Reproducibility of Published Results: Practical MHP Analysis for Java

Vinicius Silva Gomes

Universidade Federal de Minas Gerais, Brazil
vinicius.gomes@dcc.ufmg.br

1 Choose a paper presented in a recent compiler related conference, such as CGO, PLDI or CC.

I chose the paper "Practical MHP Analysis for Java", published at CC 2026. It was written by A. Samuel Moses and V. Krishna Nandivada, both from IIT Madras, and is available at https://doi.org/10.1145/3771775.3786279.

2 Download the material that the authors have made publicly available for the paper. If there is no such material, but the student still wants to reproduce the paper's results, he/she can write to the authors of the paper, asking for their implementation.

The material is publicly available at https://doi.org/10.6084/m9.figshare.30928775.v1. It includes the paper's novel GRIP-MHP implementation, the baseline LV implementation, and a Python orchestration script (`run_benchmarks.py`). Furthermore, it provides a Dockerfile and a Makefile to containerize the environment and streamline the reproduction of the experiments. The benchmark datasets used in the study are managed via the script and mounted into the container at runtime.

3 Run the implementation, trying to reproduce at least one of the experiments in the paper. Even if it is not possible to reproduce the experiments, the student can still write a report about his/her tries, to claim the extra points.

Most of the experiments were successfully reproduced. The following section presents the reproduction process in detail, highlighting the experimental setup, obtained results, and the main observations derived from the replication.

4 Write a short report about the entire procedure. This report must contain the URL of the material that has been downloaded, plus a brief description of the student's experience with it. If the student has not been able to reproduce those experiments, than the report must explain the reasons for this failure.

The first step in reproducing the results was building the Docker image and listing the available benchmarks. Alongside this, I ran a quick test suite (predefined by the authors) to ensure the environment was functioning correctly, before proceeding to execute the more extensive tests and comparing my results with those reported in the paper.

Before successfully running the experiments, I encountered a minor issue in the author's Makefile. For all the defined targets (`shell`, `run`, `list`, etc.), the variables containing paths to critical files and directories, such as the benchmarks and the log directory, are not escaped with double quotes. Consequently, if the files are located in a directory path containing spaces, the scripts fail to execute properly, mostly returning errors indicating that the container image could not be found.

To fix this, one can either add quotes around the directories in the Makefile or move the files to a path where no folder names contain spaces. To avoid modifying the author's original files, I opted to simply move the project folder to a new location to run the scripts. After this adjustment, the `build` and `list` commands from the Makefile worked correctly, successfully building the Docker image with all the code and listing all available benchmark instances. With this resolved, I was able to perform a quick test using the following command:

```
make run ARGS="--mhp_method GRIP_MHP --inlining-mode targetted --suite fast"
```

The initial results were successfully generated in the `docker-logs` folder, which stores the logs organized by method (the proposed approach or the baseline) and by benchmark instance. From this point onward, I proceeded to execute the remaining benchmarks.

To automate the experimentation phase, a Bash script (Listing 1.1) was developed to execute both algorithms against all available benchmarks, considering both targetted and full inlining strategies. The automated execution of all configurations took a total of 10 hours and 48 minutes to complete.

```
algo=("LV" "LV_SIM" "GRIP_MHP")
inlining=("targetted" "full")
benchmarks=("executor-futures" "java-dining-philosophers" "thread-join" "fj-
    kmeans" "graphchi" "h2" "jme" "luindex" "lusearch" "pmd" "sunflow" "xalan"
     "zxing")

for bench in "${benchmarks[@]}"; do
  for a in "${algo[@]}"; do
    for inl in "${inlining[@]}"; do
      echo "Benchmark=$bench | Algoritmo=$a | Inlining=$inl"
      make run ARGS="--mhp_method $a --inlining-mode $inl --benchmark $bench"
    done
  done
done
```

Listing 1.1. Script used to run all benchmarks with all options.

Following the execution, it was observed that several instances exceeded the allocated heap memory limit, while others terminated due to timeouts. A timeout threshold of 7200 seconds was adopted for all experiments. Table 1 summarizes which instances completed successfully and which were interrupted by heap exhaustion or timeout events.

