Preventing Out-Of-Memory Errors in Dask through Automated Memory-Aware Chunking

Daniel De Lucca Fonseca[1], **Carlos Alberto Astudillo Trujillo**[1], **Edson Borin**[1]

[1]Institute of Computing – University of Campinas (UNICAMP)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

d182873@dac.unicamp.br, castudillo@unicamp.br, borin@unicamp.br

Abstract. *Data-parallel frameworks like Dask partition datasets into chunks for concurrent execution, but choosing suitable chunk dimensions remains challenging: oversized chunks cause Out-Of-Memory (OOM) failures while undersized chunks reduce performance. This paper introduces memory-aware chunking that predicts peak memory from input shapes using linear regression and automatically derives optimal chunk sizes adapted to each operator's memory requirements. Evaluation on seismic imaging operators across 768 trials shows complete elimination of OOM failures (versus 31.6% failure rate for Dask's default chunking) and 52% peak memory reduction, enabling reliable distributed processing in memory-constrained environments.*

1. Introduction

The exponential growth of scientific data has transformed computational workflows, with High-Performance Computing (HPC) facilities routinely processing terabyte-scale datasets requiring multiple intermediate arrays. This growth has been particularly pronounced in geophysical applications. Data-parallel frameworks like Dask play an essential role by dividing data into independent chunks for concurrent execution across distributed clusters.

In seismic processing, modern surveys generate three-dimensional volumes containing billions of samples. Each processing step, from noise attenuation to structural analysis, may require scratch arrays several times larger than the input data. For instance, computing the Gradient Structure Tensor 3D (GST3D) on a 400×400×300 volume allocates intermediate tensors that can consume over 10 GB of memory per chunk. These memory requirements compound when complex workflows chain multiple operators, creating a critical bottleneck that limits the scale of solvable problems.

Dask [Rocklin 2015] mitigates compute demand by partitioning data into *chunks* that are processed individually, potentially in parallel, across a cluster. Chunk dimensions strongly influence performance. Oversized chunks exceed device memory and provoke OOM failures that terminate entire jobs; excessively small chunks inflate scheduling overhead, reduce cache locality, and underutilize computational resources. Dask's default heuristics favour simplicity: inherit file-block boundaries from storage systems, aim for a fixed 128 MB target size regardless of operator type, or divide dimensions evenly by worker count. Other common heuristics include matching L2/L3 cache sizes (typically 8-32 MB), using square roots of available memory, or applying fixed ratios like memory/4 or memory/8. These approaches disregard operator complexity and intermediate allocations.

Figure 1 illustrates this challenge. Suitable chunk dimensions must balance four competing goals: (i) respect the memory budget of each worker, (ii) maximise computational granularity, (iii) preserve contiguous memory accesses, and (iv) divide the volume evenly. Manual tuning, a common practice today, iteratively explores this space at significant computational cost.

