models in maintenance: Markov Chains, Markov Decision Processes (MDPs), Hidden Markov Models (HMM) and Partially Observable MDPs (POMDPs).

More recently in 2014, Neilsen and Sorensen [4] compared multiple ways of solving decision making problems and found MDP to be one of the most promising ways of optimizing decision policies. Yao et al. [5] proposed a two layer model to perform maintenance scheduling for semiconductor manufacturing systems. The MDP model acts as the higher-level model and provides "maintenance windows" in which maintenance tasks should be performed.

Naman Sharma is with the Department of Electrical and Computer Engineering, National University of Singapore, 4 Engineering Drive 3, Singapore (phone: 65-8403-4113; email: a0133877@u.nus.edu).

B. Computational Offloading

Edge computing allows computational tasks to be performed near to the devices responsible for data collection. However, edge devices usually have a smaller computational capacity than the cloud. We have therefore seen research being focused on how to distribute computation over mobile cloudlets.

In [6], Zhang et al. use an MDP to dynamically decide when to offload different application phases to connected mobile devices based on the number of connected devices available at any given point in time. This was extended in [7] by considering not an application but instead tasks that can be performed in parallel in cloudlets. It also aims to minimize the processing cost by taking into account cases when the offloading is unsuccessful when the mobile user "moves out" of the range of the cloudlet, requiring the task to be reallocated to another device. Le and Tham [8] further improved upon this idea to take into account distances between the mobile devices and also the varying properties of the wireless channel.

In this paper, we will develop algorithms to solve large MDPs over mobile cloudlets. The decision on whether to offload this computation to nearby cloudlets can itself be formulated as an MDP based on the above mentioned papers and solved on a cluster, allowing for a complex system intelligently deciding how to solve MDP models to provide predictive maintenance over time.

III. MDP Model

In this paper, we will use Markov Decision Processes (MDPs) to model the status of the machines. MDPs can be mathematically defined using the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$, [9] [10] where:

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the set of possible actions
- $\mathcal{P}$ describes the state transition probabilities. Therefore,
$$\mathcal{P}^a_{ss'} = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a)$$
- $\mathcal{R}$ is the reward function.
$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$
- γ is the discount factor, $\gamma \in [0, 1]$

The behavior of any agent following a MDP can be completely defined by a policy $\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$. Based on a policy, we can define the value function $V_\pi(s)$ of the MDP as:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(\mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} V_\pi(s') \qquad (1)$$

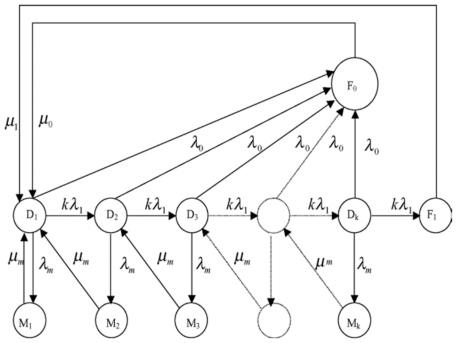

Fig. 1. Markov Process model for Predictive Maintenance as proposed in [11]

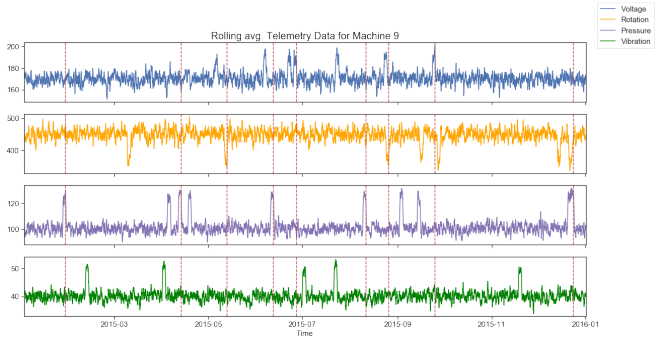

Fig. 2. Telemetry data for machine 9 belonging to model 1. Dotted vertical lines correspond to failure events.

Eq. (1) can be described as the immediate reward of performing action a ($\mathcal{R}_s^a$) added with the discounted reward from the future states, averaged over the probability of performing action a ($\pi(a|s)$).

