# First Experiments to Evaluate the Relevance of Task-based Runtime Systems to Implement Large Language Model Applications

Lionel Eyraud-Dubois, Théo Grandsart, Philippe Swartvagher



# First Experiments to Evaluate the Relevance of Task-based Runtime Systems to Implement Large Language Model Applications

Lionel Eyraud-Dubois[1][0000−0003−2475−3309], Théo Grandsart[2,1], and Philippe Swartvagher[3,1][0000−0003−3786−7364]

[1] Inria, Talence, France
lionel.eyraud-dubois@inria.fr
[2] Université de Bordeaux, Talence, France
theo.grandsart@u-bordeaux.fr
[3] Bordeaux INP, Inria, Talence, France
philippe.swartvagher@inria.fr

**Abstract.** During the last decade, a new kind of computation-intensive and complex application has emerged: Large Language Models (LLM). These applications require the computing power offered by HPC clusters and are complex to implement efficiently: several kinds of parallelisms are possible, limited available memory is often a major constraint, accelerators (GPUs, TPUs, NPUs, ...) need to schedule data transfers and computations, the problem size imposes distributed executions, ... All these challenges are already well-known by developers of the first-class citizen applications running on HPC clusters: linear algebra, numerical simulation, ... That is why task-based runtime systems have been proposed: to ease the writing of HPC applications by providing an abstraction of the machine and its efficient programming. Despite task-based runtime systems being used for a long time now for classic HPC applications, they are not used to implement LLM applications. In this paper, we present our first experiments to try to understand why this is the case and whether using task-based runtime systems for LLM applications (both training and inference) is relevant. We describe our implementation of a small LLM with StarPU, discuss the different choices we had to make and evaluate performance.

**Keywords:** Large Language Model · Task-based Runtime · Application.

## 1 Introduction

Large Language Models (LLMs) have become a widespread application for High Performance Computing centers, whose efficient implementation requires a careful organisation of several kinds of parallelism, with heterogeneous accelerators and strong constraints on the memory hierarchies and data transfers. Standard implementations are based on Python frameworks which require a significant programming effort to efficiently utilize all the computing capabilities of the

hardware. In this paper, we explore the possibility to use an efficient task-based runtime system to implement a reference LLM. Task-based runtime systems have been developed as an abstraction to help the implementation of complex HPC applications, but their usage for LLMs and more generally deep learning applications is still to be explored.

### 1.1 Task-based Runtime Systems

In a task-based implementation of an application, computations made by the application are modeled as a task graph. Each task is a pure function taking data as input and producing output data. Tasks are the nodes of the task graph. Edges of the graph represent the data dependencies between tasks: if task $A$ produces a data, which task $B$ needs as input, there will be a directed edge from the node $A$ to the node $B$. This edge means than task $B$ must be executed after task $A$. With this model, the runtime system can automatically schedule tasks in order to minimize application runtime, while respecting data dependencies. In addition to scheduling, task-based runtime systems can manage data transfers between main memory and accelerators, distributed executions, performance models, *etc.* The goal of task-based runtime systems is to abstract the architecture of the machine and let it handle most of its complexity, which makes writing parallel applications easier.

In this work, we use StarPU [4], a distributed task-based runtime system which leverages the *Sequential Task Flow*. With StarPU, each task is defined as a function to execute, but also with a set of data access modes: read-only (input data), write-only (output data) or read-write (both). Data accessed by tasks are abstracted as *data handles*. Then, the application to parallelize is written in a sequential way and consists of a sequence of task creation, by precising which data is accessed. By knowing which data are accessed and how, StarPU can generate the task graph and infer the parallelism. Tasks are then executed by *workers*, which is an abstraction of computing units, *e.g.* a CPU core or a GPU.

For distributed executions, StarPU uses MPI [1], usually with one MPI process per node. Each MPI process executes the same program: same tasks and same data handles, but data is not allocated on all nodes, but is distributed across the available nodes. The real location of the data is recorded in data handles. Since each MPI process builds the whole task graph, the actual location of data allows StarPU to decide by which node each task should be executed. Moreover, edges in the task graph which now span across two distinct nodes will require an MPI communication, managed directly by StarPU.

