TU WIEN Informatics

Über die Auslöser von Laufzeitvariabilität im Hochleistungsrechnen

Wie man sie bestimmt und wie man mit ihnen umgeht

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Niklas Roth, BSc
Matrikelnummer 01425096

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Wien, 28. Jänner 2021

Niklas Roth	Sascha Hunold

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

The Causes of Run Time Variability in HPC

How to Pin Them down and How to Handle Them

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Niklas Roth, BSc
Registration Number 01425096

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Vienna, 28th January, 2021

Niklas Roth	Sascha Hunold

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Niklas Roth, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. Jänner 2021

Niklas Roth

Danksagung

Hiermit möchte ich mich bei all jenen Personen bedanken, die maßgeblich zum Gelingen dieser Arbeit beigetragen haben. Als erstes sei an dieser Stelle mein Hauptbetreuer, Dr. Sascha Hunold erwähnt, der mich in allen erdenklichen Punkten bei dieser Arbeit unterstützt hat, und ohne den es diese Arbeit nicht geben würde. Des weiteren bedanke ich mich beim gesamten Team der Forschungsgruppe für die herzliche Aufnahme und das Büro dass mir hier freundlicherweise zur Verfügung gestellt wurde.

Des weiteren möchte ich meine große Dankbarkeit gegenüber meiner Partnerin, Michaela Faller, ausdrücken, die mich durch meine gesamte Studienzeit hindurch unterstützt hat und immer ein offenes Ohr für meine Probleme hatte, sowie meinem Kommilitonen Dipl.-Ing. Markus Lehr der mir immer ein Auge oder Ohr lieh, wenn ich eines brauchte.

Zu guter Letzt möchte ich mich bei meiner Familie bedanken, die mich über die gesamte Zeit des Studiums sowohl finanziell als auch emotional unterstützt hat, und mir damit sehr geholfen hat meine Ausbildung abzuschließen.

Acknowledgements

Hereby, I would like to thank all those people who have contributed significantly to the success of this work. First of all, I would like to mention my main supervisor, Dr. Sascha Hunold, who has supported me in every conceivable aspect of this work, and without whom this work would not exist. Furthermore, I would like to thank the entire team of the research group for the warm welcome and the office that was kindly made available to me.

Furthermore, I would like to express my great gratitude to my partner, Michaela Faller, who supported me throughout my studies and always had an open ear for my problems, as well as to my fellow student Dipl.-Ing. Markus Lehr who always lent me an eye or an ear whenever I needed one.

Last but not least, I would like to thank my family, who supported me financially and emotionally throughout my studies and thereby greatly helped me to complete my education.

Kurzfassung

High Performance Computing (HPC)-Cluster werden weltweit gebaut, um rechenintensive Probleme aus verschiedenen Forschungsbereichen zu bewältigen. Da diese Maschinen teuer in Bau und Wartung sind, ist es entscheidend, die maximale Leistung aus einer gegebenen Hardware herauszuholen, um sie so effizient wie möglich zu nutzen. Um die schnellsten und effizientesten Algorithmen und Implementierungen für eine bestimmte Operation und Maschine zu finden, wurden Mikrobenchmarks entwickelt, die die Laufzeit einzelner Operationen messen, indem sie viele von ihnen in einem Durchlauf ausführen. Leider können die Ergebnisse sehr stark variieren, aber sich auch um bestimmte Laufzeiten herum häufen, die nicht dem Optimum entsprechen. In dieser Arbeit werden wir einen Überblick über die verschiedenen Aspekte geben, von denen bekannt ist, dass sie die Laufzeit beeinflussen. Darüber hinaus werden wir die Ursache für die Variabilität in einer spezifischen Instanz ermitteln und dabei die Verwendung verschiedener, in der HPC-Community bekannter Tools aufzeigen. Dabei werden wir auch eine neue rudimentäre, leichtgewichtige Tracing-Bibliothek vorstellen, die verwendet wird, um mehr Informationen während der Messung zu erhalten, sowie eine ausgeklügelte statistische Methode, um das Ende der Aufwärmphase zu bestimmen. Schließlich werden wir den MPI Micro-Benchmark Fingerprint (MPI-MiBFi) als neue Methode zur Darstellung von Mikrobenchmark-Ergebnissen auf reproduzierbare und vergleichbare Weise vorschlagen.

Abstract

High Performance Computing (HPC) clusters are built worldwide to tackle computationally expensive problems from various research fields. Since these machines are expensive to build and maintain, it is crucial to get the most performance out of a given hardware in order to use it as efficient as possible. To find the fastest and most efficient algorithms and implementations for a given operation and machine, micro-benchmarks were developed which measure the run time of individual operations while executing a lot of them in a batch. Unfortunately, the results may not only vary a lot, but sometimes even cluster around specific run times other than the optimum. In this thesis, we will give an overview on the various parts of the software and hardware stack which are known to influence run time. Furthermore, we will track down the cause of variability in one specific case, while showing the usage and shortcomings of various tools known to the HPC community. Thereby, we will also introduce a new rudimentary light-weight tracing library which is used to get more information during a measurement and propose an elaborate statistical method to determine the end of the warm-up phase. Finally, we will propose the MPI Micro-Benchmark Fingerprint (MPI-MiBFi) as a new method of representing micro-benchmark results in a reproducible and comparable way.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

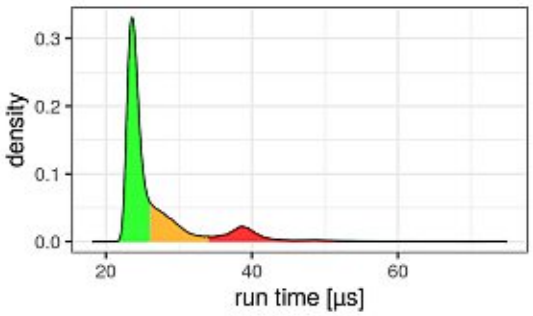
1.1 Problem Statement

There are a lot of computational problems that require more than a single compute node to be solved in a reasonable time. Therefore, many compute clusters were built worldwide to tackle these computationally expensive but parallelizable problems. The Message Passing Interface (MPI) [1–3] is widely used to utilize these supercomputers and was implemented by various companies and institutions (e.g., Open MPI [4], MVAPICH or Intel® MPI). To compare the different implementations of MPI and the compute clusters they are running on, benchmarks were designed to measure the performance of different MPI implementations on different HPC setups. One subgroup of these benchmarks are micro-benchmarks, which measure the run time of specific operations provided by the MPI.

Figure 1.1a shows a simple pseudo implementation of such a benchmark. Here, the time for each rank and each repetition is measured and saved.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD,
              &rank);

for (int i = 0; i < nrep; i++) {
   sync();

