A Client-Side Architecture for Enhancing Productivity of Interactive Parallel Scalability Analysis with PaScal Suite

Igor Sérgio de França Correia[1], **Samuel Xavier-de-Souza**[1]

[1] Departamento de Engenharia de Computação e Automação
Universidade Federal do Rio Grande do Norte (UFRN) – Natal, RN – Brazil

`igor.sergio.017@ufrn.edu.br, samuel@dca.ufrn.br`

Abstract. *Traditional profilers struggle with scalability analysis. The PaScal Suite's Viewer tool addresses this, but its server-side architecture creates performance bottlenecks with large datasets. This paper proposes a client-side Single-Page Application (SPA) architecture that transfers all data processing to the browser, enabling fluid, instantaneous analysis. We also introduce a novel hierarchical visualization for comparing code regions, which simplifies diagnostics. The result is an agile platform that enhances the identification of scalability bottlenecks.*

1. Introduction

The number of cores in high performance computing (HPC) systems continues to grow exponentially, a trend that is consistently observed in the Top500 list reports [Dongarra et al. 2025]. This scenario makes software scalability a permanent and crucial challenge. To extract maximum performance from these machines, developers must ensure that their applications efficiently benefit from the increase in resources, which requires the use of robust performance analysis tools.

Scalability analysis demands a longitudinal perspective, comparing the behavior of an application under multiple execution configurations. However, the ecosystem of profiling tools has been mostly designed with a focus on optimizing a single execution. Canonical tools such as Gprof [Graham et al. 1982], VAMPIR [Nagel et al. 1996], TAU [Shende and Malony 2006], and HPCToolkit [Adhianto et al. 2010] are essential for finding bottlenecks in a specific environment, but they do not offer native support for the comparative analysis of scalability trends. Manually consolidating their results is impractical, and solutions that sought to automate this comparison faced other challenges. PerfExplorer [Huck and Malony 2005], for example, uses line charts that quickly become unintelligible as the parameter space increases [Silva et al. 2018], while powerful tools like SCALASCA [Wolf et al. 2008] present a complexity that can be a barrier to quick visual diagnostics.

In this context, the PaScal Suite was proposed, aiming to automate and simplify the visualization of scalability analysis. The suite is composed of two components: the PaScal Analyzer [Silva et al. 2022], responsible for orchestrating the executions and generating a consolidated JSON file with performance data, and the PaScal Viewer [Silva et al. 2018, Cunha 2018], the visualization interface. The conceived workflow involves the user loading this JSON file into a web application — the PaScal Viewer — which, in turn, interprets the data and presents it intuitively. The first version of the

Viewer introduced an effective visual paradigm, based on four color diagrams to map efficiency trends, simplifying the identification of bottlenecks.

However, the original architecture of the PaScal Viewer, implemented with Python and Django in a server-side model, proved to be a performance bottleneck. The processing of large JSON files on the server, necessary to generate the diagrams, resulted in high latency. This waiting time directly compromised interactivity and the user experience, making the analysis of larger datasets a slow and impractical process.

To overcome these limitations, this paper proposes and validates a client-side architectural paradigm for interactive scalability analysis tools. We present a complete redesign of the PaScal Viewer as a modern Single-Page Application (SPA), which shifts the computational load from a centralized server to the end-user's browser. This approach not only solves the performance bottlenecks of the previous version but also enables a more fluid and powerful analytical workflow. The new version combines the consolidated visual paradigm of its predecessor with a modern Single-Page Application (SPA) [Gama et al. 2018] architecture, using the Angular framework. This approach transfers all data processing logic to the client (client-side), resulting in a drastically faster, more interactive analysis platform capable of handling large volumes of data instantly in the browser.

The main contributions of this work are:

- The proposal and validation of a high-performance client-side architecture for interactive scalability analysis, demonstrating its effectiveness in eliminating server-side latency bottlenecks.
- A novel visualization paradigm for hierarchical analysis of code regions, which integrates a performance summary directly into a navigable tree structure, reducing the user's cognitive load.
- An enhanced interactive model for direct comparative analysis of multiple code regions, streamlining the discovery of performance patterns.
- The implementation of these concepts in a modern, extensible platform that serves as a blueprint for future client-based performance tools.