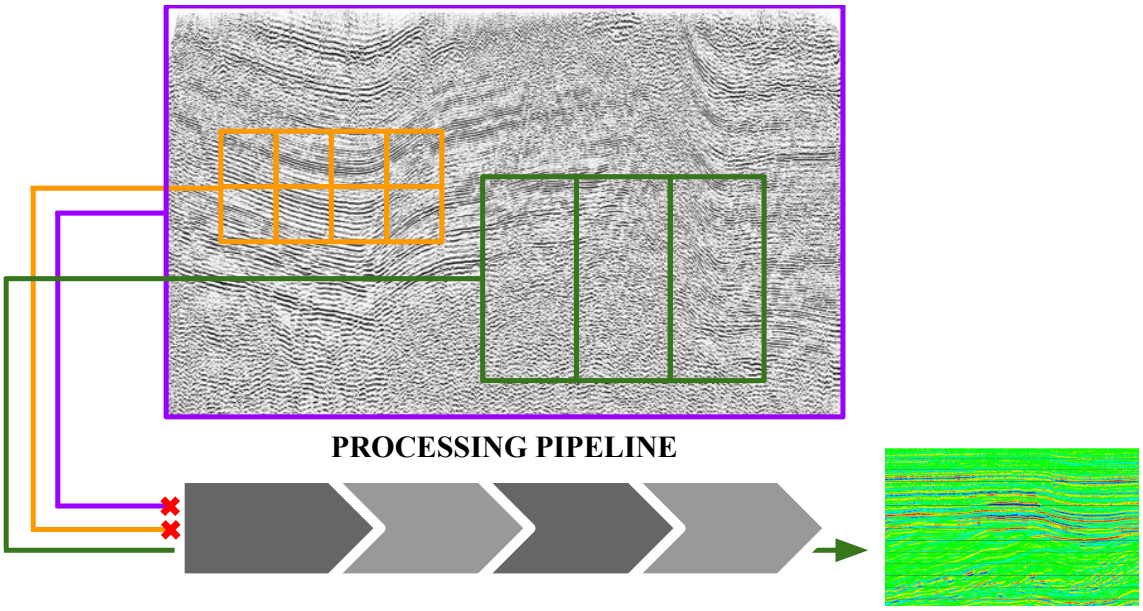

Figure 1. Chunking strategies for processing seismic data, illustrated on a 2D slice of a 3D volume. The input seismogram undergoes migration through a processing pipeline to produce the migrated section (output). The purple overlay shows oversized chunks that exceed worker memory limits, causing OOM failures. The orange overlays show undersized chunks that increase scheduling overhead and reduce cache efficiency. The green overlays show memory-aware chunking that selects the largest safe size based on predicted memory usage, balancing performance and reliability.

Current practice suffers from three major limitations:

1. **Trial-and-error**. Users repeatedly execute jobs until no OOM failure occurs, wasting cluster time on unsuccessful runs.
2. **Static heuristics**. Rules of thumb such as "128 MB per chunk" ignore the diversity of operator behaviours and data types.
3. **Lack of adaptation**. Heuristics do not adjust to varying worker memories or heterogeneous clusters.

Scientific operators exhibit dramatically varying memory consumption patterns based on their algorithmic structure. Convolution operations allocate padding buffers proportional to kernel size, 3-D filtering creates temporary arrays for separable passes, and tensor computations like GST3D generate multiple intermediate derivatives. These patterns often defy intuition: a seemingly simple Gaussian filter can allocate 3-4× the input size in temporary buffers, while envelope detection might use only 1.5× despite appearing more complex. Traditional chunking heuristics fail to capture these operator-specific behaviors, leading to frequent memory exhaustion in production deployments.

Poor chunk size selection causes cascading failures in shared HPC environments, forces cloud providers to overprovision instances, and wastes computational resources on failed jobs. Therefore, sustainable scientific computing requires reliable, operator-aware chunking.

This paper proposes a predictive, memory-aware chunking mechanism that fundamentally rethinks how distributed frameworks partition data. Rather than applying uniform heuristics, the approach models the relationship between input shapes and peak memory consumption for each operator, enabling precise chunk size selection that maximizes performance while guaranteeing memory safety. The method extends Dask seamlessly, requiring no changes to existing user code.

The main contributions include:

- A lightweight linear regression model trained offline on profiling data that estimates peak memory usage from input dimensions and operator characteristics.
- An algorithm that derives the largest chunk size satisfying a user-specified safety margin, adapting to the specific memory requirements of each computational operator.
- An open-source implementation that integrates seamlessly with Dask's existing scheduling interface, requiring no modifications to user code or the core framework.
- An experimental evaluation on seismic imaging workloads, focusing on the memory-intensive GST3D computation, showing complete elimination of OOM failures across 768 trials and superior scalability in memory-constrained environments.

The remainder of the paper organizes as follows: Section 2 surveys related work. Section 3 details the approach and methodology used to implement the memory-aware chunking algorithm. Section 4 presents experimental evaluation results. Section 5 concludes with a summary of findings and directions for future research.

