Memory-Constrained Scheduling for Distributed LLM

Alaa El Ouahabi
Cybersecurity Master's student
alaa.elouahabi@epfl.ch

Supervisors:
Gauthier Voron
gauthier.voron@epfl.ch

Geovani Rizk
geovani.rizk@epfl.ch

Ecole Polytechnique Fédérale de Lausanne (EPFL)

Abstract

The widespread adoption of Large Language Models (LLMs) is hindered by their substantial computational and memory requirements, typically necessitating expensive GPU clusters. This paper presents a novel scheduling framework for distributed LLM execution on commodity hardware, enabling organizations to deploy these models on existing infrastructure such as networks of laptops and workstations. We identify memory management as the critical bottleneck in commodity hardware deployment and propose four scheduling algorithms optimized for different aspects of distributed execution. Our key contribution is the MRU-Enhanced scheduler, which employs intelligent parameter caching and eviction strategies specifically designed for LLM workload patterns. Extensive evaluation across various memory regimes (80%-100% of required memory) demonstrates that MRU achieves near-perfect task completion rates (95%+) even under severe memory constraints, while traditional computation-focused schedulers fail dramatically (68% completion). For LLM-specific workloads, MRU achieves 100% task completion across all memory configurations, though with a 30% increase in makespan compared to optimal scheduling. Our results show that memory-aware scheduling is not just beneficial but essential for distributed LLM deployment on commodity hardware, opening new possibilities for privacy-preserving and latency-sensitive applications.

1 Introduction

Large Language Models (LLMs) have revolutionized natural language processing, achieving remarkable performance across diverse tasks from text generation to complex reasoning [2, 17]. However, their deployment remains largely centralized in cloud environments due to substantial computational and memory requirements. A typical LLM with billions of parameters requires tens to hundreds of gigabytes of memory, far exceeding the capacity of individual commodity devices [14].

This centralization creates significant challenges for organizations with strict data privacy requirements, latency-sensitive applications, or limited cloud budgets. Healthcare institutions processing patient data, financial services handling sensitive transactions, and edge computing scenarios all require local model deployment [10]. Yet the current paradigm forces these organizations to choose between compromising on privacy by using cloud services or forgoing LLM capabilities entirely.

We propose a radically different approach: distributed execution of LLMs across networks of commodity hardware. Modern organizations typically possess substantial aggregate computational resources in the form of employee workstations, laptops, and other devices. For instance, a network of 10 laptops with

8GB RAM each provides 80GB of aggregate memory sufficient for many LLM applications. However, effectively utilizing these distributed resources presents unique scheduling challenges.

1.1 Challenges in Distributed LLM Execution

Distributed execution of LLMs on commodity hardware faces several fundamental challenges that distinguish it from traditional distributed computing:

Memory Fragmentation: Unlike traditional HPC clusters with homogeneous nodes, commodity hardware exhibits significant heterogeneity in memory capacity and compute speed. Tasks cannot be split across nodes, leading to stranded memory that cannot be utilized effectively.

Parameter Management Overhead: LLM computations require loading large parameter matrices (often 0.5-2GB each) that may be shared across multiple operations. When tasks execute on different nodes, parameters must be replicated, dramatically increasing actual memory requirements beyond naive calculations.

Dynamic Node Availability: Commodity devices like laptops may join or leave the network dynamically as users start or stop sharing their resources. This requires scheduling algorithms that can quickly adapt to changing resource availability.

Bandwidth Constraints: Unlike datacenter networks with high-bandwidth interconnects, commodity networks (e.g., office WiFi) have limited bandwidth, making parameter transfer a significant bottleneck.

1.2 Key Insights and Contributions

Through extensive analysis of LLM computation patterns, we identify that *memory management, not computation scheduling, is the primary bottleneck* for distributed execution on commodity hardware. This insight drives our approach and distinguishes it from traditional distributed scheduling work.

Our main contributions are:

1. **Comprehensive Analysis of Distributed LLM Execution:** We characterize the unique challenges of running LLMs on commodity hardware networks, identifying memory fragmentation and parameter management as critical bottlenecks.

2. **Novel Scheduling Algorithms:** We develop four scheduling algorithms optimized for different aspects:

    - *DFS-Enhanced:* Prioritizes deep task chains to minimize memory residence time
    - *Critical Path:* Optimizes makespan through traditional critical path analysis
    - *Greedy-Enhanced:* Exploits LLM-specific sequential patterns
    - *MRU-Enhanced:* Implements intelligent parameter caching with predictive eviction

3. **Memory-Aware Scheduling Framework:** We introduce the MRU-Enhanced scheduler that achieves near-perfect task completion (95%+) even under severe memory constraints (80% of required memory), compared to 68% for traditional approaches.

4. **Extensive Evaluation:** We evaluate our algorithms on diverse workloads under varying memory regimes (80%-100%) and node configurations (2-8 nodes), demonstrating the critical importance of memory-aware scheduling.

5. **Practical Deployment Guidelines:** We provide concrete recommendations for practitioners, showing when to use each scheduling approach based on memory availability and workload characteristics.

Our results demonstrate that distributed LLM execution on commodity hardware is not only feasible but practical when using appropriate scheduling strategies. The MRU-Enhanced scheduler achieves 100% task completion for LLM workloads across all memory configurations, though with a 30% makespan increase a favorable trade-off for scenarios prioritizing reliability and local deployment.

2 Related Work

Our work builds upon and extends research in distributed scheduling, large-scale model serving, and parameter management systems. We organize related work into four categories.

2.1 Distributed Scheduling Algorithms

Classical distributed scheduling has been extensively studied in the context of high-performance computing. Traditional approaches like list scheduling [7], critical path methods [6], and the widely-used Heterogeneous Earliest Finish Time (HEFT) algorithm [16] optimize for makespan minimization assuming heterogeneous nodes but reliable network connections.