The remainder of this paper is organized as follows: Section 2 describes the new architecture proposed for the PaScal Viewer; Section 3 presents the tool's performance analysis metrics; Section 4 details the new features of the version; Section 5 provides a validation of the application through a benchmark; and finally, Section 6 presents our conclusions and discusses directions for future work.

2. New Architecture

The proposed solution, detailed in this paper, was designed to overcome the performance bottlenecks of its predecessor [Cunha 2018], migrating from a server-side model to a modern, fully client-focused (client-side) Single-Page Application (SPA) architecture [Gama et al. 2018]. This section details this new architecture, starting with its integration into the PaScal suite and the data format it consumes.

2.1. The PaScal Suite and the Input Data Format

PaScal Viewer is the visualization component of the PaScal suite, designed to work in conjunction with the PaScal Analyzer [Silva et al. 2022]. The Analyzer orchestrates the execution of a parallel application under multiple configurations and consolidates performance metrics into a single JSON output file, which serves as the standardized interface between the tools.

The JSON structure is self-contained, divided into two main objects. The **`config`** object contains metadata, such as the workloads used (`arguments`) and the structure of the data keys (`data_descriptor.keys`). The **`data`** object is a map where each key represents a unique execution, containing its total execution time and detailed measurements for instrumented code regions (`regions`) and load imbalance (`imbalances`). A simplified example is shown in Listing 1.

A simplified example of the input JSON file structure can be seen in Listing 1.

```
{
  "config": {
    "command": "blackscholes 1 inputs/in_4K.txt",
    "arguments": ["inputs/in_4K.txt", "inputs/in_16K.txt"],
    "data_descriptor": {
      "keys": ["cores", "input", "repetitions"]
    },
    "extras": {
      "regions": {
        "values": ["start_time", "stop_time", "start_line", "
   stop_line", "thread_id", "filename"]
      },
      "imbalances": {}
    }
  },
  "data": {
    "1;0;0": {
      "start_time": 1712078902.1,
      "stop_time": 1712078903.5,
      "regions": {
        "1": [
          [1712078902.5, 1712078903.2, 472, 495, 0, "
   blackscholes.c"]
        ]
      },
      "imbalances": { "1": 0.05 }
    }
    // ...
  }
}
```

Listing 1. Simplified example of the input JSON file structure.

Upon receiving this file, the new application processes it entirely in the browser, eliminating the need for communication with a server.

2.2. From Server to Client: A Re-engineering for Performance

The primary motivation for re-engineering PaScal Viewer was the performance bottleneck of its original architecture. In the first version, the entire pipeline was synchronous and centralized on the server: the JSON file was read entirely into memory, metrics were calculated through multiple operations, and all 16 possible visualizations were generated before any response was sent to the client [Cunha 2018]. This model resulted in high initial latency, excessive consumption of server CPU and I/O resources, and an interface that did not respond well with larger data files, thereby compromising interactivity.

The new client-side architecture, implemented with the Angular framework, resolves these issues by transferring the entire workload to the user's browser. This approach tackles the root of the latency problem by eliminating the need for file uploads and the waiting time for remote processing, allowing the analysis to begin instantly in the browser. Furthermore, the rendering of the graphs is now on-demand; unlike the previous version that pre-calculated all visualizations, the new architecture dynamically generates the diagrams as the user interacts with the interface, making it more agile and responsive.

2.3. Client-Side Data Processing Flow

The workflow of the new tool, from file loading to interactive visualization, is detailed below and illustrated in the flowchart in Figure 1.