   tstart[rank][i] = get_time();
   collective_operation();
   tend[rank][i] = get_time();
}
```

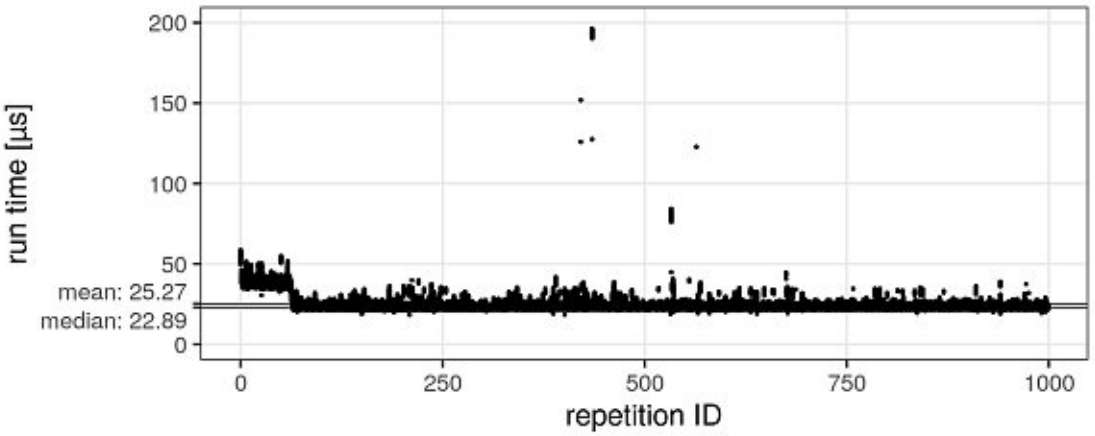
(a) Example code



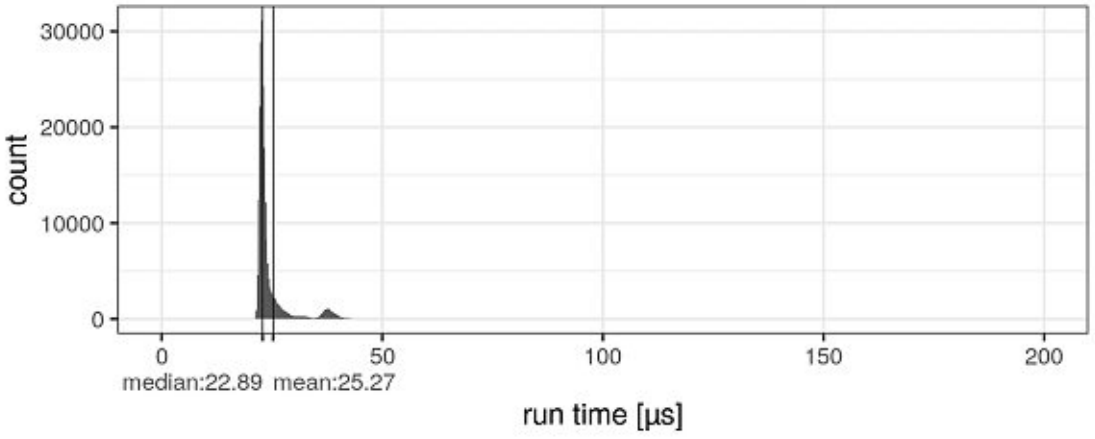
(b) Run time distribution

Figure 1.1: An example code for a simple micro-benchmark and the suboptimal run time distribution measured by it.

Considering the very simple structure of these benchmarks, we would expect to measure the same, or at least very similar, run times when executing code as shown in Listing 1.1a. Unfortunately, this is often not true. Figure 1.1b shows a typical run time distribution for such a micro-benchmark. Figure 1.2a shows the run time of each process in every repetition of a 1000-byte `MPI_Allreduce` operation. We can see that the run time can be up to 13 times the average run time. Furthermore, we see that there are about 70 repetitions in the beginning, where the average is around the double of the time of the later repetitions. Looking at the distribution of the run times in Figure 1.2b, we also see that the run times are clustered into multiple bumps on the time axis, indicating different modes. This is a very peculiar behavior, considering that the operation is the same in every iteration.



(a) The run time of every process for 1000 repetitions



(b) The distribution of the run times. The bin width is a single timer interrupt ($0.2384\mu s$).

Figure 1.2: The run time per repetition ID and process for 1000 repetitions of Allreduce (Open MPI 4.0.4/Recursive Doubling) with 1000 bytes on *Hydra*. Measured with ReproMPI [5] using round-time [6]. The straight vertical line represents the arithmetic mean. Only run times $\leq 200\,\mu s$ are shown.

This kind of run time variability is often experienced in larger HPC applications and it has many problematic impacts [7]. First, the variability always shifts the overall run time to the worse, since it can not get better than optimal, therefore less work is done in the same time. Second, variability can make measuring performance and tuning applications very hard because the benchmarks have to be performed very exhaustively to find consistent and statistically stable run time measurements. Moreover, not being able to predict the run time of applications can be very cumbersome in environments where processor time is bought for a specific application. If the processing time is predicted too low, the computations fail because the scheduler terminates the session. If it is predicted too high, it is more expensive to run, or the scheduler is not able to utilize the machine at its maximum potential.

Therefore, this thesis sets out to classify run time measurements of MPI collectives. The goal is to detect and analyze performance degradation in the benchmarking process. This knowledge will be beneficial to design a statistically sound comparison of algorithms and to eventually improve the performance of MPI libraries.

1.2 Research Questions

How can we measure the run time of MPI operations more consistently and reproducibly? Consistent benchmarks are crucial to find more performant implementations. If the benchmarks, which are used to determine whether one implementation is better than another, are inconsistent, the derived results are corrupted as well. This means that consistent and reproducible benchmarks are a key to well-performing MPI implementations. Thus, we will give advice on how to make benchmarks more consistent and reproducible.

How can we pin down the causes of performance variability for specific instances? To fix the performance issues of a given implementation, we need to find the cause for this specific variability. There are a lot of tools and libraries that should support developers during this process, but it is often hard to pick the right tool for the right task. We will give an overview of the available tools and how to use them to find the cause of performance variability in a given experimental configuration.

Why are HPC applications not performing consistently over time? To find the cause for a given experimental configuration and a certain run time behaviour, we have to know what could go wrong during execution and why the run time may differ from iteration to iteration. Therefore, we will discuss multiple causes which may lead to performance degradation on some iterations and thus introduce run time variability.

1.3 Structure of the Thesis

The rest of the thesis is structured as follows. Chapter 2 gives an overview of the possible reasons for run time variability found in the literature, introduces some tools used to find them, and approaches of other researchers in this field. Chapter 3 gives an overview on the system setup for our benchmarks and compares the different tools used during this thesis considering various aspects. In Chapter 4, we are going to exemplarily determine the reason for one specific run time variability problem on one of the HPC clusters at our research group, using multiple tools and statistical methods. We further propose a new rudimentary tracing library as well as a novel method to represent the results of micro-benchmarks in a reproducible and comparable way. Chapter 5 concludes the work and gives an outlook on future work in this field.

CHAPTER 2

Related Work

There is a lot of research tackling performance issues of HPC applications. We start with the research on performance variability (Section 2.1) and the different causes which are related to it. Then we give an introduction to different MPI Profiling and Tracing frameworks (Section 2.2) as well as to methods for MPI introspection (Section 2.3). Finally, we will relate to recent work using machine learning approaches to find performance variations and their underlying causes.

2.1 Performance Variability

Since modern compute clusters are very complex and have a lot of collaborating parts, there are many possible reasons why they sometimes perform worse than usual [7].

2.1.1 Influence of other Jobs

An obvious reason why jobs sometimes perform worse than expected can be that other jobs are interfering with them. Most HPC clusters therefore have schedulers like SLURM in place to prevent multiple jobs from sharing the same node. Nonetheless, there could be performance drawbacks even if the jobs do not share nodes because they use the same network. Research has shown that especially network-intensive jobs can interfere with jobs on the same cluster [8].

2.1.2 Hardware Throttling

Modern CPUs have the possibility to throttle themselves if the power consumption is too high or if they get overheated. Those throttle mechanisms can massively influence the performance of the running application. Fortunately, this throttling is mostly deactivated in big compute clusters or is prevented by sufficient cooling.

2.1.3 Cache Prediction

Caches are useful and very fast memory components which store often used parts of the main memory to make them available faster. Modern processors have very sophisticated

mechanisms to use these caches as efficiently as possible. Often these cache predication mechanisms need to "warm up", i.e., they monitor which parts of the RAM are used frequently and try to hold them in cache. Therefore, many benchmarks tend to skip the first few iterations to get only consistent measurements on warmed-up cache.

2.1.4 Branch Prediction

Since modern CPUs use pipelines for the execution of commands, they often need to prefetch code and data before even knowing exactly which instruction they will have to do next. For this, they use so-called branch prediction [9]. It predicts which line of code is executed next after a conditional statement. This part of the hardware/firmware also needs to warm up like the cache prediction described in Section 2.1.3.

2.1.5 Kernel Processes

Even if we have exclusive access to a compute cluster, the operating system's kernel will always use up some compute resources for it's own processes. These processes can also interfere with the job and cause run time peaks. Most supercomputer operators tend to lower the number of kernel processes as low as possible to prevent this [7], but a minimum of processes will always be there.

2.1.6 Different Software Branches

Nowadays, several sophisticated libraries tune themselves at run time to get more performance out of the underlying hardware [10]. This fine tuning at run time often depends on parameters which are not accessible from outside of the library and are therefore hard to track. This can lead to performance variability on the software side, while it is not visible to the user, program, or benchmark. Especially, libraries which use Machine Learning (ML) approaches to tweak internal parameters at run time are notoriously hard to debug or to pin down the reason for sudden performance variation.

2.2 MPI Profiling and Tracing

Profilers and tracers are tools which can give insights into the mechanics and interior of programs by collecting run times and performance counters without changing the code of the application. Since no code changes are needed, they are very handy at giving a first overview and hinting at the further research direction. Due to their limited interference, they are good at minimizing measurement artifacts but are also limited in what they can actually measure.

2.2.1 PAPI

Performance Application Programming Interface (PAPI) [11, 12] provides two interfaces to hardware counters: (1) a simple-to-use high-level interface which is used for simple measurements where the values are grouped into predefined sets and (2) a fully programmable low-level interface which is used to get fine-grained performance measurements. PAPI is designed to be portable across many different platforms and is widely used in this research field.

2.2.2 LIKWID-PerfCtr

LIKWID [13] stands for "Like I Knew What I'm Doing" and is a performance tool suite for GNU/Linux operating systems dedicated to performance-oriented programming. It contains a tool named `likwid-perfctr`, which measures performance counter metrics either for a whole program or between arbitrary points in the code by using a simple API. The main difference to PAPI is that LIKWID focuses on the command line tools while PAPI sees itself more as a library.

2.2.3 HPCToolkit

HPCToolkit [14] is a tool suite, maintained by the Rice University, which contains tools to sample timers and performance counters, visualize and analyze them. It uses statistical sampling on a fully optimized binary (e.g. `gcc -O3`) to measure run time and multiple hardware performance counters. It then links the measurements back to the source code and provides a tool which shows which part of the program takes up most of time. This leads to less influence on the running program but makes it hard to measure specific iterations of a given program.

2.2.4 Score-P

The Score-P measurement infrastructure [15] is a tool suite for profiling, event tracing, and online analysis of HPC applications. It works by instrumenting the code during compile time and then sampling the running binary. Score-P then exports this data into either the Open Trace Format 2 (OTF2), the CUBE4, or the TAU snapshot format for post mortem analysis or directly hands the data over to an on-line analysis tool like Periscope [16]. Tools which can be used to perform the post mortem analysis include VAMPIR [17], Scalasca [18], TAU [19] and Score-P's profiling back-end Cube GUI 4.

Score-P can be used in two modes. First, there is a tracing mode, which allows to retain temporal and spatial connections, and it can reflect the dynamical behavior to an arbitrary precision. This mode, however, uses a lot of resources and produces a big amount of data. Second, there is a profiling mode, which introduces far less perturbations to the measurements because it is much more lightweight. It also uses far less storage space for the produced trace/profile files.

2.3 Introspection

Introspection tools are harder to utilize than the profiling and tracing tools since they often require changes to the code. Some frameworks are already implemented and only have to be switched on in modern MPI implementations. The additional work pays off in more detailed granularity and tailored measurements.

2.3.1 PERUSE-Interface

The PERUSE-interface was designed in 2006 [20] to make performance variables from the inside of Open MPI available to the user of the library. It works by implementing callback functions which can be set outside of the MPI implementation to get called at interesting spots in the MPI code. For that, user-defined methods get linked to events defined by the PERUSE-interface using its Application Programming Interface (API). It was proposed as a MPI standard, but did not win standardization. It is currently supported in Open MPI but may be dropped in the future.

2.3.2 MPI_T

MPI 3.0 [2] introduced the MPI Tools Information (MPI_T) [21, 22] interface in 2012. This interface is supposed to give access to the performance and control variables of the MPI implementation. These variables may represent a particular property, setting, or performance measurement. They are initialized, set and incremented in various parts of the MPI implementation. Functions are provided to list, select and query values of the MPI_T pvars. It highly depends on the MPI implementation, how useful it is for monitoring the performance. Since every implementation exposes other performance counters and gives different control over the underlying mechanisms, this is not a very portable approach. In the open source implementation Open MPI, the PERUSE interface was replaced by the MPI_T concept. Furthermore, it is relatively easy to add additional software performance counters. Eberius, Patinyasakdikul, and Bosilca [23] proposed a few Software Performance Counters (SPCs) in 2017, which count the number of calls of different MPI methods. Those are currently available in Open MPI 4.0.4.

2.3.3 Caliper

Caliper is a “universal abstraction layer for general-purpose, cross-stack performance introspection” [24]. It provides mechanisms for software developers, tool developers, and performance engineers to add and measure arbitrary attributes for the whole HPC software stack. Therefore, it implements annotations for C, C++, and Fortran and a runtime environment which keeps track of the values and saves snapshots. Caliper uses an internal blackboard where every component of the HPC Software Stack can write current states and values of attributes. A snapshot engine saves snapshots of this blackboard whenever a trigger is called and callback functions can be registered to different events too, to allow on-line analysis. The memory management is optimized for storing the contextual information needed for typical performance engineering use cases.

The performance analysis of a Caliper annotated stack suggests that the annotation can be let in the production code and only be activated at run-time if needed. If a lot of software in HPC stacks were annotated with Caliper, it would be very easy for tool developers and users to analyze the performance.

2.4 Machine Learning Approaches

Due to the sheer amount of data which needs to be considered by monitoring-software and the very hard-to-determine thresholds for the different values, ML approaches were studied by various researchers. Tuncer et al. [25] proposed a concept in 2017 where they gathered multiple statistical features from healthy and problematic program runs and fed them into a ML system. The problematic runs were manually tagged in advance to give the ML system the possibility to learn the reasons of performance degradation in these instances. With this method, the program learned how to identify problematic runs and even the type of problem which was on hand by analyzing the data of new runs. The resulting program was able to identify most run time problems after the application stopped. For that, it analyzed the performance counters post mortem.

CHAPTER 3

Approach

In this chapter, we are going to clarify the environment of our benchmarks and how we will tackle our research questions. Section 3.1 introduces collectives, Sections 3.2 and 3.3 will introduce the hardware and software setup of our benchmarks respectively, and Section 3.4 will introduce the methodology we will use in our work.

3.1 MPI Collective Operations

MPI collective operations, or collectives for short, are functions which are defined in the MPI standard [1–3]. Those functions have to be called by all processes in a communicator and the underlying implementation performs the individual send and receive operations. These collectives include operations like:

- `MPI_Reduce`, which gathers the data from all nodes and computes a value using this data, like the mean or sum,
- `MPI_Bcast`, which distributes a specific value to all processes in the communicator, or
- `MPI_Allreduce`, which has the same result as executing `MPI_Reduce` and `MPI_Bcast` successively but is usually uses a more efficient implementation.

Run time of these operations is crucial for a well performing MPI implementation. Micro-benchmarks use various methods for synchronizing the processes and measuring the run time of single iterations. For our measurements, we do not use barriers because we found that those would distort the obtained values. We use window synchronization instead. This method works by communicating a start time and a latest stop time to every process in advance. Since all compute nodes in our hardware stacks have synchronized clocks, all processes will start their collective opertation at the same time. If a process finishes its work after the latest stop time, the measurement gets discarded. Otherwise, the run time is measured using the MPI wall clock time.

3.2 Hardware Stacks

Before we can get started, we have to introduce the systems we are performing our benchmarks on. We took the measurements for this work with the systems listed in Table 3.1. All of these systems are located at TU Wien and are maintained by the Research Group Parallel Computing. The two distributed memory systems *Hydra* and *Jupiter* have one additional head node which is not listed in this table and the compute nodes are managed by the node scheduler SLURM [26]. All these systems are synced to Precision Time Protocoll (PTP) hardware clocks [27]. Most of the in-depth analysis was done on the *Hydra* cluster.

Table 3.1: The hardware configurations of the used systems

Name	Nodes	Sockets	Processor Model	Cores	Network	OS
Hydra	36	2	Intel Xeon Gold 6130F	16	Omnipath	Debian
Jupiter	35	2	AMD Opteron 6134	8	Infiniband	CentOS

3.3 Software Setup

Usually, a HPC software stack looks like shown in Figure 3.1. The application uses the MPI API to communicate between processes on different machines on a high abstraction level using collectives and generic send and receive operations.

There are multiple implementations of MPI like Open MPI, Intel® MPI, and MVAPICH. These implementations are commonly layered by themselves. The layers are usually split by their different concerns.

The first and most exposed layer is the API which is specified by the current MPI-standard. This layer includes all methods which are specified by the standard and relays the request to the specific implementations.

These implementations of different algorithms for the MPI methods (especially for collective operations) form another layer which is separated from the others. Different algorithms and implementations perform better or worse on specific message and communicator sizes. Therefore, multiple algorithms are implemented to dynamically use the ones with the best known performance for a given task.

The deepest layers of the MPI implementations usually translate the generic send and receive operations to the network-stack-specific pendants. This layer highly depends on the underlying communication layer for shared and distributed memory systems and functions as an adapter between the MPI implementation and the underlying communication hardware.

The MPI implementations rely on the basic send and receive operations of the underlying network drivers and interfaces. These are tightly bound to the hardware of the system. For example, the `PSM2` interface handles message queues on an Intel® OPA network.

Finally, all software stacks are set up on the hardware. For the HPC application, the network and shared memory are crucial for good performance and scalability.

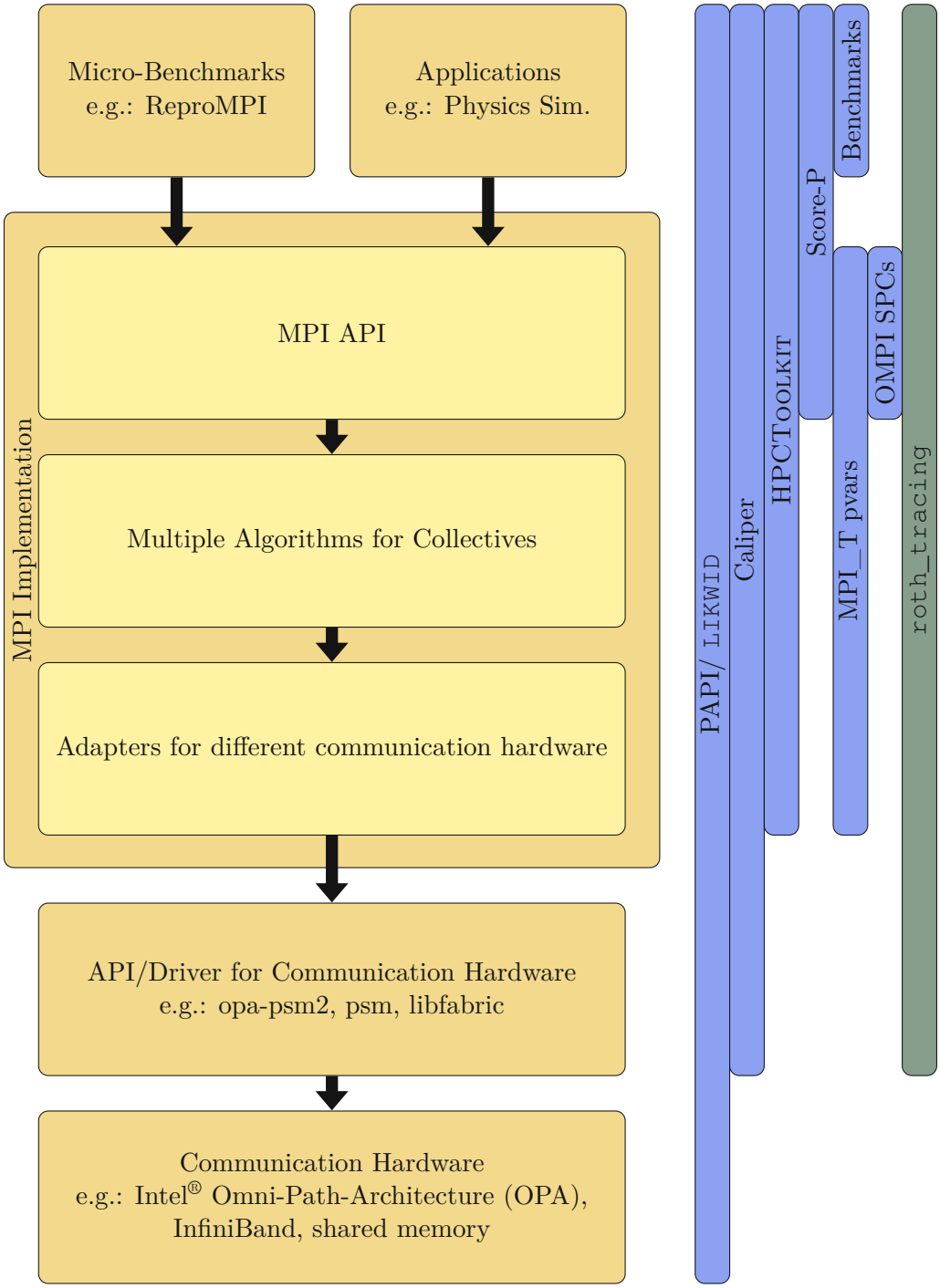


Figure 3.1: Software stack and reach of different tools and libraries

While HPC applications rely on highly performant MPI implementations and network drivers, it is hard to figure out the algorithms and implementation details which get the most out of the underlying hardware. (Micro-)benchmarks are used to perform very specific and granular run time tests for specific MPI operations like collectives. The idea behind testing the performance of individual operations is that when the performance of single operations is as high as possible, then the performance of the overall application improves too.

Table 3.2: Comparison of specifics of different tools and libraries

	likwid-perfctr	LIKWID Marker API	PAPI	HPC Toolkit	Score-P	MPI_T pvars	OMPI_SPCs	Caliper	roth_tracing
supports regions (for iterations)		x	x		x	x[a]	x[a]	x	x
needs code additions		x	x		x[d]	x	x	x[c]	x
needs recompilation		x	x		x	x[b]	x[b]	x	x
measures time	x	x	x	x	x	x	x	x	x
supports software counters						x	x	x	x
supports hardware counters	x	x	x					x	
footprint	med	med	low	med	med	low	low	med	low

[a] Counters have to be read at each iteration and stored by the benchmark.
[b] Software performance counters are often not included in production builds.
[c] Caliper intends that all major libraries are already annotated with Caliper, but this is not the case yet.
[d] Score-P only needs code additions for custom phases (region in Score-P).

Even by using micro-benchmarks, the lines of code which may have to be tuned in the MPI implementation to get better performance is very high and it is quite demanding to find the reasons for performance degradation. Therefore, a lot of tools have been proposed to help finding reasons, or at least locations, of potential performance issues. These tools have different scopes and reaches in the software stack, and some of them even record hardware performance counters. Figure 3.1 shows which parts of the software stack are reached by the different tools and libraries. The reach in the software stack is only one dimension in which the different tools differ and is not as useful on its own. Table 3.2 lists other specifics of the different libraries and compares them.

As we can see, PAPI and `LIKWID` seem to reach the whole stack, which is only partially true. While recording hardware performance over the whole stack, it is very hard to explicitly tell which part of the stack causes the performance counters to rise by only using these tools. Especially `LIKWID` without the marker API, only measures counters for the whole run of the application or benchmark without any distinction of regions or iterations.

Another tool which has a relatively far reach is HPCToolkit, and to use it, no part of the software has to be recompiled. It records the stack periodically and therefore, we

do not know when iterations or other dynamic regions start and end. Thus, there is no possibility to differentiate the measured run times by iteration. This makes it impossible to filter the measured time throughout the software stack for good and bad iterations of the micro-benchmark.

Score-P is intended to give an overview of the time spent in different methods of MPI and therefore to analyze applications rather that the MPI implementations. It does not reach beyond the MPI API layer and therefore does not give more insight into the MPI implementation than using a micro-benchmark by itself.

MPI_T pvars are designed to give insight into the MPI implementation and various performance relevant metrics. Open MPI implemented SPCs using the MPI_T framework. Unfortunately, the implemented SPCs only contain counters for the number of calls to the MPI API operations, but the framework can be used to add counters to the algorithmic layer. The adapters to the underlying communication hardware are not reachable with this method since the `ompi_spc.h`-file containing the SPC-framework results in a cyclic dependency if included further down the call stack. MPI_T pvars and therefore Open MPI SPCs do not support regions or iterations on their own, but it is possible to read and save the values of these counters at each iteration in the measuring micro-benchmark.

Caliper is a framework designed to enable performance introspection over the whole software stack. It provides the possibility to add counters and context from any part of the software stack to the Caliper blackboard. It also allows every part of the software stack to trigger snapshots of this blackboard and to consume the records produced by them. The idea behind Caliper is that a lot of applications, libraries, runtimes, and tools add Caliper annotations to their code and the user can configure which information should be recorded in a given run. The problem is that it is not widely used by now and therefore very time consuming to annotate all parts of the software stack by oneself.

To measure time for the whole software stack for single iterations of a micro-benchmark with a low performance impact, we will propose the new tracing library `roth_tracing` (see Section 4.6.2).

3.4 Methodology

To find the reasons for the run time variability on our HPC clusters, we will utilize many different measurement tools to get interesting values and different kinds of plots to analyze them.

Reproducible scripted benchmarks To find a baseline for the run times, we use the Open MPI 4.0.4 release and ReproMPI to benchmark the run time of MPI collective operations. With this baseline in mind, we add different tools and libraries which are designed to give additional information concerning the run. The different tools should be able to be easily switched on and off using C-macros, scripts, and compiler parameters. This allows us to perform consistent measurements with as few libraries activated as possible, which may influence the run time. Most of the used libraries, tools and frameworks are rebuilt from source and the parameters for doing this get documented too. To make the benchmarks even more reproducible, the dynamically

generated SLURM batch job script, the output of every build script, and the environment during the run get saved to an experiment directory for further analysis.

Melting the data to a CSV-file Due to the different tools used for the benchmarks, the data comes in different formats and is located in different structures in the experiment directory. To get the data into an easier-to-process format, the relevant output files get converted into a CSV-file with four columns, namely:

- iteration id,
- MPI rank,
- metric name, and
- value,

using a Python script. For example, if a line of the CSV-file is `0,255,PAPI_L3_TCM,229`, it means that during the zeroth iteration at rank 255 PAPI recorded 229 total level 3 cache misses. Since the CSV-files obtained by the scripted benchmarks can be as large as multiple gigabytes, they were compressed to save storage and IO-bandwidth.

Automatically generated plots Due to the amount of data generated by the benchmarks, it is essential to represent it in a workable format. Therefore, a lot of plots are generated by an R script to visualize the data and find clusters of run time as well as correlation with other variables obtained by the different tools.

Histograms The first widely used plot during this thesis is the histogram, which was first described by Pearson [28]. It is intended to give an approximation of the probability distribution of a given variable. This is done by plotting the frequency of a variable having a value in a certain range. This method will help to find clusters of run time and may give a hint that different states of the machine lead to different run times.

Scatter Plots Scatter plots, on the other hand, are very useful to identify the type of relationship between two variables visually, if there is any [29]. Scatter plots visualize 2 variables of a single record on a Cartesian grid where the x-coordinate represents the value of one variable and the y-coordinate represents the value of the other variable. If the pattern of the dots slopes upward from left to right, it would indicate a positive correlation between the two variables. If the pattern slopes downward, it would indicate a negative correlation. If it is not possible to identify any slope in the pattern of points, they are likely not correlated at all.

Pearson correlation coefficient Additionally to the visual approach for finding correlation, we use the Pearson correlation coefficient to get a quantifyable value for linear correlation. The Pearson correlation coefficient is 1.0 if all data points are on a line sloping upwards and -1.0 if all are on a line sloping downwards. The closer the Pearson correlation coefficient is to 1.0 or -1.0, the stronger the correlation.

Validate causation with tailored experiments Even though correlation often is a hint to causation, it does not imply it. Therefore, we need to strengthen our findings by giving a plausible causal link for the correlation and try to construct experiments which show that the causation holds.

Constructing a new representation Using the experience obtained by measuring the run time in various parts of the stack, and comparing it to other counters and timers, we will give advice on how to make benchmarks more consistent and reproducible. Therefore, we go over our findings and evaluate how these can be utilized to make better and more expressive benchmarks.

CHAPTER 4

Pinning Down the Causes for Run Time Variability

Most researchers in HPC have come across run time measurements which vary a lot, either over time, number of iterations, or other parameters [7, 30]. It is often not easy to find the reasons for this variability [5, 25].

We at the Research Group Parallel Computing at TU Wien, also had the problem that simple micro-benchmarks had seemingly different states of execution because the run times of single repetitions piled up at specific values. Figures 4.1a and 4.1b show the distribution of run times for individual Allreduce operations on two different machines at our research group. This chapter guides through the process of finding the cause of the run time variability on our systems.

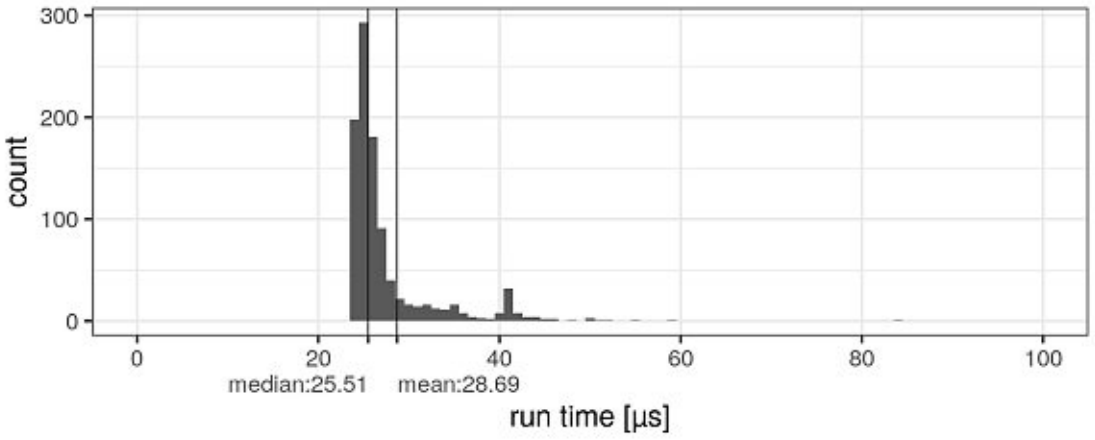
4.1 Measuring Run Time of MPI Collectives

As stated on the GitHub page of ReproMPI [31]:

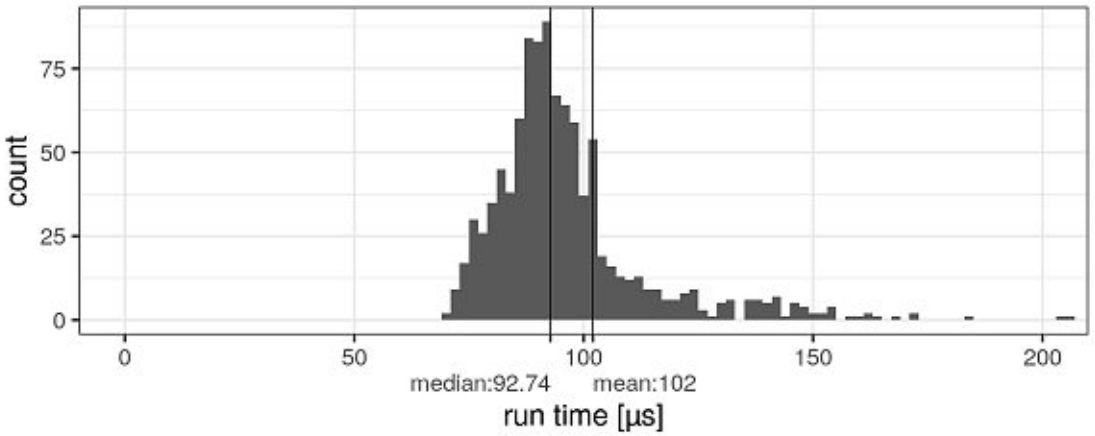
> The ReproMPI Benchmark is a tool designed to accurately measure the run-time of MPI blocking collective operations. It provides multiple process synchronization methods and a flexible mechanism for predicting the number of measurements that are sufficient to obtain statistically sound results.

Therefore, we used this tool developed by Hunold and Carpen-Amarie [5] to give the baseline of the run time measurements. This micro-benchmark also gives the possibility to print all measured run times at each rank and at each repetition to `stdout` after execution. This is a nice feature to find the variability of run times during the execution of one run. Most of the other variables measured by different tools, libraries, and frameworks were correlated with these run times to find the causes for the run time variability.

Listing 4.1 shows an shortened example output file to illustrate what is measure and how it is represented. The abbreviated sections are marked by "[...]".



(a) The distribution of the run times on *Hydra*.



(b) The distribution of the run times on *Jupiter*.

Figure 4.1: The run time of 1000 repetitions of `MPI_Allreduce` (Open MPI 4.0.4/Recursive Doubling) with 1000 B on *Hydra* and *Jupiter* with 16×16 ranks. Measured with ReproMPI [5] using round-time [6]. Only run times $\leq 100\,\mu s$ and $\leq 200\,\mu s$ respectively are shown.

We modified ReproMPI by adding modules for the libraries, tools, and frameworks that are explained in Sections 4.2 to 4.6. The added modules are easily switched on and off using compiler flags to minimize the performance impact while they are not needed.

4.2 Accessing Hardware Performance Counters

In Figures 4.1a and 4.1b, we can see that the measurements not only vary but cluster around some run times. In Figure 4.1a we see that the run time on *Hydra* clusters at approximately 25 µs and 42 µs. In Figure 4.1b we see that the runtime clusters are not that clear at *Jupiter*, but we can see multiple modes around 90 µs, 102 µs, and 135 µs to 155 µs. Therefore, we guessed that this is maybe a hint that cache misses on the different cache levels correlate with the observed run times.

Listing 4.1: Example output of ReproMPI (shortened)