### 1.2 Large Language Models (LLMs)

Large Language Models are specific type of deep neural networks, mostly designed to generate output text from a given input text. They have been popularized by the success of ChatGPT, Gemini, Claude, among others, which allow users to have a "conversational experience" with the model. These models are

composed of many layers (computational operations which are successively applied to the input), and each layer is associated with parameters that describe the corresponding transformation. These parameters must be optimized in a training process so that the large language model can accurately perform the required task. Then, the model and the parameters can be used together to generate an output in what is called the inference process. In this work, we consider only the training of the model.

Large Language Models do not process words, but *tokens*, which are numbers representing portions of words. Hence, a tokenizer is required to convert the input data into a sequence of tokens in a pre-processing step which is performed before the actual training and not considered in our work. The goal of the model is to predict the most likely token which will follow an input sequence of tokens.

The training of a model is composed of the following steps:

1. *forward*: compute the output from the input data and current parameters;
2. *loss*: the output (a probability distribution over all possible output tokens) is compared to the ground truth and the difference is computed as a *loss* value that the training process tries to minimize;
3. *backward*: the layers are traversed in the reverse order to compute gradients, which represent how each parameter should be adjusted to reduce the loss;
4. *update*: the parameters are modified according to the computed gradients;
5. Repeat from step 1. until an acceptable loss is reached.

One sequence of these four steps forward→loss→backward→update is called an *iteration*. The input data for a given iteration is called a *batch*, containing several independent input sequences. The number of these sequences is called the *batch size* and is an important parameter that affects the computational efficiency and the convergence of the training procedure. The data that goes from one layer to the next in the forward pass are called *activations*.

There are several techniques to parallelize the training of LLMs. In this work, we will consider tensor parallelism where the computations inside a layer are parallelized; and data parallelism where several iterations are executed in parallel on different input data.

Many different models of LLMs exist. For the purpose of our work, the choice of the model does not really matter: the general shape of the task graph comes from the sequence of the training steps which need to be done regardless of the model; the model dictates which layers will be executed, how they are connected and what are the trainable parameters. For this work, we adapted the code of Andrej Karpathy [11], which is a simple implementation of the GPT-2 model [17], which is composed of 74 layers, organised in 12 transformer blocks.

### 1.3 Contributions and Outline of the paper

In this work, we implement a task-based version of the GPT-2 model, based on the code by Karpathy [11]. We describe our design decisions in Section 2, and show in Section 3 how the task-based approach allows to easily explore different

kinds of parallelism: tensor and data parallelism, as well as their combination. In Section 4, we extend this implementation to a distributed, multi-node solution and evaluate its performance. Related works are described in Section 5.

## 2 Implementation with a Task-based Runtime System

We based our task-based implementation on KARPATHY's CPU version of the training of the GPT-2 model, which is mostly sequential, only several OpenMP pragmas are used to trivially parallelize `for` loops in the most expensive model layers.

### 2.1 Taskification

The first work was to taskify the code. Fortunately, the implementation of LLMs is highly compatible with the task paradigm: each layer of the model becomes a task. KARPATHY's code already defines one function for each layer (GeLU, matrix multiplication, norm, softmax, ...). Each of these functions has become a type of task in our version. Hence, defining the tasks and data inputs/ouputs is quite straightforward.

### 2.2 Data Organization

The different tasks will access different types of data. *Input tokens* for training are loaded from the training dataset and are processed in the forward step, generating *activations*. *Target values* are also loaded from the dataset and will only be used to compute the loss of the model by comparison with the output of the model. *Model parameters* (weights and biases) are specific to each model layer. They are read during the forward and backward passes, and written in the update step. *Gradients* are also specific to each model layer and are temporary data which are used between the backward and update steps to know how to update the parameters.

All these data are represented as data handles, which are registered before the task submission by specifying their type and size. Actual memory allocation for pieces of data is done by STARPU, before executing the first task which produces that data. Data initialisation (loading from datasets or default values) is also performed inside dedicated tasks.

While input and target values for two different iterations are different and completely independent data, model parameters are updated between two iterations. This means the same memory allocations (*i.e.* data handles) will be used for model parameters in all iterations, but input/output data and gradients could use a different memory space for each iteration. Generally speaking in STARPU, to avoid generating spurious dependencies between tasks (which could limit parallelism), two independent pieces of data should be represented with two different data handles. It would thus be natural to register activations of different iterations as their own separate data handle.

### 2.3 Task Graph Creation

Tasks are created sequentially and for each task, accessed data handles are provided together with the access mode (read, write, or both). This information allows StarPU to generate the corresponding task graph.