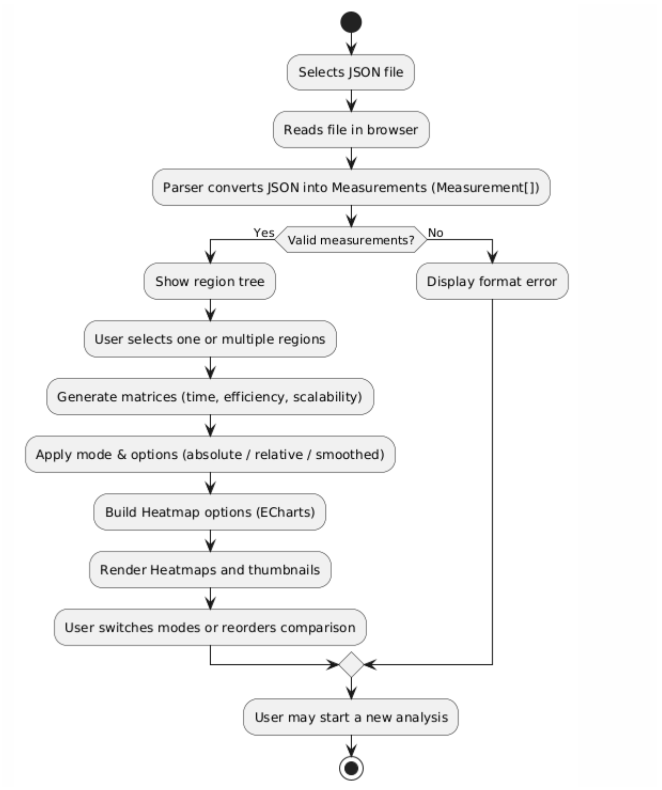
Selects JSON file
Reads file in browser
Parser converts JSON into Measurements (Measurement[])
Yes
Valid measurements?
No
Show region tree
Display format error
User selects one or multiple regions
Generate matrices (time, efficiency, scalability)
Apply mode & options (absolute / relative / smoothed)
Build Heatmap options (ECharts)
Render Heatmaps and thumbnails
User switches modes or reorders comparison
User may start a new analysis

Figure 1. Data processing flowchart of the new client-side architecture.

The process is divided into four main stages. First, upon file selection, a parsing service converts the JSON's nested structure into a linear list of measurements, pre-calculating derived metrics like total region time and imbalance. Second, when a user selects code regions, the application dynamically builds the necessary metric matrices (e.g., execution times, efficiencies). Third, a centralized service generates the color diagram configurations for the visualization library, applying color palettes and interpolation algorithms as needed. Finally, the charts are rendered in the interface, allowing the user to interact with them, switching between different modes and comparing regions.

This new implementation distributes the processing load, efficiently leveraging the capabilities of modern browsers and removing the bottleneck of a centralized server. As a result, the analysis experience becomes drastically faster, more interactive, and capable of handling large volumes of data instantaneously.

3. Metrics and Performance Analysis

PaScal Viewer is a performance analysis system based on statistical metrics and classic parallel scalability models. The calculation pipeline transforms raw execution data into visualizations capable of revealing bottlenecks, theoretical limits, and potential gains from using multiple processors.

3.1. Fundamentals and Notation

Let w be the workload and p the number of processors. For each pair (w, p), the system aggregates all available repetitions and uses the median of the execution time, $T(w, p)$, as it is more robust to outliers and occasional variations. The sequential time is defined as $T_{\text{seq}}(w) = T(w, 1)$, requiring the existence of at least one execution with $p = 1$ for each w.

The central analysis metric is parallel efficiency:

$$E(w,p) = \frac{T_{\text{seq}}(w)}{p \cdot T(w,p)},$$

which expresses how close the observed performance is to the ideal linear *speedup*. Values $E > 1$, characterizing superlinearity, may occur in specific situations and should be interpreted with caution. The behavior of $E(w, p)$ is consistent with Amdahl's Law [Amdahl 1967], according to which the sequential fraction of an application imposes a ceiling on the maximum gain obtained through parallelization.

