Analyzing Efficiency of High-Performance Applications

Bachelor's Thesis of

Dieu Lam Vo

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr.-Ing. Anne Koziolek
Second examiner:	Prof. Dr. Ralf Reussner
First advisor:	M.Sc. Larissa Schmid
Second advisor:	M.Sc. Timur Sağlam

18. December 2023 – 16. May 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 16. May 2024

. . . . . .

(Dieu Lam Vo)

Abstract

Leading computer systems in the area of high-performance computing (HPC) offer immense computing power with millions of cores. On the downside, the upkeep and energy requirements of such systems cost millions of euros each year. This makes the efficient utilization of the resources of HPC systems inevitable. However, the Bulk Synchronous Parallel model followed by most HPC applications could lead to unevenly distributed workloads due to unforeseen internal or external influences. Related research in this area often only considers the load imbalance in a way that allows no comparison between ranks. Our approach aims to consider a broader set of metrics to make an assessment of load imbalance possible in a more generalized context. This will help understand where inefficiencies appear in HPC applications.

Zusammenfassung

Führende Supercomputer im Bereich des High-Performance Computing (HPC) bieten immense Rechenleistung mit Millionen von Kernen. Allerdings kostet der Unterhalt und Energiebedarf solcher Systeme jedes Jahr Millionen von Euro. Dies macht eine effiziente Nutzung der Rechenressourcen von HPC-Systemen unabdingbar. Das von den meisten HPC-Anwendungen verwendete Bulk-Synchronous-Parallel-Modell könnte aber durch unvorhergesehene interne oder externe Einflüsse zu ungleich verteilter Arbeitslast führen. Verwandte Forschung in diesem Bereich berücksichtigt oft nur die sogenannte Load Imbalance auf eine Weise, die keinen Vergleich zwischen MPI-Rängen ermöglicht. Unser Ansatz zielt hingegen darauf ab, einen breiteren Satz von Metriken zu betrachten, um eine allgemeinere Bewertung der Load Imbalance zu ermöglichen. Unser Beitrag ermöglicht ein besseres Verständnis der Ursachen von Ineffizienzen in HPC-Anwendungen.

Contents

List of Figures

List of Tables

1. Introduction

According to the *Top500* list of supercomputers, today's computing systems reach peta- or even exascale computing power [49]. Eight out of the top ten systems provide over a million cores each, reaching up to 8,699,904 cores in the case of the first-spot supercomputer, *Frontier*. Furthermore, running a High-Performance Computing (HPC) system costs millions of euros a year [4, 11]. This raises the need for an efficient resource utilization of these massively parallel systems.

Most HPC applications follow the *Bulk Synchronous Parallel (BSP)* model, where calculations are synchronized with a barrier. These applications generally consist of *Single-Program Multiple-Data (SPMD)* executions and communicate via messages [8, 30, 7, 6]. Since the same calculation is performed for each data set, this leads to the assumption that these calculations are of approximately the same duration [8]. Unfortunately, this is no certainty due to unforeseen internal or external influences. Processes could be waiting on memory accesses or computing power in case other processes are using the same resources or have an unequal amount of operations if the workload is unequally distributed among processes [6]. The more processes are used for the execution of an application, the more processes potentially have to wait for the process with the longest execution time due to the synchronization after every time step. Especially in the field of HPC, this phenomenon of load imbalance wastes an enormous amount of resources [8]. For this reason, balancing workloads in between cores is essential for the high scalability of massively parallel systems [16]. However, there is no classification of different real-world HPC applications regarding their efficiency and especially their load imbalance. Furthermore, the current state of the art lacks an evaluation of which metrics are best suited for analyzing efficiency. Overall, this thesis aims to characterize real-world HPC applications for different workloads in terms of their resource utilization. To this end, we address the following research questions:

RQ1: Which metrics characterize HPC applications regarding their efficiency?

RQ2: How are the efficiency metrics linked to one another, and do they locate the same inefficiencies?

RQ3: How efficient are the selected real-world HPC applications?

RQ4: In the case of inefficiencies, where are those located, and how big is their impact on the overall performance of the HPC application?

To answer these research questions, this thesis is structured as follows: In chapter 2, we present the basic concepts of the thesis. Thereafter, chapter 3 distinguishes our work from related work. The detailed approach of the thesis is further specified in chapter 4. In

chapter 5, we answer our research questions and evaluate the results of our efficiency analysis[1]. Finally, chapter 6 concludes the most important findings of this thesis and gives an outlook on future work regarding efficiency optimization of HPC applications.

[1]Link to supplementary material: `https://doi.org/10.5281/zenodo.13854764`

2. Foundations

This chapter explains the fundamental terms of this thesis. Given that this thesis concerns the efficiency of High-Performance Computing applications, this chapter first has to introduce High-Performance Computing in general and its main terms in section 2.1. Furthermore, we define efficiency in the context of this thesis in section 2.2. Since this thesis aims to optimize selected High-Performance Computing applications in future work, we classify different strategies to handle potential inefficiencies in HPC applications in section 2.3.

2.1. High-Performance Computing

High-Performance Computing (HPC) is the field of computing with high processing power and/or huge memory space [6]. HPC clusters consist of interconnected computers, so-called nodes, which provide the processing and storage capacity of the supercomputer. The high computing power of HPC clusters itself relies on the parallel usage of many processors with a large quantity of cores [28]. Consequently, applications can only benefit from HPC clusters if they can utilize high parallelism. Typically, supercomputers execute Single-Program Multiple-Data (SPMD) instructions [8] and communicate via the *de facto* standard Message Passing Interface (MPI) [12, 6]. For that matter, the processes of the MPI application communicate via the communicator and are addressed by their unique rank number. Within the limits of this thesis, we focus on distributed memory parallelization with MPI, while shared memory parallelization with OpenMP remains out of consideration. HPC use cases include computationally intensive calculations, which are mostly scientific calculations and simulations, since these profit from massive parallelism [11, 6]. Some areas of application cover weather forecast, aerodynamics, tsunami, earthquakes, molecular dynamics, or even chess simulation. HPC applications can take hours, days, or months to complete. In line with this thesis, we focus on benchmarks with an execution time in the minute-to-hour range. Nevertheless, we assume that these findings are representative and can be transferred to longer execution times.

2.1.1. Bulk Synchronous Parallel model

Most HPC applications follow the Bulk Synchronous Parallel (BSP) model [6]. The BSP model divides a program in a series of supersteps which consist of three components each: local computation, communication and barrier synchronization (see Figure 2.1). First, each process runs its calculations. After that, partial results are communicated with other

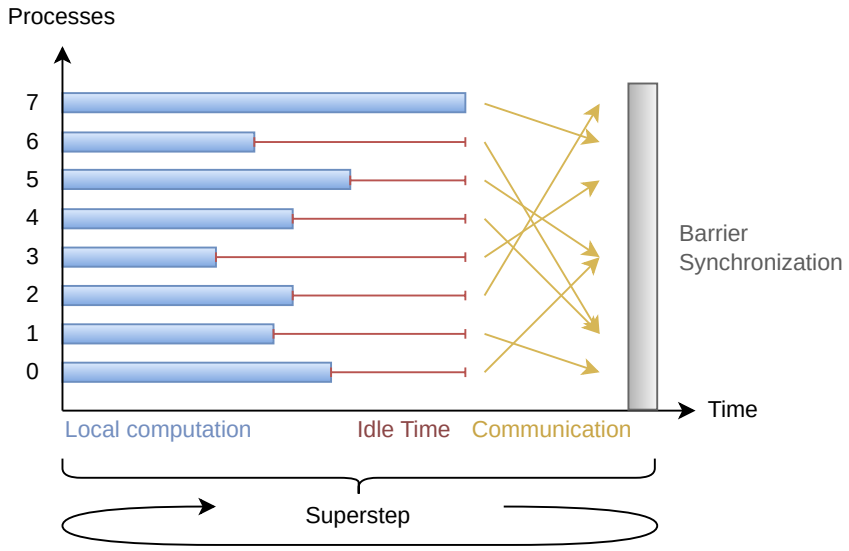


Figure 2.1.: Bulk Synchronous Parallel Model with Load Imbalance

processes via message passing, mostly with MPI. Finally, all processes are synchronized at the barrier, which leads to waiting phases if the workload is not balanced between all processes [5]. Even if single idle times might be short, they accumulate since the supersteps are executed numerous times. Idle times are the root cause for load imbalance in HPC applications [5].

2.1.2. Scalability

In the context of HPC, scalability describes how well a system or an application handles increasing workloads [44]. In order to scale an application, there are two different methods: strong scaling and weak scaling. Strong scaling applications execute a fixed-sized problem with a different number of processes, whereas weak scaling applications linearly increase the total workload subject to the number of ranks. Consequently, the problem size per process remains constant. In this thesis, we examine weak scaling applications.

2.1.3. Slurm

Slurm is an open-source cluster management and job scheduling system [24]. In line with this thesis, we execute HPC applications on the bwUniCluster 2.0 which uses Slurm as workload-managing software [10]. Execution scripts are submitted as batch jobs to Slurm which then queues and runs these jobs. Batch jobs define, for example, the requested time, memory space, and number of nodes and cores.

2.1.4. Profiling of HPC Applications

The Scalable Performance Measurement Infrastructure for Parallel Codes Score-P is a performance analysis tool for the profiling and tracing of HPC applications [27, 6]. Score-P, in particular, can be used with other performance analysis tools like Scalasca, Vampir and TAU [27]. In line with this thesis, we use Score-P for profiling in conjunction with the Performance Application Programming Interface (PAPI) [38]. For analyzing selected HPC applications, we use the default compiler instrumentation of Score-P, which instruments every function. During the execution of the HPC application, Score-P saves the measurement profiles in the CUBE4 data format [42]. These profiles can later be viewed in CUBEGUI [43].

2.2. Efficiency

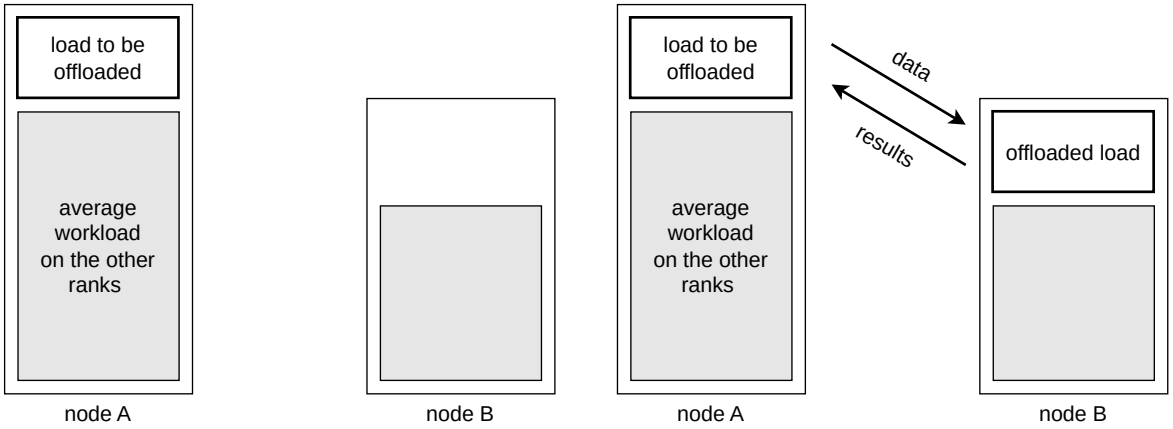
With the trend of massively parallel systems and their high maintenance costs [4, 11], using these systems and their resources efficiently is getting increasingly crucial. This thesis deals with the efficiency analysis of HPC applications and limits itself thereby to the efficiency regarding hardware resources while leaving energy efficiency out of consideration. For this reason, we define different efficiency metrics in section 4.1. Furthermore, we focus on the evaluation of the implementation of HPC applications instead of analyzing the methods and algorithms they use.

2.3. Efficiency Optimization Strategy

When inefficiencies occur in HPC applications, the goal is to reduce or even better eliminate these inefficiencies. A further goal of optimizing HPC applications is to lower communications costs between processors [45]. For these purposes, there are different efficiency optimization strategies. The most important approaches are explained in this section.

2.3.1. Load Balancing

Load balancing is an efficiency optimization strategy based on workload reallocation. Load balancing is a static approach that is executed before runtime, for example, with the mesh partitioner METIS [26], or during runtime at predetermined synchronization points [1]. At these synchronization points, the workload is equally distributed among the cores in case of a workload imbalance. However, the load balancing algorithm usually needs to be implemented for every application anew [1], which requires an understanding of the application's code. Thus, load balancing is a time-consuming approach.



(a) State before Offloading **(b)** State after Offloading

Figure 2.2.: Offloading

2.3.2. Hardware Disaggregation

As the name suggests, hardware disaggregation reallocates hardware resources to different jobs. Although this approach promises good results, it is inconvenient since the realization requires dedicated hardware and is very costly [15].

2.3.3. Adaptive MPI

Adaptive MPI (AMPI) [3] is an implementation of the MPI standard [1], such as OpenMPI or MPICH. AMPI is based on Charm++ [25, 34], a dynamic load balancing approach through permanent object migration. Consequently, a workload is permanently moved to another processor until the processes get rebalanced and a potential next migration step is evoked [1]. Furthermore, Charm++ relies on the concept of processor virtualization. In the case of AMPI, MPI programs are allocated on virtual MPI processors (VPs) [23, 34], which are then mapped to the physical processors by the run-time system.

2.3.4. Software Resource Disaggregation

Software resource disaggregation is an efficiency optimization strategy based on elastic resource allocation via *Function-as-a-Service*, as proposed by Copik et al. [15]. In most cases, nodes do not use every single core available for their jobs. This approach utilizes these idle cores to execute fine-grained serverless functions, as depicted in Figure 2.2. However, the communication overhead with remote services and the overhead to offload jobs and aggregate the results need to be taken into account. One advantage with regard to Hardware Disaggregation is that Software Resource Disaggregation needs no dedicated hardware. Thus, Software Resource Disaggregation is less expensive and easier to realize.

3. Related Work

This chapter gives an overview of related work and distinguishes the approach of this thesis from related work. Section 3.1 deals with related work regarding different efficiency metrics, whereas section 3.2 concerns itself with the efficiency analysis of related work with regard to our work.

3.1. Efficiency Metrics

The most widely used efficiency metric is load imbalance [30, 8, 33, 1, 5, 7], which provides a factor for the average idle time of ranks. This metric ranges from one to the number of ranks, which makes it incomparable for executions with different numbers of ranks. DeRose et al. [16] go even further and close this gap. Instead of load imbalance, they define two new metrics, namely imbalance percentage and imbalance time, which simplify the estimation of the impact of inefficiencies. Regarding future efficiency optimization of HPC applications, the imbalance time metric ranks functions according to the profitability of optimization. Moreover, Linford et al. [30] differentiate between global and local imbalance metrics at the process level. Furthermore, Faulk et al. [18] incorporate the development time in addition to the execution time in their efficiency analysis. However, we have no access to the development time of established HPC applications and cannot consider this metric. In addition, Tallent et al. [47] define the metrics imbalance waste and floating point waste. Imbalance waste refers to the waste due to imbalance, and floating point waste sets the ideal number of FLOPS in relation to the actual number. Further metrics include CPU and memory utilization and code size [14].

In contrast to the related work, this thesis presents a wider variety of efficiency metrics, evaluates them, and links them to each other. Furthermore, all defined metrics are measured for the presented HPC applications. In the best case, more inefficiencies can be identified with a broader variety of metrics, which leads to a more detailed characterization of HPC applications.

3.2. Efficiency Analysis

For analyzing efficiency in HPC applications, there are two different methods for collecting performance data, namely profiling and tracing. Böhme et al. [5, 7] and Linford et al. ([30]) use the tracer *Scalasca*, Tallent et al. [47] use the profiler *HPCToolkit* and Zirwes et al. [53]

use the *Score-P* measurement infrastructure [27] in combination with Vampir. In this thesis, Score-P is used in conjunction with the performance report explorer *CUBE* [42] for profiling. Furthermore, hardware counters are read out with *PAPI* [38].

Other works evaluate HPC applications that are analyzed in this thesis. These applications include *MicroPP* [1], *sam(oa)²* [22, 32], *LAMMPS* [48, 19] and *OpenFOAM* [53].

4. Approach

Prior work raises the assumption that HPC applications tend to have inefficiencies regarding their resource utilization [22, 53, 52, 34, 13, 15]. Not addressing this problem of inefficiency leads to a significant waste of computing potential. This raises the main research question of this thesis: How can real-world HPC applications be characterized for different workloads in terms of their resource utilization? This leads to two successive goals of this thesis. The first goal is to identify appropriate efficiency metrics (see section 4.1) and then use them to analyze a selection of real-world HPC applications (see section 4.2). In the context of this thesis, we examine the efficiency of HPC applications with regard to future efficiency optimization. For this reason, we evaluate the impact of inefficiencies by means of our defined metrics.

4.1. Efficiency Metrics

Since HPC applications run in parallel, especially load imbalance between ranks can be a bottleneck as explained in subsection 2.1.1. This section introduces metrics that reveal how efficient HPC applications are. These metrics are later used for our efficiency analysis. Related works [8, 16, 30, 18, 47] propose different metrics to measure the efficiency of HPC applications. We bring together a selection of different efficiency metrics which are later evaluated. In line with this thesis, requirements for useful efficiency metrics include computability and comparability.

4.1.1. Collected Metrics

The following metrics are collected during the execution of HPC applications, either by Slurm or by Score-P in conjunction with PAPI. These metrics are collected to derive more complex metrics afterward.

Total Execution Time is the time a job totally needs. In the example in Figure 4.1, the total execution time equals the sum of the individual execution times of the eight ranks.

Wall-clock time specifies the time a job requires from start to finish. In the case of our HPC applications, this is limited by the rank with the longest execution time. In Figure 4.1, rank 7 takes the longest and, consequently, defines the wall-clock time.

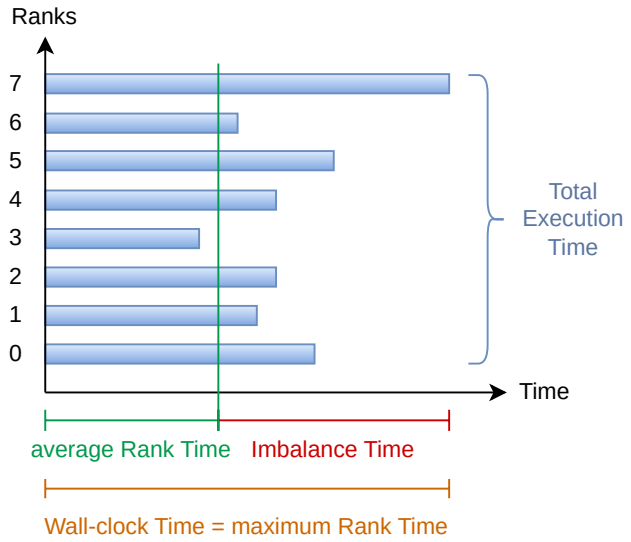


Figure 4.1.: Relation of Efficiency Metrics

Double Precision Operations indicate the number of double precision floating point operations and are read out with the PAPI metric *PAPI_DP_OPS*.

Load/Store Instructions provide insight about how many times information is read from or written to the main memory. The load and store instructions are read out with the PAPI metric *PAPI_LST_INS*.

Memory Utilization shows how much memory space is used for the job and is directly provided by the workload manager Slurm.

4.1.2. Derived Metrics

The following metrics are more complex and are derived by means of the basic collected metrics.

Floating Point Operations Per Second (FLOPS) describes the computing power of a system or a processor. As the name suggests, FLOPS measure floating-point operations per second. In order to calculate the used FLOPS, the floating point operations measured with the PAPI metric *PAPI_DP_OPS* are divided by the wall-clock time.

$$\text{FLOPS} = \frac{\text{double precision operations}}{\text{wall-clock time}}$$

FLOPS per Rank relate the FLOPS to the number of ranks. This way, FLOPS of executions with different numbers of ranks are better comparable.

$$\text{FLOPS per rank} = \frac{\text{FLOPS}}{\text{number of ranks}}$$

CPU Utilization indicates how much CPU time is effectively used in percent. The closer the wall-clock time gets to the average rank time, the closer the CPU utilization gets to 100 %. The CPU utilization is also provided by Slurm. However, Slurm calculates this metric with the assumption that all requested cores for the job are used. This is not the case in our executions, since we just request the whole node for our performance evaluation to remove the impact of other calculations on the same node. For this reason, we calculate the CPU utilization on our own.

$$\text{CPU utilization} = \frac{\text{total execution time}}{\text{wall-clock time} \times \text{number of processors}}$$

Communication Time is the time spent on communication. In our case of MPI applications, the communication time amounts to the time needed for MPI calls.

Communication Time Percentage relates the communication time to the total execution time.

$$\text{communication time percentage} = \frac{\text{communication time}}{\text{total execution time}}$$

Weak Scaling Efficiency describes the efficiency of weak scaling applications [44]. Weak scaling applications execute the same workload per rank on different numbers of ranks. In the ideal case, the weak scaling efficiency remains constant at 1. However, the overhead through splitting jobs into several ranks or the communication and idle time reduce the weak scaling efficiency as a rule.

$$\text{weak scaling efficiency} = \frac{\text{serial execution time}}{\text{parallel execution time}}$$

Load Imbalance indicates how equally the workloads are distributed among the cores [1, 33] and sets the execution time of the most loaded rank in relation to the average execution time of all ranks. In the case of perfectly balanced workloads, the load imbalance is equal to 1. The more the workloads are balanced, the closer the load imbalance gets to 1. The maximum value of load imbalance is the number of ranks, which is reached when all work in a parallel execution is done by one rank [1]. Although this efficiency metric is the most widely used, we can deduce that the load imbalance can only be poorly compared to the load imbalance of an execution with a different number of ranks.

$$\text{load imbalance} = \frac{\text{maximum rank time}}{\text{average rank time}} \geq 1$$

Imbalance Percentage indicates the percentage of time that is caused by the imbalance [16]. Contrary to the load imbalance metric, this metric allows the comparison of executions with different numbers of ranks. Although the following formula is not defined for serial executions, it is evident that serial operations always have an imbalance percentage of 0 %. Thus, the imbalance percentage alone does not give information about whether optimizing calculations would be profitable.

$$\text{imbalance percentage} = \frac{\text{maximum rank time} - \text{average rank time}}{\text{maximum rank time}} \times \frac{\text{number of ranks}}{\text{number of ranks} - 1}$$

Imbalance Time indicates how much time can be saved, at most, if the cores were perfectly balanced [16]. Hence, the imbalance time in combination with the imbalance percentage offers optimal guidance whether optimizing an operation might be profitable. It is crucial to consider the imbalance time while optimizing an HPC application, since the overhead of optimization might even decrease the efficiency. The imbalance time of an execution is the difference between maximum and average rank time, as depicted in Figure 4.1.

$$\text{imbalance time} = \text{maximum rank time} - \text{average rank time}$$

Imbalance Time Percentage relates the imbalance time to the average rank time in order to estimate the overall impact of the load imbalance.

$$\text{imbalance time percentage} = \frac{\text{imbalance time}}{\text{average rank time}}$$

4.2. HPC Applications

This section presents the HPC applications that are analyzed in this thesis. We select these HPC applications due to the fact that they were already used as case studies in related work and suffer from load imbalance. Furthermore, these HPC applications can be optimized in future work with the acquired knowledge of their load imbalance.

4.2.1. Micropp

Micropp[1] is a hybrid CPU/GPU Finite Element Method (FEM) code for solving solid mechanics problems of composite materials [21], implemented by Giuntoli [20]. For this purpose, Micropp models the microscale behavior of these composite materials by computing FEM calculations on structured grids [21]. Furthermore, Micropp is written in C++ and supports coupling with different macro-scale codes written in C, C++, or Fortran in order to create a FE2 multiscale application [**giuntoli_hybrid_nodate**, 21]. In recent research, Micropp has been coupled with the macro-scale code Alya [**giuntoli_hybrid_nodate**, 21, 20]. However, Micropp can also be run in standalone mode and provides benchmarks for this purpose [21]. This thesis focuses on Micropp without coupling.

4.2.2. eSam(oa)^2

eSam(oa)^2[2] is an acronym for Elastic Space-Filling Curves and Adaptive Meshes for Oceanic And Other Applications. It is adapted by Mo-Hellenbrand et al. [22] and is the elastic version of the HPC application **sam(oa)^2**[3] written in Fortran by Meister et al. [31]. sam(oa)^2 is a

[1] `https://github.com/gagiuntoli/Micropp`, Commit Hash: b0b44ea
[2] `https://github.com/mohellen/eSamoa`, Commit Hash: 20662c1
[3] `https://github.com/meistero/Samoa`, Commit Hash: ed9904a

hybrid OpenMP/MPI, Adaptive Mesh Refinement (AMR) framework for parallel simulation on dynamically adaptive 2D triangular meshes based on the Sierpinski space-filling curve (SFC) traversal [17, 40, 32, 22, 31]. Furthermore, sam(oa)^2 performs dynamic load balancing after each time step [22, 31, 32]. sam(oa)^2 provides two text cases, namely porous media flow and tsunami wave propagation [40, 32]. While the porous media flow is represented in the SPE10 oil exploration benchmark [31], the tsunami simulation benchmark is based on the 2011 Tohoku tsunami in Japan [32]. Mo-Hellenbrand et al. focus on the tsunami simulation in their elastic version of sam(oa)^2. Furthermore, Meister et al. [32] execute both benchmarks using sam(oa)^2 with multiple configurations. However, their performance data is not extensive regarding the use of different metrics.

4.2.3. LAMMPS

LAMMPS[4] is an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator and is a classical molecular dynamics (MD) code in the field of Computational Chemistry and Materials Science. Generally, MD simulations model the behavior of molecules in different states of matter: solid, liquid, and gas [19, 39] and derive advantage from massively parallel computations [48]. LAMMPS in particular is written in C/C++, Python, and Fortran and supplementary material is provided on its website [29]. LAMMPS offers a wide variety of benchmarks and log files for executions of each benchmark on different machines with varying numbers of processors for comparison. The Lennard-Jones (LJ) benchmark is often used in related work [48, 39, 34]. Furthermore, LAMMPS supports load balancing with different load balancing algorithms [48, 34].

4.2.4. OpenFOAM

OpenFOAM[5] is an acronym for Open Source Field Operation and Manipulation. It is one of the most used applications for Computational Fluid Dynamics (CFD) [51, 37, 53, 41, 35] and is updated semiannually. OpenFOAM provides different solvers and multiple tutorial cases using these. The most commonly used solvers of OpenFOAM in related work cover `reactingFoam` [53], `pimpleFoam` [53], and variations of `interFoam` [35, 41].

[4] `https://github.com/lammps/lammps`, Commit Hash: 4c99249
[5] `https://develop.openfoam.com/Development/openfoam`, Version: v2312

5. Evaluation

The main goal of this thesis is to characterize real-world HPC applications regarding their resource utilization efficiency. For this purpose, we first identify appropriate efficiency metrics and then analyze real-world HPC applications with these efficiency metrics, as described in chapter 4. In this chapter, we evaluate the results of this thesis. To this end, we first set our goals in context with our defined research questions and metrics using a Goal-Question-Metric (GQM) plan, which is presented in section 5.1. Afterward, section 5.2 describes the methodology of this thesis and the tools used. Thereafter, the HPC applications are analyzed in section 5.3, ordered by HPC application. To this end, benchmarks of an HPC application are first analyzed on their own and, subsequently, compared to one another. The assumptions and limitations of this thesis are discussed in section 5.5. Lastly, section 5.6 depicts the threats to the validity of this thesis.

5.1. Goal-Question-Metric

The Goal-Question-Metric method is used to structure a development process [2]. For that matter, we first establish goals. Then we define suitable questions that reveal whether the corresponding goal is met. And lastly, we answer these questions using our defined metrics.

This thesis is based on the goals, questions, and metrics shown in Table 5.1. Overall, the main goal of this thesis is to characterize real-world HPC applications for different workloads in terms of their resource utilization. For this purpose, we establish two subgoals: the identification of appropriate efficiency metrics and the efficiency analysis of real-world HPC applications. In order to validate whether the identified efficiency metrics are appropriate, we link them to each other and evaluate whether they locate the same inefficiencies. Furthermore, we analyze real-world HPC applications with the defined metrics in section 4.1.

5.2. Methodology

The aforementioned HPC applications are executed on the *bwUniCluster 2.0* [9] on Intel Xeon Gold 6230 processors. The bwUniCluster provides computational nodes with 2 sockets and 40 cores each and uses *Slurm 23.02.6* [24] as a job scheduling system. Generally in the field of computer science and especially on the bwUniCluster 2.0, the unit of memory spaces is specified in megabytes (MB), although accurately, mebibytes (MiB) are used [10]. In the

Goal 1	Identify appropriate efficiency metrics
Question 1.1	Which metrics characterize HPC applications regarding their efficiency?
Metrics	Similarity between metrics
Question 1.2	How are the efficiency metrics linked to one another?
Metrics	Similarities between metrics
Question 1.3	Do the efficiency metrics locate the same inefficiencies?
Metrics	Similarities between metrics

(a) Goal 1: Identify appropriate efficiency metrics

Goal 2	Analyze real-world HPC applications in regard to resource efficiency
Question 2.1	How efficient are the selected real-world HPC applications?
Metrics	FLOPS, Flops per Rank, CPU Utilization, Memory Utilization, Communication Time, Communication Time Percentage, Weak Scaling Efficiency, Load Imbalance, Imbalance Percentage, Imbalance Time, Imbalance Time Percentage
Question 2.2	In case of inefficiencies, where are those located?
Metrics	CPU Utilization, Weak Scaling Efficiency, Load Imbalance, Imbalance Percentage, Imbalance Time
Question 2.3	In case of inefficiencies, how big is their impact on the overall performance of the HPC application?
Metrics	CPU Utilization, Weak Scaling Efficiency, Load Imbalance, Imbalance Percentage, Imbalance Time, Imbalance Time Percentage

(b) Goal 2: Analyze real-world HPC applications in regard to resource efficiency

Table 5.1.: GQM plan of this thesis

context of this thesis, we follow this convention. Unless mentioned otherwise, the HPC applications are compiled with the *GNU compiler suite 11.2.0* and run with *OpenMPI 4.1*, an open-source MPI implementation.

The HPC applications are profiled with the measurement infrastructure *Score-P 7.1* [27] in conjunction with the *PAPI* [38]. For every HPC application, we use the default compiler instrumentation of Score-P, which instruments every function. Score-P saves the measurement profiles in the CUBE4 data format, which can be viewed in *CUBEGUI 4.8.2* [43]. After extracting the performance data from the measurement profiles, we calculate our defined metrics for every function of the HPC application as aforementioned (see section 4.1). Finally, all metrics are visualized with the Python library *seaborn 0.13.2* [50]. All scripts for extracting, calculating, filtering, and plotting the metrics are implemented in *Python 3.11.7*.

The chosen benchmarks of the HPC applications are executed with variable problem sizes and a variable number of ranks each. In order to scale the problem size of an HPC application, we use weak scaling. Regarding the number of ranks, we choose five different powers of two, with 64 as the maximal number of ranks to cover the execution on two different nodes.

These executions give an insight into the communication between nodes in the efficiency analysis. The other executions with 32 or fewer ranks are run on one node. Furthermore, we run each test case serially to provide a reference value for the weak scaling efficiency metric. In line with this thesis, each rank of an execution is run on a different core. Consequently, we use n cores for an execution with n ranks. However, we request the whole node for the execution of the HPC applications to minimize the influence of other users on the bwUniCluster 2.0. Furthermore, each configuration of a benchmark is executed five times. In order to prevent variation through noise, we use the mean of the five repetitions to calculate our metrics. For all our benchmarks, the coefficient of variation is below 3 %.

5.3. Efficiency Analysis

In this section, the collected performance data is analyzed ordered by HPC application. Given that three-dimensional plots are inconvenient in the context of a thesis, our metrics are visualized on the one hand as a function of the number of ranks and on the other hand as a function of problem size. Furthermore, the metrics of memory utilization, communication time, and communication time percentage are only examined at the benchmark level whereas all other metrics can also be evaluated at the function level. Since analyzing every function of an HPC application goes beyond the scope of this thesis, we limit our efficiency analysis to the functions of the HPC applications whose execution time is greater than 5 % of the total execution time of the respective function.

5.3.1. Micropp

In the case of the HPC application Micropp, we choose the only two provided benchmarks communicating with the MPI standard, namely `mpi-load-balance` and `benchmark-sc2019`. Both benchmarks accept the same three input parameters. The first parameter describes the side length of the simulated cube. The second parameter strongly scales the size of the problem. In the execution of the benchmarks, the second parameter is divided by the number of ranks. In order to weakly scale the problem size, we increase the input value of this parameter linearly with the number of ranks. The third parameter defines the number of time steps. We use the default setting of this parameter, which is set to 10. In the context of this thesis, we scale the problem size with different cube sizes set by the first parameter.

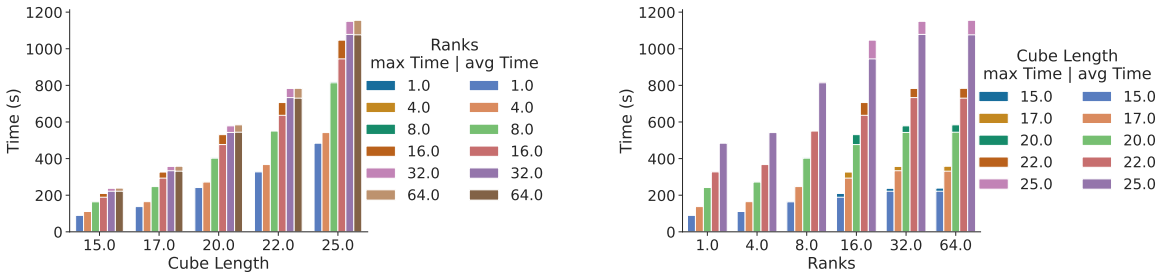
5.3.1.1. `mpi-load-balance` Benchmark

In the following, we analyze the results of the performance measurement of the `mpi-load-balance` benchmark. We examine the only two functions of this benchmark that have a higher execution time percentage than 5 %. As depicted in Table 5.2, the function `void ell_mvp(...)` has a significantly higher time percentage than 5 %, with over 80 %. Thereby, it is the decisive factor for the performance results of this benchmark.

Function	Time Percentage
`void ell_mvp(const ell_matrix*, const double*, double*)`	78.40 % to 87.07 %
`MPI_Barrier`	0.000 02 % to 9.57 %
other functions	<5 %

Table 5.2.: Time Percentage of `mpi-load-balance` functions

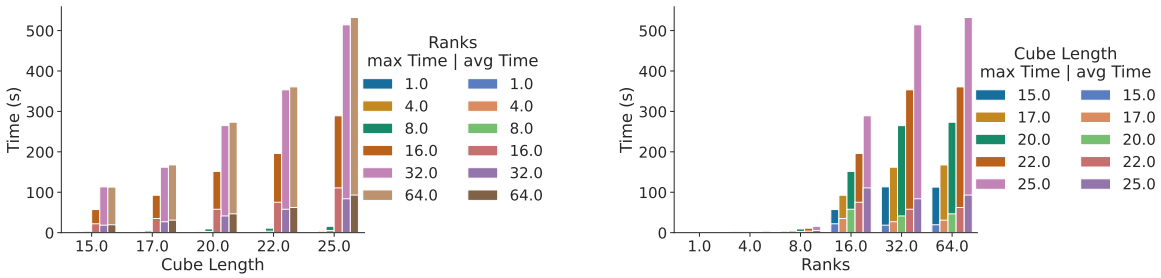
Time Figure 5.1 shows that the average time of `ell_mvp` is nearly identical to its maximum time. This reveals that this function is well-balanced. At the barrier, however (see Figure 5.2), the difference between average and maximum time is relatively large. This shows that there are high idle times in this function. Noticeably, executions with eight ranks or fewer spend nearly no time at the barrier.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.1.: Time for `mpi-load-balance:void ell_mvp(...)`

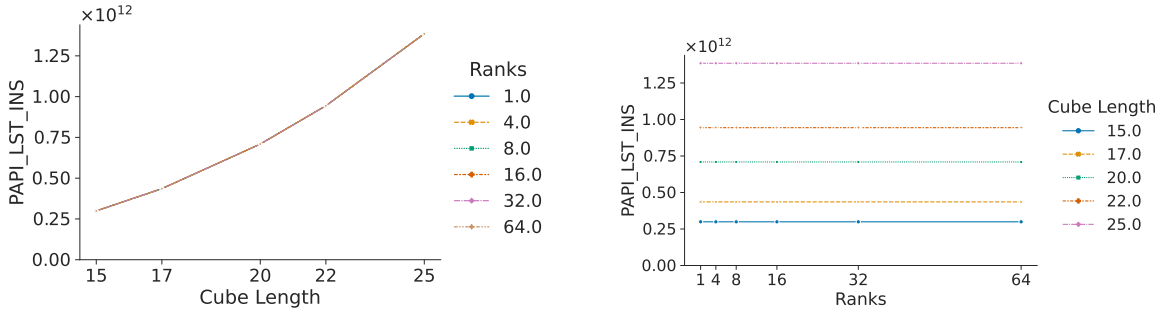


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.2.: Time for `mpi-load-balance:MPI_Barrier`

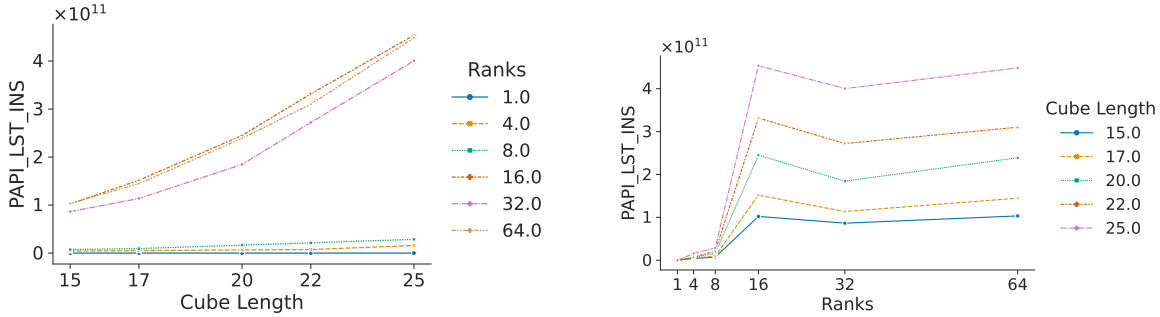
Load/Store Instructions For both functions `ell_mvp` (see Figure 5.3) and `MPI_Barrier` (see Figure 5.4), the load and store instructions rise above 1.25×10^{12} with increasing cube length. However, the load and store instructions of `ell_mvp` stay constant for different numbers of ranks. At the MPI barrier, the load and store instructions remain close to zero for executions with one, four, and eight ranks. At 16 ranks, the load and store instructions explode to a range from 1×10^{11} to 4×10^{11}, where they remain mostly constant for higher numbers of ranks.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.3.: Load/Store Instructions for `mpi-load-balance:void ell_mvp(...)`

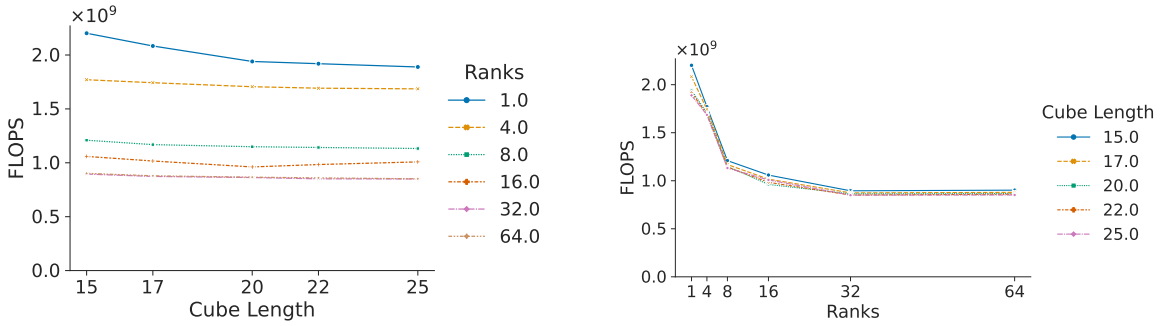


(a) Variable problem size

(b) Variable number of MPI ranks

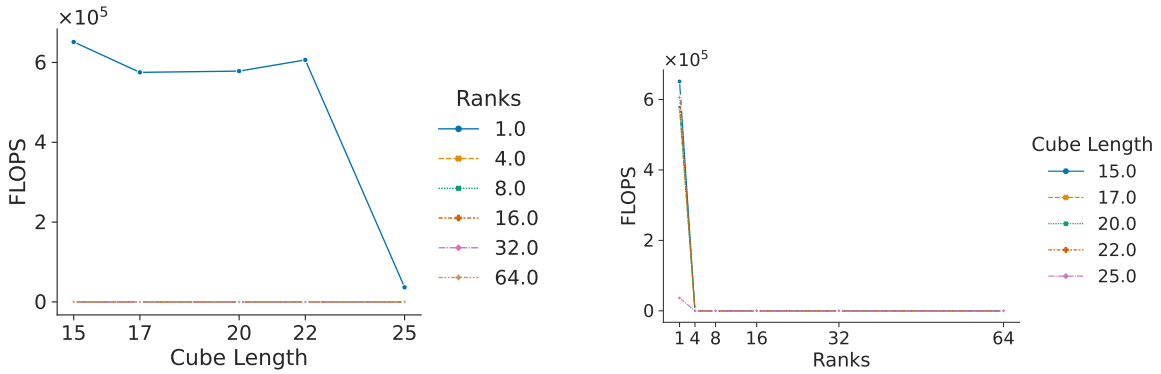
Figure 5.4.: Load/Store Instructions for `mpi-load-balance:MPI_Barrier`

FLOPS While the number of FLOPS nearly remains constant for different cube lengths for the `ell_mvp` function (see Figure 5.5a), they exponentially decrease for rising numbers of ranks, as depicted in Figure 5.5b. The FLOPS reach their maximum value of approximately $2{\times}10^9$ FLOPS and converge to roughly $1{\times}10^9$ FLOPS. At the MPI barrier, shown in Figure 5.6, the FLOPS for all parallel executions are nearly 0. Only the curve of the serial execution has around 6×10^5 FLOPS until it drops to nearly 0 FLOPS for a cube length of 25. Only parallel executions potentially have to wait at the barrier, which explains the fact that they have a smaller number of FLOPS. The longer the ranks have to wait at the barrier, the smaller the value of FLOPS gets.



(a) Variable problem size **(b)** Variable number of MPI ranks

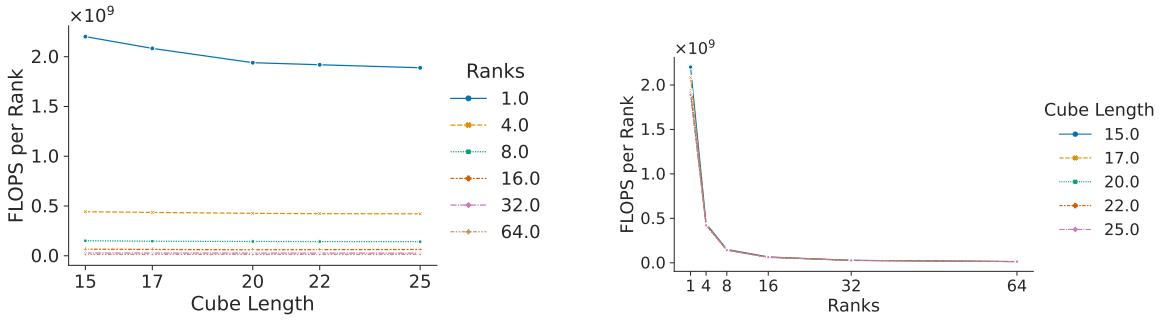
Figure 5.5.: FLOPS for `mpi-load-balance:void ell_mvp(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

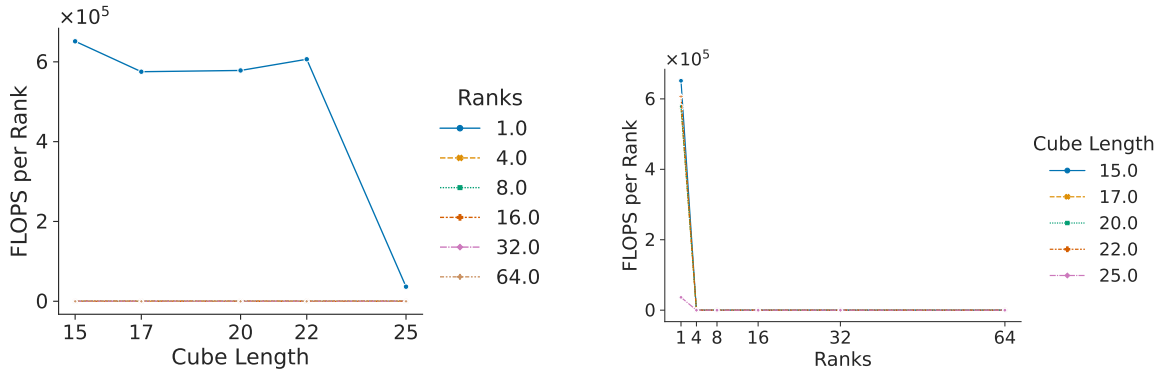
Figure 5.6.: FLOPS for `mpi-load-balance:MPI_Barrier`

FLOPS per Rank FLOPS per rank are more comparable for executions with different numbers of ranks. At the barrier, Figure 5.8 looks identical to Figure 5.8, which represents the FLOPS at the barrier. This is due to the fact that parallel executions have 0 FLOPS at the barrier, and the FLOPS of the serial executions are divided by 1 to receive the FLOPS per rank. On the other hand, the FLOPS of `ell_mvp` drop almost immediately to zero for an increasing number of ranks.