The results obtained in our replication are consistent with the observations reported by the original authors. As previously documented, baseline approaches such as the LV algorithm fail to scale to complex applications due to the state-space explosion caused by thread modeling. This phenomenon directly explains the Out of Memory (OOM) and timeout errors observed in the more demanding benchmarks. Reproducing these same practical bottlenecks reinforces the paper's central motivation: graph-reduction techniques, such as those introduced by GRIP-MHP, are essential for enabling the analysis of real-world systems within feasible computational limits.

The results obtained for the `luindex` and `sunflow` benchmarks under the targeted inlining strategy provide particularly strong evidence supporting this observation. While the baseline algorithm failed to complete within the allotted time, the executions using GRIP-MHP finished successfully. This contrast directly supports the main claim that combining advanced graph-reduction techniques with a targeted inlining strategy makes the analysis of complex benchmarks feasible under practical resource constraints.

Conversely, the OOM failures observed during the full inlining strategy for the `luindex`, `lusearch`, and `zxing` benchmarks were largely expected. For these highly concurrent and computationally

Table 1. Execution status of the *benchmarks* highlighting Out of Memory (OOM) errors and Timeouts.

Benchmark	LV		GRIP-MHP	
	Targetted	**Full**	**Targetted**	**Full**
executor-futures	✓	✓	✓	✓
java-dining-philosophers	✓	✓	✓	✓
thread-join	✓	✓	✓	✓
fj-kmeans	✓	✓	✓	✓
graphchi	✓	✓	✓	✓
h2	✓	✓	✓	✓
jme	✓	✓	✓	✓
pmd	✓	✓	✓	✓
xalan	✓	✓	✓	✓
luindex	Timeout	OOM	✓	OOM
lusearch	✓	OOM	✓	OOM
sunflow	Timeout	OOM	✓	OOM
zxing	✓	OOM	✓	OOM

intensive workloads, fully expanding the state space becomes impractical due to hardware resource limitations. As a result, the memory exhaustion observed in these scenarios, even when using GRIP-MHP, is, somehow, expected.

Finally, based on the collected metrics, it was possible to compare the numerical results obtained in our experiments with those presented in the original paper. This comparison allowed us to evaluate whether the central intuition and scalability claims of the proposed approach remain valid when reproduced under a distinct experimental environment. Besides that, some differences between our results and those reported in the paper are expected due to variations in the experimental environment.

In particular, differences in hardware configuration, available heap memory, JVM implementation and version, and system workload during execution may all influence execution time and memory consumption. While the original paper does not provide a complete specification of the experimental setup, such as processor model, memory configuration, operating system, or JVM version, our replication experiments were conducted on a machine equipped with an Intel Xeon E5-2680 v4 processor (14 cores and 28 threads), 16 GB of RAM, running Ubuntu 24.04 and OpenJDK 25.0.2. The JVM executions used the default heap memory configuration provided by the Java Virtual Machine.

Despite the unavoidable variations on numerical reproduction, the overall scalability trends, bottlenecks, and performance improvements observed in our replication remained consistent with those reported by the authors, suggesting that the core behavior of the proposed approach is robust across different execution environments. Tables 2, 3, and 4 (corresponding respectively to Tables 2, 3, and 4 of the original paper) summarize part of the collected results in an aggregated form, following a structure similar to that adopted in the original article.

A comparison between the reproduced experimental results and those reported in the original paper shows a strong overall consistency in both qualitative trends and quantitative behavior. Across all evaluated dimensions, the GRIP-MHP approach consistently preserves the key advantages claimed by the authors when compared to the LV baseline, particularly in terms of scalability, execution efficiency, and reduction of the underlying state space used in the analysis.

In terms of state-space complexity, the reproduced results confirm the central claim that GRIP-MHP substantially reduces the size of the concurrency graphs before the expensive analysis phase. The number of nodes in the constructed CSEG structures is significantly lower under

Table 2. Number of nodes in CSEG.

