

Low-Intrusion Profiling and Scalability Analysis in Distributed-Memory Environments using PaScal Suite

Felipe H. Santos-da-Silva¹, Ítalo A. S. Assis², João B. Fernandes¹,
Igor S. F. Correia¹, Samuel Xavier-de-Souza¹

¹Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte (UFRN), Rio Grande do Norte, Brasil

²Departamento de Engenharias e Tecnologia
Universidade Federal Rural do Semi-Árido (UFERSA), Rio Grande do Norte, Brasil

{felipe.santos.087, igor.sergio.017}@ufrn.edu.br, samuel@dca.ufrn.br
joao.batista.fernandes.094@ufrn.edu.br, italo.assis@ufersa.edu.br

Abstract. *The continuous expansion of computational power in High-Performance Computing (HPC) systems has not been accompanied by a proportional increase in the scalability of general parallel applications. Traditional profiling tools provide detailed diagnostics for individual executions but often generate large trace volumes and require substantial manual effort when scalability must be evaluated across multiple configurations. To address this gap, this work extends the Parallel Scalability Suite to natively support distributed-memory applications. The proposed architecture adopts the Sidecar pattern to decouple metric management from the main execution flow, using Unix Domain Sockets for low-latency local communication and deferring transmission to a central Orchestrator until application completion. The solution was evaluated using dense matrix multiplication and the miniFE finite element mini-application under multiple scalability configurations. Results show low measurement intrusion while preserving the scalability trends observed through conventional analysis.*

1. Introduction

The computational capacity of High-Performance Computing (HPC) systems continues to increase; however, outside highly optimized numerical kernels such as dense linear algebra, there is no strictly proportional increment in the scalability of general parallel applications. Developing software capable of efficiently exploiting these computational resources in large-scale environments is essential, which drives the research potential in this area [Weber et al. 2019] [Islam et al. 2019].

Existing profiling and visualization tools play a key role in HPC by enabling developers to understand application behavior and identify bottlenecks that compromise code efficiency [Adhianto et al. 2024, Zhukov and Wylie 2014, Knüpfer et al. 2012]. Traditionally, however, these tools are primarily designed for in-depth diagnostics of individual executions. Although such approaches provide detailed information at the microarchitectural and interprocess levels, exhaustive trace and event generation may result in large volumes of data and complicate the extraction of global performance trends.

Scalability analysis instead requires comparing and synthesizing performance data across multiple input sizes and computational resource configurations [Pacheco 2021, Bergel et al. 2019, da Silva et al. 2019]. Because most profiling tools focus on single-run diagnostics, developers often need to manually structure and manage multiple experiments to infer such trends from raw data [Thürigen et al. 2021]. To address this operational gap, the Parallel Scalability (PaScal) Suite [Silva et al. 2025] was developed to automate multi-execution measurement and visual analysis.

The original architecture of the PaScal Suite was restricted to shared-memory applications. As the main contribution of this work, we extend PaScal to natively support distributed-memory applications. The developed solution adopts a Sidecar-based architecture [Huf and Siqueira 2019] that decouples metric management from the application execution flow and enables automated orchestration of multiple experiments with low runtime intrusion. The framework, publicly available at the official PaScal repository¹, is evaluated using both a dense matrix multiplication benchmark and the open-source miniFE finite element mini-application.

The remainder of this paper is organized as follows: Section 2 presents the theoretical background, while Section 3 contextualizes the related work; Section 4 describes the proposed architecture; Section 5 evaluates its runtime intrusion and scalability analysis capabilities; and Section 6 summarizes the main conclusions and future directions.

2. Background

This section outlines the core concepts and architectural foundations required to understand the proposed extension. It addresses parallel programming models, scalability assessment metrics, the original PaScal Suite ecosystem, and the principles behind the Sidecar pattern.

2.1. Scalability Assessment Metrics

Evaluating parallel performance relies on Speedup ($S = T_{\text{serial}}/T_{\text{parallel}}$) and Parallel Efficiency ($E = S/p$, where p represents the processor count) [Pacheco 2021]. Scalability reflects an application’s ability to maintain efficiency as computational resources scale: strong scalability measures efficiency retention when increasing p for a fixed problem size, whereas weak scalability evaluates efficiency when scaling problem size proportionally with p .

2.2. PaScal Suite

PaScal Suite is an integrated set of tools designed to automate measurement, data collection, and visualization of scalability trends in parallel applications [Silva et al. 2025]. Its workflow allows users to evaluate multiple combinations of input sizes and processing units while maintaining low measurement overhead. The suite consists primarily of the PaScal Analyzer [da Silva et al. 2022] and PaScal Viewer [da Silva et al. 2019].

PaScal Analyzer automates application execution, instrumentation, and timing metric collection. In its original implementation, it supports shared-memory C/C++ programs based on OpenMP or POSIX Threads. Instrumentation can be performed automatically by intercepting parallel runtime calls through a preloaded library or manually using

¹<https://gitlab.com/lappsufrn/pascal-releases>

paired `pascal_start(id)` and `pascal_stop(id)` markers, where static integer IDs delimit named regions of interest across single or iterative invocations.

PaScaI Viewer [da Silva et al. 2019] translates data collected by PaScaI Analyzer into graphical representations of scalability. Its color-map-based visualization allows developers to identify efficiency patterns across workloads and processing configurations and supports hierarchical analysis of monitored application regions.

2.3. Sidecar Pattern

Sidecar pattern is a software engineering abstraction focused on the design of distributed and concurrent systems, defining a topology in which an auxiliary process runs co-located and coupled to the lifecycle of a main application. The fundamental goal of this pattern is the separation of concerns, isolating cross-cutting concerns from the system’s central computational logic [Burns and Oppenheimer 2016]. By operating on the same computational node, the auxiliary process shares the local execution domain but maintains an independent address space, inspecting and extending the system’s behavior in a modular manner.