(a) Variable problem size **(b)** Variable number of MPI ranks

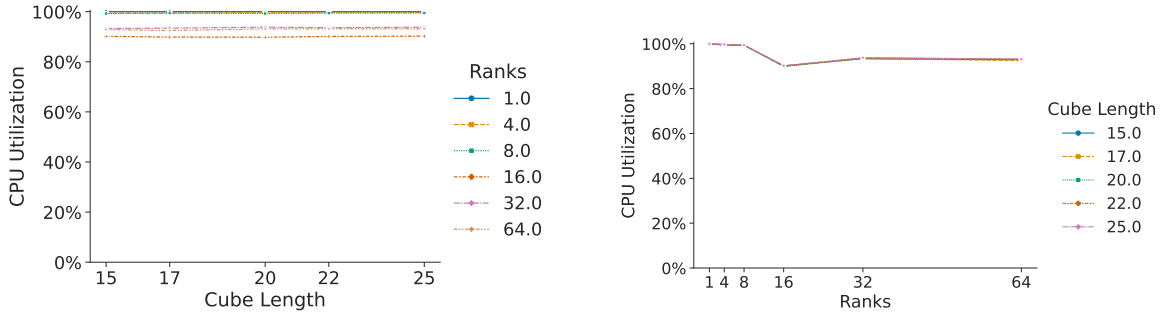
Figure 5.7.: FLOPS per Rank for `mpi-load-balance:void ell_mvp(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

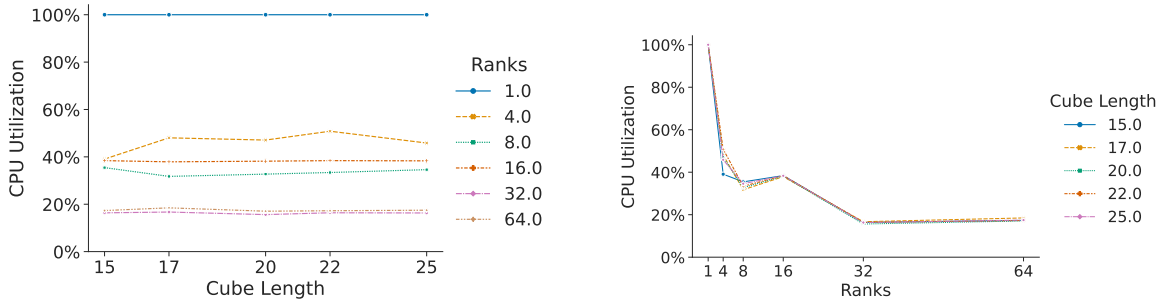
Figure 5.8.: FLOPS per Rank for `mpi-load-balance:MPI_Barrier`

CPU Utilization As shown in Figure 5.9, `ell_mvp` uses its CPU resources almost ideally. The CPU utilization for `MPI_Barrier`, however, drops to 40 % for the executions with four ranks and further declines for increasing numbers of ranks (see Figure 5.10b).



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.9.: CPU Utilization for `mpi-load-balance:void ell_mvp(...)`

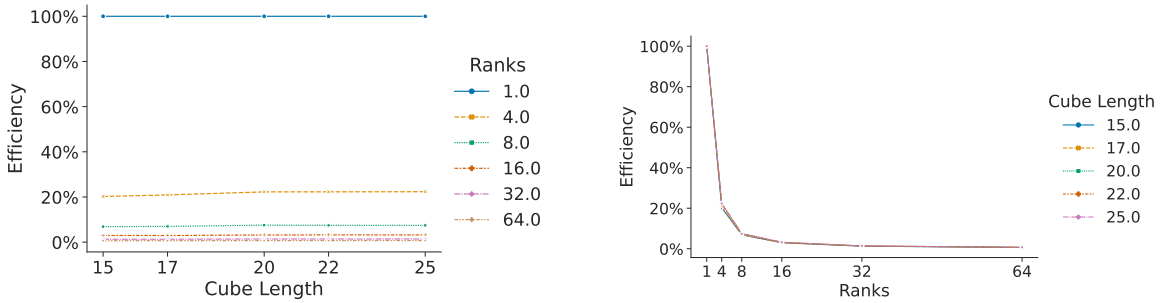


(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.10.: CPU Utilization for `mpi-load-balance:MPI_Barrier`

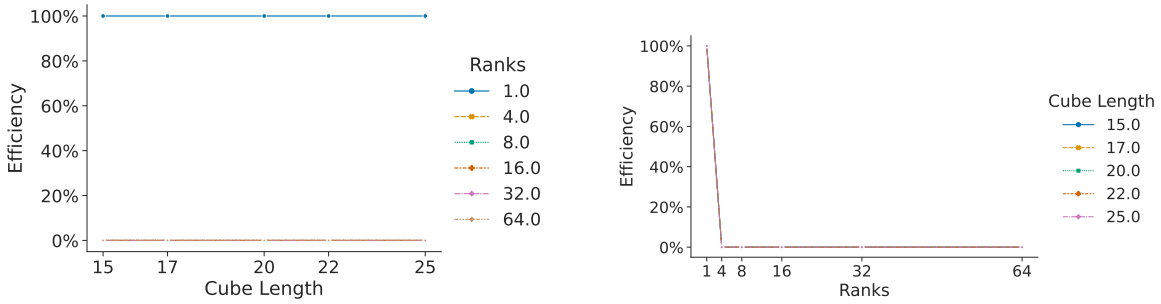
Weak Scaling Efficiency Figure 5.11 and Figure 5.12 show that the weak scaling efficiency is nearly 0 % for parallel executions. While the weak scaling efficiency stays constant for

different problem sizes, we lose nearly 80 % of efficiency by scaling the number of ranks from one to four. Evidently, this curve converges to zero. On the contrary, the weak scaling efficiency drops immediately to zero at the MPI barrier for parallel executions, as shown in Figure 5.12. This is due to the fact that ranks are waiting for ranks with a longer execution time. In the ideal case, the weak scaling efficiency would remain constant at 100 %.



(a) Variable problem size **(b)** Variable number of MPI ranks

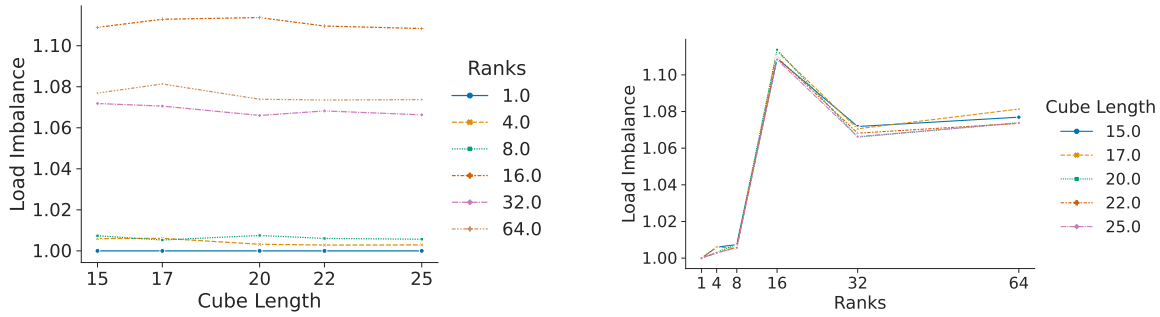
Figure 5.11.: Efficiency for `mpi-load-balance:void ell_mvp(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

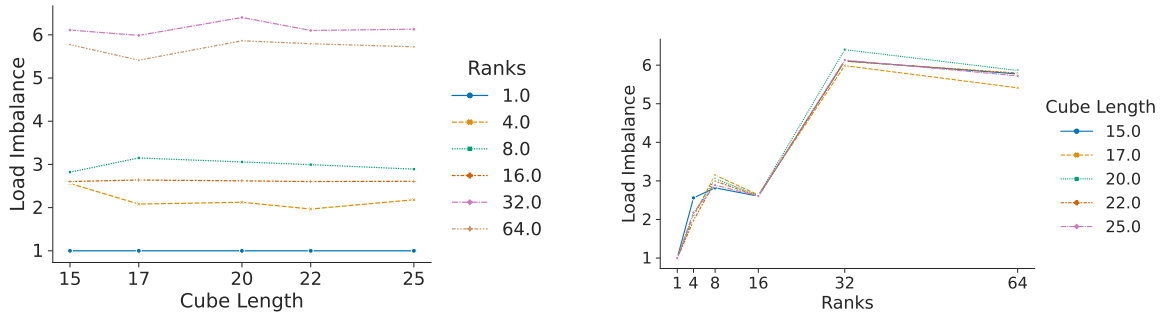
Figure 5.12.: Efficiency for `mpi-load-balance:MPI_Barrier`

Load Imbalance Although the load imbalance of `ell_mvp` peaks at the executions with 16 ranks (see Figure 5.13), this function still seems nearly perfectly balanced. For varying cube lengths (see Figure 5.13a), the load imbalance remains nearly unchanged. At the barrier, however, the load imbalance reaches six for the executions with 32 and 64 ranks. And the executions with 16 ranks locally peak at three, as shown in Figure 5.14b. Furthermore, the load imbalance remains constant for different cube lengths.



(a) Variable problem size **(b)** Variable number of MPI ranks

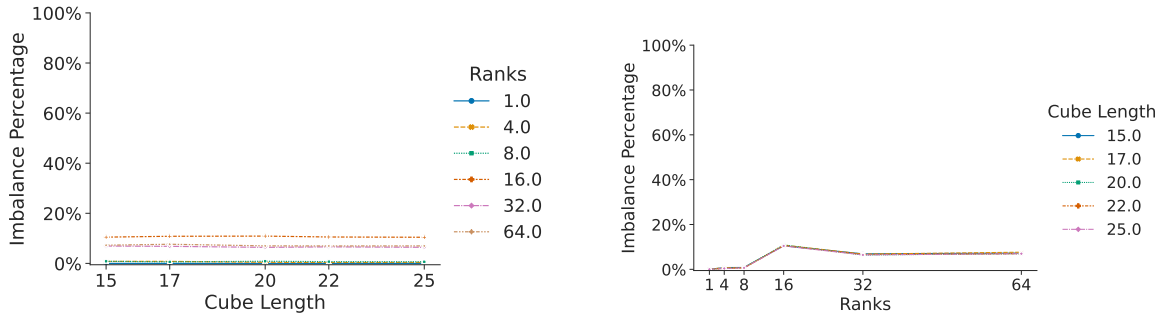
Figure 5.13.: Load Imbalance for `mpi-load-balance:void ell_mvp(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

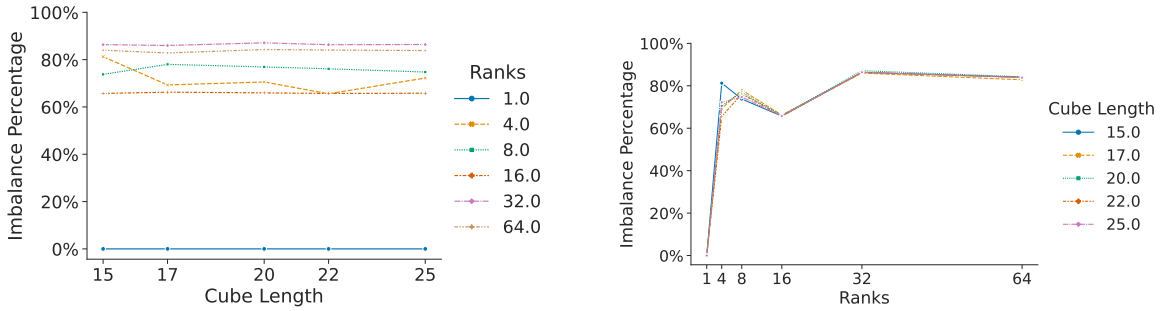
Figure 5.14.: Load Imbalance for `mpi-load-balance:MPI_Barrier`

Imbalance Percentage The observation of the load imbalance metrics can be transferred to the imbalance percentage. However, the imbalance percentage relates the load imbalance to the number of ranks. While the imbalance percentage of `ell_mvp` stays below 10 % for all executions (see Figure 5.15), the imbalance percentage is over 65 % at the barrier for all parallel executions.



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.15.: Imbalance Percentage for `mpi-load-balance:void ell_mvp(...)`

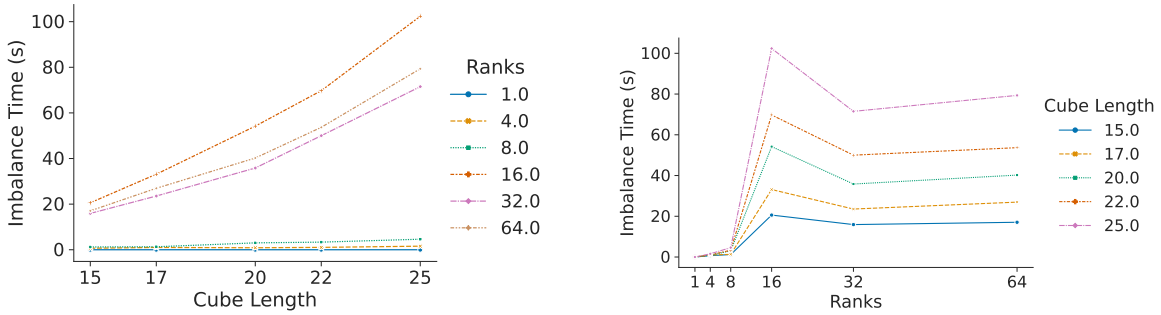


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.16.: Imbalance Percentage for `mpi-load-balance:MPI_Barrier`

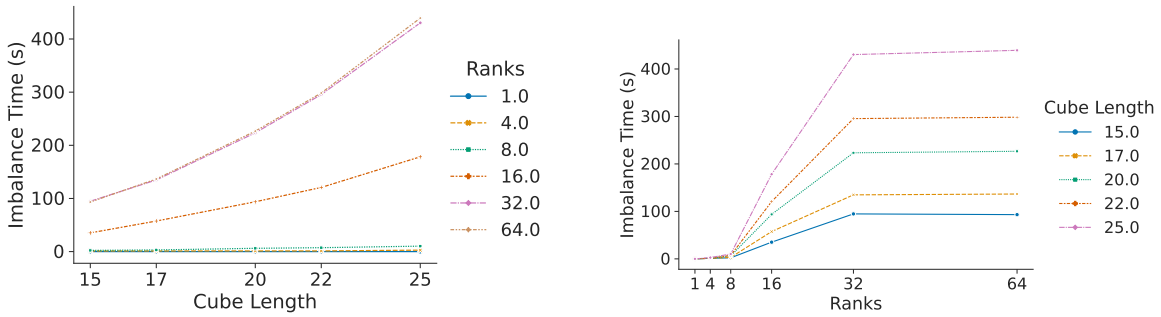
Imbalance Time The imbalance time of `ell_mvp` is nearly zero for executions using eight or fewer ranks and peaks for executions with 16 ranks (see Figure 5.17). Furthermore, the imbalance time rises with increasing cube lengths (see Figure 5.17a). While the imbalance time at the barrier (see Figure 5.2) stays constant at zero for executions with one, four, and eight ranks, it linearly increases for 16 and 32 ranks. Thereafter, the imbalance time remains constant (see Figure 5.2b). As shown in Figure 5.2a, the imbalance time increases for greater cube lengths, and executions with 32 and 64 ranks have almost identical imbalance times.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.17.: Imbalance Time for `mpi-load-balance:void ell_mvp(...)`

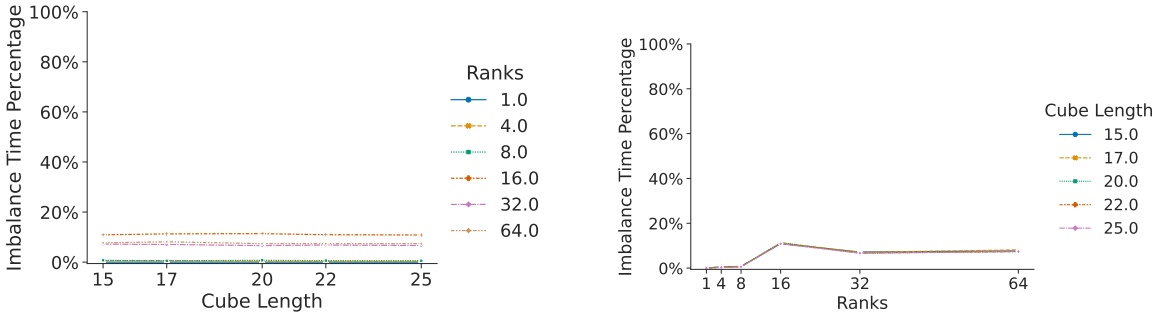


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.18.: Imbalance Time for `mpi-load-balance:MPI_Barrier`

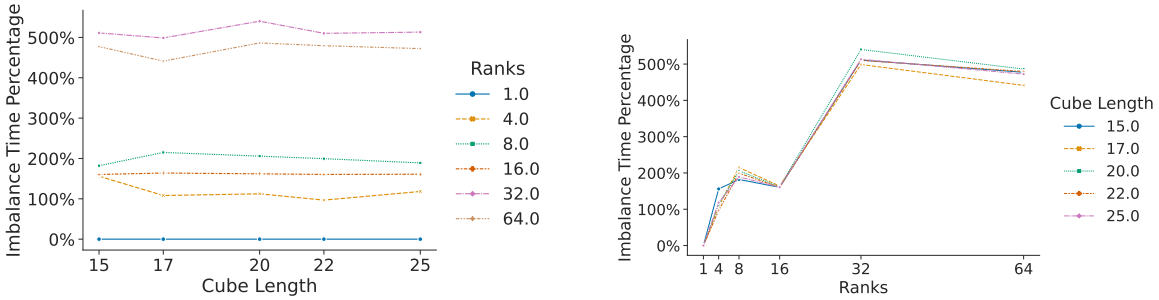
Imbalance Time Percentage The plots of the imbalance time percentage are reminiscent of the imbalance percentage. While the imbalance time percentages of `ell_mvp` (see Figure 5.19) hold the same values as the imbalance percentage (see Figure 5.15), the imbalance time percentage at the barrier reaches 500 % for executions with 32 ranks. In order to derive the imbalance time percentage, we divide the imbalance time by the average time spent in the function. In the case of the barrier, the average function time is smaller than the imbalance time due to the fact that executions with different numbers of ranks last different lengths of time. The serial executions, for example, do not need to wait at the barrier. For these reasons, imbalance time percentages higher than 100 % are possible and reveal that execution times strongly differ between executions.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.19.: Imbalance Time Percentage for `mpi-load-balance:void ell_mvp(...)`

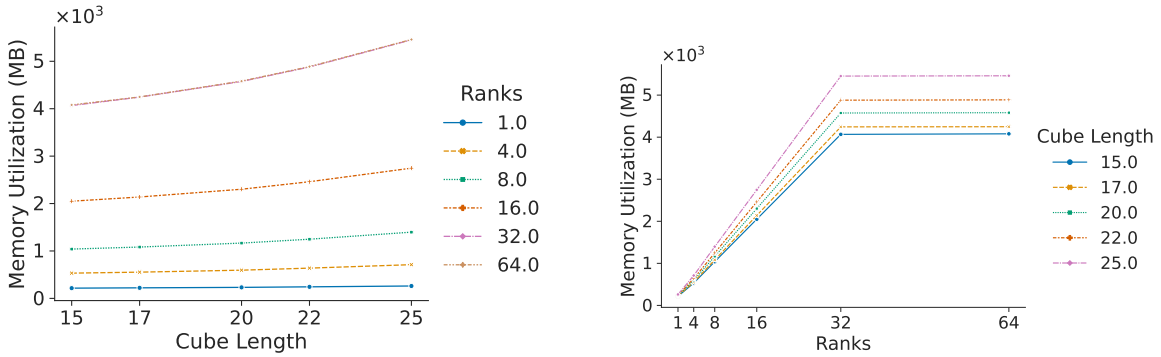


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.20.: Imbalance Time Percentage for `mpi-load-balance:MPI_Barrier`

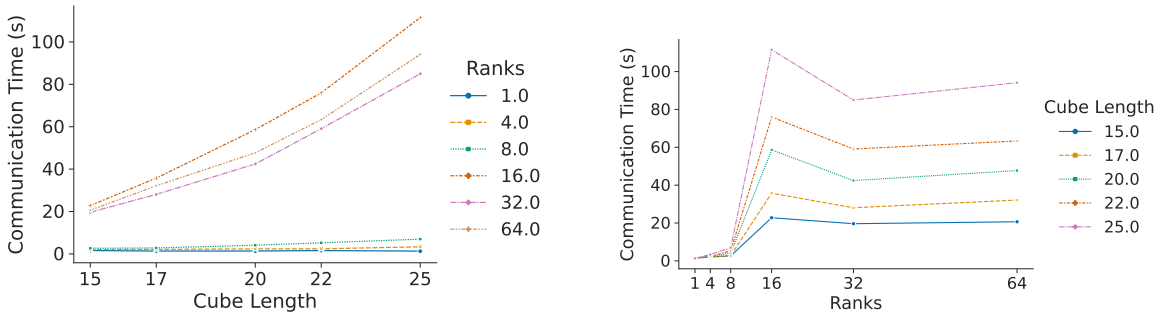
Memory Utilization As shown in Figure 5.21b, the memory utilization increases linearly until 32 ranks, for a rising number of ranks. However, executions with 32 and 64 ranks show the same memory utilization. For executions with more than eight ranks, the memory utilization is in the lower gigabyte range. With increasing cube length (see Figure 5.21a), memory utilization is slowly rising.



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.21.: Memory Utilization for `mpi-load-balance`

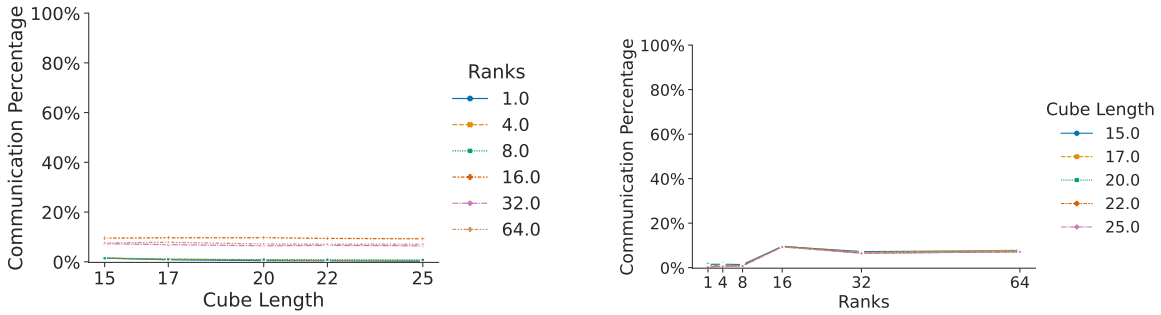
Communication Time Although the communication time is derived with regard to the whole benchmark, Figure 5.22b resembles the imbalance time shown in Figure 5.17b. This can be traced back to the high time percentage of `ell_mvp` which is over 78 %.



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.22.: Communication Time for `mpi-load-balance`

Communication Time Percentage When relating the communication time of the `mpi-load-balance` benchmark to the entire execution time, the communication time only makes up a small portion of the benchmark.



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.23.: Communication Time Percentage for `mpi-load-balance`

Evaluation of `mpi-load-balance` The two analyzed functions show a great difference in load imbalance. Since the time spent in `ell_mvp` almost makes up the entire time of this benchmark, the efficiency of this function determines the overall efficiency of this benchmark. Due to the fact that executions with more ranks have a higher execution time, the imbalance percentage between 5 % and 10 % of this function for executions with 16 or more ranks is not insignificant. Furthermore, the imbalance time ranges from 20 s to 100 s. Especially, executions with a higher number of ranks and/or greater cube lengths could profit from optimizing this function. Noticeably, the metric values of executions with 32 and 64 ranks are similar or identical for most metrics.

The following tables depict the detailed performance data for the overall benchmark with varying cube lengths:

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	108.00	3.63×10^{11}	215.78	2.40×10^{10}	2.40×10^{10}	0.00	1.60	1.48 %
4	131.92	1.47×10^{12}	532.27	2.10×10^{10}	5.24×10^{9}	2.14	2.03	1.54 %
8	188.72	2.96×10^{12}	1.04×10^{3}	1.81×10^{10}	2.27×10^{9}	3.97	2.65	1.40 %
16	239.17	7.44×10^{12}	2.05×10^{3}	1.48×10^{10}	9.23×10^{8}	57.70	22.79	9.53 %
32	271.26	1.44×10^{13}	4.07×10^{3}	1.38×10^{10}	4.30×10^{8}	114.13	19.63	7.24 %
64	272.40	2.98×10^{13}	4.08×10^{3}	1.40×10^{10}	2.18×10^{8}	113.91	20.66	7.58 %

Table 5.3.: `mpi-load-balance`: Total Values for Cube Length 15

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	164.17	5.30×10^{11}	222.51	2.29×10^{10}	2.29×10^{10}	0.00	1.34	0.82 %
4	194.79	2.14×10^{12}	553.30	2.07×10^{10}	5.19×10^{9}	2.55	2.14	1.10 %
8	283.85	4.31×10^{12}	1.08×10^{3}	1.77×10^{10}	2.22×10^{9}	4.96	2.82	0.99 %
16	371.14	1.09×10^{16}	2.14×10^{3}	1.44×10^{10}	8.98×10^{8}	93.60	35.76	9.63 %
32	407.39	2.06×10^{16}	4.24×10^{3}	1.36×10^{10}	4.24×10^{8}	163.42	28.06	6.89 %
64	408.20	4.32×10^{13}	4.25×10^{3}	1.37×10^{10}	2.14×10^{8}	168.99	32.12	7.87 %

Table 5.4.: `mpi-load-balance`: Total Values for Cube Length 17

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	285.60	8.66×10^{11}	232.06	2.22×10^{10}	2.22×10^{10}	0.00	1.35	0.47 %
4	320.75	3.49×10^{12}	594.91	2.04×10^{10}	5.10×10^{9}	2.77	2.36	0.74 %
8	462.21	7.06×10^{12}	1.17×10^{3}	1.76×10^{10}	2.20×10^{9}	10.93	4.09	0.88 %
16	605.00	1.78×10^{13}	2.30×10^{3}	1.39×10^{10}	8.70×10^{8}	153.28	58.64	9.69 %
32	660.50	3.36×10^{16}	4.57×10^{3}	1.34×10^{10}	4.19×10^{8}	267.33	42.43	6.42 %
64	667.21	7.07×10^{13}	4.58×10^{3}	1.35×10^{10}	2.11×10^{8}	276.26	47.69	7.15 %

Table 5.5.: `mpi-load-balance`: Total Values for Cube Length 20

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	386.00	1.16×10^{12}	243.65	2.23×10^{10}	2.23×10^{10}	0.00	1.58	0.41 %
4	434.05	4.65×10^{12}	638.10	2.02×10^{10}	5.05×10^{9}	3.35	2.39	0.55 %
8	633.83	9.41×10^{12}	1.25×10^{3}	1.72×10^{10}	2.15×10^{9}	12.35	5.23	0.83 %
16	808.94	2.38×10^{13}	2.46×10^{3}	1.41×10^{10}	8.78×10^{8}	197.71	76.00	9.39 %
32	897.22	4.57×10^{13}	4.88×10^{3}	1.33×10^{10}	4.15×10^{8}	356.50	59.08	6.58 %
64	897.39	9.38×10^{13}	4.89×10^{3}	1.34×10^{10}	2.10×10^{8}	363.78	63.35	7.06 %

Table 5.6.: `mpi-load-balance`: Total Values for Cube Length 22

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	571.33	1.70×10^{12}	261.17	2.21×10^{10}	2.21×10^{10}	0.00	1.33	0.23 %
4	641.96	6.86×10^{12}	711.12	2.02×10^{10}	5.04×10^{9}	5.72	3.35	0.52 %
8	942.72	1.38×10^{13}	1.40×10^{3}	1.70×10^{10}	2.13×10^{9}	17.99	6.96	0.74 %
16	1.20×10^{3}	3.44×10^{13}	2.75×10^{3}	1.42×10^{10}	8.90×10^{8}	292.58	111.49	9.25 %
32	1.32×10^{3}	6.72×10^{13}	5.45×10^{3}	1.32×10^{10}	4.13×10^{8}	519.31	84.96	6.41 %
64	1.33×10^{3}	1.37×10^{14}	5.46×10^{3}	1.33×10^{10}	2.08×10^{8}	538.54	94.11	7.07 %

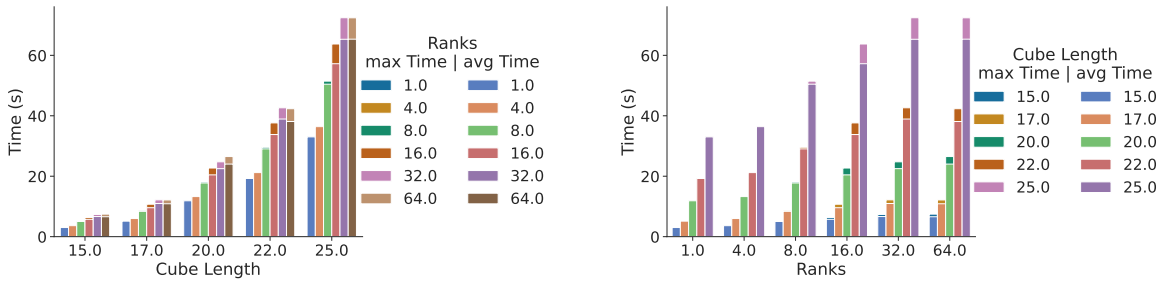
Table 5.7.: `mpi-load-balance`: Total Values for Cube Length 25

5.3.1.2. `benchmark-sc2019` Benchmark

In the following, we analyze the results of the measured metrics for the `benchmark-sc2019` benchmark. Furthermore, functions with a time percentage less than 5 % can be left out of this efficiency analysis without changing the overall result. For comparability reasons, we only show the plots of the functions also used in the efficiency analysis of the `mpi-load-balance` benchmarks. The plots of the other functions with a time percentage over 5 % (see Table 5.8) are shown in subsection A.1.1. Since the plots and their trend of this benchmark are similar to the plots of the `mpi-load-balance` benchmark (see subsubsection 5.3.1.1), we do not describe every plot in detail.

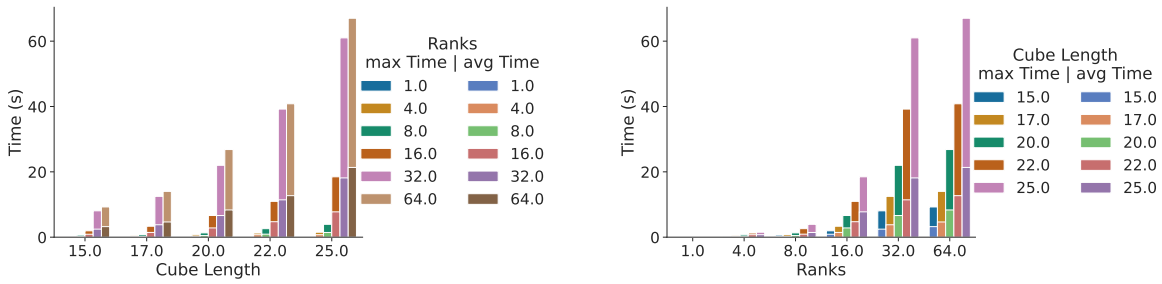
Function	Time Percentage
`void ell_mvp(const ell_matrix*, const double*, double*)`	13.64 % to 28.97 %
`virtual void get_stress(const double*, double*, const double*) const`	9.95 % to 12.46 %
`void get_elem_mat(const double*, const double*, double*, int, int, int) const`	8.23 % to 10.23 %
`bool damage_law(const double*, double, double, double*, double*, double*) const`	8.03 % to 9.80 %
`MPI_Barrier`	0.000 08 % to 8.85 %
`MPI_Init`	0.33 % to 7.50 %
`void apply_perturbation(const double*, double*, const double*) const`	5.69 % to 7.16 %
other functions	<5 %

Table 5.8.: Time Percentage of `benchmark-sc2019` functions



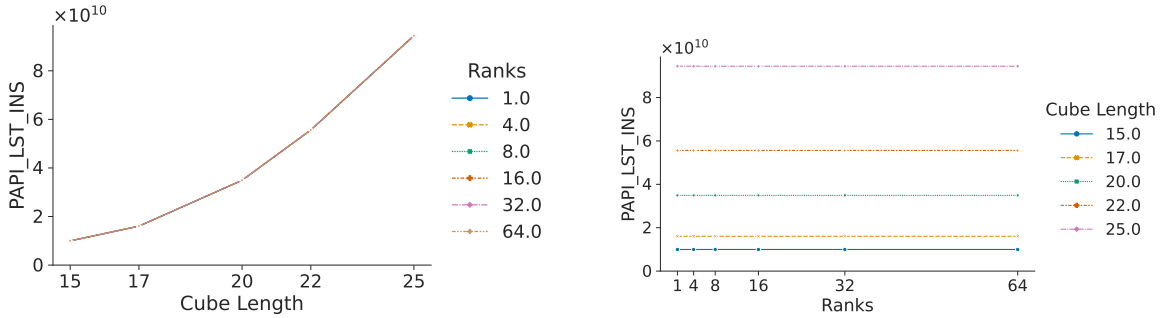
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.24.: Time for `benchmark-sc2019:void ell_mvp(...)`



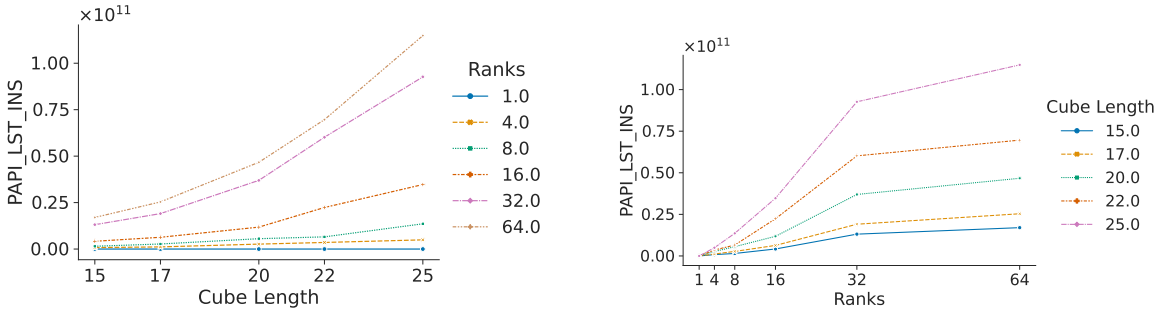
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.25.: Time for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size **(b)** Variable number of MPI ranks

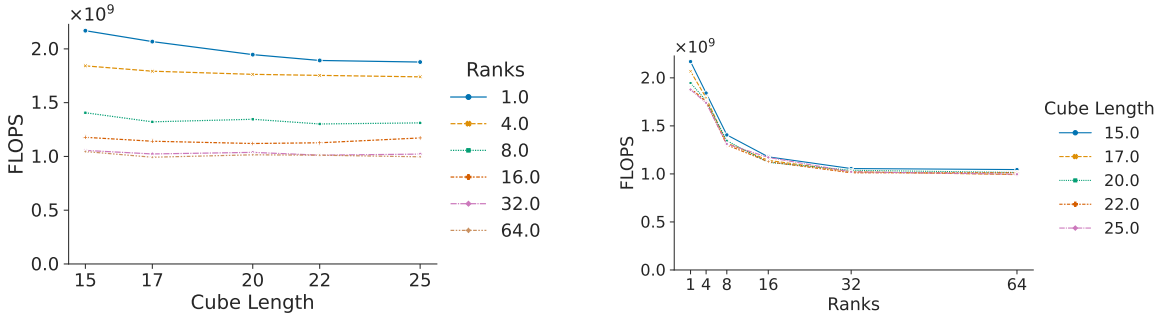
Figure 5.26.: Load/Store Instructions for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

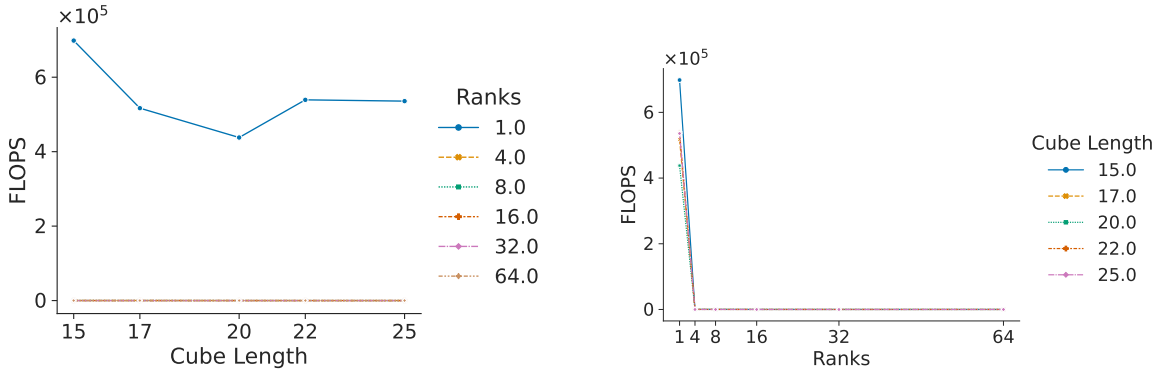
Figure 5.27.: Load/Store Instructions for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

(b) Variable number of MPI ranks

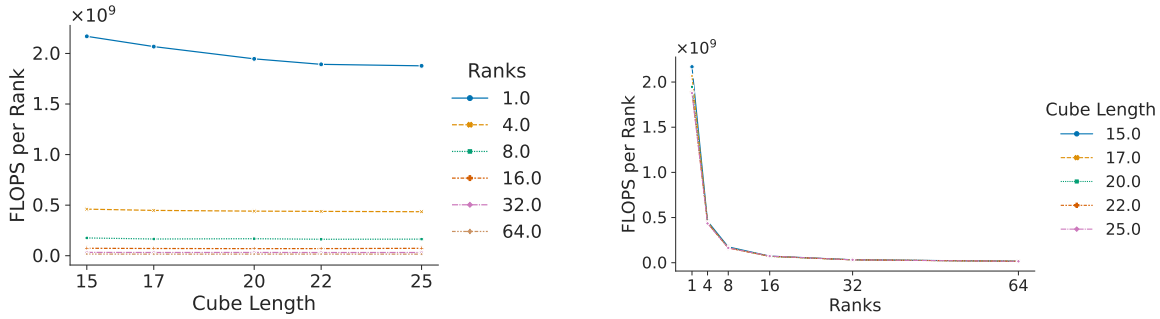
Figure 5.28.: FLOPS for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

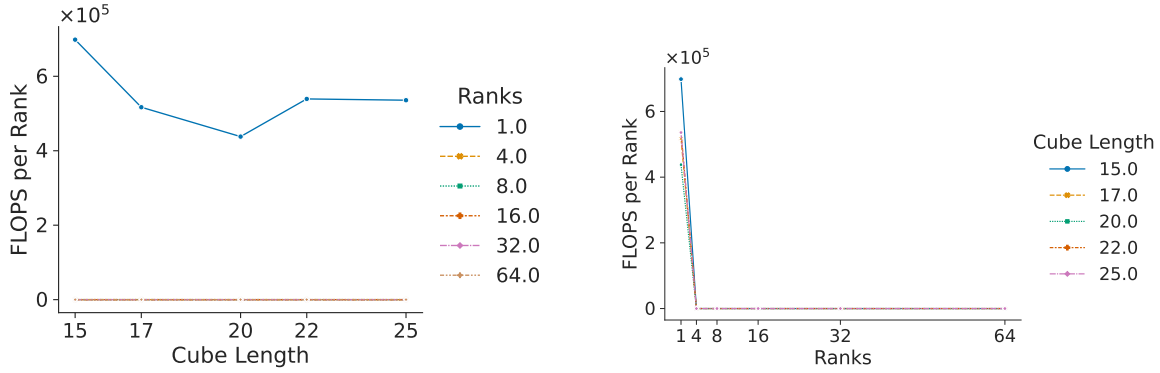
(b) Variable number of MPI ranks

Figure 5.29.: FLOPS for `benchmark-sc2019:MPI_Barrier`



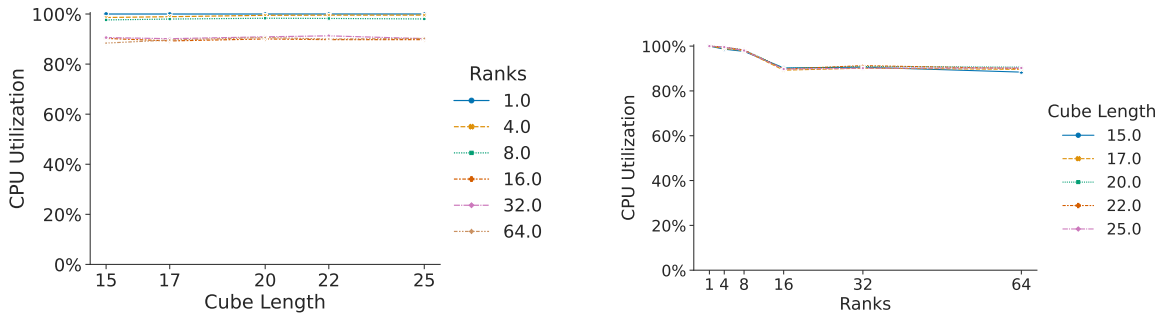
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.30.: FLOPS per Rank for `benchmark-sc2019:void ell_mvp(...)`



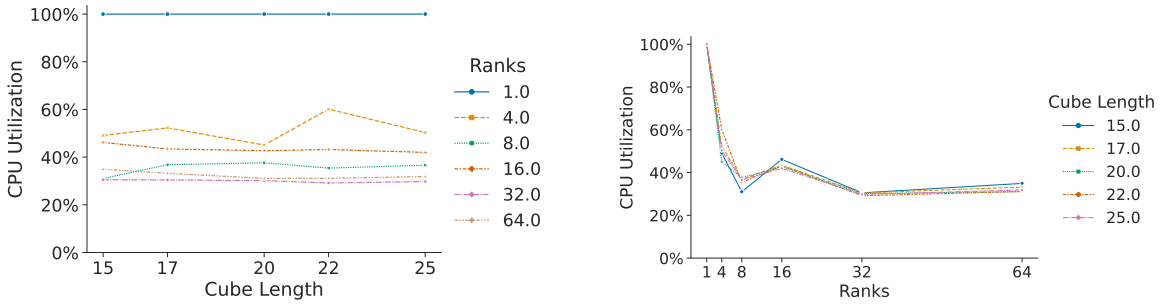
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.31.: FLOPS per Rank for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size **(b)** Variable number of MPI ranks

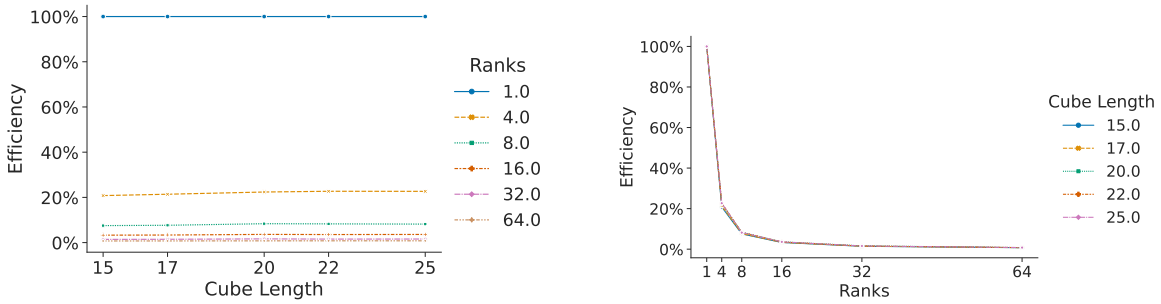
Figure 5.32.: CPU Utilization for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

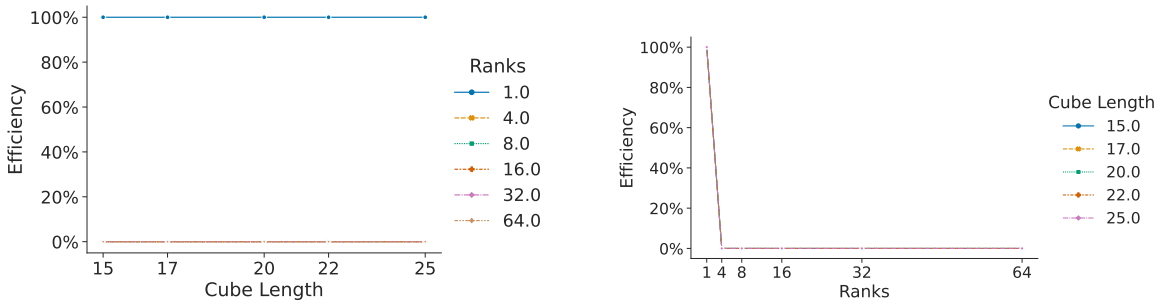
Figure 5.33.: CPU Utilization for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

