CS159-R: Parallel Processing and Machine Learning

Brian Qian
Computer Science Department
San Jose State University
San Jose, CA 95192
brian.qian@sjsu.edu

ABSTRACT
Parallel processing and machine learning make up two different, but overlapping, topics. We discuss the overlap between parallel processing and machine learning. We start with how parallel processing is used in machine learning. In particular, we look at select applications in data processing and model training. Moving to parallel processing introduces tradeoffs that merit judgment and are to be further discussed. Proceeding onwards is how machine learning can be used for parallel processing. We examine learning techniques for parallel processing issues.

1. INTRODUCTION
Parallel processing and machine learning are two entirely different topics, but they do overlap. We specifically look at this overlap in two contexts. The first is Parallel Processing in Machine Learning - how can parallel processing be utilized within the machine learning process. The second is Machine Learning in Parallel Processing - how could machine learning techniques be used for problems that arise in parallel processing.

2. PARALLEL PROCESSING IN MACHINE LEARNING
Broadly speaking, machine learning is using old data to analyze the new, incoming data. This is essentially the learning aspect. Machine learning models that seem to have "learned" more may have simply worked with more data or are just better than other models. In practical applications that are effective and accurate, it's not unusual to see them trained with large amounts of data. Parallel processing is often utilized to help handle these large amounts of data and the training involved.

The "machine learning process" mentioned earlier is somewhat vague. We'll define it as building a model. There are several steps to do so.

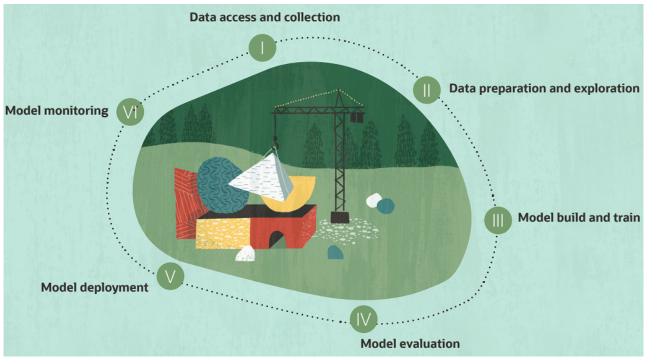


Figure 1. Steps to build a model [17].

Figure 1 depicts the steps to build a model from scratch. Not all models may strictly follow the exact same roadmap; not all models may follow exactly six steps. However, the steps generally apply to most models.

Two terms in the roadmap show up most frequently: "data" and "model". These two dominate the workload and are the focus. In particular, we will specifically focus on steps II and III as the bulk of parallel processing occurs here. Step II of "Data preparation and exploration" deals with data processing, cleansing, and analysis. Step III of "Model build and train" deals with building and training our model. We do not go into great depth on data processing, and we try to place our focus on the training step.

2.1 Data Preparation and Exploration
As aforementioned, machine learning deals with large volumes of data. That data needs to be processed, cleansed, and analyzed before it can be sent on to be used as training data for the model. With the amount of data to work through, this could pose a significant amount of work for sequential processing. Parallel processing has overhead, but the overall efficiency benefit from parallelization of the significant work may outweigh that overhead. Machine learning applications do use parallel processing for data processing. The exact specifics of data cleansing and formatting can vary based on use cases. We will only briefly introduce commonly used tools in machine learning such as MapReduce.

At a high-level overview, MapReduce is a technique to process large data sets by breaking them down into many, smaller pieces. Of course, MapReduce is not solely limited to machine learning; it is used for data processing in general, especially big data. Another prominent framework is Apache Spark with libraries that specifically target and support machine learning.

2.2 Model Build and Train
Without training, the large volume of data remains as just data. There must be a model to be trained for there to be machine learning. The model training should run through all data and is often parallelized to do so.

Besides large volumes of data, the models can grow in size as well. They can grow larger than a single processor can fit. In that case, the model is broken up into pieces. Each piece is given to a parallel processing unit.

2.2.1 Training Distribution
The training of the model is overall parallelized, but the nature of distributing that work can differ. There are two main, simple distributions: data parallelism and model parallelism.