```
#Command-line arguments:  /home/staff/roth/reprompi-dev/bin/mpibenchmark [...]
#MPI calls:
#       MPI_Allreduce
#Message sizes:
#       100
#@operation=MPI_BOR
#@datatype=MPI_BYTE
#@datatype_extent_bytes=1
#@datatype_size_bytes=1
#@root_proc=0
#@reproMPIcommitSHA1=unknown
#@nprocs=256
#@clock=clock_gettime_REALTIME
#@clocksync=None
#@procsync=roundtime
#@bcast_nrep=30
#@bcast_runtime_s=0.0000863552
#@barrier_switch_count=-1
#@runtime_type=global
#@nrep=1000
process           test  nrep  count loc_tstart_sec          loc_tend_sec
      0  MPI_Allreduce     0    100 1604323187.8179886341   1604323187.8180692196
      0  MPI_Allreduce     1    100 1604323187.8298249245   1604323187.8298730850
      0  MPI_Allreduce     2    100 1604323187.8311803341   1604323187.8312242031
      0  MPI_Allreduce     3    100 1604323187.8325388432   1604323187.8325812817
      0  MPI_Allreduce     4    100 1604323187.8338882923   1604323187.8339321613
      0  MPI_Allreduce     5    100 1604323187.8352437019   1604323187.8352870941
[...]
    255  MPI_Allreduce     997  100 1604323189.1455829144   1604323189.1456046104
    255  MPI_Allreduce     998  100 1604323189.1465573311   1604323189.1465790272
    255  MPI_Allreduce     999  100 1604323189.1475298405   1604323189.1475510597

# Benchmark started at Mon Nov  2 14:19:47 2020
# Execution time: 104s
```

To verify this theory, we measured the cache misses using two different tools established in the field. Afterwards, we correlate them with the run time and hope to find a correlation between the number of cache misses and the run time.

4.2.1 Applying `LIKWID`

The first tool we used was `LIKWID`. The part of `LIKWID` in which we are interested in this work is the performance counter tool `likwid-perfctr` and the MPI wrapper of it `likwid-mpi`.

The performance counter tool has a so-called marker API, which allows us to measure named code regions by changing the code. Listing 4.2 shows an example of how to use the regions to measure performance counters for every iteration. As we can see, regions are started with dynamic names for each iteration of the benchmark. This is needed to perform measurements for every iteration and to find the causes for the different run times in different iterations.

As we can see, the `LIKWID` marker API needs initialization with the two commands `LIKWID_MARKER_INIT` and `LIKWID_MARKER_THREADINIT`. Furthermore, it is recommended by the developers to register the different regions in advance using the function `LIKWID_MARKER_REGISTER` to minimize the initialization impact during measurement [32].

The code using the marker API has to be executed using the aforementioned `likwid-mpi`

Listing 4.2: Example for code that measures LIKWID performance counters using one region for every iteration

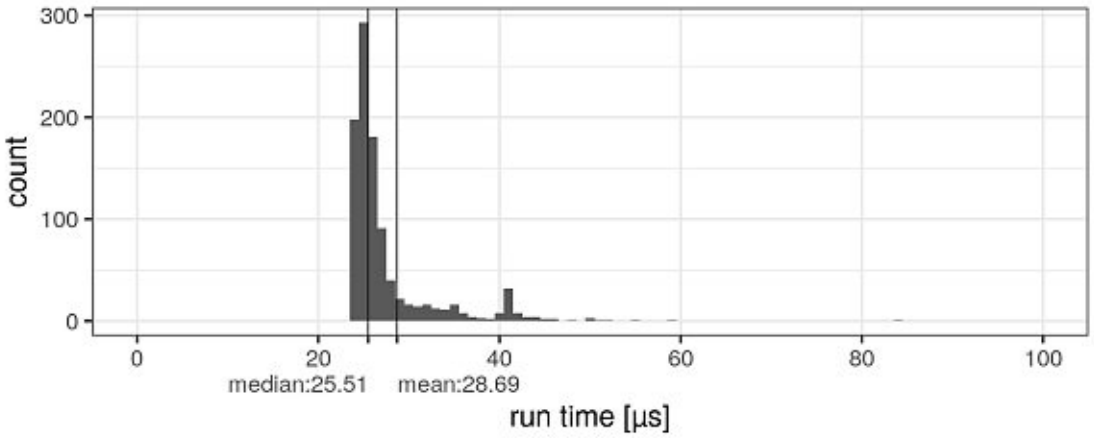
```
#include "likwid.h"
#include "mpi.h"

int main() {
    int rank; int retval;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    LIKWID_MARKER_INIT;
    LIKWID_MARKER_THREADINIT;
    for (long i = 0; i < nrep; i++) {
        char *region_name = get_region_name(i, nrep);
        LIKWID_MARKER_REGISTER(region_name); // register tags
        free(region_name);
    }
    for (int i = 0; i < nrep; i++) {
        char *region_tag = get_region_name(i);
        LIKWID_MARKER_START(regionTag); // use tags
        sync();
        tstart[rank][i] = get_time();
        collective_operation();
        tend[rank][i] = get_time();
        LIKWID_MARKER_STOP(regionTag);
    }
    LIKWID_MARKER_CLOSE;
}
```

or the LIKWID performance counter tool likwid-perfctr directly. The counters which should be measured are specified using command-line arguments. In the end of the run, tables (CSV or ANSI) for every region are printed to stdout containing the different ranks as well as the measured performance counters and metrics. There is also a table containing statistical values like sum, minimum, maximum, and average over the different ranks printed for each region.

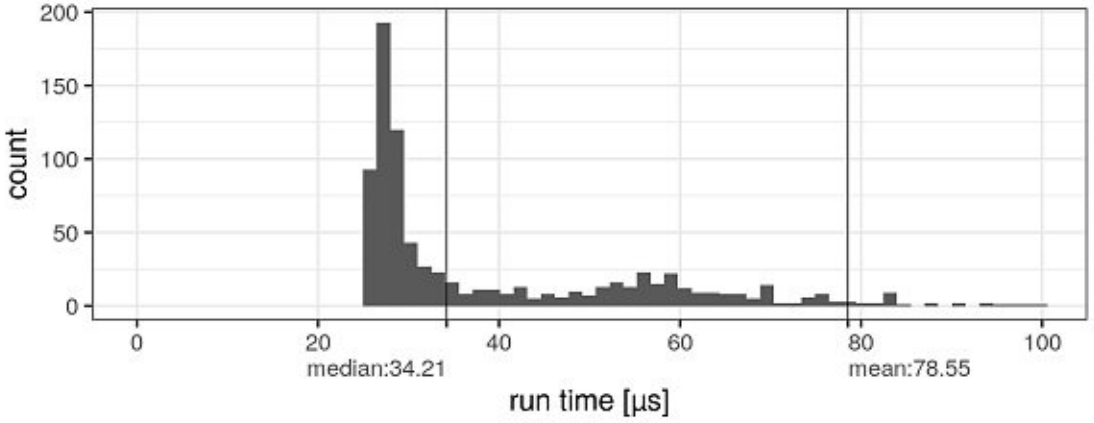
Multiple counters and metrics can be measured by likwid-perfctr. For this work, we used the predefined CACHES measurement group. As the name already implies, it contains different performance counters and metrics which are linked to the caches.

Unfortunately, the output data measured by LIKWID was seemingly too big to be calculated and printed to the output file in a reasonable time (15 minutes) for runs greater than 1000 iterations and 16×16 ranks.

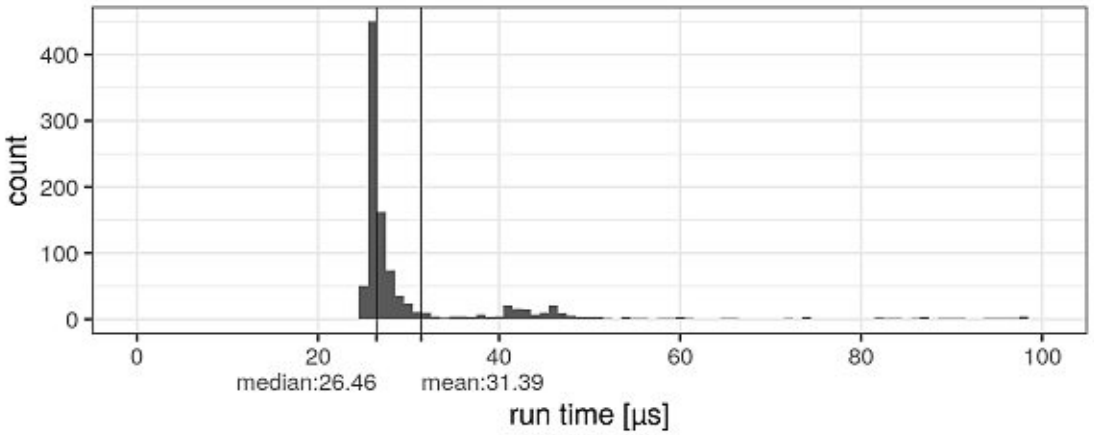
Furthermore, as we can see, if we compare Figures 4.2a and 4.2b, the accessing of hardware counters with LIKWID distorted the time measurements a lot. For this reason, correlating the data would not lead to in satisfying results. Therefore, LIKWID was not considered further and another method had to be found to measure the hardware performance counters and especially cache misses.



(a) Run time distribution without additional measurements. (Also see Figure 4.1a)



(b) Run time distribution with `LIKWID` switched on.



(c) Run time distribution with PAPI switched on

Figure 4.2: Run time distribution for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra* with nothing, `LIKWID` and PAPI switched on respectively. Only run times $\leq 100\,\mu s$ are shown. The comparison of Figures 4.2b and 4.2c with 4.2a shows that `LIKWID` introduces worse perturbation than PAPI.

4.2.2 Applying PAPI

PAPI [11, 12] was used as a second option to measure hardware performance counters. While LIKWID is a tool with optional library parts, PAPI is library-centric. This means that while LIKWID can be run without changing the source code of the program, PAPI needs to be included into the source code to be usable.

Since we want to measure the counters for each iteration, we need to change the source code anyway. PAPI is used very similar to LIKWID. See Listing 4.3 to find an example code which wraps each iteration into its own measurement region.

Listing 4.3: Example for code which measures PAPI performance counters in regions

```
#include "papi.h"
#include "mpi.h"

