
Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2024 SCPE. Volume 25, Issues 5, pp. 3812–3821, DOI 10.12694/scpe.v25i5.3001

SCALABLE AND DISTRIBUTED MATHEMATICAL MODELING ALGORITHM DESIGN AND PERFORMANCE EVALUATION IN HETEROGENEOUS COMPUTING CLUSTERS

ZHOUDING LIU* AND JIA LI†

Abstract. A growing number of scalable and distributed methods are required to effectively simulate complicated events as computing needs in the research and industrial sectors keep growing. A novel approach for developing and accessing mathematically modeled methods in heterogeneous computing clusters is proposed in this study to meet this difficulty. The suggested methodology uses DRL based Parallel Computational model for the evaluation of Heterogenous computing clusters. The algorithms makes use of parallelization methods to split up the processing burden among several nodes, supporting the variety of topologies seen in contemporary computing clusters. Through the utilization of heterogeneous hardware parts such as CPUs, GPUs, and acceleration devices, the architecture seeks to maximize speed and minimize resource usage. To evaluate the effectiveness of the proposed approach, a comprehensive performance assessment is conducted. The evaluation encompasses scalability analysis, benchmarking, and comparisons against traditional homogeneous computing setups. The research investigates the impact of algorithm design choices on the efficiency and speed achieved in diverse computing environments.

Key words: heterogeneous computing clusters, scalability, distributed mathematical modeling, parallelization methods

1. Introduction. The intricacy of mathematical representations has increased in the dynamic field of computational disciplines, calling for creative methods of algorithm creation and efficiency enhancement. A key concept for addressing the growing computing needs of complicated mathematical models in a range of scientific and engineering fields is scaled distributed computers. With an emphasis on the assessment of performance in heterogeneous computing clusters, this research sets out to investigate and expand the boundaries of scalability and dispersed mathematical modeling method design.

High Performance Computing (HPC) is the term used to describe the process of solving challenging issues in the sciences, engineering, or industry by pooling computing resources in a way that yields efficiency substantially greater than that of a typical personal computer or workstation [2, 10]. The terms comparable to HPC are parallel computing and supercomputing. The underlying principle of HPC is the fact that we can accomplish a problem with 100 processors in an hour, whereas a single computer requires 100 hours to finish. While utilizing all the resources available to it, a single node inside the supercomputer might not be stronger than others.

Heterogeneous ML structures, such as TensorFlow [2], MXNet [10], and PyTorch [16], are frequently used to perform ML workloads to speed up the training process over large datasets or large models. In a distributed machine learning task, the data set is split up and taught by a distinct worker. To obtain the global parameters, the workers share computed model parameters with one another (either directly via an all-reduce aggregate or via parameter servers). It is typical for workforce and parameter hosts in a parameter server (PS) architecture to be dispersed across multiple physical servers, either because they cannot be fully hosted on a single server or to optimize capacity fragmentation use on servers.

In recent times, numerous high-performance computing (HPC) applications, including modeling of the climate and environment, computational fluid dynamics (CFD), molecular nanotechnology for smart planet rockets, and numerous other big data uses, have required extremely powerful computing systems to handle them. According to experts and HPC pioneers, "exascale systems," a new class of supercomputing computers, won't be introduced until the beginning of the following decade [16, 6]. Compared to current Petascale systems, this heterogeneous architectural-based HPC computing platform will offer a thousand-overlay speed boost. With an HPC machine this powerful, many scientific puzzles will be solved in a matter of seconds, completing

*College of Arts & Science. New York University. 10003 NY USA, (zhoudingliusee@outlook.com)
†Master of Economics Arts and Social Sciences, University of Sydney, city road, camperdown, NSW 2006, Australia

ExaFlops worth of computations [7].

Developing and optimizing device executable software to take advantage of the substantial amount of parallelism is the primary difficulty in GPGPU computation. Programmers have two possibilities: the vendor specific CUDA [9, 4] programming environment or the OpenCL standard programming framework [15], which allows programs to operate across the GPU and CPU architectures of most manufacturers. Most apps, including MPI apps that make use of GPU gadgets, presently execute their kernel code locally on the same devices as their CPU routines.