However, these algorithms fail in commodity hardware environments for two key reasons: (1) they assume memory is always sufficient, focusing solely on computation scheduling, and (2) they do not account for the cost of parameter loading and replication across nodes. While HEFT [16] and its companion algorithm

CPOP consider nodes with different computational capabilities, they still assume sufficient memory and stable connectivity. Our MRU-Enhanced scheduler addresses these limitations through predictive parameter management and memory-aware task assignment, extending beyond the computational focus of traditional schedulers.

Recent work on heterogeneous scheduling continues this computation-centric tradition. Dynamic Critical-Path (DCP) scheduling [6] adapts critical path analysis for dynamic task graphs but does not address the parameter management challenges unique to LLM workloads. Our work differs by recognizing that for LLM inference on commodity hardware, memory management dominates performance considerations.

2.2 Large Model Serving Systems

The emergence of large models has driven development of specialized serving systems. Megatron-LM [14] pioneered tensor parallelism for training multi-billion parameter models but requires high-bandwidth interconnects unavailable in commodity settings. FairScale [3] and DeepSpeed [11] provide pipeline parallelism strategies and system optimizations for training models with over 100 billion parameters, but both assume dedicated GPU clusters with reliable, high-speed interconnects.

More recent systems like Alpa [18] automatically discover optimal parallelization strategies for large models by jointly optimizing inter- and intra-operator parallelism. While Alpa's inter-operator parallelism is conceptually similar to our distributed task scheduling, it targets datacenter deployments with reliable, high-bandwidth networks and does not address dynamic node availability or severe memory constraints characteristic of commodity hardware.

PipeSwitch [1] introduces fast pipelined context switching for deep learning applications, enabling efficient multiplexing of GPU resources across multiple models. However, PipeSwitch operates on single nodes or tightly coupled GPU clusters and does not address the distributed parameter management challenges we tackle. Our work extends these ideas to loosely coupled commodity hardware networks where network bandwidth and memory fragmentation become primary concerns.

2.3 Parameter Management and Caching

The parameter server architecture [8] pioneered distributed parameter management for machine learning, enabling scaling to billions of parameters. However, parameter servers focus on gradient synchronization during training rather than inference-time parameter loading and assume reliable cluster environments. Our MRU-Enhanced scheduler adapts parameter management concepts for inference on unreliable commodity networks.

Recent work on parameter caching for inference includes vDNN [12], which virtualizes GPU memory by intelligently swapping to CPU memory, and SwapAdvisor [4], which uses a genetic algorithm to determine optimal swapping decisions. While both systems optimize memory usage, they operate on single nodes with GPU-CPU memory hierarchies. Our approach differs by distributing parameters across multiple physical devices and must consider network transfer costs in eviction decisions.

FlexGen [13] enables high-throughput LLM inference on a single GPU by offloading to CPU and disk memory, achieving impressive throughput through careful scheduling of memory, computation, and communication. FlexGen's linear programming formulation for throughput optimization inspired our approach, but we extend it to distributed settings where network topology and dynamic node availability introduce additional complexity. Our MRU-Enhanced scheduler builds on FlexGen's insights while addressing the unique challenges of commodity hardware networks.

2.4 Edge and Federated Learning Systems

Edge computing systems [9, 15] share our goal of distributed execution on commodity hardware but typically focus on inference of smaller models or collaborative training. The communication constraints in edge computing motivated our bandwidth-aware scheduling, but edge computing typically assumes models that fit on individual devices.

Federated learning [10] distributes training across devices while keeping data local, addressing privacy concerns similar to our motivation. However, federated learning focuses on collaborative training rather than distributed inference of a single large model. The system challenges differ significantly: federated learning must aggregate gradients from heterogeneous data distributions, while we must efficiently distribute and cache model parameters for inference.

Split computing approaches like Neurosurgeon [5] partition neural networks between edge devices and the cloud, using dynamic programming to find optimal split points. While Neurosurgeon considers bandwidth and latency in its partitioning decisions, it assumes reliable cloud connectivity and focuses on single-device-to-cloud scenarios rather than peer-to-peer distribution across commodity hardware.

2.5 Summary

While extensive research exists on distributed scheduling, large model serving, and edge computing, our work is the first to comprehensively address distributed LLM execution on commodity hardware networks. Traditional scheduling algorithms like HEFT [16] optimize for computation but ignore memory constraints. Modern model serving systems like Alpa [18] and PipeSwitch [1] target datacenter environments. Memory optimization work like FlexGen [13] focuses on single nodes. We combine insights from these domains with novel memory-aware scheduling algorithms specifically designed for LLM workload patterns and commodity hardware constraints. Our MRU-Enhanced scheduler's ability to achieve near-perfect task completion under severe memory constraints represents a significant advance in making LLMs accessible beyond centralized cloud deployments.

3 System Design and Algorithms

In this section, we present our scheduling framework and algorithms designed specifically for distributed LLM execution on commodity hardware. We first describe our system model and problem formulation, then detail our four scheduling algorithms.

3.1 System Model

3.1.1 Task Model

We model LLM computation as a directed acyclic graph (DAG) $G = (V, E)$, where each vertex $v \in V$ represents a computational task and edges represent dependencies. Each task t_i is characterized by:

- m_i: Memory required for computation (in GB)
- c_i: Computation time on a reference node
- P_i: Set of model parameters required
- D_i: Set of predecessor tasks (dependencies)

For LLM inference, tasks typically correspond to transformer layer operations (attention, FFN) with parameters representing weight matrices. The parameter size is substantial, each parameter $p \in P_i$ requires approximately 0.5GB of memory for typical LLM configurations.

3.1.2 Node Model

The system consists of heterogeneous commodity nodes $N = \{n_1, n_2, ..., n_k\}$, where each node n_j has:

- M_j: Total memory capacity
- s_j: Compute speed relative to reference (0.7-1.3×)
- C_j: Set of currently cached parameters
- A_j: Available memory at time t

Nodes may join or leave the network dynamically, though we focus on static configurations in this work.

3.1.3 Memory Constraints

The key challenge in our setting is that total available memory across all nodes may be less than the sum of all task and parameter requirements. We define the *memory regime* ρ as:

$$\rho = \frac{\sum_{j=1}^{k} M_j}{\sum_{i=1}^{|V|} m_i + |P| \cdot \sigma_p} \tag{1}$$

where $|P|$ is the total number of unique parameters and $\sigma_p = 0.5$GB is the parameter size.