Benchmark	(c) MLVTI	(d) MLVFI	(e) PTI	(f) MTI	(g) PFI	(h) MFI
executor-futures	9	9	14	14	14	14
fj-kmeans	165	167	201	24	203	23
graphchi	1273	2874	1493	277	2583	390
h2	452	529	278	64	345	82
java-dinning-philosophers	257	261	224	146	232	146
jme	559	562	367	61	373	61
lusearch	3099	-	2441	425	-	-
pmd	454	584	190	41	240	26
tests	-	-	-	-	-	-
thread-join	77	77	51	51	51	51
xalan	356	368	274	150	285	147
zxing	3614	-	2086	249	-	-

Table 3. Benchmarks result in seconds (Simplified Analysis Time).

	LV		GRIP-MHP			
Benchmark	(c) MHP CSEG	(d) MHP Anly	(f) PTS CSEG	(g) PTS Anly	(h) MHP CSEG	(i) MHP Anly
executor-futures	0.000	0.003	0.044	0.006	0.012	0.017
fj-kmeans	0.001	0.072	0.149	0.009	0.015	0.021
graphchi	0.004	17.462	0.630	0.016	0.235	0.875
h2	0.001	0.886	0.319	0.012	0.041	0.053
java-dinning-philosophers	0.001	0.472	0.070	0.009	0.031	0.260
jme	0.002	9.158	0.390	0.012	0.051	0.075
lusearch	0.007	623.754	0.850	0.026	0.283	3.049
pmd	0.001	0.491	0.458	0.007	0.024	0.033
thread-join	0.001	0.029	0.051	0.005	0.013	0.033
xalan	0.001	0.745	0.231	0.010	0.034	0.183
zxing	0.005	333.456	0.863	0.014	0.088	0.260

Table 4. Number of must-joins identified.

Benchmark	Must Joins GRIP-MHP
executor-futures	1
fj-kmeans	0
graphchi	5
h2	2
java-dinning-philosophers	7
jme	2
lusearch	2
pmd	0
thread-join	3
xalan	0
zxing	0

the GRIP-based configuration than under the LV-targeted baseline, with reductions that become more pronounced in highly concurrent benchmarks. This behavior directly mirrors the trend reported in the original study, where large applications that are intractable or extremely costly under full inlining or baseline configurations become manageable once semantic pruning is applied.

Execution time results exhibit an even more striking alignment with the original findings. The GRIP-MHP configuration consistently reduces the cost of the MHP analysis phase by orders of magnitude when compared to the LV baseline, especially in larger workloads. While the method introduces additional overhead in earlier phases, such as points-to and preprocessing steps, this cost remains marginal relative to the overall savings achieved during the main analysis. The reproduced measurements therefore reinforce the same cost trade-off structure described in the paper: a small upfront investment in graph construction leads to substantial reductions in downstream computational complexity.

Finally, the analysis of synchronization-related outputs (must-join detection) confirms that the method preserves the semantic precision necessary for its pruning strategy. The detected synchronization points align with expected thread coordination patterns across benchmarks, indicating that the algorithm is successfully capturing the concurrency structure required to safely eliminate infeasible interleavings. This mechanism provides the conceptual link between the structural reductions observed in the graphs and the performance improvements measured in execution time.

Overall, while minor numerical deviations can be observed, attributable to differences in runtime environment, hardware, and JVM behavior, the reproduced results maintain the same relative ordering, scaling behavior, and asymptotic trends reported in the original work. The consistency across all evaluated aspects strongly supports the validity of the original claims and confirms that GRIP-MHP remains effective in translating theoretical improvements into practical performance gains. Although some additional datasets and tables were not explicitly reported in detail, they were still included in a qualitative analysis, where it was possible to observe that they follow the same trends identified in the main metrics, further reinforcing the consistency of the results across the entire experimental suite.

Reproducing the paper's experiments was a highly positive experience. The provided Dockerfile, Makefile, and benchmark execution scripts significantly facilitated the setup and execution of the experimental pipeline. In particular, the generated outputs, ranging from detailed program logs to aggregated metric CSV files, made the subsequent analysis of results straightforward and systematic. Additionally, the clarity of the provided instructions contributed to an efficient artifact evaluation process, minimizing ambiguity and reducing the need for external intervention.