(b) Variable number of MPI ranks

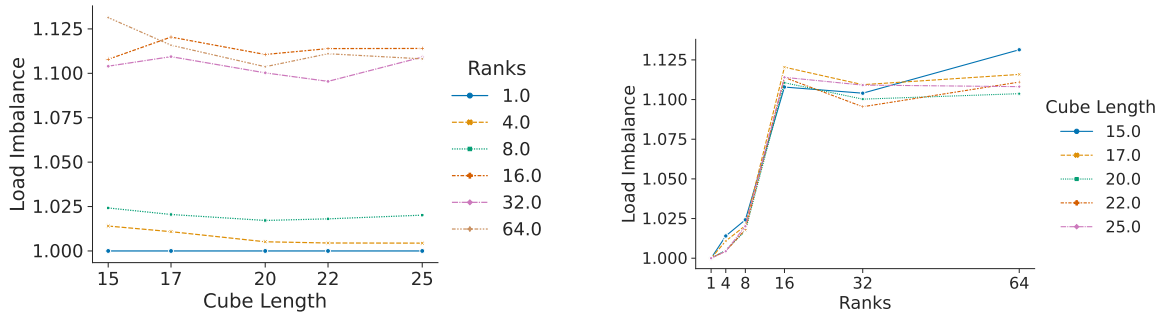
Figure 5.34.: Efficiency for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

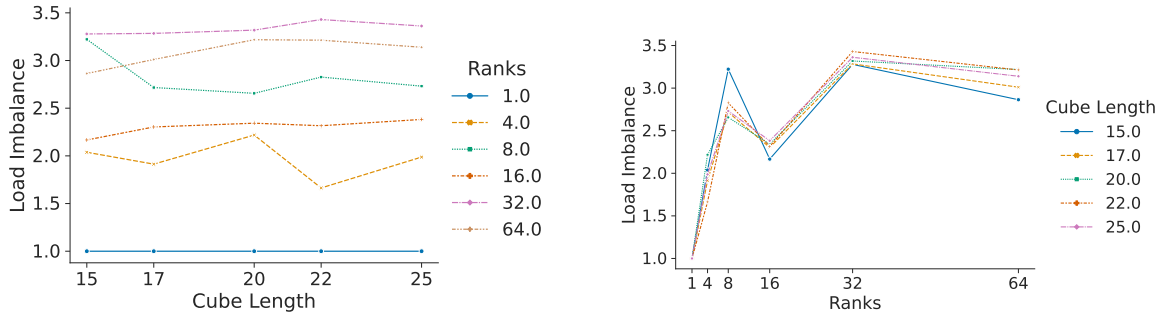
Figure 5.35.: Efficiency for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

(b) Variable number of MPI ranks

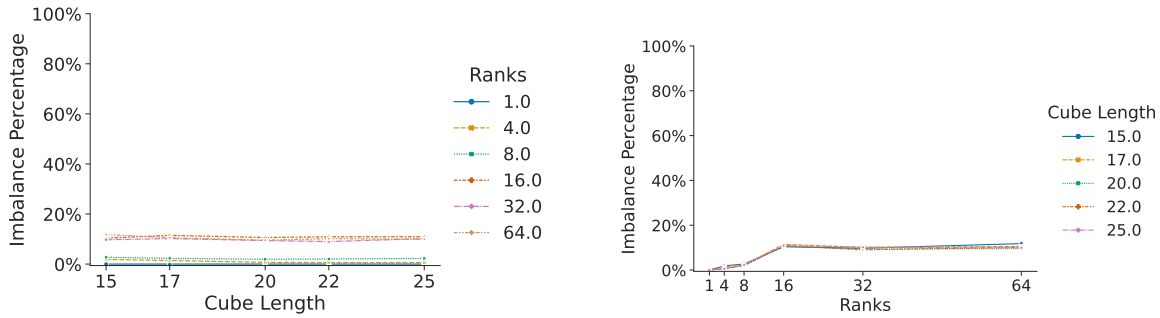
Figure 5.36.: Load Imbalance for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

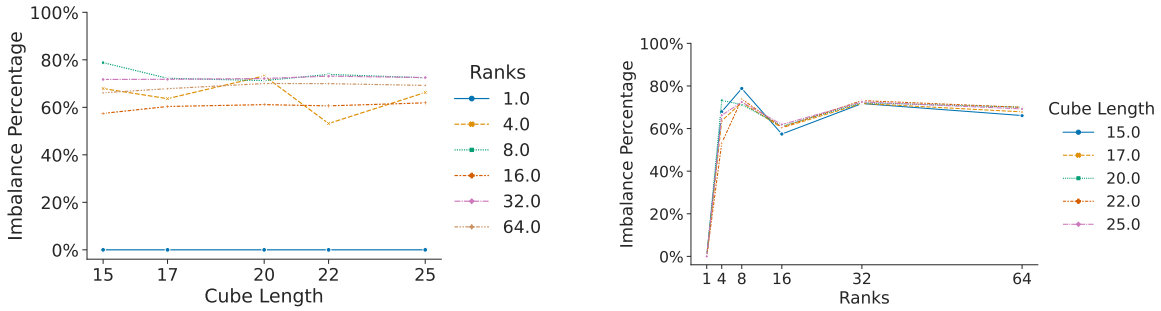
Figure 5.37.: Load Imbalance for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

(b) Variable number of MPI ranks

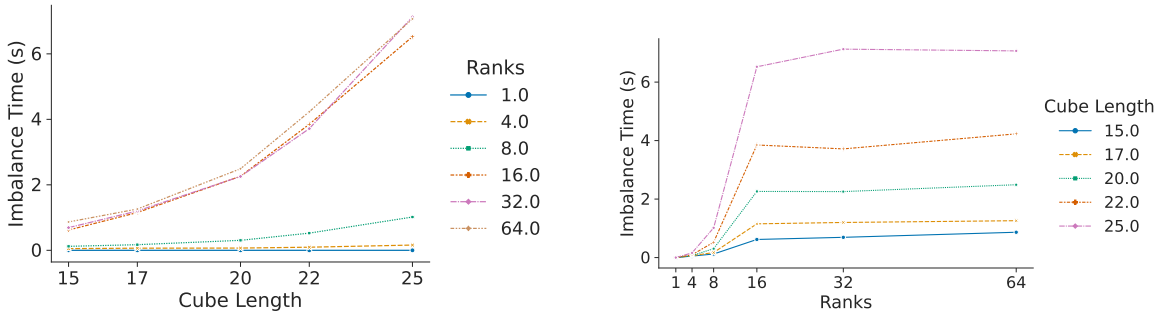
Figure 5.38.: Imbalance Percentage for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

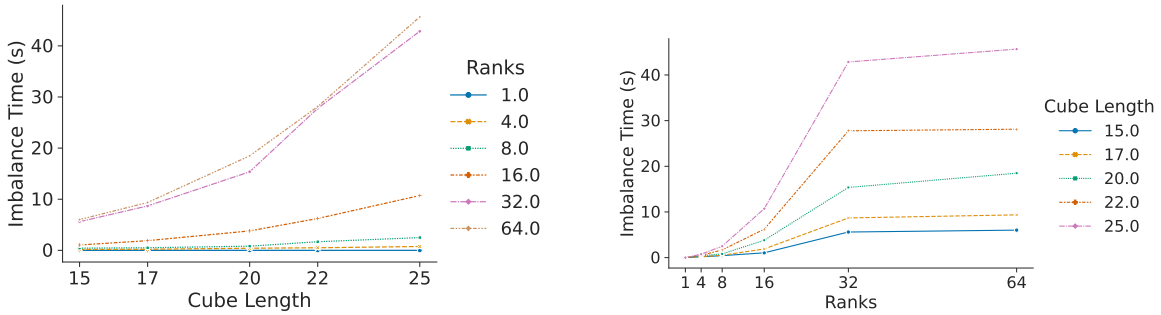
Figure 5.39.: Imbalance Percentage for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

(b) Variable number of MPI ranks

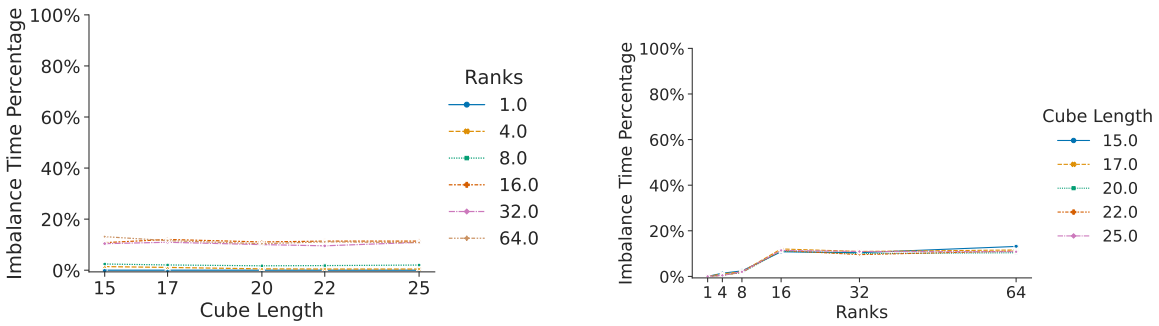
Figure 5.40.: Imbalance Time for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

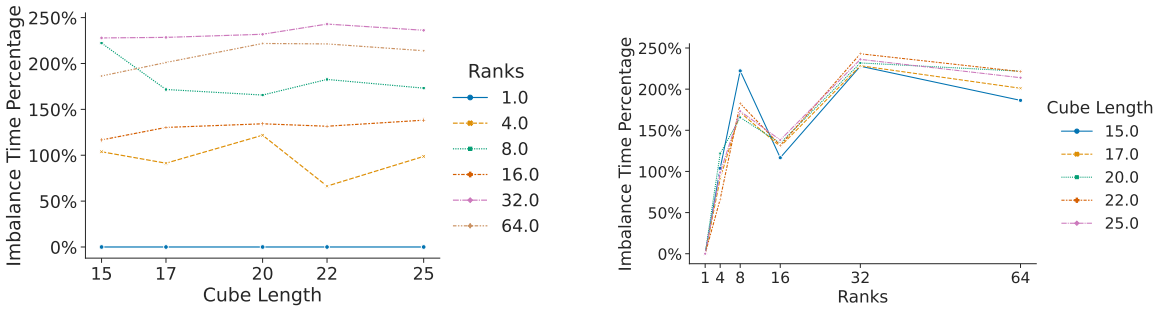
Figure 5.41.: Imbalance Time for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

(b) Variable number of MPI ranks

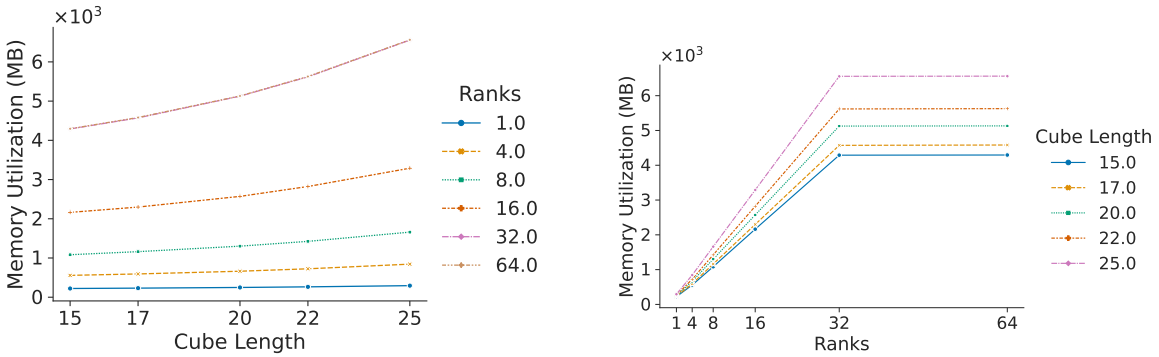
Figure 5.42.: Imbalance Time Percentage for `benchmark-sc2019:void ell_mvp(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

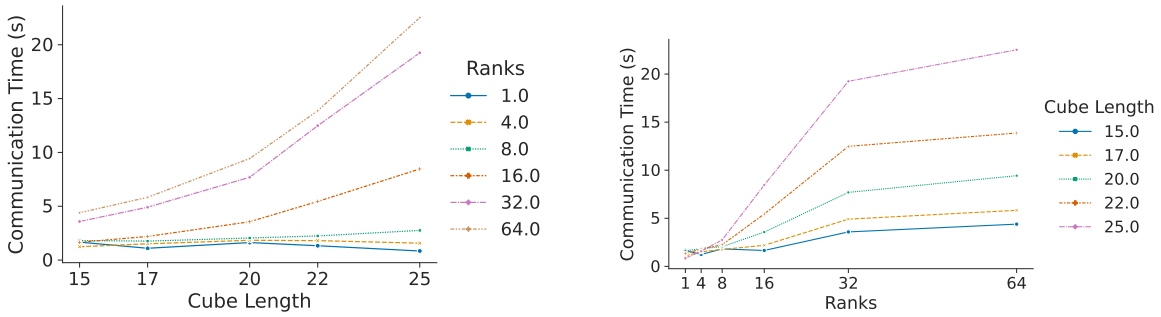
Figure 5.43.: Imbalance Time Percentage for `benchmark-sc2019:MPI_Barrier`



(a) Variable problem size

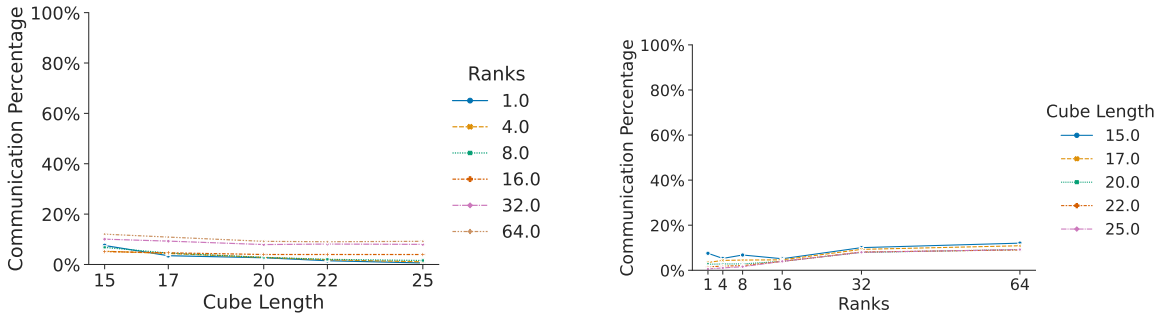
(b) Variable number of MPI ranks

Figure 5.44.: Memory Utilization for `benchmark-sc2019`



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.45.: Communication Time for `benchmark-sc2019`



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.46.: Communication Time Percentage for `benchmark-sc2019`

Evaluation of `benchmark-sc2019` Similarly to the `mpi-load-balance` benchmark, the two shown functions of this benchmark also have a great difference in load imbalance. For executions with 16 or more ranks, the imbalance percentage of `ell_mvp` is approximately 10 %. Given that executions with a higher number of ranks tend to take longer, the imbalance of this function is not insignificant. However, since the actual imbalance time is 7 s at most for all executions, the overhead of optimizing this function might outweigh the potential efficiency gain. Likewise to the `mpi-load-balance` benchmark, the metric values of executions with 32 and 64 ranks are similar or identical for most metrics.

The following tables depict the detailed performance data for the overall benchmark with varying cube lengths:

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	22.14	9.42×10^{10}	222.47	2.38×10^{10}	2.38×10^{10}	0.00	1.67	7.53 %
4	23.56	3.80×10^{11}	557.20	2.10×10^{10}	5.26×10^{9}	0.46	1.24	5.26 %
8	26.81	7.66×10^{11}	1.08×10^{3}	1.87×10^{10}	2.33×10^{9}	0.91	1.82	6.77 %
16	32.35	1.58×10^{12}	2.16×10^{3}	1.52×10^{10}	9.53×10^{8}	2.54	1.66	5.12 %
32	35.67	3.44×10^{12}	4.29×10^{3}	1.45×10^{10}	4.52×10^{8}	8.77	3.58	10.04 %
64	36.49	7.12×10^{12}	4.29×10^{3}	1.42×10^{10}	2.22×10^{8}	10.24	4.39	12.04 %

Table 5.9.: `benchmark-sc2019`: Total Values for Cube Length 15

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	31.54	1.38×10^{11}	231.53	2.26×10^{10}	2.26×10^{10}	0.00	1.10	3.48 %
4	34.56	5.57×10^{11}	593.42	2.07×10^{10}	5.19×10^{9}	0.56	1.51	4.37 %
8	38.91	1.13×10^{12}	1.16×10^{3}	1.80×10^{10}	2.25×10^{9}	1.23	1.76	4.53 %
16	47.37	2.31×10^{12}	2.30×10^{3}	1.49×10^{10}	9.33×10^{8}	4.27	2.19	4.63 %
32	52.94	5.03×10^{12}	4.57×10^{3}	1.43×10^{10}	4.46×10^{8}	13.81	4.91	9.28 %
64	53.63	1.05×10^{13}	4.58×10^{3}	1.39×10^{10}	2.18×10^{8}	15.56	5.83	10.87 %

Table 5.10.: `benchmark-sc2019`: Total Values for Cube Length 17

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	60.77	2.62×10^{11}	248.70	2.19×10^{10}	2.19×10^{10}	0.00	1.63	2.68 %
4	65.57	1.06×10^{12}	662.69	2.04×10^{10}	5.11×10^{9}	0.94	1.84	2.80 %
8	72.72	2.14×10^{12}	1.30×10^{3}	1.82×10^{10}	2.28×10^{9}	2.17	2.05	2.82 %
16	89.91	4.38×10^{12}	2.57×10^{3}	1.46×10^{10}	9.13×10^{8}	8.11	3.56	3.96 %
32	97.28	9.57×10^{12}	5.13×10^{3}	1.42×10^{10}	4.43×10^{8}	24.69	7.69	7.91 %
64	102.68	1.98×10^{13}	5.13×10^{3}	1.40×10^{10}	2.18×10^{8}	29.57	9.43	9.19 %

Table 5.11.: `benchmark-sc2019`: Total Values for Cube Length 20

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	89.42	3.85×10^{11}	264.31	2.18×10^{10}	2.18×10^{10}	0.00	1.33	1.49 %
4	96.68	1.56×10^{12}	724.50	2.04×10^{10}	5.11×10^{9}	1.62	1.81	1.87 %
8	109.87	3.14×10^{12}	1.42×10^{3}	1.78×10^{10}	2.23×10^{9}	3.71	2.24	2.04 %
16	137.43	6.52×10^{12}	2.82×10^{3}	1.47×10^{10}	9.20×10^{8}	13.84	5.45	3.96 %
32	154.07	1.43×10^{13}	5.62×10^{3}	1.39×10^{10}	4.35×10^{8}	42.31	12.48	8.10 %
64	153.65	2.91×10^{13}	5.63×10^{3}	1.37×10^{10}	2.14×10^{8}	45.28	13.87	9.03 %

Table 5.12.: `benchmark-sc2019`: Total Values for Cube Length 22

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	139.53	5.99×10^{11}	294.86	2.18×10^{10}	2.18×10^{10}	0.00	0.84	0.60 %
4	150.68	2.41×10^{12}	843.51	2.03×10^{10}	5.07×10^{9}	2.02	1.57	1.04 %
8	174.14	4.90×10^{12}	1.66×10^{3}	1.76×10^{10}	2.20×10^{9}	6.05	2.76	1.58 %
16	214.13	1.01×10^{13}	3.29×10^{3}	1.50×10^{10}	9.36×10^{8}	21.89	8.46	3.95 %
32	241.03	2.21×10^{13}	6.56×10^{3}	1.39×10^{10}	4.35×10^{8}	66.12	19.25	7.99 %
64	244.89	4.57×10^{13}	6.56×10^{3}	1.38×10^{10}	2.15×10^{8}	73.50	22.52	9.20 %

Table 5.13.: `benchmark-sc2019`: Total Values for Cube Length 25

5.3.1.3. Discussion

Noticeably, executions of the `mpi-load-balance` benchmark spend relatively more time in the compute-intensive function `ell_mvp`. Hence, the efficiency of these functions affects the efficiency of `mpi-load-balance` more. On the other hand, `benchmark-sc2019` has more functions that are relevant in regard to the overall efficiency of this function. Although the compute-intensive function `ell_mvp` does not show a high imbalance time, the overall efficiency of this benchmark might be lower since we need to incorporate the imbalance times of the other functions. However, optimizing `mpi-load-balance` might promise better results regarding efficiency due to the fact that only one function needs to be optimized. Since the overhead of efficiency optimization limits the efficiency gain, optimizing an application, especially with software resource disaggregation, profits from offloading fewer functions.

Furthermore, the load imbalance at the barrier is only half as high for the `benchmark-sc2019` benchmark. However, `mpi-load-balance` takes approximately 10 times longer. Overall, both benchmarks might not be comparable since they have different purposes. In line with this thesis, we do not examine the functionality of the benchmarks in detail, which makes it impossible to compare these benchmarks.

5.3.2. eSam(oa)^2

Although supplementary data is specified in related work [32], we could not generate any valid input files. With the provided permeability and porosity input files for the `SPE10` benchmark [46], we could not create the two separate NetCDF files requested by the benchmark with the provided script. Regarding the `tohoku` benchmark, related work [32] mentions an old version of a bathymetric input file from the General Bathymetric Chart of the Oceans (GEBCO) [36], which is no longer available. Unfortunately, GEBCO does not offer an equivalent, newer version of this input file.

5.3.3. LAMMPS

For analyzing the efficiency of the HPC application LAMMPS, we choose the `Lennard-Jones` benchmark since it has often been used in related work [48, 39, 34]. Furthermore, LAMMPS offers load-balancing benchmarks with several load-balancing algorithms. We choose the LJ benchmark with the recursive-coordinate bisectioning (RCB) load-balancing algorithm as the second benchmark, with the motive to compare both benchmarks. This second benchmark is also used in related work [48], which we can compare our results with. Since these benchmarks differ in a few input parameters, we keep the input parameters of the RCB benchmark as default and customize the input parameters of the LJ benchmark, respectively. Each benchmark consists of two runs with 250 time steps each. For scaling the problem size, we vary the number of atoms per core. For comparability reasons, we adopt the values for the number of atoms from Thompson et al. [48]. The benchmarks are executed with three different problem sizes, namely 32×10^3, 256×10^3, and 2048×10^3 atoms per core.

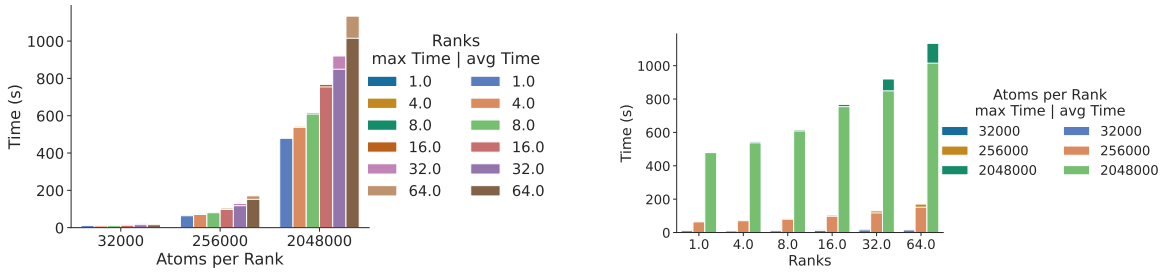
5.3.3.1. Lennard-Jones (LJ) Benchmark without Load Balancing

In the following, we analyze the results of the performance measurement of the `LJ` benchmark. The functions with a time percentage greater than 5 % are depicted in Table 5.14. Since the plots for `compute`, `build`, and `ev_tally`, as well as `MPI_Wait` and `MPI_Send`, are similar, we only examine the plots of `compute` and `MPI_Wait` in this thesis. The other plots can be found in the appendix (see subsection A.2.1). Since `MPI_Wait` does not get called for serial executions, the plots only display the functions for parallel executions.

Function	Time Percentage
`virtual void compute(int, int)`	28.19 % to 51.18 %
`void build(NeighList*)`	25.70 % to 39.24 %
`void ev_tally(int, int, int, int, double, double, double, double, double, double)`	2.54 % to 6.59 %
`MPI_Wait`	0.28 % to 8.85 %
`MPI_Send`	0.21 % to 6.52 %
`MPI_Init`	0.04 % to 5.93 %
other functions	<5 %

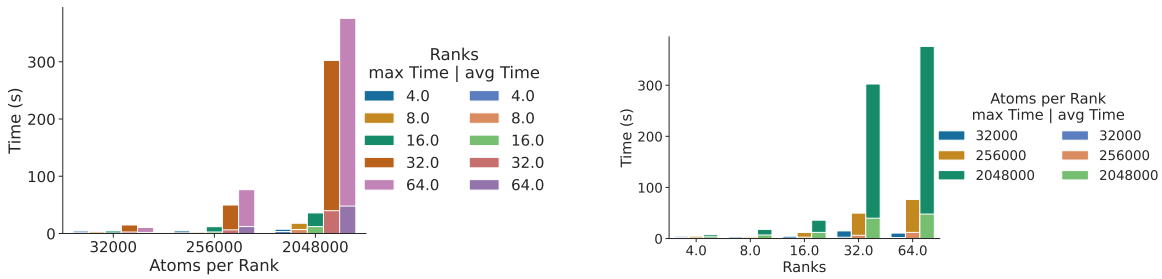
Table 5.14.: Time Percentage of `LJ` functions

Time Figure 5.47 shows that the average time of `compute` is nearly identical to its maximum time for all executions. This reveals that this function is well-balanced. For `MPI_Wait`, however (see Figure 5.48), the difference between average and maximum time is relatively large. This shows that there are high idle times in this function. Noticeably, executions with eight ranks or fewer barely spend any time at `MPI_Wait`.



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.47.: Time for `LJ:void compute(...)`

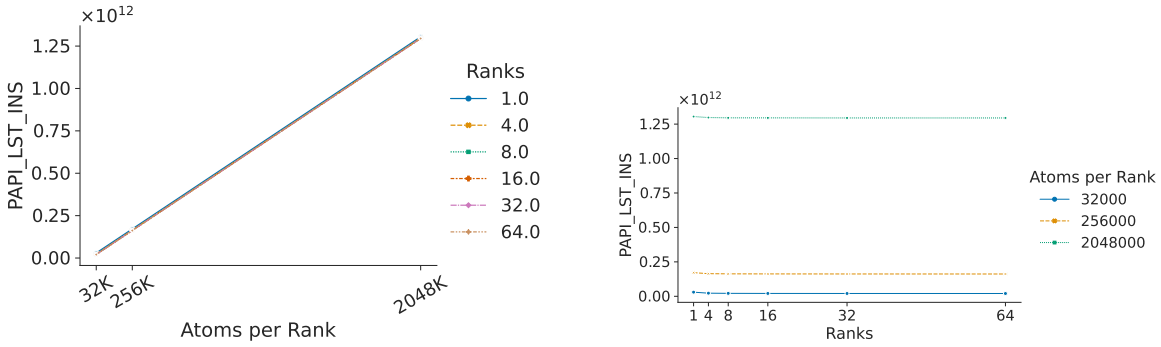


(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.48.: Time for `LJ:MPI_Wait`

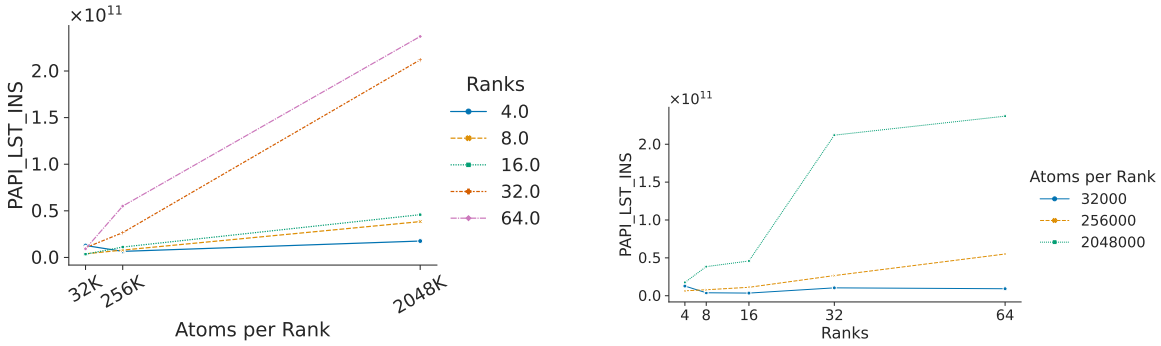
Load/Store Instructions As shown in Figure 5.49a, the load and store instructions increase linearly with a rising number of atoms per rank and are constant for executions with different

numbers of ranks (see Figure 5.49b). Executions with 2 048 000 ranks reach 1.25×10^{12} load and store instructions. At `MPI_Wait`, the load and store instructions rise linearly for different numbers of atoms per rank (see Figure 5.50a). As depicted in Figure 5.50b, the load and store instructions reach their maximum value for an execution with 64 ranks and 2 048 000 atoms per rank.



(a) Variable problem size **(b)** Variable number of MPI ranks

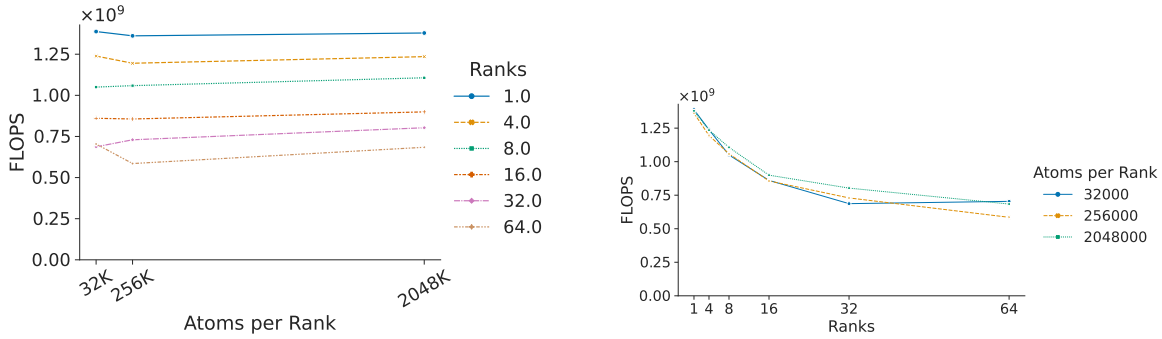
Figure 5.49.: Load/Store Instructions for `LJ:void compute(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

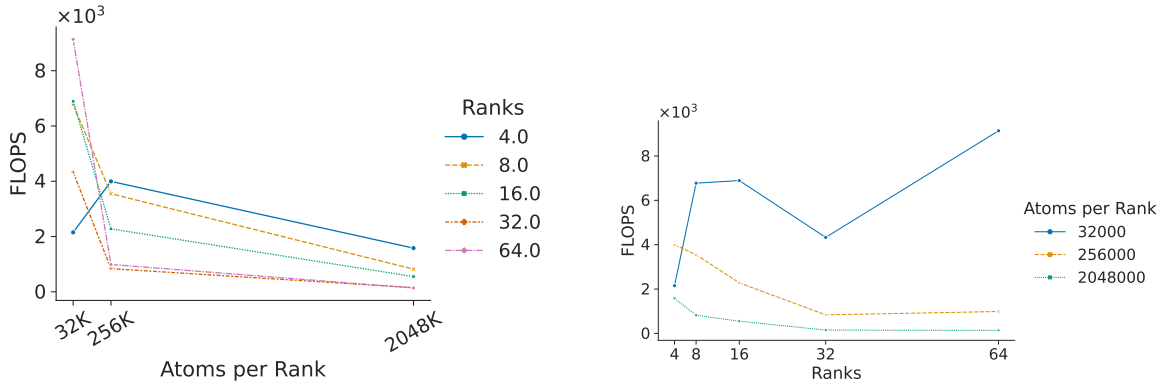
Figure 5.50.: Load/Store Instructions for `LJ:MPI_Wait`

FLOPS While the number of FLOPS nearly remains constant for different numbers of atoms per rank for the `compute` function (see Figure 5.51a), they exponentially decrease for rising numbers of ranks, as depicted in Figure 5.51b. The FLOPS reach their maximum value of approximately 1.3×10^9 FLOPS and converge to roughly 0.75×10^9 FLOPS. At `MPI_Wait`, shown in Figure 5.52, the FLOPS are in the range of 10^3. The longer the ranks have to wait at `MPI_Wait`, the smaller the value of FLOPS gets.



(a) Variable problem size **(b)** Variable number of MPI ranks

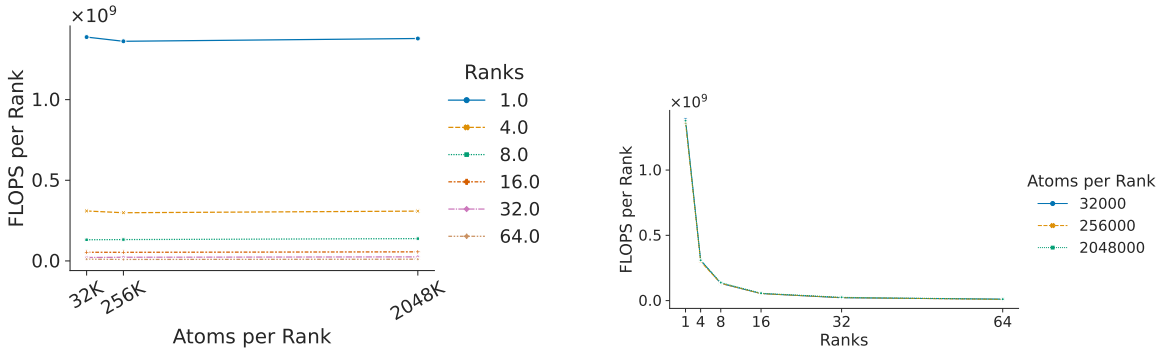
Figure 5.51.: FLOPS for `LJ:void compute(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

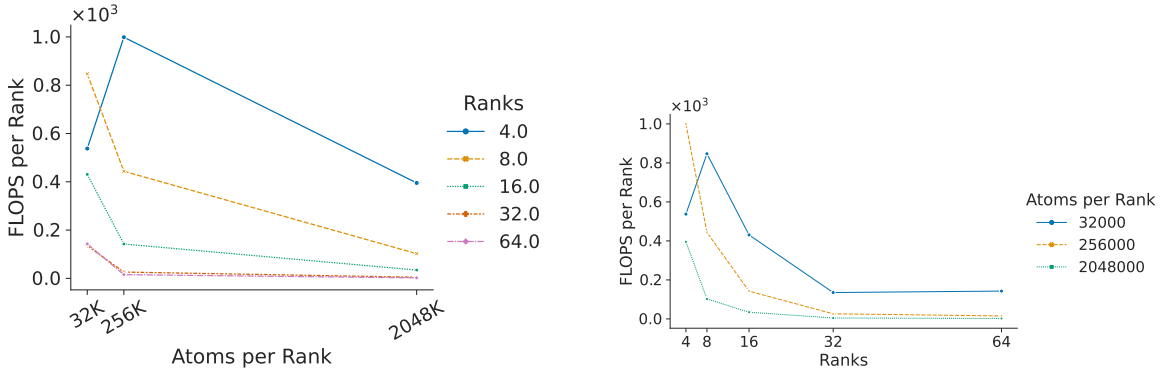
Figure 5.52.: FLOPS for `LJ:MPI_Wait`

FLOPS per Rank FLOPS per rank are more comparable for executions with different numbers of ranks. As depicted in Figure 5.53b, the FLOPS per rank drop almost immediately to zero for parallel instructions. While the FLOPS per rank exponentially decrease at `MPI_Wait` for executions with 32×10^3 and 256×10^3 ranks, the FLOPS per rank for the executions with 2048×10^3 atoms per rank peak for an execution with eight ranks.



(a) Variable problem size **(b)** Variable number of MPI ranks

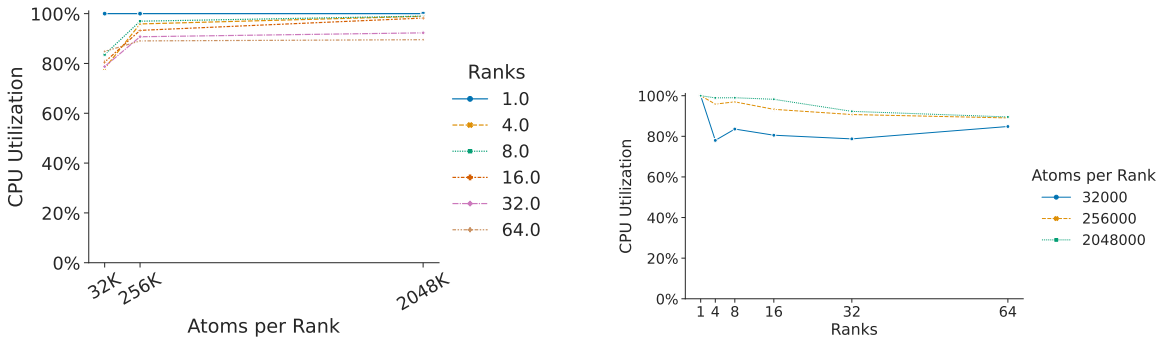
Figure 5.53.: FLOPS per Rank for `LJ:void compute(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

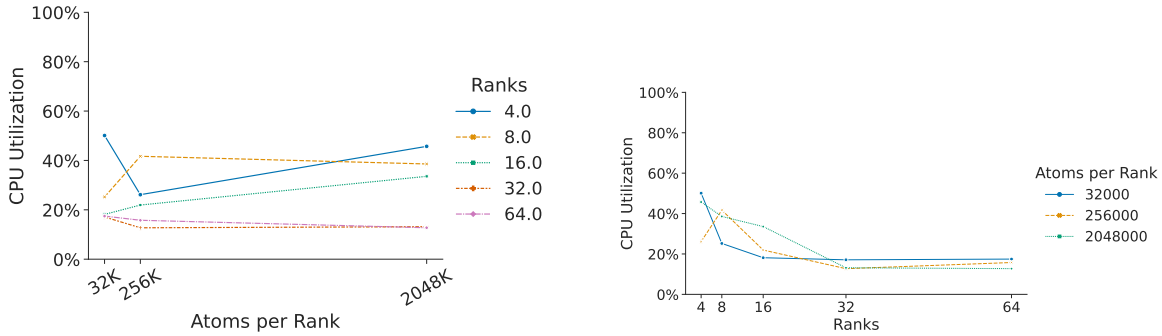
Figure 5.54.: FLOPS per Rank for `LJ:MPI_Wait`

CPU Utilization While `compute` almost perfectly uses its CPU resources. The CPU utilization for `MPI_Wait`, however, ranges from approximately 15 % to 50 %, as depicted in Figure 5.56b.



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.55.: CPU Utilization for `LJ:void compute(...)`

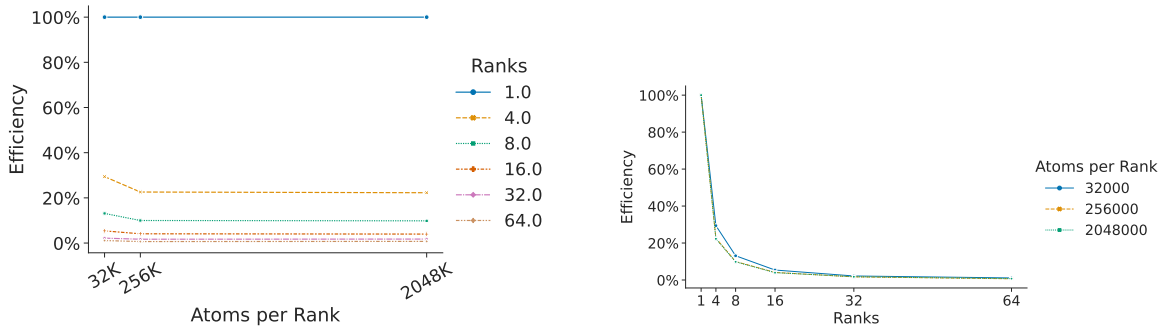


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.56.: CPU Utilization for `LJ:MPI_Wait`

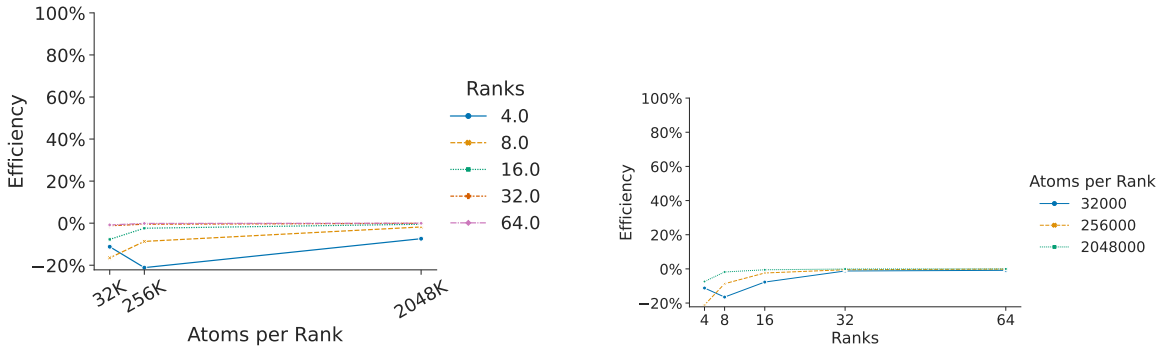
Weak Scaling Efficiency Figure 5.57 and Figure 5.58 show that the weak scaling efficiency is nearly 0 % for parallel executions. While the weak scaling efficiency stays constant for different problem sizes, we lose nearly 80 % of efficiency by scaling the number of ranks from one to four. Evidently, this curve converges to zero. The weak scaling efficiency is the fraction of serial time to parallel time. In the ideal case, the weak scaling efficiency would remain constant at 100 %. However, `MPI_Wait` is not executed for serial executions. Therefore, we cannot derive the weak scaling efficiency for this particular function.



(a) Variable problem size

(b) Variable number of MPI ranks

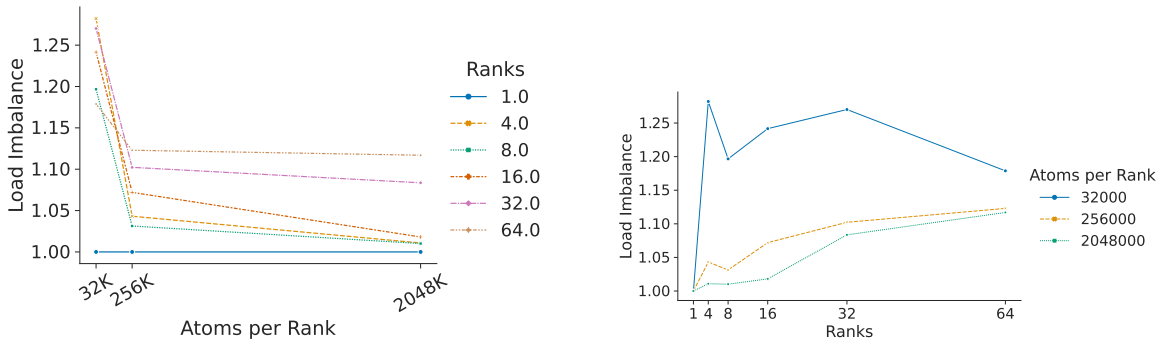
Figure 5.57.: Efficiency for `LJ:void compute(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

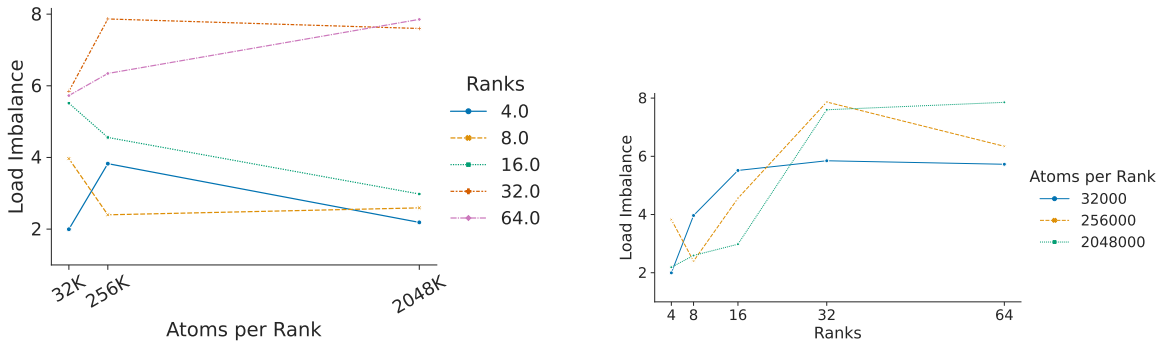
Figure 5.58.: Efficiency for `LJ:MPI_Wait`

Load Imbalance The load imbalance of `compute` peaks at nearly 1.30 (see Figure 5.59). Hence, this function is nearly perfectly balanced. At `MPI_Wait`, however, the load imbalance goes above 6 for the executions with 32 and 64 ranks.