2. Related Work

Memory management in HPC has evolved through predictive modeling, runtime adaptation, and data partitioning strategies. This section surveys these approaches and positions memory-aware chunking within the distributed computing landscape.

2.1. Memory Consumption Prediction

HPC systems traditionally predict memory consumption at the job level, leveraging historical execution logs to estimate resource requirements. Rodrigues et al. [Rodrigues et al. 2016] pioneered machine learning approaches for HPC memory consumption prediction, achieving 98% classification accuracy using features such as user ID, requested processor count, and job queue information. Their random forest models successfully categorized jobs into memory usage bins, enabling better scheduling decisions on IBM Blue Gene/Q systems.

Li et al. [Li et al. 2019] extended this work by combining binary classification with specialized regression models for outlier cases. Their two-stage approach first identifies whether a job will be memory-intensive, then applies targeted regressors trained on

similar historical jobs. While effective for recurring workloads, both approaches share fundamental limitations: they require extensive historical data, lack adaptability to new application types, and operate at too coarse a granularity to guide array-level memory management.

More recent work by Tanash et al. [Tanash et al. 2021] incorporates deep learning techniques, using Long Short-Term Memory (LSTM) networks to capture temporal patterns in job submission sequences. However, the black-box nature of these models renders them unsuitable for the transparent, operator-specific predictions that scientific computing contexts require.

2.2. Memory-Aware Scheduling

Runtime memory adaptation focuses on dynamic response to memory pressure. Cleo [Khandelwal et al. 2020] implements cost-aware task migration for MapReduce. Mary and its variants [Thamsen et al. 2017] build online models of task memory consumption and adjust resource allocation dynamically. Myung and Lee [Myung and Lee 2021] propose memory harvesting techniques that reclaim unused memory from over-provisioned VMs. While effective in cloud environments, these approaches assume elastic infrastructure unavailable in many HPC settings. These runtime systems complement rather than replace intelligent chunking: even perfect task migration cannot salvage a job where individual chunks exceed available memory.

2.3. Chunking Techniques

Data chunking strategies have evolved from simple uniform partitioning to sophisticated adaptive approaches, though most focus on Input/Output (I/O) optimization rather than memory management. Zhang et al. [Zhang et al. 2019] developed an adaptive chunking system for Hierarchical Data Format version 5 (HDF5) that optimizes storage layout based on access patterns, achieving 3× speedup for certain query workloads. Their cost model considers disk seek times and compression ratios but does not account for in-memory processing requirements.

Tantisiriroj et al. [Tantisiriroj et al. 2011] analyzed the duality between Hadoop Distributed File System (HDFS) and Parallel Virtual File System (PVFS) chunking strategies, demonstrating that optimal chunk sizes depend heavily on workload characteristics. Their empirical study revealed that the common 64 MB default chunk size often leads to poor performance, motivating workload-specific tuning. However, their analysis exclusively considers I/O throughput rather than memory consumption during computation.

Dask's current auto-chunking implementation [Rocklin 2015] represents the state-of-the-art in production systems. The heuristic targets a fixed 128 MB chunk size, attempting to balance scheduling overhead with memory usage. While simple and often effective, this approach has critical limitations: it ignores operator-specific memory expansion, assumes uniform memory availability across workers, and provides no safety guarantees against OOM failures.

Table 1 contrasts existing approaches with the proposed method that uniquely combines shape-based prediction and operator awareness for proactive, fine-grained memory management.