Ultimately, solving a MDP problem corresponds to finding the optimum value function $V_\pi^*(s)$ which corresponds to the optimum policy $\pi^*(a|s)$.

A. Value Iteration

Value Iteration (VI) is one of the most commonly used dynamic programming algorithms to solve an MDP. It involves starting with an initial value function $V_0(s)$ and then iteratively applying Eq. 2. This is known to converge to the optimum value function $V^*(s)$.

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \tag{2}$$

Each iteration of the Bellman update has the complexity of $\mathcal{O}(|\mathcal{A}||\mathcal{S}|^2)$ and the maximum number of iterations required to achieve a greedy policy equal to the optimal π^* is polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$ and $\frac{1}{(1-\gamma)\log\frac{1}{1-\gamma}}$.

B. MDP Model for Predictive Maintenance

Chan and Asgarpoor [11] provide a Markov Process model that can be used to model a PM task and find the optimal time to preventive maintenance. Their model is depicted in Fig. 1. They differentiate between failures caused due to deterioration, state F_1 and random unforeseen failure, state F_0. They also split the deterioration of the machine into k discrete states D_k, where a higher k refers to a larger degree of deterioration. The time spent in state D_k is exponentially distributed with a mean of $1/k\lambda_1$. Maintenance tasks occur as a Poisson process with parameter λ_m. The duration of these maintenance tasks is exponentially distributed with a mean of $1/\mu_m$. It is assumed that a maintenance task at state D_k improves the state of the machine only partially, bringing it back to state D_{k-1}. The random failures that occur are distributed as a Poisson process with parameter λ_0.

This Markov model is then converted into a MDP by considering a decision of "do maintenance" or "do nothing" at every deterioration state. We can then assign use case specific rewards associated with every state. For example, D_k state can have decreasing positive rewards as k increases. Maintenance can have a small negative reward. F_0 and F_1 should both have large negative rewards, where potentially $|R(F_0)| < |R(F_1)|$ since F_0 failures are by nature random whereas F_1 failures can be prevented.

This FYP aims to apply the above MDP model to the publicly available data by Microsoft Azure [12].

IV. Estimating MDP Model Parameters

The simulated machine dataset consists of 1000 machines belonging to 4 different models. In Fig. 2 we plot the telemetry data for one of the machines. We can immediately see a pattern between disturbances in the telemetry data and the failure events.

A. Data Preprocessing

The performance of machine learning algorithms depends to a large extent on preprocessing of data and removing its inherent noise.

Table I describes the pre-processing of the dataset to get the input X for the ML algorithms. Averaging the telemetry data over rolling windows of multiple lengths allows us to get reduced noise and yet keep accurate values for the sensor readings. For defining the deterioration failure state F_0, the labels y are defined as:

$$y = \begin{cases} 0, & normal \\ 1, & failed \end{cases}$$

where $failed$ corresponds to any data-entry within 1.5 days from the failure event.

B. ML Results Comparison

The data preprocessed in Section IV-A consists of 62,135 data points. Out of these, 60056 (96.65%) are labeled *normal*, whereas only 2079(3.35%) is labeled *failed*. Hence we see that our data is highly imbalanced. For such data, accuracy is not a good measure of performance since a model that labels all points as healthy is already 96.65% accurate. In such cases we will rely on other measurements of performance.

1) Precision $= \frac{TP}{TP+FP}$ where TP stands for True Positives and FP stands for False Positives.

TABLE I

DESCRIPTION OF PROCESSED DATASET X GIVEN AS INPUT TO THE ML ALGORITHMS

Column	Description
$0-3$	Telemetry data (voltage, rotation speed, pressure, vibration) averaged over a 12hr rolling window. The telemetry data is aligned with the other data by averaging the values over a tumbling window for 12 hours for each rolling window.
$4-7$	Standard deviation of telemetry data over a 12hr rolling window.
$8-23$	Mean and standard deviation of telemetry data over a rolling window of 24hrs and 36hrs.
$24-28$	Total number of errors (type 1 to 5) that occurred in the 12hr rolling window.
$29-38$	Total number of errors (type 1 to 5) that occurred in the rolling window of 24hrs and 36 hrs.
$39-42$	Days since the component (1 to 4) was replaced for that machine
43	Age of the machine.

2) Recall $= \frac{TP}{TP+FN}$ where FN stands for False Negatives.