2.2.1.1 Data Parallelism

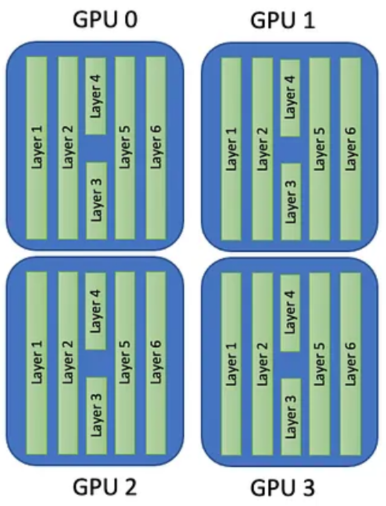


Figure 2. Data parallelism spread among GPUs [15]. The model is a deep neural network separated into layers, wholly stored within each GPU.

Data parallelism splits up the work by data. It will take the whole data set and split it among the parallel processing units. The model is individually trained in parallel on the partitioned data sets. As in Figure 2, the model is wholly stored in each processing unit. If the model is too large to fit on a single unit, then strictly applying data parallelism will not be effective [8].

One common issue arises from parallelization, and data parallelism is not exempt. Each processing unit trains its own copy of the model, but there is no overall model that combines the work that each unit has done. The overall model must be synchronized among the units. Otherwise, the training procedure will iteratively run on the same model without considering any calculations and updates from other processing units (i.e. all units will train independently). In the end, all units will then simply give out many independent copies of the model trained on smaller parts of the training data.

2.2.1.2 Data Parallelism Synchronization

There are two ways to do synchronization for data parallelism: synchronous and asynchronous. Both ways are valid, but each comes with its own benefits and drawbacks. In general, larger models use asynchronous. Comparatively smaller models such as between 32 to 50 nodes use synchronous [2].

Synchronous synchronization has two approaches: decentralized and centralized. We discuss both approaches because not one approach will always be the superior choice. It really depends on a variety of factors specific to your use case (e.g. network topology, bandwidth, communication latency, parameter update frequency, and desired fault tolerance) [2].

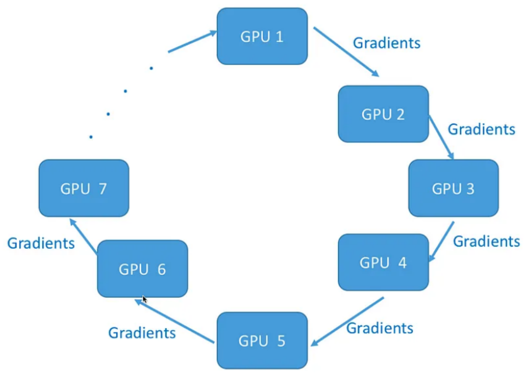


Figure 3. Decentralized approach to synchronous synchronization [16]. Gradients are the calculate updates to the model. GPUs communicate their updates to the next.

The decentralized approach has all units communicate with each other to update the model. Figure 3 specifically shows the Ring All-reduce method, creating the ring that facilitates synchronizing the model across all the GPUs. However, that ring causes an issue. Any iteration or cycle through the ring will be as quick as the slowest GPU. This is why it is “synchronous” as all the processing units must have completed their computations in sync. The decentralized part comes from how all these individual units communicate with each other.

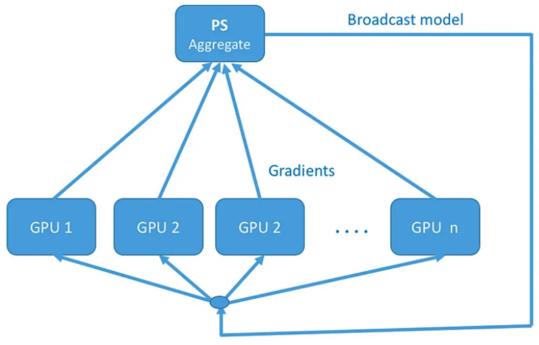


Figure 4. Centralized approach to synchronous synchronization [16]. GPUs communicate the computed gradients to a central aggregate PS (Parameter Server). The PS broadcasts the aggregated model to the individual GPUs.