(a) Variable problem size **(b)** Variable number of MPI ranks

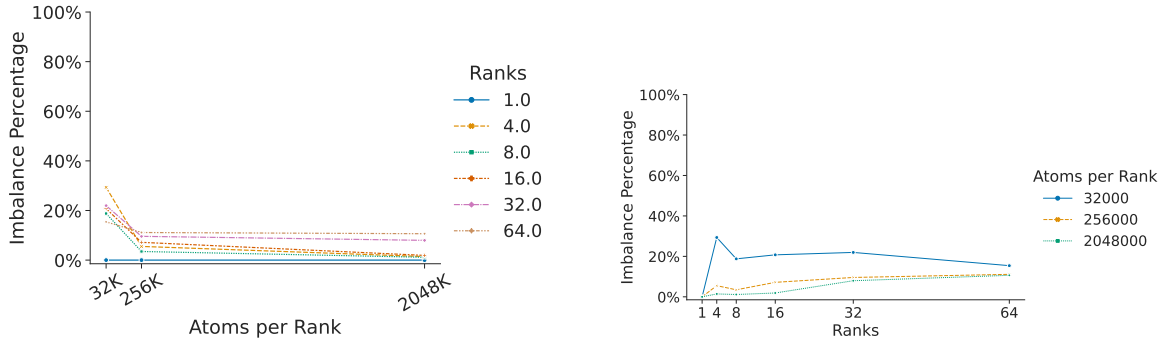
Figure 5.59.: Load Imbalance for `LJ:void compute(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

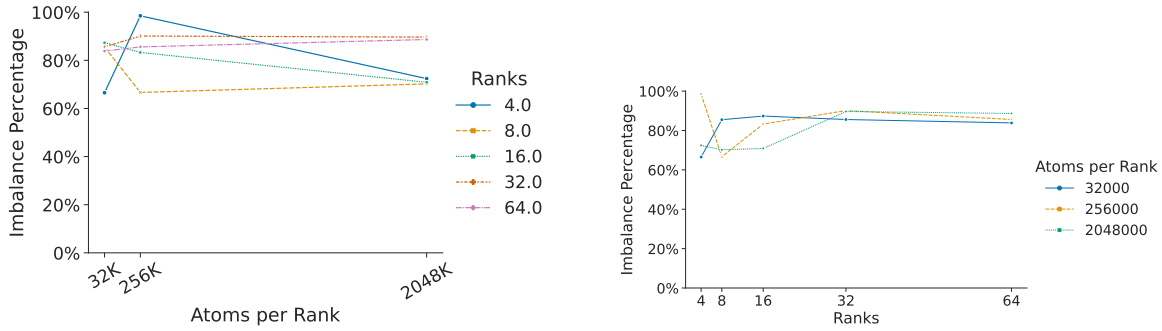
Figure 5.60.: Load Imbalance for `LJ:MPI_Wait`

Imbalance Percentage The observation of the load imbalance metrics can be transferred to the imbalance percentage. However, the imbalance percentage relates the load imbalance to the number of ranks. While the imbalance percentage of `compute` stays below 30 % for all executions (see Figure 5.61), the imbalance percentage is over 60 % at `MPI_Wait` for all parallel executions.



(a) Variable problem size **(b)** Variable number of MPI ranks

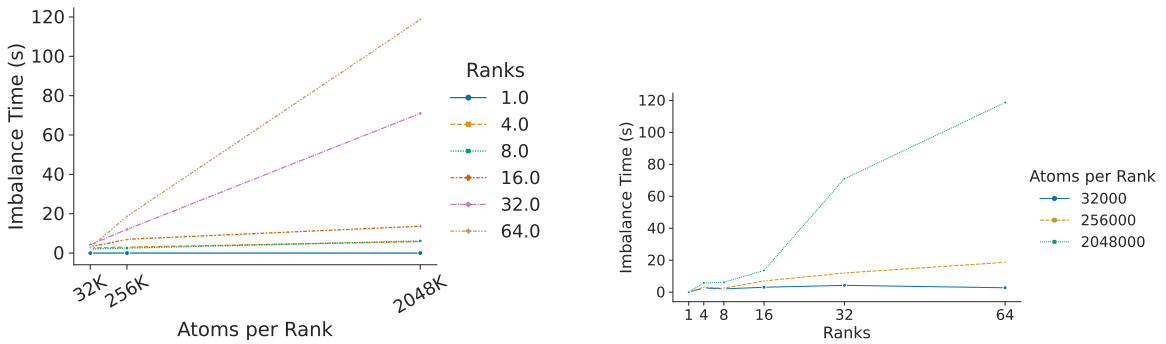
Figure 5.61.: Imbalance Percentage for `LJ:void compute(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.62.: Imbalance Percentage for `LJ:MPI_Wait`

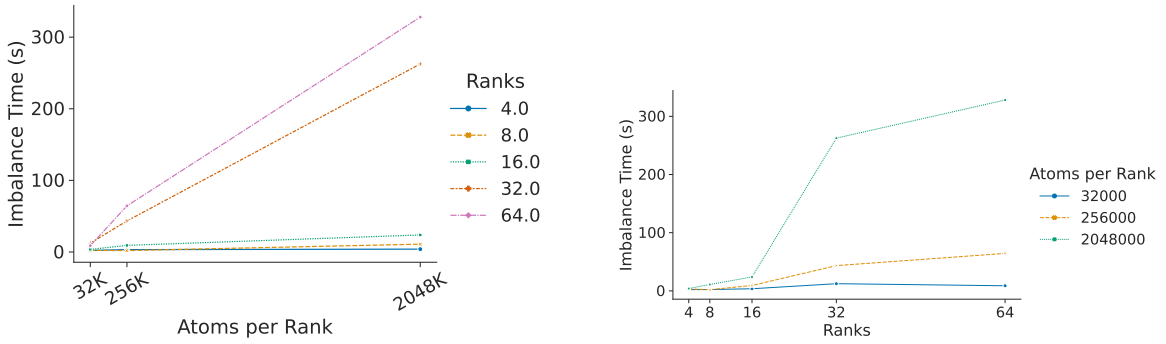
Imbalance Time For both functions (see Figure 5.63 and Figure 5.64), the imbalance time of `compute` is nearly zero for executions using eight or fewer ranks and then rises with an increasing number of ranks (see Figure 5.63). As shown in Figure 5.63a and Figure 5.48a, the imbalance time increases for greater numbers of atoms per rank, and executions with 32 and 64 ranks have almost identical imbalance times.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.63.: Imbalance Time for `LJ:void compute(...)`

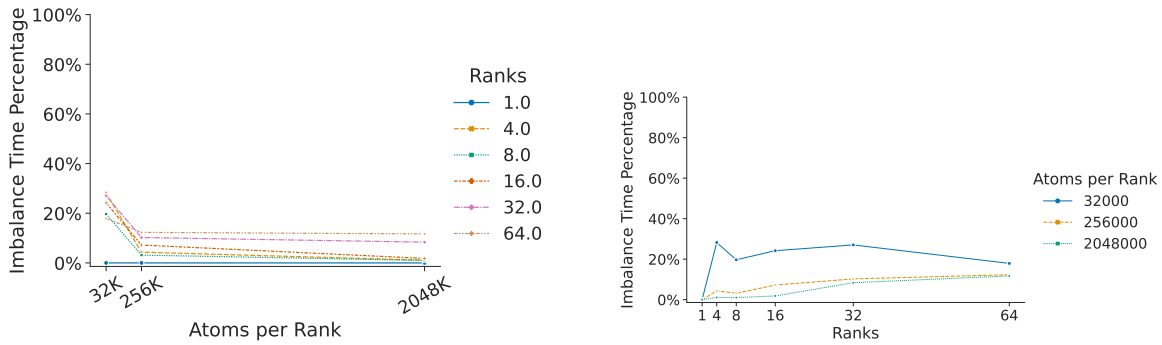


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.64.: Imbalance Time for `LJ:MPI_Wait`

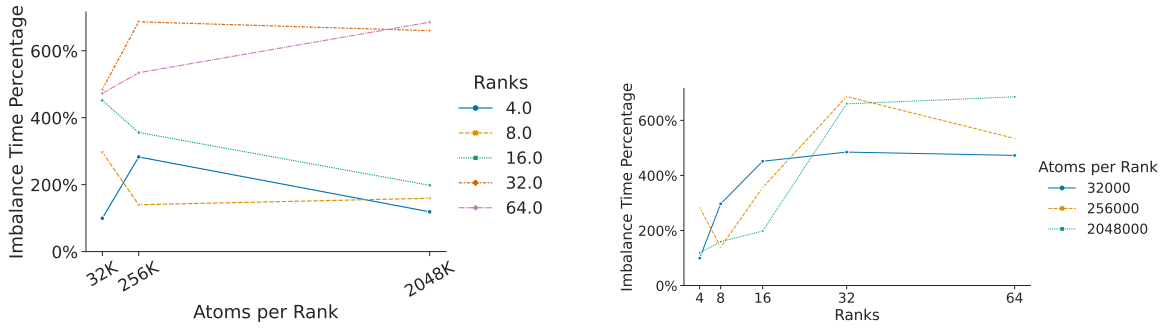
Imbalance Time Percentage The plots of the imbalance time percentage are reminiscent of the imbalance percentage. While the imbalance time percentages of `compute` (see Figure 5.65) hold the same values as the imbalance percentage (see Figure 5.61), the imbalance time percentage at `MPI_Wait` goes above 400 % for executions with 32 ranks. For executions with 256×10^3 and 2048×10^3 ranks, the imbalance time percentage reaches nearly 400 %. In order to derive the imbalance time percentage, we divide the imbalance time by the average time spent in the function. In the case of `MPI_Wait`, the average function time is smaller than the imbalance time due to the fact that executions with different numbers of ranks last different lengths of time. Serial executions, for example, do not need to wait at `MPI_Wait`. For these reasons, imbalance time percentages higher than 100 % are possible and reveal that execution times strongly differ between executions.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.65.: Imbalance Time Percentage for `LJ:void compute(...)`

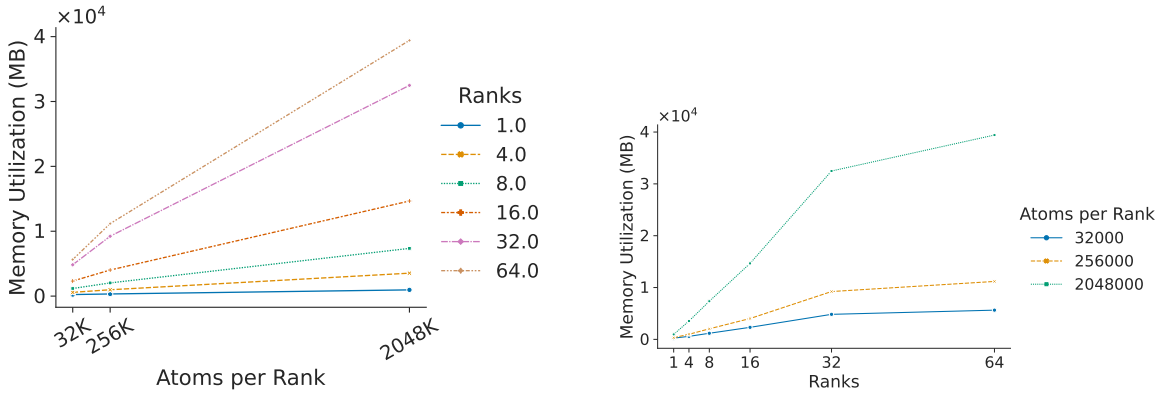


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.66.: Imbalance Time Percentage for `LJ:MPI_Wait`

Memory Utilization As shown in Figure 5.67b, the memory utilization increases linearly until 32 ranks, for a rising number of ranks. However, executions with 32 and 64 ranks show almost the same memory utilization. For parallel executions, the memory utilization is in the gigabyte range. With an increasing number of atoms per rank (see Figure 5.67a), memory utilization is linearly rising.

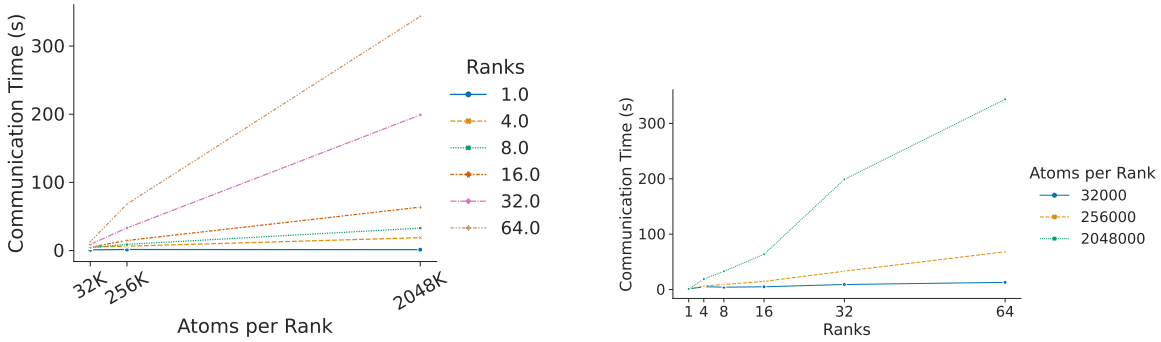


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.67.: Memory Utilization for LJ

Communication Time The communication time of this benchmark (see Figure 5.68) is similar to the imbalance time of `compute`, as shown in Figure 5.63. The communication time increases linearly for both increasing numbers of ranks and atoms per rank.

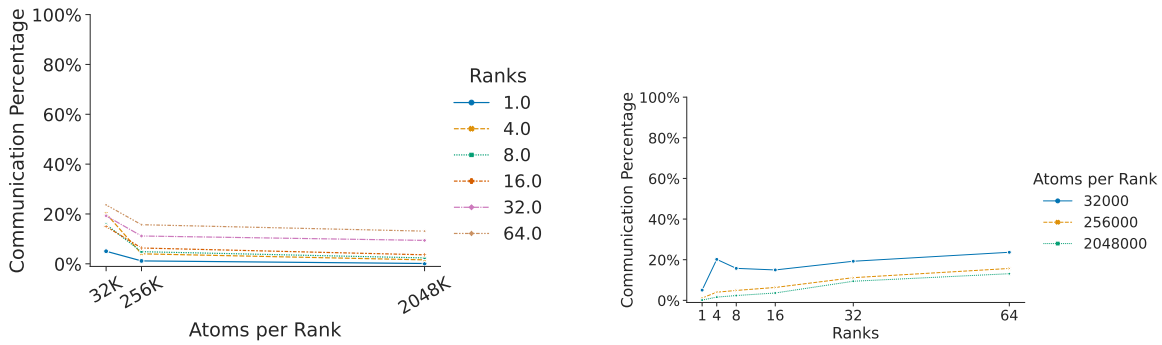


(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.68.: Communication Time for LJ

Communication Time Percentage When relating the communication time of the LJ benchmark to the whole execution time, the communication time percentage is below 25 % for all executions of this benchmark.



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.69.: Communication Time Percentage for `LJ`

Evaluation of `LJ` The two analyzed functions show a great difference in load imbalance. Although the load imbalance of `compute` might raise the assumption that this function is well-balanced, the imbalance percentage is above 20 % for parallel executions with 32×10^3 atoms per rank. While the imbalance time for executions with 32×10^3 and 256×10^3 atoms per rank remain close to zero, the executions with 2048×10^3 atoms per rank and 64 ranks have an imbalance time of nearly two minutes. Even though the imbalance time is relatively small in relation to the total execution time, it describes the time that can be saved at most if this function were perfectly balanced. Therefore, optimizing this function might be profitable for higher numbers of ranks and atoms per rank. Given that the plots of the functions `build` and `ev_tally` are similar to the plots of `compute`, we can assume that these functions also show inefficiencies in executions with higher numbers of ranks and atoms per rank. Since the sum of the time percentages of these functions (see Table 5.14) ranges from 56.43 % to 97.01 %, we can deduce that the load imbalance of these functions determines the overall load imbalance. Given that these functions are inefficient for higher numbers of ranks and atoms per rank, the same applies for the whole benchmark. Furthermore, it is noticeable that the metric values of executions with 32 and 64 ranks are similar or identical for most metrics.

The following tables depict the detailed performance data for the overall benchmark with varying cube lengths:

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	21.55	6.65×10^{10}	238.21	6.84×10^{10}	6.84×10^{10}	0.00	1.09	5.04 %
4	25.20	3.17×10^{11}	580.43	5.99×10^{10}	1.50×10^{10}	9.06	5.08	20.16 %
8	27.24	5.88×10^{11}	1.18×10^{3}	5.24×10^{10}	6.54×10^{9}	10.82	4.29	15.74 %
16	33.40	1.22×10^{12}	2.32×10^{3}	4.51×10^{10}	2.82×10^{9}	16.79	5.00	14.96 %
32	47.58	3.02×10^{12}	4.83×10^{3}	3.91×10^{10}	1.22×10^{9}	34.56	9.15	19.24 %
64	55.01	8.21×10^{12}	5.64×10^{3}	3.87×10^{10}	6.05×10^{8}	37.16	13.01	23.65 %

Table 5.15.: `lj`: Total Values for 32×10^3 Atoms per Rank

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	133.81	4.31×10^{11}	330.71	6.40×10^{10}	6.40×10^{10}	0.00	1.56	1.16 %
4	160.58	1.86×10^{12}	979.32	5.21×10^{10}	1.30×10^{10}	10.31	6.43	4.00 %
8	188.29	3.86×10^{12}	2.03×10^{3}	4.41×10^{10}	5.52×10^{9}	14.24	9.17	4.87 %
32	296.97	1.93×10^{13}	9.23×10^{3}	3.59×10^{10}	1.12×10^{9}	166.37	33.14	11.16 %
64	435.00	5.34×10^{13}	1.12×10^{4}	3.44×10^{10}	5.37×10^{8}	264.25	68.25	15.69 %

Table 5.16.: `lj`: Total Values for 256×10^3 Atoms per Rank

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	1.04×10^{3}	3.42×10^{12}	960.99	6.18×10^{10}	6.18×10^{10}	0.00	1.34	0.13 %
4	1.19×10^{3}	1.40×10^{13}	3.54×10^{3}	5.22×10^{10}	1.30×10^{10}	35.71	18.91	1.59 %
8	1.39×10^{3}	2.90×10^{13}	7.36×10^{3}	4.48×10^{10}	5.60×10^{9}	57.98	32.92	2.36 %
16	1.76×10^{3}	6.01×10^{13}	1.47×10^{4}	3.66×10^{10}	2.29×10^{9}	160.69	63.70	3.62 %
32	2.12×10^{3}	1.49×10^{14}	3.25×10^{4}	3.53×10^{10}	1.10×10^{9}	1.11×10^{3}	198.94	9.40 %
64	2.62×10^{3}	3.60×10^{14}	3.94×10^{4}	3.53×10^{10}	5.51×10^{8}	1.56×10^{3}	343.60	13.11 %

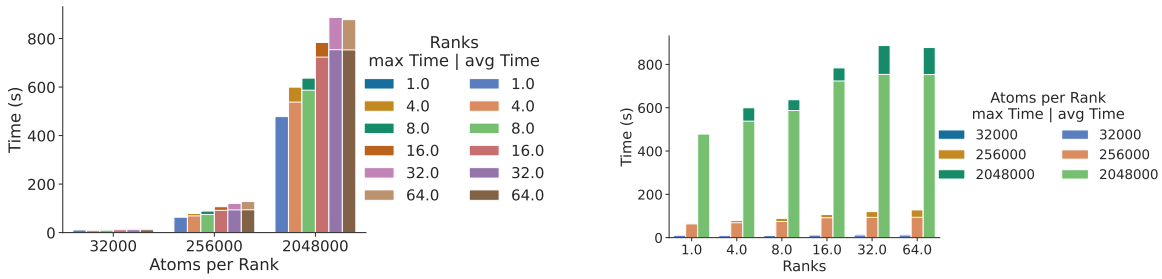
Table 5.17.: `lj`: Total Values for 2048×10^3 Atoms per Rank

5.3.3.2. Lennard-Jones (LJ) Benchmark with RCB Load Balancing Algorithm

In the following, we analyze the results of the measured metrics for the `LJ` benchmark with load balancing through the `RCB` algorithm. The functions with a time percentage greater than 5 % are depicted in Table 5.18. Similarly to the `LJ` benchmark without load balancing, the plots for `compute`, `build`, and `ev_tally` are similar. For comparability reasons, we only show the metrics for `compute`. Furthermore, we present the plots of the MPI function with the highest time percentage, in this case, `MPI_Waitany`. The remaining plots are featured in the appendix (see subsection A.2.2). Since `MPI_Waitany` does not get called for serial executions, the plots only display the functions for parallel executions. Given that the plots and their trend of this benchmark are similar to the plots of the `LJ` benchmark without load balancing (see subsubsection 5.3.3.1), we do not describe every plot in detail.

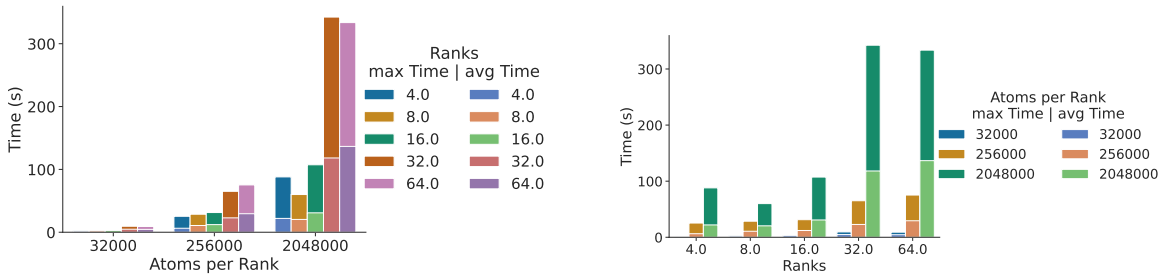
Function	Time Percentage
`virtual void compute(int, int)`	28.78 % to 49.19 %
`void build(NeighList*)`	29.01 % to 41.66 %
`void ev_tally(int, int, int, int, double, double, double, double, double, double)`	3.26 % to 6.31 %
`MPI_Waitany`	1.25 % to 12.29 %
`MPI_Send`	0.23 % to 10.04 %
`MPI_Init`	0.04 % to 6.14 %
other functions	<5 %

Table 5.18.: Time Percentage of `LJ with RCB` functions



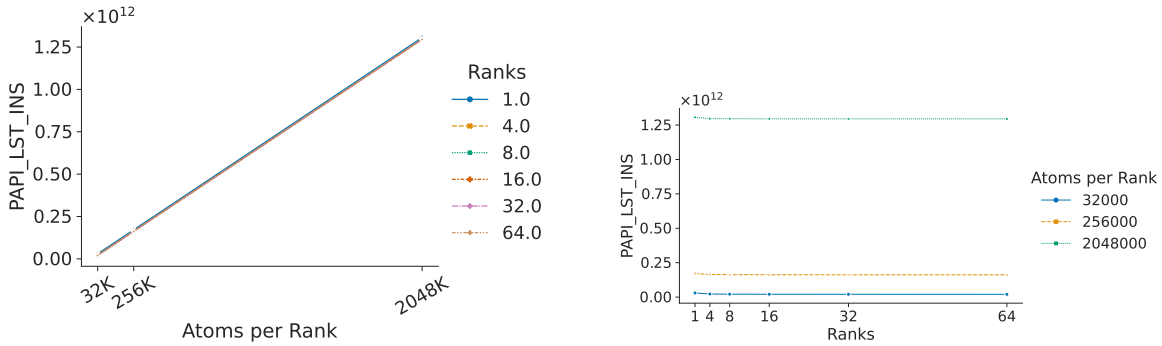
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.70.: Time for `RCB:void compute(...)`



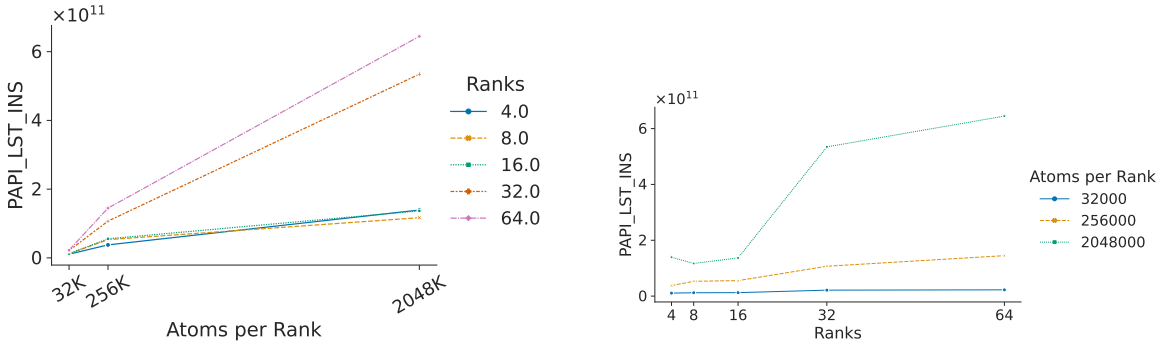
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.71.: Time for `RCB:MPI_Waitany`



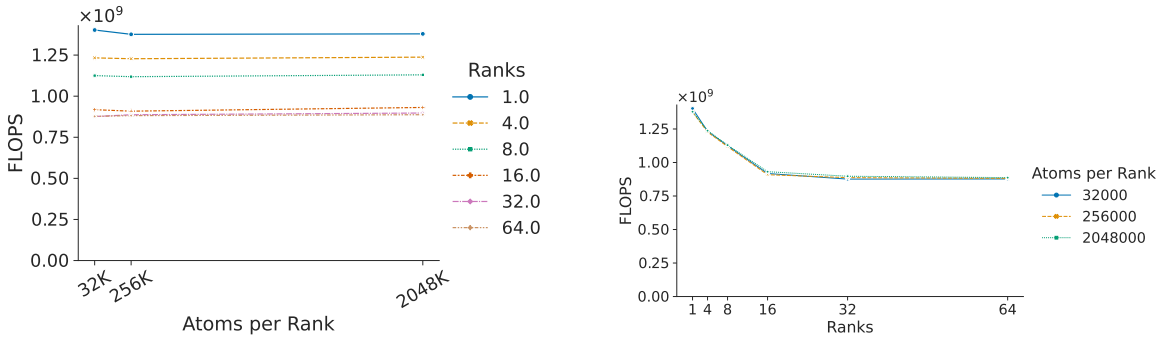
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.72.: PAPI_LST_INS for `RCB:void compute(...)`



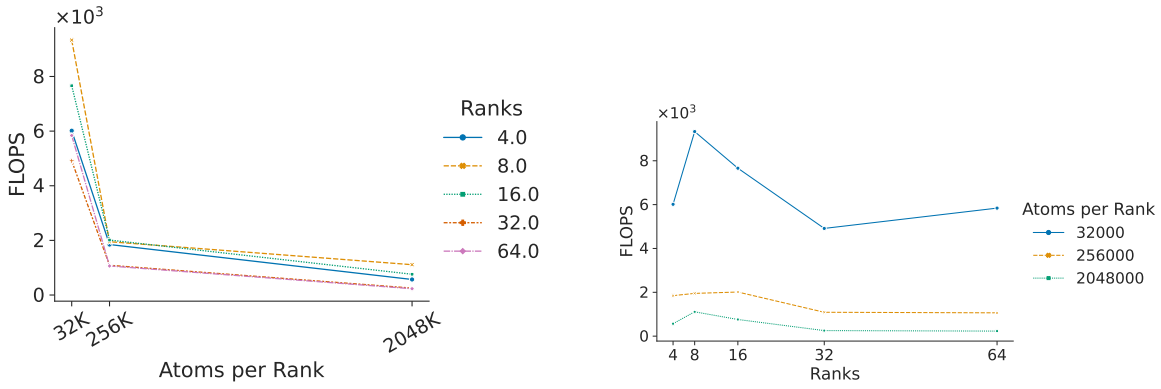
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.73.: PAPI_LST_INS for `RCB:MPI_Waitany`



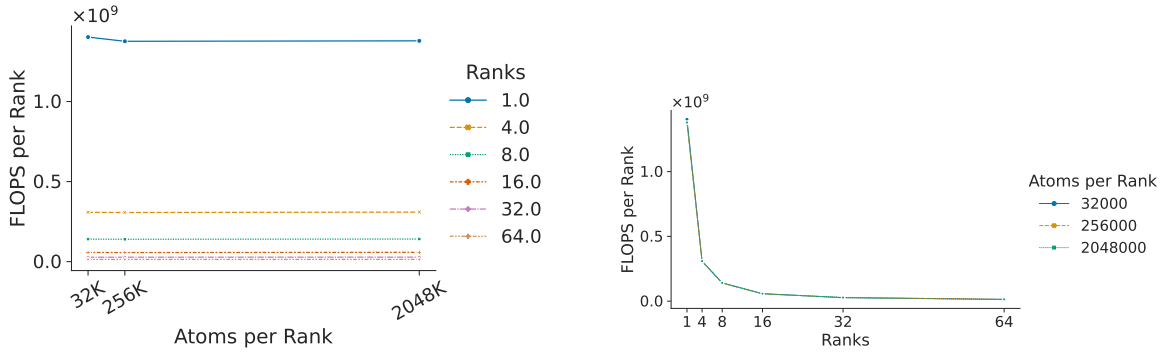
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.74.: FLOPS for `RCB:void compute(...)`



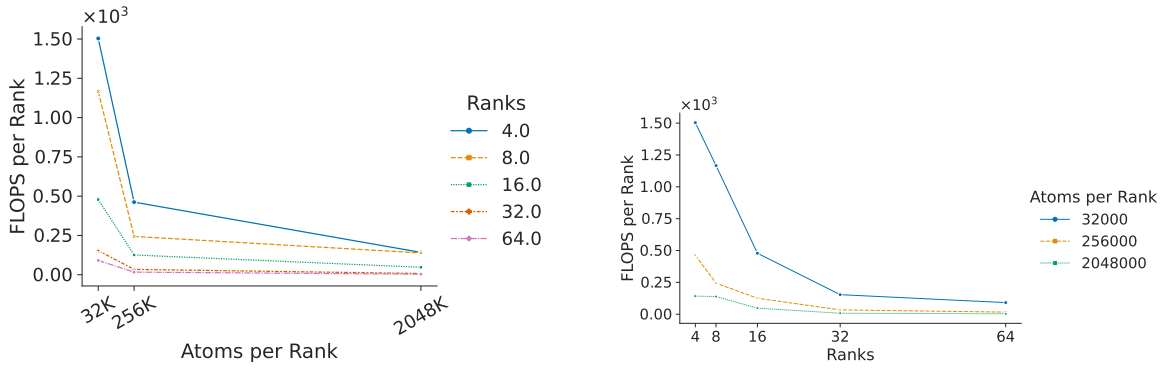
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.75.: FLOPS for `RCB:MPI_Waitany`



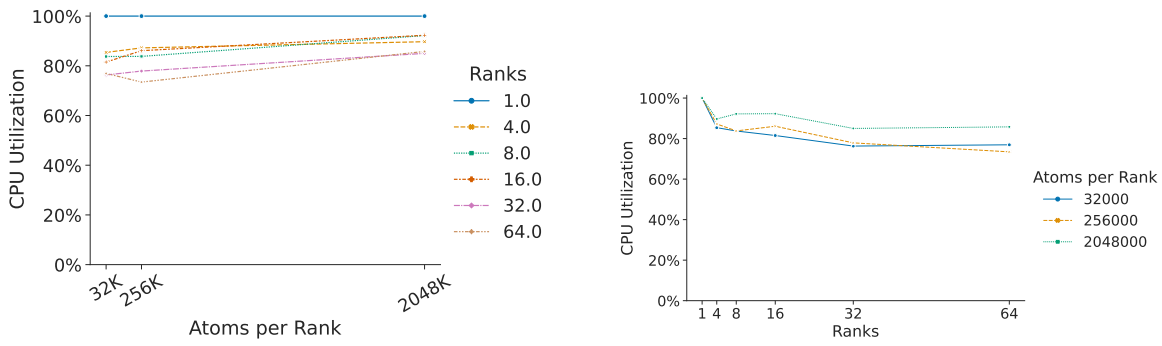
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.76.: FLOPS per Rank for `RCB:void compute(...)`



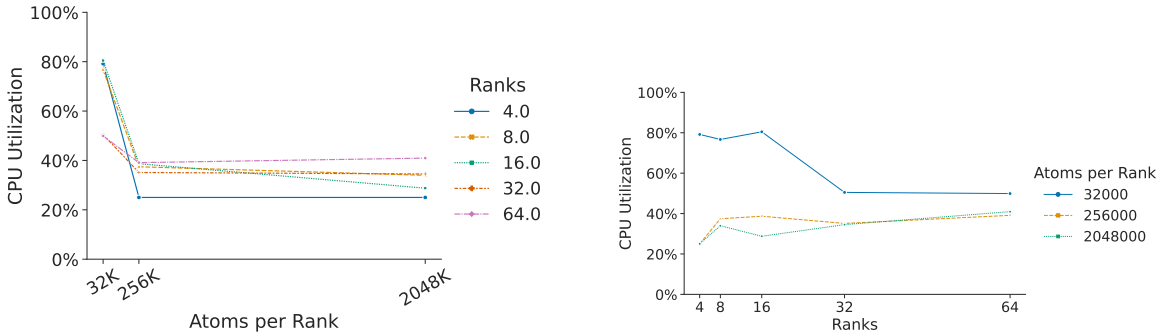
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure 5.77.: FLOPS per Rank for `RCB:MPI_Waitany`



(a) Variable problem size **(b)** Variable number of MPI ranks

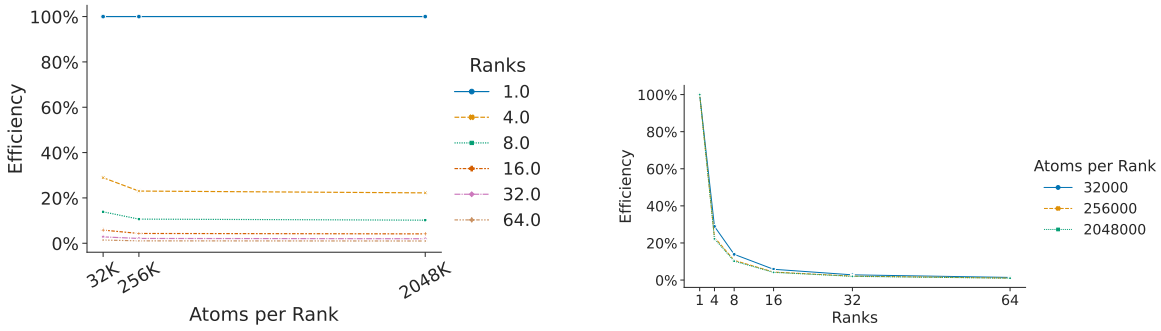
Figure 5.78.: CPU Utilization for `RCB:void compute(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

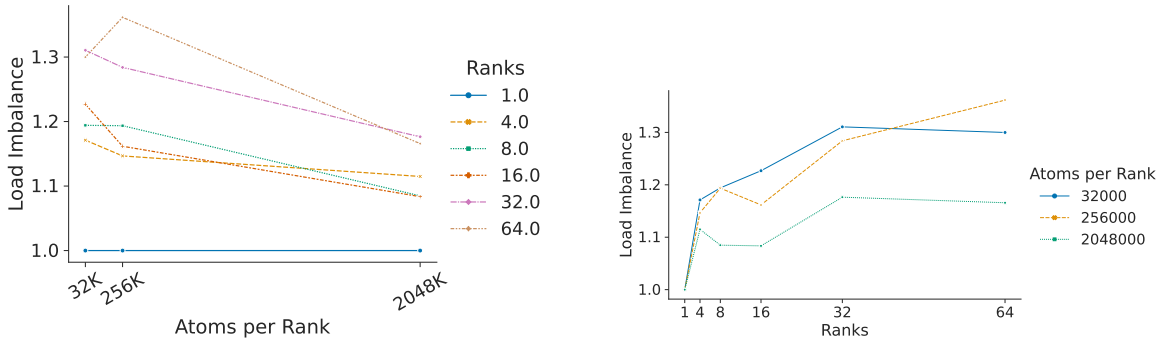
Figure 5.79.: CPU Utilization for `RCB:MPI_Waitany`



(a) Variable problem size

(b) Variable number of MPI ranks

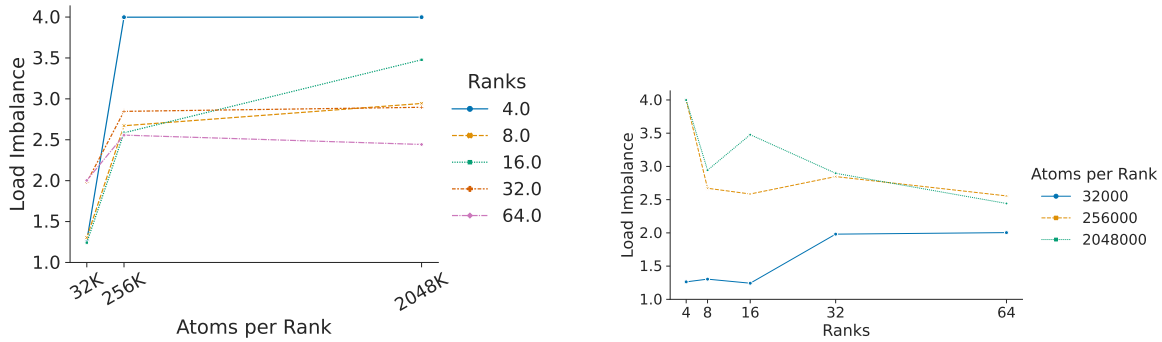
Figure 5.80.: Efficiency for `RCB:void compute(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

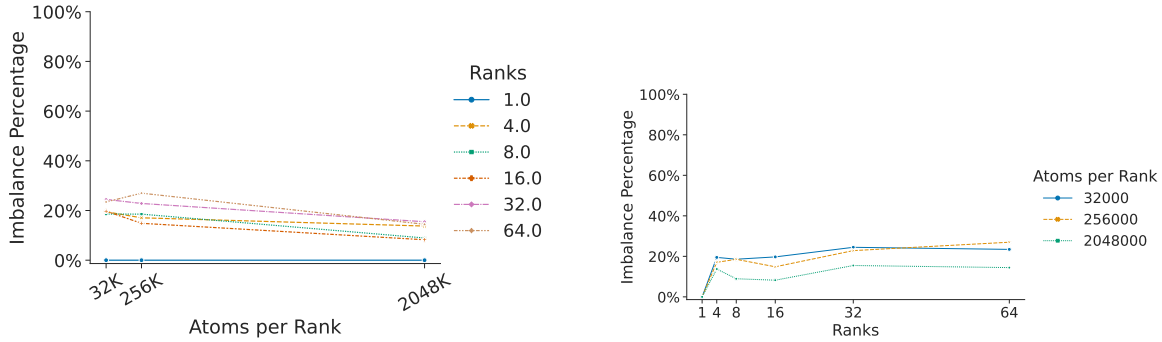
Figure 5.81.: Load Imbalance for `RCB:void compute(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

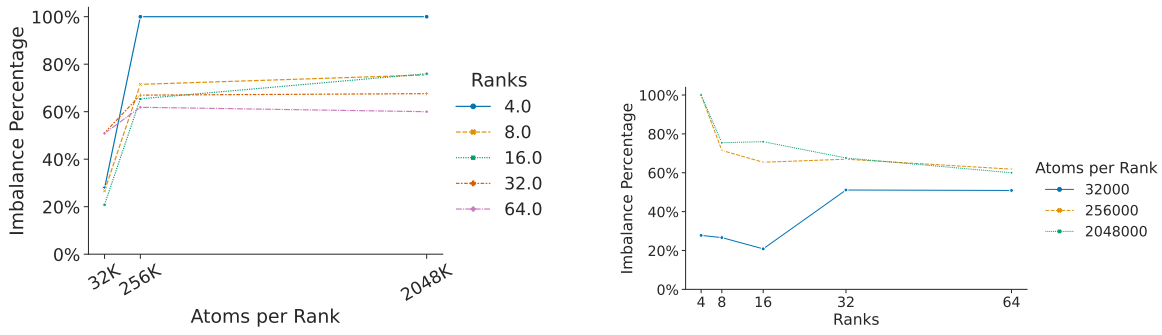
Figure 5.82.: Load Imbalance for `RCB:MPI_Waitany`



(a) Variable problem size

(b) Variable number of MPI ranks

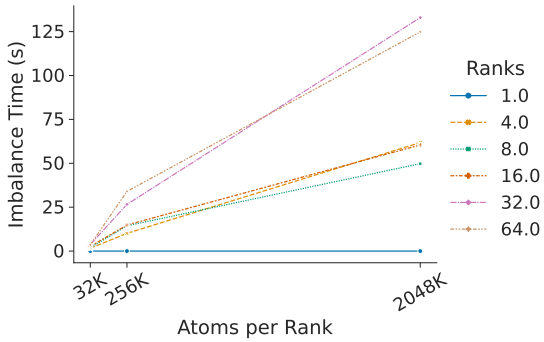
Figure 5.83.: Imbalance Percentage for `RCB:void compute(...)`



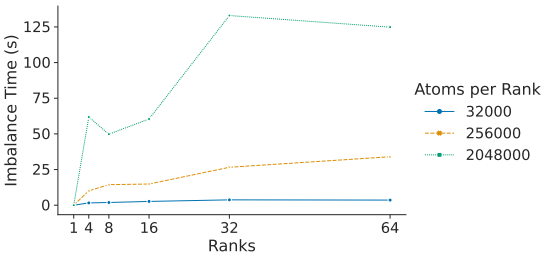
(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.84.: Imbalance Percentage for `RCB:MPI_Waitany`

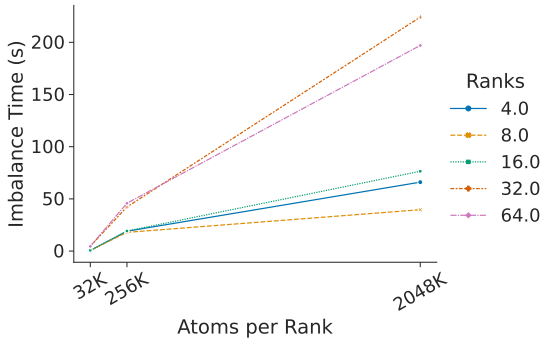


(a) Variable problem size

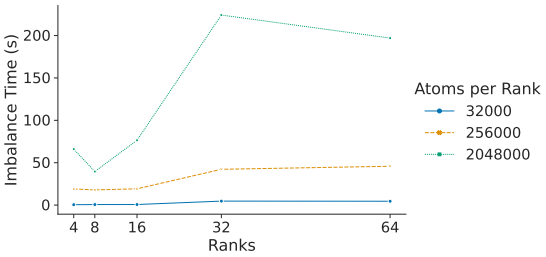


(b) Variable number of MPI ranks

Figure 5.85.: Imbalance Time for `RCB:void compute(...)`

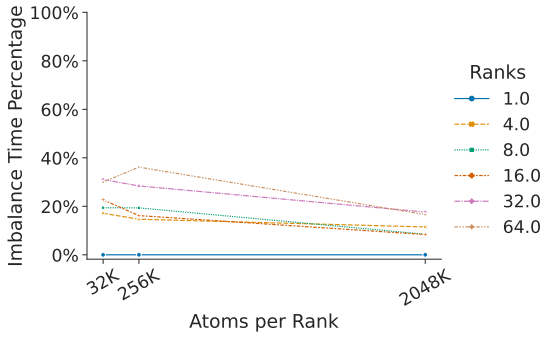


(a) Variable problem size

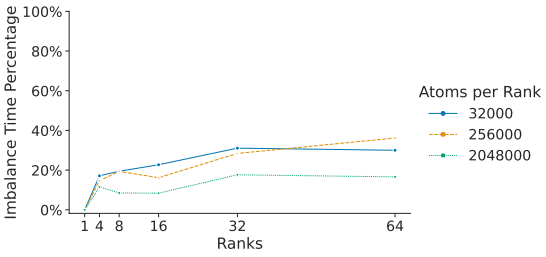


(b) Variable number of MPI ranks

Figure 5.86.: Imbalance Time for `RCB:MPI_Waitany`

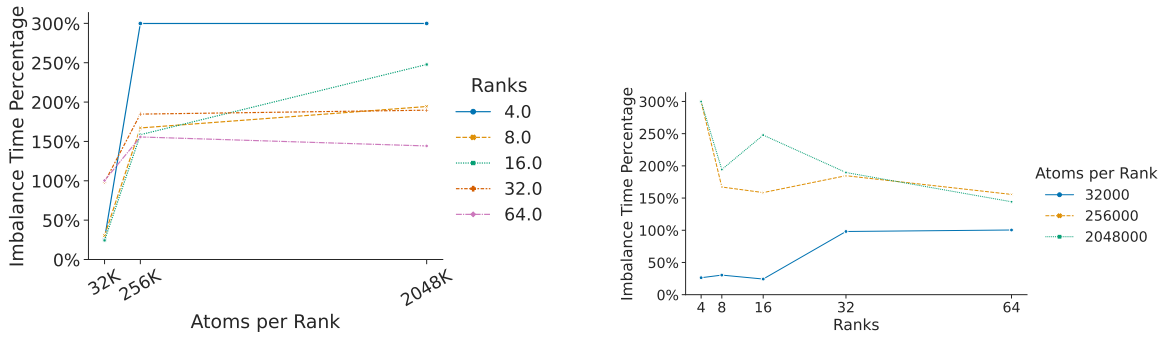


(a) Variable problem size



(b) Variable number of MPI ranks

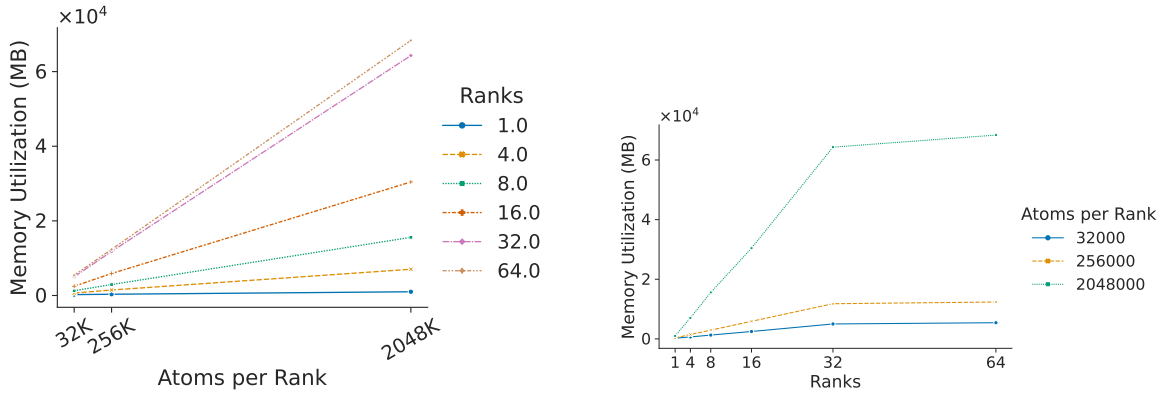
Figure 5.87.: Imbalance Time Percentage for `RCB:void compute(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

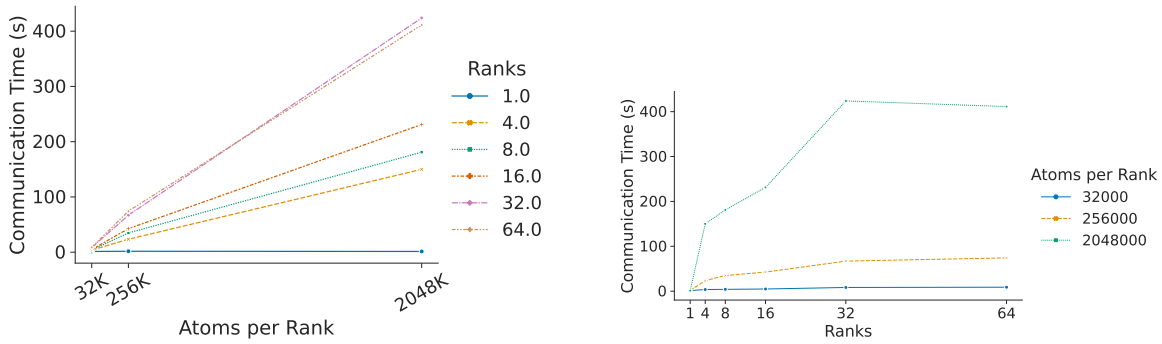
Figure 5.88.: Imbalance Time Percentage for `RCB:MPI_Waitany`



(a) Variable problem size

(b) Variable number of MPI ranks

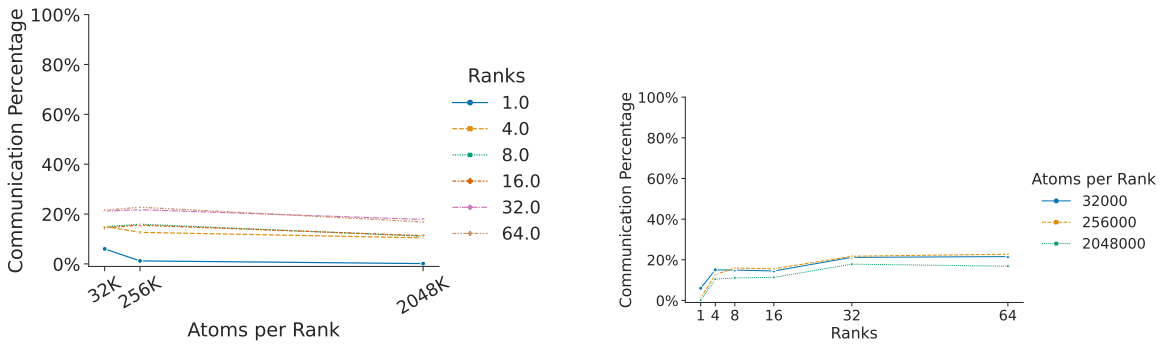
Figure 5.89.: Memory Utilization for `RCB`



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.90.: Communication Time for `RCB`



(a) Variable problem size

(b) Variable number of MPI ranks

Figure 5.91.: Communication Time Percentage for `RCB`

Evaluation of `RCB` Similarly to the `LJ` benchmark without load balancing, the two shown functions of this benchmark also have a great difference in load imbalance. Since the imbalance percentage of `ell_mvp` ranges from 10 % to 20 % for parallel executions, we can deduce that this compute-intensive function shows inefficiencies. With regard to the imbalance time, we can surely state that this benchmark might profit from an efficiency optimization for parallel executions with 256×10^3 or 2048×10^3 atoms per rank. Given that the plots of the functions `build` and `ev_tally` are similar to the plots of `compute`, we can assume that these functions also show the same inefficiencies. Since the sum of the time percentages of these functions (see Table 5.14) ranges from 61.05 % to 97.85 %, we can deduce that the load imbalance of these functions determines the overall load imbalance. Since these functions are inefficient, the same applies for the whole benchmark. Likewise to `LJ` without load balancing, the metric values of executions with 32 and 64 ranks are similar or identical for most metrics.