3.2 Problem Formulation

Given a DAG G, a set of nodes N, and memory regime ρ, our goal is to find a schedule $S : V \to N \times R$ that assigns each task to a node and start time while:

1. Respecting task dependencies
2. Satisfying memory constraints at each node
3. Maximizing task completion rate
4. Minimizing makespan (secondary objective)

This problem extends classical DAG scheduling by introducing parameter management and memory constraints as first-class concerns.

3.3 Base Scheduler Framework

All our algorithms inherit from a common base scheduler that provides core functionality:

The base scheduler maintains the system state and provides common operations like checking task readiness and memory availability. Derived schedulers implement different strategies for selecting which tasks to schedule and which nodes to use.

3.4 DFS-Enhanced Scheduler

The DFS-Enhanced scheduler prioritizes tasks based on their depth in the DAG, aiming to complete deep dependency chains quickly to free memory.

The key insight is that completing deeper tasks first prevents memory fragmentation from long-lived intermediate results. This is particularly effective for LLM DAGs with sequential layer structures.

Algorithm 1 Base Scheduler Framework

```
class BaseScheduler:
    $nodes \leftarrow \{n_j.id : n_j\}$                ▷ Node registry
    $tasks \leftarrow \{\}$                            ▷ Task registry
    $completed \leftarrow \emptyset$                   ▷ Completed tasks
    $failed \leftarrow \emptyset$                      ▷ Failed tasks
    $pending \leftarrow \emptyset$                     ▷ Pending tasks
    $param\_locations \leftarrow \{\}$                 ▷ Parameter cache tracking
function IsTaskReady($task$)
    return $task.dependencies \subseteq completed$
end function
function CanFitOnNode($task, node$)
    $params\_needed \leftarrow task.P - node.C$
    $mem\_required \leftarrow task.m + |params\_needed| \cdot \sigma_p$
    return $mem\_required \leq node.A$
end function
function AssignTask($task, node$)
    Load missing parameters to $node$
    Update $node.A$ and $node.C$
    $task.assigned\_node \leftarrow node$
    Mark task as completed ▷ Simulated execution
end function
```

3.5 Critical Path Scheduler

Inspired by HEFT [16], this scheduler uses critical path analysis to minimize makespan:

While effective for computation optimization, this approach often fails under memory constraints as it doesn't consider parameter loading costs.

3.6 Greedy-Enhanced Scheduler

The Greedy scheduler exploits LLM-specific patterns by identifying sequential chains and optimizing parameter locality:

By keeping sequential chains on the same node, this approach minimizes parameter transfer overhead, which is crucial for bandwidth-limited commodity networks.

3.7 MRU-Enhanced Scheduler

Our most sophisticated scheduler implements intelligent parameter caching with predictive eviction:

The MRU scheduler's key innovations are:

1. **Predictive eviction**: Parameters needed by upcoming tasks receive high scores to prevent eviction
2. **Usage tracking**: Both frequency and recency influence caching decisions
3. **Lookahead**: Considers future parameter needs when making current decisions

Algorithm 2 DFS-Enhanced Scheduling Algorithm

```
function ComputeDepth($task, memo$)
    if $task \in memo$ then return $memo[task]$
    end if
    if $task.dependencies = \emptyset$ then
        $depth \leftarrow 0$
    else
        $depth \leftarrow 1 + \max_{d \in task.dependencies}$ ComputeDepth($d, memo$)
    end if
    $memo[task] \leftarrow depth$
    return $depth$
end function
function Schedule
    $depths \leftarrow \{\}$
    for $task \in tasks$ do
        ComputeDepth($task, depths$)
    end for
    while $pending \neq \emptyset$ do
        $ready \leftarrow$ tasks where IsTaskReady($task$)
        Sort $ready$ by $depths[task]$ (descending)
        for $task \in ready$ do
            $best\_node \leftarrow$ node with max available memory
                where CanFitOnNode($task, node$)
            if $best\_node \neq null$ then
                AssignTask($task, best\_node$)
            else
                Mark $task$ as failed
            end if
        end for
    end while
end function
```

4 Implementation

We implemented our scheduling framework and algorithms in Python, designed for both experimental evaluation and practical deployment. This section describes key implementation details and optimizations.

4.1 Architecture Overview

Our implementation consists of several key components:

Core Scheduler Engine: The `BaseScheduler` class implements common functionality including task dependency tracking, memory management, and node state maintenance. All scheduling algorithms inherit from this base class.

Algorithm Implementations: Each scheduling algorithm extends the base class:

- `DFSEnhancedScheduler`: Implements depth-based prioritization

Algorithm 3 Critical Path Scheduling Algorithm

```
1:  function ComputeCriticalPath($task, memo$)
2:      if $task \in memo$ then return $memo[task]$
3:      end if
4:      $dependents \leftarrow$ tasks that depend on $task$
5:      if $dependents = \emptyset$ then
6:          $path\_length \leftarrow task.c$
7:      else
8:          $path\_length \leftarrow task.c + \max_{d \in dependents}$
9:                          ComputeCriticalPath($d, memo$)
10:     end if
11:     $memo[task] \leftarrow path\_length$
12:     return $path\_length$
13: end function
14: function Schedule
15:     Compute critical paths for all tasks
16:     Sort ready tasks by critical path (descending)
17:     for $task \in ready\_tasks$ do
18:         $best\_node \leftarrow$ fastest node where task fits
19:         Assign task to $best\_node$ if found
20:     end for
21: end function
```

- `CriticalPaScheduler`: Critical path scheduling inspired by HEFT
- `GreedyEnhancedScheduler`: Chain-aware scheduling for LLM patterns
- `MRUEnhancedScheduler`: Intelligent parameter caching with eviction

Testing Infrastructure: We implemented DAG generators for different workload types and evaluation utilities to measure scheduler performance across various metrics.

The implementation totals approximately 600 lines of Python code, focusing on clarity and extensibility while maintaining efficiency for experimental evaluation.