A good model should have a high precision and high recall. However, these objectives are usually opposing. In our case, precision is much more important than recall. This is because we would like to be sure that we define the failure state F_0 with a high degree of confidence.

The dataset is split so that we have a $75\% - 25\%$ train-test split. We then perform a grid search to find the optimal model that fits the data best. The performance results for each ML model is tabulated in Table II.

TABLE II

RESULTS OF GRID SEARCH ON DATASET

		Decision Tree	Random Forest	SVM
Train	**Accuracy(%)**	99.26	99.36	97.79
	Precision(%)	99.69	95.13	96.23
	Recall(%)	78.93	85.87	37.92
	F-Score(%)	88.11	90.26	54.40
Test	**Accuracy(%)**	98.94	98.82	98.07
	Precision(%)	90.49	86.81	96.53
	Recall(%)	71.91	70.75	35.91
	F-Score(%)	79.95	77.96	52.35

From Table II, we see that for all models the accuracy score is high, as expected. However, the SVC model is most stable across the training and test sets. The random forest performs low in terms of both precision and accuracy. Since the data is extremely imbalanced, the number of trees in the forest mislabeling true positives easily outnumbers the ones that correctly label it. The decision tree provides a good balance between precision and recall, and hence has the highest F-score. However, since we prefer to have high precision, we make a design choice of selecting the SVC model to define the deterioration failure state F_0.

C. Deterioration States

The next step to model the MDP using the data is to define the different deterioration states. The SVM model

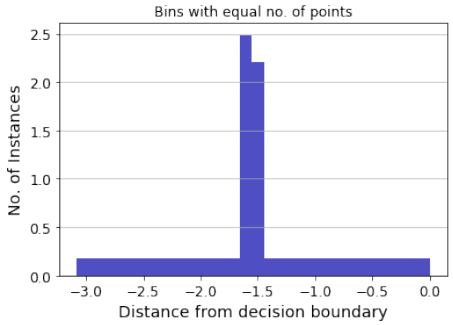

Fig. 3. Distribution of distance of data from decision boundary of SVC model with bins of equal density

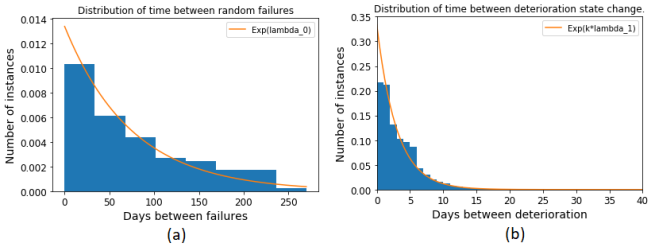

Fig. 4. Density distribution of time between (a)random failures, and (b)deterioration state change

classification is based on a decision boundary. Hence, we can naturally define the deterioration states based on how far the machines' current statistics lie from the decision boundary. The closer it is to the decision boundary, the greater the deterioration state $D_i, \forall i \in \{0,1,2,3\}$. Given that the time spent in each deterioration state is identically distributed, we make the design choice of splitting the distribution into deterioration states such that each state has equal density. This results in the deterioration states being defined based on the bins depicted in Fig. 3.

We can now use this information to distinguish between deterioration failures (F_0) and random failures (F_1). Deterioration failure is defined as the failures caused when the machine was in state D_3. Any failure that occurs even though the deterioration state was $D_i, \forall i \in \{0,1,2\}]$ is considered as random failure. We calculate time between random failures and use these values to calculate the Maximum Likelihood Estimate (MLE) of λ_1 using Eq. 3.

$$\hat{\lambda_1}^{MLE} = \frac{n}{\sum_{j=1}^{n} x_j} \tag{3}$$

Fig. 4(a) plots the density distribution of the days between random failure. The assumption about the data being exponentially distributed holds well. The average time between consecutive random failures is 74.82 days. Similarly, from Fig. 4(b) we see that the average time to deterioration failure is 12 days.