The following tables depict the detailed performance data for the overall benchmark with varying cube lengths:

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	22.06	6.81×10^{10}	240.62	7.44×10^{10}	7.44×10^{10}	0.00	1.33	6.02 %
4	24.52	2.93×10^{11}	649.50	7.33×10^{10}	1.83×10^{10}	2.96	3.69	15.03 %
8	27.27	6.02×10^{11}	1.28×10^{3}	6.31×10^{10}	7.89×10^{9}	4.27	4.07	14.93 %
16	33.51	1.24×10^{12}	2.48×10^{3}	5.24×10^{10}	3.27×10^{9}	8.01	4.85	14.48 %
32	39.07	3.00×10^{12}	5.01×10^{3}	4.83×10^{10}	1.51×10^{9}	15.36	8.28	21.18 %
64	40.96	6.64×10^{12}	5.43×10^{3}	4.73×10^{10}	7.40×10^{8}	15.17	8.83	21.56 %

Table 5.19.: `rcb`: Total Values for 32×10^3 Atoms per Rank

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	136.55	4.43×10^{11}	335.95	6.94×10^{10}	6.94×10^{10}	0.00	1.65	1.21 %
4	185.26	2.38×10^{12}	1.47×10^{3}	6.43×10^{10}	1.61×10^{10}	54.84	23.44	12.65 %
8	218.50	5.12×10^{12}	2.93×10^{3}	5.53×10^{10}	6.92×10^{9}	71.39	34.79	15.92 %
16	273.66	1.06×10^{13}	5.89×10^{3}	4.64×10^{10}	2.90×10^{9}	84.94	42.54	15.54 %
32	307.94	2.53×10^{13}	1.18×10^{4}	4.42×10^{10}	1.38×10^{9}	164.87	66.95	21.74 %
64	326.59	5.57×10^{13}	1.24×10^{4}	4.21×10^{10}	6.57×10^{8}	185.14	74.25	22.73 %

Table 5.20.: `rcb`: Total Values for 256×10^3 Atoms per Rank

Ranks	Time (s)	Load/Store Ins.	Mem. Util. (MB)	FLOPS	FLOPS p.R.	Imbalance (s)	Com (s)	Com (%)
1	1.10×10^3	3.62×10^{12}	1.00×10^3	6.88×10^{10}	6.88×10^{10}	0.00	1.32	0.12 %
4	1.44×10^3	1.82×10^{13}	7.03×10^3	6.45×10^{10}	1.61×10^{10}	273.73	149.92	10.45 %
8	1.63×10^3	3.76×10^{13}	1.56×10^4	5.46×10^{10}	6.82×10^9	273.44	181.02	11.08 %
16	2.04×10^3	7.60×10^{13}	3.04×10^4	4.39×10^{10}	2.74×10^9	456.69	231.13	11.32 %
32	2.38×10^3	1.83×10^{14}	6.43×10^4	4.33×10^{10}	1.35×10^9	1.03×10^3	423.90	17.84 %
64	2.44×10^3	3.86×10^{14}	6.84×10^4	3.98×10^{10}	6.22×10^8	1.00×10^3	411.28	16.82 %

Table 5.21.: `rcb`: Total Values for 2048×10^3 Atoms per Rank

5.3.3.3. Discussion

Since both benchmarks concern the Lennard-Jones (LJ) problem with the same execution configurations, the metrics of both benchmarks can be set directly in relation. The efficiency analysis of both benchmarks leads to the assumption that the benchmark without load balancing performs better in regard to efficiency. However, both benchmarks show inefficiencies in their main compute-intensive function. Consequently, they both might profit from an efficiency optimization for executions with a higher number of ranks and/or atoms per rank.

5.3.4. OpenFOAM

In the context of this thesis, the standard tutorial case `counterflowFlame2D_GRI` with the `reactingFoam` solver of OpenFOAM is used as a benchmark. Unfortunately, OpenFOAM v2312 could not be compiled and executed with the used measurement infrastructure Score-P. Consequently, we could not collect any performance data for OpenFOAM.

5.4. Efficiency Metrics Evaluation

This section evaluates the different efficiency metrics used in this thesis and compares them with each other. Although load imbalance is the most widely used metric for determining the efficiency of an HPC applications, the different range of this metric does not allow comparison between executions with different numbers of ranks. The imbalance percentage, however, relates the load imbalance to the number of ranks used. To locate the impact of an inefficiency, it is crucial to determine the imbalance time, which reveals how much time can be saved at most if the HPC application is perfectly balanced. In cases where the imbalance time is insignificant, a high load imbalance is irrelevant. In order to evaluate whether an efficiency optimization of the HPC application is profitable, the imbalance time needs to be related to the overhead caused by the optimization.

5.5. Assumptions and Limitations

The main limitation of this thesis is that we only analyze HPC applications with regard to their hardware resource efficiency, without considering energy efficiency. Furthermore, we focus on distributed memory parallelization with MPI, while shared memory parallelization with OpenMP remains out of consideration. In addition, we do not try to recommend a specific HPC application for a specific use case. We only examine a set of real-world applications while analyzing their resource efficiency. Moreover, we do not concern ourselves with the analyzed benchmarks in detail. Hence, we cannot contextualize occurring inefficiencies.

5.6. Threats to Validity

This section presents the threats to the validity of this thesis. Generally, threats to validity can be categorized into internal and external threats. The internal threats to the validity of this thesis include primarily the usage of external tools. These tools could not work as expected, which would invalidate our results. Furthermore, the default compiler instrumentation with Score-P induces an overhead that cannot be prevented when measuring the metrics for each callpath in detail. Moreover, we have different runs for each recorded PAPI metric, which creates different degrees of overhead. Hence, we derive our metrics from the measurement data of the same PAPI execution. Another internal threat is the system noise of the bwUniCluster 2.0 on which we execute the HPC applications. To reduce the impact of system noise on our measurement data, we execute each configuration of our benchmarks five times. Over these five repetitions, we calculate the average in order to derive our metrics. External threats to validity cover the generalization of our results. Although our executions are in the minute range, we assume that our findings are also representative for longer executions. Furthermore, our input values do not cover all possible values. Additionally, we only consider functions with a time percentage greater than 5 % in our efficiency analysis. For these reasons, care must be taken when generalizing our results.

6. Conclusion

Finally, we conclude this thesis in this last chapter. Section 6.1 gives a summary of the work of this thesis, and section 6.2 provides an outlook on future work.

6.1. Summary

In this thesis we have analyzed the inefficiencies of HPC applications and evaluated metrics on how well they represent these inefficiencies. We have considered more than pure load imbalance to allow a better comparison of benchmark runs with different numbers of ranks. In this thesis, we worked out that the imbalance time is crucial for evaluating the efficiency of an HPC application. With our metrics, we identified inefficiencies in the examined HPC applications and found out that, in most cases, with higher numbers or ranks and/or a larger problem size, an efficiency optimization might raise the efficiency of the application. Our contributions are the measurements of different metrics that describe how well the resources of HPC systems are utilized by the applications we examined. This helps researchers assess whether an optimization strategy to improve the efficiency of HPC applications is profitable.

6.2. Future Work

Based on the efficiency analysis of real-world HPC applications in this thesis, future work could deal with the efficiency optimization of those HPC applications by means of different efficiency optimization approaches. Future studies could create a performance model that estimates beforehand whether optimizing a specific HPC application could show positive efficiency results. Metrics for the creation of the performance model in the case of the Software Resource Disaggregation approach could include response time, latency, bandwidth, or throughput. If the performance model promises an efficiency gain, the selected efficiency optimization approach could be implemented, and the results could be related to the execution time and efficiency of the original implementation. Furthermore, the results of different efficiency optimization approaches for different HPC applications could be compared to one another in order to find the best optimization strategy and save resources in the long run.

Acknowledgments

The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Bibliography

[1] Jimmy Aguilar Mena et al. “Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB”. In: *Proceedings of the 51st International Conference on Parallel Processing*. ICPP ’22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–11. ISBN: 978-1-4503-9733-9. DOI: 10.1145/3545008.3545045.

[2] Victor R. Basili and David M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984). Conference Name: IEEE Transactions on Software Engineering, pp. 728–738. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010301.

[3] Milind Bhandarkar et al. “Adaptive Load Balancing for MPI Programs”. en. In: *Computational Science - ICCS 2001*. Ed. by Vassil N. Alexandrov et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 108–117. ISBN: 978-3-540-45718-3. DOI: 10.1007/3-540-45718-6_13.

[4] Christian Bischof, Dieter an Mey, and Christian Iwainsky. “Brainware for green HPC”. en. In: *Computer Science - Research and Development* 27.4 (Nov. 2012), pp. 227–233. ISSN: 1865-2042. DOI: 10.1007/s00450-011-0198-5.

[5] David Böhme, Felix Wolf, and Markus Geimer. “Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. May 2012, pp. 2538–2541. DOI: 10.1109/IPDPSW.2012.321.

[6] David Böhme and Felix Gerd Eugen Wolf. “Characterizing load and communication imbalance in parallel applications”. en. Number: RWTH-CONV-144050. PhD thesis. Forschungszentrum Jülich, Zentralbibliothek, 2014. URL: https://publications.rwth-aachen.de/record/229075 (visited on 02/27/2024).

[7] David Böhme et al. “Scalable Critical-Path Based Performance Analysis”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. ISSN: 1530-2075. May 2012, pp. 1330–1340. DOI: 10.1109/IPDPS.2012.120.

[8] Carlos Boneti et al. “Balancing HPC applications through smart allocation of resources in MT processors”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. ISSN: 1530-2075. Apr. 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536293.

[9] *BwUniCluster2.0 - bwHPC Wiki*. URL: https://wiki.bwhpc.de/e/BwUniCluster2.0 (visited on 12/04/2023).

[10] *BwUniCluster2.0/Slurm - bwHPC Wiki*. URL: https://wiki.bwhpc.de/e/BwUniCluster2.0/Slurm#OpenMPI_with_Multithreading (visited on 05/10/2024).

[11] Alexandru Calotoiu et al. “ExtraPeak: Advanced Automatic Performance Modeling for HPC Applications”. en. In: *Software for Exascale Computing - SPPEXA 2016-2019*. Ed. by Hans-Joachim Bungartz et al. Lecture Notes in Computational Science and Engineering. Cham: Springer International Publishing, 2020, pp. 453–482. ISBN: 978-3-030-47956-5. DOI: 10.1007/978-3-030-47956-5_15.

[12] Marcin Copik et al. “rFaaS: Enabling High Performance Serverless with RDMA and Leases”. en. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. St. Petersburg, FL, USA: IEEE, May 2023, pp. 897–907. DOI: 10.1109/IPDPS54959.2023.00094.

[13] Marcin Copik et al. *rFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing*. June 2021.

[14] Marcin Copik et al. “SeBS: a serverless benchmark suite for function-as-a-service computing”. In: *Proceedings of the 22nd International Middleware Conference*. Middleware ’21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 64–78. ISBN: 978-1-4503-8534-3. DOI: 10.1145/3464298.3476133.

[15] Marcin Copik et al. “Software Resource Disaggregation for HPC with Serverless Computing”. en. In: (May 2022). URL: https://spcl.inf.ethz.ch/Publications/.pdf/2022_copik_serverless_hpc_report.pdf (visited on 11/28/2023).

[16] Luiz DeRose, Bill Homer, and Dean Johnson. “Detecting Application Load Imbalance on High End Massively Parallel Systems”. en. In: *Euro-Par 2007 Parallel Processing*. Ed. by Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 150–159. ISBN: 978-3-540-74466-5. DOI: 10.1007/978-3-540-74466-5_17.

[17] *eSam(oa)² - SFCs and Adaptive Meshes for Oceanic And Other Applications*. URL: https://github.com/mohellen/eSamoa (visited on 11/30/2023).

[18] Stuart Faulk et al. “Measuring High Performance Computing Productivity”. en. In: *The International Journal of High Performance Computing Applications* 18.4 (Nov. 2004). Publisher: SAGE Publications Ltd STM, pp. 459–473. ISSN: 1094-3420. DOI: 10.1177/1094342004048539.

[19] James Fischer, Vincent Natoli, and David Richie. “Optimization of LAMMPS”. In: *2006 HPCMP Users Group Conference (HPCMP-UGC’06)*. June 2006, pp. 374–377. DOI: 10.1109/HPCMP-UGC.2006.56.

[20] Guido Giuntoli. “Hybrid CPU/GPU implementation for the FE2 multi-scale method for composite problems”. eng. Doctoral thesis. Universitat Politècnica de Catalunya, Jan. 2020. URL: https://upcommons.upc.edu/handle/2117/180785 (visited on 04/29/2024).

[21] Guido Giuntoli et al. “Hybrid CPU/GPU FE2 multi-scale implementation coupling Alya and Micropp”. en. In: *SC ’19: The International Conference for High Performance Computing, Networking, Storage, and Analysis*. New York, NY, USA: Association for Computing Machinery, Nov. 2019.

[22] Ao Mo-Hellenbrand et al. “A Large-Scale Malleable Tsunami Simulation Realized on an Elastic MPI Infrastructure”. In: *Proceedings of the Computing Frontiers Conference*. CF’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 271–274. ISBN: 978-1-4503-4487-6. DOI: 10.1145/3075564.3075585.

[23] Chao Huang, Orion Lawlor, and L. V. Kalé. “Adaptive MPI”. en. In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 306–322. ISBN: 978-3-540-24644-2. DOI: 10.1007/978-3-540-24644-2_20.

[24] Morris A. Jette and Tim Wickberg. “Architecture of the Slurm Workload Manager”. en. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dalibor Klusáček, Julita Corbalán, and Gonzalo P. Rodrigo. Cham: Springer Nature Switzerland, 2023, pp. 3–23. ISBN: 978-3-031-43943-8. DOI: 10.1007/978-3-031-43943-8_1.

[25] Laxmikant Kale et al. *UIUC-PPL/charm: Charm++ version 7.0.0*. Oct. 2021. DOI: 10.5281/ZENODO.3370873.

[26] *KarypisLab/METIS*. Mar. 2020. URL: https://github.com/KarypisLab/METIS (visited on 04/30/2024).

[27] Andreas Knüpfer et al. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir”. en. In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31475-9. DOI: 10.1007/978-3-642-31476-6_7.

[28] David J. Kuck. “Productivity in High Performance Computing”. en. In: *The International Journal of High Performance Computing Applications* 18.4 (Nov. 2004). Publisher: SAGE Publications Ltd STM, pp. 489–504. ISSN: 1094-3420. DOI: 10.1177/1094342004048541.

[29] *LAMMPS Benchmarks*. URL: https://www.lammps.org/bench.html#lj (visited on 04/16/2024).

[30] John Christian Linford et al. “Detecting Load Imbalance in Massively Parallel Applications”. en. In: *Forschungszentrum Jülich, Tech. Rep. FZJ-JSC-IB-2008-09* (Dec. 2008).

[31] Oliver Meister and Michael Bader. “2D adaptivity for 3D problems: Parallel SPE10 reservoir simulation on dynamically adaptive prism grids”. In: *Journal of Computational Science*. Computational Science at the Gates of Nature 9 (July 2015), pp. 101–106. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2015.04.016.

[32] Oliver Meister, Kaveh Rahnema, and Michael Bader. “Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells”. In: *ACM Transactions on Mathematical Software* 43.3 (Sept. 2016), 19:1–19:27. ISSN: 0098-3500. DOI: 10.1145/2947668.

[33] Harshitha Menon et al. “Automated Load Balancing Invocation Based on Application Characteristics”. In: *2012 IEEE International Conference on Cluster Computing*. ISSN: 2168-9253. Sept. 2012, pp. 373–381. DOI: 10.1109/CLUSTER.2012.61.

[34] Julian Morillo et al. "Hybrid parallelization of molecular dynamics simulations to reduce load imbalance". en. In: *The Journal of Supercomputing* 78.7 (May 2022), pp. 9184–9215. ISSN: 1573-0484. DOI: 10.1007/s11227-021-04214-4.

[35] Michael Moyles. *Performance Analysis of Fluid-Structure Interactions using OpenFOAM*. Tech. rep. Zenodo, Sept. 2012. DOI: 10.5281/zenodo.814504.

[36] General Bathymetric Chart of the Oceans. *Gridded bathymetry data (General Bathymetric Chart of the Oceans)*. en. URL: https://www.gebco.net/data_and_products/gridded_bathymetry_data/ (visited on 03/21/2024).

[37] *OpenFOAM*. June 2023. URL: https://www.openfoam.com/ (visited on 11/30/2023).

[38] *PAPI*. URL: https://icl.utk.edu/papi/ (visited on 12/04/2023).

[39] Steve Plimpton. "Fast Parallel Algorithms for Short-Range Molecular Dynamics". In: *Journal of Computational Physics* 117.1 (Mar. 1995), pp. 1–19. ISSN: 0021-9991. DOI: 10.1006/jcph.1995.1039.

[40] Sebastian Rettenberger et al. "ASAGI: A Parallel Server for Adaptive Geoinformation". In: *Proceedings of the Exascale Applications and Software Conference 2016*. EASC '16. New York, NY, USA: Association for Computing Machinery, Apr. 2016, pp. 1–9. ISBN: 978-1-4503-4122-6. DOI: 10.1145/2938615.2938618.

[41] Roberto Ribeiro, Luís Paulo Santos, and João Miguel Nóbrega. "nSharma: Numerical Simulation Heterogeneity Aware Runtime Manager for OpenFOAM". en. In: *Computational Science – ICCS 2018*. Ed. by Yong Shi et al. Cham: Springer International Publishing, 2018, pp. 429–443. ISBN: 978-3-319-93698-7. DOI: 10.1007/978-3-319-93698-7_33.

[42] Pavel Saviankou and Cube Developer Community. *CubeLib: General purpose C++ library and tools*. Sept. 2023. DOI: 10.5281/ZENODO.8345175.

[43] Pavel Saviankou, Anke Visser, and Cube Developer Community. *CubeGUI: Graphical explorer*. Sept. 2023. DOI: 10.5281/ZENODO.8345207.

[44] *Scaling - HPC Wiki*. URL: https://hpc-wiki.info/hpc/Scaling (visited on 05/13/2024).

[45] Zhi Shang. "Impact of mesh partitioning methods in CFD for large scale parallel computing". In: *Computers & Fluids* 103 (Nov. 2014), pp. 1–5. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2014.07.016.

[46] *SPE Comparative Solution Project - dataset 2*. URL: https://www.spe.org/web/csp/datasets/set02.htm (visited on 03/21/2024).

[47] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. "Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles". In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. ISSN: 2167-4337. Nov. 2010, pp. 1–11. DOI: 10.1109/SC.2010.47.

[48] Aidan P. Thompson et al. “LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales”. In: *Computer Physics Communications* 271 (Feb. 2022), p. 108171. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2021.108171.

[49] *TOP500*. Nov. 2023. URL: https://www.top500.org/lists/top500/2023/11/ (visited on 12/04/2023).

[50] Michael Waskom. “seaborn: statistical data visualization”. In: *Journal of Open Source Software* 6.60 (Apr. 2021), p. 3021. ISSN: 2475-9066. DOI: 10.21105/joss.03021.

[51] H.G. Weller et al. “A Tensorial Approach to Computational Continuum Mechanics Using Object Orientated Techniques”. In: *Computers in Physics* 12 (Nov. 1998), pp. 620–631. DOI: 10.1063/1.168744.

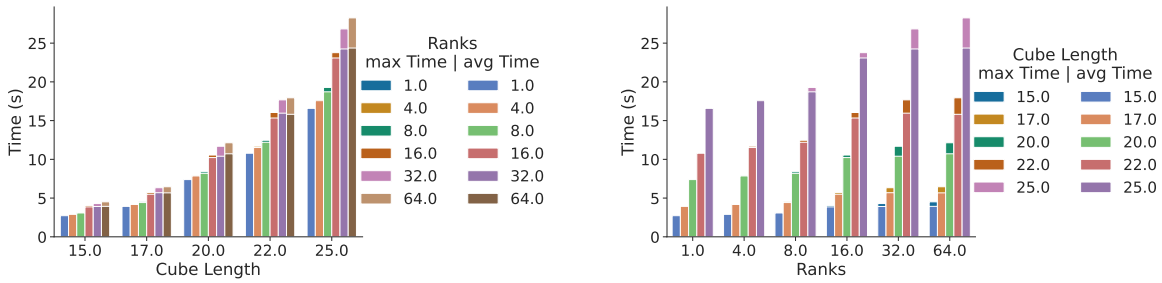
[52] T. Zirwes et al. “Optimizing Load Balancing of Reacting Flow Solvers in OpenFOAM for High Performance Computing”. de. In: *6th ESI OpenFOAM User Conference 2018, Hamburg, 23.-25. Oktober 2018* (2018), p. 1. URL: https://publikationen.bibliothek.kit.edu/1000085571 (visited on 04/07/2024).

[53] Thorsten Zirwes et al. “Enhancing OpenFOAM’s Performance on HPC Systems”. en. In: *High Performance Computing in Science and Engineering ’19*. Ed. by Wolfgang E. Nagel, Dietmar H. Kröner, and Michael M. Resch. Cham: Springer International Publishing, 2021, pp. 225–239. ISBN: 978-3-030-66792-4. DOI: 10.1007/978-3-030-66792-4_16.

A. Appendix

A.1. Micropp

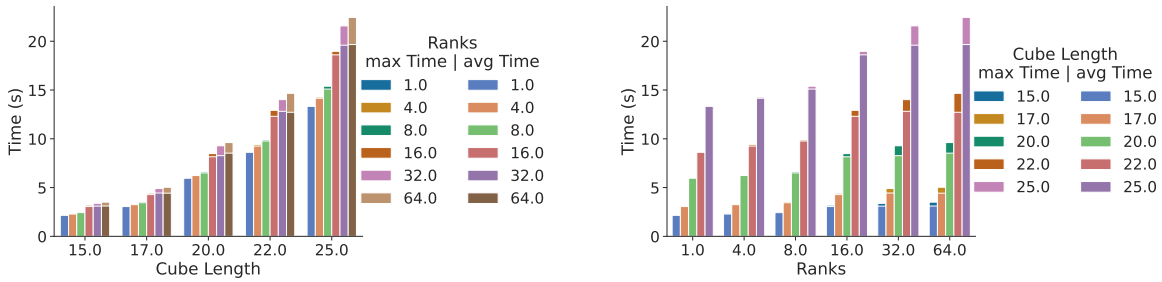
A.1.1. `benchmark-sc2019` Benchmark



(a) Variable problem size

(b) Variable number of MPI ranks

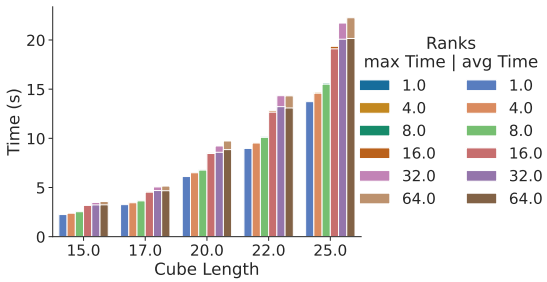
Figure A.1.: Time for `benchmark-sc2019:virtual void get_stress(...)`



(a) Variable problem size

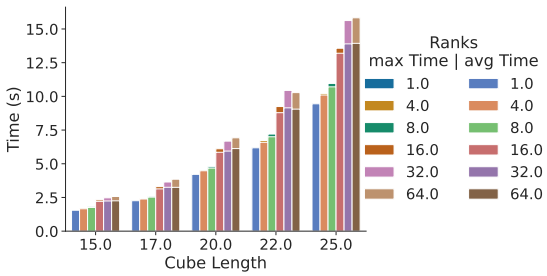
(b) Variable number of MPI ranks

Figure A.2.: Time for `benchmark-sc2019:bool damage_law(...)`



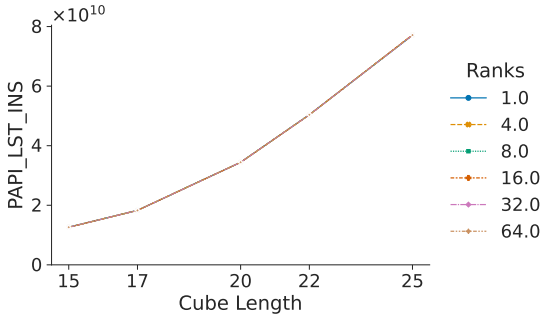
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.3.: Time for `benchmark-sc2019:void get_elem_mat(...)`



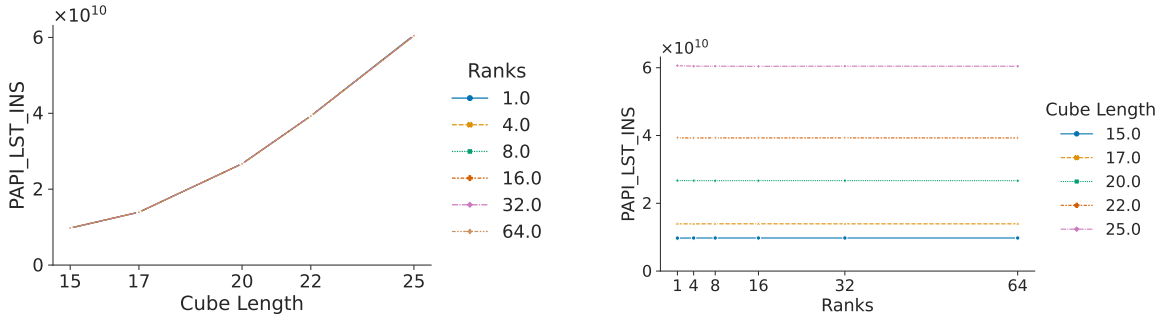
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.4.: Time for `benchmark-sc2019:void apply_perturbation(...)`



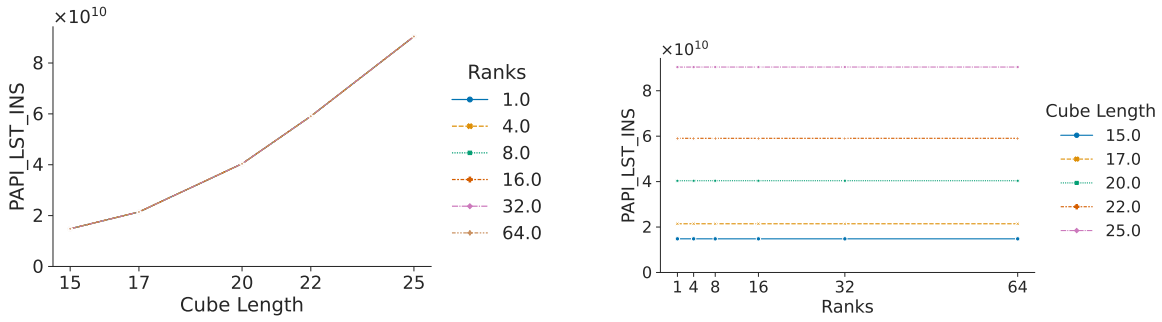
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.5.: PAPI_LST_INS for `benchmark-sc2019:virtual void get_stress(...)`



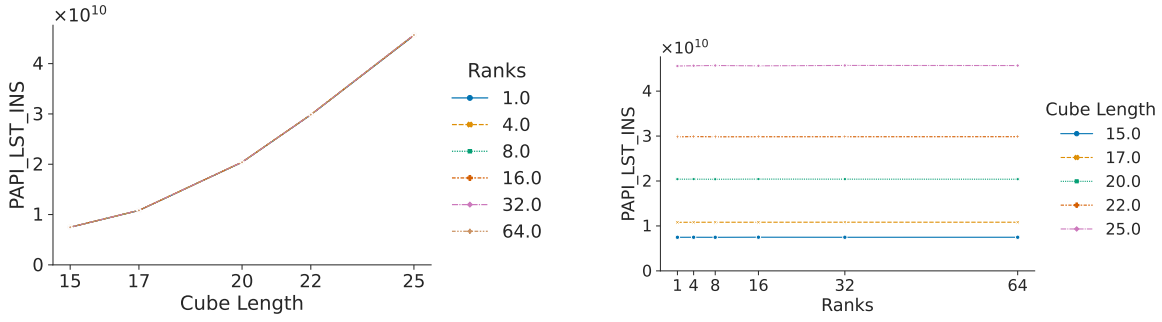
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.6.: PAPI_LST_INS for `benchmark-sc2019:bool damage_law(...)`



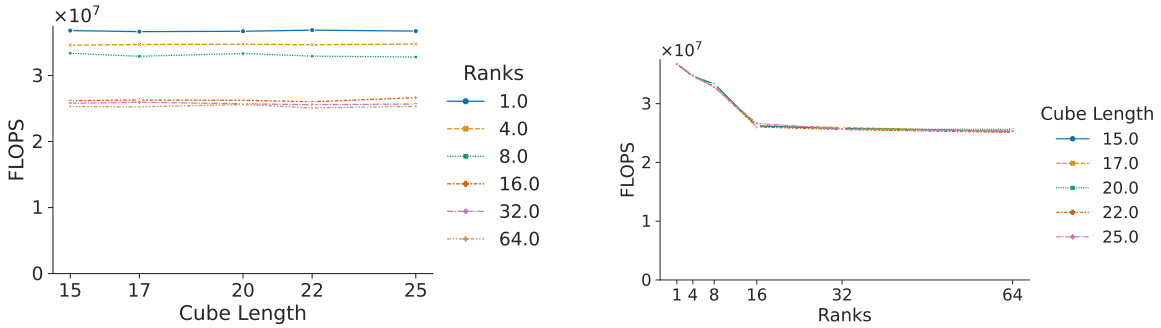
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.7.: PAPI_LST_INS for `benchmark-sc2019:void get_elem_mat(...)`



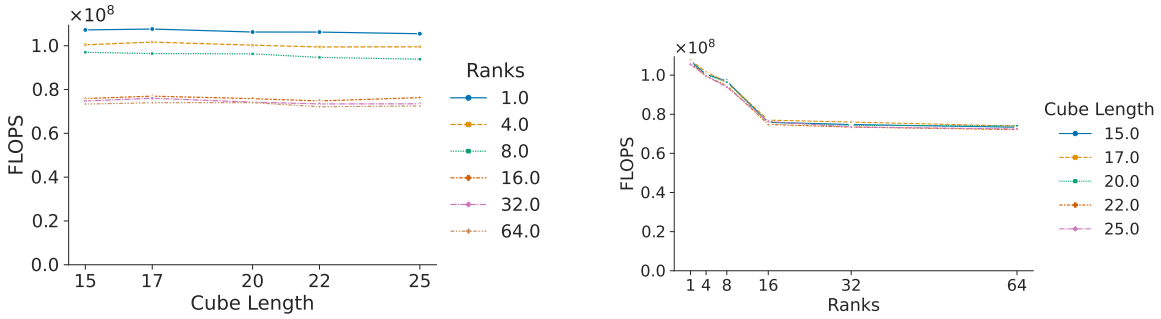
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.8.: PAPI_LST_INS for `benchmark-sc2019:void apply_perturbation(...)`



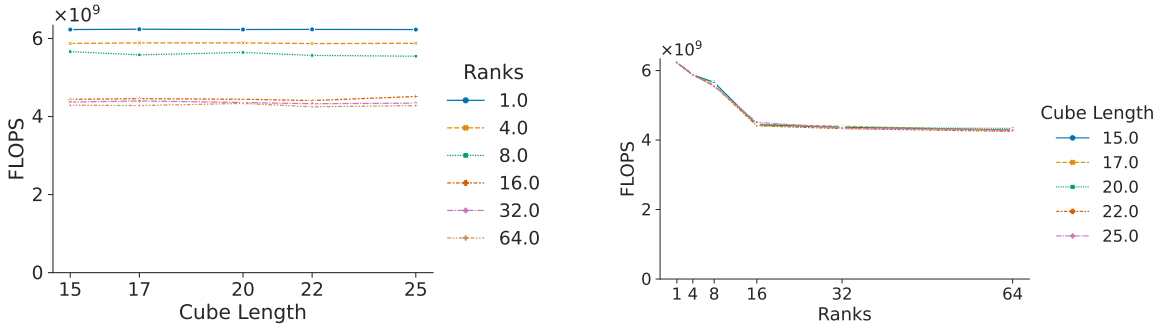
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.9.: FLOPS for `benchmark-sc2019:virtual void get_stress(...)`



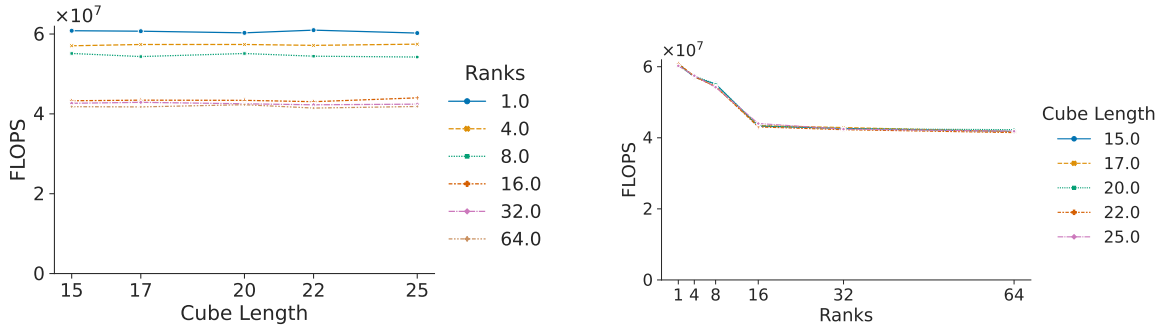
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.10.: FLOPS for `benchmark-sc2019:bool damage_law(...)`



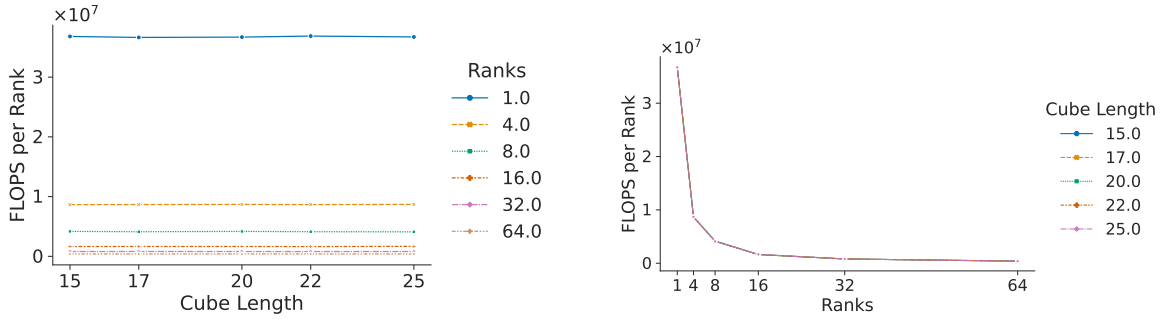
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.11.: FLOPS for `benchmark-sc2019:void get_elem_mat(...)`



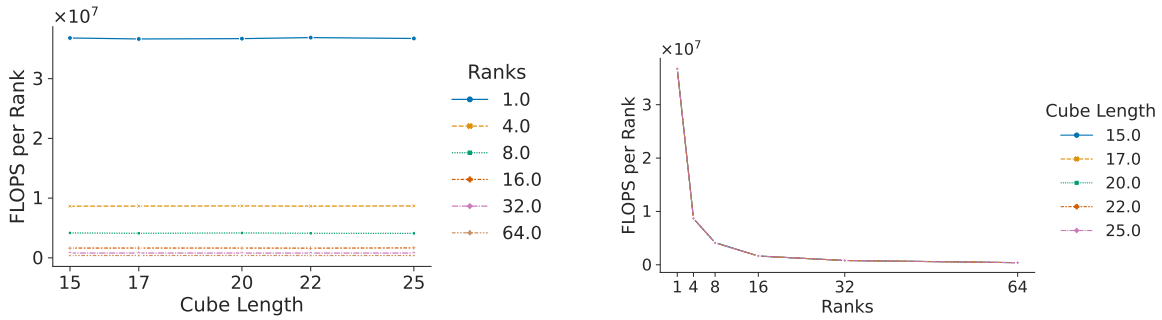
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.12.: FLOPS for `benchmark-sc2019:void apply_perturbation(...)`



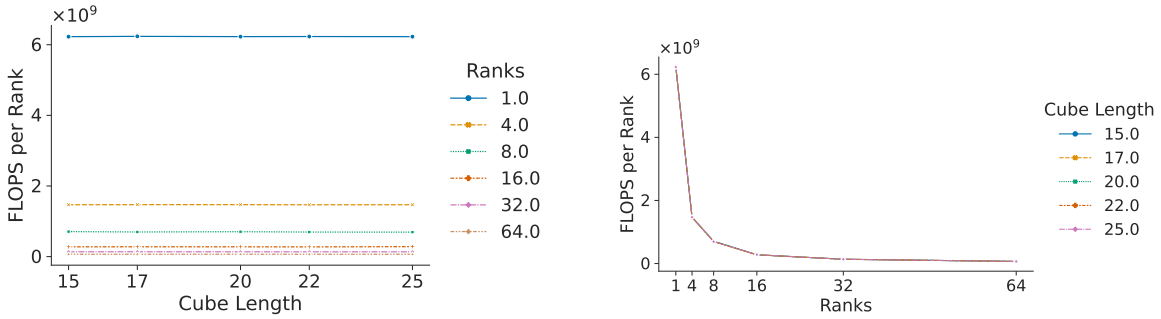
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.13.: FLOPS per Rank for `benchmark-sc2019:virtual void get_stress(...)`



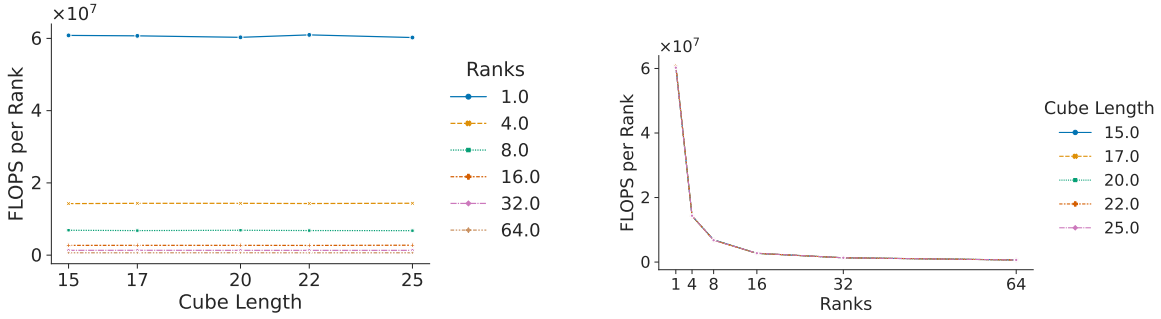
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.14.: FLOPS per Rank for `benchmark-sc2019:bool damage_law(...)`



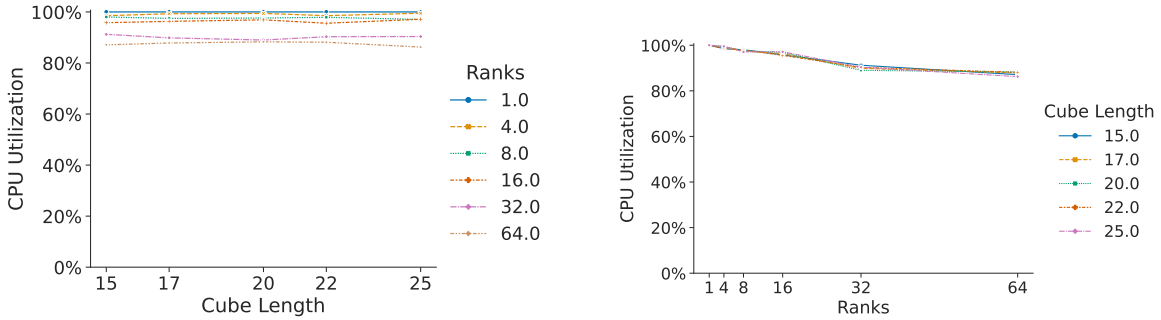
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.15.: FLOPS per Rank for `benchmark-sc2019:void get_elem_mat(...)`



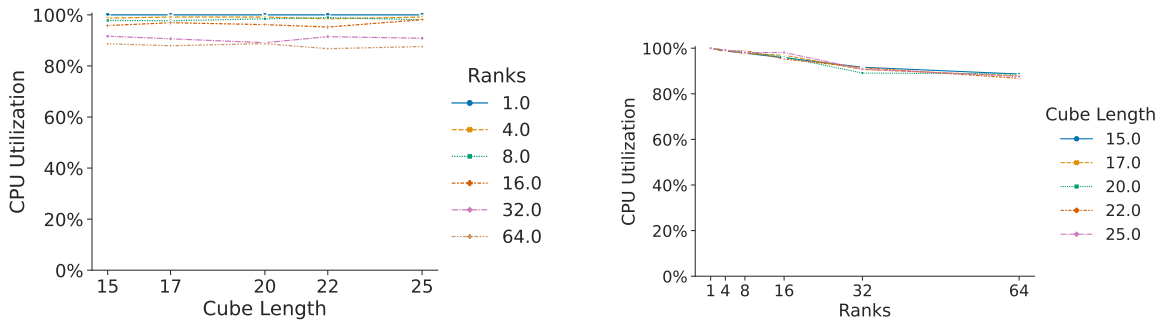
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.16.: FLOPS per Rank for `benchmark-sc2019:void apply_perturbation(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