Although widely adopted in cloud-based microservices architectures, Sidecar’s principles of task isolation and local communication align perfectly with the rigorous requirements of HPC. In MPI-based environments, traditional profiling and tracing mechanisms introduce severe runtime intrusions, leading to I/O over-billing that distorts scalability results [Xuan et al. 2025]. The conceptual transposition of Sidecar to the HPC ecosystem allows the entire burden of management, temporary persistence, and handling of performance metrics to be delegated to a secondary process, underpinning the architecture proposed in this work and detailed in Section 4.

3. Related Work

Performance analysis in HPC has traditionally relied on tools designed for detailed diagnostics of individual executions. HPCToolkit [Adhianto et al. 2010, Adhianto et al. 2024] uses asynchronous sampling of optimized binaries, while instrumentation-based ecosystems such as Scalasca [Zhukov and Wylie 2014] and Score-P [Knüpfer et al. 2012] collect execution information for MPI and OpenMP applications. Paraver [Jost et al. 2004], Vampir [Nagel et al. 1996], and TAU [Shende and Malony 2006, Huck et al. 2025] similarly provide detailed measurement and visualization infrastructures. Although these approaches offer substantial diagnostic detail, extensive trace generation can produce large data volumes and impose significant I/O requirements.

Recent approaches have therefore focused on reducing data volume and automating performance interpretation. Caliper [Boehme et al. 2016] associates semantic annotations with runtime performance metrics, while TinyProf [Fan et al. 2024] employs in-memory structures and specialized I/O strategies to generate compact profiles. Scalable visualization techniques have also been explored, including the scatter-plot approach proposed by Muelder et al. [Muelder et al. 2009] for identifying communication patterns in large-scale executions.

Other works seek to automate scalability diagnosis, statistical validation, or prediction. Touati et al. [Touati et al. 2013] formalize median-based statistical methodologies for speedup evaluation across repeated runs, SCALANA [Jin et al. 2025] combines

static program information with lightweight dynamic profiling to identify performance bottlenecks, whereas Calotoiu et al. [Calotoiu et al. 2013] derive empirical models from performance measurements to predict scalability behavior.

These approaches address detailed diagnosis, data reduction, visualization, or scalability modeling. However, scalability assessment in practical HPC development still requires orchestrating multiple executions across different problem sizes and resource configurations. The PaScal Suite was designed specifically to automate this workflow, but its original implementation was restricted to shared-memory applications. The extension presented here expands the PaScal Analyzer to natively support distributed-memory MPI applications.

4. Low-intrusion Sidecar-based MPI Profiling for PaScal Suite

The architecture developed in this work extends the PaScal Suite ecosystem to natively support MPI-based distributed memory environments. Aiming to reduce temporal intrusion during data collection, the infrastructure was completely restructured using an in-memory aggregation and deferred transmission strategy based on the Sidecar architectural pattern.

The operational workflow is illustrated in Fig. 1. A wrapper script launches both the target MPI application and the associated PaScal Sidecar daemon. One Sidecar instance is created per physical compute node, establishing a many-to-one topology in which all MPI ranks executing locally transmit their metrics to the same auxiliary process.

During application execution, the PaScal Analyzer library intercepts and extracts metadata only at the boundaries of configured regions rather than recording exhaustive MPI event traces. Instead of immediately writing these records to disk or transmitting them through the cluster network, the library communicates locally with the Sidecar using Unix Domain Sockets. Because each message carries only compact binary metadata (region ID, rank, thread ID, and start/stop timestamps—tens of bytes per region invocation), the Sidecar temporarily stores the received data in memory with a minimal local RAM footprint of a few kilobytes or megabytes.

To handle concurrent MPI ranks, the Sidecar uses a thread-pool architecture and centralized buffering. Incoming connections are managed through the Unix Domain Socket queue, while individual data streams are processed concurrently. Access to the shared buffer is synchronized and data are consolidated into batches before final transmission.

Only after the parallel application completes does each node-level Sidecar daemon flush and transmit its consolidated buffer over the network via dynamic TCP/IP sockets to the central PaScal Analyzer Orchestrator. To avoid network bottlenecks and handle simultaneous incoming connections from all distributed compute nodes, the Orchestrator instantiates a concurrent thread pool. Each incoming stream is managed by an asynchronous worker thread that deserializes raw binary records back into high-level measurement structures, preserving metadata such as region identifiers, process ranks, thread IDs, and high-resolution start and stop timestamps.