3.2. Scalability: Amdahl and Gustafson

In addition to the fixed-problem scenario described by Amdahl's Law, *PaScal Viewer* also incorporates Gustafson's formulation [Gustafson 1988], which considers an increase in workload proportional to the number of processors. From these two perspectives, the tool calculates three complementary scalability metrics, all derived from the efficiency matrix $E(w, p)$:

$$\text{scalab}(w,p) = E(w,p) - E(w_0,p), \tag{1}$$

$$\text{strong}(w,p) = E(w,p) - E(w,1), \tag{2}$$

$$\text{weak}(w,p) = E(w,p) - E(w-m,\ p-m). \tag{3}$$

These metrics represent absolute scalability, in which each value is calculated relative to a fixed reference point:

- **Scalable**: compares the efficiency of a workload w with that of a reference workload w_0, while keeping the number of processors constant;
- **Strong scaling**: measures the variation in efficiency when increasing the number of processors p while keeping the workload w constant, relative to the sequential execution ($p = 1$);
- **Weak scaling**: evaluates the simultaneous growth of workload and number of processors, comparing each point (w, p) with the initial configuration along the diagonal.

In addition to these, PaScal Viewer also allows for the calculation of relative scalabilities, where the comparison is always made with the immediate neighbor (i.e., between consecutive executions in terms of workload or number of processors). Thus, while absolute metrics show the deviation from a global *baseline*, relative metrics highlight the marginal increments at each configuration step. The ability to switch between both views offers the user the flexibility to analyze both the cumulative evolution and the incremental gain.

Figure 2 illustrates this set of analyses: on the left, the heatmap of absolute efficiency, and in the three remaining positions, the corresponding maps for the absolute *scalab*, *strong*, and *weak* scalabilities. This visual organization allows the user to quickly correlate the behavior of efficiency with its different scalability interpretations.

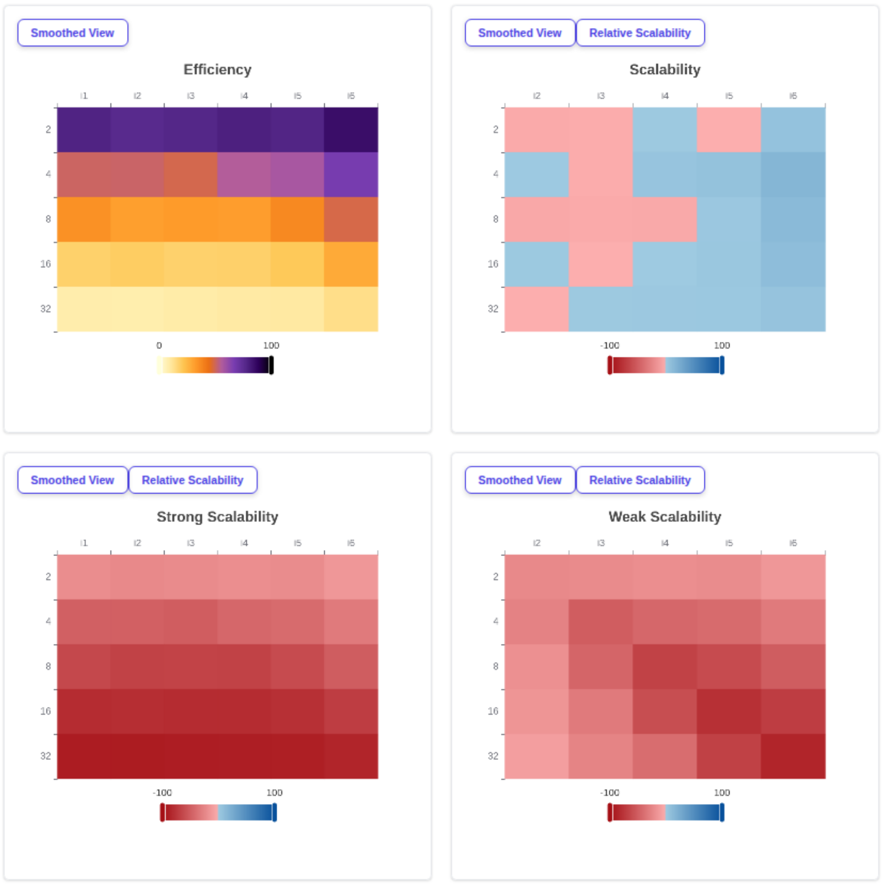

Figure 2. Heatmaps of efficiency and the three calculated scalability metrics (absolute mode).