The produced task graph of the LLM training is depicted on Figure 1: each iteration forms a long chain of tasks. During the backward step, the update task of each layer can be executed as soon as the backward task of the layer is finished. With this first task-based version, the resulting task graph offers very little parallelisation. This is especially true because update tasks take less time than backward tasks.

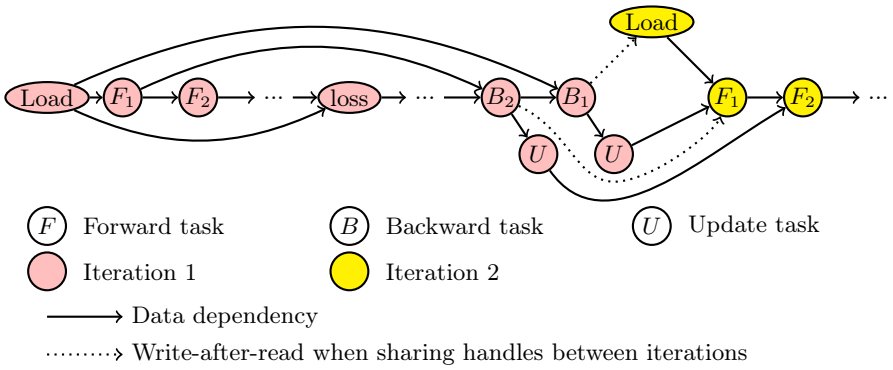



Fig. 1: Structure of the task graph for one iteration, and dependencies to the next iteration. Dependencies between forward and backward tasks are carried by activations, dependencies between backward and update tasks are carried by gradients, and dependencies between update and forward tasks are carried by model parameters. The dependency between the load and loss tasks is carried by the target ground truth.

It is usually a good practice to submit all tasks required by the application at once, so that StarPU is aware of the whole task graph and can optimize its execution. However, when activations of each iteration are registered as different data handles, this means that all initialization tasks can be executed immediately, which will trigger memory allocations before performing the actual initialization. Indeed, as can be seen on Figure 1 without the dotted edges, even for data that will be used very late in the task graph, these tasks do not have any preceding task and StarPU will start the task graph execution with them. This can lead to memory consumption issues: all of this memory allocation will take place at the beginning of the application, although it is not necessary that all data exist simultaneously during the application execution.

For this reason, in our case we have decided to reuse the same data handles for activations of different iterations. This reduces the memory usage by ensuring

that only one set of activations can be present in memory at any given time. Since the last update after the backward pass needs to be finished to start the forward pass of the next iteration, the write-after-read dependencies that are added by this change (shown as dotted edges in Figure 1) are already implied in the task graph by the real data dependencies. In practice, this means that in all iterations, we use the same data handles for inputs, target values, loss and gradient. This reduces the total number of data handles and consequently the total allocated memory. Initialization tasks for one iteration are submitted after all tasks of the previous iterations have been submitted. Since StarPU builds the task graph using data dependencies between tasks, it will automatically add a dependency between the last task which reads a data in the previous iteration and the first which writes a data in the next iteration, ensuring the correctness of the execution.

## 3 Get More Parallelism

As mentioned in the previous section, there is almost no parallelism in the basic task version of the LLM training, because of the structure of the training procedure itself. Training neural networks can however still exploit different types of parallelism. In this section, we implement two of them with StarPU: tensor and data parallelism, and eventually combine both of them.

### 3.1 Parallelism inside Tasks with OpenMP (Tensor Parallelism)

Among the operations executed by in the layers of LLM models, many of them can actually be performed in parallel: the most time-consuming layers consist of matrix multiplications (GEMM), and many layers apply the same operation to each element of a vector. This is called *Tensor Parallelism*.

There are several ways to express this parallelism with StarPU. The most natural would be to decompose a task into a set of tasks and let StarPU schedules the sub-tasks. This method requires to adapt both the operation kernels taken from Karpathy and the task submission logic. To avoid diverging too much from the original version and be able to have a fair comparison between the task- and non-task-based versions, we choose to retain the OpenMP pragmas above the `for` loops, and use an MKL implementation of matrix multiplication in both versions. In the context of StarPU, this means that one task can have a parallel section in its execution and needs to use several cores. The common usage of StarPU is to have CPU tasks executed by only one core, but StarPU also supports *parallel workers* [7]: available cores are clustered into a set of specific workers, which can execute tasks with parallel sections. All parallel workers encompass the same number of cores.