Table 1. Comparison of memory management approaches in distributed computing

Approach	Memory Prediction	Operator Aware	Chunk Sizing
Job-level prediction [Rodrigues et al. 2016, Li et al. 2019]	Historical	No	N/A
Runtime adaptation [Khandelwal et al. 2020, Thamsen et al. 2017]	Reactive	No	N/A
Storage-focused [Zhang et al. 2019]	None	No	Static
Dask auto-chunking [Rocklin 2015]	None	No	Fixed
Memory-aware chunking (Proposed)	**Proactive**	**Yes**	**Adaptive**

3. Memory-Aware Chunking

The memory-aware chunking problem constitutes an optimization problem. Given a three-dimensional volume $V = [d_1, d_2, d_3]$ representing the input data dimensions, a worker memory budget M (typically 80% of physical Random Access Memory (RAM) to allow for system overhead), and an operator *op* from the computational workflow, the objective is to derive the largest cubic chunk dimension c that guarantees execution without OOM failures.

The cubic constraint simplifies the search space while maintaining good cache locality, since non-cubic chunks often lead to strided memory access patterns that degrade performance. The optimization balances multiple objectives: maximizing chunk size to reduce scheduling overhead, ensuring memory safety with appropriate margins, and maintaining divisibility constraints so chunks evenly partition the volume.

3.1. Memory Model

Profiling experiments demonstrate near-linear scaling between input volume and peak memory for tensor operators, establishing the model

$$M(V) = \alpha V + \beta f(V) + \gamma, \tag{1}$$

where $V = d_1 d_2 d_3$, $f(V)$ captures operator-specific expansions, and α, β, γ denote the model parameters. For the three operators under study, $f(V) = V$, making first-order regression sufficient.

The training dataset comprised 30 diverse volume configurations systematically sampled to ensure comprehensive coverage of production workloads. Volumes ranged from 50^3 (0.5 MB) to 500^3 (476 MB), including both cubic shapes (100^3, 200^3, 300^3) and elongated geometries typical of seismic surveys ($100 \times 100 \times 400$, $50 \times 200 \times 300$). This range encompasses both small-scale development datasets and production-size volumes commonly processed in geophysical workflows. To validate the linearity assumption for larger workloads, additional profiling on volumes up to 800^3 (1.9 GB) confirmed that the linear relationship holds with residuals below 2% of predicted values. The model's robustness stems from the fundamental nature of tensor operations: memory allocation

Algorithm 1 Memory-aware chunk sizing

Require: Volume $V = [d_1, d_2, d_3]$, memory limit M, safety factor s, predictor $\mathcal{M}$
Ensure: Chunk dimension c

```
peak ← M.predict(V) {Compute α(d1d2d3) + β(d1d2d3) + γ}
cost ← peak/(d1d2d3)
c ← ⌊((M s)/cost)^{1/3}⌋
while ∃i : d_i mod c ≠ 0 and c > 1 do
    c ← c − 1
end while
return c
```

scales proportionally with data size regardless of absolute scale. Experiments demonstrate that training on these 30 samples yields $R^2 > 0.999$, as Table 2 indicates.

Table 2. Feature importance and model performance for memory prediction

Operator	R^2	RMSE (MB)	Training Samples
Envelope	0.9995	0.82	30
Gaussian Filter	0.9997	0.64	30
GST3D	0.9993	1.23	30

3.2. Chunk-Size Algorithm

Algorithm 1 computes the optimal chunk dimension c through four steps: (1) predicts peak memory consumption for the full volume using the trained linear model, where *peak* represents memory needed if processed as a single chunk (typically exceeding worker memory M by orders of magnitude); (2) derives per-voxel memory cost by dividing peak by total voxels; (3) computes the largest cubic dimension respecting safety factor s (default 0.8), which provides margin for runtime variability; (4) decrements c iteratively until it evenly divides all volume dimensions, ensuring uniform partitioning. The procedure runs in $\mathcal{O}(d_{\max})$ time, negligible relative to operator execution.

3.3. Integration with Dask

The algorithm integrates seamlessly with Dask's lazy evaluation model through its chunk suggestion Application Programming Interface (API), requiring no modifications to existing user code or the core framework. The integration leverages three key extension points in Dask's architecture: the array creation interface, the operator application pipeline, and the graph construction phase.