3.3. Load Balance and Statistical Robustness

Another reported indicator is the load imbalance $I(w, p)$, which quantifies the disparity between the individual execution times of threads within a region. By construction, $I \in [0, 1]$ (or $[0, 100]\%$ in the visualization): values close to 0 indicate ideal balancing, while high values suggest poor work distribution or effective serialization, reducing the overall efficiency $E(w, p)$.

The robustness of the analysis is reinforced by the use of the median, which mitigates the impact of outliers and occasional performance fluctuations. Superlinear cases, when present, are preserved in the efficiency matrix for investigation but can be attenuated in the scalability differences to avoid visual distortions.

4. New Features

The transition to an SPA was not limited to a technological update. The new architecture enabled the introduction of more interactive and analytically powerful visualization paradigms that address the limitations of the previous version, especially concerning the analysis of programs with multiple nested code regions. The two central features of this new version are the hierarchical visualization of regions and the interactive comparative analysis, which together reduce the developer's cognitive load and expedite the identification of scalability bottlenecks.

4.1. Hierarchical Visualization of Code Regions

In complex parallel applications, performance analysis focuses on specific regions of interest. These regions are not detected automatically; rather, they are explicitly demarcated by the developer in the source code using functions from the PaScal library. This instrumentation process itself defines a hierarchy, as regions can be nested within one another (e.g., a parallel loop within a larger function). The list-based representation in the previous version of PaScal Viewer [Silva et al. 2018] was inadequate for capturing this explicit, hierarchical structure, making the navigation and correlation between parent and child regions a manual and unintuitive task.

The new tree view (Figure 3) hierarchically organizes all these instrumented regions of the program, treating the complete execution ("whole program") as the root node (ID "0") and the sub-regions as its descendants (e.g., "0.1", "0.2", "0.1.1").

4.1.1. Hierarchy Construction and Visual Synthesis

The construction of the tree is carried out based on the unique identifiers of each region, which follow a dot notation (e.g., '0.3.2'), where each dot represents a level of depth. The visualization component processes the list of regions from the input JSON file, ordering them to ensure that parent nodes are created before their children.

The main differentiator of this visualization is that each node in the tree is not just a textual label, but a *visual performance summary*. For each region, a thumbnail of its respective efficiency heatmap is generated.

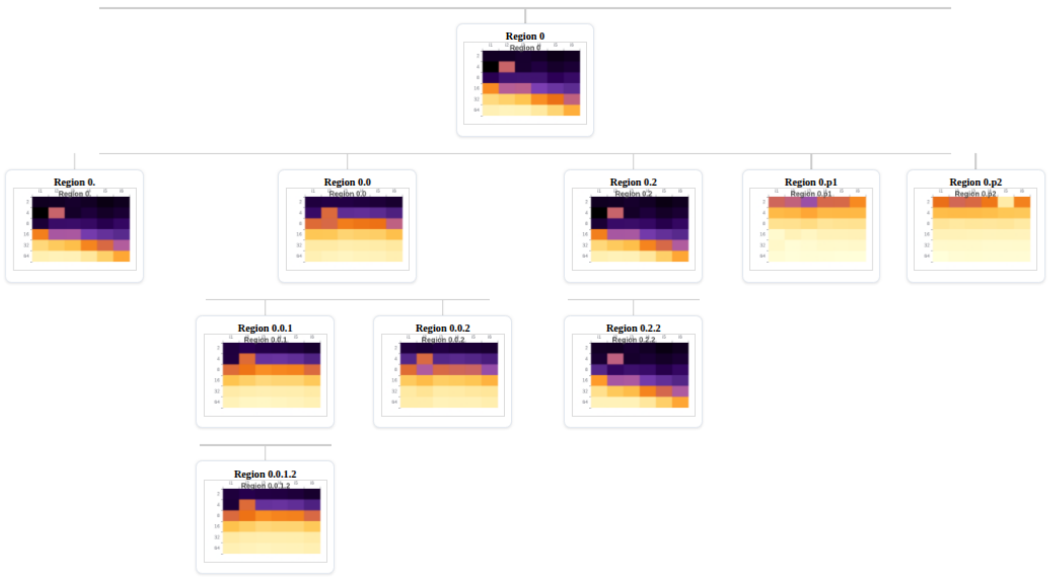

Figure 3. Tree visualization structure for regions.