Finally, in Fig. 5 we plot the same telemetry data as in 2. However, in this figure we represent the deterioration state of the machine by the color of the data-point, and also add information on when the maintenance was performed. We

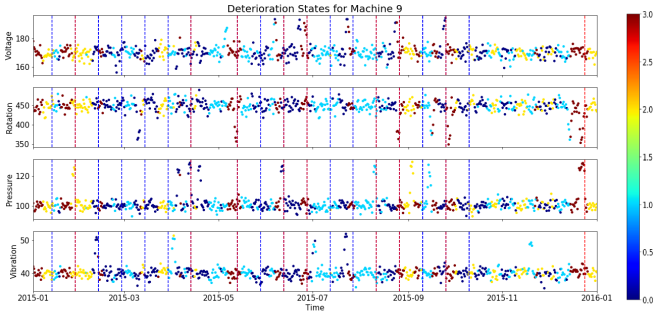

Fig. 5. Scatter plot of machine 9 telemetry data. Color of data-point depicts deterioration state. Red (resp. Blue) vertical dotted lines correspond to failure (resp. maintenance).

observe that the deterioration states are defined satisfactorily, with most failures occurring when the machine is in D_3. An exception to this is the $2nd$ and $6th$ failures that occurs on approximately $2015-04-15$ and $2015-08-10$ respectively. These two failures correspond to the *random* failures of machine 9.

D. Optimum Policy

Now that we have fit the MDP model to our data, we shall solve the MDP using the Nova library [13] on a single machine and comment on the results obtained. Table III lists the parameters of the MDP.

TABLE III
SUMMARY OF MDP PARAMETERS AND OPTIMAL POLICY.

	State	Parameters	Reward	Optimal Policy a_0: do nothing a_1: maintenance
Deterioration	D_0	$k\lambda_0 = 0.3334$	1000	a_0
	D_1		900	a_0
	D_2	$\lambda_1 = 0.0134$	600	a_1
	D_3		500	a_1
Maintenance	M_i	$\mu_m = 0.5$	-200	-
Failure	F_0	$\mu_0 = 0.25$	-1000	a_1
	F_1	$\mu_1 = 0.125$	-700	a_1

The optimal policy obtained is expected since the probability of failure is much higher for the higher deterioration states. The optimal value function for the initial state D_0 is calculated to be 6164.5. It is interesting to note that as the mean time to deterioration failure is increased, the number of states with optimal policy of '*do maintenance*' decreases.

V. MDP VALUE ITERATION: A DISTRIBUTED APPROACH

A. Algorithm Description

We denote the k nodes of the cluster by $n_c, \forall c \in \{1, 2, \ldots, k\}$. The dimensions of the MDP problem are denoted by: $N = |\mathcal{S}|$ the number of states, $M = |\mathcal{A}|$ the number of actions. Therefore, the transition probability matrix $\mathcal{P}^a_{ss'}$ has dimensions of (N^2M). The reward function $\mathcal{R}^a_s$, the value function $V_\pi(s)$ and the policy π are all vectors of length N.

Algorithm 1: Distributed Value Iteration to solve MDP

Input: Transition Probabilities $\mathcal{P}^a_{ss'}$, Reward Function $\mathcal{R}^a_s$, horizon h
Output: Optimal Policy $\pi^*(s)$

Procedure `DistributedVI(`c`)`**:**
  `GPU_Init(`$\mathcal{P}^a_{ss'}$, $\mathcal{R}^a_s$`)`
  $s_{start} \leftarrow (c-1)\lceil \frac{N}{k} \rceil$
  $V(s) \leftarrow [0] \quad \forall s \in \mathcal{S}$
  for $i \in \{1, 2, \ldots, h\}$ **do**
    $V(s), \pi(s) \leftarrow$ `GPUBellman(`$\mathcal{P}^a_{ss'}, \mathcal{R}^a_s, s_{start}$`)`
    $V_{temp}(\tilde{s}) \leftarrow$ `GPU_copy(`$V(s)$`)` $\quad \forall \tilde{s} \in \{s_{start}, \ldots, s_{start} + N_c\}$
    $V(s) \leftarrow$ `MPI_AllGatherv(`V_{temp}, V, c`)`
  `MPI_AllGatherv(`π, c`)`
  return $\pi(s)$;

Any attempt to distribute the value iteration algorithm over a cluster requires that the value function be shared across all nodes after every iteration of the Bellman update. In our implementation, the MDP is split across the different nodes by assigning a fraction of the total states to a single node for the Bellman update. Therefore, the number of states N_c that node n_c is responsible for is given by:

$$N_c = \begin{cases} \lceil \frac{N}{k} \rceil & , \forall c \in \{1, 2, \ldots, k-1\} \\ N \mod \lceil \frac{N}{k} \rceil & , c = k \end{cases}$$