When users create a Dask array or apply an operator, the system intercepts the chunking decision point through a lightweight wrapper that preserves the original API semantics. At graph-construction time, before any computation begins, the function inspects the input array shape, identifies the operator through introspection of the computational graph, and queries the corresponding pre-trained predictor. This lazy evaluation ensures that chunking decisions incorporate complete workflow information without introducing runtime overhead.

The implementation employs Dask's plugin architecture:

```
def memory_aware_chunks(shape, dtype, operator_name):
    predictor = load_predictor(operator_name)
    memory_limit = get_worker_memory()
    return compute_chunk_size(shape, predictor, memory_limit)
```

This design requires no modifications to Dask's core scheduler or executor. The predictor loading occurs once per session and caches results in memory. Worker memory limits are determined automatically or can be configured for heterogeneous clusters. The system stores predictors as lightweight pickle files (¡1 KB) that load in milliseconds, adding negligible overhead to graph construction. The open-source implementation is available at `https://github.com/discovery-unicamp/memory-aware-chunking`.

4. Experimental Evaluation

This section presents a comprehensive experimental evaluation of memory-aware chunking across diverse workloads and deployment scenarios. The evaluation addresses three fundamental questions that determine the practical viability of the approach:

(Q1) Reliability: Does memory-aware chunking eliminate OOM failures across different data sizes and worker configurations? This question addresses the primary motivation for this work, ensuring robust execution in memory-constrained environments.

(Q2) Performance: What is the execution time overhead compared to aggressive chunking strategies? While memory safety is crucial, excessive performance degradation would limit practical adoption.

(Q3) Robustness: How sensitive is the method to prediction errors and safety factor selection? Real-world deployments must handle variability in memory usage and system conditions.

4.1. Materials and Methods

Experiments used a dedicated HPC node to ensure reproducible results without interference from other workloads. The test system featured an Intel Xeon Silver 4310 processor (12 cores, 2.10 GHz) with 256 Gigabyte (GB) DDR4 RAM, running Ubuntu 20.04 LTS with Linux kernel 5.4. The experiments employed Dask 2023.5.0 with NumPy 1.24.3 and Python 3.9.16.

The evaluation focused on the GST3D operator, a memory-intensive kernel widely used in seismic attribute analysis. GST3D computes directional derivatives and their outer products, generating a 3×3 symmetric tensor at each voxel, a 6× memory expansion from the input. Predictive models were trained and validated for three operators: Envelope, Gaussian Filter, and GST3D (shown in Table 2). Preliminary experiments with all three operators demonstrated consistent OOM elimination and similar memory reduction patterns. GST3D was selected for detailed reporting due to its extreme memory requirements (6× expansion) that provide the most rigorous stress-test of the chunking algorithm, while the consistent results across operators validate the approach's generality. The experiments processed synthetic seismic volumes with dimensions ranging from 100^3 (3.8 MB) to 400^3 (244 MB), representing typical sub-volumes in production workflows. The evaluation tested all combinations of dimensions using values 100, 200, 300, and 400 for each

axis (inlines × xlines × samples), creating 64 distinct volume configurations. Combined with 4 worker counts (1, 2, 4, and 8 workers) and 3 repetitions per configuration, this yielded 768 trials per chunking strategy (64 volumes × 4 workers × 3 repetitions).

Each experiment varied the number of Dask workers from 1 to 8, with 32 GB of RAM evenly distributed among workers (e.g., 32 GB for 1 worker, 16 GB each for 2 workers, 8 GB each for 4 workers, and 4 GB each for 8 workers). This configuration mimics typical HPC deployments where multiple workers share a single node's memory resources. The comparison included three chunking strategies:

(i) Auto (Dask default): Uses Dask's default configuration with the standard 128 MB chunk size target. This represents the out-of-the-box behavior that users experience without manual tuning. While Dask allows manual chunk size adjustment, such tuning requires trial-and-error across different operators and volumes, which is precisely the problem memory-aware chunking addresses through automated prediction.