The establishment of Exascale computing systems is expected to consist of many heterogeneous nodes, each of which will be outfitted with multiple-core enhanced GPU devices and regular multi-core CPUs [14, 5]. At a moment when the need for more computing capacity is growing, the emergence of heterogeneity in HPC systems is resulting in increasingly complicated platforms. The major use of electricity while HPC processing of information is a challenge for current supercomputing systems. Recent HPC supercomputing systems support up to 10 million cores per node, with an annual electricity consumption of 25–60 MW.

The main contribution of proposed method is given below:

1. To bring together coarse-grain, fine-grain, and greater granularity through inter-node, intra-node, and enhanced GPU calculations, a novel DRL based hybrid MPI + OpenMP + CUDA (MOC) massive parallel computing paradigm was proposed for Exascale computing systems.
2. Using various kernel widths, we applied MOC to dense matrix multiplication in linear algebra and assessed HPC parameters such as energy consumption and speed.
3. We solved the identical issue using two of the most well-known linear algebra subroutines archives, CuBLAS and KAUST basic linear algebra subprograms (KBLAS). Moreover, we contrast the outcomes with the framework proposed by MOC.

Remaining sections of this paper are structured as follows: Section 2 discusses about the related research works, Section 3 describes the Heterogenous Computing Clusters, Parallelization and Deep Learning methods, Section 4 discusses about the experimented results and comparison and Section 6 concludes the proposed optimization method with future work.

2. Related Works. It makes sense to co-locate occupations with minimal levels of interference to maximize training success [21]. Unfortunately, because it is challenging to determine the possible interference levels of several jobs, schedulers now in use in real-world machine learning clusters ([25], Mesos [4]) are primarily unaware of disruption, which results in prolonged training times and less-than-ideal utilization of resources. Numerous studies have demonstrated the potential and efficacy of interference-aware planning in the literature, such as when it comes to taking network traffic into account for MapReduce operations [17, 18], and cache access severity for HPC jobs [1]. Based on specific facts or hypotheses (e.g., that disruption slows back performance exponentially), these researchers construct an explicit delay model of the goal performance and use custom heuristics to include interference in scheduling.

In contrast to previous methods, we adopt a black-box strategy in this study for ML employment placement that welcomes interruption and does not rely on in-depth analytical effectiveness prediction. We incorporate deep reinforcement learning (DRL) into our scheduler architecture, motivated by the recent successes of DRL in video streaming [20], job planning [3], [24], [22], and Go [19]. We introduce Balance, an ML cluster planner powered by deep learning. In a neural network (NN) that translates basic clusters and task information (e.g., resources at hand, jobs' capacity requirements) to job placement choices (i.e., the server you want to put every employee on or the variable server of an assignment onto), harmonization inherently encodes load disturbance.

Utilizing the advantages of both the MPI and OpenMP models for parallel program execution on clusters can be done in two ways. One method distributes jobs among cluster nodes using MPI on top of OpenMP, and then distributes the work further within each node using OpenMP. In the second method, MPI is used by OpenMP to create a distributed shared memory (DSM) that spans the entire cluster [12]. The key drawback of the second technique is the difficulty and cost required for operating DSM in large-scale arrangements, despite its appeal due to OpenMP's programming simplicity. A novel, MOSIX-like [23, 8] method is presented by MGP. It circumvents the issues related to DSM by running the CPU portion of the program on a single node and the GPU kernel on hardware that is shared by the entire cluster.

Increasing the clock speed is a common way to update an HPC system's architecture. This strategy will be