Rather than communicate with each other, each unit communicates with a centralized aggregate. This is the primary difference between the decentralized approach and the centralized approach. However, the centralized approach equally suffers from the synchronous waiting. As in Figure 4, the PS can only finish aggregation and broadcast it after all GPUs have sent their computed gradients. It cannot do so before the slowest GPU has completed and communicated its computation.

No matter the approach, synchronous synchronization makes each unit to be in sync. It is possible for units to remain idle as they wait for other units to catch up. As the name “asynchronous” implies, the asynchronous approach eliminates this sync. The tradeoff is that individual copies of the model may not be the exact same at every point in time.

The asynchronous approach works similarly to the centralized synchronous approach. Each GPU finishes computing and sends the updates to a central aggregate. The PS updates the overall model it has stored. The difference is that the PS will then send back the model information it currently has to the finished GPU. That GPU will proceed to train the received model, eliminating a significant part of the waiting in the synchronous approach. The PS will not wait for all updates and then broadcast to all.

Zooming into a single iteration, each unit may not hold the same updated copy of the model. There might be slight inconsistencies amongst the trained models on each unit. Over time, these inconsistencies can amplify through the training iterations. It could overall negatively impact the convergence and stability of the model [1].

2.2.1.3 Model Parallelism

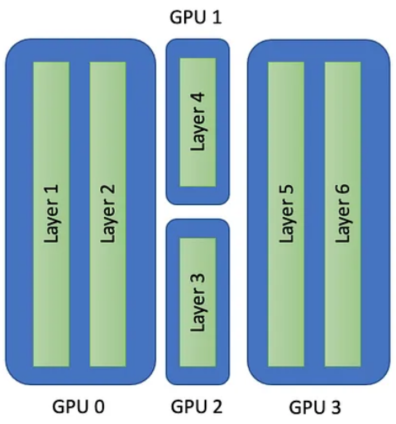


Figure 5. Model parallelism spread among GPUs [15]. The layers of the deep neural network model are distributed to the GPUs.

Model parallelism splits up work by the model. The whole model is split up into parts among the processing units. Each unit will hold a different part of the model, but they store the entire training data. Figure 5 depicts how the different parts of a model can be split up among parallel processing units.

Data parallelism by itself cannot handle large models that do not fit on a single processing unit. This calls for model parallelism to break down that model such that the pieces will fit. That said, data parallelism is preferred over model parallelism if possible [6]. Breaking down models is more complex than breaking down data sets. Models often have interdependencies between layers that complicate how those models should be broken down [2]. In addition, the "parallelism" in model parallelism is slightly different from that of data parallelism.

Model parallelism is really only effectively parallel when there are no layer interdependencies (i.e. the model can be broken up into parallel pieces). Otherwise, the training in model parallelism runs in a serial fashion. A layer with a dependency on a previous layer cannot begin until that previous one has completed. The processing unit holding the previous layer will communicate its updated model information to the next unit. In a way, it can be similar to a ripple in ripple carry adders because of a sequential dependency.

In Figure 5, the layers are abstracted, but take an example where Layer 5 is dependent on Layer 4. We cannot train Layer 5 without having done Layer 4. Layer 5 lies in GPU 3, and Layer 4 lies in GPU 1. GPU 3 must wait for GPU 1 to have completed.

Without knowing how the specific model is split and the possible dependencies, we cannot simply assume all parts can be run in parallel. Thus, we naively run iterations in a serial manner, sequentially propagating results through the processing units. There are two serial propagations to discuss: forward pass and backward pass.

Forward pass is simply getting the output from the model. The model has been spread throughout the processing units, but the outputs have to be calculated by going through the model. Thus, the forward pass traverses through the processing units to get the output and evaluate it. The backward pass comes after and calculates the according model updates (i.e. gradients) [3]. Without going too deep into the mathematics, it is "backwards" to recursively apply the chain rule [14]. We define a training iteration as starting and completing the forward pass and backward pass.