In general, the scalability of tensor parallelism (*i.e.* its ability to fully exploit all cores of a node) may be limited by problem size, because there might not be enough work to sufficiently feed all available cores.

### 3.2 Several Iterations in Parallel (Data Parallelism)

The input data of one iteration is a batch of several independent token sequences, and can thus be further divided into several smaller sub-batches. Each of these batches can then be processed in parallel in independent (sub-)iterations. This technique is called *Data Parallelism*, because the parallelism happens by splitting the data. All sub-iterations performed in parallel use the same model parameters, but input data, activations and gradients are specific to each sub-iteration. However, there is a new *reduction* step: after the backward tasks, gradients obtained by all sub-iterations must be averaged. Only after that, parameters of the model can be updated using the averages of gradients. This additional synchronisation step is required to take into account the work done by each sub-iteration.

With StarPU, data parallelism is easy to implement: the same sequences of tasks are submitted several times, with tasks of each sub-iteration accessing their own data handles. Since tasks of different sub-iterations manipulate different data, they can be executed in parallel. Averaging the gradients of all parallel sub-iterations is actually a *reduction* operation. One could manually submit tasks to perform this reduction, but StarPU has built-in support to reduce data between data handles. This feature uses internally a binary tree to perform the reduction.

If only data parallelism is used (*i.e.* tasks are not parallel) and given the little parallelism inside the task graph of one sub-iteration, StarPU can schedule one sequence per core, so that the level of parallelism is equal to the number of cores.

### 3.3 Mixing Data and Tensor Parallelisms

Our implementations of data and tensor parallelisms can be used independently, but it is also possible to use both simultaneously. From a technical perspective, it is possible that tasks used in parallel sub-iterations (data parallelism) are actually parallel tasks (tensor parallelism). Mixing both parallelisms can be interesting to cope with the limitations of tensor parallelism, especially when the matrices manipulated by tasks are too small and do not require a large number of cores to be efficiently parallelized.

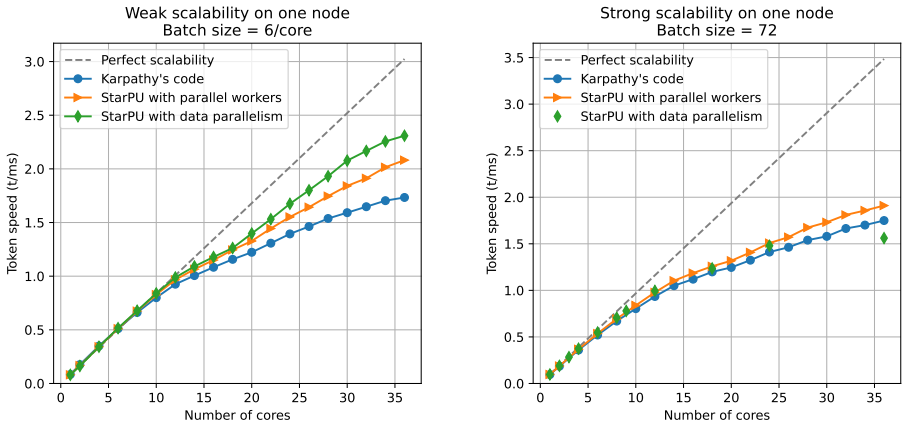
This raises the question of how to partition the machine: the number of available cores is always the product of the number of cores used by one parallel task and the number of parallel sequences, so that this partitioning can be specified with one parameter. In the next section, we explore how this additional parameter affects the training efficiency.

### 3.4 Experimental Results

To evaluate efficiency of these parallelisms, we used one node with dual Intel Xeon Gold 6240 at 2.6 GHz with 36 cores and 192 GB RAM, and equipped with Intel Omni-Path 100 series network. GPUs are not used for this study. Since our implementation is based on Karpathy's code, we also use the same training conditions: the dataset is a tokenized version of a subset of the work of Shakespeare, the model is GPT-2 with 124M parameters, 12 attention heads,

with channel size 768 and sequence length is 64. KARPATHY's code was modified only to use the GEMM routine provided by the MKL. GCC 15.1, STARPU 1.4.10 and INTEL MKL 2029.4 with GNU threads were used to build the applications.