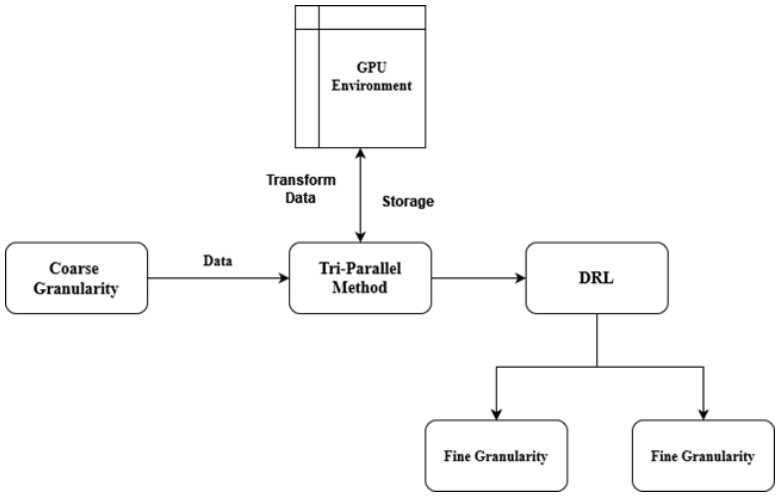

Fig. 3.1: Architecture Diagram of Proposed Method

fixed at 1 GHz due to exceptional dissipation of heat, and a different strategy to boost the number of cores will be used [11]. We are unable to add additional cores greater than 100 million in Exascale systems in accordance with the previously mentioned limitations. In the end, more cores can achieve the level of performance that is needed, but at the cost of extremely high-power consumption. "Massive parallelism" is an alternative approach that necessitated bettering the environment for coding. The efficiency of multi-level parallelism in the tri-hierarchy paradigm can be encouraging for Exascale computing systems, claims the author [13].

3. Proposed Methodology. The suggested tri-hybrid parallel programming model for Exascale computing systems, based on DRL, has been given in the next section. The suggested method, known as MOC, is a combination of MPI, OpenMP, and CUDA and is based on the hierarchical navigation of earlier parallel code methods. Three main levels of computation are present in MOC: intra-node, inter-node, and enhanced GPU devices. Figure 3.1 shows the specific procedure for each of these parallel computing levels.

3.1. Computation of Inter-Node. The targeted system's construction, host CPU core count, the number of shelves (if the system is a larger cluster), total number of nodes, type of GPUs (for accelerated computing), memory kind and stages, and other details must be determined before engaging with the MOC model. Parallel computing zones were initiated upon the determination of these specifications. Fundamentally, MOC offers three layers of parallel zones, with inter-node computation providing the first and top levels. By enabling communication between host CPUS units in every linked node, MPI was able to accomplish inter-node computing. Within MPI, there are two distinct categories of processes: master and slave. The former is denoted by a rank of '0,' while the latter is denoted by a rank that is not zero.

To specify each rank and transmission size across the MPI universe, a few basic MPI assertions must come before distributing data over processes. MPI master processes continue the parallel computation by using slave processes to spread the data among all linked nodes. There are other methods to send and receive the data. We developed the blocking methods MPI_Send() and MPI_Recv() for transferring and receiving information for the MOC model. While blocking techniques like Isend() and Irec() are more efficient than non-blocking ones, they nevertheless preserve synchronization. Although we did not employ any optimization during the data distribution process in our solution, this kind of parallelism only offers coarse-grain parallel. The following parallel processing zone began because of data being mistrusted over CPU processes.

3.2. Computation of Intra-Node. The processing of dispersed data across host CPU cores occurs inside the node during intra-node computing, which is the second degree of parallelism. There are multiple CPU processes used for the calculation. Several parallel programming models can be used to parallelize these threads of code. OpenMP is among the most well-known models for parallel programming that parallelizes CPU

threads. As was previously mentioned, GPU devices and CPU cores can both be programmed via OpenMP. We accomplished fine-grained parallelism in the MOC implementation by programmatically parallelizing CPU threads using OpenMP. There is only one primary outer pragma in the OpenMP coding model, which starts with the parallel zone.

3.3. Computation of Accelerated GPU. The data analysis over accelerated GPU devices was used to carry out the third level of parallel in the MOC paradigm. Every GPU device had a reserved CPU process. As a result, a looping statement transfers information from the host to the GPU device and reserves a certain GPU device each time. This data is subsequently processed using the CUDA kernel, which runs the program on a particular GPU. At this point, data is calculated in parallel across hundreds of cores to produce finer resolution. It is challenging to write the kernels every time in a cluster system with more GPU devices. Nonetheless, the MOC model included a generic CUDA kernel that executes in accordance with the template format and receives and returns data in that format.

Following the completion of data processing on GPU gadgets, the data is sent back over host cores and is managed by OpenMP processes from the original source. In a similar vein, OpenMP finishes running inside the pragma and sends data back to MPI slave operations. The MPI master thread gathers data from slave threads after obtaining input from all these levels and relays the findings back to the person making the call. We can attain three levels of parallel from the MOC model in this way.