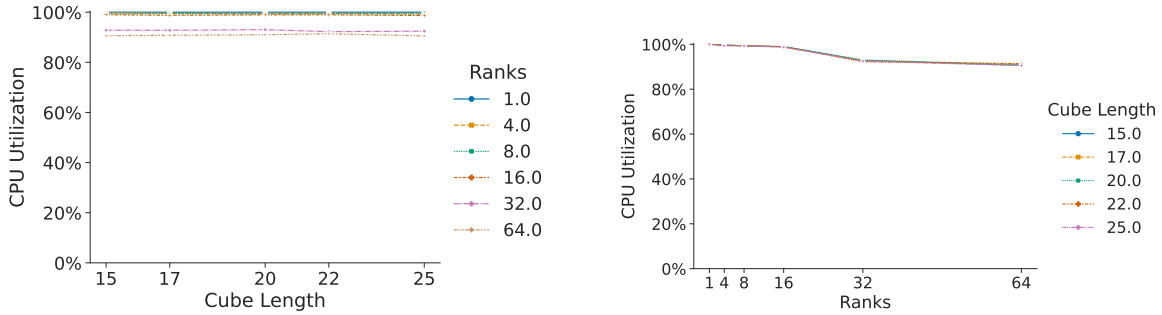
Figure A.17.: CPU Utilization for `benchmark-sc2019:virtual void get_stress(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

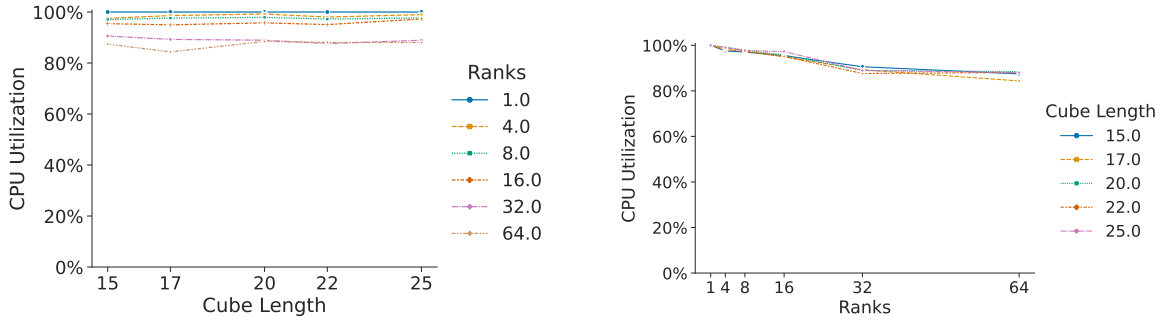
Figure A.18.: CPU Utilization for `benchmark-sc2019:bool damage_law(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

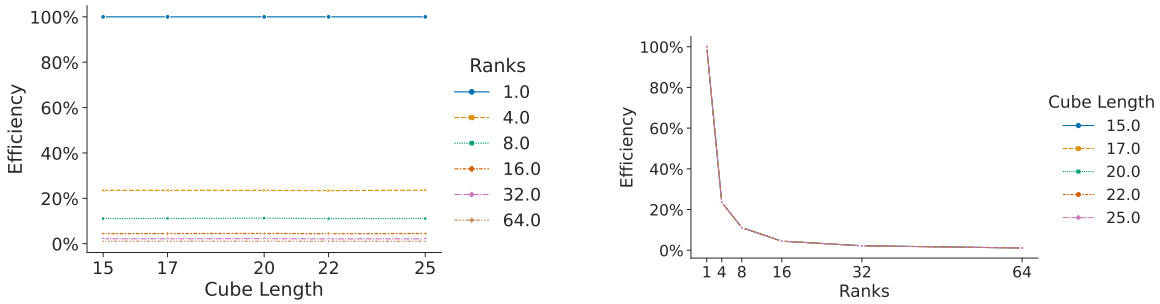
Figure A.19.: CPU Utilization for `benchmark-sc2019:void get_elem_mat(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

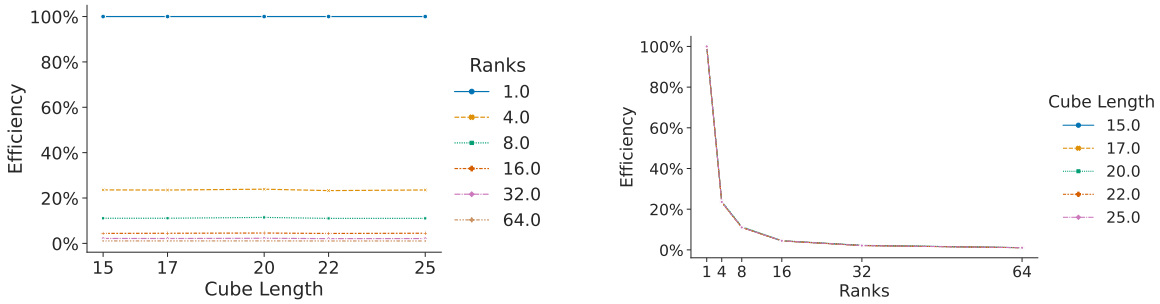
Figure A.20.: CPU Utilization for `benchmark-sc2019:void apply_perturbation(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

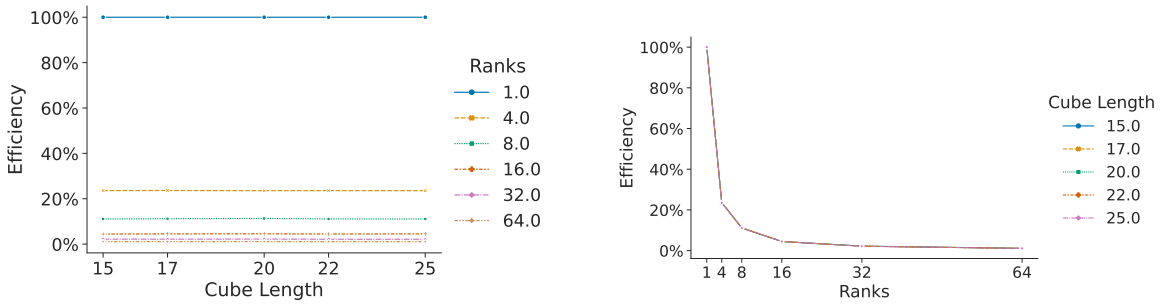
Figure A.21.: Efficiency for `benchmark-sc2019:virtual void get_stress(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

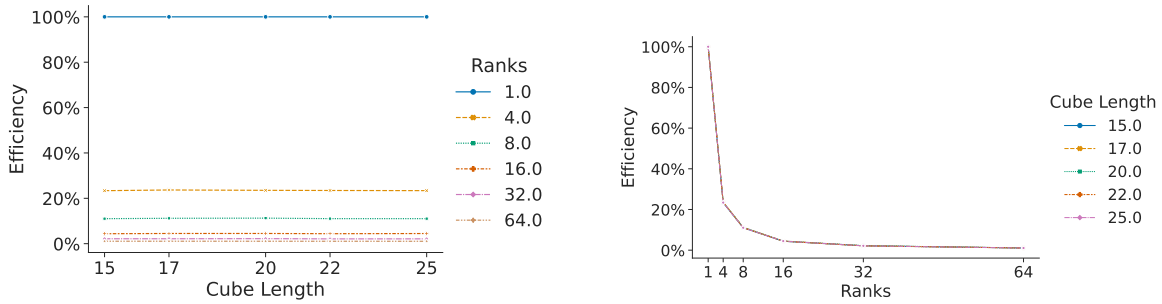
Figure A.22.: Efficiency for `benchmark-sc2019:bool damage_law(...)`



(a) Variable problem size

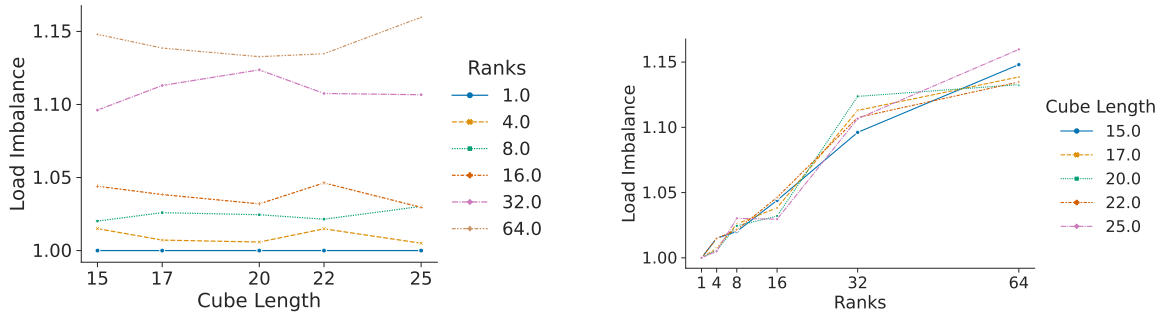
(b) Variable number of MPI ranks

Figure A.23.: Efficiency for `benchmark-sc2019:void get_elem_mat(...)`



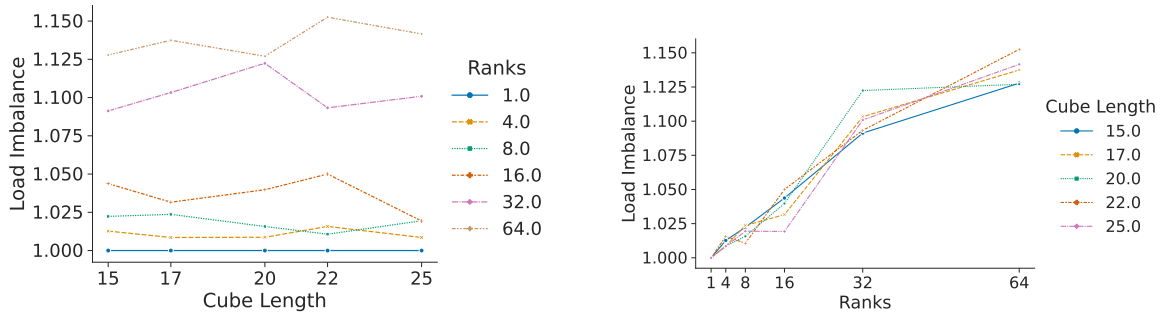
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.24.: Efficiency for `benchmark-sc2019:void apply_perturbation(...)`



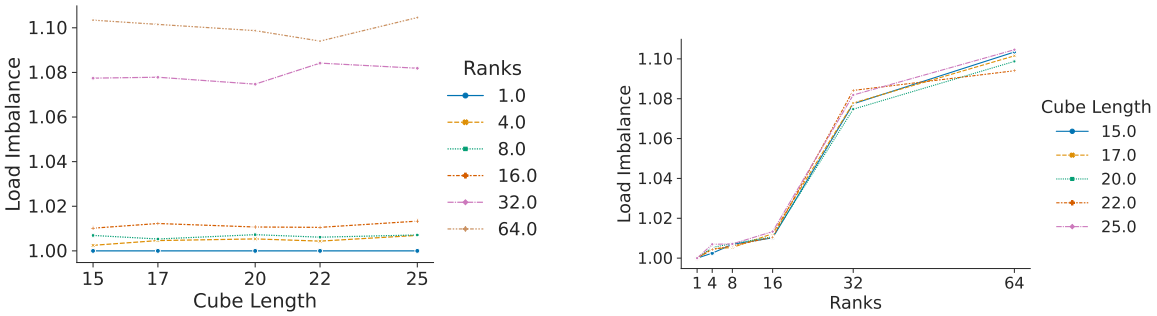
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.25.: Load Imbalance for `benchmark-sc2019:virtual void get_stress(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

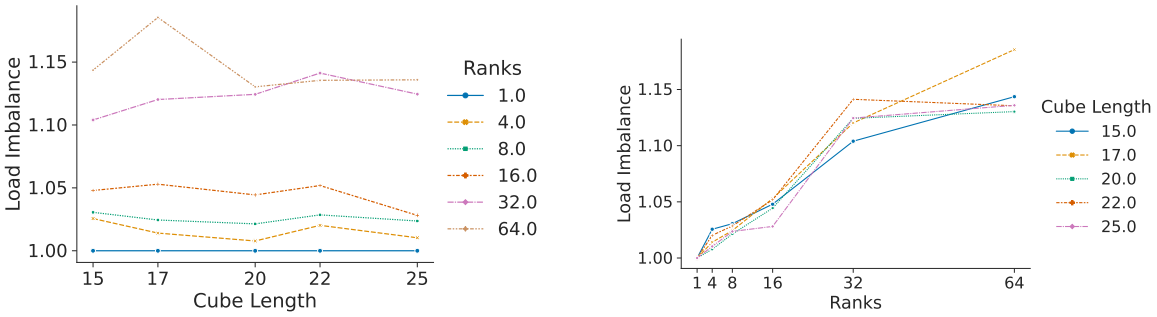
Figure A.26.: Load Imbalance for `benchmark-sc2019:bool damage_law(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

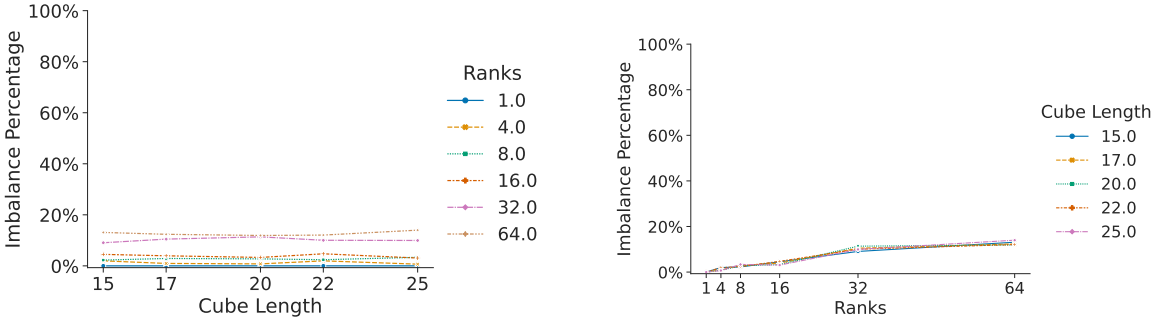
Figure A.27.: Load Imbalance for `benchmark-sc2019:void get_elem_mat(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

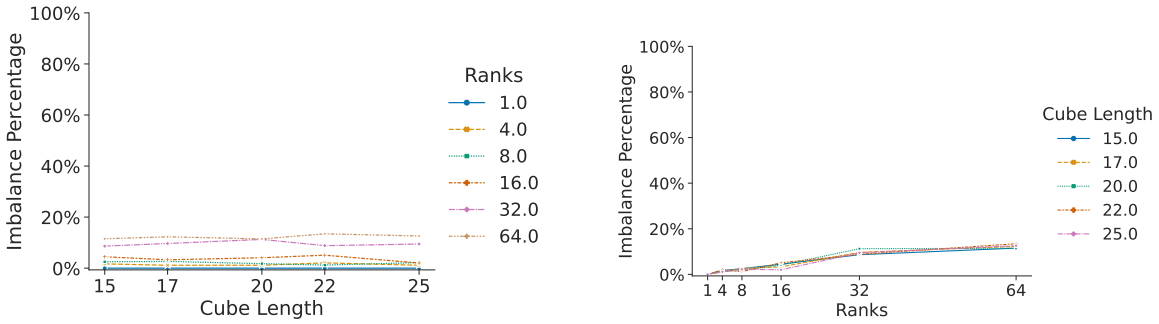
Figure A.28.: Load Imbalance for `benchmark-sc2019:void apply_perturbation(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

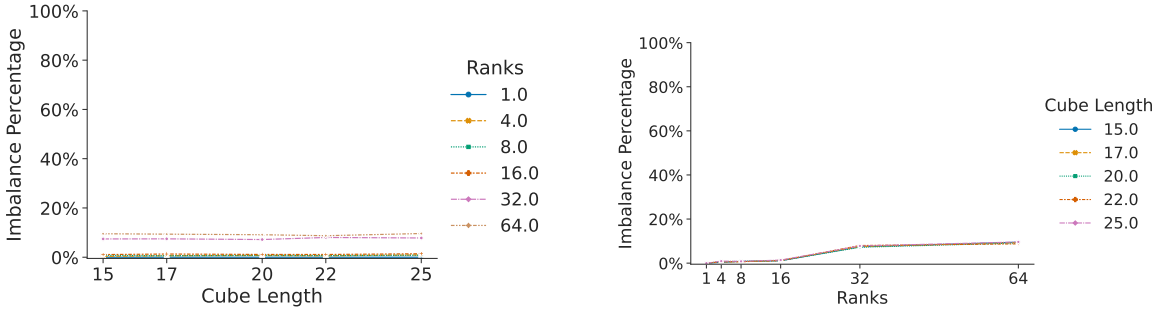
Figure A.29.: Imbalance Percentage for `benchmark-sc2019:virtual void get_stress(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

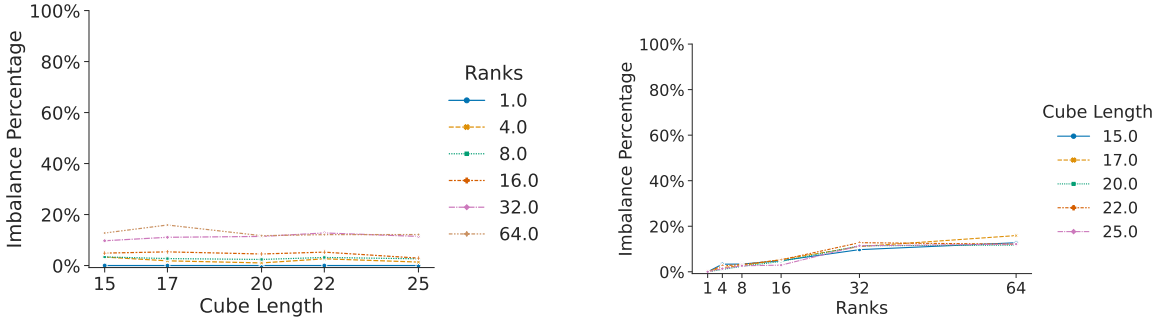
Figure A.30.: Imbalance Percentage for `benchmark-sc2019:bool damage_law(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

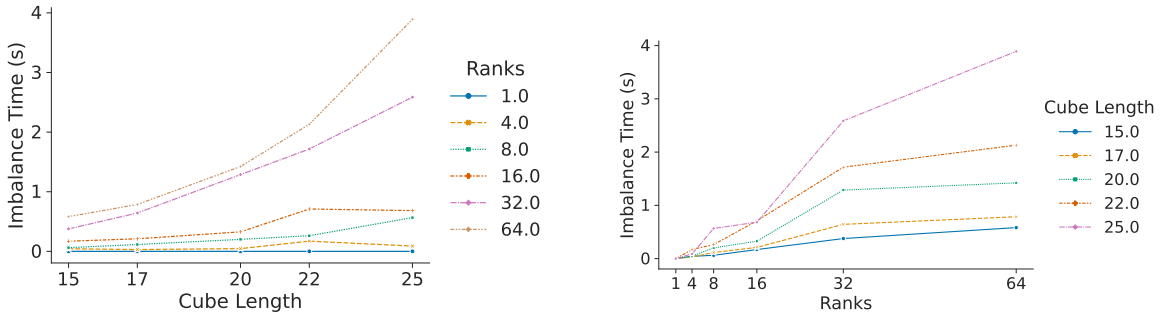
Figure A.31.: Imbalance Percentage for `benchmark-sc2019:void get_elem_mat(...)`



(a) Variable problem size

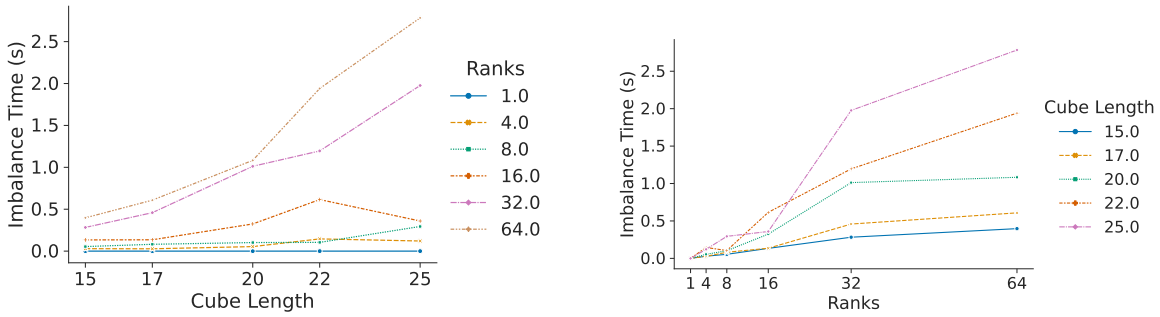
(b) Variable number of MPI ranks

Figure A.32.: Imbalance Percentage for `benchmark-sc2019:void apply_perturbation(...)`



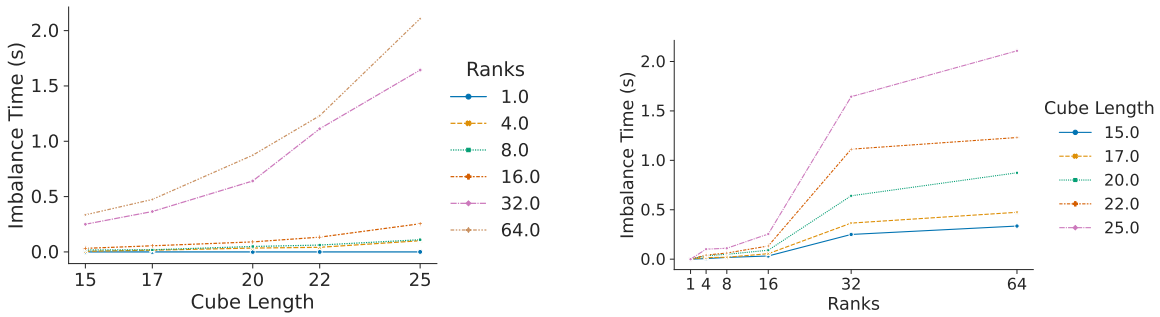
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.33.: Imbalance Time for `benchmark-sc2019:virtual void get_stress(...)`



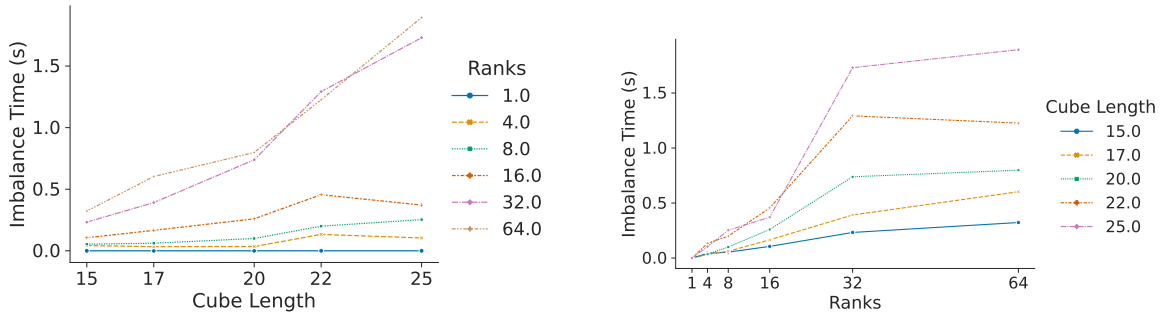
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.34.: Imbalance Time for `benchmark-sc2019:bool damage_law(...)`



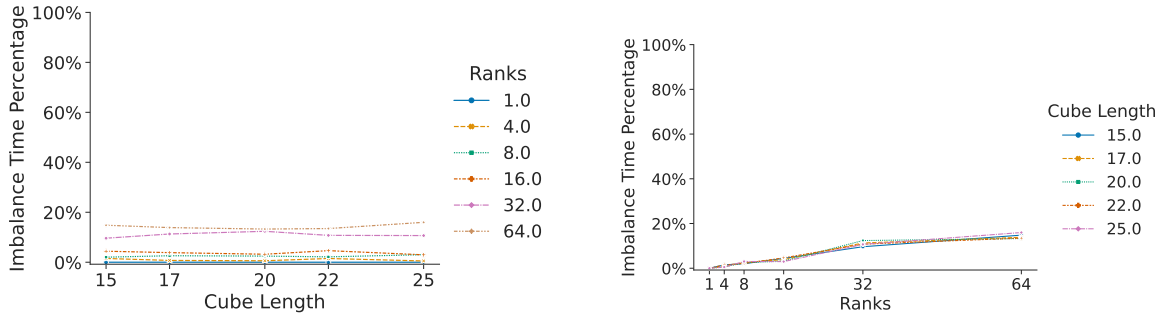
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.35.: Imbalance Time for `benchmark-sc2019:void get_elem_mat(...)`



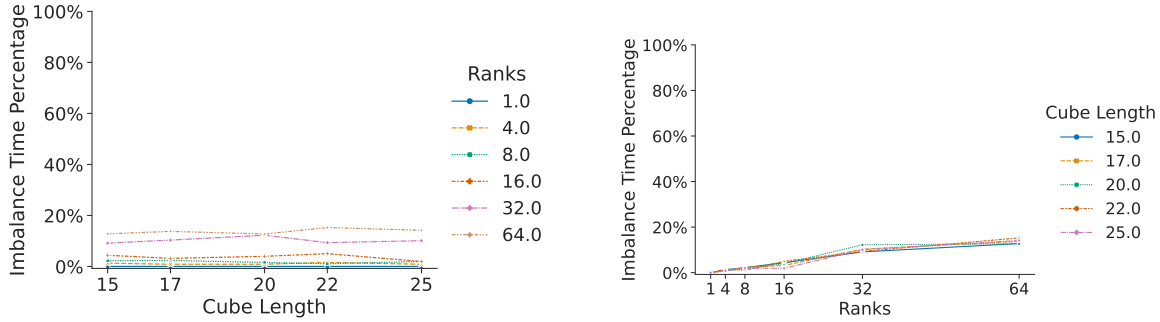
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.36.: Imbalance Time for `benchmark-sc2019:void apply_perturbation(...)`



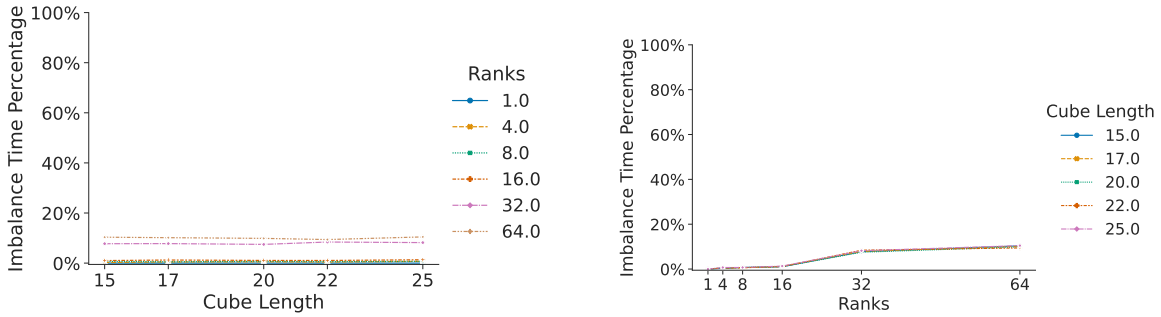
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.37.: Imbalance Time Percentage for `benchmark-sc2019:virtual void get_stress(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

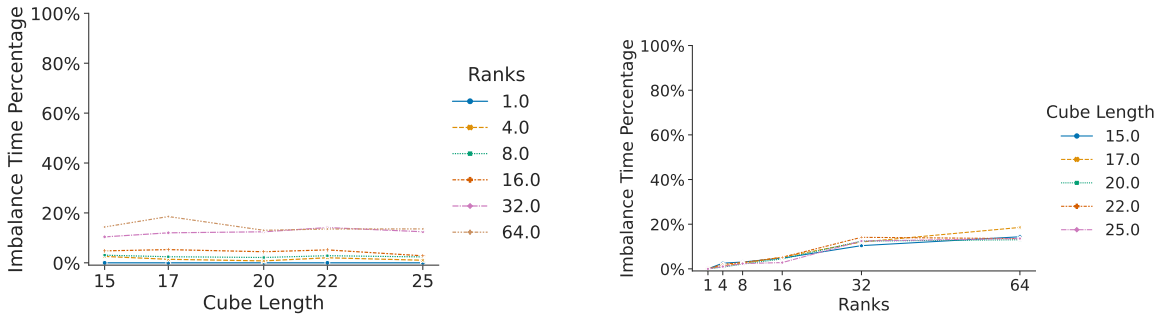
Figure A.38.: Imbalance Time Percentage for `benchmark-sc2019:bool damage_law(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.39.: Imbalance Time Percentage for `benchmark-sc2019:void get_elem_mat(...)`



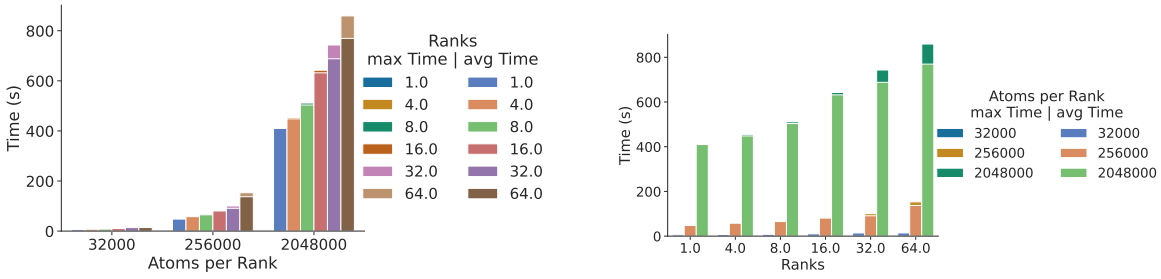
(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.40.: Imbalance Time Percentage for `benchmark-sc2019:void apply_perturbation(...)`

A.2. LAMMPS

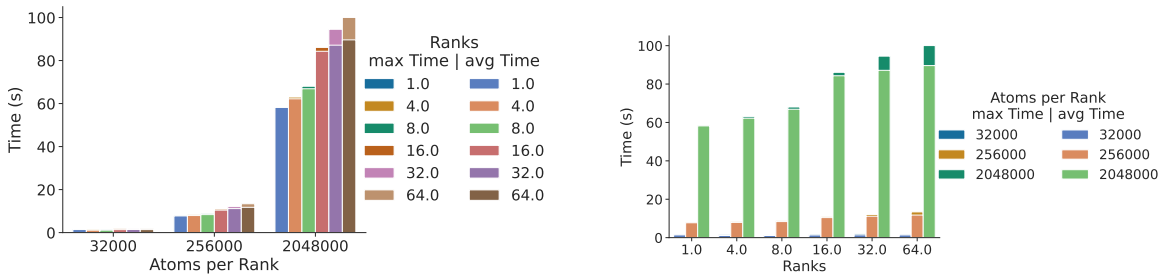
A.2.1. Lennard Jones (LJ) Benchmark



(a) Variable problem size

(b) Variable number of MPI ranks

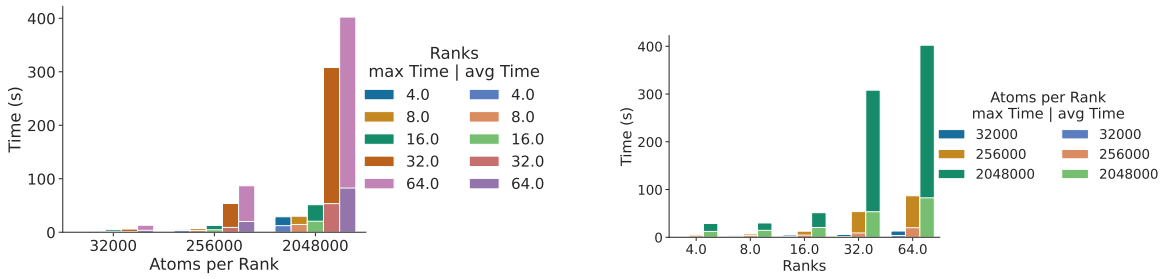
Figure A.41.: Time for `LJ:void build(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

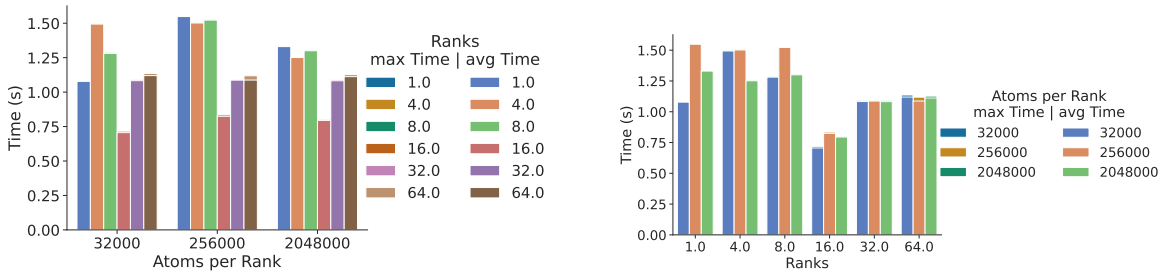
Figure A.42.: Time for `LJ: void ev_tally(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

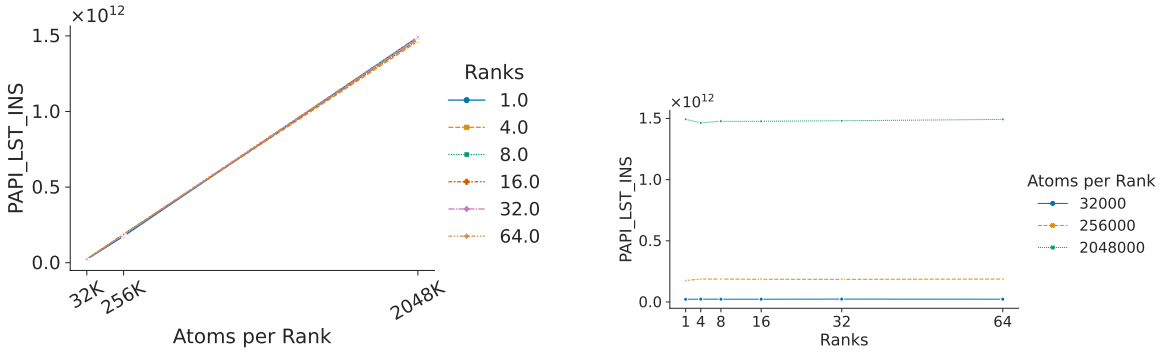
Figure A.43.: Time for `LJ: MPI_Send`



(a) Variable problem size

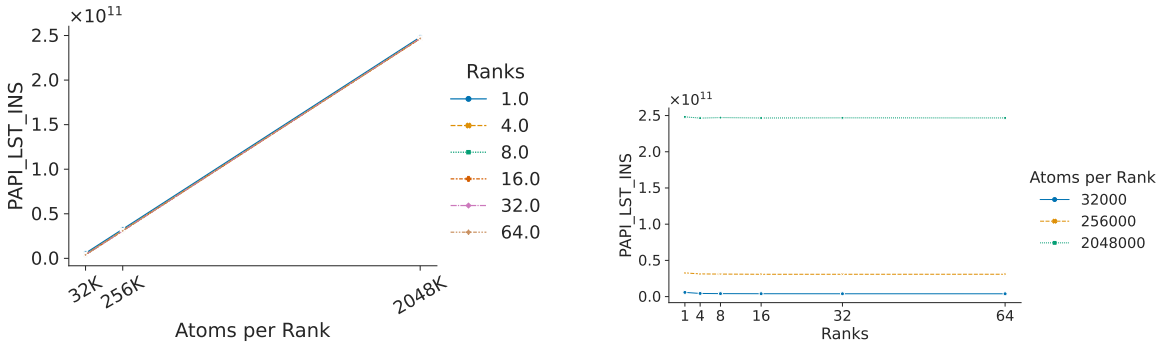
(b) Variable number of MPI ranks

Figure A.44.: Time for `LJ: MPI_Init`



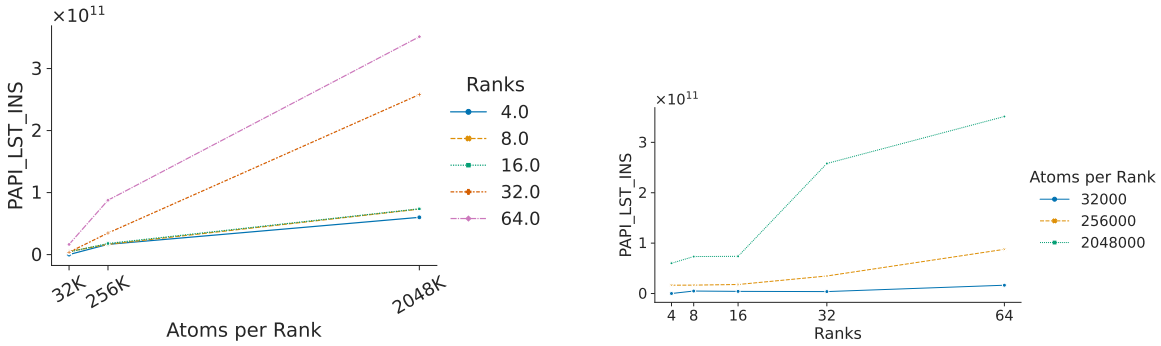
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.45.: PAPI_LST_INS for `LJ:void build(...)`



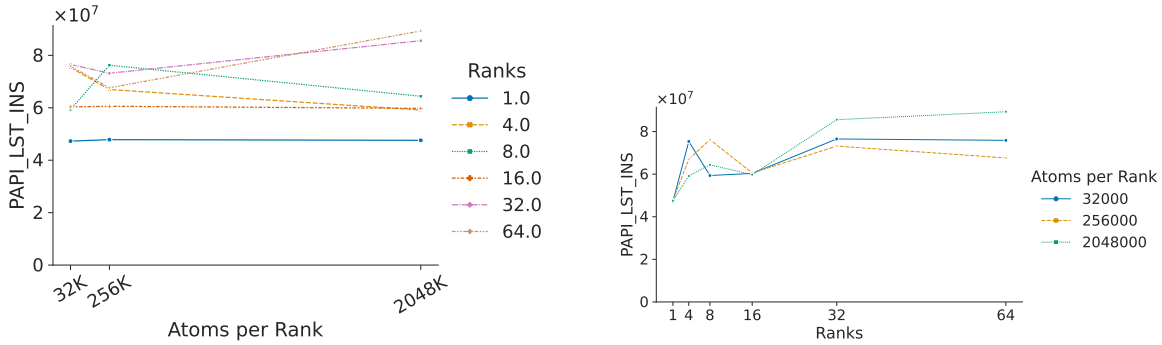
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.46.: PAPI_LST_INS for `LJ: void ev_tally(...)`



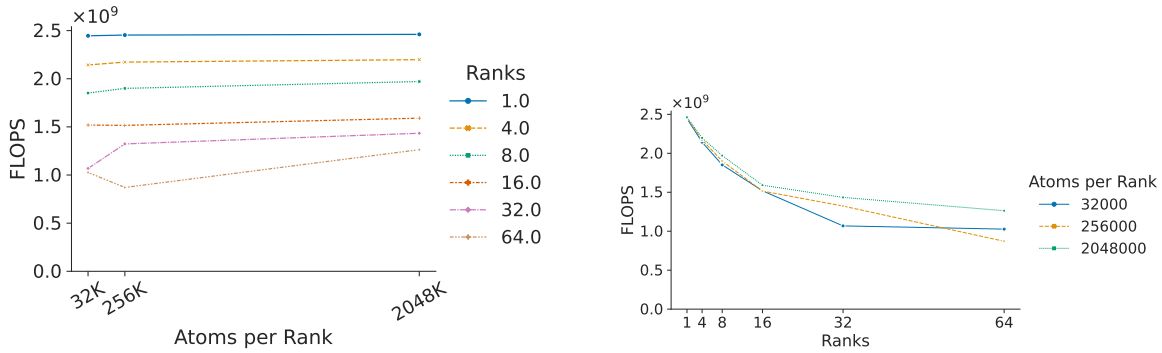
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.47.: PAPI_LST_INS for `LJ: MPI_Send`



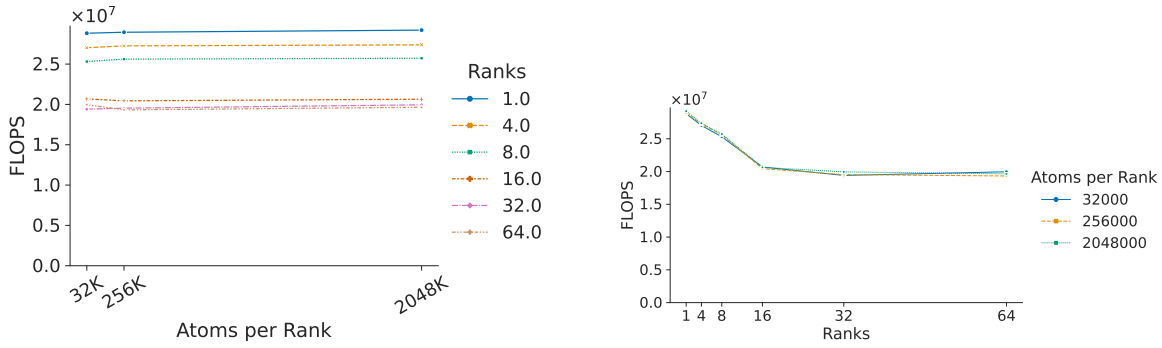
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.48.: PAPI_LST_INS for `LJ: MPI_Init`



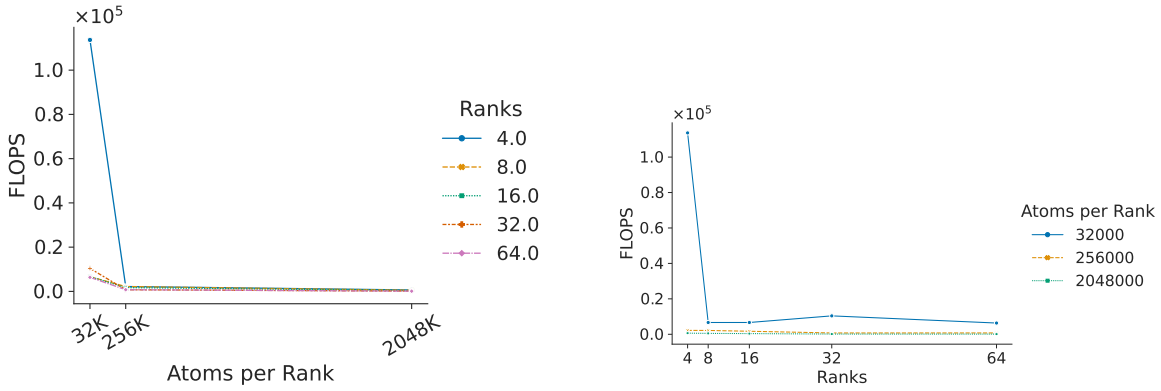
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.49.: FLOPS for `LJ:void build(...)`



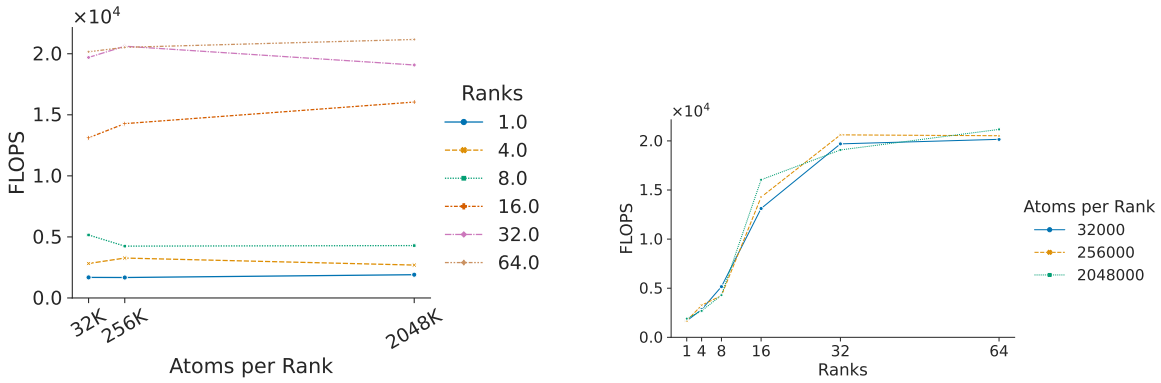
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.50.: FLOPS for `LJ: void ev_tally(...)`