To ensure readability in programs with many regions at the same level, the component implements an adaptive layout. It calculates the number of nodes in the most populated level of the tree and, if it exceeds a predefined threshold, applies a scaling factor to all nodes, preserving the proportion of the thumbnails and avoiding horizontal overflow.

Additionally, each node provides a tooltip with contextual metrics that enrich the analysis without cluttering the main interface. The metrics include:

- **Execution Percentage:** The range (minimum and maximum) of that region's contribution to its parent region's total time.
- **Time Variation:** A derived metric that indicates the observed load imbalance.
- **Static Metadata:** The file name and the range of code lines corresponding to the region.

4.2. Interactive Comparative Analysis of Multiple Regions

The hierarchical structure serves as the foundation for a powerful comparative analysis tool. The user can select multiple nodes in the tree with a simple click. The list of selected regions is managed by the application's state and emitted through an event, allowing other components to react to this selection.

The multiple selection functionality is a central feature of the new interface, allowing the heatmaps of different regions to be displayed side-by-side for direct comparative analysis (Figure 4). This visualization simplifies the identification of distinct scalability patterns between different parts of the code. Additionally, the tool offers greater interactivity by allowing the user to dynamically reorganize the order of the graphs using drag and drop functionality.

The combination of the tree as a **visual performance index** and the flexibility of multiple selection transforms the analysis from one of ”searching” to one of ”discovery”.

The developer can quickly identify problematic regions, understand their structural location in the program, and compare them with well-performing regions to extract insights that direct optimization efforts.

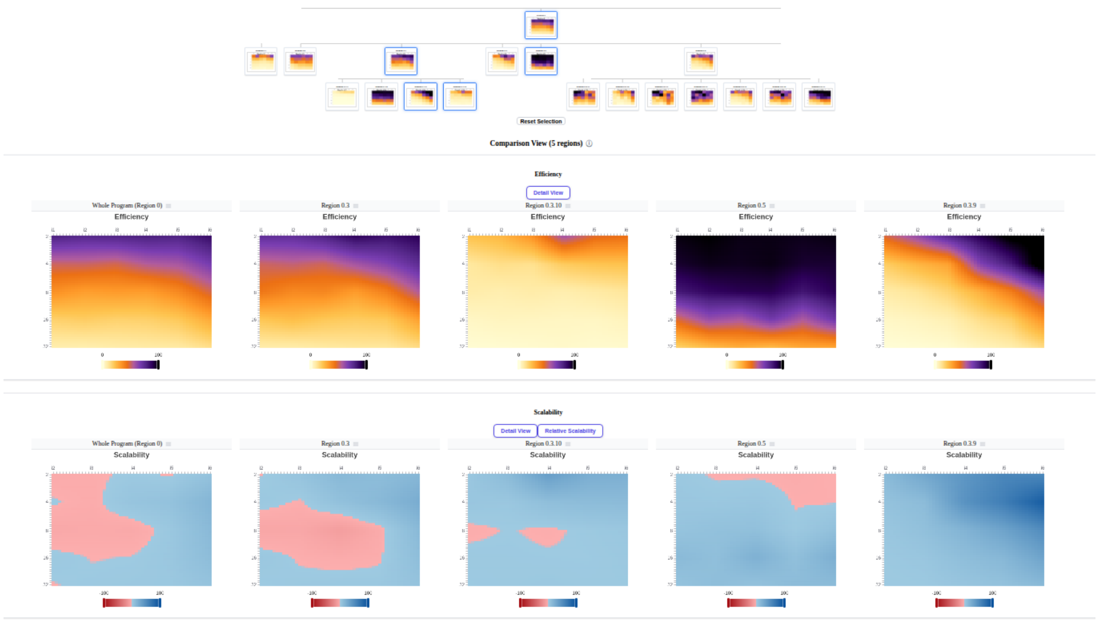

Figure 4. Tree with selected regions (indicated by a blue border at the top) followed by the display of their efficiency and scalability graphs.