There is generally no mandatory, sequential order based on the GPUs themselves. In Figure 5, each GPU is numbered. Also, notice that layer 3 is in GPU 2 and layer 4 is in GPU 1. It is not mandatory that GPU 1 must run before GPU 2 (other than potential layer dependencies). Following the numbered layers, the order for forward pass in Figure 5 could be GPUs 0, 2, 1, 3. Backward pass is simply the reverse order of forward pass. The same freedom for forward pass applies to backward pass. The corresponding order for backward pass would be GPUs 3, 1, 2, 0.

In implementation, each processing unit only needs to know the next unit to communicate during forward and backward passes [7]. It is similar to the idea of a linked list. The training iteration can be accordingly thought of as traversing a linked list all the way through and back.

The propagation is sequential to handle dependencies, but the issue is that processing units are mostly idle in our serial implementation. For the linked list analogy, the traversal deems which "node" to work on currently. Every other node in the linked list remains idle, waiting for the traversal to get to it. Only one node is ever active at any time. Our nodes are the mostly-idle GPUs, and the training is the sequential traversal. An alternative solution is to introduce a pipeline.

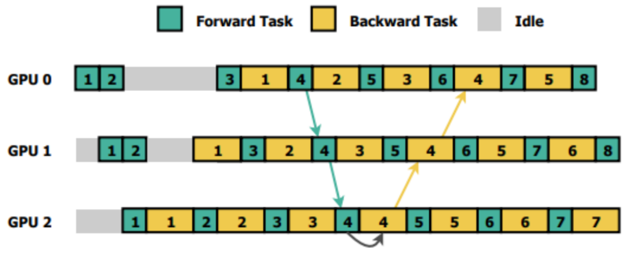


Figure 6. Pipelined model parallelism [3]. The numbers in the boxes represent a training iteration. An example flow for a single iteration is represented by the directed arrows.

The goal is to minimize the time that processing units are idle. In other words, we want to improve the throughput of training iterations. We can do so by overlapping iterations in a pipeline fashion as shown in Figure 6.

Each training iteration involves two tasks: a forward and a backward pass. This may somewhat complicate how to think about the pipeline, but the central idea is that the processing units all try to be working on different iterations at the same time. This is how we try to do the overlap of iterations. Notice how in Figure

6 each GPU would be simultaneously working on a different iteration at any time. Of course, typical of a pipeline, the first execution or iteration seems sequential. Figure 6 has idle time during the timeframe of that first iteration. The pipeline overlapping really enters after that first iteration has been completed. No GPU is idle at any time in Figure 6.

The perceived speedup in throughput is directly related to the number of processing units in the pipeline. Figure 6 utilizes three GPUs, and the speedup is by three. Look at the timeframe for iteration 4 bounded by the directed arrows in Figure 6. Sequential processing means that only one iteration would complete during this timeframe. The pipeline model completes iterations 2, 3, and 4 during this timeframe - a speedup in throughput by three. The reality is this is the ideal value. The figure depicts an ideal scenario where all iterations sync perfectly, but these rely on data arrival time and latency [2].

One issue to notice is that training results may be inconsistent and not fully accurate [3]. We are starting new training iterations before model updates from the previous ones have returned. Overall, this can result in unstable training and an inaccurate model [12]. This is one reason why data parallelism is preferred over model parallelism if possible. Data parallelism tends to become more accurate with more training, but model parallelism fluctuates [3].

2.2.1.4 Other Distributions

Data parallelism and model parallelism are not mutually exclusive. They can be used in conjunction. The well-known ChatGPT-3 uses a combined approach of data and model parallelism [7].