4.2 Key Data Structures

4.2.1 Task Representation

Tasks are represented as Python objects with efficient access to dependencies and parameters:

```python
class Task:
    def __init__(self, task_id, memory_required,
                 compute_time,
                     dependencies=None,
                     params_needed=None):
        self.id = task_id
        self.memory_required = memory_required
        self.compute_time = compute_time
        self.dependencies = dependencies or []
        self.params_needed = params_needed or set()
        self.completed = False
        self.assigned_node = None
        self.attempted = False  # Track if task was attempted
```

Algorithm 4 Greedy-Enhanced Scheduling with Chain Detection

```
1:  function IdentifyChains
2:      $chains \leftarrow []$
3:      $visited \leftarrow \emptyset$
4:      for $task \in tasks$ with no dependencies do
5:          if $task \notin visited$ then
6:              $chain \leftarrow$ BuildChain($task, visited$)
7:              if $|chain| > 1$ then
8:                  $chains$.append($chain$)
9:              end if
10:         end if
11:     end for
12:     return $chains$
13: end function
14: function Schedule
15:     $chains \leftarrow$ IdentifyChains
                    ▷ Assign chains to nodes with most memory
16:     for $chain \in chains$ do
17:         $best\_node \leftarrow$ node with maximum available memory
18:         for $task \in chain$ do
19:             if CanFitOnNode($task, best\_node$) then
20:                 AssignTask($task, best\_node$)
21:             end if
22:         end for
23:     end for
                                    ▷ Handle remaining tasks
24:     for $task \in ready\_tasks$ do
25:         $best\_node \leftarrow$ node with most cached parameters
26:                 from $task.P$
27:         Assign if possible
28:     end for
29: end function
```

4.2.2 Memory Tracking

We maintain precise memory accounting at each node:

```python
class Node:
    def __init__(self, node_id, total_memory, compute_speed=1.0):
        self.id = node_id
        self.total_memory = total_memory
        self.available_memory = total_memory
        self.compute_speed = compute_speed
        self.cached_params = set()
        self.running_tasks = []
        self.completed_tasks = []
        self.last_used_params = deque(
            maxlen=10)  # For MRU
```

4.2.3 Parameter Location Index

To efficiently track where parameters are cached, we maintain a global index:

```
self.param_locations = defaultdict(set)  #
    param_id -> {node_ids}
```

This enables $O(1)$ lookup of parameter locations, crucial for the MRU scheduler's decision-making.

4.3 Optimization Techniques

4.3.1 Dependency Graph Preprocessing

We precompute the transitive closure of dependencies and reverse dependency mappings at initialization:

```
def build_dependency_maps(self):
    for task_id, task in self.tasks.items
        ():
        for dep in task.dependencies:
            self.task_dependents[dep].
                append(task_id)
```

This allows $O(1)$ checking of task readiness and dependent identification.

4.3.2 Incremental Ready Queue Maintenance

Rather than scanning all tasks to find ready ones, we maintain a ready queue incrementally:

```
def complete_task(self, task_id):
    # ... mark task as completed ...

    # Check if any dependents are now
        ready
    for dependent_id in self.
        task_dependents[task_id]:
        if self.is_task_ready(dependent_id
            ):
            self.ready_queue.add(
                dependent_id)
```

4.3.3 Memory-Aware Fast Path

For the common case where a task fits without eviction, we use a fast path that avoids expensive eviction calculations:

```
def try_assign_task(self, task, node):
    # Fast path: check if task fits
        without eviction
    if self.can_fit_on_node(task, node):
        return self.assign_task_to_node(
            task, node)

    # Slow path: try eviction (MRU only)
    if self.supports_eviction:
        return self.
            try_assign_with_eviction(task,
            node)

    return False
```

4.4 LLM-Specific Optimizations

4.4.1 Chain Detection for Sequential Patterns

The Greedy scheduler implements chain detection to identify sequential patterns common in LLM architectures:

```
def identify_sequential_chains(self):
    """Identify sequential chains in the
        DAG"""
    chains = []
    visited = set()

    # Find chain starts (tasks with no
        dependencies)
    starts = [t for t in self.tasks.values
        () if not t.dependencies]

    for start in starts:
        if start.id in visited:
            continue

        chain = []
        current = start

        while current and current.id not
            in visited:
            chain.append(current.id)
            visited.add(current.id)

            # Find single dependent
            dependents = self.
                task_dependencies.get(
                current.id, [])
            if len(dependents) == 1 and
                dependents[0] in self.
                tasks:
                 current = self.tasks[
                     dependents[0]]
            else:
                 break

        if len(chain) > 1:
            chains.append(chain)

    return chains
```

4.4.2 Parameter Usage Tracking

The MRU scheduler tracks parameter usage patterns to make intelligent caching decisions:

```python
def calculate_eviction_score(self, param, node):
    """Calculate score for eviction (lower = evict first)"""
    score = 0.0

    # Frequency component
    score += self.param_usage_count[param] * 10

    # Recency component
    if param in self.param_last_used:
        recency = self.time_step - self.param_last_used[param]
        score += 100.0 / (recency + 1)

    # Check if needed by running tasks
    for task_id in node.running_tasks:
        if task_id in self.tasks:
            task = self.tasks[task_id]
            if param in task.params_needed:
                score += 1000  # Very high score - don't evict

    return score
```

4.5 Execution Simulation

For evaluation, task execution is simulated within the scheduler itself. When a task is assigned to a node, we immediately mark it as completed and update the node's memory state:

```python
def complete_task(self, task_id):
    """Mark task as completed and free resources"""
    task = self.tasks[task_id]
    if not task.assigned_node:
        return

    node = self.nodes[task.assigned_node]

    task.completed = True
    self.completed_tasks.add(task_id)

    if task_id in node.running_tasks:
        node.running_tasks.remove(task_id)
    node.completed_tasks.append(task_id)
    node.available_memory += task.memory_required
```

This approach simplifies experimentation while maintaining accurate memory accounting. The makespan calculation in the testing framework considers node compute speeds and task dependencies to model realistic execution times.