An algorithm's usefulness can be determined by analyzing its computing and transmission costs. Any method's execution time is typically influenced by several variables, including the input data, the bit system (32/64 bits), the single/multiprocessor system, and the read/write speed to memory. In theory, the computational and space complexity of an algorithm is determined to evaluate. System memory types have an impact on space complexity. Modern memory devices solve the space constraints and, as a result, do not take the complexity of space into account.

Every parallel method has some overhead for communication while it is being processed. We attempted to minimize the number of interactions rounds in MOC implantation, which consisted of communicating, computation, and getting, and we assumed that the cost of overhead would be To. Let us presume that the process pi from the working region will send s bytes of data during the sending round. For transmitting s bytes, the communication overhead will consequently be O(N Sp). In a similar vein, multithreaded programs can use shared storage to calculate C bytes of data. There are numerous overhead opportunities during data processing across processes, including waiting times for shared information access and procedure timing, among others.

The MOC algorithm's overall time complexity can be summed up as (Tm = Tc + To), where Tc is the input data calculation cost and To is the overhead associated with communication cost.

$$T_c = O(\frac{N}{pT_t}) \tag{3.1}$$

3.4. Deep Reinforcement Learning (DRL). The DRL NN generates choices regarding placement for each new task in the set based on inputs such as different work sets, current assignment, and cluster resource availability. To gain incentive for DRL training, we calculate reward using the reward model. We can efficiently increase the size of the trace set that is accessible and produce enough samples for DRL offline instruction by using the reward prediction model.

3.4.1. State Space. The series s = (s1,..., sN) is the input state of the DRL NN. The number of simultaneous jobs running at any given moment is the sequence's width, N. The purpose of including current employment that has already been determined is to enable the DRL models to learn about possible conflicts among fresh positions and existing jobs on servers that are shared. The concurrent jobs include both recently arrived jobs and incomplete jobs that were submitted previously.

3.4.2. Action Space. The DRL agent chooses an action (a) based on a policy (s, a) that is a probability distribution over the action space after receiving s. An NN generates the policy, with $\pi\theta$ representing the parameters within the NN. The placement of all jobs can then be produced by a single inference, which naturally includes all feasible placement decisions of all new jobs in a scheduling interval (keep in mind that we do not adjust the placement of existing jobs). However, this results in an action space that is exponentially

large because there are an enormous number of placement combinations for all workers and parameter servers in all jobs. Large action spaces can result in longer training times and less satisfactory outcomes.

To accelerate policy learning, we assign placements to recently arriving jobs one at a time, creating an order that includes an employment decision for every new job. We reduce the complexity of the action the definition, and the 2M actions in our action space. New work on server m, where m ∈ [1, M]; (ii) (1, m), where a single parameter server of the new task is placed on server m, where m ∈ [1, M]. We individually reset the chance of (ii) to a value of zero and rescale all non-zero chances so that their total remains equal to one to accommodate the circumstance of employing the all-reduce design.

3.4.3. Reward. By teaching the approach NN to become more efficient at using resources and less prone to inter-job interference, we hope to minimize the average job completion time. Though it only exists when a project is completed, which could be many scheduling periods afterwards, job completion time seems like a natural incentive to watch. Since the postponed incentive offers little assistance in improving the early selections, the training community finds it unsatisfactory that the prize has a considerable feedback lag. Furthermore, future job deployments (which can interfere with this job by deploying on the same servers) determine a job's completion time in addition to the work placement condition at that moment.

$$r = \sum_{m \in [N]} \frac{C_n}{E_n} \tag{3.2}$$

The total of all concurrent jobs' standardized training speeds within a single scheduling interval determines the reward (r) that is noticed when action (a) is taken under state (s).

3.4.4. NN model. Before being linked to the representation system for encoding, each job's and server's state is first embedded in a fully connected layer (the Job/Server Embedding block). The NN may extract features as pre-processing from each job or server by integrating. When each entry in the input sequence is similar, pre-processing can also help the input sequence stand out more. One by one, the pre-processed states of running jobs are sent into the representation network, which learns in a manner akin to sequence learning. An end-to-end training process will be employed for the representation network and decoder network.