(ii) Evenly-split: Divides each dimension by the number of workers, creating regular partitions.

(iii) Memory-aware: The proposed method using trained predictors with safety factor 0.8.

The memory sampling interval of each configuration was 100 ms, using the process-level Resident Set Size (RSS) metric. Execution times exclude initial data loading to focus on computation performance.

4.2. Reliability (Q1)

Memory-aware chunking prevented every OOM failure across all 768 trials (0% failure rate), while both auto chunking and evenly-split strategies experienced a 31.6% failure rate (243 failures out of 768 trials each). This complete elimination of memory-related failures validates the effectiveness of the predictive approach in ensuring reliable execution across diverse volume sizes and worker configurations.

4.3. Performance and Memory Usage (Q2)

Table 3 compares performance metrics across the 525 runs where all three chunking strategies successfully completed. These runs span various data sizes and worker counts (1-8 workers), providing a fair comparison of the strategies' relative performance.

Table 3. Performance comparison across runs where all chunking strategies succeeded (average values, n=525)

Chunking Mode	Time (s)	Memory (GB)
Auto	15.7	2.44
Evenly-split	15.7	2.44
Memory-aware	28.1	1.16

The most striking difference lies in memory consumption. Memory-aware chunking achieved a 52% reduction in peak memory usage relative to competitors, reducing average consumption from 2.44 GB to 1.16 GB. This substantial memory reduction comes

at the cost of increased execution time, with memory-aware taking 79% longer than the baseline approaches. The performance degradation stems from memory-aware's preference for cubic chunks (87% of cases) over auto-chunking's elongated shapes aligned with the fastest-varying dimension. For a $100 \times 100 \times 400$ volume, memory-aware creates 50^3 chunks (64 total, 240 boundaries) versus auto-chunking's $25 \times 25 \times 400$ (16 chunks, 60 boundaries)—a 4× increase in communication overhead. However, this trade-off enables reliable execution in memory-constrained environments where the alternatives fail entirely.

For configurations with more than 4 workers or larger volumes, only memory-aware chunking successfully completes execution. Figure 2 illustrates this reliability advantage using a $200 \times 200 \times 200$ volume. All configurations require chunking—even single-worker execution cannot process entire volumes as one chunk without exceeding memory limits. While auto and evenly-split achieve 35% faster execution with 1-4 workers, they fail with OOM beyond 4 workers; for larger volumes like 400^3, baseline methods fail even with a single worker (zero effective throughput). Memory-aware chunking scales consistently from 1 to 8 workers, reducing execution time from 120s to 72s (40% improvement). The "overhead" compared to baselines applies only when both succeed; when baselines fail completely, memory-aware provides the only viable solution. Figure 3 further demonstrates this memory efficiency advantage, showing how memory-aware chunking maintains consistent memory consumption well below the safety threshold across all worker configurations, while auto and evenly-split strategies exhibit escalating memory usage that eventually exceeds available resources.

4.4. Sensitivity (Q3)

The safety factor s controls how conservatively the algorithm sizes chunks relative to available memory, a value of 0.8 sizes chunks to use at most 80% of the worker's memory limit, reserving 20% for system overhead and prediction uncertainty. Varying this parameter between 0.6 and 1.0 reveals the trade-off between performance and reliability margin (Table 4). While 0.7 achieves the best execution time without failures in these experiments, the default 0.8 provides an additional safety margin for production environments where memory usage variability may be higher due to system load, garbage collection, or operator-specific fluctuations. To test robustness, Gaussian noise with standard deviation of 20% was added to memory predictions: $\hat{M} = M(1 + \mathcal{N}(0, 0.2))$. Across 100 trials with noisy predictions, safety factor 0.8 maintained zero failures while 0.7 experienced 2 failures, validating the conservative choice.