int main() {
    int rank; int retval;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (int i = 0; i < nrep; i++) {
        char *region_tag = get_region_name(i);
        PAPI_hl_region_begin(region_tag);
        sync();
        tstart[rank][i] = get_time();
        collective_operation();
        tend[rank][i] = get_time();
        PAPI_hl_region_end(region_tag);
    }
    PAPI_hl_stop();
}
```

All activated performance counters are saved into JSON-like files for each rank at the end of the run by calling PAPI_hl_stop. In each file, the measurements are listed by the region they were taken in.

The hardware performance counters analyzed in this thesis are listed in Table 4.1.

Table 4.1: The PAPI Hardware Performance Counters measured during this work

Counter identifier	Description
PAPI_L1_TCM	Level 1 total cache misses
PAPI_L2_TCM	Level 2 total cache misses
PAPI_L3_TCM	Level 3 total cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_PRC	Conditional branch instructions correctly predicted

Figure 4.2c shows the run time distribution with active PAPI instrumentation. The difference to the run times without instrumentation is measurable but tolerable (compare to Figure 4.2a).

Figure 4.3 shows the correlation between run time and the number of cache misses on different cache levels. Each dot represents one rank at one iteration. If the main cause for the variability would be the cache misses we would expect that the runs with longer run times tend to have a higher number of cache misses and therefore, we would expect the points in the scatter plot to follow an upward sloping line. Unfortunately, we did not find such results. We can see that there are measurements, where the run time is very high while the cache misses are low, and we can also see iterations where the cache misses are very high but the run time is only slightly increased. Additionally, we checked the Pearson correlation coefficient which was not even close to 1.0 and thus did not suggest a correlation either. Therefore, we conclude that cache misses are not the (only) metric influencing the run time.

4.3 Using Software Performance Counters

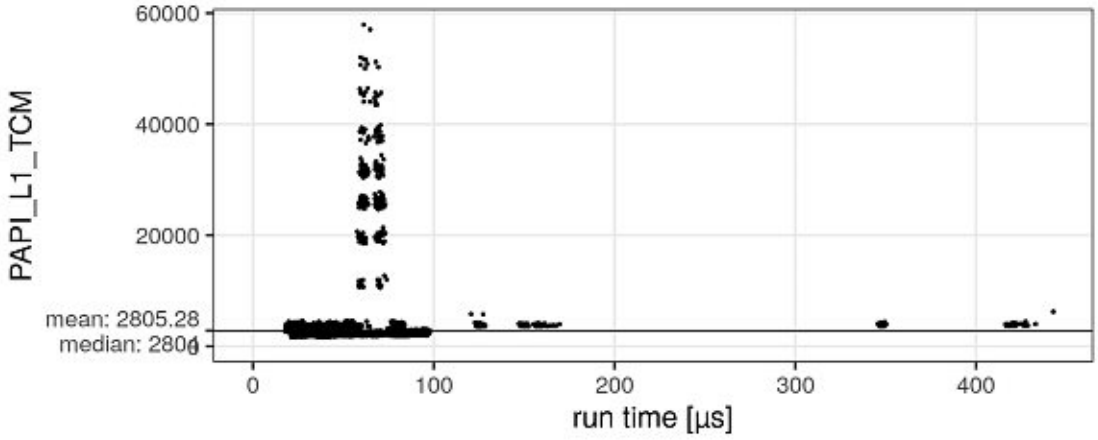
Since we did not find any hardware performance counters with a high correlation to the run times, we tried to find Software Performance Counters (SPCs) which are correlated with the run time. MPI_T provides a mechanism for MPI implementors to expose variables from within the MPI implementation. Especially the insight into internal performance information makes this interface very valuable for our project.

4.3.1 Open MPI 4.0.4

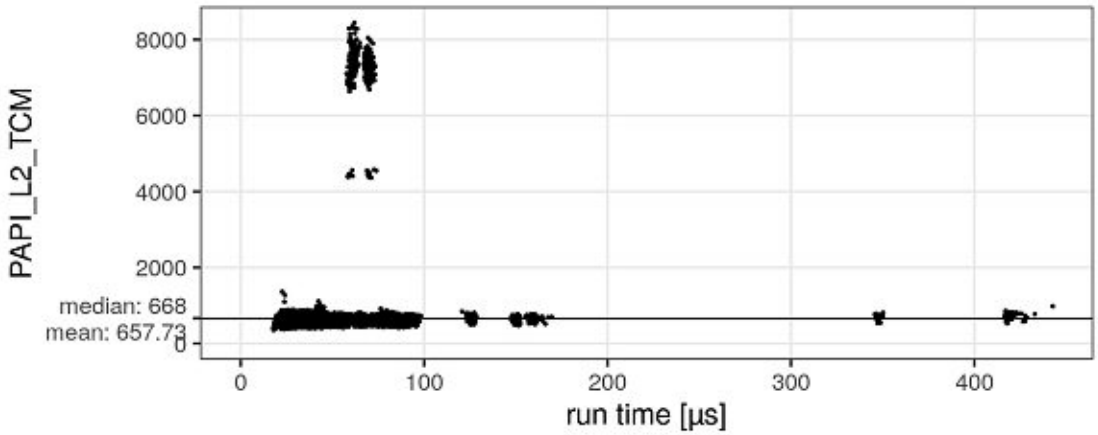
In 2017, Eberius, Patinyasakdikul, and Bosilca [23] proposed SPCs for Open MPI using pvars in the MPI_T interface. Table 4.2 shows the performance counters introduced by them. The variable names reveal their purpose. In 2018, the performance counters were merged into the master branch of Open MPI. During that, additional counters were added to have a call counter for nearly every MPI method available in Open MPI.

Table 4.2: The names of the performance counters from Eberius, Patinyasakdikul, and Bosilca [23]. The Point-to-Point Management Layer (PML) is the level directly below the MPI level and represents the abstract send and receive operations.

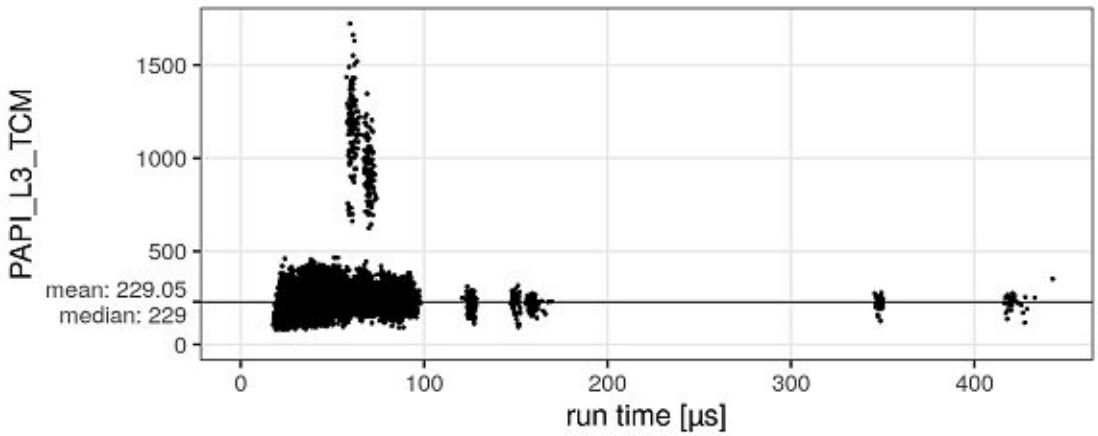
MPI Level	PML Level
OMPI_SEND	OMPI_BYTES_RECEIVED_USER
OMPI_RECV	OMPI_BYTES_RECEIVED_MPI
OMPI_ISEND	OMPI_BYTES_SENT_USER
OMPI_IRECV	OMPI_BYTES_SENT_MPI
OMPI_BCAST	OMPI_BYTES_PUT
OMPI_REDUCE	OMPI_BYTES_GET
OMPI_ALLREDUCE	OMPI_UNEXPECTED
OMPI_SCATTER	OMPI_OUT_OF_SEQUENCE
OMPI_GATHER	OMPI_MATCH_TIME
OMPI_ALLTOALL	OMPI_OOS_MATCH_TIME
OMPI_ALLGATHER	



(a) Level 1 Total Cache Misses



(b) Level 2 Total Cache Misses



(c) Level 3 Total Cache Misses

Figure 4.3: Correlation between cache-misses and runtime of each repetition on every rank for 1000 repetitions of 16 × 16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra* measured with PAPI.

We accessed the pvars by saving the difference of the variables before and after the run using a struct array. At the end of the run, the measured data is collected at one rank and printed to `stdout`.

Unfortunately, only the topmost MPI layer was targeted by these counters and also timers were only sparsely introduced by Eberius, Patinyasakdikul, and Bosilca [23]. These counters only showed which MPI collectives were called and how many bytes were sent which was already obvious in our case, but it was easy to add our own SPCs further down the call stack.

4.3.2 Required Novel Additions to Open MPI

We needed more fine-grained SPCs. So, we added multiple counters and timers. Since we mainly focused on the `MPI_Allreduce` collective operation, the new SPCs are centered around this call stack. Table 4.3 shows our new performance counters and gives a description of them.

We used these SPCs to check whether Open MPI uses different algorithms from time to time which could explain the clusters. Additionally, we wanted to determine if the additional time is lost during calculation or while communicating. To do the runs with additional performance counters, the newest release tag `v4.0.4` was checked out from the Open MPI GitHub repository [33] and the additional SPCs were added.

We found that Open MPI did not change the algorithm during run time for fixed message sizes, since only a single algorithm SPC was incremented for every iteration.

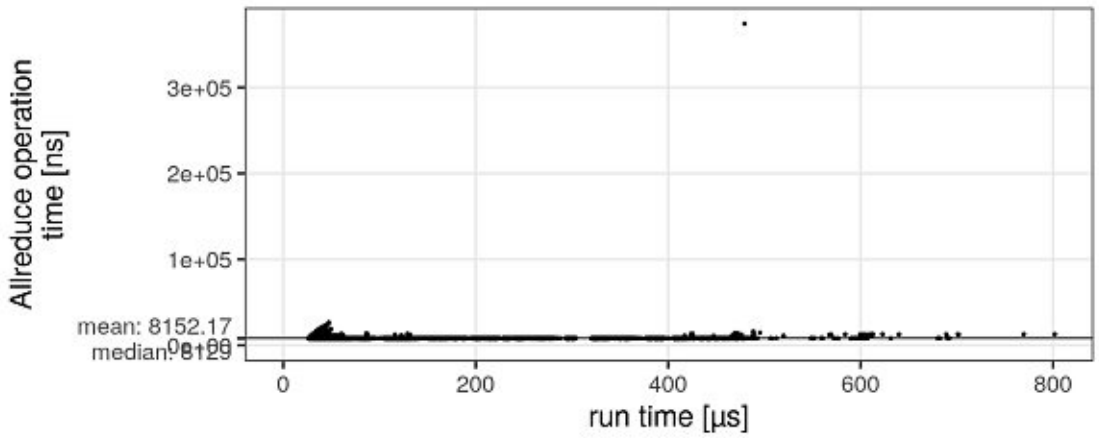
After incremental addition of counters, we found that most of the time was spent during communication and that the operation part of Allreduce only takes nearly constant time (see Figures 4.4a and 4.4b). Figure 4.4a shows the time which was spent during the calculation part of the `MPI_Allreduce` and Figure 4.4b shows the time which is spent during communication part. Since the correlation between communication time and overall run time is very high (a line sloping upwards in the scatter plot), and the time for calculations stays virtually constant, we took a look at the communication and found that a major amount of time is spent during the wait for the `ompi_request_wait(...)` call of the underlying `ompi_coll_base_sendrecv_actual(...)` (see Figure 4.4c which compares the overall run time to the wait time).

Unfortunately, due to the internal structure of the SPCs framework, we were not able to get further down the call stack using this method without getting cyclic dependencies.

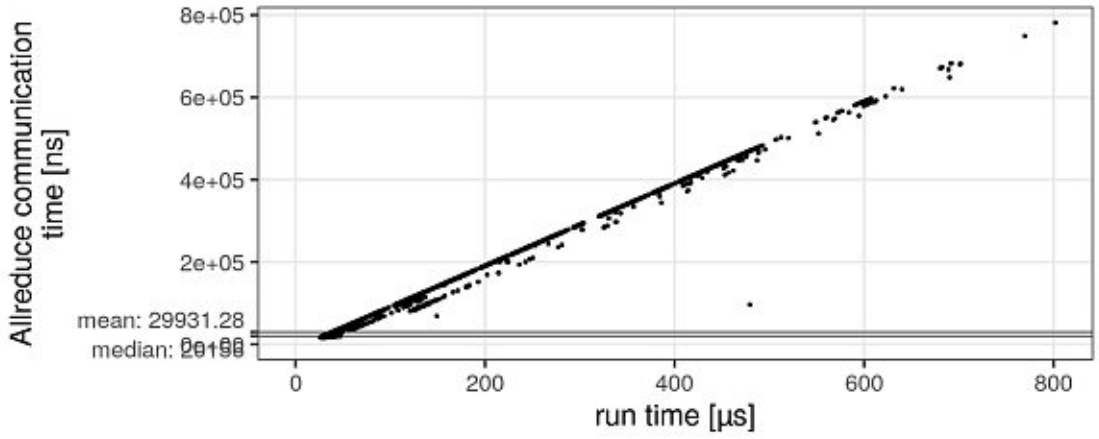
The complete set of plots for the various measured SPCs can be found in the Appendices A and B.

Table 4.3: The newly introduced SPCs and their description

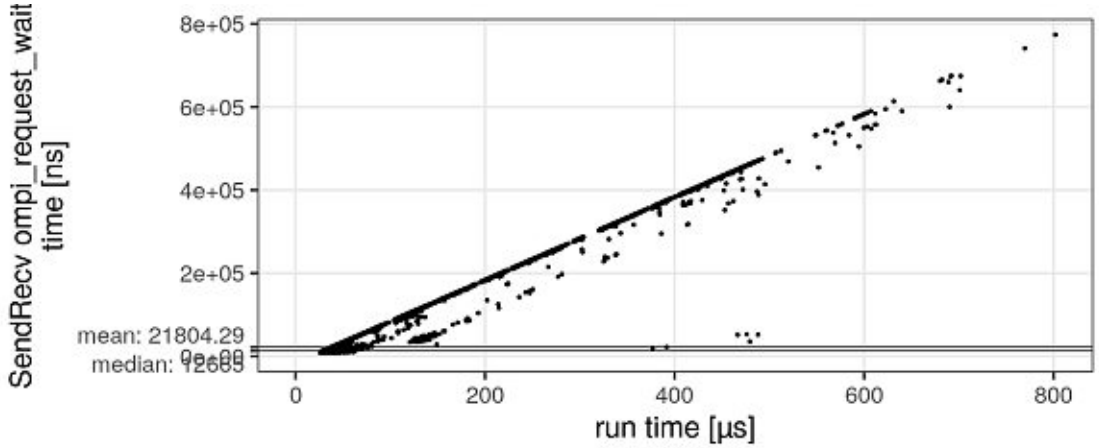
OMPI_SPC_BASIC_ALLREDUCE_INTRA The number of times MPI_Allreduce used the basic intra algorithm.
OMPI_SPC_BASIC_ALLREDUCE_INTER The number of times MPI_Allreduce used the basic inter algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_NONOVERLAPPING The number of times MPI_Allreduce used the INTRA_NONOVERLAPPING algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_RECURSIVEDOUBLING The number of times MPI_Allreduce used the INTRA_RECURSIVEDOUBLING algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_RING The number of times MPI_Allreduce used the INTRA_RING algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_RING_SEGMENTED The number of times MPI_Allreduce used the INTRA_RING_SEGMENTED algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_BASIC_LINEAR The number of times MPI_Allreduce used the INTRA_BASIC_LINEAR algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_REDSCAT_ALLGATHER The number of times MPI_Allreduce used the REDSCAT_ALLGATHER algorithm.
OMPI_SPC_SEND_TIME The time in nanoseconds spent while sending.
OMPI_SPC_SENDRECV_TIME The time in nanoseconds spent while calling sendrecv.
OMPI_SPC_RECV_TIME The time in nanoseconds spent while receiving.
OMPI_SPC_BASE_ALLREDUCE_INTRA_RECURSIVEDOUBLING_DATA_EXCHANGE_TIME The time in nanoseconds spent for data exchange in the INTRA_RECURSIVEDOUBLING algorithm.
OMPI_SPC_BASE_ALLREDUCE_INTRA_RECURSIVEDOUBLING_OPERATION_TIME The time in nanoseconds spent for applying the operation in the INTRA_RECURSIVEDOUBLING algorithm.
OMPI_SPC_SENDRECV_POST_IRECV_TIME The time in nanoseconds spent in sendrecv to post the IRECV.
OMPI_SPC_SENDRECV_SEND_TIME The time in nanoseconds spent spent in sendrecv to SEND.
OMPI_SPC_SENDRECV_REQUEST_WAIT_TIME The time in nanoseconds spent sendrecv waiting for the request.