3.4.5. Representation Network. Once characteristics are extracted, the visualization networks create an image (a smaller vector) that is used by the network of decoders to make scheduling decisions. The representation network receives as input the state of each concurrent job and server state at each scheduled period. The primary difficulty is the fact that the quantity of ongoing tasks is uncertain in advance and subject to fluctuations. Nonetheless, a fixed-size input is necessary for many neural network structures, including feed-forward NN. Setting a maximum limit on the number of concurrent tasks and using buffering in the input sequence—that is, marking an entry as 0 if the job in that entry does—are simple ways to use feed-forward NN to handle input of non-fixed size.

If the real number of simultaneous jobs is less than the upper bound that has been predetermined, then this will function. Nevertheless, when the total number of simultaneous jobs is significantly less than the upper constraint on the pre-defined job quantity, zero-padding results in a large amount of duplicate data within the state of the input. In a similar vein, we must eliminate some jobs if the actual number of concurrent jobs exceeds the upper-bound that has been predetermined, which results in a loss of input data. We use the encoder portion of Inverter to encode the task and server data into a series of fix-sized matrices in order to allow decoding of any length of input. The attention model then aids in capturing the correlation between the various jobs in the order of inputs.

3.4.6. Decoder Network. The representation network's encoded sequence is analyzed by the decoder network, which then generates a placement choice for each freshly arrived job individually. The decoder receives the produced distribution for the placement of other concurrent jobs as input, and it applies an attention operation to handle the influence of the placement choices made by other simultaneous tasks. The decoder can obtain broad data by employing the attention process, instead of relying just on a single job placement decision for inference. The output of the model network and the decoder's inputs processed by attention are

then combined to create a decoder with a few hidden layers that are fully linked and the ReLU function for activation.

The last result of the layer generates a series of judgments for each unscheduled job individually using the softmax function as the activation function. To honor server resources abilities, we mask incorrect activities in the output layer of the NN by setting the likelihood of them to 0 in the policy distribution. These invalid actions involve deploying a worker or parameters server on a server that lacks the resources necessary to run it. Next, we adjust the odds for each decision to make sure the total remains at 1.

3.4.7. Design and Discussion. DRL models typically use neural networks as their core architecture. For different tasks, various architectures like Convolutional Neural Networks (CNNs) for spatial data or Recurrent Neural Networks (RNNs) for sequential data are employed. The model consists of an agent interacting with an environment. The agent receives states from the environment, takes actions, and receives rewards. The goal is to learn a policy that maximizes the cumulative reward. In a heterogeneous computing environment, the model may be designed to optimize resource allocation, task scheduling, or load balancing. This involves tailoring the state, action, and reward definitions to suit these specific computational tasks.

3.4.8. Training of DRL Models. The agent learns by interacting with the environment. This can be a simulated environment or real-world data, depending on the task. Algorithms like Q-Learning, Deep Q-Networks (DQN), or Policy Gradient methods are used. These algorithms help the agent learn from experiences (state, action, reward sequences) by updating the neural network weights. In heterogeneous environments, training can be parallelized across different hardware units like CPUs, GPUs, and TPUs. This accelerates the learning process and allows the model to handle complex, high-dimensional environments. The model is trained to explore the environment to learn new strategies and to exploit known strategies to maximize rewards. Balancing these two aspects is crucial for effective learning.

3.4.9. Integration into Parallel Computation in Heterogeneous Environments. Once trained, the DRL model is deployed in the heterogeneous environment. This involves integrating the model with various computing units like CPUs, GPUs, and specialized accelerators. The DRL model can dynamically allocate computational tasks to different processors based on their capabilities and current load, optimizing the overall performance. The model can predict the most efficient ways to schedule tasks and balance loads across the different processors, considering factors like computational intensity, memory requirements, and data dependencies. In a real-world environment, the DRL model continues to learn and adapt. It can adjust its strategies based on performance feedback and changing conditions in the computing environment. Key considerations include ensuring that the DRL model scales effectively with the size and complexity of the environment and maintains robust performance under various operational conditions.

3.5. Parallelization Methods in DRL. This is the most common form of parallelism where training data is distributed across different nodes. Each node processes a subset of the data and updates a local copy of the model. After processing, these updates are aggregated to update the global model. This combines data and model parallelism. Some layers of the neural network might be parallelized across different nodes (model parallelism), while the data fed into these layers is distributed across nodes (data parallelism).