In all of the results in this paper, the measurements are performed over 10 successive iterations, and the performance is displayed in terms of number of processed tokens per millisecond, obtained by dividing the total number of tokens processed by the execution time of the 10 iterations; higher values are better. Each training takes approximately one minute.




(a) Weak scalability of training with the same batch size per core. The data parallel version does not use parallel workers.

(b) Strong scalability of training with constant batch size showing the token speed according the number of cores.

Fig. 2: Scalability experiments for the data parallel and parallel workers versions.

Figure 2a compares the weak scalability of KARPATHY's version and the two parallelisms we implemented with STARPU. The batch size increases with the number of cores, so that the total work per core remains constant. In this experiment and the next one, STARPU with parallel worker uses only one parallel worker with a given number of cores, whereas the STARPU version with data parallelism does not use parallel tasks and the number of sub-batches trained in parallel is equal to the number of cores. The version of STARPU with parallel workers is very similar to KARPATHY's version, since they both use only tensor parallelism. We observe that up to 18 cores, the scalability is very good, but with more cores it starts to decrease. With more than 18 cores, the used cores are spread on the two processors, which implies more costly memory accesses. The tensor parallel versions have slightly worse scaling because they induce fine-grained synchronizations between cores, whereas with data parallelism all cores can work independently for most of the training iteration.

The performance of strong scalability is shown on figure 2b, where the batch size is kept constant. In that case, StarPU with data parallelism can only be evaluated for a number of cores such that number of cores × size of sub-batch = batch size = 72, *i.e.* the number of cores must be a divisor of 72. We obtain similar performance for the three implementations, with the same advantage for the StarPU versions as before. However, for the data parallel version on 36 cores, the performance is worse than using tensor parallelism (1.5 t/ms *vs* 1.7 and 1.9 t/ms for Karpathy and StarPU with parallel workers respectively) because sub-iterations work on batch sizes of $\frac{72}{36} = 2$ which is too small, making the overhead of StarPU noticeable, and there is an additional reduction in each iteration. One can also notice that the execution with 12 cores is the same than the execution with 12 cores in Figure 2a (where total batch size = 12 cores × batch size 6/core = 72).

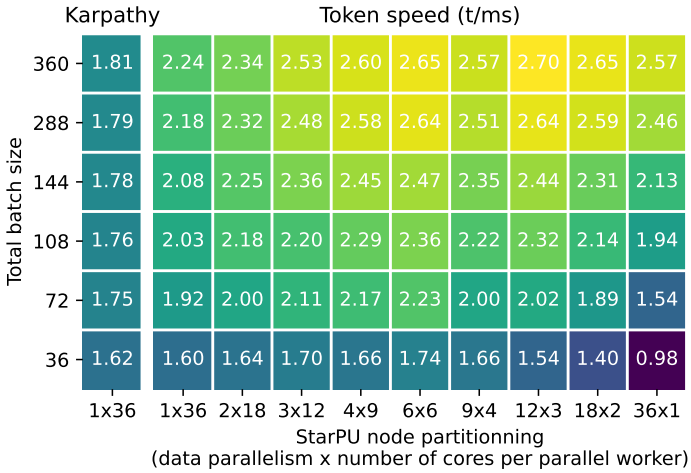



Fig. 3: Token speed for different total batch sizes according to how both parallelisms are mixed.

As explained previously, combining both tensor and data parallelisms requires to decide how many cores will be in each parallel worker. Since in this experiment we are always using all the available cores of the node, the number of parallel workers needs to be a divisor of 36. Parallelism inside one parallel worker is tensor parallelism and the different sub-iterations executed in parallel on different sub-batches by different parallel workers is data parallelism. Figure 3 shows the performance of the possible node partitioning for different total batch sizes. The total batch size has an impact on the size of the data going through the task graph. Karpathy's version does not support data parallelism. The partitioning `1x36` corresponds to having only tensor parallelism with only one parallel worker and the partitioning `36x1` corresponds to having only data