OMPI_SPC_REQUEST_DEFAULT_WAIT The number of times ompi_request_default_wait was called.
OMPI_SPC_REQUEST_DEFAULT_WAIT_ANY The number of times ompi_request_default_wait_any was called.
OMPI_SPC_REQUEST_DEFAULT_WAIT_ALL The number of times ompi_request_default_wait_all was called.
OMPI_SPC_REQUEST_DEFAULT_WAIT_SOME The number of times ompi_request_default_wait_some was called.
OMPI_SPC_OPAL_PROGRESS The number of times opal_progress was called in ompi_request_wait_completion.
OMPI_SPC_REQUEST_WAIT_COMPLETION_THREADS The number of times the threads branch was executed in ompi_request_wait_completion
OMPI_SPC_OPAL_PROGRESS_TIME The time spent in opal_progress in ompi_request_wait_completion.
OMPI_SPC_REQUEST_DEFAULT_WAIT_REQUEST_WAIT_COMPLETION_TIME The time spent in ompi_request_wait_completion in ompi_request_default_wait.
OMPI_SPC_REQUEST_DEFAULT_WAIT_CRCP_REQUEST_COMPLETE The time spent in OMPI_CRCP_REQUEST_COMPLETE in ompi_request_default_wait.
OMPI_SPC_REQUEST_DEFAULT_WAIT_AFTERMATH The time spent in the aftermath of ompi_request_default_wait.



(a) Time for the operation part of Allreduce



(b) Time for the communication part of Allreduce



(c) Time for the `ompi_request_wait(...)`

Figure 4.4: Correlation between run time and the time for communication and operation for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The SPCs from Table 4.3 represented here are `OMPI_SPC_BASE_ALLREDUCE_INTRA_RECURSIVEDOUBLING_OPERATION_TIME`, `OMPI_SPC_BASE_ALLREDUCE_INTRA_RECURSIVEDOUBLING_DATA_EXCHANGE_TIME`, and `OMPI_SPC_SENDRECV_REQUEST_WAIT_TIME` respectively.

4.4 Interference of Other Processes

Plotting the run time for each repetition, we find that the outliers are also clustered together over time. This led to the idea that other processes are interfering with the benchmark and that the run times vary because other processes are using the communication layer too. Therefore, the `proc` filesystem was used to find possible culprits.

The `proc` filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at `/proc`. The proc filesystem contains a lot of mostly read-only files which contain process information of the Linux kernel. During this project, mainly the stat file under `/proc/[pid]/stat` was used to determine whether other processes consume time on the processor and if they may influence the run time.

Unfortunately, the reading of the stat files is not very lightweight and distorts the run time measurements. When we compare Figures 4.6a and 4.6b, we see that the mean is approximately 30 % larger, although we read the `proc` filesystem outside of the time measurement region. Since Linux based operating systems only provide this interface to find information about other processes [34] we were not able to find another way to measure the influence of other processes without patching the Linux kernel, e.g., by adding an interface like `task_diag` developed by Vagin and Kolyshkin [34].

Another disadvantage is that the granularity of the time spent in a process is 1 Linux clock tick which is 10 ms on *Hydra* and therefore around 300 times higher than the time we measure with our micro-benchmark. Nevertheless, we evaluated the data and were not surprised that no correlation between run time and process time of other processes could be found with this method (see Figure 4.5). The `proc` filesystem's time resolution is too coarse and has too much impact on the run time to neither discard the hypothesis of process interference nor to show that it holds.

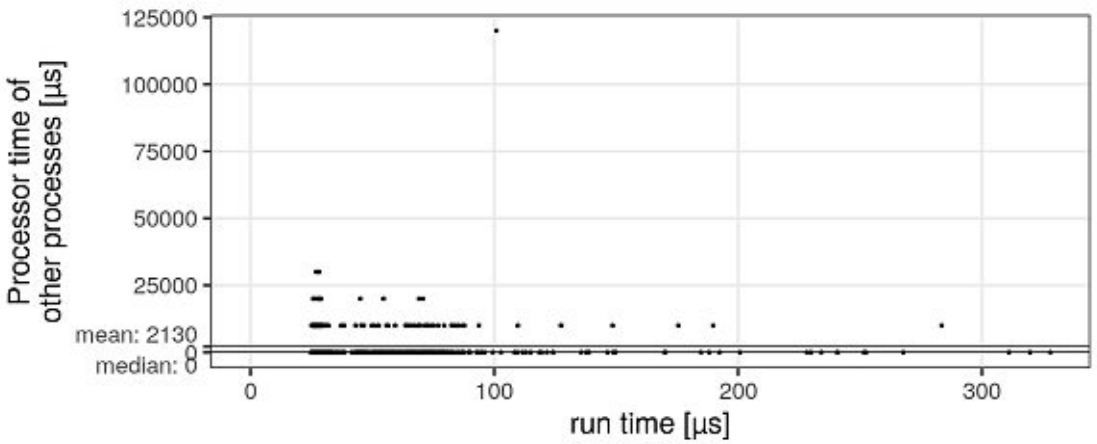
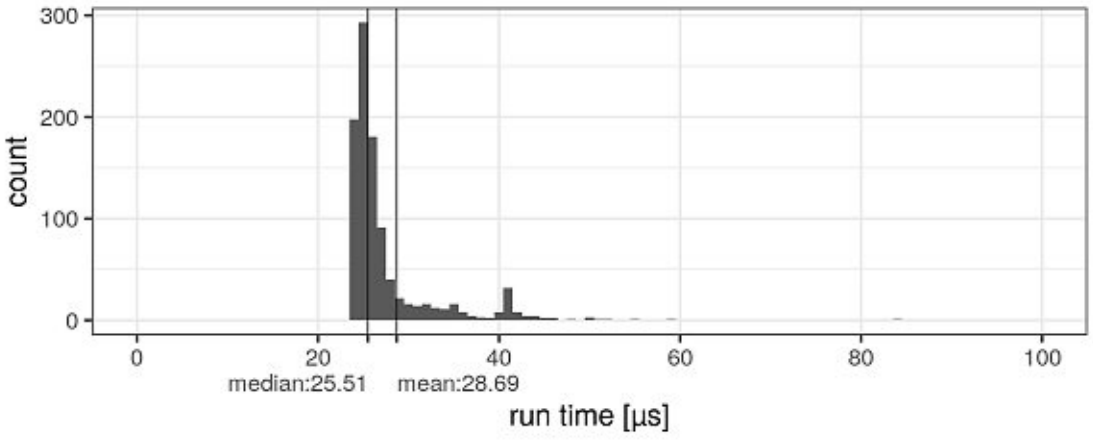
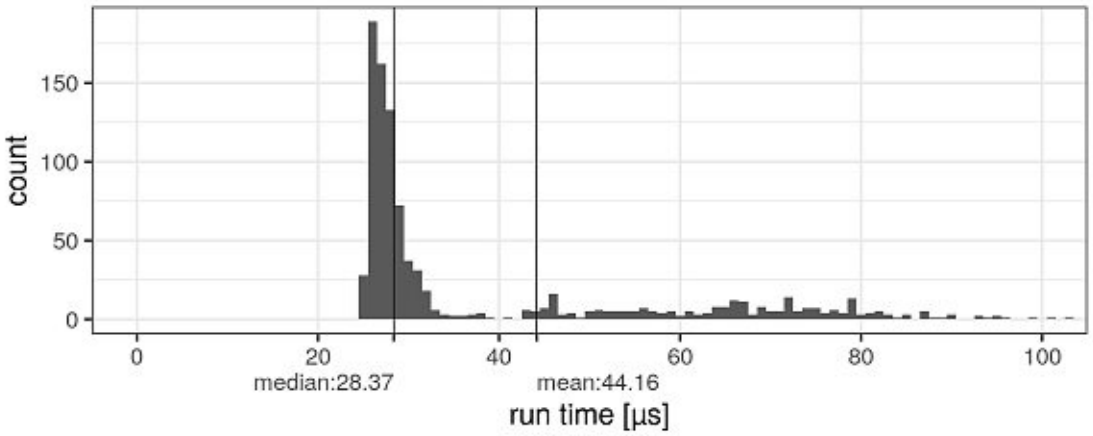


Figure 4.5: The processor time attributed to other processes during each repetition compared to the run time of that repetition. Data for 1000 repetitions of 16 × 16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*.



(a) Run time distribution without additional measurements. (Also see Figure 4.1a)



(b) The run time distribution with measurements using the `proc` filesystem.

Figure 4.6: Run time distribution for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra* with measurements using the `proc` filesystem switched on. Only run times $\leq 1000\,\mu s$ are shown. The comparison between Figure 4.6a and Figure 4.6b shows that reading on the `proc` filesystem introduces major perturbation.

4.5 Categorization of Repetitions

During our measurements, we noticed that there might be different causes for different run times. Consequently, we tried to categorize iterations into different categories.

- **first**: The first repetition naturally has a lot of cache misses and other counters seemed to be unreasonable high too, but the overall run time was not as worse as during other outliers. This leads to misrepresentation of the scale on various plots.
- **warmup**: The runs on *Hydra* showed a rather specific elevated run time region for the first approximately 60 repetitions. This seems to point to cache misses as a reason.

- **outliers**: Some runs have very high run times and are very obvious on the run times per repetition plot but insignificant on the histogram showing the distribution of run times.
- **good**: The other repetitions which do not fall into the above categories.

While determining the first iteration is trivial, it is not as trivial to find the number of iterations of the warm-up phase. Looking at a graph displaying the runtime per iteration, it is easy to see when this cliff happens, but it is not so trivial to compute this cut-off. We tried multiple metrics to find this cliff as reliable as possible.

To find the cliff, we need a type of change detection. We use the following difference function,

$$\delta_{\text{metric},j} = \begin{cases} 0, & \text{if } j = 0. \\ \text{metric}_j - \text{metric}_{j-1}, & \text{otherwise.} \end{cases} \tag{4.1}$$

where "metric" is one of the metrics we specify later and j is the index in the metric array to find a change in a metric. This δ should have a relatively high negative value at the iteration where the cliff ends. Using it directly on the run time values is too sensitive to outliers and therfore it is used on statistical features. Figure 4.7 shows the difference function of the various metrics.

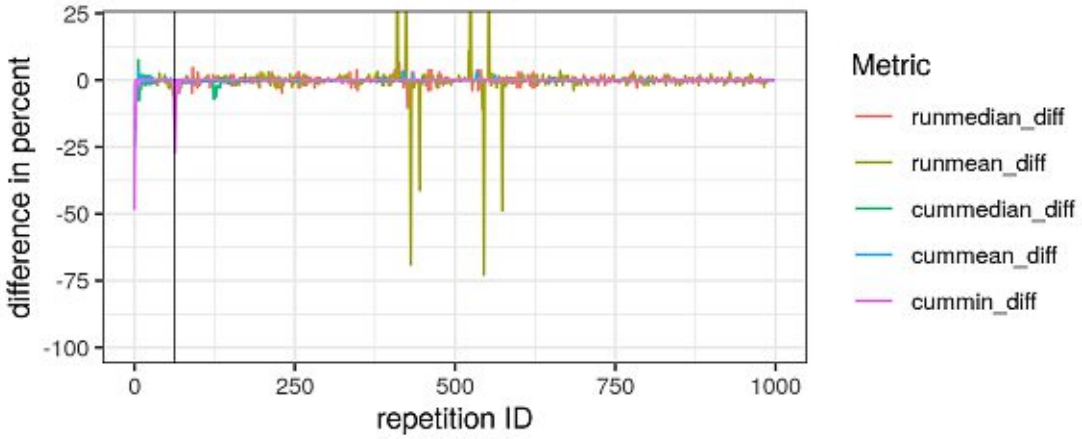


Figure 4.7: $\frac{\delta}{\tilde{x}}$ in percent where $\tilde{x}$ is the run time median for each metric for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The vertical line represents the end-of-warm-up iteration w. The warm-up iterations i are those iterations where $0 < i < w$.

The first metric we tried was the cumulative mean. It is calculated using

$$\text{cummean}_j = \frac{\sum_{i=0}^{j} x_i}{j+1}, \tag{4.2}$$

where x_i is the maximum run time of the i-th iteration. This represents the run time mean of all iterations until this iteration. As we can see in Figure 4.8, this metric does not transport the cliff well, but it illustrates how problematic the warm-up phase is for the mean as a good statistical measure.

The cumulative median function

$$\text{cummedian}_i = \text{median}\,(x_0, \ldots, x_i) \tag{4.3}$$

actually has a steeper cliff, but this cliff is at about double the iteration it should be (see Figure 4.8). This also indicates that the median will be wildly off if the number of measurements is not at least double the number of iterations the system needs to warm up.

To find a metric which changes at the right iteration, we used a running mean and running median

$$\text{runmean}_j = \frac{\sum_{i=j-\frac{k-1}{2}}^{j+\frac{k-1}{2}} x_i}{k} \quad \text{, and} \tag{4.4}$$

$$\text{runmedian}_j = \text{median}\left(x_{j-\frac{k-1}{2}}, \ldots, x_j \ldots .., x_{j+\frac{k-1}{2}}\right) \quad . \tag{4.5}$$

In these two equations, k represents the width of the window which is slid along the iterations. The value at index j, represents the mean or median respectively of the surrounding window where j is in the center. If the window sticks out left or right of the data, it uses the outer-most value for the unknown values of x_i. In Figure 4.8 we used $k = 21$ and we can see while performing better at the initial cliff, these two metrics where quite sensitive to the outliers. This is even more pronounced in Figure 4.7 where we can see the difference function on top of the metrics. Additionally, the cliff is not as steep, and therefore harder to detect with our difference function.

Finally, the cumulative minimum function

$$\text{cummin}_i = \text{minimum}\,(x_0, \ldots, x_i) \tag{4.6}$$

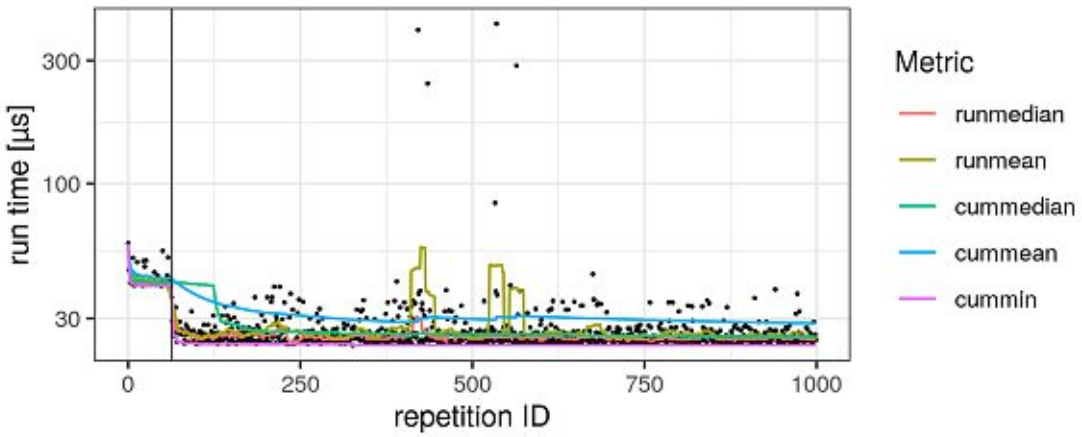