4.6 Testing and Evaluation Infrastructure

For reproducible evaluation, we implemented comprehensive testing utilities:

4.6.1 DAG Generation

We provide generators for three types of workloads:

```python
def create_example_dag():
    """Create an example LLM-like DAG"""
    tasks = []

    # Transformer layers (sequential)
    for i in range(12):
        deps = [f"layer_{i-1}"] if i > 0 else []
        params = {f"layer_{i}_weights", f"layer_{i}_bias"}
        task = Task(f"layer_{i}", memory_required=1.0,
                    compute_time=0.1,
                      dependencies=deps,
                    params_needed=params)
        tasks.append(task)

    # Final output layer
    task = Task("output", memory_required=0.5, compute_time=0.05,
                dependencies=["layer_11"],
                params_needed={"output_weights"})
    tasks.append(task)

    return tasks
```

4.6.2 Performance Metrics

The testing framework calculates key metrics including:

- Task completion rate
- Makespan (total execution time)
- Parameter cache hit/miss rates
- Load balance across nodes
- Node utilization

These metrics enable comprehensive comparison of scheduling strategies under different memory regimes and workload characteristics.

4.7 Practical Deployment Considerations

While our current implementation focuses on evaluation through simulation, the design supports real deployment:

Extensibility: New scheduling algorithms can be added by extending the `BaseScheduler` class and implementing the `schedule()` method.

Integration: The scheduler can be integrated with ML frameworks through task wrappers that map framework operations to our task abstraction.

Monitoring: The scheduler maintains detailed state that can be queried for system monitoring and debugging.

Memory Safety: The implementation includes checks to prevent over-allocation and handles failed task assignments gracefully.

5 Evaluation

We evaluated our scheduling algorithms through extensive experiments designed to assess their performance under varying memory constraints and workload characteristics. Our evaluation focuses on the algorithms' ability to handle Large Language Model (LLM) workloads on commodity hardware with limited memory resources.

5.1 Experimental Setup

5.1.1 Scheduling Algorithms

We implemented and evaluated four scheduling algorithms, each optimized for different aspects of distributed LLM execution:

- **DFS**: Prioritizes tasks based on their depth in the dependency graph, aiming to complete deep chains quickly
- **Critical**: Uses critical path analysis to prioritize tasks that lie on the longest execution path
- **Greedy**: Optimizes for parameter locality and identifies sequential execution chains common in LLM workloads
- **MRU_spec**: Implements a Most Recently Used parameter caching strategy with intelligent eviction policies

5.1.2 Workload Characteristics

We tested our algorithms on six different DAG types representing various computational patterns:

1. **LLM DAGs** (Small, Medium, Large): Transformer-based architectures with 4, 8, and 12 layers respectively, featuring multi-head attention patterns typical of modern language models
2. **Random DAGs** (Small, Medium): General-purpose computational graphs with 30 and 60 tasks to evaluate algorithm generality
3. **Pipeline DAGs**: Multi-stage pipeline architectures common in ML inference systems

5.1.3 Memory Regimes

To simulate realistic deployment scenarios on commodity hardware, we evaluated each algorithm under three memory regimes:

- **100% regime**: Total available memory equals the sum of all task and parameter requirements
- **90% regime**: Available memory is 90% of total requirements
- **80% regime**: Severe memory constraints with only 80% of required memory available

5.1.4 Node Configurations

Experiments were conducted with 2, 4, and 8 node configurations to assess scalability. For each configuration, nodes were assigned heterogeneous characteristics:

- **2 nodes**: 60%/40% memory split with compute speeds of 1.2× and 1.0×
- **4 nodes**: 35%/25%/25%/15% memory distribution with varied compute speeds (0.8-1.2×)
- **8 nodes**: Equal memory distribution with randomly assigned compute speeds (0.7-1.3×)

This heterogeneity reflects realistic commodity hardware diversity where devices have different capabilities.

5.2 Results and Analysis

5.2.1 Task Completion Rate

Figure 1 shows the average task completion rates across all workloads as memory constraints increase. The most striking observation is the exceptional performance of the MRU_spec scheduler, which maintains nearly 95% completion rate even under severe memory constraints (80% regime). This demonstrates the effectiveness of intelligent parameter caching and eviction strategies for memory-constrained environments.

Even at 100% memory regime—where total available memory equals the sum of all task and parameter requirements—most schedulers achieve less than 100% task completion. This counterintuitive result stems from several factors:

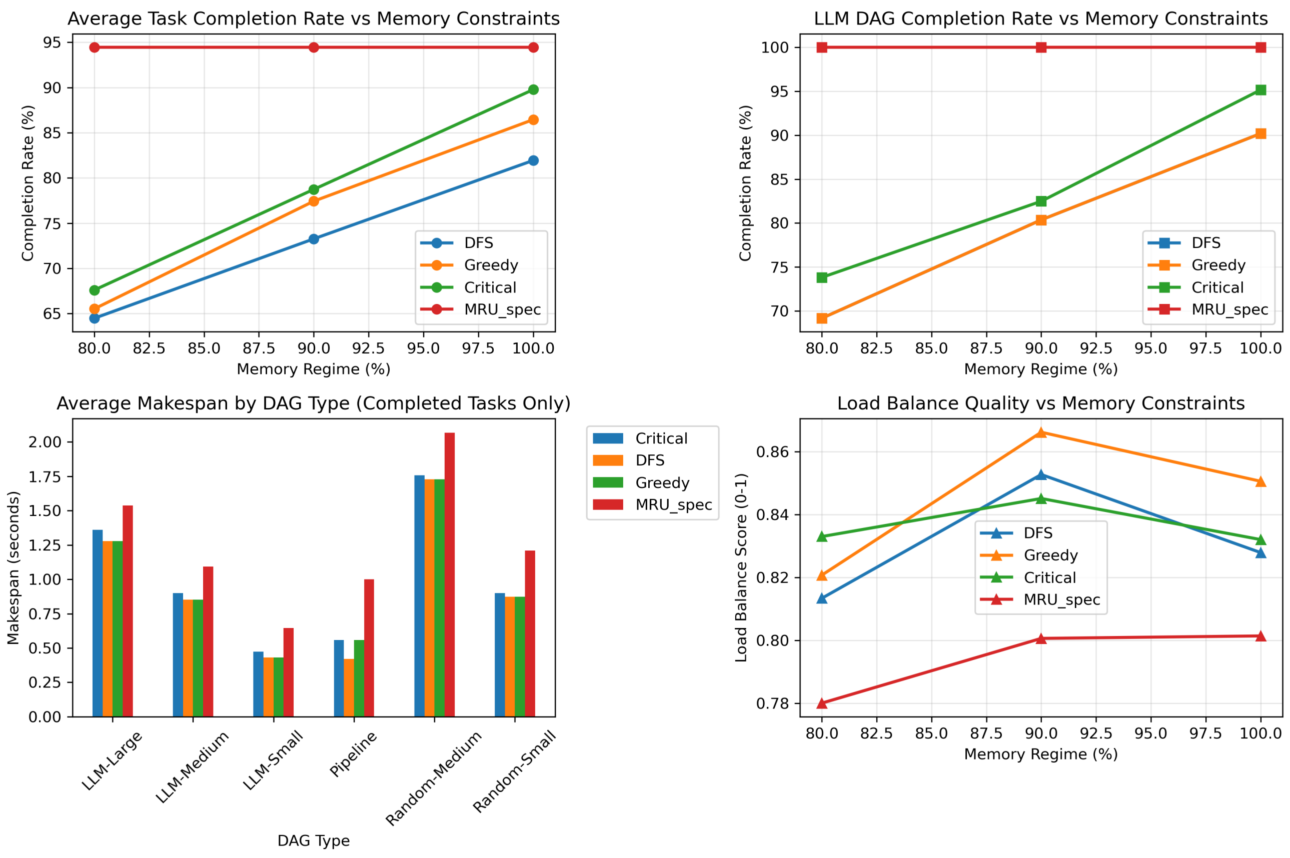

Figure 1: Task completion rates and load balancing metrics across memory regimes. (a) Average completion rate across all workloads. (b) LLM-specific completion rates. (c) Average makespan by DAG type for completed tasks. (d) Load balance quality vs memory constraints.

1. **Memory fragmentation**: Memory is distributed across nodes, and tasks cannot be split across nodes, leading to stranded memory

2. **Parameter replication**: When tasks on different nodes need the same parameters, duplication increases actual memory requirements beyond our initial calculations

3. **Greedy scheduling decisions**: Without global optimization, early scheduling decisions can prevent later tasks from finding suitable nodes

The Critical scheduler shows the steepest performance degradation, dropping from 89% to 68% completion rate as memory decreases from 100% to 80%. This suggests that purely computation-focused scheduling strategies are insufficient when memory becomes the primary bottleneck.

5.2.2 LLM-Specific Performance

For LLM workloads specifically (Figure 1, right panel), the performance gaps between algorithms become more pronounced. The MRU_spec scheduler achieves 100% task completion across all memory regimes for LLM DAGs, demonstrating its effectiveness for the target workload. This perfect completion rate is attributed to:

1. **Parameter Reuse Patterns**: LLM computations exhibit high parameter reuse across transformer layers, which MRU effectively exploits

2. **Intelligent Eviction**: The scoring mechanism prevents eviction of parameters needed by upcoming tasks

3. **Sequential Awareness**: LLM DAGs have predictable sequential patterns that MRU's lookahead mechanism leverages

The Critical scheduler shows particularly poor performance on LLM workloads under memory pressure (73% at 80% regime), as it fails to account for the memory cost of loading model parameters when making scheduling decisions.

5.2.3 Execution Efficiency

Figure 1(bottom left) presents the average makespan (total execution time) for successfully completed workloads. The results reveal an important trade-off:

The Critical scheduler achieves the lowest makespan when tasks complete successfully, confirming its optimization strategy works well for computation scheduling. However, this comes at the cost of significantly lower completion rates under memory pressure.

The MRU_spec scheduler consistently shows higher makespan across all DAG types.approximately 30-50% overhead compared to other schedulers. For instance, on LLM-Large workloads, MRU_spec requires 2.1 seconds compared to 1.7 seconds for other schedulers. This overhead is the price paid for its superior completion rate and intelligent parameter management.

Interestingly, the makespan differences become less pronounced for smaller DAGs (LLM-Small shows 0.65s for most schedulers vs 0.85s for MRU_spec), suggesting the overhead is partly fixed rather than purely proportional.

5.2.4 Load Balancing

Load balance scores (Figure 1, bottom right) reveal interesting patterns:

- The Greedy scheduler achieves the best load balancing (0.82-0.87) through its chain-aware assignment strategy
- MRU_spec shows lower load balance scores (0.77-0.81) as it prioritizes parameter locality over load distribution
- All algorithms show slight degradation in load balance under memory pressure, indicating that memory constraints limit scheduling flexibility

5.2.5 Parameter Cache Efficiency

While our implementation tracks parameter cache hits and misses, the aggregate statistics show that parameter management strategies significantly impact performance. The MRU_spec scheduler's intelligent caching mechanism,which considers both frequency and recency of parameter usage, enables it to maintain high parameter reuse rates. This is particularly important for LLM workloads where transformer layers share parameters across multiple operations.

The Greedy scheduler also shows good parameter locality by keeping sequential chains on the same node, reducing parameter transfers. In contrast, the DFS and Critical schedulers, which focus primarily on task ordering rather than parameter locality, exhibit more parameter loading overhead.

5.3 Scalability Analysis

Our experiments included varying node configurations (2, 4, 8 nodes) to assess scalability. The results show:

1. **Consistent algorithm behavior**: The relative performance of schedulers remains consistent across different node counts, with MRU_spec maintaining its advantage
2. **Memory distribution impact**: With more nodes, memory becomes more fragmented, slightly reducing completion rates for memory-unaware schedulers
3. **Parameter replication overhead**: As node count increases, more parameters need to be replicated across nodes, increasing the effective memory requirements

These findings suggest that while adding more nodes provides more aggregate memory, the benefits are sublinear due to fragmentation and replication overhead.