3.6. Handling Synchronization Issues. Nodes update the model autonomously without waiting for others. This can speed up training but might leading to stale gradients and slow merging. All nodes harmonize their updates, ensuring that the model is always current. This can avoid issues like stale gradients but might reduce speed of the training process. In asynchronous methods, techniques like stale synchronous parallel (SSP) can be used. SSP permits a degree of asynchrony but limits the maximum allowed staleness of gradients. Regular barriers are created to save the state of the model. This is crucial to improve from node failures without losing important progress. Dynamic load rebalancing can be executed to adjust the load among nodes during runtime, reliant on their current load and performance.

The workload is split in a way that each computing unit (CPU, GPU, etc.) operates at optimal capacity without being overburdened. The splitting logic considers the specific capabilities of each processor. For example, GPUs are more efficient for parallelizable tasks like matrix operations, while CPUs handle sequential

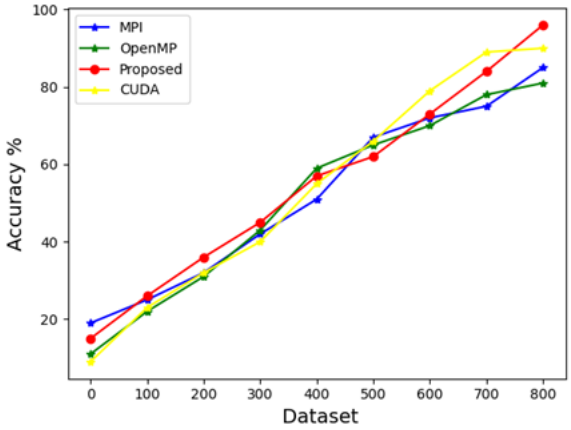

Fig. 4.1: Accuracy

tasks better. The model or data is split in a way that minimizes the need for communication between nodes, as this can be a major bottleneck in parallel computing.

In distributed systems, MPI is often used for communication between nodes. It allows efficient data transfer and synchronization across different computing nodes. In data parallelism, gradients computed on each node are shared and aggregated. Techniques like All-Reduce can be used for efficient gradient aggregation. In this model, a central server is responsible for maintaining the global model. The nodes compute gradients and send them to the parameter server, which updates the model and sends it back to the nodes.

4. Result Analysis. Six GPU servers are assembled into a testbed and linked via a Dell Networking Z9100-ON 100GbE switch. One 480GB SSD, one 4TB HDD, two GTX 1080Ti GPUs, 48GB RAM, one MCX413A-GCAT 50GbE NIC, and an 8-core Intel E5-1660 CPU are all included in each server. Kubernetes 1.7 is set up as the cluster management.

The proposed method evaluates the parameter metrics such as accuracy, scheduling interval, error rate and energy efficiency.

One of the most important evaluation metrics for evaluating a classification model's overall effectiveness is its accuracy. In relation to the overall number of occurrences in the data set, it indicates the proportion of correctly forecast instances (including true positives and true negatives).

$$Accuracy = \frac{Total\ number\ of\ truly\ predicted\ samples}{Total\ Samples} \tag{4.1}$$

Accuracy is an indicator that's frequently employed in mathematical modeling and algorithms evaluation to assess how well an estimate extends to new, unknown information. In figure 4.1 shows the Accuracy of proposed method. The proposed method achieves better accuracy compared with other parallel methods.

An assessment measure used to gauge the categorization model's overall accuracy is the error rate, sometimes referred to as the misclassification rate. It shows the percentage of cases in the information set that were erroneously classified. The ratio of the overall amount of misunderstandings (the sum of the false positives and false negatives) to the total number of instances in the dataset is used to compute the error rate.

$$Error\ Rate = \frac{Number\ of\ Misclassifications}{Total\ Number\ of\ Instances} \tag{4.2}$$

In figure 4.3 shows the evaluation of error rate. The proposed method achieves minimum error rate compared with the existing methods.