proved to be very robust in terms of outliers, since outliers naturally only occur with higher run times and do not have lower ones. Additionally, it immediately reacts to the cliff and represents it very steep. Another advantage is that it can be computed online in contrast to the running averages, since it only needs values from past iterations. This can be useful when designing a benchmark which eliminates this kind of warm-up values during execution.

We define w as the index where warm-up ends. It is smallest $w \in \mathbb{N}$ where $w \geq 10$ and $\delta_{\text{cummin},w} \leq -0.1 \cdot \tilde{x}$ where $\tilde{x}$ is the median of the run time. The 10 % of the median $\tilde{x}$ were chosen because we obtained the most reliable detection with it. We define every iteration i where $0 < i < w$ as part of the warm-up.

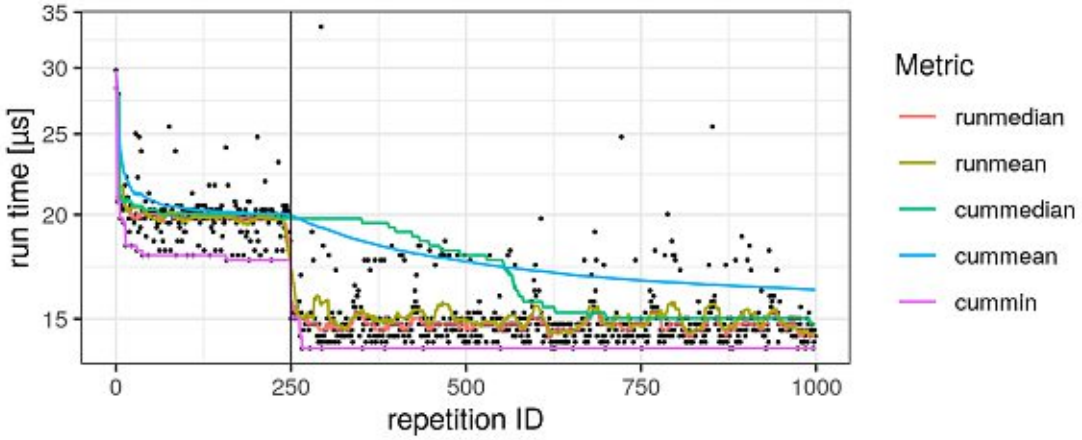
The outliers were defined as the iteration where the run time was at least double the median of all run times. This again is based on trial and error to find a good value.

Figure 4.9 shows the run time per repetition as well as the distribution of run times and is colored using this categorisation. Figure 4.9c implies that the second bump in the histogram is induced by the warm-up phase (approximately 60 iterations).

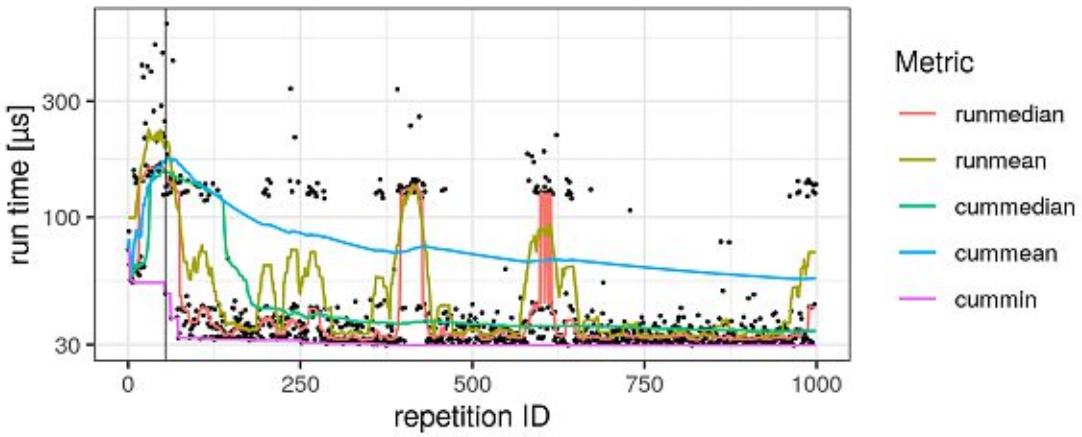
Another conclusion we drew from the categorized plots is that the outliers, while very visible in the plot per repetition ID, hardly pop up in the histogram plots. Therefore, they do not have a big influence on the overall run time in this case.



(a) Run time and metrics for 16×16.

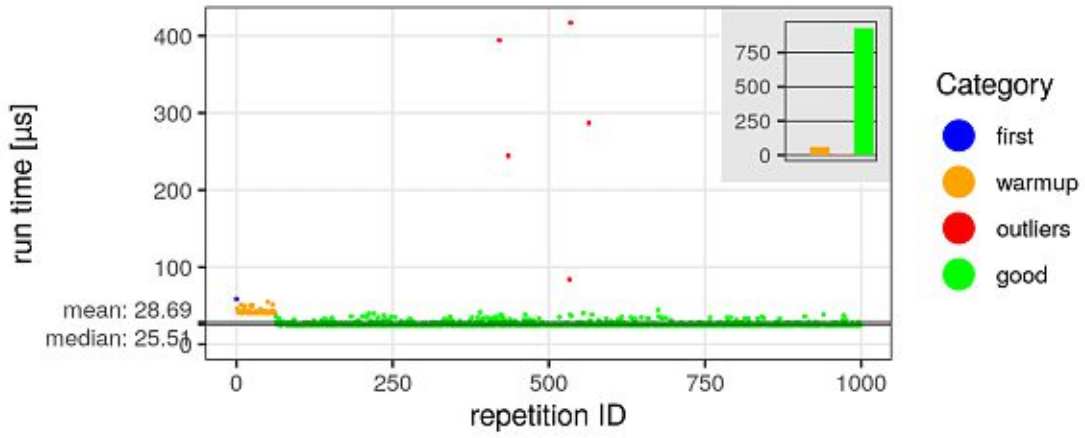


(b) Run time and metrics for 16×2.

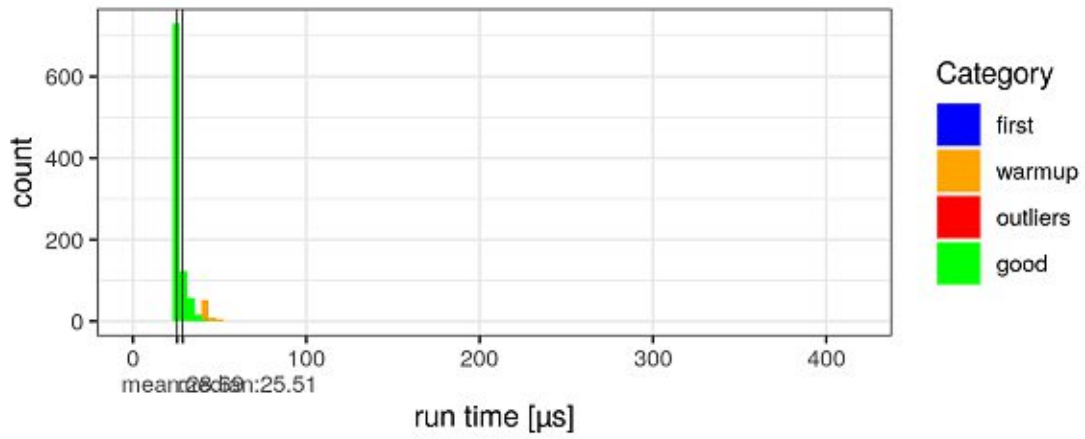


(c) Run time and metrics for 16×32.

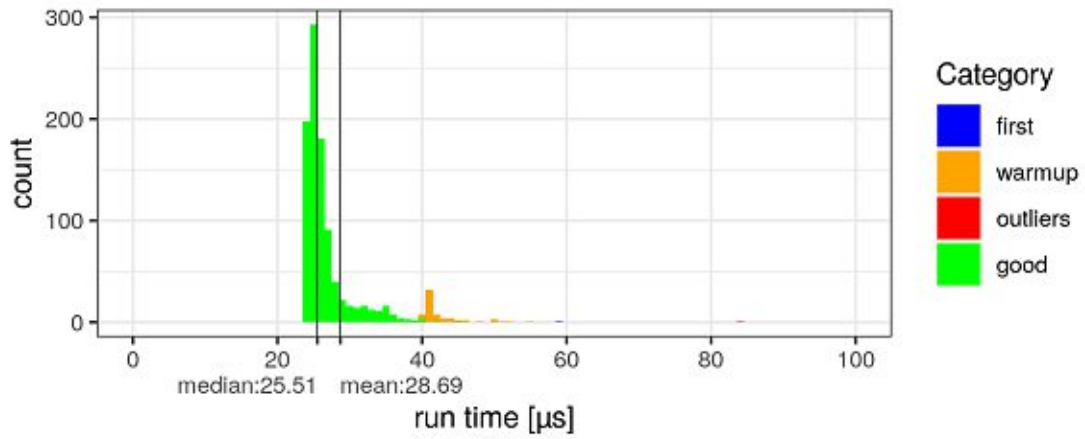
Figure 4.8: Run times and metrics for categorization for 1000 repetitions, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. Window width $k = 21$ for all running averages. The vertical line represents the end-of-warm-up iteration w. The vertical run time axis is in logarithmic scale to make the metrics more visible while still plotting the outliers.



(a) Run time per repetition



(b) Run time distribution



(c) Run time distribution where run time $\leq 200\,\mu s$

Figure 4.9: Run times categorized for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The outliers are not visible in Figure 4.9b because the resolution is too low.

4.6 Communication Introspection

Since we assumed that the run time problem is located in the communication between the processes, we need a way to locate the lines of code, where the difference between warm-up and good runs manifests. Therefore, we need tools that can measure run time in any depth of the call stack.

4.6.1 Call-path Analysis with HPCToolkit

HPCToolkit was used to find the hot path down the call stack, which is the path that is responsible for a considerable amount of run time. Unfortunately, this tool does not provide any possibility to add regions or markers to the code because it operates on and analyzes the fully optimized binary without adding any additional code. Therefore, it was not possible for us to do measurements for single iterations using this tool. Nevertheless, we could follow the call path of the send method below the point which was reachable with MPI_T pvars. We found out that the most time of the run is used in polling in the `PSM2` library (see Figure 4.10). Unfortunately, it was not clear if this is also the reason for the different run time for different repetitions and what the callstack looks like beyond the borders of the `PSM2` library. This is caused by an incomplete analyzis performed by HPCToolkit. Unfortunately, we could not fix this incomplete structuring.

4.6.2 Designing the Novel Library `roth_tracing`

Since all existing tools we tried ultimately failed us in either granularity or flexibility, we implemented our own small tracing library called `roth_tracing`.

This library should be able to

- increment counters and start and stop timers at every part of the software stack where the source code is available,
- start and stop timers and accumulate the time spent between start and stop for one iteration,
- support iterations and save the counters and timers according to their iteration,
- light weight saving of the timers and counters in simple arrays (no structs, no multi-dimensional arrays) to limit the influence on the memory usage,
- initialize the data structures needed to save the measurements in advance to avoid perturbation in the measured parts,
- retrieval of the counter and timer data as arrays (no gather logic to avoid cyclic dependencies to MPI), and
- translation of counter and timer enumerations to strings for easily readable output files.

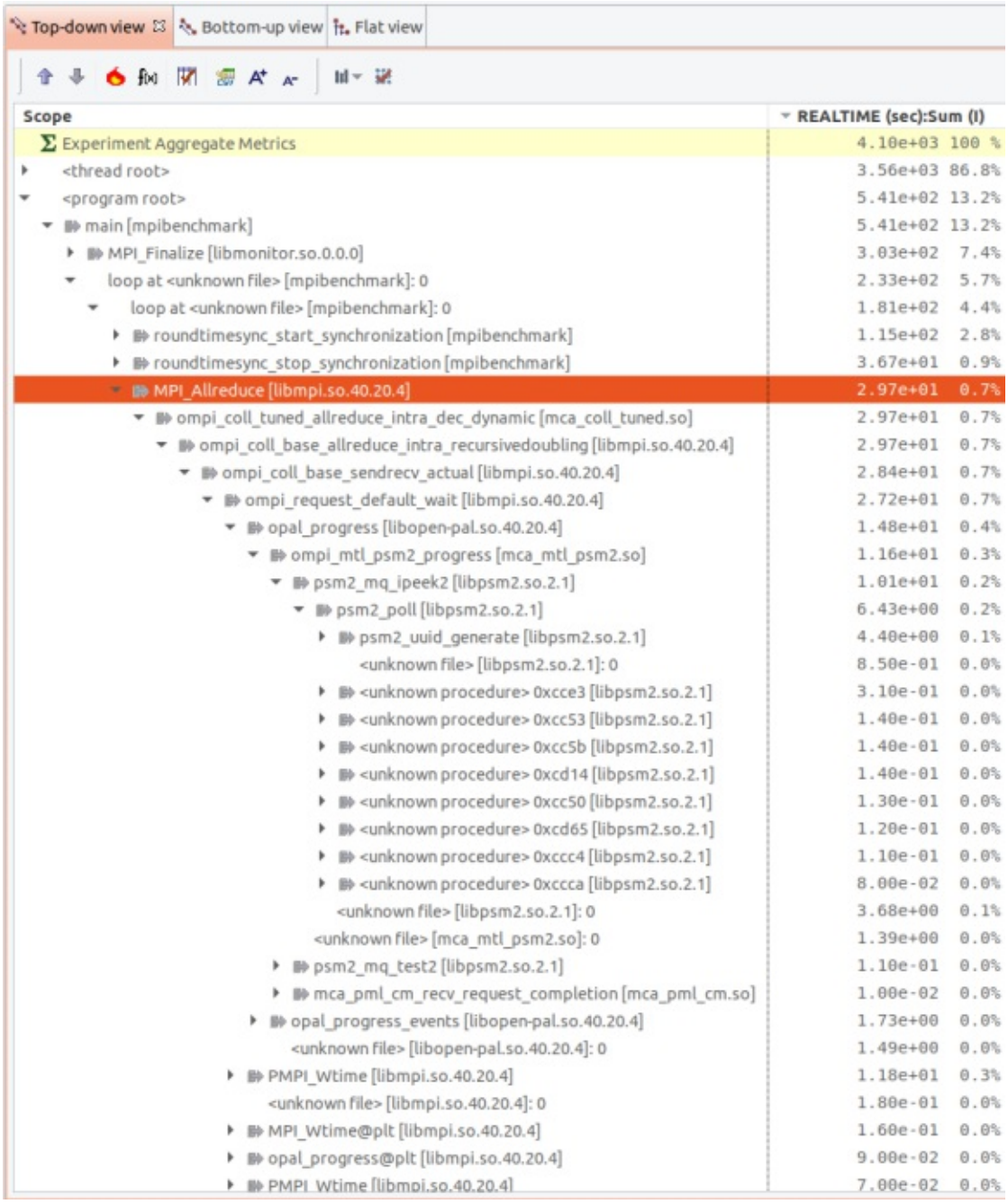


Figure 4.10: A screenshot of the top down analysis for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra* in the `hpcviewer` of HPCToolkit. The path for the measured `MPI_Allreduce` operation is expanded at the parts which took up the most time during execution.

This small library consists of some simple functions and multiple underlying arrays.

`void init_tracer(long number_of_repetitions)`
Allocates the arrays to store the measurements.

`void roth_tracing_start_repetition(long set_rep_id)`
Resets the measured variables for this repetition, sets the current repetition, and activates the measurement functions.

`void roth_tracing_increment_counter(counter_metric metric_id)`
Increments the counter specified by the enum counter_metric by one.

`void roth_tracing_start_timer(time_metric metric_id)`
Saves the current wall-clock-time into a temporary array as the start time for this time metric.

`void roth_tracing_stop_timer(time_metric metric_id)`
Adds the difference of the start time of the given metric and the current wall-clock-time to the time measurement for the current repetition and the given metric.

`int *roth_tracing_get_counter_data(void)`
Returns the pointer to the data measured for the counter metrics.

`double *roth_tracing_get_time_data(void)`
Returns the pointer to the data measured for the time metrics.

`int get_number_of_counter_metrics(void)`
Returns the number of counter metrics which are measured.

`int get_number_of_time_metrics(void)`
Returns the number of time metrics which are measured.

`const char *get_counter_metric_name(counter_metric metric_id)`
Returns the name of a counter metric as a string.

`const char *get_time_metric_name(time_metric metric_id)`
Returns the name of a time metric as a string.

As we can see in the working schema in Figure 4.11, the functions for initialization, starting and stopping of repetitions, and `get_...`-functions were used in ReproMPI to set up the measurement environment and print out the data in the end. The functions for starting and stopping timers as well as for incrementing counters were used in different parts of Open MPI where the MPI_T pvars were not usable anymore because it was too far down the call stack.

The counters were hard coded in the library as enumerations. This is not a very flexible approach but has the advantage that the access times are much lower than a search by string or another dynamic approach and for this work, this was flexible enough.

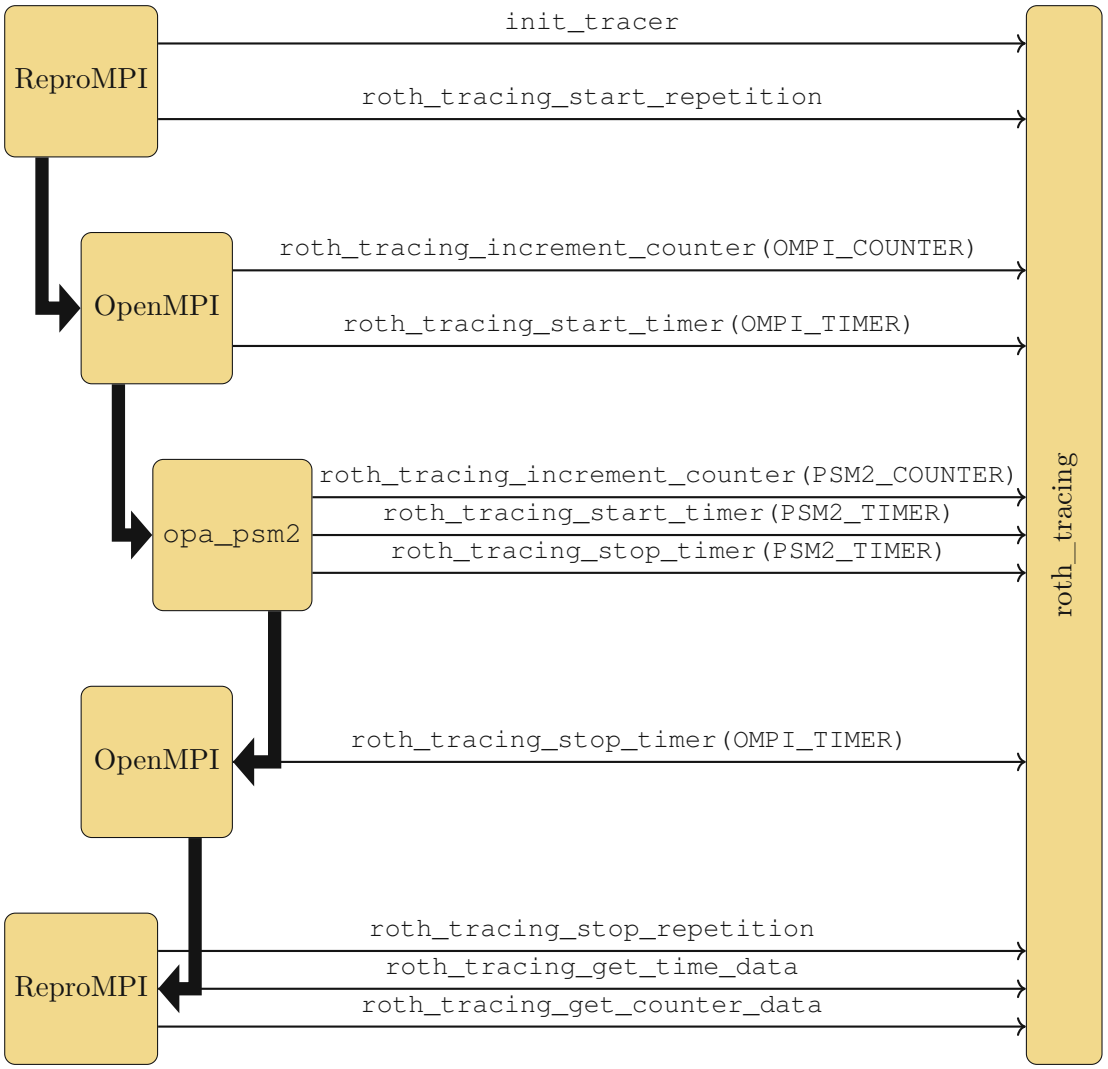


Figure 4.11: Working scheme of the `roth_tracing` library.

4.6.3 Analyzing the `PSM2` library

Additionally, the `roth_tracing` library was used in the Intel® Performance Scaled Messaging 2 (PSM2) API to get even further down into the underlying messaging layer of the Intel® OPA. Therefore, we checked out the tag `PSM2_11.2.185` of Intel®'s opa-psm2 GitHub repository [35] and added additional measurement code.

After an incremental measurement of times further down the call stack, we eventually came across the method `am_ctl_getslot_pkt_inner` in which the flag of the packet struct is checked and set (see Listing 4.4).

These two operations, check and set, are very clearly correlated with the run time increase in the warm-up phase (see Figure 4.12). This increase of access time for these read and write operations can only be linked to cache misses. When we revisit the cache misses measured by PAPI and color them according to the categorization presented in Section 4.5, we see a slight increase during the warm-up phase in level 3 cache misses as

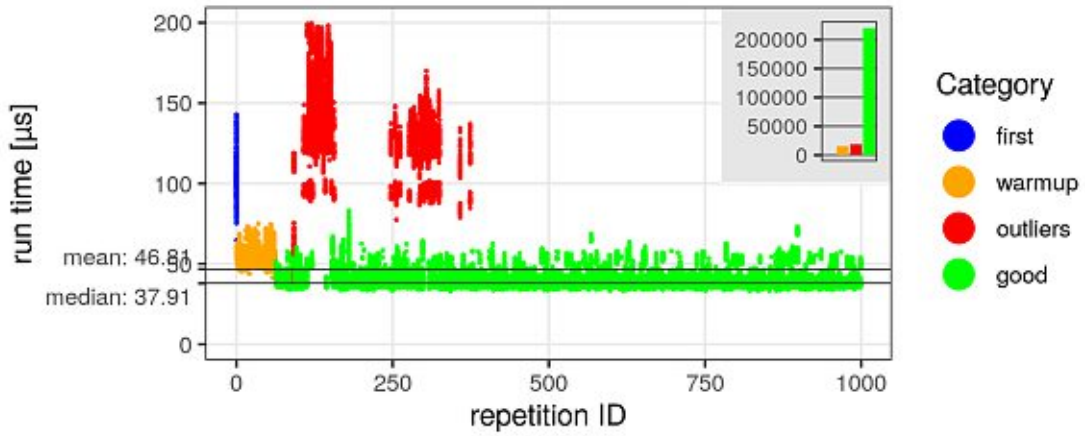
Listing 4.4: The culprit of the run time variability on *Hydra* 16×16 processes 1000 B `MPI_Allreduce`