5. Performance Validation: Server-Side vs. Client-Side Architecture

To validate the effectiveness of the new architecture, we conducted a benchmark comparing the processing time of the original server-side implementation of PaScal Viewer with the new client-side SPA approach proposed in this paper.

5.1. Experimental Setup

The experiments were conducted on a desktop computer with an Intel Core i3-8100 CPU and 8 GB of RAM. The client-side tests were run on the Firefox browser. For each file size, the processing time was measured 5 times, and the average value is reported in Table 1, excluding the first run to account for caching effects. To ensure a fair comparison of processing capabilities, the server-side tests were executed on the same machine.

The tests were executed on multiple input file sizes (the consolidated JSON from the Analyzer). Table 1 presents the average times in seconds; Figure 5 illustrates the same comparison.

5.2. Results and Performance Gain

The results show that the client-side approach consistently outperforms the original implementation, with time reductions reaching almost three orders of magnitude. For most common-use cases (files up to $\sim$1 MB), the new platform effectively produces responses in the range of 20–35 ms ($\approx 0.02 - 0.03$ s). Even with files of 47.7 MB, the average processing time remains at ≈ 0.29 s, and for 147.7 MB, it is ≈ 0.79 s.

From a user experience perspective, responses up to $\sim$100 ms tend to be perceived as *instantaneous*; up to $\sim$1 s they maintain the cognitive flow without interruption; and above $\sim$10 s they become disruptive to attention [Miller 1968, Nielsen 1993]. Thus, for the majority of common use cases (files up to $\sim$ 1 MB), the new platform is effectively *almost instantaneous*. For very large inputs (tens to hundreds of MB), it remains well below the 1–2 s threshold, preserving a sense of fluidity.

Table 1. Average time by file size: a performance comparison between the original server-side architecture and the new client-side SPA architecture.

File size	Original (s)	Proposed (s)	Speedup (Original/Proposed)
22.4 kB	1.79	0.0185	×97
140 kB	5.58	0.0240	×232
1030 kB	32.32	0.0340	×951
47.7 MB	52.15	0.291	×179
143.8 MB	140.73	0.789	×178

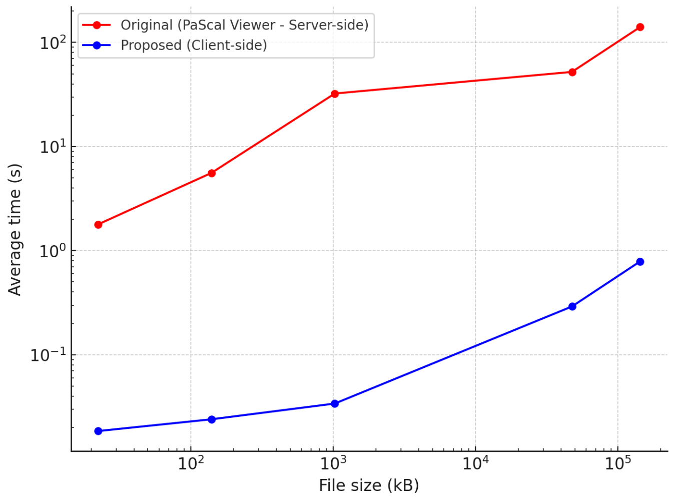

Figure 5. Performance comparison between the server-side (original) and client-side (proposed) architectures as a function of the input file size.

5.3. Discussion: The Trade-offs of Client-Side Processing

The original server-side architecture imposed a theoretical limit of 500 MB for file uploads. However, in practice, we have observed timeouts and execution errors in the 100–200 MB range, due to the combined cost of data transfer, parsing, and centralized computation on a server under load.