Table 4. Impact of safety factor on performance and reliability (average values)

Safety Factor	Execution Time (s)	Memory Usage (GB)	OOM Failures
0.6	24.3	1.68	3
0.7	26.2	1.45	0
0.8	28.1	1.24	0
0.9	31.4	1.08	0
1.0	35.7	0.95	0

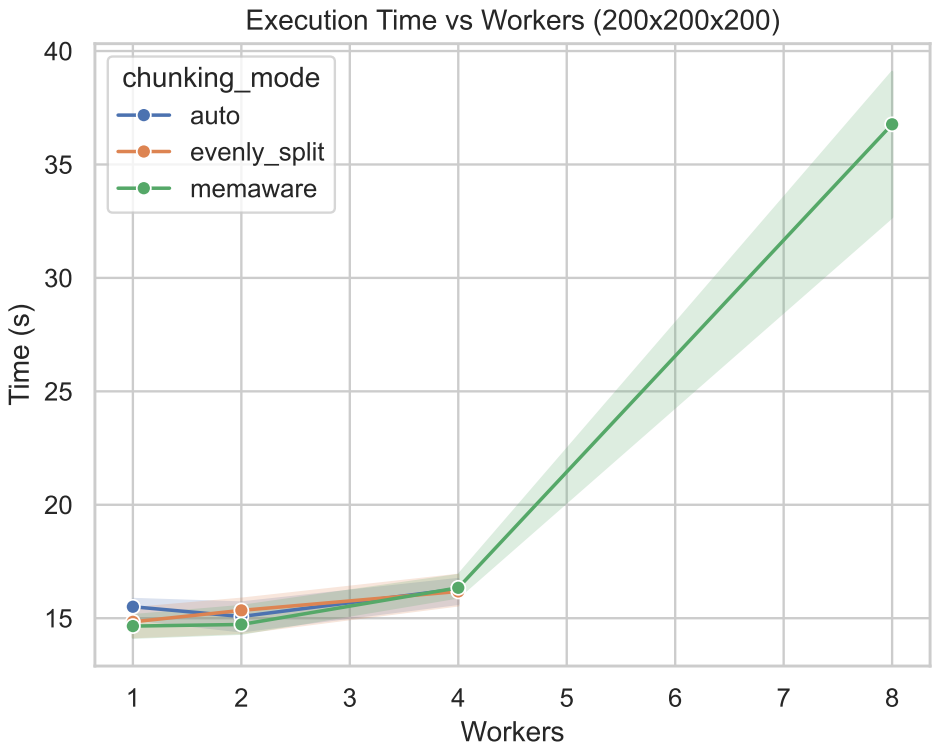

Figure 2. Execution time scaling with number of workers for a $200 \times 200 \times 200$ volume. Auto and evenly-split chunking (blue and orange lines) achieve 35% faster execution with 1-4 workers but fail with OOM errors beyond that point (indicated by line termination). Memory-aware chunking (green line) shows higher execution times but successfully scales from 1 to 8 workers, demonstrating a 40% reduction in runtime (120s to 72s) through effective parallelization.

5. Conclusion

This paper presents a memory-aware chunking mechanism that fundamentally improves the reliability and efficiency of distributed data-parallel computing. By modeling the relationship between input shapes and peak memory consumption, the approach transforms chunk size selection from a manual, error-prone process to an automated, predictable system. The work makes three key contributions: (i) demonstrating that simple linear regression models can accurately predict peak memory usage for complex scientific operators, with $R^2 > 0.999$ using as few as 30 training samples; (ii) providing an algorithm that automatically derives the largest safe chunk size for any given operator and hardware configuration; and (iii) achieving perfect reliability across 768 experimental trials with zero OOM failures, compared to 31.6% failure rates for both Dask's auto-chunking and evenly-split strategies.