Algorithm 1 describes the overall algorithm that is followed by each node n_c. The `GPUBellman` function performs the Bellman update to change the values of $V_\pi(\tilde{s})$and$\pi(\tilde{s})$, where $\tilde{s}$ are the states for which the node is responsible for. The part of $V_\pi(s)$ that was updated is then copied into a temporary variable V_{temp} using `GPU_Copy`. The function `MPI_AllGatherv` is defined under the Message Passing Interface (MPI) standard which is responsible for performing a concatenation of data from different nodes. Therefore, if we consider V^c_{temp} from each node n_c, after `MPI_AllGatherv`, $V_\pi(s) = [V^1_{temp}, V^2_{temp}, \ldots, V^k_{temp}]$, which is the value function of the MDP problem after one iteration. This is repeated for number of horizons, h times.

B. Time Complexity Analysis

The bellman update iterates over all states and computes the reward obtained by taking all actions, averaged by the probability of reaching all other states. Therefore, a single iteration of the bellman update has a time complexity of $\mathcal{O}(N^2M)$.

In our proposed algorithm, the cost of a single iteration C_T can be divided into processing cost C_P and communication cost C_C. By parallelizing over all possible states, we reduce the processing time required for the update to $\mathcal{O}(NM)$. However, to prepare for the communication process, we need to copy segments of the value function into temporary memory spaces. This is done is parallel for each state and hence can be performed in constant time $\mathcal{O}(1)$.

$$C_P = C_{Bellman\ update} + C_{copy} = \mathcal{O}(NM) + \mathcal{O}(1) \quad (4)$$

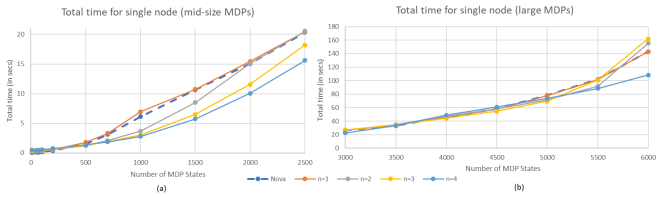

Fig. 6. Comparison of time using Nova and Distributed algorithm for (a) Medium-size MDPs and (b) Large MDPs

The communication cost incurred by the proposed algorithm depends on the number of nodes that are present in the network. The concatenation algorithm implemented in most versions of MPI was proposed by Bruck et al. [14]. It uses a circulant graph to perform the communication between k nodes in $d = \lceil \log_{p+1} n \rceil$ iterations, where p is the number of ports available for the communication. In the first $d-1$ rounds, each node sends its neighbor(in the circular sense) its data, with the neighbor being an *offset* away from the node in each iteration. The last iteration involves solving a table partitioning problem to schedule the communication between the nodes for the remaining data. Bruck et al. show that the amount of data transmitted in this entire process is $C_{data} = \lceil \frac{b(k-1)}{p} \rceil$ where b is the size of the largest data segment amongst all the nodes.

Therefore, given a bandwidth B between the nodes, the total time complexity of the proposed algorithm can be written as:

$$\begin{aligned} C_T &= C_P + C_C \\ &= \mathcal{O}(NM) + \frac{1}{B}\lceil \frac{b(k-1)}{p} \rceil \end{aligned} \tag{5}$$

As we can see this greatly reduces the time complexity for the VI algorithm. Moreover, in comparison with the Nova implementation, this algorithm allows us to distribute the computation over a cluster, which allows us to get better occupancy in a single GPU as well as use multiple GPUs to solve larger problems.

VI. Results

This section is split into two sub-sections, where the first deals with analyzing the performance of the proposed algorithm on a single GPU node and the second focuses on how this approach performs in a cluster with multiple GPUs.

A. Single GPU / Node

1) Nova vs. Distributed: Fig. 6 shows how the distributed approach performs in comparison to the bare C++ implementation of Nova. In Fig. 6(a) the dotted line corresponds to Nova. We see that once the size of the MDP becomes larger than around 200 states, the distributed algorithm is more time-efficient.As the number of processes increases from 2-4 we see that the improvements become progressively larger. This can be explained using Fig. 7 where we see that a small problem that does not saturate the CUDA cores on the GPU can be further split into multiple kernels to be run in parallel, leading to time saving.