As these streams arrive, the Orchestrator routes the records to an in-memory ag-

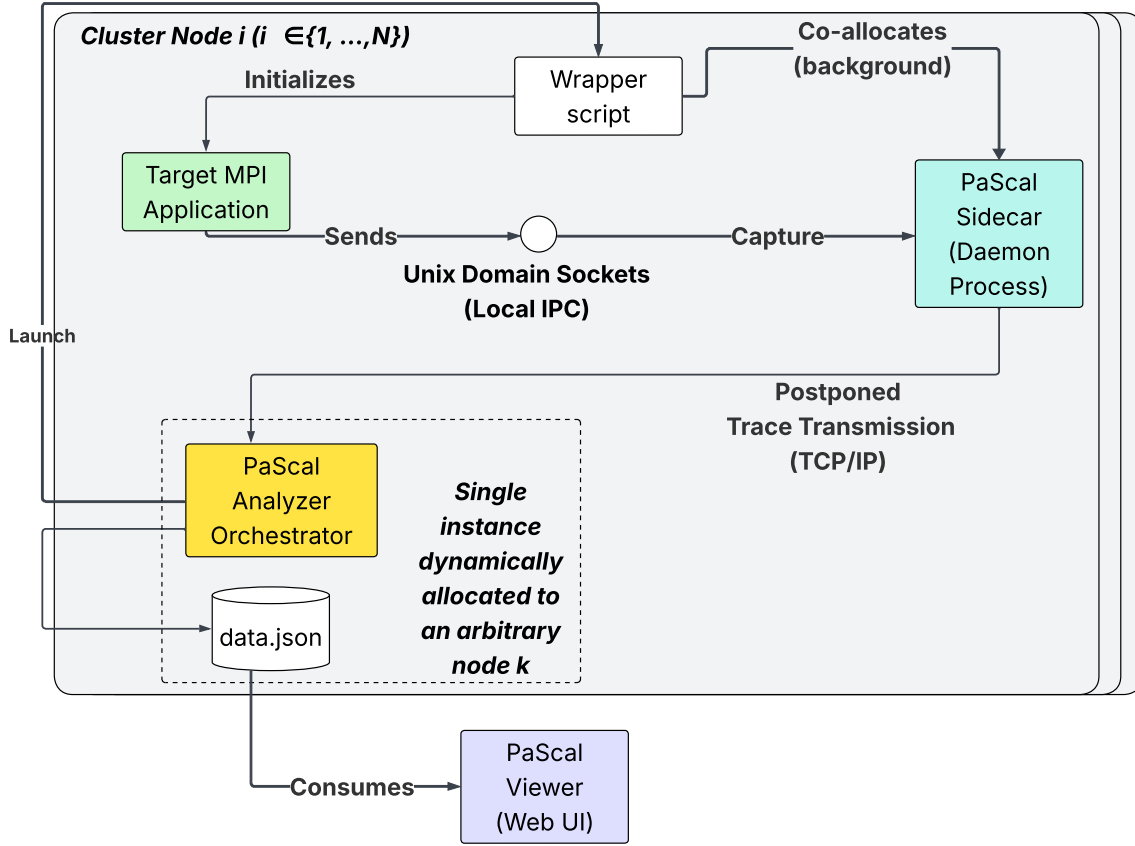


Figure 1. Architecture of the PaScal Analyzer extension using the Sidecar pattern in a distributed-memory HPC cluster, illustrating local node telemetry and the centralized orchestrator dynamic allocation.

gregation engine to consolidate metrics across all executing processes. The aggregation mechanism first performs region-wise grouping, unifying all discrete measurements that share identical hierarchical markers into a single internal structure. Next, the engine extracts the global temporal boundaries across participating ranks by determining the earliest initiation timestamp among all processes ($\min(t_{\text{start}})$) and the slowest completion timestamp ($\max(t_{\text{stop}})$), which captures the global execution span and reflects synchronization delays or load imbalances. Concurrently, the total active time across ranks is iteratively accumulated ($\sum \Delta t$) alongside the sample count, providing the statistical basis required to compute aggregate means and deviations. Finally, this unified in-memory representation is formatted and emitted into the structured `data.json` file consumed by the PaScal Viewer.

PaScal Viewer was also extended to complement changes introduced in PaScal Analyzer and to process results from shared-memory and distributed-memory applications in an integrated manner. PaScal Viewer interprets the parallel configurations recorded in the output file and associates each measurement with its workload, monitored region, and amount of parallel resources. Measurements are then organized into matrices relating workloads to parallel configurations, while repeated executions of the same configuration are aggregated using the median to reduce the influence of occasional runtime variations [Touati et al. 2013].

From these matrices, PaScal Viewer computes and presents execution time,

speedup, parallel efficiency, scalability, speedup loss, load imbalance, and energy-related metrics. The resulting heatmaps relate workloads to the corresponding parallel configurations, allowing performance trends and bottlenecks to be identified across different resource levels.

Therefore, PaScal Analyzer and PaScal Viewer extensions address complementary stages of PaScal Suite’s workflow. PaScal Analyzer collects and consolidates measurements from executions with different memory organizations, whereas PaScal Viewer interprets these measurements and produces visualizations compatible with the corresponding parallel configurations. This integration preserves analysis by region, workload, and resource configuration while enabling comparisons across shared-memory and distributed-memory scenarios.

5. Experiments, Results, and Discussion

This section describes the experimental environment and target applications and evaluates the runtime intrusion and scalability analysis capabilities of the proposed extension.

All experiments were conducted at the High-Performance Computing Center (NPAD) of the Federal University of Rio Grande do Norte (UFRN), using 8 homogeneous compute nodes (the maximum dedicated allocation per user queue to guarantee isolated nodes and network paths across repeated trials). Each node contains two Intel Xeon Sixteen-Core E5-2698v3 processors (dual-socket) and 128 GB of DDR4 RAM at 2133 MHz. The environment uses Linux, GCC 8.3.0, and OpenMPI 4.1.1. The MPI ranks were distributed across up to 8 physical compute nodes using a mapping of 2 processes per node (one rank per physical CPU socket/NUMA domain) to prevent intra-socket memory bandwidth contention and force inter-node network communication starting at 4 ranks, directly stressing the distributed Sidecar and TCP/IP Orchestrator architecture.

As a benchmark application to validate the framework, we used a dense matrix multiplication algorithm ($C = A \times B$) parallelized via MPI. The algorithm employs a row-block data decomposition strategy, distributing the computational load evenly among processes through collective communication routines. To maximize performance and mitigate spurious implementation noise, the code was optimized using cache-referenced locality ($I - K - J$), `restrict` qualifiers for explicit vectorization, and strict 64-byte memory alignment. Matrix dimensions (4096, 5792, 8192, and 16384) were selected in geometric progression to scale the computational workload across process configurations.

Figure 2 shows the application’s abstract structure and the mapping of the three regions of interest monitored by the PaScal Analyzer. The baseline scenario consists of exactly this same logical structure, but without the `pascal_*` directives.

```

Start_MPI();
Allocate_64-Byte_Aligned_Memory();
Synchronize_Ranks(); // MPI_Barrier
pascal_start(0);      // Total Execution Time
MPI_Bcast(Matrix_B);
MPI_Scatter(Matrix_A, Local_Submatrix_A);
Synchronize_Ranks();
pascal_start(1);      // Pure Computation
Multiply_Cache-Friendly_Submatrices();
pascal_stop(1);
Synchronize_Ranks();
pascal_start(2);      // Result Collection
MPI_Gather(Local_Submatrix_C, Total_Matrix_C);
pascal_stop(2);
Synchronize_Ranks();
pascal_stop(0);
Finish_MPI();

```

Figure 2. Logical flow of the matrix multiplication application and insertion points for PaScal Suite analyzer.