```
PSMI_ALWAYS_INLINE(
am_pkt_short_t *
am_ctl_getslot_pkt_inner(volatile am_ctl_qhdr_t *shq, am_pkt_short_t *pkt0))
{
    roth_tracing_start_timer(AM_CTL_GETSLOT_PKT_INNER_TIME);
    am_pkt_short_t *pkt;
    uint32_t idx;
#ifndef CSWAP
    pthread_spin_lock(&shq->lock);
    idx = shq->tail;
    pkt = (am_pkt_short_t *) ((uintptr_t) pkt0 + idx * shq->elem_sz);
    roth_tracing_start_timer(AM_CTL_GETSLOT_PKT_INNER_CHECK_QFREE_TIME);
    uint_fast32_t is_qfree = pkt->flag == QFREE;
    roth_tracing_stop_timer(AM_CTL_GETSLOT_PKT_INNER_CHECK_QFREE_TIME);
    if (likely(is_qfree)) {
        ips_sync_reads();
        roth_tracing_start_timer(AM_CTL_GETSLOT_PKT_INNER_SET_QUEUE_USED_TIME);
        pkt->flag = QUSED;
        roth_tracing_stop_timer(AM_CTL_GETSLOT_PKT_INNER_SET_QUEUE_USED_TIME);
        shq->tail += 1;
        if (shq->tail == shq->elem_cnt)
            shq->tail = 0;
    } else {
        pkt = 0;
    }
    pthread_spin_unlock(&shq->lock);
#else
    [...]
#endif
    roth_tracing_stop_timer(AM_CTL_GETSLOT_PKT_INNER_TIME);
    return pkt;
}
```

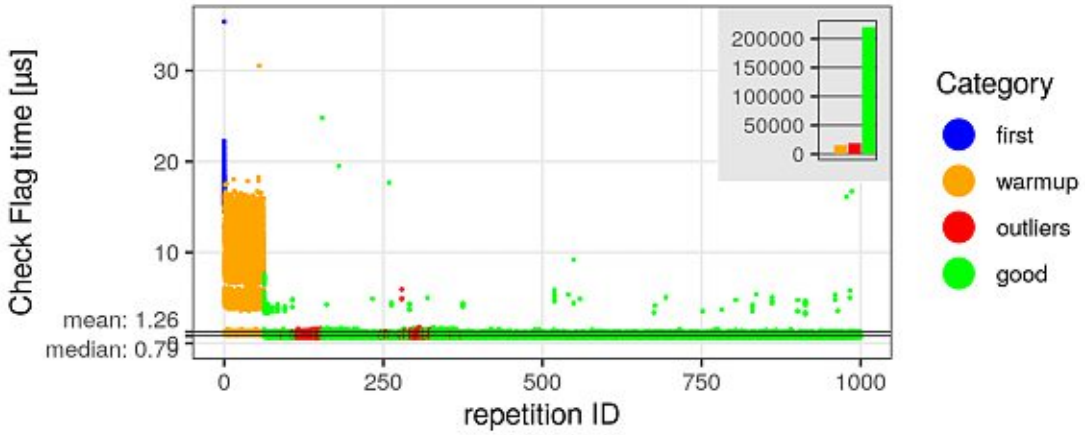
shown in Figure 4.13. While hardly noticeable in Figure 4.13b because of the outliers and first repetition, Figure 4.13c shows that there is a correlation between the number of cache misses when we compare warm-up and good repetitions. This leads to the conclusion that the position of the cache misses is crucial for the run time performance in this special case.

A lot more cache misses did not increase the run time by the same ratio, and far longer run times did not necessarily have more cache misses. While this was the reason we discarded cache misses as the cause for our run time variability in the first place (see Section 4.2.2), we now see that they are part of the cause in the warm-up phase of the run.

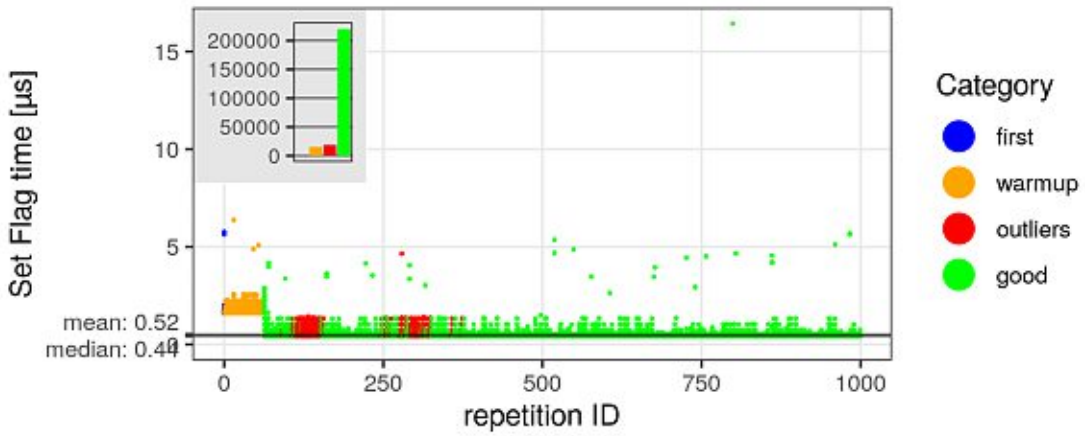
To verify if the shared memory access of multiple processes is not only correlated to the increase of run time in the warm-up phase but causes it, we compared the histograms of one node with 32 processes and 32 nodes with one process per node in Figure 4.14. As we can see, the warm-up cluster completely disappears if the processes are not sharing memory. In Figure 4.15 we can see that even adding one additional process on each node reintroduces a warm-up phase with elevated run time in the beginning. This warm-up phase does not add as much overhead as 32 processes on one node, but the warm-up phase lasts longer (approximately 250 iterations with 16×2 processes vs. approximately 60 iterations with 1×32 processes). We conclude that the problem which induces this run time variability is located in the shared memory communication and caused by cache misses in two specific lines of code. We were not able to fix this issue on our own, but it is perhaps possible to fix it.



(a) Run time per repetition. Only run times $\leq 200\,\mu s$ are shown.

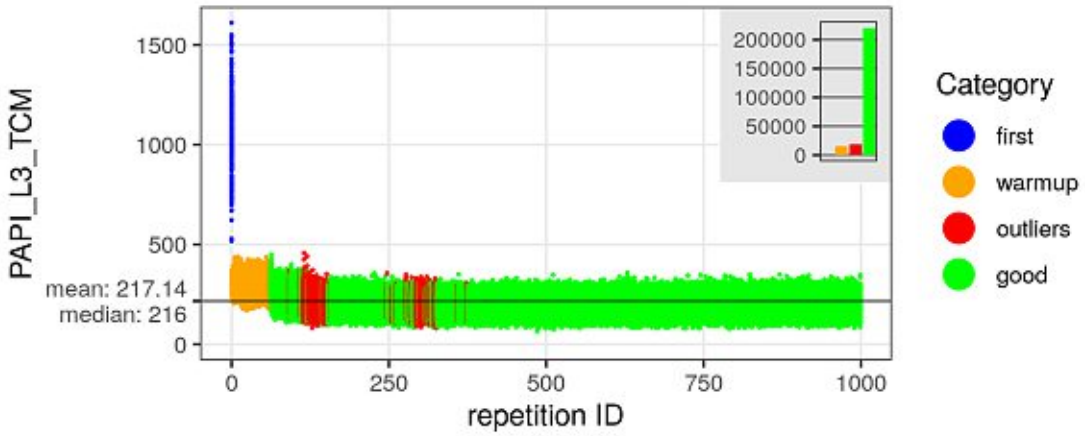


(b) Run time for checking the flag

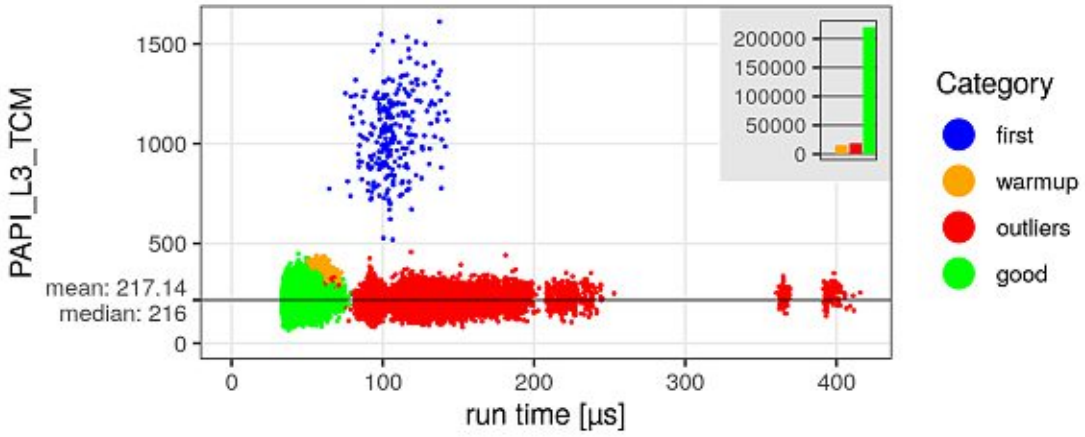


(c) Run time for setting the flag

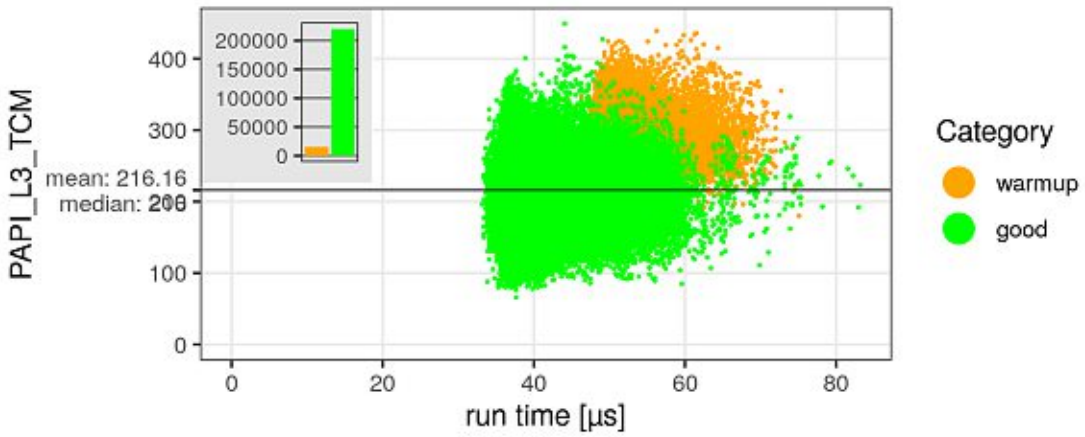
Figure 4.12: Run Time for checking and setting the packet flag compared to the overall run time for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The histograms in the corner represent the number of data points attributed to the respective category.



(a) Level 3 cache misses per repetition

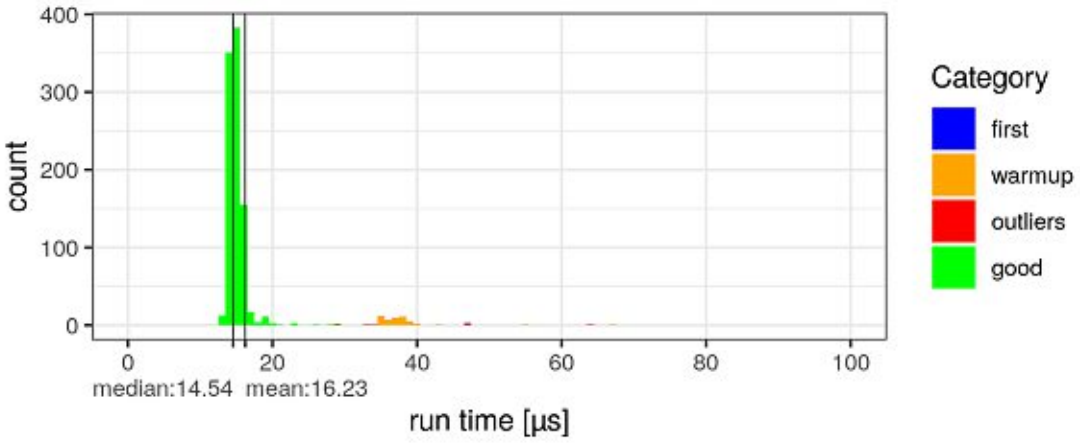


(b) Level 3 cache misses compared to run time

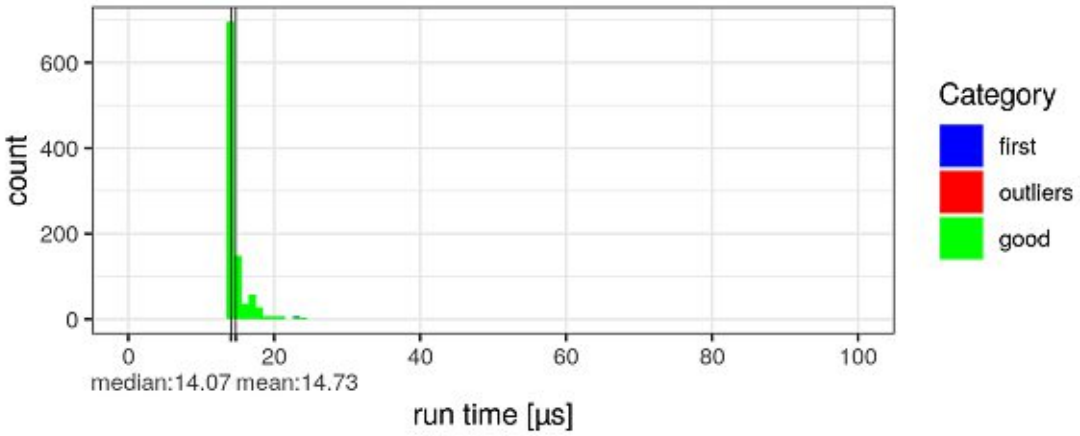


(c) Level 3 cache misses compared to run time without outliers and first repetition

Figure 4.13: Level 3 cache misses per repetition and run time with categorization applied for 1000 repetitions of 16 × 16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The histograms in the corner represent the number of data points attributed to the respective category.

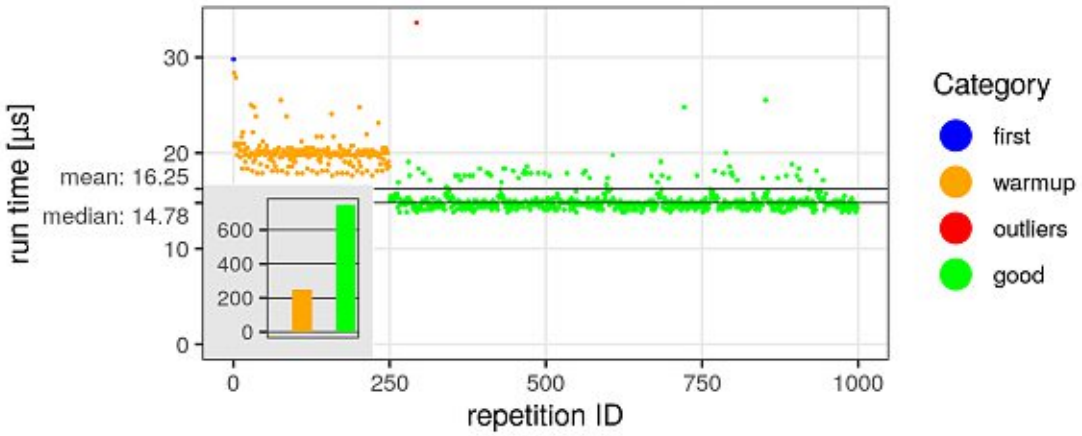


(a) Run time histogram for 1×32 processes.

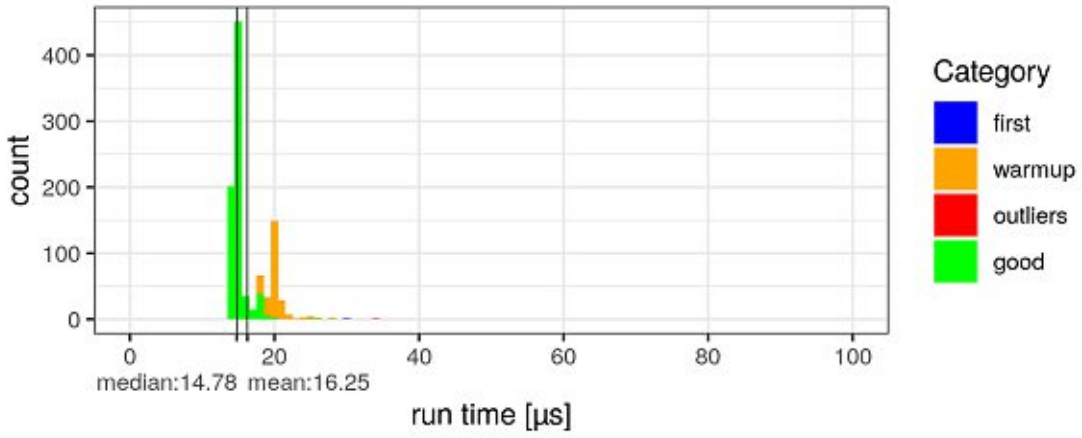


(b) Run time histogram for 32×1 processes.

Figure 4.14: The comparison of the run time histograms for 1000 repetitions of 1×32 and 32×1 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*.



(a) Run time per repetition.



(b) Run time histogram.

Figure 4.15: Run time for 1000 repetitions of 16×2 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The histogram in the corner represents the number of data points attributed to the respective category.

4.7 How to Deal with Run Time Variability?

After we described how we can detect and pin down individual causes of run time variability, we now want to give advice on how to deal with the found variability. The main goal is to make the measurement of run time of MPI operations more consistent and reproducible. As we can see in various examples throughout this chapter, the mean is a very bad measure for how fast a given implementation is. It is easily influenced by the outliers which occur occasionally and likely do not depend on the implementation which should be tested.

Especially for short runs, the median is a bad choice too because it is also influenced by the warm-up phase and does not represent the run time in the long run. Instead of choosing one value as the result of the benchmark, it would be far more expressive if the measured data would be visually presented using histograms and time per repetition graphs. The portable profiling infrastructure IPM [36] also uses visualization to visualize the performance of various benchmarks. This way, it can also easily be seen if the run time variations of the different iterations only consist of occasional outliers, clustered chunks and if they only appear in the beginning or at specific time intervals. Another representation which would come in handy for some analyses is a plot showing run time and ranks. This would be especially useful if the performance variability is only caused by single nodes or processes.

Finding clusters of different run times would also be a good feature for a MPI micro-benchmark. These clusters could help to find the underlying performance problems by giving a clue of the number of different run times and their proportions.

To get a good overview over the results of a micro-benchmark, we propose the MPI Micro-Benchmark Fingerprint (MPI-MiBFi). It should act as a fingerprint for a specific micro-benchmark. There are multiple plots and values included in this fingerprint, namely

- the operation which is benchmarked,
- the message size used for the benchmark,
- the number of nodes and tasks per node,
- the machine used for the benchmark,
- a list of nodes used,
- the configuration of the micro-benchmark tool,
- the configuration and version of the MPI implementation,
- statistical values like various quantiles, minimum, maximum, mean and median,
- a Box Plot of the benchmark,
- an Empirical Cumulative Distribution Function (ECDF) plot of the runtimes,
- the run times per repetition with all data and a version without the outliers, and
- a run time histogram with all data and a zoomed version without the outliers.

Figures 4.16, 4.17, and 4.18 show examples of MPI-MiBFis. As we can see, Figures 4.16 and 4.17 present results of the same experiment (for 10 000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*). Figure 4.18 shows the results of a quite similar experiment but with the Rabenseifner's `MPI_Allreduce` algorithm (`OMPI_MCA_coll_tuned_allreduce_algorithm=6`) used instead of Recursive Doubling (`OMPI_MCA_coll_tuned_allreduce_algorithm=3`).

The comparison of the results shows that the two experiments for Recursive Doubling are virtually identical, while the one with Rabenseifner's algorithm differs. We can see that while Rabenseifner's algorithm has a higher median (26.94 µs and 27.18 µs vs. 30.28 µs), it has a lower 95 %-quantile (69.38 µs and 69.62 µs vs. 68.19 µs). This is also obvious if we look at the ECDF plots. The first steep rise in Rabenseifner's algorithm's ECDF plot occurs later than for the one for Recursive Doubling, but it goes up further. This suggests that the Rabenseifner algorithm, while being slower on average, performs better under whichever condition is introducing the outliers. This shows that assumptions based on single statistical values can be misleading in finding superior algorithms or implementations. Therefore, it is useful to provide multiple statistical feature as well as graphical representations of this data.

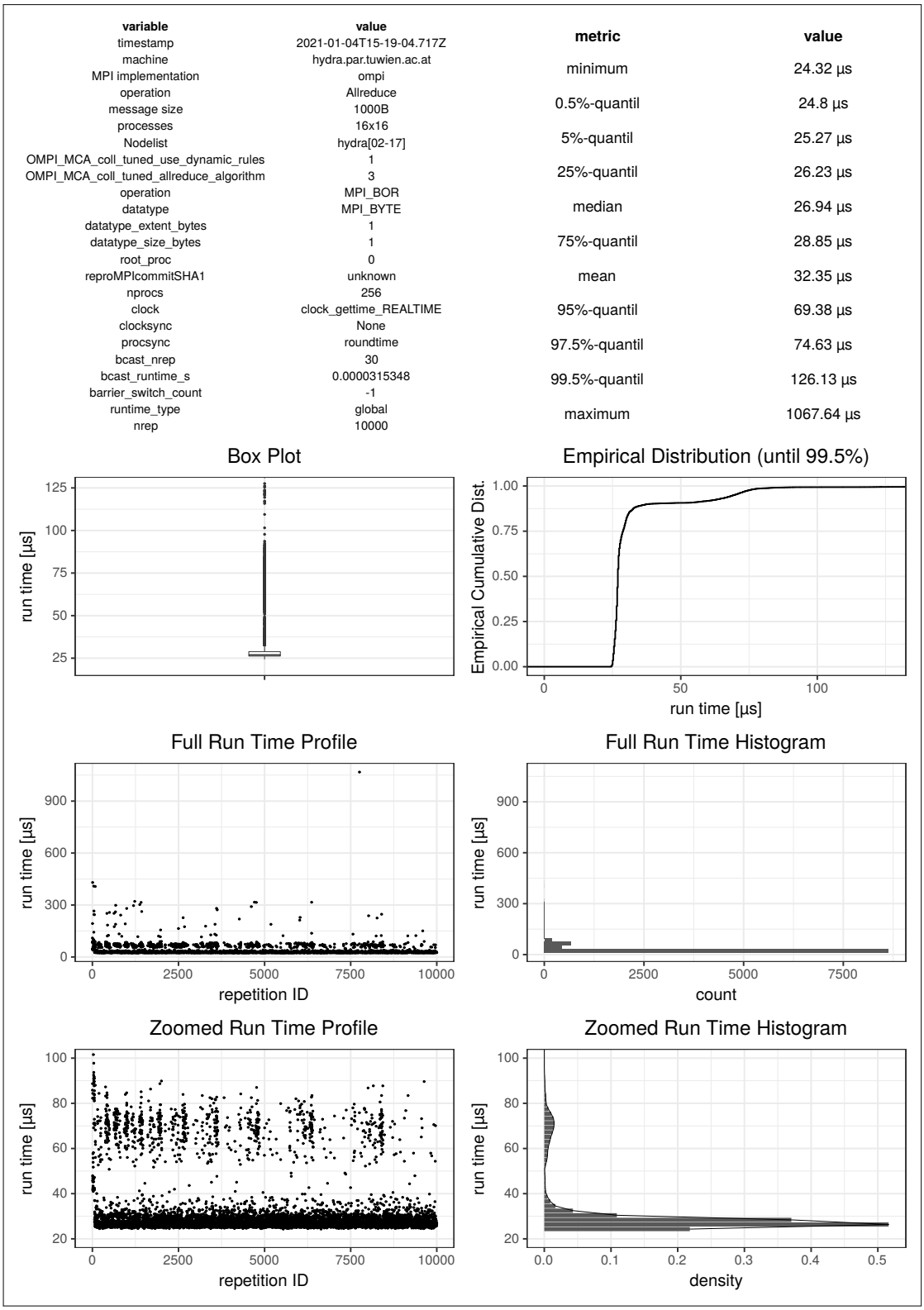


Figure 4.16: An example MPI-MiBFi for 10 000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*.

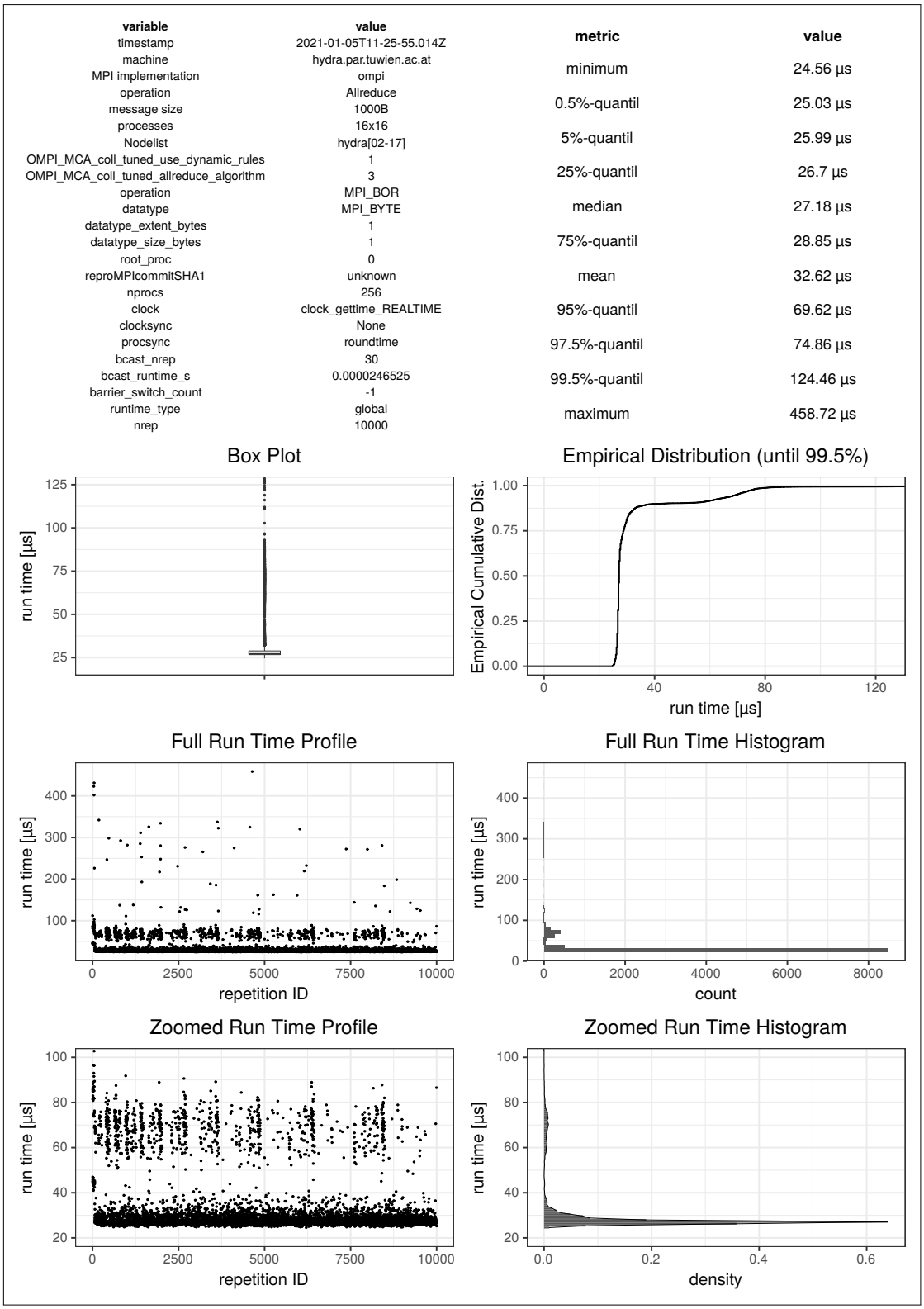


Figure 4.17: An example MPI-MiBFi for 10 000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*.

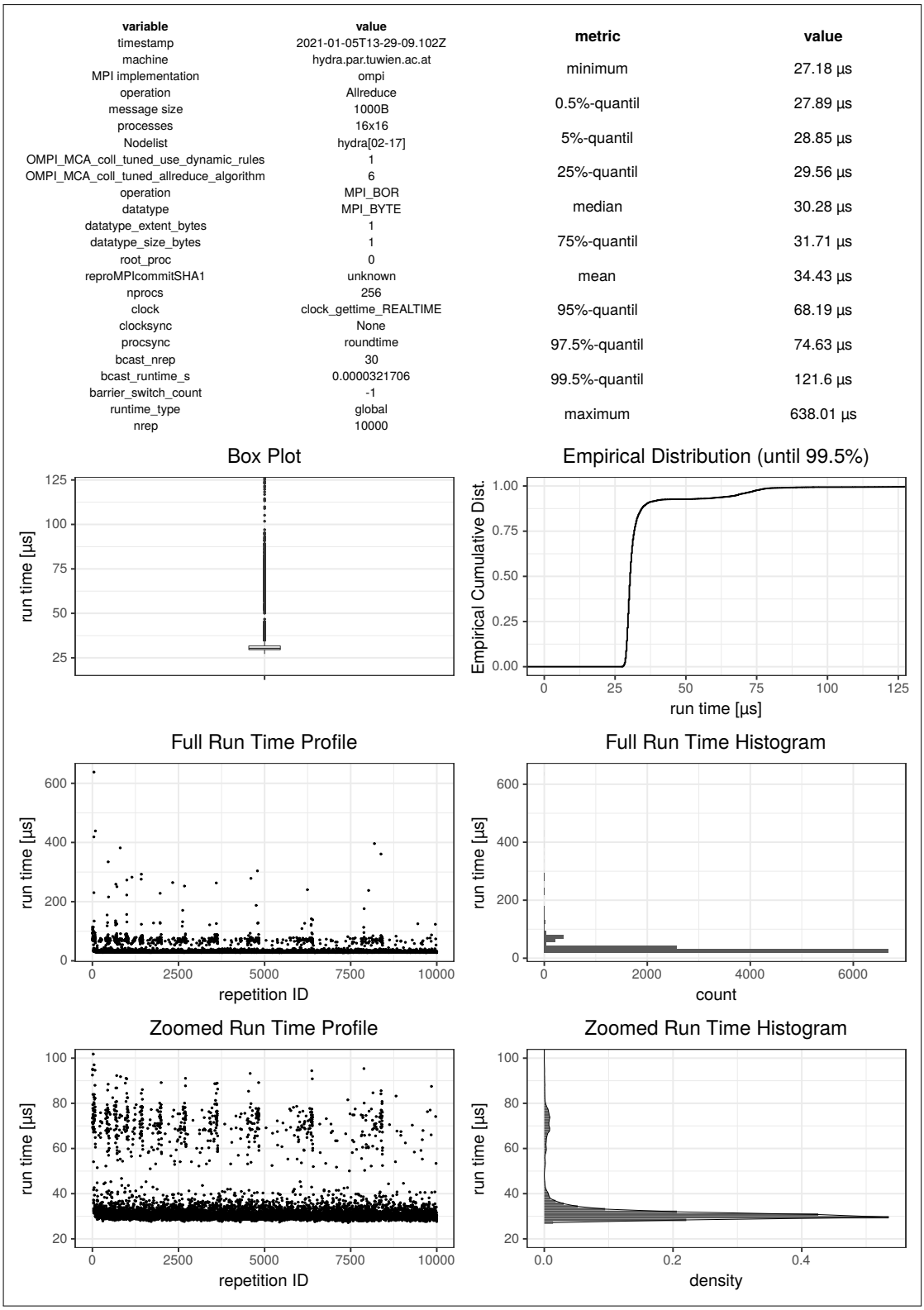


Figure 4.18: An example MPI-MiBFi for 10 000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Rabenseifner) on *Hydra*.

CHAPTER 5

Conclusion

To find the reasons for run time variability in HPC, we first measured the run time of MPI collectives using ReproMPI and visualized them using histograms. This showed that the run time variation is clustered around specific run times, and we figured that this could be related to cache misses.

We then decided to measure hardware performance counters using `LIKWID` and PAPI. Unfortunately, `LIKWID` induced more overhead and hence distorted the results. Comparing the cache misses to the run time of each repetition and rank, we could not find conclusive evidence that the cache misses are the main reason for the clustered run time.

Afterwards, we tried to find the causes for the variability using SPCs and especially wanted to check if the run time variability is caused by the dynamic use of different implementations of the collective operation. We used the already implemented SPCs from Open MPI as well as our newly implemented ones using the MPI_T pvars. We had to implement new ones because the existing ones were too coarse and did not give us a deeper insight into the reasons of the increased run time. Using these counters, we found that the algorithms did not change between repetitions and that the elevated run time was correlated with the communication part of the operation, but we were limited by the maximum reach of the Open MPI SPCs and could not examine it further without additional tools.

To get even further down the call stack, we used HPCToolkit. With this tool, we found which methods used up most of the run time, but we were not able to get into the `PSM2` library and we had no possibility to attribute the run times to single repetitions.

Since all tried methods ultimately failed us by then, we proposed our own rudimentary tracing library which should close the gap with the system calls. Using this library, we were able to attribute the elevated run time to two lines of code in the `PSM2` library which set and checked a specific flag.

Due to the run time being especially elevated in the beginning of a run, we revisited the cache misses and cleared up the plots by applying one of four categories to each repetition, namely, first, warm-up, outliers, and good. We used elaborate statistic functions to determine the end of the warm-up phase. After filtering away the outliers and the first