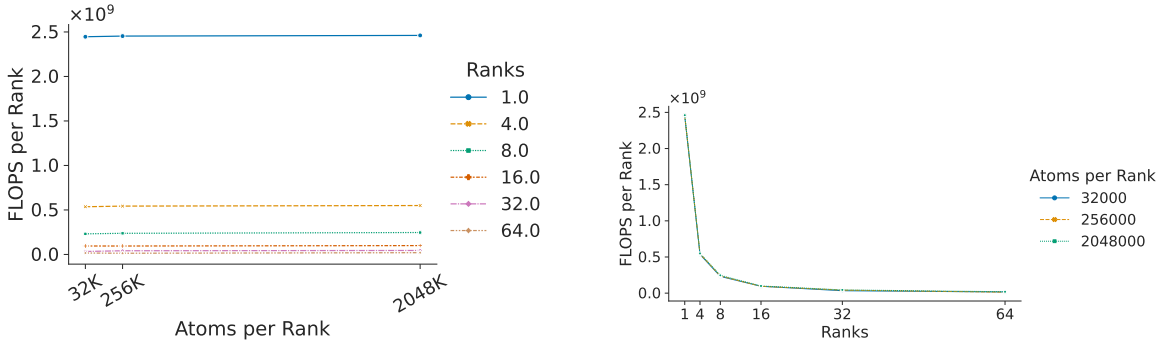
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.51.: FLOPS for `LJ: MPI_Send`



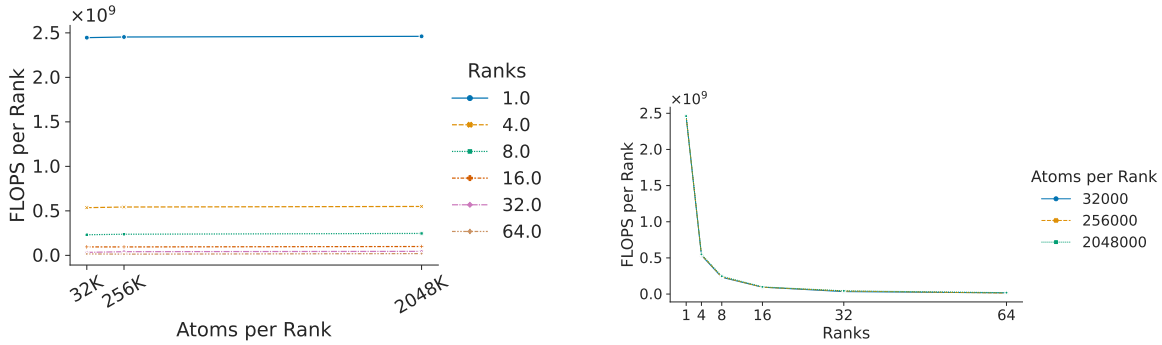
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.52.: FLOPS for `LJ: MPI_Init`



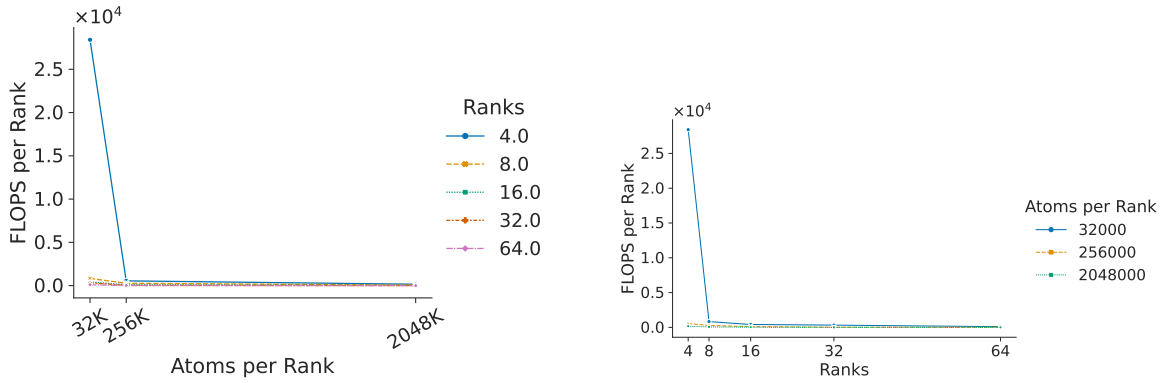
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.53.: FLOPS per Rank for `LJ:void build(...)`



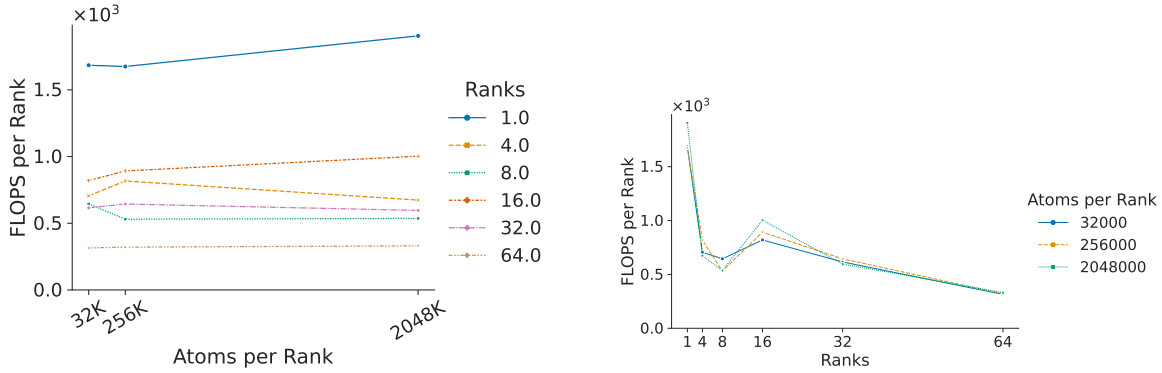
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.54.: FLOPS per Rank for `LJ: void ev_tally(...)`



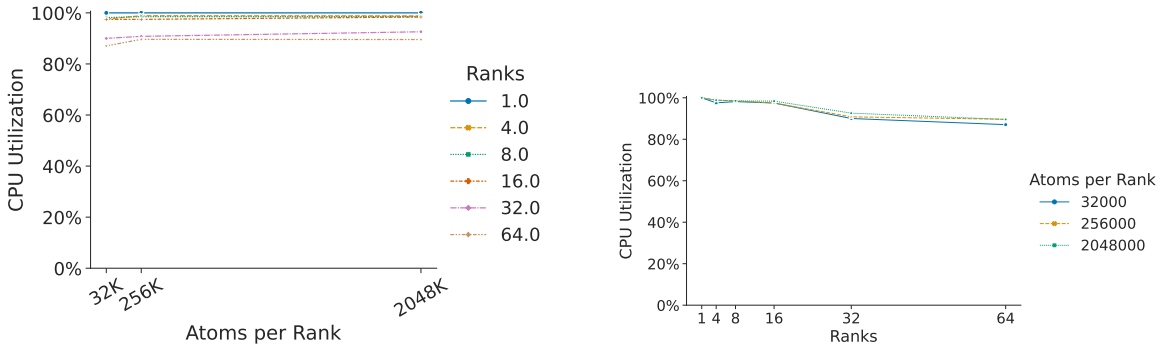
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.55.: FLOPS per Rank for `LJ: MPI_Send`



(a) Variable problem size **(b)** Variable number of MPI ranks

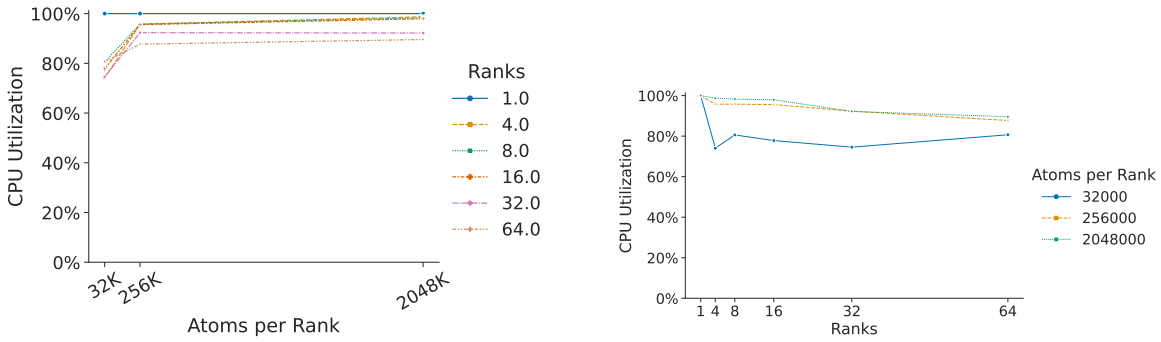
Figure A.56.: FLOPS per Rank for `LJ: MPI_Init`



(a) Variable problem size

(b) Variable number of MPI ranks

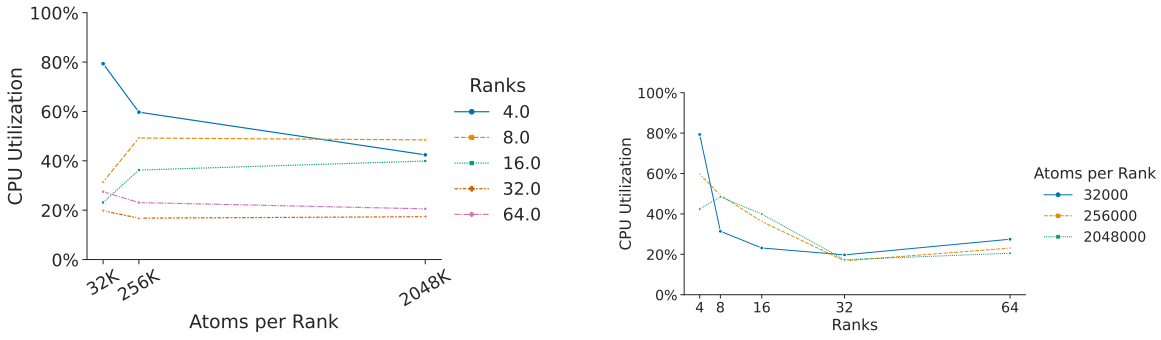
Figure A.57.: CPU Utilization for `LJ:void build(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

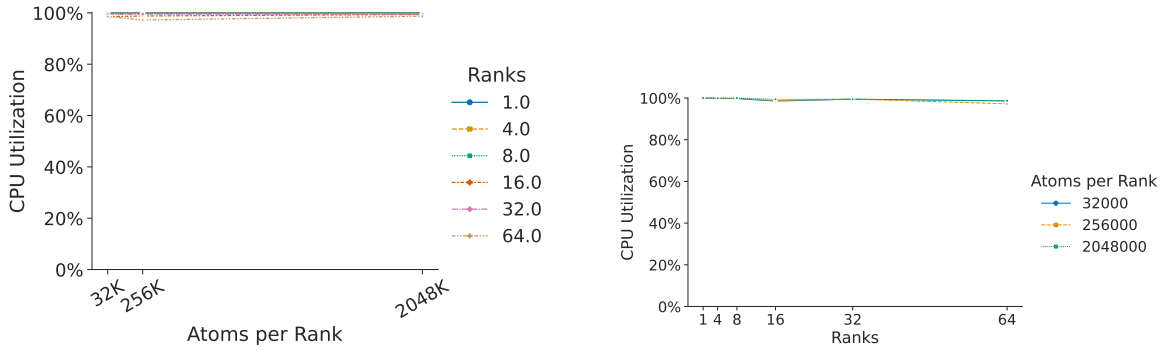
Figure A.58.: CPU Utilization for `LJ: void ev_tally(...)`



(a) Variable problem size

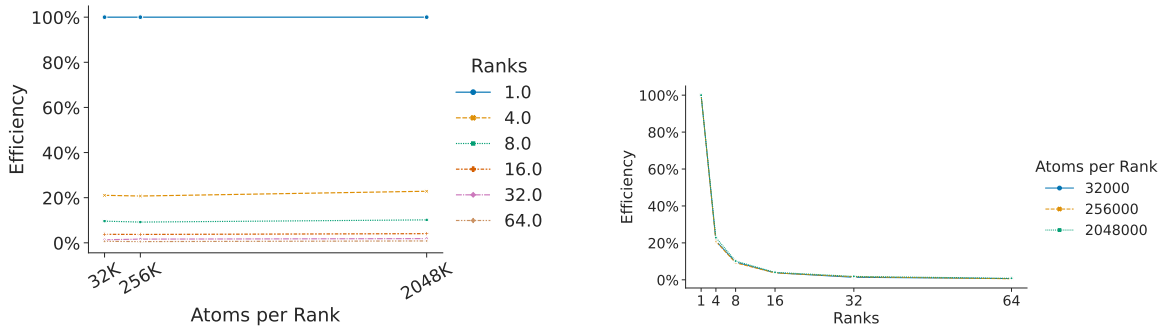
(b) Variable number of MPI ranks

Figure A.59.: CPU Utilization for `LJ: MPI_Send`



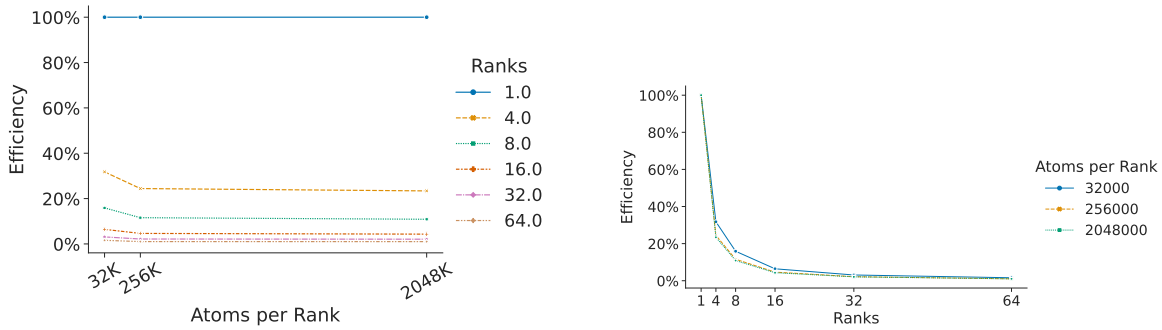
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.60.: CPU Utilization for `LJ: MPI_Init`



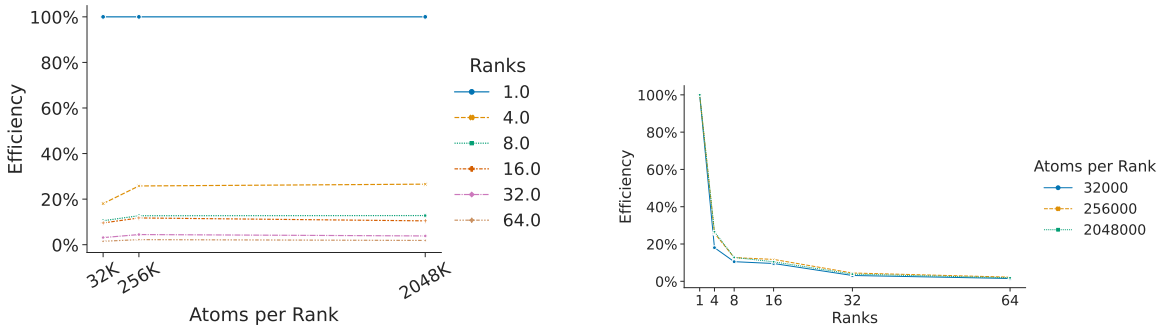
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.61.: Efficiency for `LJ:void build(...)`



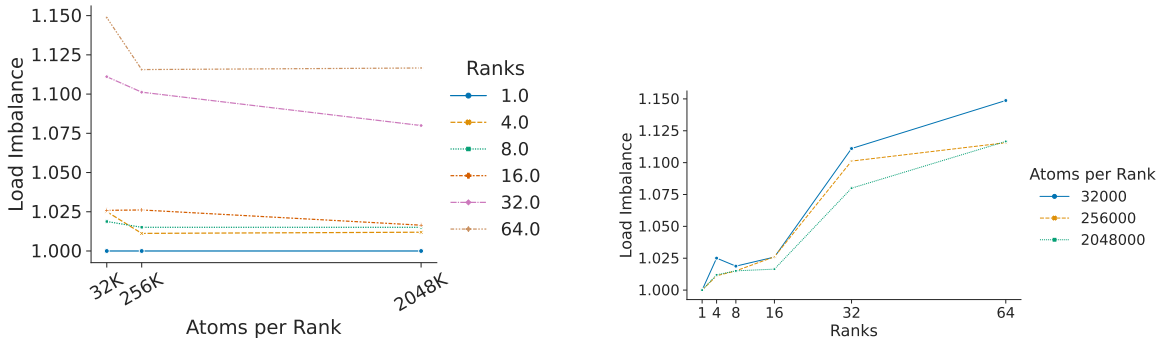
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.62.: Efficiency for `LJ: void ev_tally(...)`



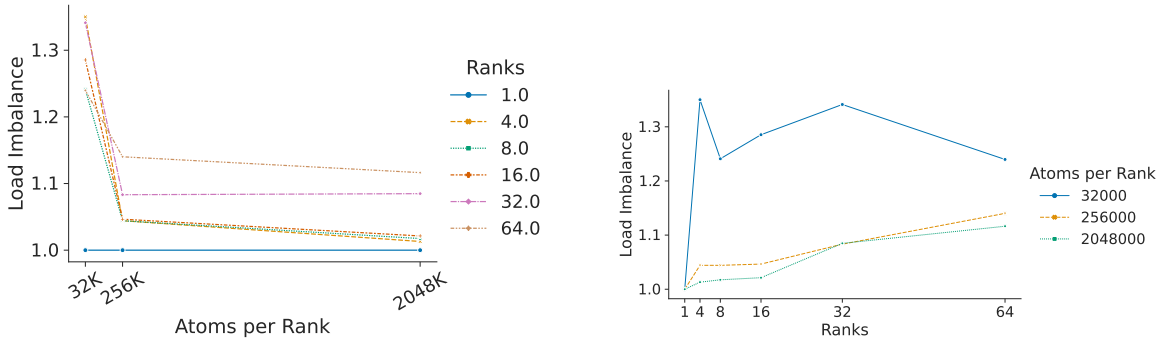
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.63.: Efficiency for `LJ: MPI_Init`



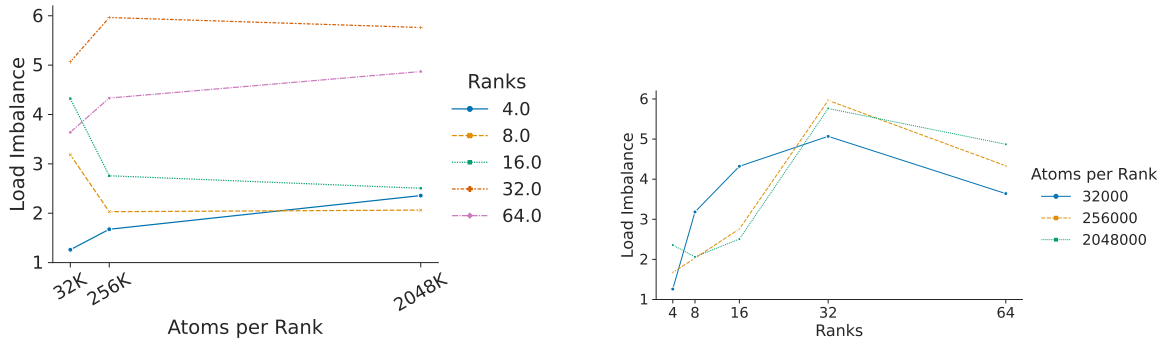
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.64.: Load Imbalance for `LJ:void build(...)`



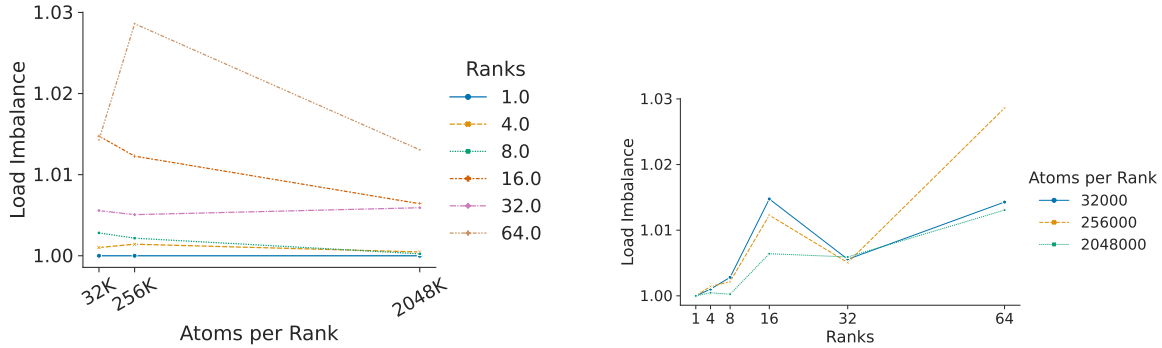
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.65.: Load Imbalance for `LJ: void ev_tally(...)`



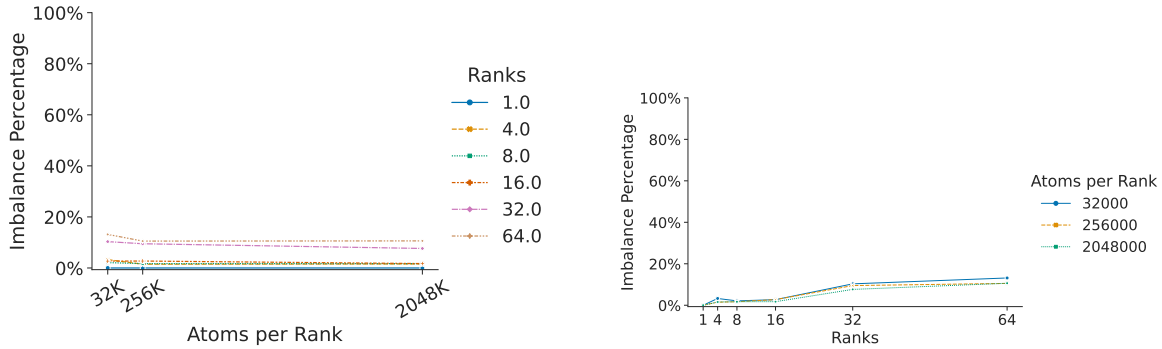
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.66.: Load Imbalance for `LJ: MPI_Send`



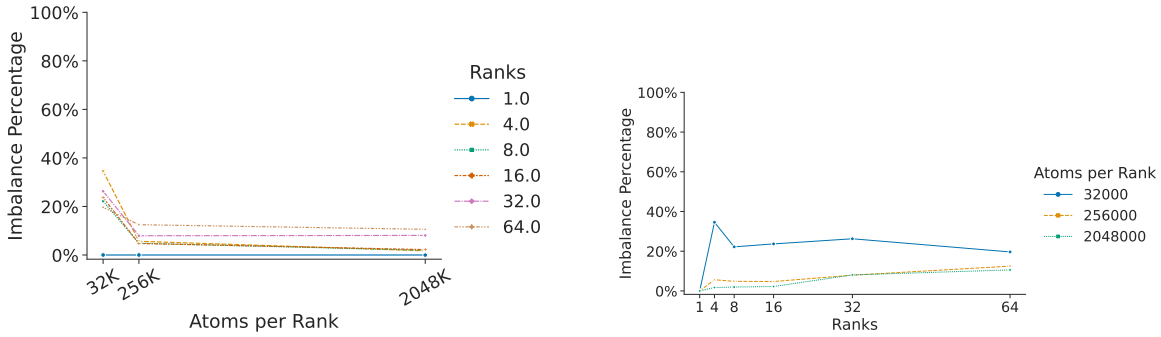
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.67.: Load Imbalance for `LJ: MPI_Init`



(a) Variable problem size **(b)** Variable number of MPI ranks

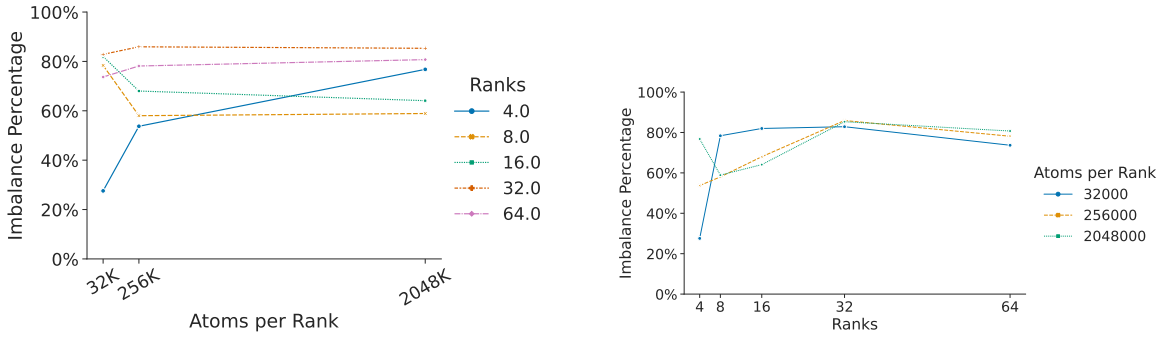
Figure A.68.: Imbalance Percentage for `LJ:void build(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

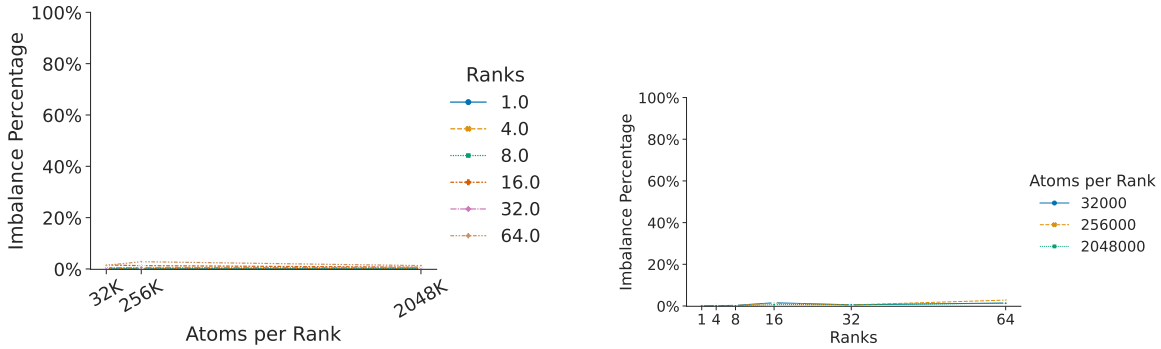
Figure A.69.: Imbalance Percentage for `LJ: void ev_tally(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

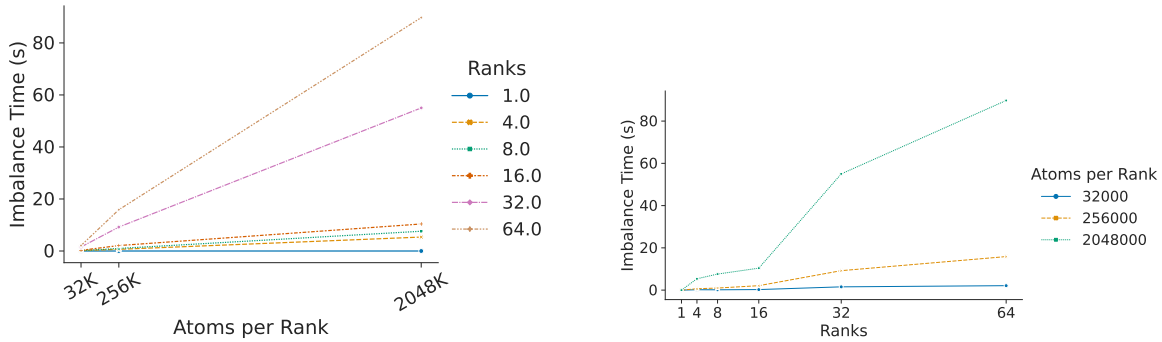
Figure A.70.: Imbalance Percentage for `LJ: MPI_Send`



(a) Variable problem size

(b) Variable number of MPI ranks

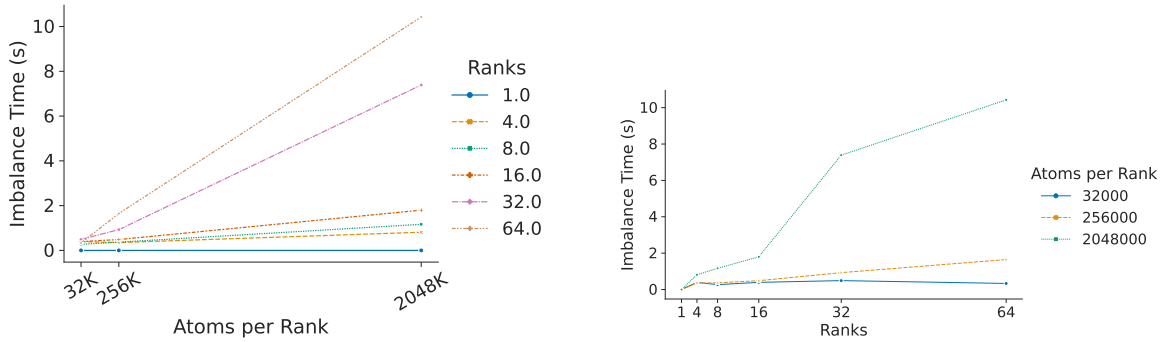
Figure A.71.: Imbalance Percentage for `LJ: MPI_Init`



(a) Variable problem size

(b) Variable number of MPI ranks

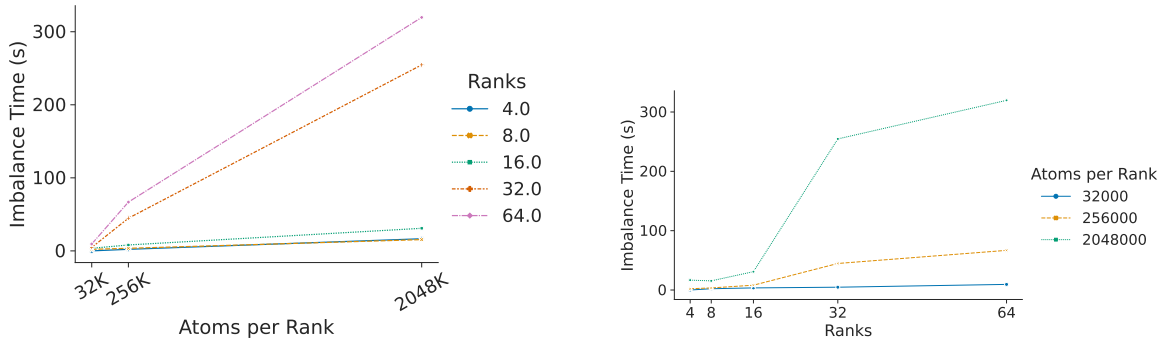
Figure A.72.: Imbalance Time for `LJ:void build(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

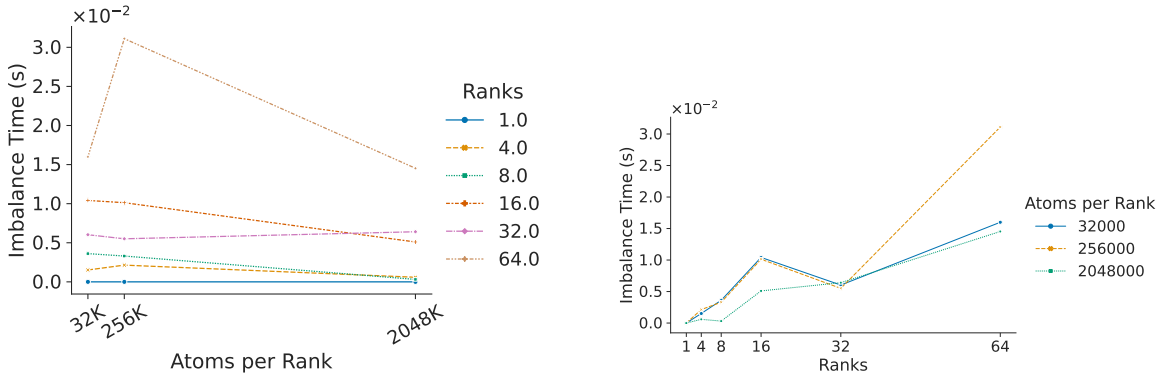
Figure A.73.: Imbalance Time for `LJ: void ev_tally(...)`



(a) Variable problem size

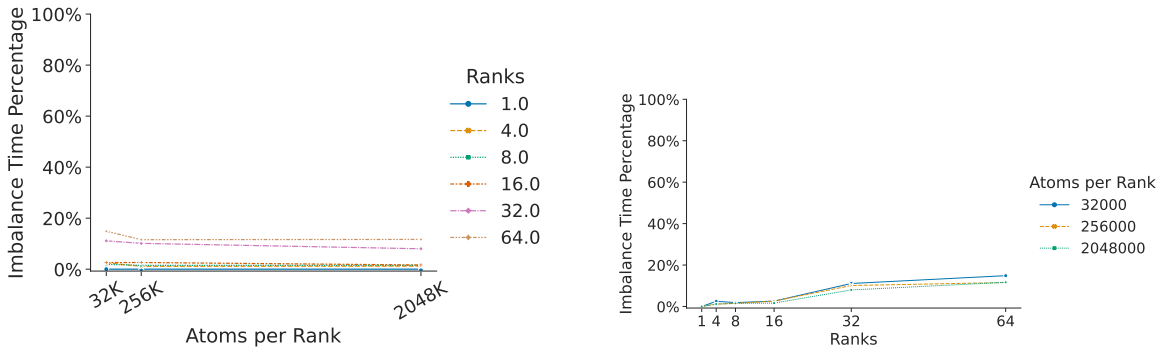
(b) Variable number of MPI ranks

Figure A.74.: Imbalance Time for `LJ: MPI_Send`



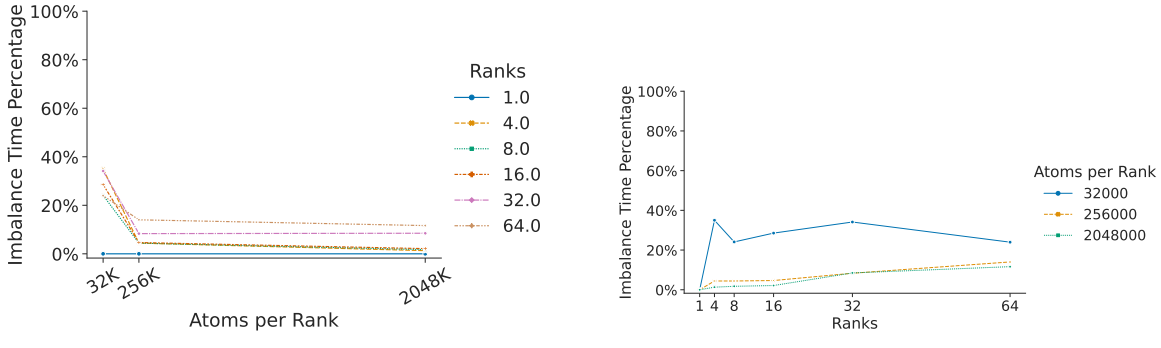
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.75.: Imbalance Time for `LJ: MPI_Init`



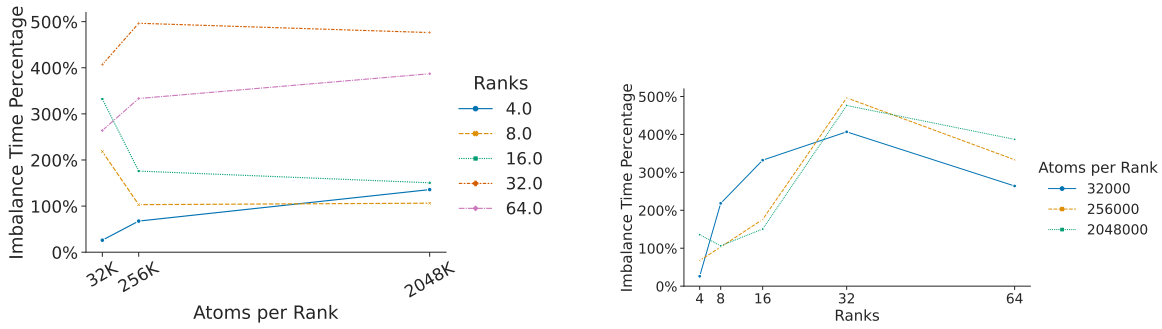
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.76.: Imbalance Time Percentage for `LJ:void build(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

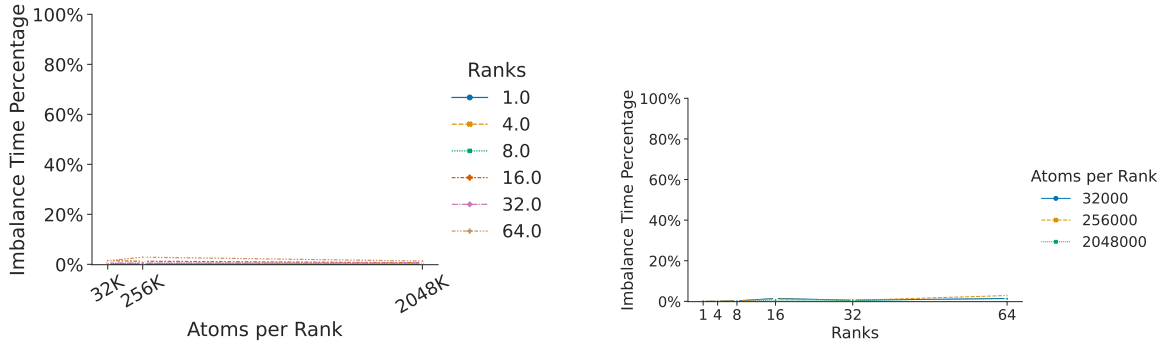
Figure A.77.: Imbalance Time Percentage for `LJ: void ev_tally(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.78.: Imbalance Time Percentage for `LJ: MPI_Send`

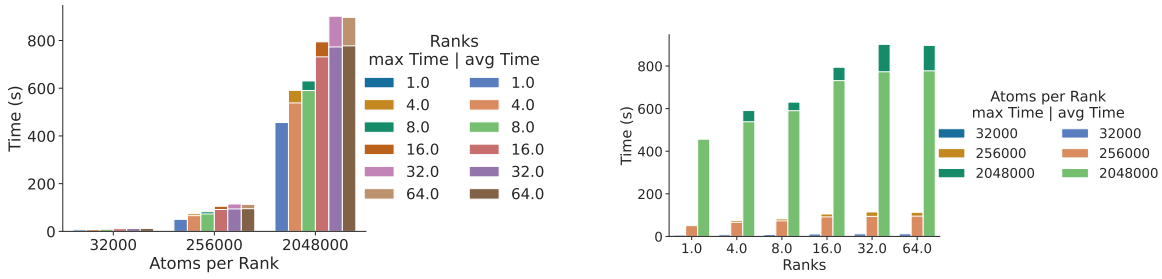


(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.79.: Imbalance Time Percentage for `LJ: MPI_Init`

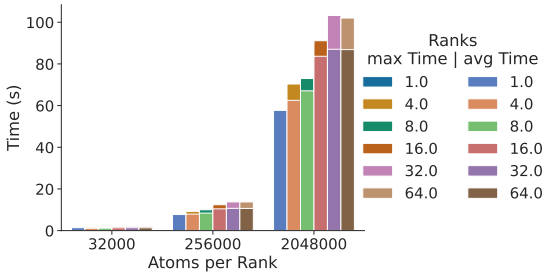
A.2.2. LJ with RCB Load Balancing Benchmark



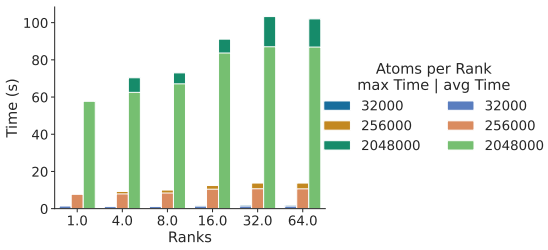
(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.80.: Time for `RCB:void build(...)`

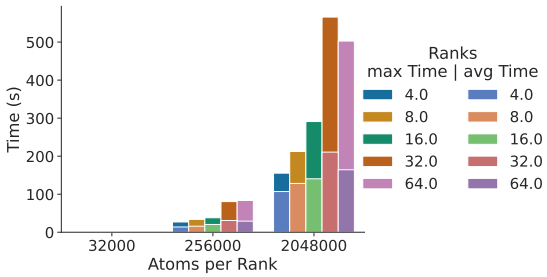


(a) Variable problem size

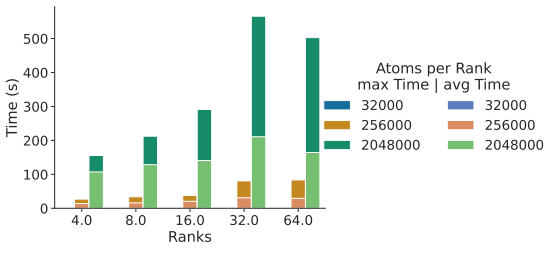


(b) Variable number of MPI ranks

Figure A.81.: Time for `RCB: void ev_tally(...)`

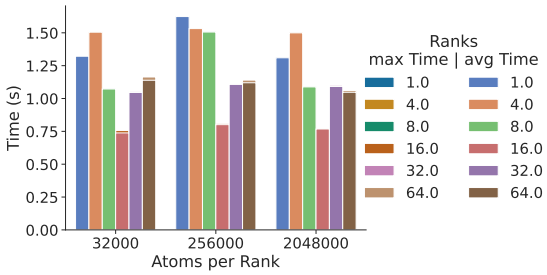


(a) Variable problem size

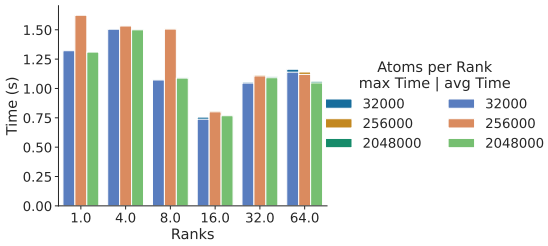


(b) Variable number of MPI ranks

Figure A.82.: Time for `RCB: MPI_Send`

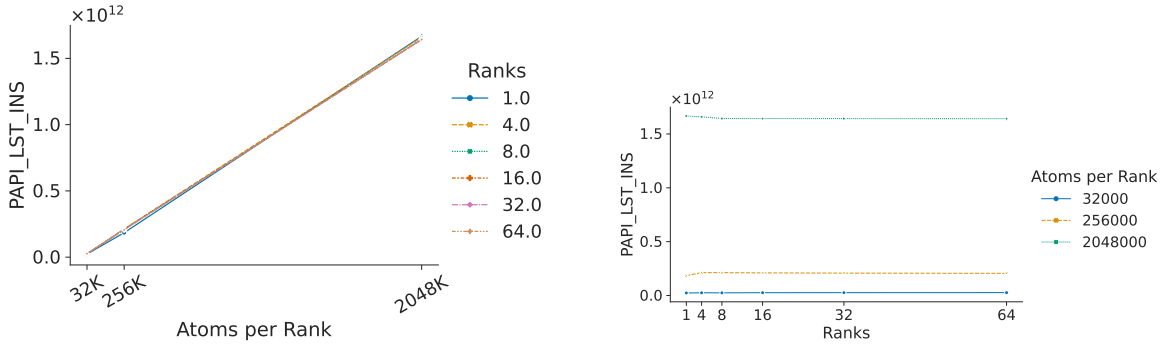


(a) Variable problem size



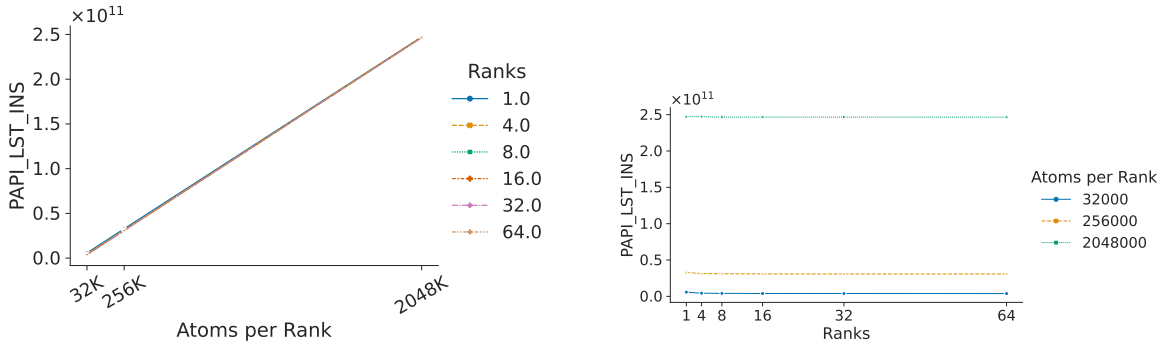
(b) Variable number of MPI ranks

Figure A.83.: Time for `RCB: MPI_Init`



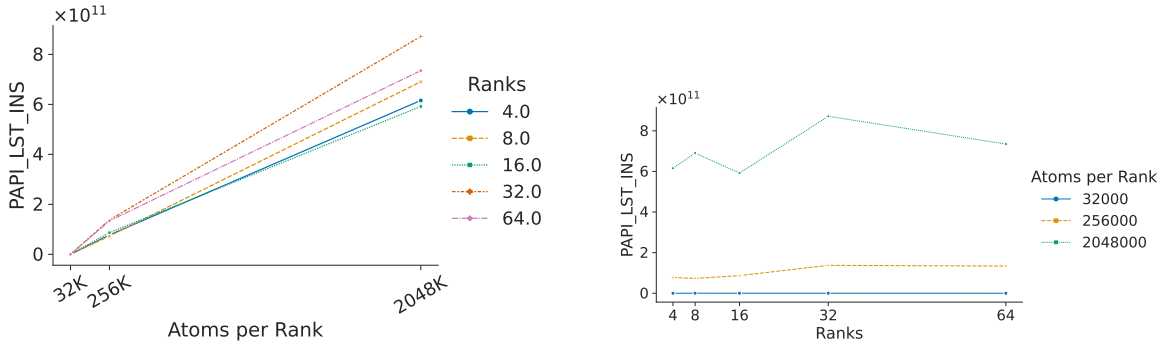
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.84.: PAPI_LST_INS for `RCB:void build(...)`