To demonstrate the tool’s applicability in more complex, unstructured scientific workloads, we also evaluated miniFE [Heroux et al. 2009], an open-source proxy application from the Sandia Mantevo benchmark suite that approximates the performance of unstructured implicit finite element codes. miniFE solves a sparse linear system using the Conjugate Gradient (CG) algorithm across four cubic 3D meshes selected in geometric progression (128^3 , 203^3 , 256^3 , and 322^3 , corresponding to $\approx 2.1\text{M}$, 8.4M , 16.8M , and 33.4M elements, i.e., $1\times$, $4\times$, $8\times$, and $16\times$ workload scaling). The instrumentation was configured to monitor two primary regions: Region 1 for the Setup and Assembly phase (which includes unstructured mesh generation, matrix initialization, and global boundary exchanges), and Region 2 for the CG Solver phase (which dominates computational time through sparse matrix-vector multiplications with point-to-point halo exchanges, vector updates, and `MPI_Allreduce` dot products).

5.1. Intrusion Assessment

To assess the temporal intrusion of PaScal Analyzer, runtime overhead experiments were conducted by executing the target applications across four problem sizes and process counts varying from 1 to 16. The primary objective is to quantify the overhead introduced by the sidecar monitoring architecture compared to the native, uninstrumented baseline.

Figure 3a presents the relative overhead of the PaScal suite when profiling the Matrix Multiplication benchmark across 10 samples for each matrix size configuration (4096, 5792, 8192, and 16384) and number of ranks. The baseline scenario maps the native execution purely in MPI without any instrumentation or modification, whereas the instrumented scenario reflects the application transparently monitored by the developed extension.

As seen in the empirical results, the overhead generated by the tool remained consistently below 1% in practically all tested configurations, demonstrating excellent stability even with the substantial growth in data volume and communication infrastructure. In certain scenarios, the execution time of the instrumented code recorded values marginally lower than the baseline, falling within the dynamic noise of the execution environment and effectively characterizing a zero intrusion impact.

A similar evaluation was conducted for the miniFE benchmark, as shown in Figure 3b. The execution time measurements for both pure (baseline) and PaScaI-instrumented executions were obtained from the total execution time reported by miniFE’s YAMl outputs. For most grid sizes and evaluated process counts, the overhead introduced by PaScaI remained below 2.6%. Small overhead peaks occurred for the smaller grid size (128^3) at parallelism levels of 8 and 16 processes, reaching a maximum overhead of 4.84%. This occurs because the absolute execution time per rank on the 128^3 grid at 8 and 16 processes becomes very short (< 2 s), causing the fixed local socket initialization/teardown latency and MPI synchronization jitter to represent a slightly larger relative fraction of total runtime. These results demonstrate the decoupling of the instrumentation workload provided by the PaScaI Analyzer sidecar model, allowing the analysis of iterative solvers with unstructured memory access patterns.

The analysis indicates that the relative standard deviation remained below 1% of execution time across most scenarios. Minor variations of up to 2.5% in specific cases are attributed to routine HPC background noise and network contention. Overall, these results confirm that the low overhead levels measured are consistent and statistically reliable.

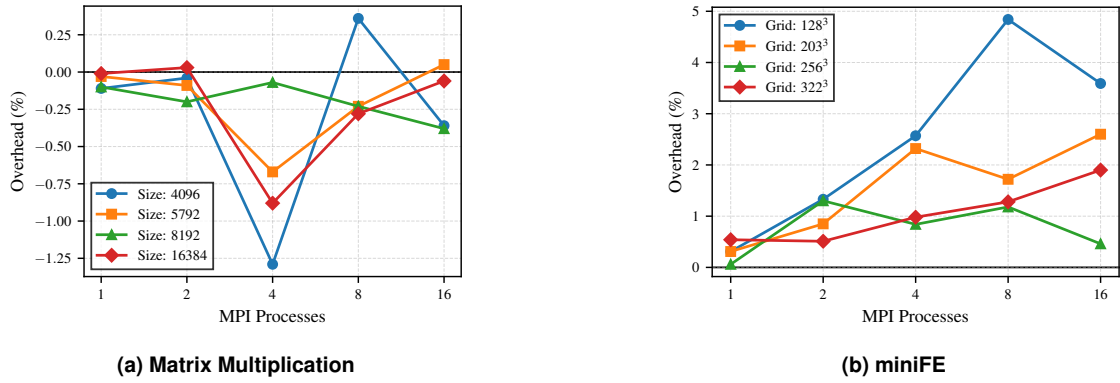


Figure 3. Relative execution overhead introduced by the PaScaI analyzer compared to the pure baseline for (a) Matrix Multiplication and (b) miniFE benchmarks.

5.2. Dense Matrix Multiplication Scalability Analysis

Using the application illustrated in Figure 2 and under the same experimental configuration adopted in previous tests, scalability and profiling tests were conducted comparing the native execution and the execution instrumented with the PaScaI Suite. For the manual approach, the execution times of each delimited region were obtained through direct calls to the `MPI_Wtime()` function.

Figure 4 presents the total execution time (average of 10 samples) for the native application, varying the number of MPI processes from 1 to 16 under different matrix dimensions. It is evident that, in all scenarios evaluated, the pure computation time accounts for the vast majority of the total time (T_{total}). Although the visualization shows a significant reduction in overall time as the number of ranks increases, the communication component remains negligible, demonstrating the solution’s low communication overhead.