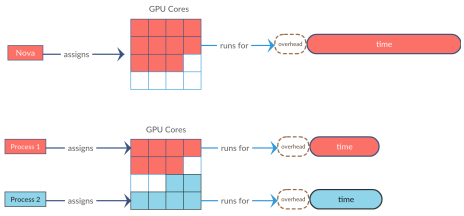

Fig. 7. How running multiple process on a single Node can reduce runtime.

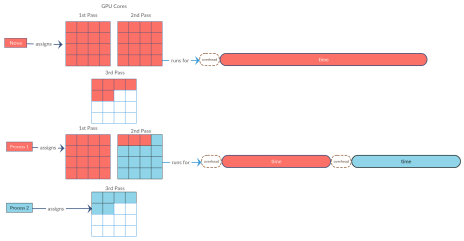

Fig. 8. How running multiple process on a single Node can increase runtime.

In Fig. 6(b) we see the timing comparison for very large MDPs. Here we see that the distributed approach performs only slightly better than the Nova approach. This is because the larger MDPs saturate the CUDA cores available to each process, forcing them to run sequentially. In the case of 6000 states, we see that the time taken by $n = 2$ and $n = 3$ is even greater than the time taken by Nova. This can be explained by referring to 8 where a large MDP even after splitting can led to a longer solve time.

2) Communication vs. Computation: Fig. 9 plots the relative importance of the three main components of the distributed VI algorithm. As expected the `GPU_Copy()` is a very small part of the VI algorithm and completes in constant time. As the size of the MDP becomes significant, the importance of the `MPI_Allgatherv()` becomes negligible too. This is primarily because the communication takes place within the same node and hence there is no bandwidth restriction involved, allowing almost instantaneous exchange of information between the processes. The algorithm's time complexity is almost entirely dependent on the time complexity of the Bellman Update.

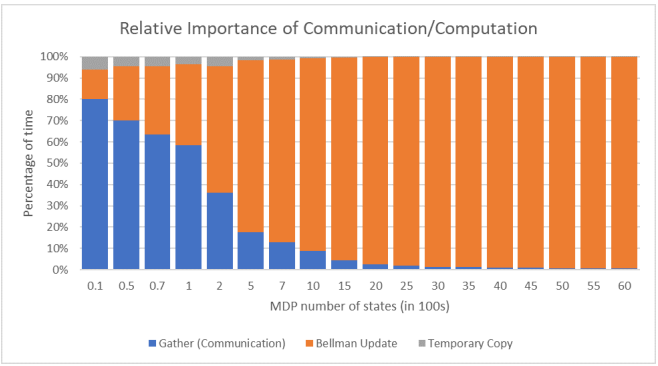

Fig. 9. Relative importance of communication time vs. computation time.

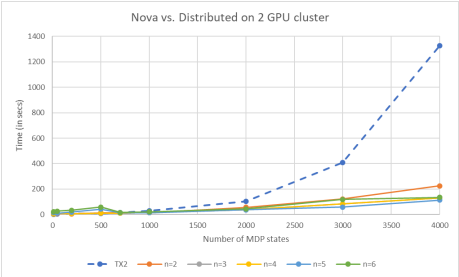

Fig. 10. Timing comparison between Nova and Distrusted Algorithm in 2-node cluster.

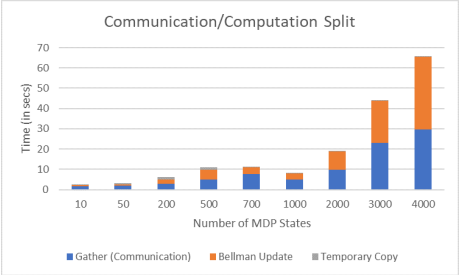

Fig. 11. Relative importance between communication and computation time for $n = 4$.

B. Multiple GPUs / Cluster

1) Nova vs. Distributed: In Fig. 10 we see that the distributed algorithm outperforms the Nova in both medium and large MDPs. However, we note that even in the case of large MDPs, increasing the number of processes does not always increase performance.