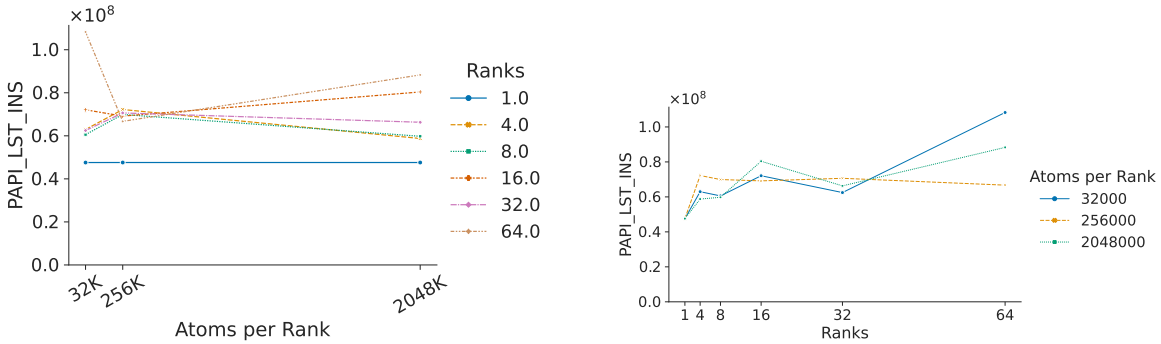
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.85.: PAPI_LST_INS for `RCB: void ev_tally(...)`



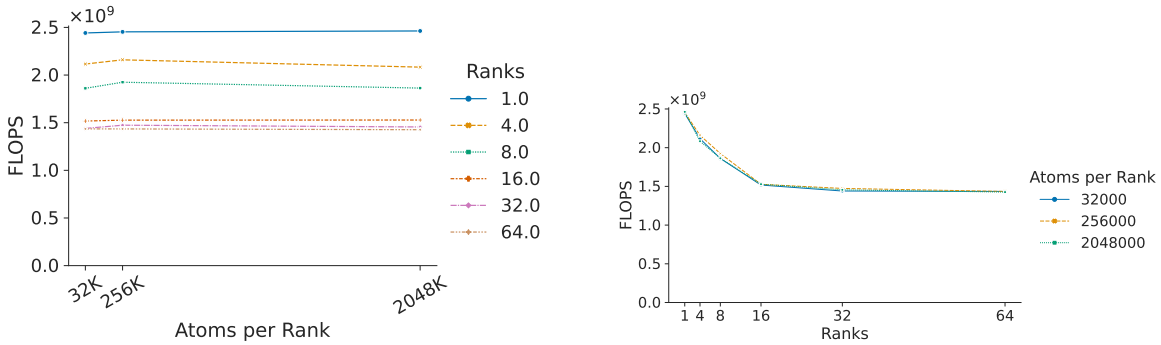
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.86.: PAPI_LST_INS for `RCB: MPI_Send`



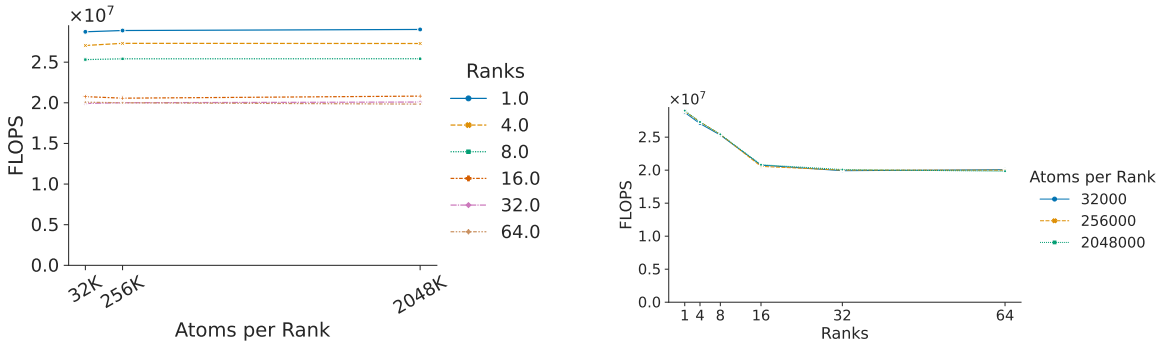
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.87.: PAPI_LST_INS for `RCB: MPI_Init`



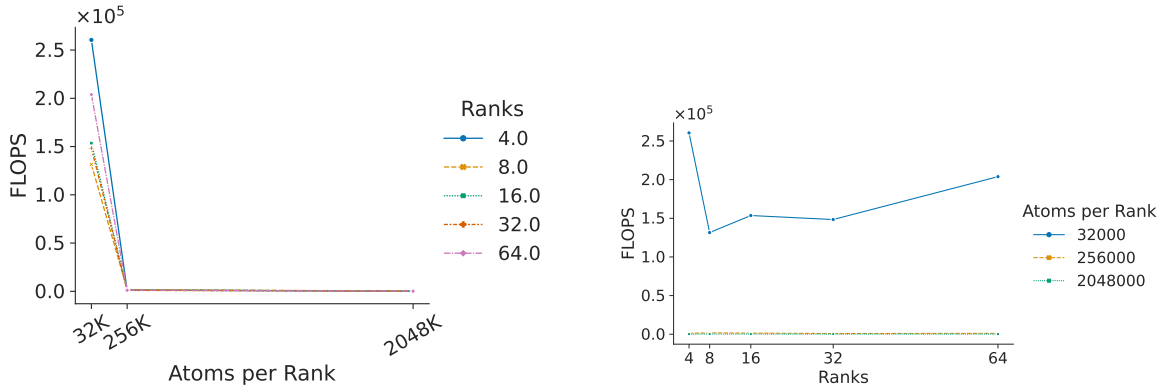
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.88.: FLOPS for `RCB:void build(...)`



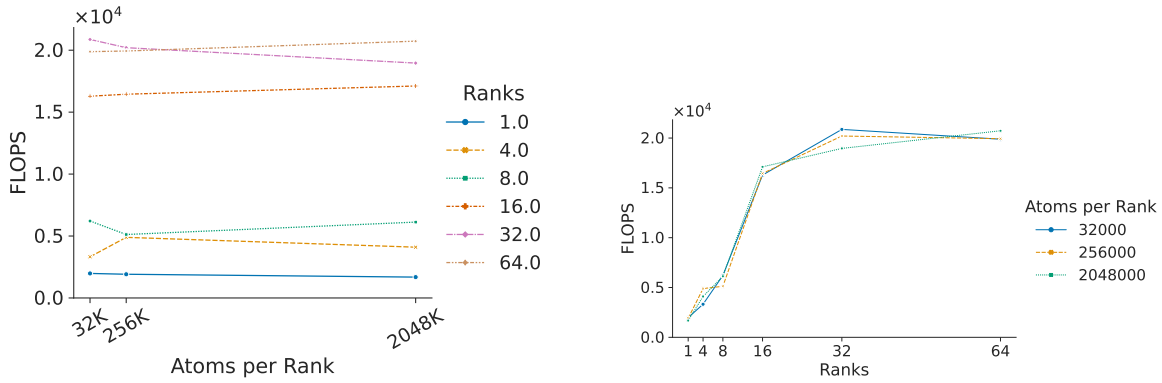
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.89.: FLOPS for `RCB: void ev_tally(...)`



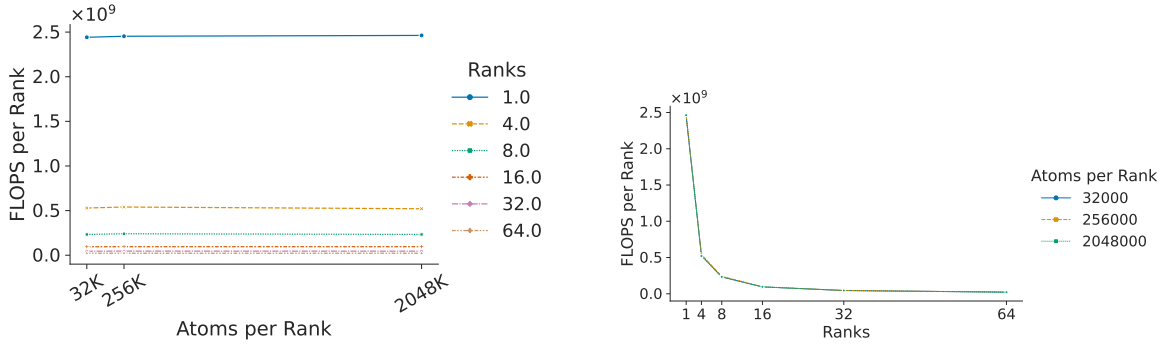
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.90.: FLOPS for `RCB: MPI_Send`



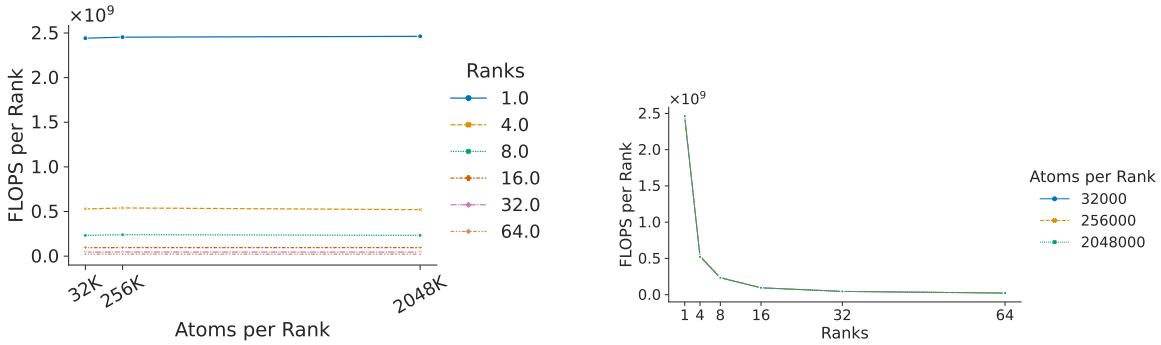
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.91.: FLOPS for `RCB: MPI_Init`



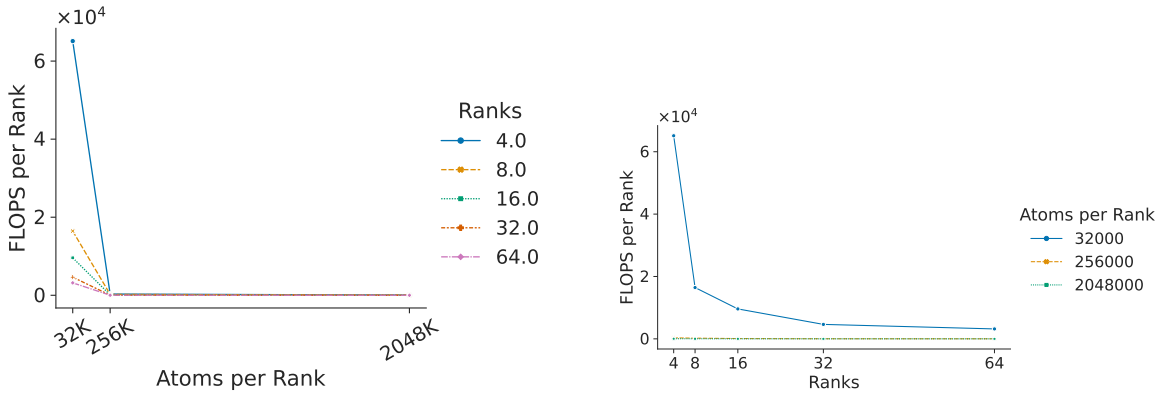
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.92.: FLOPS per Rank for `RCB:void build(...)`



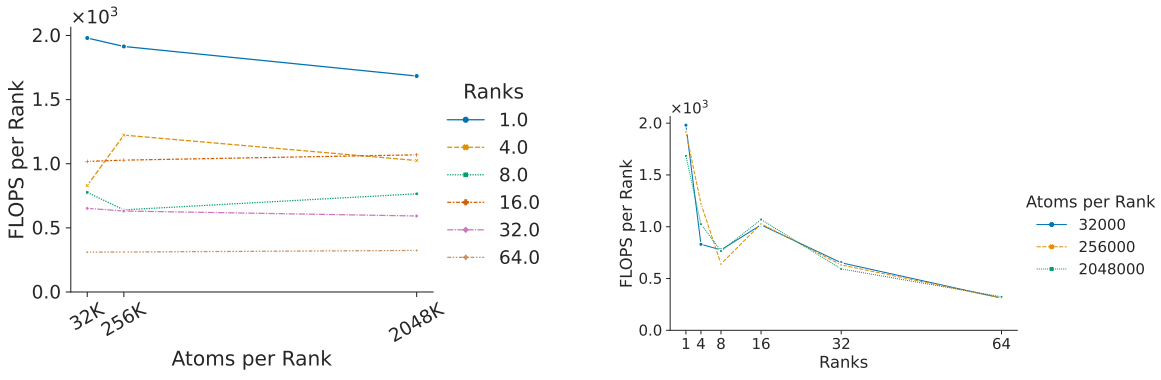
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.93.: FLOPS per Rank for `RCB: void ev_tally(...)`



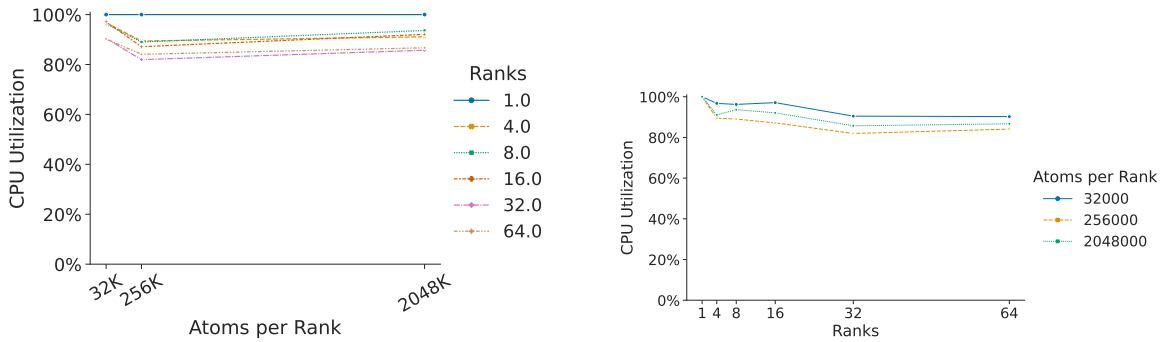
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.94.: FLOPS per Rank for `RCB: MPI_Send`



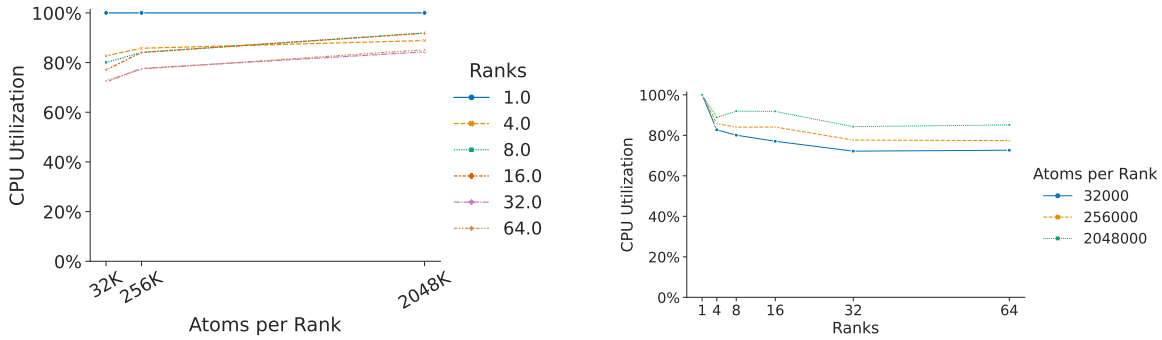
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.95.: FLOPS per Rank for `RCB: MPI_Init`



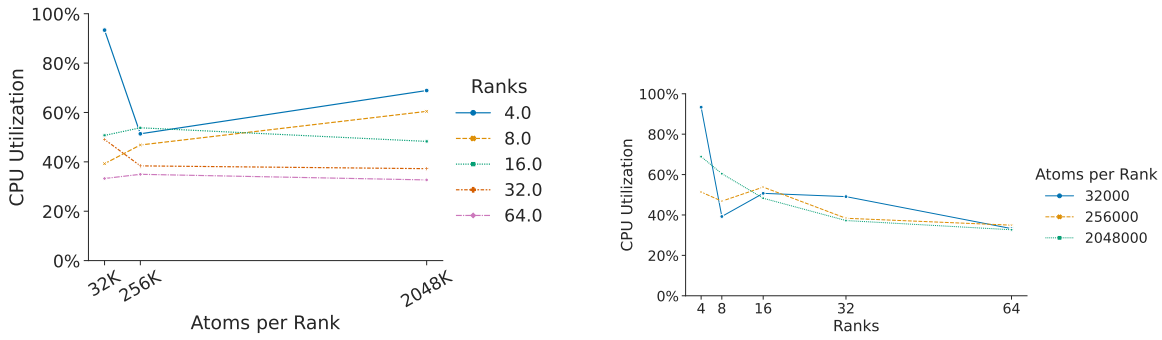
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.96.: CPU Utilization for `RCB:void build(...)`



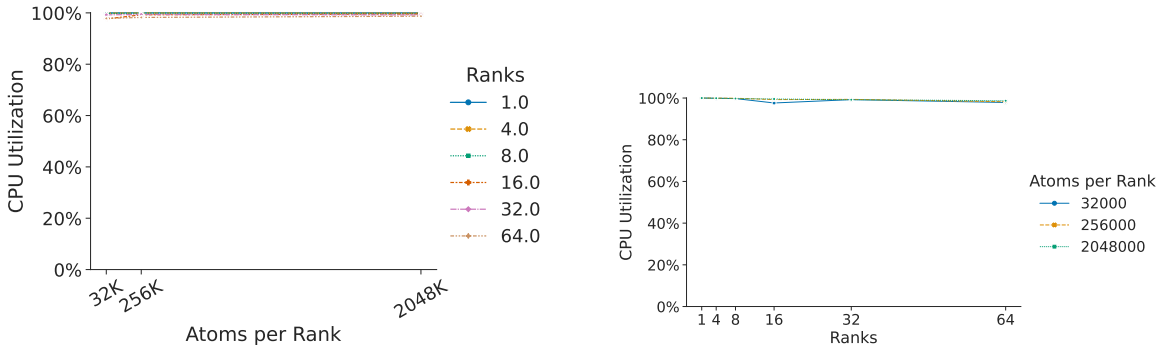
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.97.: CPU Utilization for `RCB: void ev_tally(...)`



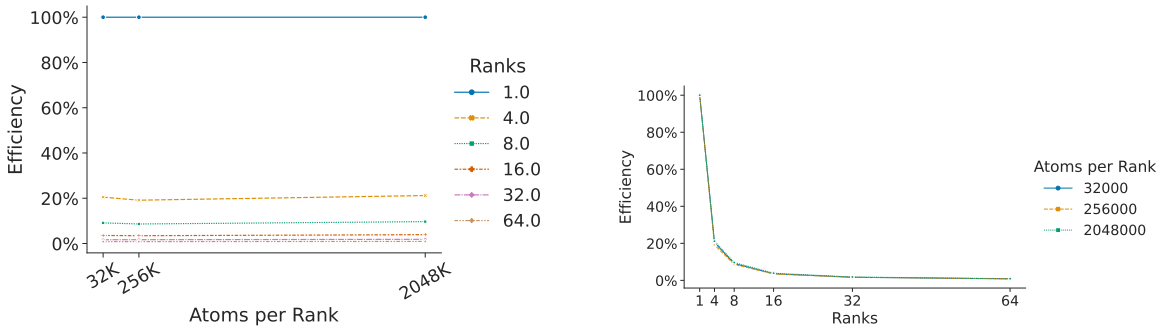
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.98.: CPU Utilization for `RCB: MPI_Send`



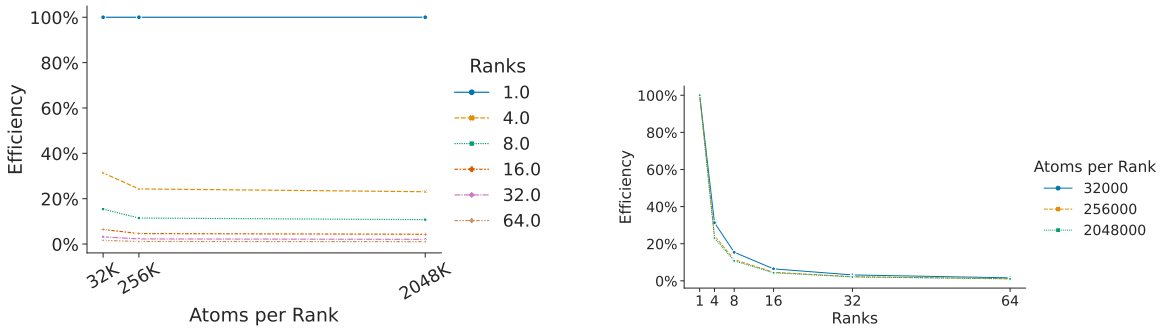
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.99.: CPU Utilization for `RCB: MPI_Init`



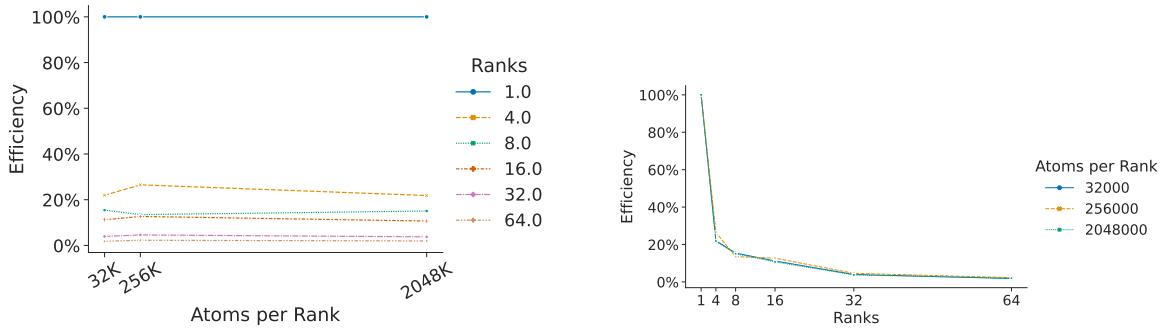
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.100.: Efficiency for `RCB:void build(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

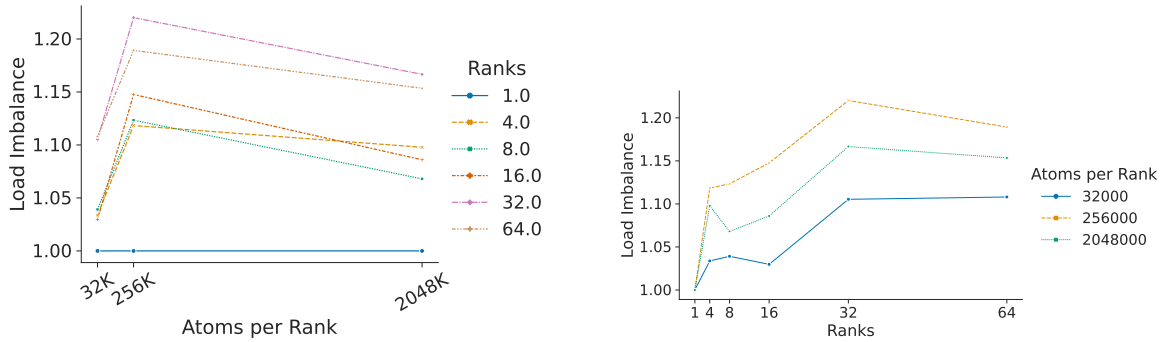
Figure A.101.: Efficiency for `RCB: void ev_tally(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

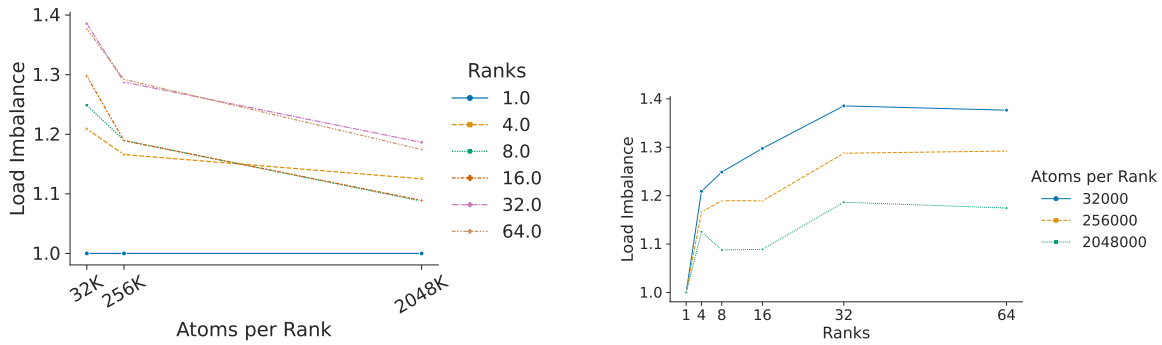
Figure A.102.: Efficiency for `RCB: MPI_Init`



(a) Variable problem size

(b) Variable number of MPI ranks

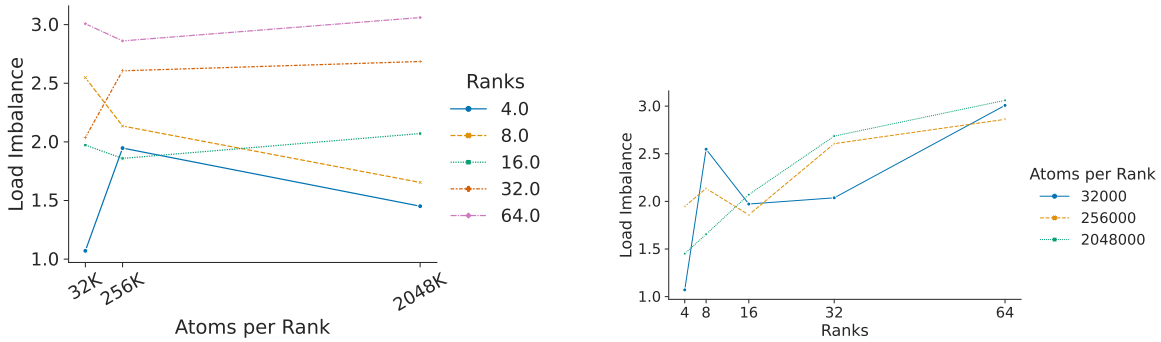
Figure A.103.: Load Imbalance for `RCB:void build(...)`



(a) Variable problem size

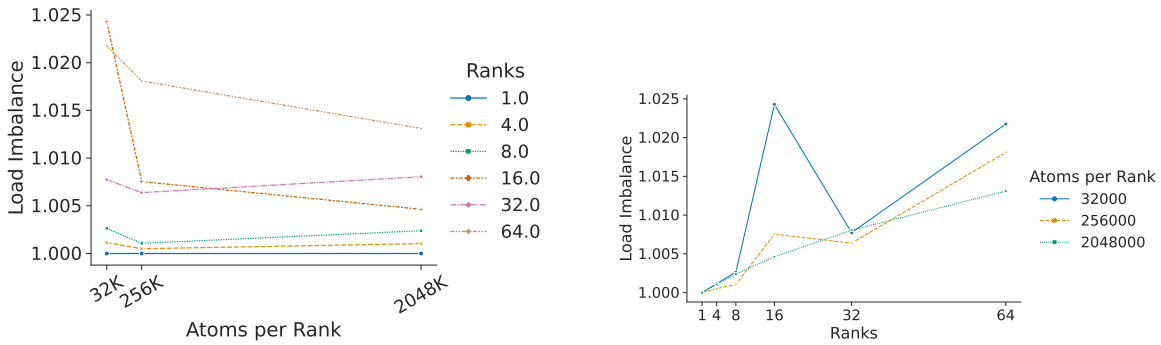
(b) Variable number of MPI ranks

Figure A.104.: Load Imbalance for `RCB: void ev_tally(...)`



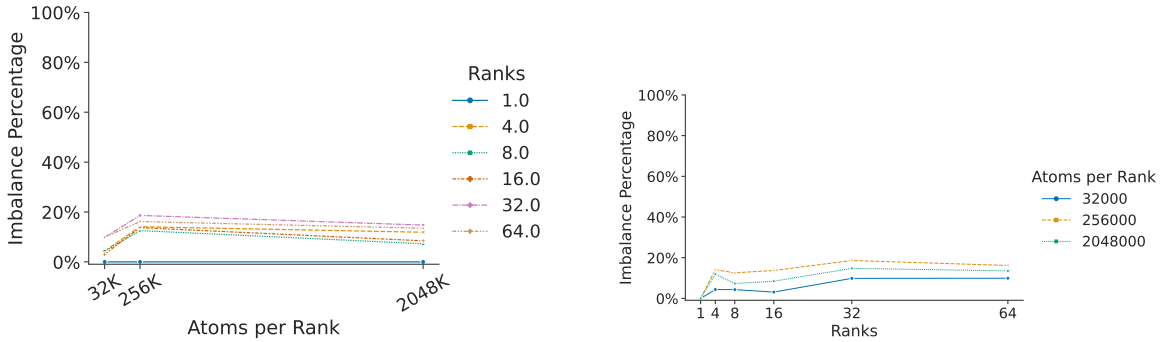
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.105.: Load Imbalance for `RCB: MPI_Send`



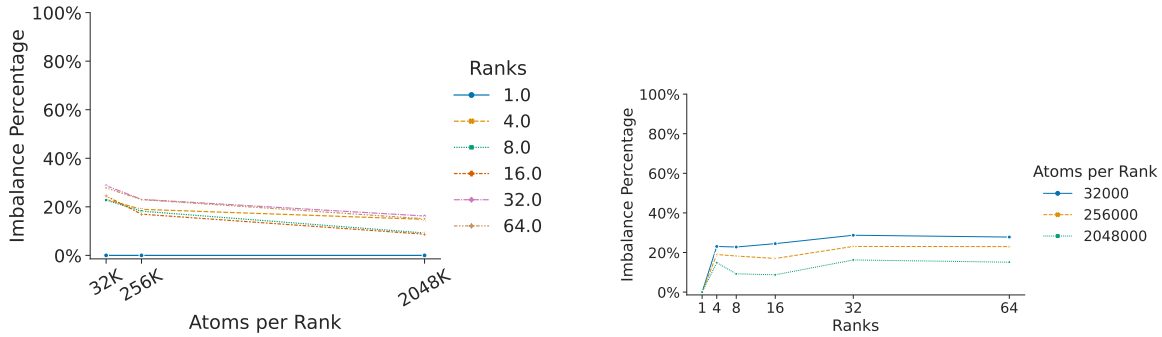
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.106.: Load Imbalance for `RCB: MPI_Init`



(a) Variable problem size **(b)** Variable number of MPI ranks

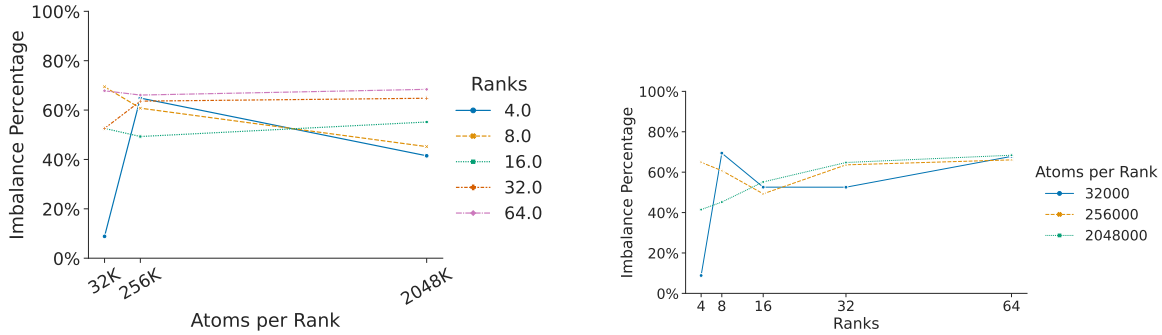
Figure A.107.: Imbalance Percentage for `RCB:void build(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

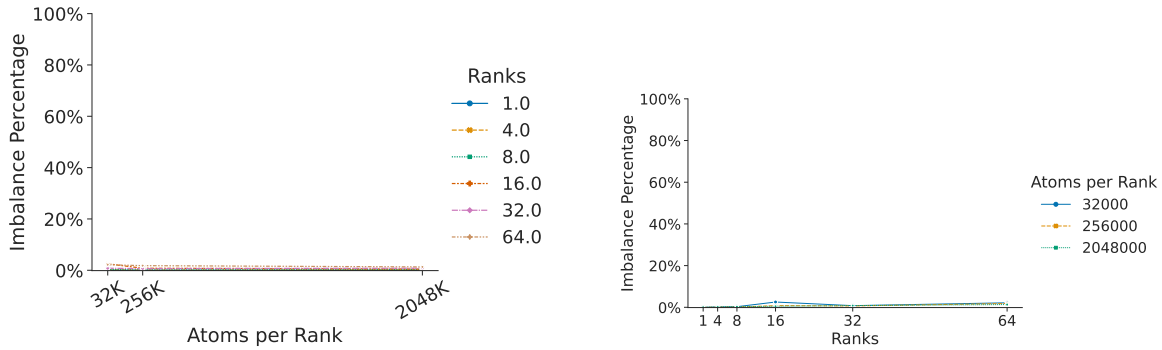
Figure A.108.: Imbalance Percentage for `RCB: void ev_tally(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

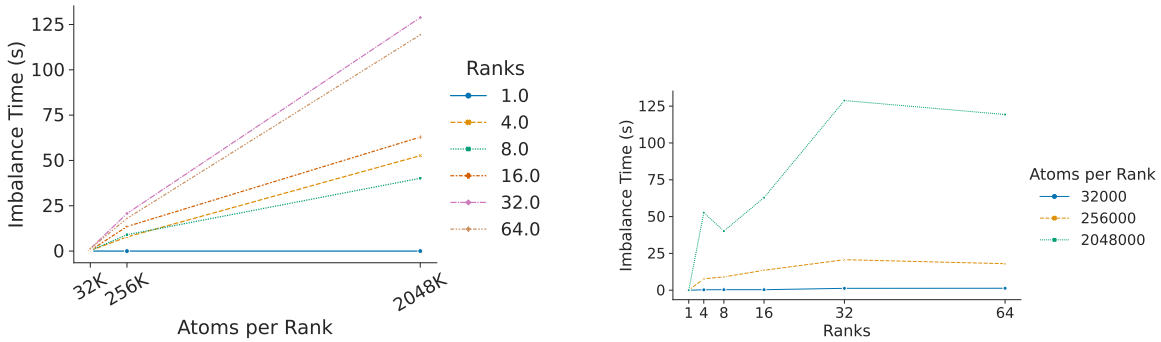
Figure A.109.: Imbalance Percentage for `RCB: MPI_Send`



(a) Variable problem size

(b) Variable number of MPI ranks

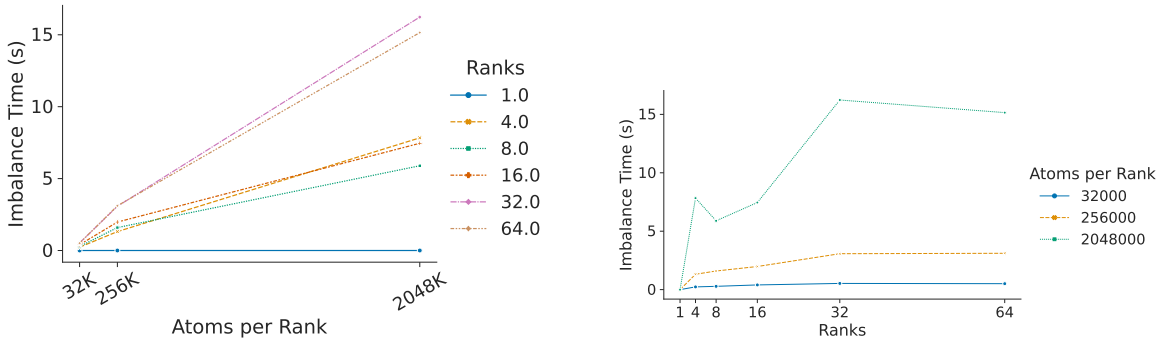
Figure A.110.: Imbalance Percentage for `RCB: MPI_Init`



(a) Variable problem size

(b) Variable number of MPI ranks

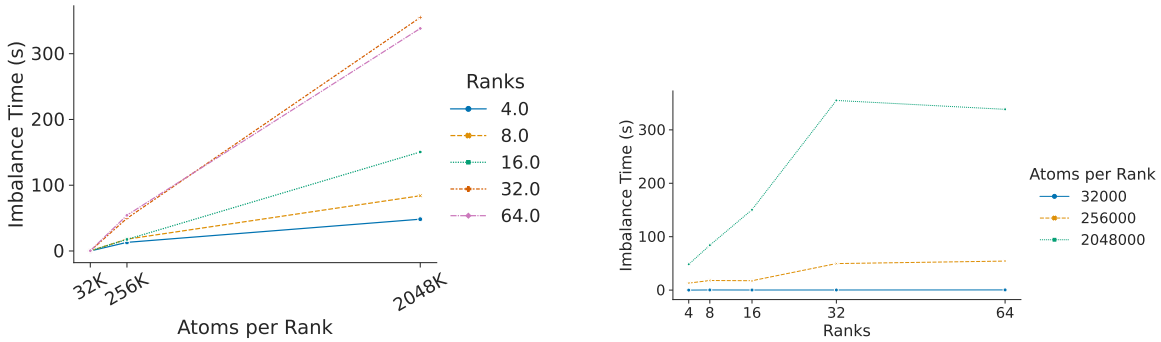
Figure A.111.: Imbalance Time for `RCB:void build(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

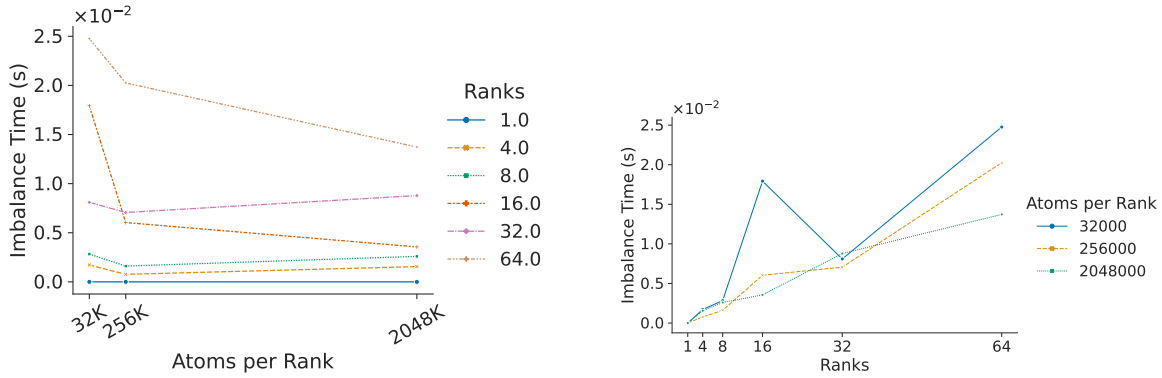
Figure A.112.: Imbalance Time for `RCB: void ev_tally(...)`



(a) Variable problem size

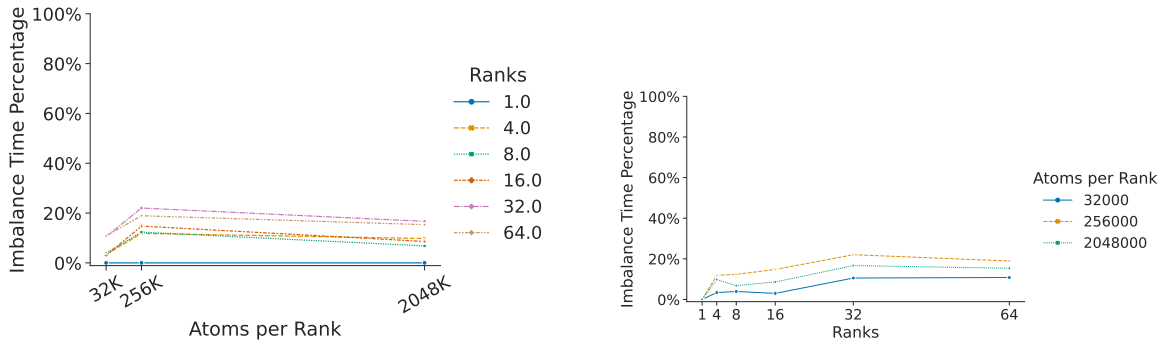
(b) Variable number of MPI ranks

Figure A.113.: Imbalance Time for `RCB: MPI_Send`



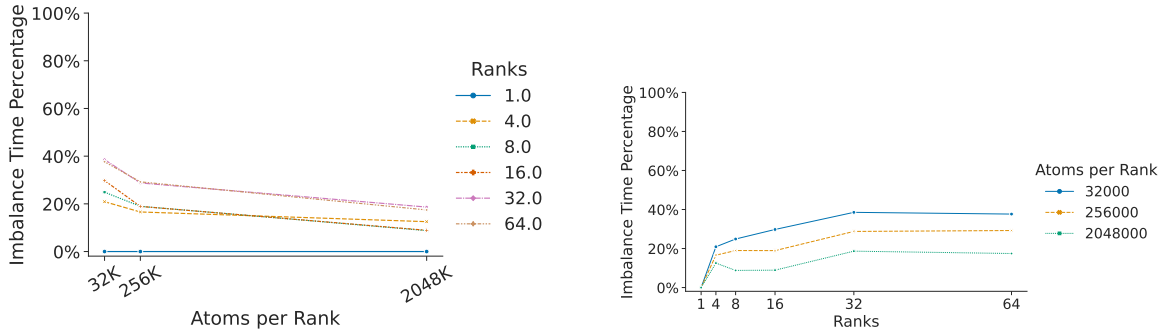
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.114.: Imbalance Time for `RCB: MPI_Init`



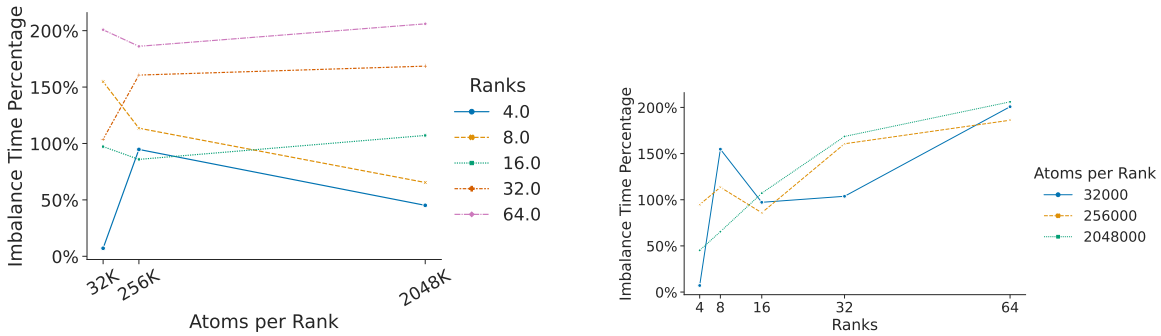
(a) Variable problem size **(b)** Variable number of MPI ranks

Figure A.115.: Imbalance Time Percentage for `RCB:void build(...)`



(a) Variable problem size **(b)** Variable number of MPI ranks

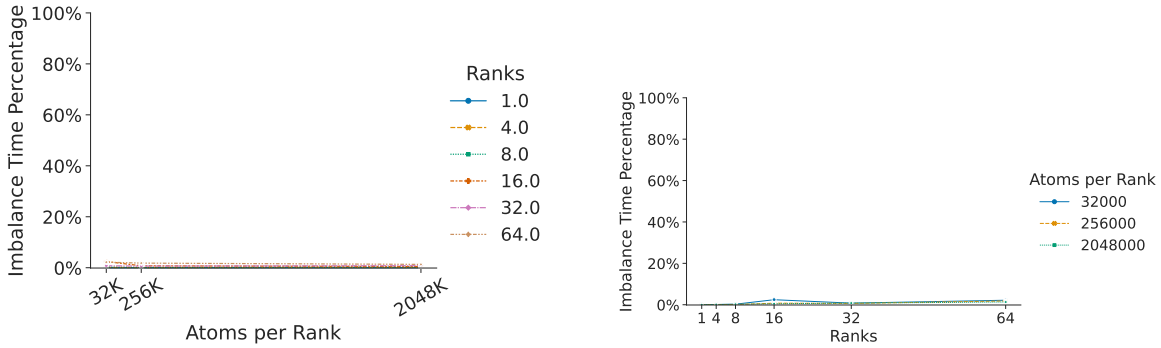
Figure A.116.: Imbalance Time Percentage for `RCB: void ev_tally(...)`



(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.117.: Imbalance Time Percentage for `RCB: MPI_Send`



(a) Variable problem size

(b) Variable number of MPI ranks

Figure A.118.: Imbalance Time Percentage for `RCB: MPI_Init`