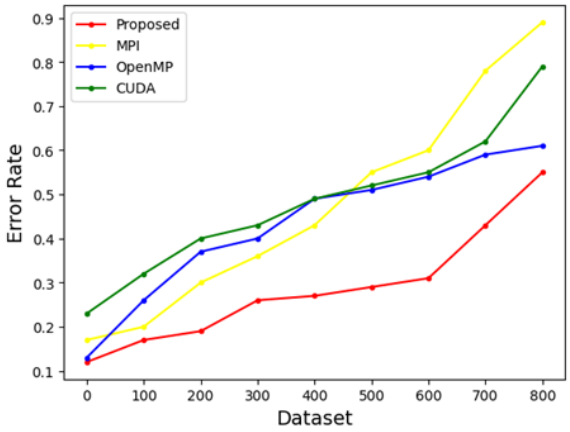

Fig. 4.2: Evaluation of Error Rate

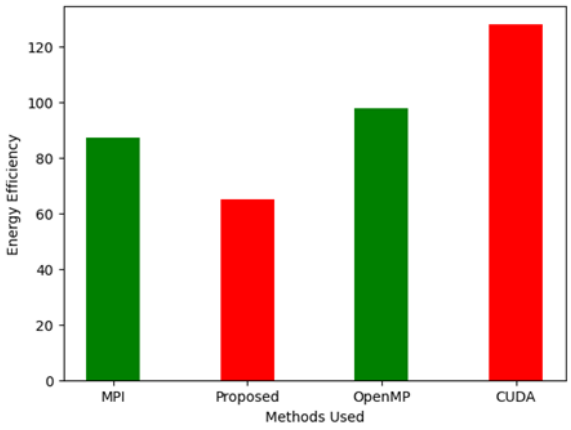

Fig. 4.3: Energy Efficiency

Energy efficiency refers to the ratio of useful energy output to the energy input in a specific system or process. It measures the effectiveness with which an entity, like a machine, system, or process, utilizes energy to perform a specific function or achieve a desired outcome. Enhancing energy efficiency is a key objective in various sectors, including industrial manufacturing, transportation, building construction, and information technology. Improved energy efficiency leads to reduced energy usage, lower operating costs, and supports sustainable development goals. In figure 4.3 shows the evaluation of Energy efficiency. The proposed method achieves less energy efficiency compared with the existing methods.

The interval of time between successive scheduled events or activities in a system or procedure is called a planning gap. It is an essential factor in many fields, like manufacturing, project management, computer networks, and communication systems. The requirements and attributes of the system or process under consideration must be considered while selecting an appropriate scheduling interval. In figure 4.4 shows the scheduling interval between data transmission. The proposed method takes less scheduling intervals compared with existing methods.

5. Conclusion. As computer demands in the research and industrial sectors continue to rise, an increasing variety of scalable and distributed techniques are needed to accurately mimic complex events. This paper suggests a novel strategy to address this challenge: creating and gaining access to mathematically modeled

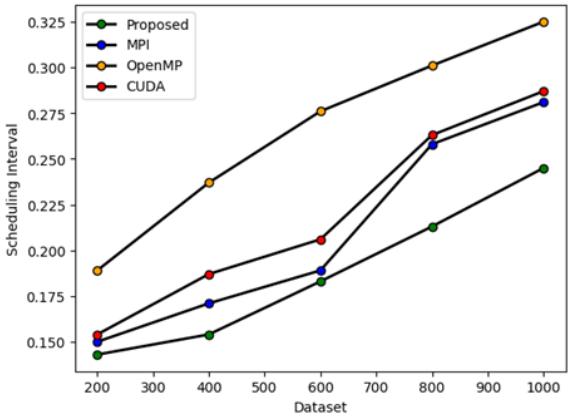

Fig. 4.4: Scheduling Interval

methodologies in heterogeneous computer clusters. The recommended methodology evaluates heterogeneous computing clusters using a parallel computational model based on DRL. The algorithms accommodate the range of topologies found in modern computing clusters by distributing the processing load among multiple nodes using parallelization techniques. The architecture aims to minimize resource usage and maximize speed by utilizing heterogeneous hardware components like GPUs, CPUs, and acceleration devices. A thorough performance assessment is carried out to determine the efficacy of the suggested strategy. Scalability study, benchmarking, and comparisons with conventional homogeneous computing configurations are all included in the review. The study investigates how different algorithm design decisions affect the speed and efficiency attained in various computing settings.

REFERENCES

[1] M. Á. Abella-González, P. Carollo-Fernández, L.-N. Pouchet, F. Rastello, and G. Rodríguez, *Polybench/python: benchmarking python environments with polyhedral optimizations*, in Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction, 2021, pp. 59–70.