Based on the execution times collected separately for each region and the total execution time, the speedup and parallel efficiency were calculated for each process configu-

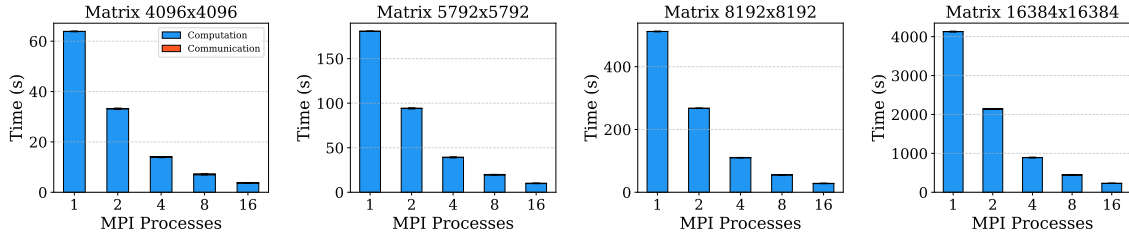


Figure 4. Breakdown of total execution time by number of MPI ranks for the native application. Each data point represents the average of 10 samples.

ration, as explained in Section 2.1. In this setup, Region 1 corresponds to pure local computation (`Multiply-Cache-Friendly-Submatrices`), whereas Region 2 monitors the result gathering phase via `MPI-Gather`. To analyze isolated code blocks, the PaScal Suite evaluates regional scaling by taking the single-rank execution time ($p = 1$) of each block as its reference baseline.

Figure 5 illustrates the resulting speedup curves. Both the total speedup and Region 1 increase proportionally to the number of allocated processes, exhibiting near-linear behavior. Conversely, Region 2 shows negligible speedup across all configurations; because collective communication incurs increasing network synchronization and data transfer overhead as more processes are added, its execution time does not decrease compared to the single-rank baseline.

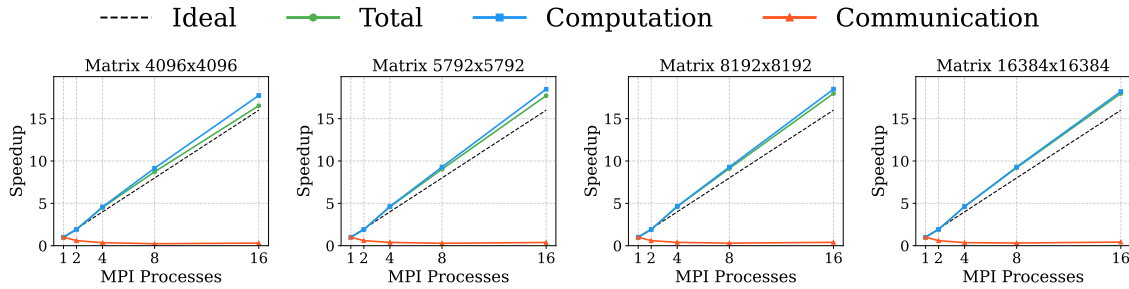


Figure 5. Total speedup curves for the computation region (Region 1) and the communication region (Region 2).

In addition, parallel efficiency was extracted for each region and process configuration, as shown in Figure 6. Starting at 4 processes, the application maintains an overall efficiency and an efficiency for Region 1 above 100%, confirming that the computational kernel exhibits strong scalability. In contrast, Region 2 exhibits a severe drop in efficiency starting as early as 2 processes, reflecting the network bottleneck.

Using the same application and the same hardware parameters and problem sizes, the experiment was conducted under the orchestration of the PaScal Suite. To automate the management, execution, and collection of the multiple rounds, the PaScal Analyzer module was used via the following terminal command:

```
pascalanalyzer "main" \
  --ranks 1,2,4,8,16 \
  -r 10 \
  -i 4096,5792,8192,16384 \
  -t man \
```

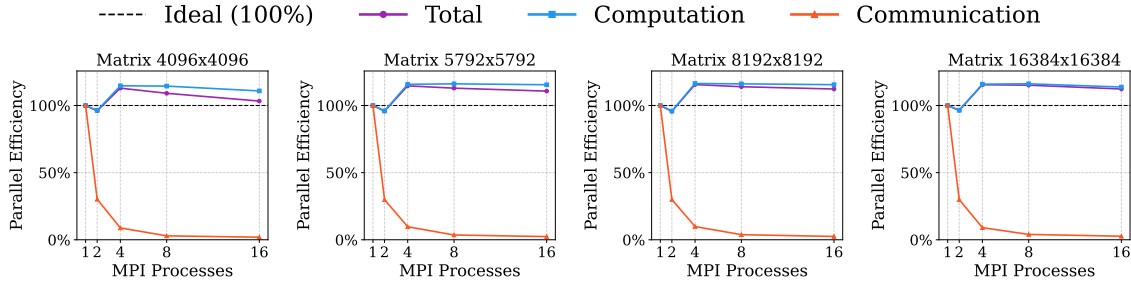


Figure 6. Parallel efficiency analysis highlighting the linear behavior in the computational phase and the bottleneck in communication.

```
--mpi mpirun \
-o output_pascal.json
```

In the command mentioned above, the `--ranks` parameter defines the list of processes that will be varied with each run, `-r` specifies the number of repetitions per configuration for statistical analysis, and `-t man` indicates that the source code was instrumented manually. Finally, the argument `--mpi mpirun` specifies the MPI environment launcher, and `-o` maps the structured output file to be generated.

After automated execution via Analyzer, the output file `output_pascal.json` was submitted directly to PaScaL Viewer for the generation and analysis of performance visualizations, the graphical results of which are shown in Figure 7.