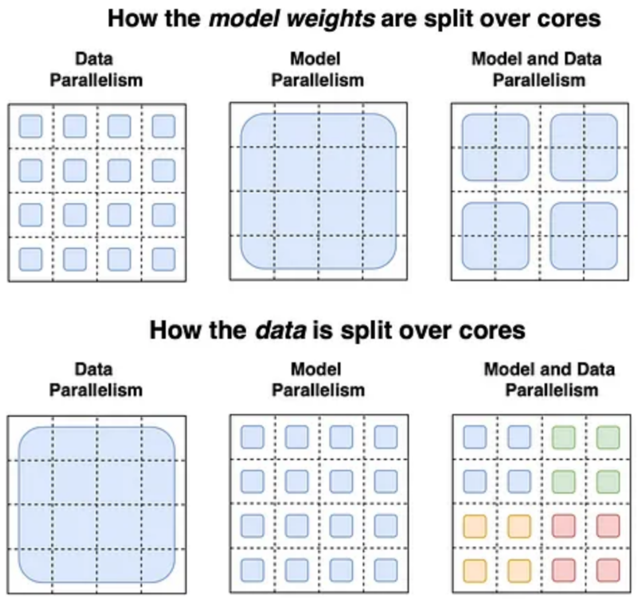

How the model weights are split over cores
Data Parallelism
Model Parallelism
Model and Data Parallelism
How the data is split over cores
Data Parallelism
Model Parallelism
Model and Data Parallelism

Figure 7. Distribution of model and data during data parallelism, model parallelism, and both at once. [5, 7]

In Figure 7, data parallelism and model parallelism are two extremes. Data parallelism shares the entire data among all cores, but the model structure is individually stored on every core. Model parallelism shares the entire model among all cores, but the data is individually stored. Combining both splits the model and data into smaller chunks, but not necessarily individual chunks. Each little chunk works on its part of the model with its part of the data.

Figure 7 shows each model chunk to span across four cores. Each of those model chunks works on a small, designated-color part of the data. Each core stores one-fourth part of the model chunk and the colored data chunk.

Of course, data parallelism and model parallelism are not the only distributions. Other types of distributions and variants exist. There have been proposals for combining attributes of these different distributions (refer to FlexFlow [8]). Nevertheless, data parallelism and model parallelism are simple and general, making them widely applicable to all model training.

2.2.2 Hyperparameter Tuning

Hyperparameters define the model structure and overall settings (e.g. size of the model such as how many layers or how many nodes). These are values that are set before the training begins. Hyperparameter tuning is similar to knobs [17]. Just like using knobs to adjust settings, you can tune the hyperparameters.

Importantly, these “knobs” are of the model, not by the model. Hyperparameters will not change during the training process; the ones that do change and are internally learned are called model parameters. Therefore, changing these hyperparameters can change the outcome of training.

As previously mentioned, machine learning models that feel they “learned” more or are better than other models might have simply been trained under more data. It is intuitive, but it is not the only explanation. Hyperparameter tuning can be one such explanation. For instance, traditional LSTM (Long Short-Term Memory) architectures are outperformed by more recent, “state-of-the-art” models. Carefully tuning hyperparameters on the LSTM architectures, they ended up outperforming those recent models [9, 11].

Hyperparameter tuning is really an optional step, but it’s encouraged and recommended to do so. As in the LSTM case, they optimize the results and performance of the model and improve accuracy.

However, these hyperparameters have to be carefully tuned, and there is no standard, clear indicator of what the best adjustments are. Thus, the adjustments are often done manually or by brute force. Finding the optimal adjustments is pretty computationally expensive, especially if by brute force [11].

2.2.2.1 Grid Search

The trivial, brute force method is to try all combinations of possible hyperparameter adjustments. This is known as grid search.

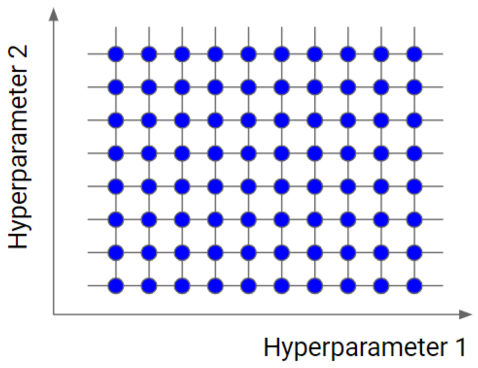


Figure 8. Illustration of grid search [10]. Each blue dot expressed as a Cartesian product represents a unique combination of hyperparameter values after tuning.

Limiting ourselves to two hyperparameters as in Figure 8 already gives a grid (or plane) to check. Generalizing for N hyperparameters we want to tune, the search space using grid search would span R^N. This becomes increasingly computationally expensive as N increases. Fortunately, this is easy to parallelize.