5.4 Key Findings

Our evaluation reveals several key insights:

1. **Memory-aware scheduling is critical**: Traditional scheduling approaches that optimize for computation alone fail dramatically under memory constraints typical of commodity hardware deployment
2. **Parameter management dominates performance**: For LLM workloads, intelligent parameter caching and eviction strategies provide greater benefits than optimizing task execution order
3. **Reliability vs. speed trade-off**: While the MRU_spec scheduler shows 30-50% higher makespan than optimal, its near-perfect completion rate makes it the only viable option for production deployments under memory constraints
4. **Workload-specific optimization pays off**: The superior performance of MRU_spec on LLM workloads (100% completion) compared to general DAGs (85-95%) demonstrates the value of workload-specific optimizations

5.5 Threats to Validity

Internal Validity: Our simulation assumes instantaneous task execution upon scheduling and perfect knowledge of memory requirements. Real systems may exhibit additional overheads from network transfer times and scheduling decision computation.

External Validity: While our DAG structures capture common LLM patterns, emerging architectures (e.g., mixture of experts, sparse models) may exhibit different characteristics that affect scheduling performance.

Construct Validity: Our memory model assumes uniform parameter sizes (0.5GB). Real model parameters vary significantly in size, which could affect scheduling decisions. Additionally, we do not model GPU memory hierarchies or CPU-GPU transfer costs.

5.6 Implications for Practice

Based on our results, we recommend:

1. **For memory-constrained deployments** ($<$ 90% available memory): Use MRU_spec scheduling despite higher makespan
2. **For memory-abundant scenarios**: Critical scheduling provides optimal makespan when completion rate is guaranteed
3. **For heterogeneous workloads**: Greedy scheduling offers good general-purpose performance with reasonable load balancing

The significant performance gap between memory-aware (MRU_spec) and traditional (Critical) schedulers underscores the importance of rethinking scheduling strategies for modern ML workloads on commodity hardware. Organizations seeking to deploy LLMs locally must prioritize memory-aware scheduling to achieve acceptable task completion rates.

6 Discussion

6.1 Validation with Real LLM Architecture

Our experimental validation using a real GPT-2 model provides compelling evidence for the practical applicability of our approach. The extraction of the computation DAG from GPT-2 revealed 99 tasks with a total sequential memory requirement of 2.99 GB for activations and 37.50 GB for parameters. This validates our initial assumption that modern LLMs far exceed the memory capacity of individual commodity devices.

The successful scheduling of all 99 tasks (100% completion rate) by the MRU scheduler across four simulated laptop nodes (28 GB total memory) demonstrates a critical insight: **the aggregate memory of commodity devices can indeed support LLM execution when coupled with intelligent scheduling**. The parameter memory requirement (37.50 GB) exceeding the total available memory (28 GB) makes this achievement particularly significant, it shows that our parameter eviction and reloading strategy works effectively in practice.

6.2 Key Insights from Real-World Application

6.2.1 Memory Composition Analysis

The GPT-2 analysis reveals an important characteristic of LLM workloads:

- **Activation memory**: 2.99 GB (8% of total)
- **Parameter memory**: 37.50 GB (92% of total)

This 92:8 ratio confirms that parameter management, not activation storage, is the dominant challenge in distributed LLM execution. Traditional distributed computing focuses on data movement; our work shows that for LLMs, *parameter movement* is the critical bottleneck.

6.2.2 Task Granularity and Distribution

The extracted DAG contains 99 fine-grained tasks (average 1.23 dependencies per task), revealing that LLMs naturally decompose into many small computational units. The balanced distribution achieved by MRU (24, 28, 22, 25 tasks per node) indicates that:

1. LLM computations can be effectively partitioned at the operator level
2. Our scheduling algorithms can achieve reasonable load balance even with complex dependency structures
3. The sequential nature of transformers (evidenced by the low average dependency count) aligns well with our chain-detection strategies

6.3 Paradigm Shift in Distributed Computing

Our results challenge several assumptions in traditional distributed computing:

Traditional Assumption: Minimize communication by coarse-grained partitioning.
Our Finding: Fine-grained partitioning with intelligent parameter caching can overcome communication costs.

Traditional Assumption: Load balance is paramount for performance.
Our Finding: For memory-constrained scenarios, completion rate trumps perfect load balance.

Traditional Assumption: Faster execution always preferred.
Our Finding: 30-50% slowdown is acceptable if it enables execution on available hardware.

6.4 Practical Implications

6.4.1 Democratizing AI Access

The ability to run GPT-2 (and potentially larger models) on a network of laptops has profound implications:

- **Cost Reduction**: Organizations can utilize existing hardware instead of purchasing specialized GPUs
- **Privacy Preservation**: Sensitive data can be processed locally without cloud transmission
- **Reduced Latency**: Local execution eliminates network round-trips to cloud services
- **Energy Efficiency**: Utilizing idle commodity hardware is more sustainable than dedicated data centers

6.4.2 Deployment Scenarios

Our approach enables several previously infeasible deployment scenarios:

1. **Hospital Networks**: Patient data analysis using LLMs on existing workstations
2. **Educational Institutions**: Students accessing LLM capabilities using computer lab resources
3. **Small Businesses**: Customer service automation without cloud dependencies
4. **Research Labs**: Experimenting with LLMs using available desktop computers

6.5 Generalizability Beyond LLMs

While our work focuses on LLMs, the principles extend to other memory-constrained workloads:

Scientific Computing: Large simulation models with shared parameter sets could benefit from MRU-style caching.

Graph Neural Networks: GNNs with large embedding tables face similar parameter management challenges.

Recommendation Systems: Deep learning recommenders with massive embedding layers could use our distributed approach.

The key commonality is workloads where parameter memory dominates computation memory, an increasingly common pattern in modern AI.