[2] Y. Bao, Y. Peng, and C. Wu, *Deep learning-based job placement in distributed machine learning clusters with heterogeneous workloads*, IEEE/ACM Transactions on Networking, 31 (2022), pp. 634–647.

[3] Y. Bao, Y. Peng, C. Wu, and Z. Li, *Online job scheduling in distributed machine learning clusters*, in IEEE INFOCOM 2018-IEEE Conference on Computer Communications, IEEE, 2018, pp. 495–503.

[4] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, *Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning*, in Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–16.

[5] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, *Elastic parameter server load distribution in deep learning clusters*, in Proceedings of the 11th ACM Symposium on Cloud Computing, 2020, pp. 507–521.

[6] Y. Gong, B. Li, B. Liang, and Z. Zhan, *Chic: experience-driven scheduling in machine learning clusters*, in Proceedings of the International Symposium on Quality of Service, 2019, pp. 1–10.

[7] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, *Tiresias: A {GPU} cluster manager for distributed deep learning*, in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 2019, pp. 485–500.

[8] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, *A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning*, in 2017 IEEE 37th international conference on distributed computing systems (ICDCS), IEEE, 2017, pp. 372–382.

[9] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, *Themis: Fair and efficient {GPU} cluster scheduling*, in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 289–304.

[10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, *Learning scheduling algorithms for data processing clusters*, in Proceedings of the ACM special interest group on data communication, 2019, pp. 270–288.

[11] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, *A hierarchical model for device placement*, in International Conference on Learning Representations, 2018.

[12] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, *Device placement optimization with reinforcement learning*, in International Conference on Machine Learning, PMLR, 2017, pp. 2430–2439.

[13] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al., *Ray: A distributed framework for emerging {AI} applications*, in 13th USENIX symposium on operating systems design and implementation (OSDI 18), 2018, pp. 561–577.

[14] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, {*Heterogeneity-Aware*} *cluster scheduling policies for deep learning workloads*, in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 481–498.

[15] A. Or, H. Zhang, and M. Freedman, *Resource elasticity in distributed deep learning*, Proceedings of Machine Learning and Systems, 2 (2020), pp. 400–411.

[16] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, *Dl2: A deep learning-driven scheduler for deep learning clusters*, IEEE Transactions on Parallel and Distributed Systems, 32 (2021), pp. 1947–1960.

[17] J. Shirako and V. Sarkar, *Integrating data layout transformations with the polyhedral model*, in Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT'19), D. Wonnacott and O. Zinenko (Eds.). Valencia, Spain. http://impact. gforge. inria. fr/impact2019/papers/IMPACT_2019_paper_8. pdf, 2019.

[18] ———, *An affine scheduling framework for integrating data layout and loop transformations*, in International Workshop on Languages and Compilers for Parallel Computing, Springer, 2020, pp. 3–19.

[19] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan, *Towards distributed machine learning in shared clusters: A dynamically-partitioned approach*, in 2017 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE, 2017, pp. 1–6.

[20] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al., *Starcraft ii: A new challenge for reinforcement learning*, arXiv preprint arXiv:1708.04782, (2017).

[21] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica, *Lineage stash: fault tolerance off the critical path*, in Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 338–352.

[22] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al., *Gandiva: Introspective cluster scheduling for deep learning*, in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 595–610.

[23] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, *Experience-driven networking: A deep reinforcement learning based approach*, in IEEE INFOCOM 2018-IEEE conference on computer communications, IEEE, 2018, pp. 1871–1879.

[24] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, *Slaq: quality-driven scheduling for distributed machine learning*, in Proceedings of the 2017 Symposium on Cloud Computing, 2017, pp. 390–404.

[25] T. Zhou, J. Shirako, A. Jain, S. Srikanth, T. M. Conte, R. Vuduc, and V. Sarkar, *Intrepydd: performance, productivity, and portability for data science application kernels*, in Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2020, pp. 65–83.

Edited by: Rajanikanth Aluvalu

Special issue on: Evolutionary Computing for AI-Driven Security and Privacy: Advancing the state-of-the-art applications

Received: Dec 7, 2023

Accepted: Dec 29, 2023