In Figure 8, all blue dots represent a unique combination. These combinations do not have dependencies on each other. They independently represent their own combination and performance metric.

As such, it is easily possible to assign points and areas among parallel processing units. Each unit maximizes the performance metric and picks out the locally optimal combination. These can be aggregated to find the most optimal combination.

Grid search and its parallelization are relatively simple, but there is one point to take note of. In most cases, the vast amount of points and combinations significantly dominate the parallel processing units. The overhead from parallelism will not overshadow the efficiency gain from splitting up such a large search space. It is not as clear when the units closely match the number of combinations. Take an extreme example where both are equal; the parallelism will be slower than sequential processing. Here, it is possible for each unit to be given just a single combination. The maximum among a single data point is itself; each unit will respond back with the same combination it was sent and call it the maximum. This makes the search identical to sequential processing, but we have added overhead from parallelism. Let's assume that other internal operations take zero time since it is likely that they can be optimized to squeeze out more efficiency. At the very least, processing units have to be able to communicate with an aggregate in some way. The aggregate search is already akin to the sequential process, but we have added overhead. The parallel processing will be slower in this extreme case. The specific cutoff or balance is not that clear; it can depend on the implementation and parallel capabilities.

2.2.2.2 Others

Besides parallelism, there are other optimizations and algorithms building off of grid search. After all, it is an exhaustive approach that will check many non-optimal combinations. One such optimization is called Random Search which will randomly select a subset of points to check, hoping that the subset will exclude many non-optimal points. Though, the parallelization process is identical to that of grid search already discussed. Parallel implementations for hyperparameter tuning default to grid search or random search [11].

3. MACHINE LEARNING IN PARALLEL PROCESSING

We move away from looking at how parallel processing is used in the machine learning process. We now examine the overlap of parallel processing and machine learning in the reverse context: machine learning utilized within parallel processing. Compared to the former, this is definitely not as conventionally used and as well-known. Nevertheless, it may help round out our insight into the overlap between the two different topics of machine learning and parallel processing.

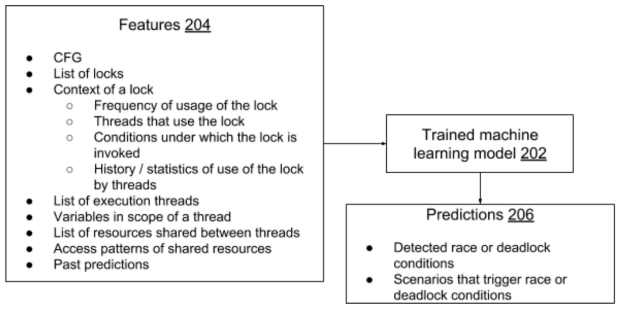
3.1 Detecting Deadlocks and Race Conditions

Parallel processing has two particularly nasty bugs: deadlocks and race conditions. These are non-deterministic bugs and are difficult to debug. Deadlocks arise in resource sharing. Race conditions arise in shared, global data. Both of these are controlled by how the underlying operating system does the scheduling. Executions will not abide by the same schedule every time, enabling the non-deterministic bugs.

One way to detect deadlocks and race conditions is to use a control flow graph. As the name suggests, it is a directed graph that represents the control flow of the execution. Nodes represent the code instructions. Directed edges represent the next instruction to execute. It should cover all possible executions such as when there are branches or jumps. We are able to see acquired locks and resources as well [12]. However, notice that "all possible executions" do not take into account the non-deterministic scheduler. This is because the scheduling sequence is implicitly modeled by having a control flow graph; it just depends on how you traverse the graph. For example, round-robin scheduling would be similar to traversing the graph by breadth-first search. Breadth-first search iteratively traverses by level, looking at one node in each branch (or instruction) at a time. This is a similar concept to round-robin scheduling of offering everyone an equal share of time.