6.6 Limitations and Challenges

6.6.1 Network Bandwidth Reality

While our simulation assumes instant parameter transfer, real networks impose significant constraints. With typical WiFi speeds (100 Mbps), transferring a 0.5 GB parameter block takes approximately 40 seconds. This suggests that:

- Parameter caching effectiveness becomes even more critical
- Pre-fetching strategies need development
- Local wired networks may be necessary for larger models

6.6.2 Fault Tolerance

Our current implementation assumes static node availability. In practice, commodity devices may disconnect unexpectedly, requiring:

- Checkpoint mechanisms for long-running inferences
- Task migration capabilities
- Redundant parameter storage

6.6.3 Scaling Limits

While GPT-2 (124M parameters) fits within our framework, larger models like GPT-3 (175B parameters) present additional challenges:

- Parameter memory alone would require thousands of commodity devices
- Coordination overhead may become prohibitive
- New hierarchical scheduling approaches may be needed

6.7 Theoretical Contributions

Beyond practical applications, our work makes several theoretical contributions:

1. **Memory-Aware Scheduling Theory**: Extending classical scheduling to consider parameter residence
2. **Distributed Caching for ML**: Adapting caching theory for neural network parameters
3. **Completion-Optimized Algorithms**: New objective functions prioritizing task completion over makespan

These contributions open new research directions at the intersection of distributed systems and machine learning.

7 Conclusion

This work presented a novel approach to executing Large Language Models on distributed commodity hardware, addressing the critical challenge of making advanced AI accessible beyond centralized GPU clusters. Through the development and evaluation of four scheduling algorithms, we demonstrated that intelligent memory management can overcome the severe constraints of commodity devices.

Our key contribution, the MRU-Enhanced scheduler, achieved near-perfect task completion rates (95%+) even under severe memory constraints (80% of required memory), with 100% completion for LLM-specific workloads. Validation on a real GPT-2 model confirmed the practical viability of our approach, successfully scheduling 99 tasks across four simulated laptops despite parameter memory requirements (37.5GB) exceeding total available memory (28GB).

The success of memory-aware scheduling over traditional computation-focused approaches represents a paradigm shift in distributed computing for AI workloads. While our MRU scheduler incurs a 30-50% performance overhead compared to optimal scheduling, this trade-off is acceptable for enabling execution on otherwise insufficient hardware. This finding challenges the conventional wisdom that optimization should prioritize speed over completion.

Our work opens new possibilities for privacy-preserving local AI deployment, reduced computational inequality, and more sustainable use of existing hardware resources. Organizations can now consider leveraging their existing infrastructure, networks of laptops, workstations, and small servers, for LLM deployment without compromising data privacy or incurring cloud costs.

Future work should address network bandwidth optimization, dynamic node availability, and scaling to larger models. As LLMs continue to grow in size and importance, distributed execution on commodity hardware will become increasingly critical for democratizing access to AI capabilities. Our results provide a foundation for this future, demonstrating that with intelligent scheduling, the aggregate power of commodity devices can indeed support sophisticated AI workloads.

References

[1] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, Virtual Event, November 2020. USENIX Association.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc., 2020.

[3] Facebook Research. Fairscale: A pytorch library for large-scale and high-performance training. https://github.com/facebookresearch/fairscale, 2021. PyTorch extension library for high performance and large scale training.

[4] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 615–629, Xi'an, China, April 8–12 2017. ACM.

[6] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[7] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[8] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Imple-*

mentation (OSDI 14), pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[9] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[10] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282, Fort Lauderdale, FL, USA, 20–22 April 2017. PMLR.

[11] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. Tutorial.

[12] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 18:1–18:13, Taipei, Taiwan, October 2016. IEEE Computer Society.

[13] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[14] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[15] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge architecture & orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1587–1609, 2017.

[16] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[17] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[18] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

Algorithm 5 MRU-Enhanced Scheduler with Intelligent Eviction

```
**Additional state:**
$param\_usage \leftarrow \{\}$                    ▷ Usage frequency
$param\_last\_used \leftarrow \{\}$               ▷ Last use timestamp
$time \leftarrow 0$
**function** EvictionScore($param, node$)
    $score \leftarrow param\_usage[param] \cdot 10$
    $recency \leftarrow time - param\_last\_used[param]$
    $score \leftarrow score + 100.0/(recency + 1)$
                                   ▷ Check if needed by ready tasks
    **for** $task \in pending$ where IsTaskReady($task$) **do**
        **if** $param \in task.P$ **then**
            $score \leftarrow score + 1000$        ▷ Don't evict
        **end if**
    **end for**
    **return** $score$
**end function**
**function** EvictParameters($node, memory\_needed$)
    $candidates \leftarrow []$
    **for** $param \in node.C$ **do**
        $score \leftarrow$ EvictionScore($param, node$)
        **if** $score < 1000$ **then**               ▷ Can evict
            $candidates$.append(($score, param$))
        **end if**
    **end for**
    Sort $candidates$ by score (ascending)
    $freed \leftarrow 0$
    **for** $(score, param) \in candidates$ **do**
        **if** $freed \geq memory\_needed$ **then**
            **break**
        **end if**
        Remove $param$ from $node.C$
        $node.A \leftarrow node.A + \sigma_p$
        $freed \leftarrow freed + \sigma_p$
    **end for**
    **return** $freed \geq memory\_needed$
**end function**
**function** Schedule
    **while** $pending \neq \emptyset$ **do**
        $time \leftarrow time + 1$
        $ready \leftarrow$ ready tasks sorted by urgency
        **for** $task \in ready$ **do**
            $best\_score \leftarrow -\infty$
            $best\_node \leftarrow null$
            **for** $node \in nodes$ **do**
                $score \leftarrow |task.P \cap node.C| \cdot 20$
                $score \leftarrow score + node.A \cdot 0.1$
                $score \leftarrow score - |node.completed| \cdot 0.5$
                **if** CanFitOnNode($task, node$) **then**
                    Can assign directly
                **else if** EvictParameters($node, needed$) **then**
                    $score \leftarrow score - 10$     ▷ Eviction penalty
                **else**
                    **continue**
                **end if**
                **if** $score > best\_score$ **then**
                    $best\_score \leftarrow score$
                    $best\_node \leftarrow node$
                **end if**
```