In contrast, the proposed architecture executes the entire performance analysis pipeline on the client. This architectural shift eliminates upload latency and contention on a central server, directly addressing the interactivity bottleneck of the previous version. The trade-off, however, is that the analysis performance becomes intrinsically limited by the user's local computing resources. The primary constraints are the browser's available RAM to hold the dataset and intermediate data structures, and the client's CPU power to perform aggregations and metric calculations. We observed that memory consumption can reach 3–6 times the input file size, which can be a limiting factor on resource-constrained machines when analyzing datasets on the order of hundreds of megabytes.

This dependency on user hardware directly impacts the scalability claimed in the paper, a crucial point raised during the review process. While our implementation efficiently processes files in the hundreds of MBs with low latency on modern developer machines, users with less powerful systems might experience degraded performance. It is worth noting, however, that the hardware configuration used in our tests (Intel Core i3-8100 with 8 GB of RAM) represents a common baseline for personal computers today. If a user experiences latency due to RAM limitations with our tool, it is highly probable that their hardware is already a bottleneck for general-purpose web browsing and other demanding applications.

Nevertheless, the client-side approach proved to be a valid and highly effective solution for enhancing the productivity and fluidity of the analysis for a vast majority of practical use cases, validating this architectural choice for performance data of parallel applications. Future work could explore hybrid models or data sampling techniques to mitigate these limitations for extremely large datasets.

6. Conclusions

This work presented and validated a client-side architectural paradigm for interactive parallel scalability analysis, implemented as a Single-Page Application (SPA). Our primary contribution is the demonstration that shifting the computational load from a centralized server to the client's browser is an effective strategy to eliminate performance bottlenecks in performance analysis tools. The benchmark results confirm this, showing that the analysis latency becomes practically instantaneous for most use cases. This transformation is critical as it preserves the user's cognitive flow, enabling a more fluid and productive diagnostic process.

Furthermore, we introduced a novel visualization paradigm for hierarchical code region analysis. By integrating visual performance summaries into a navigable tree structure, our approach reduces the cognitive load required to correlate nested regions and identify scalability issues. In summary, the proposed architecture advances the state-of-the-art for this class of tools by enabling a highly interactive and insightful analysis workflow, which was previously impractical under the server-side model.

Future work will focus on enhancing the user experience, packaging the tool for offline use, and exploring near real-time integration with the PaScal Analyzer to further accelerate the diagnostic cycle.

References

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM.

Cunha, D. A. M. (2018). Pascal viewer - graphic representation of the efficiency variation of a parallel application analyzed by time markers. Undergraduate thesis (computer engineering), Universidade Federal do Rio Grande do Norte, Natal, Brazil.

Dongarra, J., Meuer, M., Strohmaier, E., and Simon, H. (2025). The top500 list. Available at: `https://www.top500.org/`.

Gama, K., Cassino, C., and Sousa, L. (2018). Single-page applications: A textual overview of the state-of-the-art. In *2018 IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE.

Graham, S. L., Kessler, P. B., and McKusick, M. K. (1982). Gprof: a call graph execution profiler. *SIGPLAN Notices*.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

Huck, K. A. and Malony, A. D. (2005). Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. IEEE.

Miller, R. B. (1968). Response time in man-computer conversational transactions. *AFIPS Fall Joint Computer Conference*, 33:267–277.

Nagel, W. E., Arnold, A., Weber, M., Hoppe, H. C., and Solchenbach, K. (1996). Vampir: Visualization and analysis of mpi resources. *Supercomputer*.

Nielsen, J. (1993). *Usability Engineering*. Morgan Kaufmann, San Francisco, CA.

Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *International Journal of High Performance Computing Applications*.

Silva, A. B. N. d., Cunha, D. A. M., Silva, V. R. G., Furtunato, A. F. d. A., and Souza, S. X. d. (2018). Pascal viewer: a tool for the visualization of parallel scalability trends. In *3rd Workshop on Visual Performance Analysis (VPA), held in conjunction with SC18*.

Silva, V. R. G. d., Silva, A. B. N. d., 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):689.

Wolf, F., Mohr, B., Malony, A. D., and Shende, S. S. (2008). Scalasca: A toolset for performance analysis of large-scale parallel applications. In *Proceedings of the International Conference on Parallel Tools (PTOOLS)*. Springer.