We detect deadlocks and race conditions through depth-first search. In terms of scheduling execution, depth-first search would be similar to sequential processing. This is because all previously started processes and threads must complete before starting the next. We use it for detection as well because we will be able to identify potential conflicts with other processes and threads at any point in time of their execution. In doing so, we can identify the bugs in terms of graph properties. As we traverse by depth-first search, we identify intersections between branching processes or threads. These intersections represent potential race conditions [12]. Deadlocks may not be as simple to identify. They are indicated by cycles, but not all cycles will indicate deadlocks. For or while loops are modeled as a cycle, but they definitely do not deadlock every single time. We look at the cycles where different execution paths converge on a shared resource [12].

Given a larger, practical program, the control flow graph will accordingly grow larger. Traversing that graph with depth-first search becomes more computationally intensive and expensive for time and space, especially considering extra operations for the detection. In addition, there may be a concern for the stack given many recursive calls along the deep control flow graph. We can use machine learning techniques to prune the heavy search space [12].



Features 204
CFG
List of locks
Context of a lock
Frequency of usage of the lock
Threads that use the lock
Conditions under which the lock is invoked
History / statistics of use of the lock by threads
List of execution threads
Variables in scope of a thread
List of resources shared between threads
Access patterns of shared resources
Past predictions
Trained machine learning model 202
Predictions 206
Detected race or deadlock conditions
Scenarios that trigger race or deadlock conditions

Figure 9. Flow of using a machine learning model on a control flow graph to detect deadlocks or race conditions [12].

It may be possible for a machine learning model to be trained on situations of deadlocks or race conditions. In doing so, we could input features of the control flow graph into the model during the traversal [12]. We can pass a variety of features as illustrated in Figure 9. The model then gives a prediction on potential deadlocks or race conditions [12]. If the model determines that the execution path might not have these bugs, we may prune it and move on. This way, the whole graph will not have to be traversed.

For training the model, supervised learning can be used. The model is provided with a set of input features and their corresponding optimal predictions [12]. We may even use reinforcement learning to use past predictions as feedback to improve performance [12]. There are further optimizations and algorithms for a better probabilistic model using a Bayesian network [4]. We focus on control flow graphs, but we can really apply it to graphs that formally define or model our systems [4].

Control flow graphs provide a static view of executions and their underlying bugs, but they have to be heavily analyzed to find those bugs. One method that may be more empirical is by using resource footprints. The bugs are non-deterministic because of the non-deterministic scheduler, but they can exhibit a pattern. This is particularly true for deadlocks; when a deadlock appears, it is very apparent. Deadlocks represent competition for shared resources that ended up in up in a waiting cycle. We usually see the application freeze or crash, but deadlocked executions will show abnormal resource and CPU usage [13]. For example, one of the first things to usually notice when opening Task Manager after a suspected deadlock situation during a freeze could probably be high CPU usage. Theoretically, we may be able to train a model with these usage statistics from normal and deadlocked executions [13]. It could help to detect deadlocks in this manner.

4. CONCLUSION

We have looked at the overlap between parallel processing and machine learning. Of course, we can only discuss a subset of information and knowledge from the vast overlap. However, we do cover main, central concepts.

The most immediate thought for the use of parallel processing is likely efficiency. It is reasonable and is especially correct when considering machine learning. There is a significant amount of data that needs to be processed before the training step can begin. The training step itself is computationally expensive as well, taking up time and resources. We have seen how work could be split up in a parallel fashion. One of the main areas of overhead is communication. The work is parallelized, but it eventually has to be aggregated and communicated. Using parallel processing generally adds overhead, but it is overall a net gain here. These applications deal with such a great amount of operations and data that the efficiency of going parallel outweighs the overhead. That said, we do not stop and say that this is enough. In data parallelism. the overhead lies in the synchronization. There are synchronous and asynchronous approaches to it. The synchronous approach keeps the overhead as is, whereas the asynchronous approach tries to minimize it as much as possible. Though, the asynchronous approach can cost overall accuracy and consistency. These do not undermine the benefit of parallel processing as it remains a net gain, but these are important considerations to at least be aware of.

Using parallel processing in machine learning is not only for efficiency but also space. The main usage of model parallelism is to be able to even fit large models. It is not necessarily for speed. To reiterate, model parallelism is not preferred over data parallelism. The need for serial propagation in model parallelism takes away its parallel charms of speed.