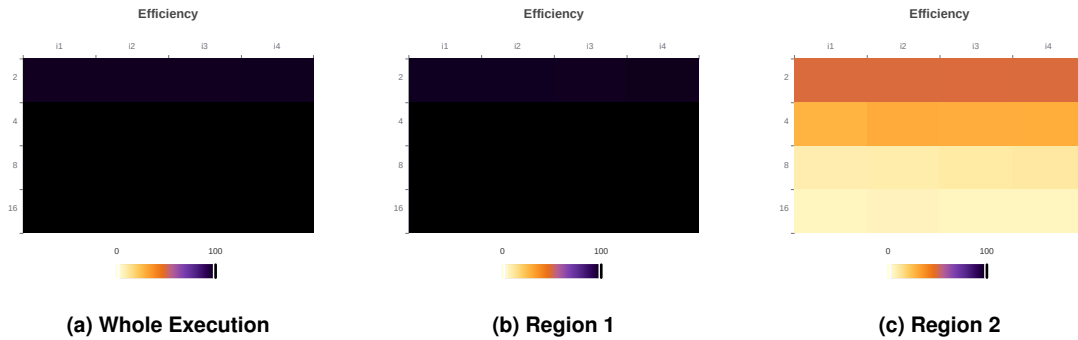


Figure 7. Parallel efficiency graphs generated by the PaScaL Viewer interface. The horizontal axis represents the problem sizes, while the vertical axis represents the number of MPI processes.

The comparative analysis reveals that the results observed through the PaScaL Viewer's automatic visualizations lead to exactly the same analytical conclusions as those obtained through the exhaustive process of manual calculation and plotting. The Viewer employs color maps where darker regions represent higher efficiency. It accurately highlights the linear efficiency behavior in Region 1 (Fig. 7b) and the critical performance decay in Region 2 (Fig. 7c) as the number of processes scales. This convergence of data validates the accuracy of the PaScaL Suite's distributed memory extension, demonstrating that the tool is capable of abstracting the complexity of post-processing voluminous logs and providing immediate visual diagnostics without introducing statistical overhead that distorts the actual behavior of the application under test.

5.3. miniFE Scalability Analysis

To further evaluate the framework under more challenging, communication-bound conditions, we extended our analysis to the miniFE benchmark. Similar to the methodology applied previously, baseline scalability metrics were gathered manually and later cross-referenced with the output automatically produced by the PaScaL Viewer.

Figure 8 presents the stacked execution time decomposition across the evaluated scenarios. It highlights that as the problem size scales, the time spent in the Setup and Assembly phase (Region 1) becomes increasingly significant, especially for higher numbers of processors where the computational burden of the CG Solver drops due to distribution.

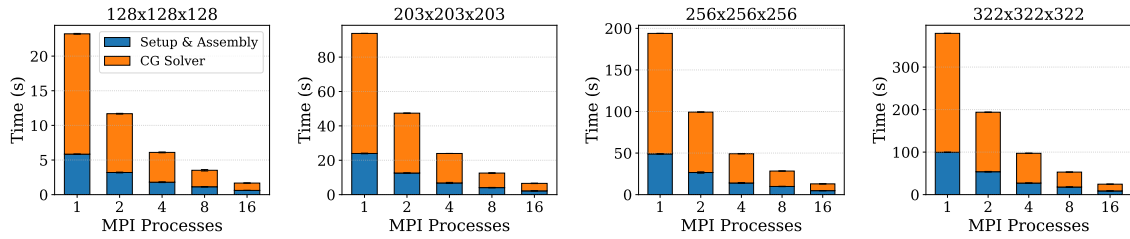


Figure 8. Breakdown of total execution time by number of MPI ranks for the miniFE application across different mesh sizes.

The speedup and parallel efficiency curves, shown in Figures 9 and 10 respectively, demonstrate distinct behaviors for each monitored region. While the CG Solver (Region 2) exhibits nearly ideal linear speedup across most configurations, the Setup and Assembly phase (Region 1) introduces a stark scalability bottleneck, severely degrading the overall efficiency as rank count increases.

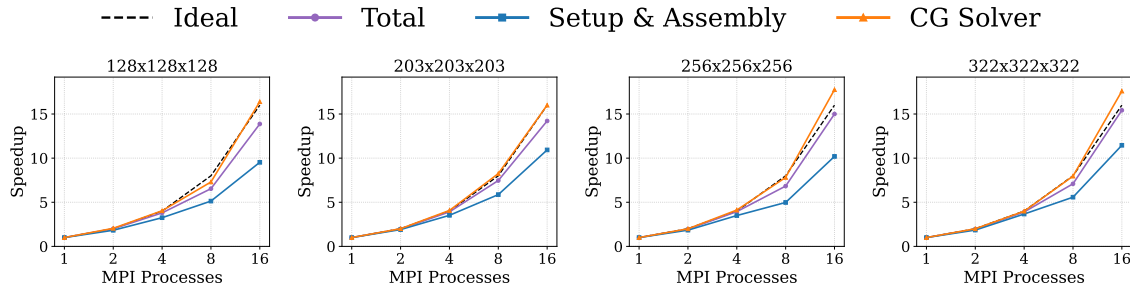


Figure 9. Total speedup curves for the Setup and Assembly phase (Region 1) and the CG Solver phase (Region 2) in miniFE.

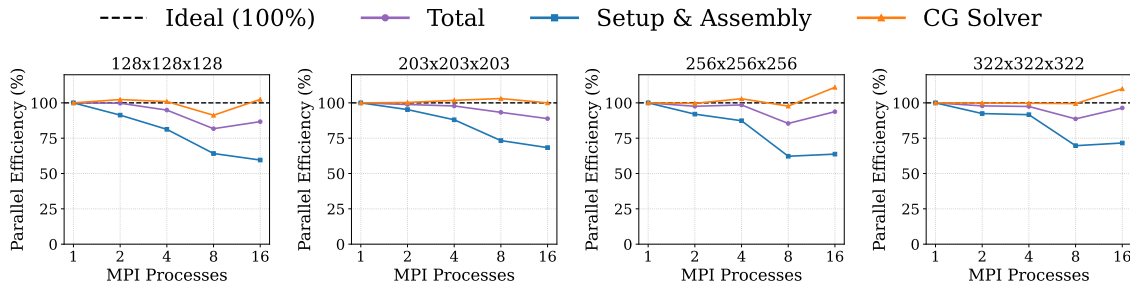


Figure 10. Parallel efficiency analysis highlighting the distinct scalability trends of the two primary regions in miniFE.