2) Communication/Computation Breakdown: Fig. 11 plots the relative importance between the three main components of the distributed algorithm for the case of $n = 4$ running on both nodes. We chose to depict the figure for the case $n = 4$ since it is the case that performs the best in most MDP inputs. As before, `GPU_Copy()` takes almost constant time. However, unlike in Fig. 9, in this case the importance of `MPI_Allgatherv()` is significant. This is expected because the data is shared over a WI-Fi connection. There is a restricted bandwidth available, and we see the communication time increases as the size of the data to be transmitted increases. The best-performing split between the GPU resources provides an almost equal balance between communication time and computational time.

C. Summary

There are multiple factors that affect the speed of the distributed algorithm:

- Throughput of CUDA Threads: CUDA threads need to do enough FLOPS per byte for the distributed approach to show benefits.
- Bandwidth: The bandwidth between the nodes plays a major role in deciding the scalability of the cluster.
- Number of processes on single node: Increasing the number of processes after the GPU is already optimally used will slow down the computation time.
- Heterogeneous GPUs: In the case of heterogeneous clusters, it is important to properly allocate the problem size to each GPU.
- Size of input: Larger MDPs show better improvements.

VII. Conclusion and Future Work

In conclusion, this thesis uses machine data to provide Predictive Maintenance by modeling the decision to perform maintenance as a Markov Decision Process (MDP). Machine Learning methods are applied to accurately fit the available data to the model.

Finally, the thesis presents an extension of the existing Nova library to provide the capability to distribute Value Iteration of an MDP over a multi-GPU cluster. The presented distributed approach is analyzed and showed to perform better than the existing library for large and complex MDPs.

Acknowledgment

The author would like to express his deepest and most sincere gratitude to his thesis advisor, Prof. Tham Chen Khong for his support and continuous guidance. He would also like to thank his examiner Prof. Mohan Gurusamy, for his constructive feedback and support during the Continuous Assessments. Finally, the author would not have been able to complete this thesis without the support of his family throughout his journey.

References

[1] D. Mather, "The Future of CMMS," 2015.
[2] Intel, "2016-2017 Intel IT Annual Performance Report," Tech. Rep., 2017.
[3] R. Dawid, D. Mcmillan, and M. Revie, "Review of Markov Models for Maintenance Optimization in the Context of Offshore Wind," *Annual Conference of the Prognostics and Health Management Society 2015*, pp. 1–11, 2015.
[4] J. S. Nielsen and J. D. Sørensen, "Methods for risk-based planning of O&M of wind turbines," *Energies*, vol. 7, no. 10, pp. 6645–6664, 2014.
[5] X. Yao, M. Fu, S. I. Marcus, and E. Fernandez-Gaucherand, "Optimization of preventive maintenance scheduling for semiconductor manufacturing systems: Models and implementation," *IEEE Conference on Control Applications - Proceedings*, pp. 407–411, 2001.
[6] Y. Zhang, D. Niyato, P. Wang, and C. K. Tham, "Dynamic offloading algorithm in intermittently connected mobile cloudlet systems," in *2014 IEEE International Conference on Communications, ICC 2014*, 2014.
[7] T. Truong-Huu, C. K. Tham, and D. Niyato, "To offload or to wait: An opportunistic offloading algorithm for parallel tasks in a mobile cloud," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2015.
[8] D. V. Le and C. Tham, "An Optimization-Based Approach to Offloading in Ad-Hoc Mobile Clouds," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.
[9] D. Silver, "Markov Decision Processes," 2015.
[10] F. Garcia and E. Rachelson, "Markov Decision Processes," in *Markov Decision Processes in Artificial Intelligence*, O. Sigaud and O. Buffet, Eds., 2010, ch. 1, pp. 3–37.
[11] G. K. Chan and S. Asgarpoor, "Optimum maintenance policy with Markov processes," *Electric Power Systems Research*, vol. 76, no. 6-7, pp. 452–456, 2006.
[12] Microsoft, "Simulated Predictive Maintenance Dataset," 2016.
[13] K. H. Wray and S. Zilberstein, "A Parallel Point-Based POMDP Algorithm Leveraging GPUs," pp. 95–96, 2015.
[14] J. Bruck, C. T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, 1997.