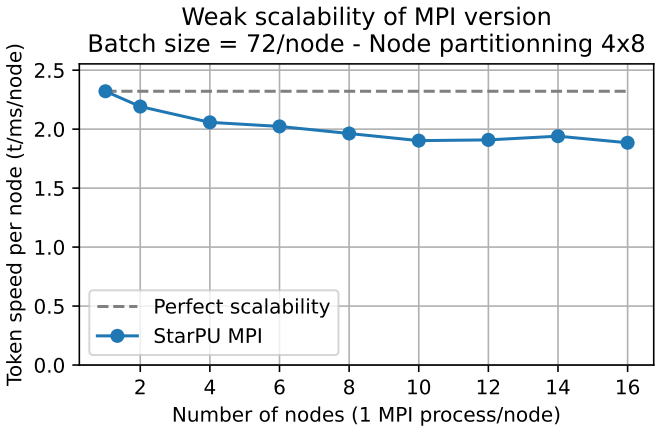



Fig. 4: Weak scalability of the StarPU-MPI version. Each node has parallel workers with 8 cores and 4 sequences are executed in parallel.

parallelism with sequential tasks. Moreover, the executions `1x36` and `36x1` with a total batch size of 72 are the same than when 36 cores are used in figure 2b. In general, we see that increasing the total batch size increases also the token speed: lower performance with smaller batch sizes can be explained by the lack of work to feed all processing units. The best node partitioning depends on the total batch size and is not symmetric: the best token speed 2.70 t/ms is achieved with a total batch size of 360 and the partitioning `12x3`, while with the same batch size, the partitioning `3x12` gives 2.53 t/ms.

Overall, we observe that the StarPU implementation has similar or better performance than Karpathy's version. When we reach the limits of tensor parallelism, the data parallelism we implemented in the StarPU version helps to better exploit the computing power of the machine, under the condition to have a large enough sub-batch.

## 4 Distributed Version with MPI

We extended our StarPU implementation to work also on several nodes. The required changes are quite minor, since StarPU handles all the communications between MPI processes [1]. All MPI processes execute the same StarPU application and submit the whole task graph. The actual memory corresponding to data accessed by tasks is really allocated only on one node and this will determine which tasks each MPI process will execute.

Our distributed version relies on data parallelism: each MPI process works on a different sub-batch, and the model parameters are duplicated on each MPI process. At the end of each iteration, gradients of all sub-iterations of all MPI

processes are reduced by StarPU to a single MPI process, which performs the parameter update. For the next iteration, all MPI processes need the updated model parameters and since this data dependency is expressed in the task graph, StarPU will send the new parameters to all MPI processes. The most notable change in the code to have a distributed version is to indicate which MPI process owns the memory of each data handle.

The weak scalability of distributed training with OpenMPI 4.1.5 and one MPI process per node is shown on figure 4. StarPU requires to dedicate one core to handle communication progress, thus in order to have correct multiples and divisors for node partitioning and total batch size, we used only 32 cores of each node split into 4 parallel workers, which according to Figure 3 is one of the best node partitioning for this batch size. Results show that between 1 and 16 nodes the token speed drops from 2.32 to 1.88 t/ms respectively, *i.e.* a decrease of 19 %, which can be considered as good scalability.

## 5 Related work

Task-based runtime systems have been developed in the last decades to provide a simple abstraction for programming parallel and distributed HPC applications. Such systems, like StarPU [4], Legion [5], ParSeC [9], or OMPsS [8], have allowed users to express complex applications [6,3] or linear algebra algorithms [1,2,16] in a performance-portable way, adapting seamlessly to the characteristics of the available hardware. In distributed settings, these systems feature an integration with the MPI communication library, performing automatic data management to ensure the correctness and efficiency of the distributed execution.

In contrast, the majority of neural network frameworks like TensorFlow [14], PyTorch [15] or JAX [18] require explicit data placement and transfers when performing parallel execution. This requires the programmer to carefully optimize the asynchronous execution, whereas the corresponding techniques (prefetching, cache mechanisms, ...) are well understood in task-based systems. Some frameworks have been proposed to obtain efficient execution of neural networks, but they are usually restricted to the inference part [12], and use a monolithic implementation that makes it difficult to benefit of the task-based approaches.

Some task-based implementations of deep neural networks have been proposed. FlexFlow [10] is based on the Legion runtime system [5] and features an automatic decomposition into different forms of parallelism. NNTile [13] is an attempt to port the standard `nn.Module` PyTorch class onto the StarPU runtime system, allowing for arbitrary tiling of tensors. These solutions require significant engineering so that they can be used on any kind of neural network. In this work, we focus on a simple representative neural network, which is why we have chosen the `llm.c` [11] implementation which has a similar purpose.