To validate the results generated by the tool, the Analyzer's output was fed into PaScaL Viewer to construct the heatmaps shown in Figure 11. In these maps, the vertical

axis represents the scale of MPI processes, while the horizontal axis maps the evaluated input mesh sizes: $i1$ ($128 \times 128 \times 128$), $i2$ ($203 \times 203 \times 203$), $i3$ ($256 \times 256 \times 256$), and $i4$ ($322 \times 322 \times 322$), where sequential labels $i1$ – $i4$ compactly represent multi-parameter command-line inputs ($-nx -ny -nz$).

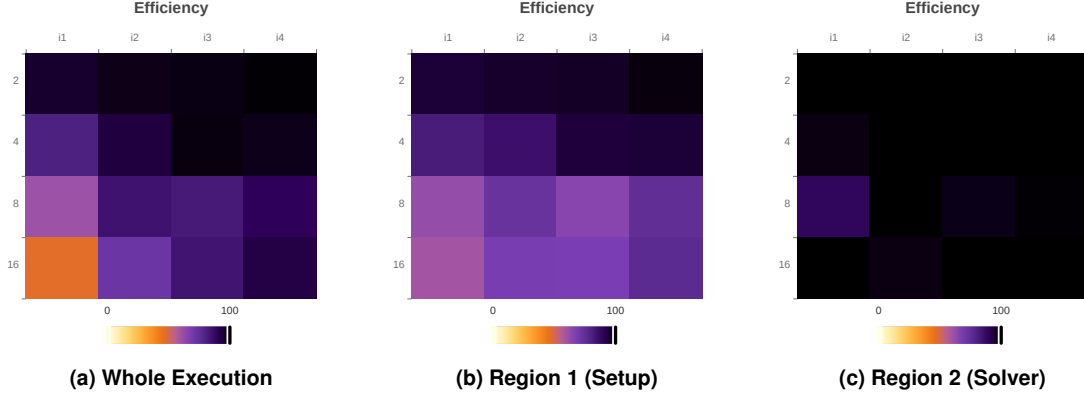


Figure 11. Automatic parallel efficiency heatmaps generated by PaScal Viewer for the miniFE benchmark. The linear scalability of the Solver phase (c) starkly contrasts with the severe bottleneck observed in the Setup phase (b).

A direct comparison between the visualization generated by PaScal and the baseline experiment metrics confirms the correspondence between the methods. In the overall execution (Figure 11a), the heatmap transitions gradually from dark to lighter shades as the number of processes increases, reflecting the decline in the total efficiency curve observed in Figure 10. For Region 1 (Figure 11b), a predominance of light colored cells is observed starting at 4 MPI processes becoming more pronounced with 8 and 16 processes across all mesh dimensions which reproduces the sharp drop recorded in the manual efficiency curves and visually identifies the Setup and Assembly phase as the scalability bottleneck. In contrast, Region 2 (Figure 11c) maintains a dark, homogeneous color pattern across virtually the entire test matrix, directly corresponding to the high, stable efficiency curves obtained manually for the CG Solver.

The consolidated results confirm that the extension of the PaScal Suite to distributed memory environments preserves the original execution characteristics of the software when evaluated under different configuration schemes.

6. Conclusion

This work presented an architectural extension of the PaScal Suite to natively support distributed-memory parallel applications based on MPI. By leveraging the Sidecar pattern, the framework decouples performance metric collection from the primary execution path. Using local Unix Domain Sockets for lightweight inter-process communication and deferring TCP/IP data aggregation until application termination, the solution eliminates runtime I/O contention and heavy trace handling typically associated with distributed profiling tools.

Empirical evaluations conducted on an 8-node HPC cluster with both a dense matrix multiplication kernel and the unstructured miniFE benchmark demonstrated the framework’s effectiveness and low measurement intrusion. Runtime overhead remained consistently below 1% for dense matrix multiplication and below 2.6% for most miniFE

configurations. Furthermore, the automated heatmaps produced by the PaScaL Viewer faithfully reproduced the scalability trends and bottleneck diagnoses obtained via manual, exhaustive profiling.

As future work, in addition to evaluating the ecosystem with a broader and more diverse range of HPC applications we plan to analyze the memory consumption of the PaScaL Sidecar implementation itself. Furthermore, we intend to demonstrate and validate automatic region instrumentation in distributed environments, as well as assess the framework on larger cluster topologies and thousands of simulated nodes using SimGrid/SMPI.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. Early stages of this research were carried out under the PIVIC and PIVIC-Af undergraduate research programs at UFRSA. The authors would like to thank the High-Performance Computing Center (NPAD) at the Federal University of Rio Grande do Norte (UFRN) for providing the computing infrastructure and technical support essential for conducting the experiments.

Declaration of AI Use

During the preparation of this work, the author(s) used Google Gemini in order to translate sections of the text, improve readability, and format LaTeX structures. After using this tool, the author(s) thoroughly reviewed and edited the content as needed and take full responsibility for the final content, originality, and accuracy of this publication.