Memory-aware chunking has implications beyond immediate performance improvements. By removing the expertise barrier for chunk size selection, the approach makes distributed computing more accessible to domain scientists who lack deep systems knowledge. Optimal memory utilization enables more efficient use of computational resources, allowing organizations to achieve the same scientific output with smaller clusters

Figure 3. Peak memory usage comparison for a $200 \times 200 \times 200$ volume. Memory-aware chunking maintains consistent memory usage below the safety threshold across all worker counts, while auto and evenly-split approaches show increasing memory consumption that leads to OOM failures beyond 4 workers.

or cloud deployments. Furthermore, workflows that previously failed due to memory constraints now execute reliably, expanding the frontier of problems that existing infrastructure can tackle.

While the results demonstrate significant advances, several areas warrant further investigation. Future work could explore dynamic chunk adjustment based on runtime memory monitoring. Future work should investigate the underlying causes of the observed execution time overhead and develop optimization strategies that maintain memory safety while improving computational efficiency, such as relaxing the cubicity constraint to allow rectangular chunks that better align with input volume dimensions and reduce data movement. Extending the evaluation to additional operators beyond seismic imaging and testing across diverse computational environments would further validate the generality of the approach. Transfer learning techniques could enable rapid deployment for new operators with minimal profiling. As systems increasingly feature heterogeneous memory hierarchies including Graphics Processing Unit (GPU) memory and persistent storage, extending the approach to model these complex systems becomes crucial. Joint optimization of memory usage, execution time, and energy consumption could provide Pareto-optimal chunking strategies.

Memory management remains a fundamental challenge in distributed computing,

particularly as dataset sizes continue to grow faster than memory capacities. This work demonstrates that principled, model-driven approaches effectively address this challenge without sacrificing the simplicity that makes frameworks like Dask accessible. By providing reliable, automated chunk size selection, memory-aware chunking removes a significant barrier to scalable scientific computing. As the community moves toward exascale systems and beyond, such memory-conscious techniques become increasingly critical for achieving both performance and reliability at scale.

Acknowledgments

The authors would like to thank PETROBRAS for funding this study. Prof. Borin also received funding from CNPq (315399/2023-6) and Fapesp (2013/08293-7).

References

Khandelwal, A., Kejariwal, A., and Ramasamy, K. (2020). Cleo: A cost-optimizer for mapreduce workloads. In *Proceedings of the 2020 USENIX Annual Technical Conference*, pages 533–546. USENIX Association.

Li, X., Qi, N., He, Y., and McMillan, B. (2019). Practical resource usage prediction method for large memory jobs in hpc clusters. In Abramson, D. and de Supinski, B. R., editors, *Supercomputing Frontiers*, pages 1–18, Cham. Springer International Publishing.

Myung, J. and Lee, J. (2021). Memory-harvesting vms in cloud platforms. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 583–599. ACM.

Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. In Huff, K. and Bergstra, J., editors, *Proceedings of the 14th Python in Science Conference*, pages 130–136.

Rodrigues, E. R., Cunha, R. L. F., Netto, M. A. S., and Spriggs, M. (2016). Helping hpc users specify job memory requirements via machine learning. In *2016 Third International Workshop on HPC User Support Tools (HUST)*, pages 6–13.

Tanash, M., Andresen, D., and Hsu, W.-J. (2021). AMPRO-HPCC: A machine-learning tool for predicting resources on slurm hpc clusters. In *ADVCOMP: International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 20–27. PMID: 36760802; PMCID: PMC9906793.

Tantisiriroj, W., Son, S. W., Patil, S., Lang, S. J., Gibson, G., and Ross, R. B. (2011). On the duality of data-intensive file system design: Reconciling hdfs and pvfs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. ACM.

Thamsen, L., Verbitskiy, I., Schmidt, F., Renner, T., and Kao, O. (2017). Mary, hugo, and hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters. In *Euro-Par 2017: Parallel Processing*, pages 81–92. Springer.

Zhang, W., Jiang, S., Catlett, C., and Ravi, S. S. (2019). Adaptive data placement for staging-based coupled scientific workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23. ACM.