## 6 Conclusion

In this work, we present a task-based implementation of a reference LLM model, highlighting the benefit of the task-based runtime system for easily and efficiently implementing different kinds of parallelism for deep learning model training. We obtain a parallel and distributed implementation with simple adaptations of the code, and our performance evaluation shows that it obtains good scalability.

This work can be extended in several ways. First of all, we would like to analyze deeper the performance. Then, we plan to adapt the CUDA-based version from Karpathy to be able to target GPU accelerators, and use the data management features of StarPU to perform automatic offloading of activations if they do not fit in memory. We will also explore the pipeline parallelism (where each node or GPU only processes a part of the model), which may require adding scheduling algorithms adapted to this particular workload.

**Software Availability** A public companion[4] contains the instructions to reproduce our study.

### Author Contributions

- Lionel Eyraud-Dubois: Writing, Methodology, Supervision
- Théo Grandsart: Implementation, Writing
- Philippe Swartvagher: Debugging, Writing, Methodology, Supervision

**Acknowledgments.** Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see `https://www.plafrim.fr`).

## References

1. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.: Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE Transactions on Parallel and Distributed Systems (2017)
2. Agullo, E., Bosilca, G., Buttari, A., Guermouche, A., Lopez, F.: Exploiting a Parametrized Task Graph model for the parallelization of a sparse direct multifrontal solver. In: Euro-Par 2016: Parallel Processing Workshops. Grenoble, France (Aug 2016)
3. Alonazi, A., Ltaief, H., Keyes, D., Said, I., Thibault, S.: Asynchronous Task-Based Execution of the Reverse Time Migration for the Oil and Gas Industry. In: CLUSTER 2019 - IEEE International Conference on Cluster Computing. pp. 1–11. IEEE, Albuquerque, United States (Sep 2019)

[4] `https://gitlab.inria.fr/pswartva/paper-wamta2026-starpu-llm-reproducibility`, archived on `https://www.softwareheritage.org/` with the ID `swh:1:snp:d38d9c09f19231e5e44a4969d588fdcb25118200`.

4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
5. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE (2012)
6. Blanchard, P., Bramas, B., Coulaud, O., Darve, E., Dupuy, L., Etcheverry, A., Sylvand, G.: ScalFMM: A generic parallel fast multipole library. In: SIAM Conference on Computational Science and Engineering (SIAM CSE 2015) (2015)
7. Cojean, T., Guermouche, A., Hugo, A., Namyst, R., Wacrenier, P.: Resource aggregation for task-based Cholesky Factorization on top of modern architectures. Parallel Computing **83**, 73–92 (2019)
8. Fernández, A., Beltran, V., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Task-based programming with ompss and its application. In: European Conference on Parallel Processing. pp. 601–612. Springer (2014)
9. Hoque, R., Herault, T., Bosilca, G., Dongarra, J.: Dynamic task discovery in parsec: A data-flow task-based runtime. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. pp. 1–8 (2017)
10. Jia, Z., Zaharia, M., Aiken, A.: Beyond data and model parallelism for deep neural networks. Proceedings of Machine Learning and Systems **1**, 1–13 (2019)
11. Karpathy, A.: llm.c – LLM training in simple, raw C/CUDA (2025), `https://github.com/karpathy/llm.c`
12. Microsoft: ONNX Runtime: cross-platform, high performance ML inferencing and training accelerator (2025), `https://www.onnxruntime.ai/`
13. Mikhalev, A., Katrutsa, A., Sozykin, K., Oseledets, I.: NNTile: a machine learning framework capable of training extremely large GPT language models on a single node. arXiv preprint arXiv:2504.13236 (2025)
14. Pang, B., Nijkamp, E., Wu, Y.N.: Deep learning with tensorflow: A review. Journal of Educational and Behavioral Statistics **45**(2), 227–248 (2020)
15. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)
16. Pichon, G., Darve, E., Faverge, M., Ramet, P., Roman, J.: Sparse supernodal solver using block low-rank compression: Design, performance and analysis. International Journal of Computational Science and Engineering **27**, 255 – 270 (Jul 2018)
17. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)
18. Schoenholz, S., Cubuk, E.D.: Jax md: a framework for differentiable physics. Advances in Neural Information Processing Systems **33**, 11428–11441 (2020)