While many of these details are hidden and abstracted behind libraries and tools, it is good to know about them. For example, data parallelism can have a synchronous or asynchronous approach. Then, synchronous can be split into centralized and decentralized. Each of these have their own advantages and disadvantages, and it is ultimately up to the user to look at their use case and determine the best tool.

Machine learning, in a way, can be thought similar to analyzing and recognizing patterns. Parallel processing may be non-deterministic, but it is not completely devoid of any patterns. We may use these patterns to understand and utilize parallel processing more effectively, especially in resolving its bugs. In any case, looking at the overlap in these two different contexts reveal intriguing perspectives and knowledge.

5. REFERENCES

[1] Addair, T. (n.d.). Decentralized and distributed machine learning ... - stanford university. Decentralized and Distributed Machine Learning Model Training with Actors. https://www.scs.stanford.edu/17au-cs244b/labs/projects/addair.pdf

[2] Ben-Nun, T., & Hoefler, T. (2018, September 15). Demystifying Parallel and Distributed Deep Learning: An in-depth concurrency analysis. arXiv.org. https://arxiv.org/abs/1802.09941

[3] Chen, C.-C., Yang, C.-L., & Cheng, H.-Y. (2019, October 28). Efficient and robust parallel DNN training through model parallelism on Multi-GPU platform. arXiv.org. https://arxiv.org/abs/1809.02839

[4] Einollah, P., Vahid, R., & Nikanjam, A. (2017, June 1). Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm. Journal of Systems and Software. https://www.sciencedirect.com/science/article/abs/pii/S0164121217301061

[5] Fedus, W., Zoph, B., & Shazeer, N. (2022, June 16). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv.org. https://arxiv.org/abs/2101.03961

[6] Hegde, V., & Usmani, S. (n.d.). Parallel and distributed deep learning - stanford university. Parallel and Distributed Deep Learning. https://web.stanford.edu/~rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf

[7] Hu, L. (2022, October 14). Distributed parallel training: Data parallelism and model parallelism. Medium. https://towardsdatascience.com/distributed-parallel-training-data-parallelism-and-model-parallelism-ec2d234e3214

[8] Jia, Z., Zaharia, M., & Aiken, A. (2018, July 14). Beyond Data and model parallelism for Deep Neural Networks. arXiv.org. https://arxiv.org/abs/1807.05358

[9] Li, L. (2018, December 19). Massively parallel hyperparameter optimization. Machine Learning Blog | MLD CMU | Carnegie Mellon University. https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/

[10] Malato, G. (2022, May 27). Hyperparameter tuning. grid search and Random Search. Your Data Teacher. https://www.yourdatateacher.com/2021/05/19/hyperparameter-tuning-grid-search-and-random-search/

[11] Melis, G., Dyer, C., & Blunsom, P. (2017, November 20). On the state of the art of Evaluation in neural language models. arXiv.org. https://arxiv.org/abs/1707.05589

[12] Shankar, K. R. (2018, August 14). Detection of deadlocks and race conditions in Computing Systems. Technical Disclosure Commons. https://www.tdcommons.org/dpubs_series/1407/

[13] Sherpa, S., Vicenciodelmoral, A., & Zhao, X. (2019, December 1). Deadlock Detection for Concurrent Programs Using Resource Footprints. ACM Digital Library. https://dl.acm.org/doi/10.1145/3368235.3369370

[14] Stanford University. (n.d.). CS231N Convolutional Neural Networks for Visual Recognition. CS231N convolutional neural networks for visual recognition. https://cs231n.github.io/optimization-2/

[15] Torres, J. (2021a, February 8). Deep Learning on Supercomputers. Medium. https://towardsdatascience.com/deep-learning-on-supercomputers-96319056c61f

[16] Torres, J. (2021b, June 3). Scalable deep learning on parallel and distributed infrastructures. Medium. https://towardsdatascience.com/scalable-deep-learning-on-parallel-and-distributed-infrastructures-e5fb4a956bef

[17] Yip, W. (n.d.). Lifecycle of machine learning models - oracle. Lifecycle of machine learning models. https://www.oracle.com/in/a/ocom/docs/data-science-lifecycle-ebook.pdf