References

- Adhianto, L., Anderson, J., Barnett, R. M., Grbic, D., Indic, V., Krentel, M., Liu, Y., Milakovic, S., Phan, W., and Mellor-Crummey, J. (2024). Refining hpctoolkit for application performance analysis at exascale. *INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS*, 38(6, SI):612–632.
- Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). Hpctoolkit: tools for performance analysis of optimized parallel programs. *CONCURRENCY AND COMPUTATION-PRACTICE & EXPERIENCE*, 22(6):685–701.
- Bergel, A., Bhatele, A., Boehme, D., Gralka, P., Griffin, K., Hermanns, M.-A., Okanović, D., Pearce, O., and Vierjahn, T. (2019). Visual analytics challenges in analyzing calling context trees. In Bhatele, A., Boehme, D., Levine, J. A., Malony, A. D., and Schulz, M., editors, *Programming and Performance Visualization Tools*, pages 233–249, Cham. Springer International Publishing.
- Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., and Schulz, M. (2016). Caliper: Performance introspection for hpc software stacks. In *SC '16: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS*, pages 550–560. IEEE Comp Soc; ACM; SIGPHC. International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Salt Lake City, UT, NOV 13-18, 2016.

- Burns, B. and Oppenheimer, D. (2016). Design patterns for container-based distributed systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO. USENIX Association.
- Calotoiu, A., Hoefler, T., Poke, M., and Wolf, F. (2013). Using automated performance modeling to find scalability bugs in complex codes. In *2013 INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS (SC)*, International Conference for High Performance Computing Networking Storage and Analysis. Assoc Comp Machinery; IEEE Comp Soc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, CO, NOV 17-22, 2013.
- da Silva, A. B. N., Cunha, D. A. M., Silva, V. R. G., Furtunato, A. F. d. A., and Xavier-de Souza, S. (2019). Pascal viewer: A tool for the visualization of parallel scalability trends. In Bhatele, A., Boehme, D., Levine, J., Malony, A., and Schulz, M., editors, *PROGRAMMING AND PERFORMANCE VISUALIZATION TOOLS*, volume 11027 of *Lecture Notes in Computer Science*, pages 250–264. 5th International Workshop on Visual Performance Analysis (VPA), Dallas, TX, NOV 11, 2018.
- da Silva, V. R. G., da Silva, A. B. N., Valderrama, C., Manneback, P., and Xavier-de Souza, S. (2022). A minimally intrusive approach for automatic assessment of parallel performance scalability of shared-memory hpc applications. *ELECTRONICS*, 11(5).
- Fan, K., Kesavan, S., Petruzza, S., and Kumar, S. (2024). Tinyprof: Towards continuous performance introspection through scalable parallel i/o. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Cited by: 2.
- Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. (2009). Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories.
- Huck, K. A., Shende, S., Malony, A. D., Coti, C., Spear, W., Alcaraz, J., Yokelson, D., Ramesh, S., Haque, M. M. A., Wood, C., Chaimov, N., Durbin, C., Johnson, A., Lambert, J., and Beekman, I. (2025). Preparing the tau performance system for exascale and beyond. *The International Journal of High Performance Computing Applications*, 39(4):532–552.
- Huf, A. and Siqueira, F. A. (2019). Composition of heterogeneous web services: A systematic review. *Journal of Network and Computer Applications*. PII: S108480451930205X; SGR: 85067853499; Cited by: 22.
- Islam, T. Z., Ayala, A., Jensen, Q., and Ibrahim, K. Z. (2019). Toward a programmable analysis and visualization framework for interactive performance analytics. *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 70–77.
- Jin, Y., Wang, H., Tang, X., Guo, Z., Zhao, Y., Hoefler, T., Liu, T., Liu, X., and Zhai, J. (2025). Leveraging graph analysis to pinpoint root causes of scalability issues for parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 36(2):308–325.

- Jost, G., Labarta, J., and Gimenez, J. (2004). Paramedir: A tool for programmable performance analysis. In Bubak, M., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, *Computational Science - ICCS 2004*, pages 466–469, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Knüpfer, A., Rössel, C., Mey, D. a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and Wolf, F. (2012). Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In Brunst, H., Müller, M. S., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Muelder, C., Gygi, F., and Ma, K.-L. (2009). Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1129–1136.
- Nagel, W., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. (1996). Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12.
- Pacheco, P. (2021). *An Introduction to Parallel Programming*. Morgan Kaufmann, 2nd edition.
- Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *INTERNATIONAL JOURNAL OF HIGH PERFORMANCE COMPUTING APPLICATIONS*, 20(2):287–311.
- Silva, A. B. N., Silva, V. R. G., Valderrama, C., and Xavier-de Souza, S. (2025). Pascal suite: A toolset for evaluating and profiling parallel scalability in shared-memory programs. *IEEE ACCESS*, 13:204708–204725.
- Thärigen, I., Protze, J., Orland, F., and Hermanns, M.-A. (2021). Differential performance analysis workflow for algorithmic changes. In *2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 7–16.
- Touati, S.-A.-A., Worms, J., and Briais, S. (2013). The speedup-test: a statistical methodology for programme speedup analysis and computation. *Concurrency and Computation: Practice and Experience*, 25(10):1410–1426.
- Weber, M., Ziegenbalg, J., and Wesarg, B. (2019). Online performance analysis with the vampir tool set. In Niethammer, C., Resch, M. M., Nagel, W. E., Brunst, H., and Mix, H., editors, *Tools for High Performance Computing 2017*, pages 129–143, Cham. Springer International Publishing.
- Xuan, Z., You, X., Feng, T., Liu, Y., and Qian, D. (2025). Simtrace: Exploiting Spatial and Temporal Sampling for Large-Scale Performance Analysis. *ACM Transactions on Architecture and Code Optimization*. SGR: 105010723962.
- Zhukov, I. and Wylie, B. J. N. (2014). Assessing measurement and analysis performance and scalability of scalasca 2.0. In an Mey, D., Alexander, M., Bientinesi, P., Canataro, M., Clauss, C., Costan, A., Kecskemeti, G., Morin, C., Ricci, L., Sahuquillo, J., Schulz, M., Scarano, V., Scott, S. L., and Weidendorfer, J., editors, *Euro-Par 2013: Parallel Processing Workshops*, pages 627–636, Berlin, Heidelberg. Springer Berlin Heidelberg.