repetition, it was obvious that the warm-up phase, in which the run time was elevated due to the aforementioned code lines, actually had increased cache misses. This leads to the conclusion that cache misses, especially misses of the flag evaluated by those two code lines, lead to the increased run time of the warm-up phase. This warm-up phase also only seems to happen if processes share memory and not if there is only a single thread running on each shared memory system.

Finally, we proposed the MPI-MiBFi, a reproducible fingerprint which pictures a lot of data about a single micro-benchmark and which makes it easier and more reliable to compare different micro-benchmarks.

In the future, benchmarks and especially micro-benchmarks should not only result in a single value, but should present more comprehensive data on a measurement. Integrated graphs using ASCII or a proper plot engine would make comparing results more versatile and robust. Another improvement which would give deeper insight into the interference of other processes would be a more fine-grained and less intrusive API for querying process information on Linux operating systems. Overall, a lot of research has been done on the benchmarking of HPC operations, but there is still a lot of promising research to be done in the future.

Acronyms

API Application Programming Interface. 8, 12, 14, 15, 21, 39, 52

ECDF Empirical Cumulative Distribution Function. 45, 46

HPC High Performance Computing. xi, xiii, 1, 3–5, 7, 8, 12, 14, 15, 19, 51, 52

ML Machine Learning. 6, 9

MPI Message Passing Interface. 1, 3, 5, 7, 8, 11, 12, 14–16, 21, 25, 27, 36, 45, 51

MPI-MiBFi MPI Micro-Benchmark Fingerprint. xi, xiii, 45–49, 52

MPI_T MPI Tools Information. 8, 15, 25, 36, 38, 51

OPA Omni-Path-Architecture. 12, 13, 39

OTF2 Open Trace Format 2. 7

PAPI Performance Application Programming Interface. 6, 7, 13, 14, 16, 23–26, 39, 51

PML Point-to-Point Management Layer. 25

PSM2 Performance Scaled Messaging 2. 39

PTP Precision Time Protocoll. 12

SPC Software Performance Counter. 8, 15, 25, 27, 29, 51, 59, 73

Bibliography

[1] MPI Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. USA, 1994.

[2] MPI Forum. *MPI: A Message-Passing Interface Standard - Version 3.0*. 2012. URL: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (visited on 11/03/2020).

[3] MPI Forum. *MPI: A Message-Passing Interface Standard - Version 3.1*. June 4, 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 01/20/2021).

[4] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. „Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation“. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack J. Dongarra. Vol. 3241. Lecture Notes in Computer Science. Springer, 2004, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19.

[5] Sascha Hunold and Alexandra Carpen-Amarie. „Reproducible MPI Benchmarking is Still Not as Easy as You Think“. In: *IEEE Trans. Parallel Distributed Syst.* 27.12 (2016), pp. 3617–3630. DOI: 10.1109/TPDS.2016.2539167.

[6] Sascha Hunold and Alexandra Carpen-Amarie. „On the Importance of Data Quality when Tuning MPI Libraries“. In: *Austrian HPC Meeting 2019-AHPC19*. 2019, p. 15.

[7] David Skinner and William Kramer. „Understanding the causes of performance variability in HPC workloads“. In: *Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE. 2005, pp. 137–149. DOI: 10.1109/IISWC.2005.1526010.

[8] Abhinav Bhatele, Kathryn Mohror, Steve H. Langer, and Katherine E. Isaacs. „There goes the neighborhood: performance degradation due to nearby jobs“. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*. Ed. by William Gropp and Satoshi Matsuoka. ACM, 2013, 41:1–41:12. DOI: 10.1145/2503210.2503247.

[9] James E. Smith. „A Study of Branch Prediction Strategies“. In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. Ed. by Gurindar S. Sohi. ACM, 1998, pp. 202–215. DOI: 10.1145/285930.285980.

[10] Andreas Gocht, Robert Schöne, and Mario Bielert. „Q-Learning Inspired Self-Tuning for Energy Efficiency in HPC“. In: *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019*. IEEE, 2019, pp. 344–347. DOI: 10.1109/HPCS48598.2019.9188112.

[11] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. „PAPI: A portable interface to hardware performance counters“. In: *Proceedings of the Department of Defense HPCMP users group conference*. Vol. 710. 1999.

[12] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. „Collecting Performance Data with PAPI-C“. In: *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Ed. by Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel. Berlin, Heidelberg: Springer, 2009, pp. 157–173. ISBN: 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_11.

[13] Jan Treibig, Georg Hager, and Gerhard Wellein. „LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments“. In: *39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010*. Ed. by Wang-Chien Lee and Xin Yuan. IEEE Computer Society, 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[14] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. „HPCTOOLKIT: tools for performance analysis of optimized parallel programs“. In: *Concurr. Comput. Pract. Exp.* 22.6 (2010), pp. 685–701. DOI: 10.1002/cpe.1553.

[15] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. „Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir“. In: *Tools for High Performance Computing 2011 - Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011*. Ed. by Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch. Springer, 2011, pp. 79–91. ISBN: 978-3-642-31476-6. DOI: 10.1007/978-3-642-31476-6_7.

[16] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. „PERISCOPE: An Online-Based Distributed Performance Analysis Tool“. In: *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Ed. by Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel. Berlin, Heidelberg: Springer, 2009, pp. 1–16. ISBN: 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_1.

[17] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. „The Vampir Performance Analysis Tool-Set“. In: *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*. Ed. by Michael M. Resch, Rainer Keller, Valentin

Himmler, Bettina Krammer, and Alexander Schulz. Springer, 2008, pp. 139–155. DOI: 10.1007/978-3-540-68564-7_9.

[18] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. „The Scalasca performance toolset architecture“. In: *Concurr. Comput. Pract. Exp.* 22.6 (2010), pp. 702–719. DOI: 10.1002/cpe.1556.

[19] Sameer Shende and Allen D. Malony. „The Tau Parallel Performance System“. In: *Int. J. High Perform. Comput. Appl.* 20.2 (2006), pp. 287–311. DOI: 10.1177/1094342006064482.

[20] Rainer Keller, George Bosilca, Graham E. Fagg, Michael M. Resch, and Jack J. Dongarra. „Implementation and Usage of the PERUSE-Interface in Open MPI“. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*. Ed. by Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack J. Dongarra. Vol. 4192. Lecture Notes in Computer Science. Springer, 2006, pp. 347–355. DOI: 10.1007/11846802_48.

[21] Tanzima Islam, Kathryn Mohror, and Martin Schulz. „Exploring the Capabilities of the New MPI_T Interface“. In: *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*. Ed. by Jack J. Dongarra, Yutaka Ishikawa, and Atsushi Hori. ACM, 2014, p. 91. DOI: 10.1145/2642769.2642781.

[22] Marc-André Hermanns, Nathan T. Hjlem, Michael Knobloch, Kathryn Mohror, and Martin Schulz. „Enabling callback-driven runtime introspection via MPI_T“. In: *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*. ACM, 2018, 8:1–8:10. DOI: 10.1145/3236367.3236370.

[23] David Eberius, Thananon Patinyasakdikul, and George Bosilca. „Using software-based performance counters to expose low-level open MPI performance information“. In: *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI/USA 2017, Chicago, IL, USA, September 25-28, 2017*. Ed. by Antonio J. Peña, Pavan Balaji, William Gropp, and Rajeev Thakur. ACM, 2017, 7:1–7:8. DOI: 10.1145/3127024.3127039.

[24] David Böhme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Giménez, Matthew P. LeGendre, Olga Pearce, and Martin Schulz. „Caliper: performance introspection for HPC software stacks“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. Ed. by John West and Cherri M. Pancake. IEEE Computer Society, Nov. 2016, pp. 550–560. DOI: 10.1109/SC.2016.46.

[25] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim M. Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. „Diagnosing Performance Variations in HPC Applications Using Machine Learning“. In: *High Performance Computing - 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18-22, 2017, Proceedings*. Ed. by Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David E. Keyes. Vol. 10266. Lecture Notes in Computer Science. Springer, 2017, pp. 355–373. DOI: 10.1007/978-3-319-58667-0_19.

[26] Andy B. Yoo, Morris A. Jette, and Mark Grondona. „SLURM: Simple Linux Utility for Resource Management“. In: *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*. Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Vol. 2862. Lecture Notes in Computer Science. Springer, 2003, pp. 44–60. ISBN: 978-3-540-39727-4. DOI: 10.1007/10968987_3.

[27] Richard Cochran, Cristian Marinescu, and Christian Riesch. „Synchronizing the Linux system time to a PTP hardware clock“. In: *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. IEEE, Sept. 2011. DOI: 10.1109/ispcs.2011.6070158.

[28] Karl Pearson. „X. Contributions to the mathematical theory of evolution.—II. Skew variation in homogeneous material“. In: *Philosophical Transactions of the Royal Society of London. (A.)* 186 (Dec. 1895), pp. 343–414. DOI: 10.1098/rsta.1895.0010.

[29] Ronald A. Rensink and Gideon Baldridge. „The Perception of Correlation in Scatterplots“. In: *Comput. Graph. Forum* 29.3 (2010), pp. 1203–1210. DOI: 10.1111/j.1467-8659.2009.01694.x.

[30] Torsten Hoefler and Roberto Belli. „Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. Ed. by Jackie Kern and Jeffrey S. Vetter. ACM, 2015, 73:1–73:12. DOI: 10.1145/2807591.2807644.

[31] Sascha Hunold and Alexandra Carpen-Amarie. *GitHub repository hunsa/reprompi*. 2018. URL: https://github.com/hunsa/reprompi (visited on 11/03/2020).

[32] Thomas Gruber. *LIKWID Wiki / likwid-perfctr / Using the Marker API*. Oct. 30, 2020. URL: https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr#using-the-marker-api (visited on 01/22/2021).

[33] The Open MPI Project. *GitHub repository open-mpi / ompi*. 2020. URL: https://github.com/open-mpi/ompi (visited on 12/01/2020).

[34] Andrei Vagin and Kir Kolyshkin. *How fast is Procfs*. 2016. URL: https://avagin.github.io/how-fast-is-procfs.html (visited on 12/02/2020).

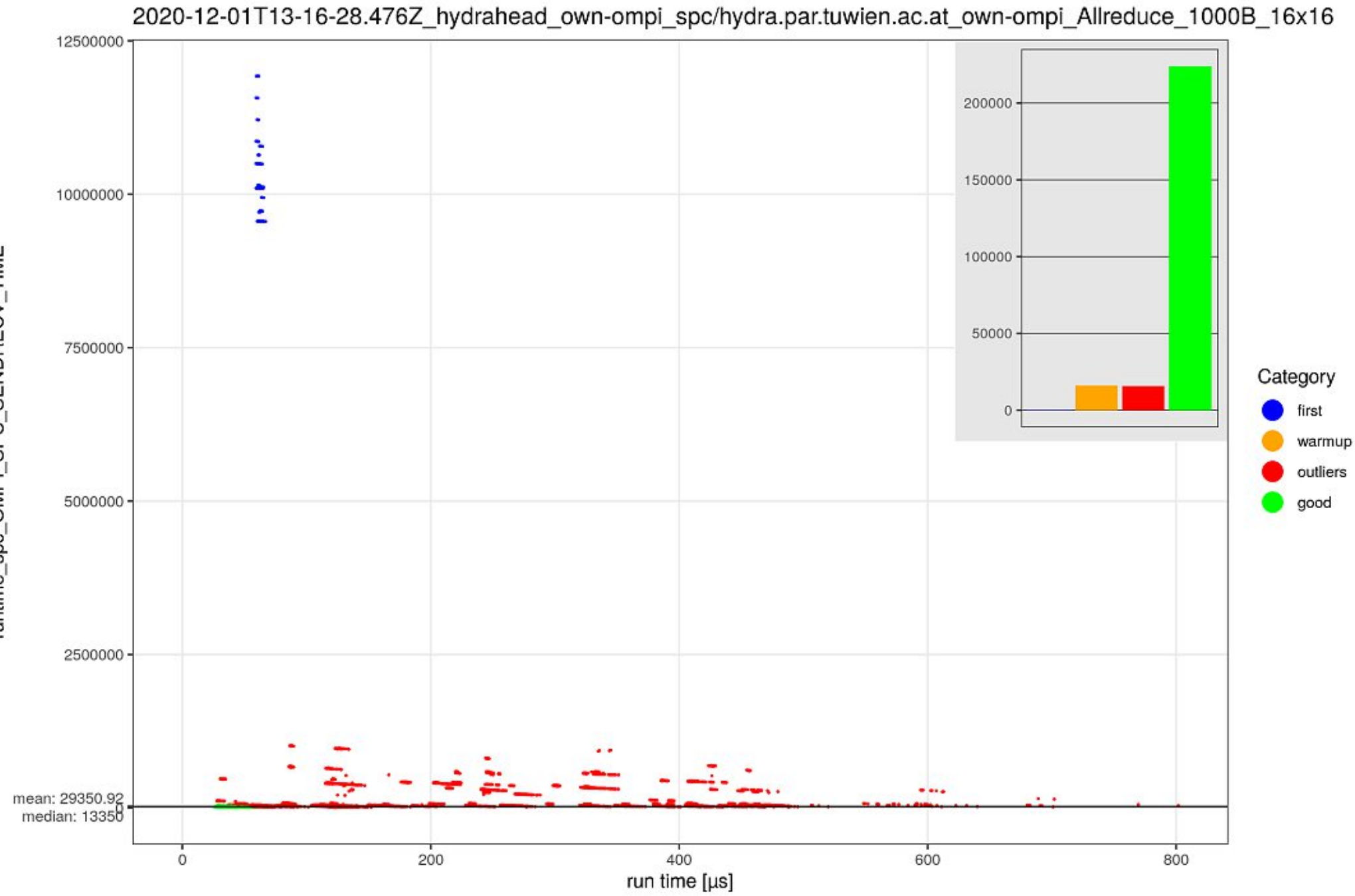
[35] Intel®. *GitHub repository intel / opa-psm2*. 2020. URL: https://github.com/intel/opa-psm2 (visited on 12/04/2020).

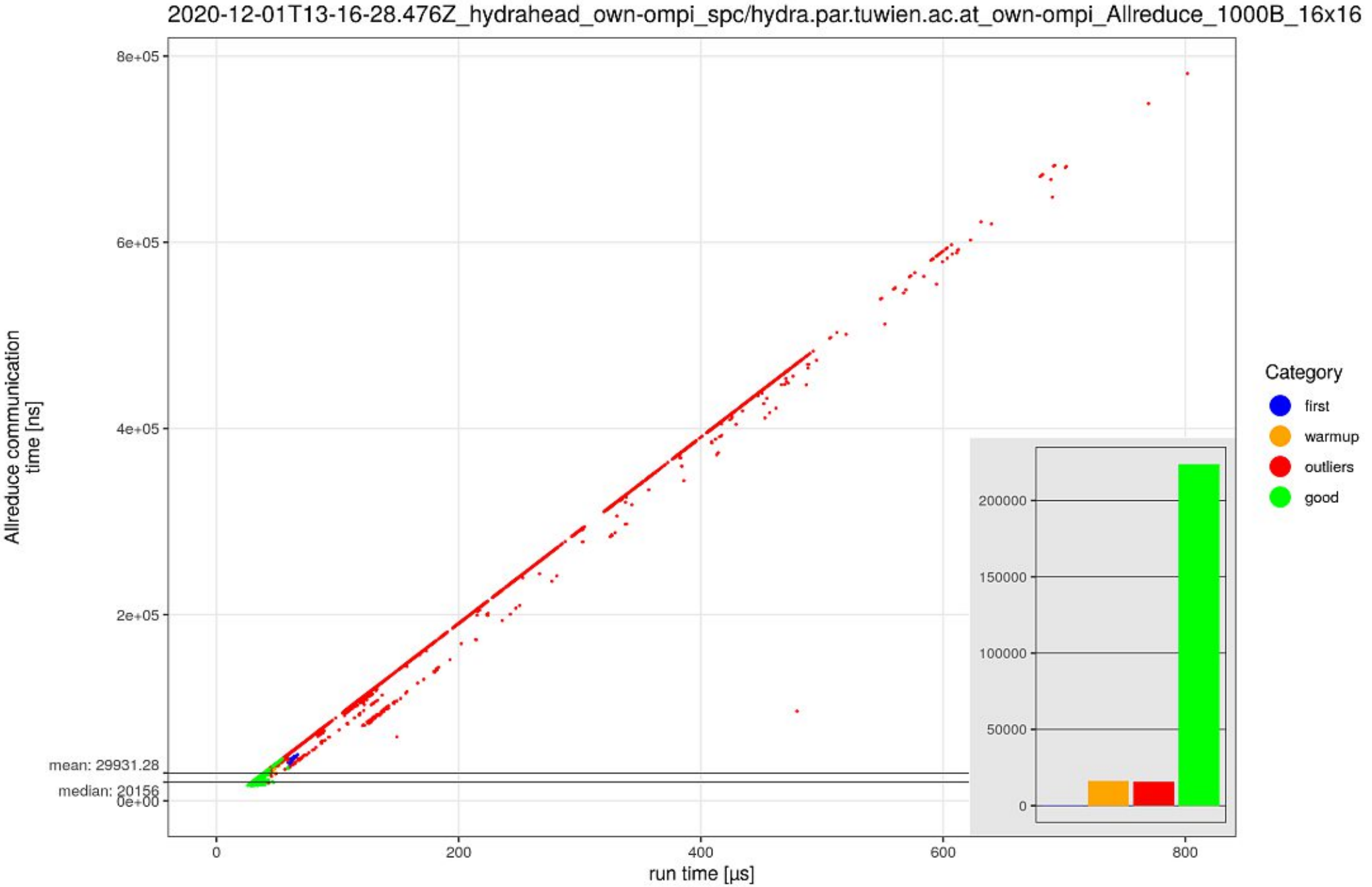
[36] Karl Fürlinger, Nicholas J. Wright, and David Skinner. „Effective Performance Measurement at Petascale Using IPM“. In: *16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010*. ICPADS '10. IEEE Computer Society, 2010, pp. 373–380. ISBN: 9780769543079. DOI: 10.1109/ICPADS.2010.16.

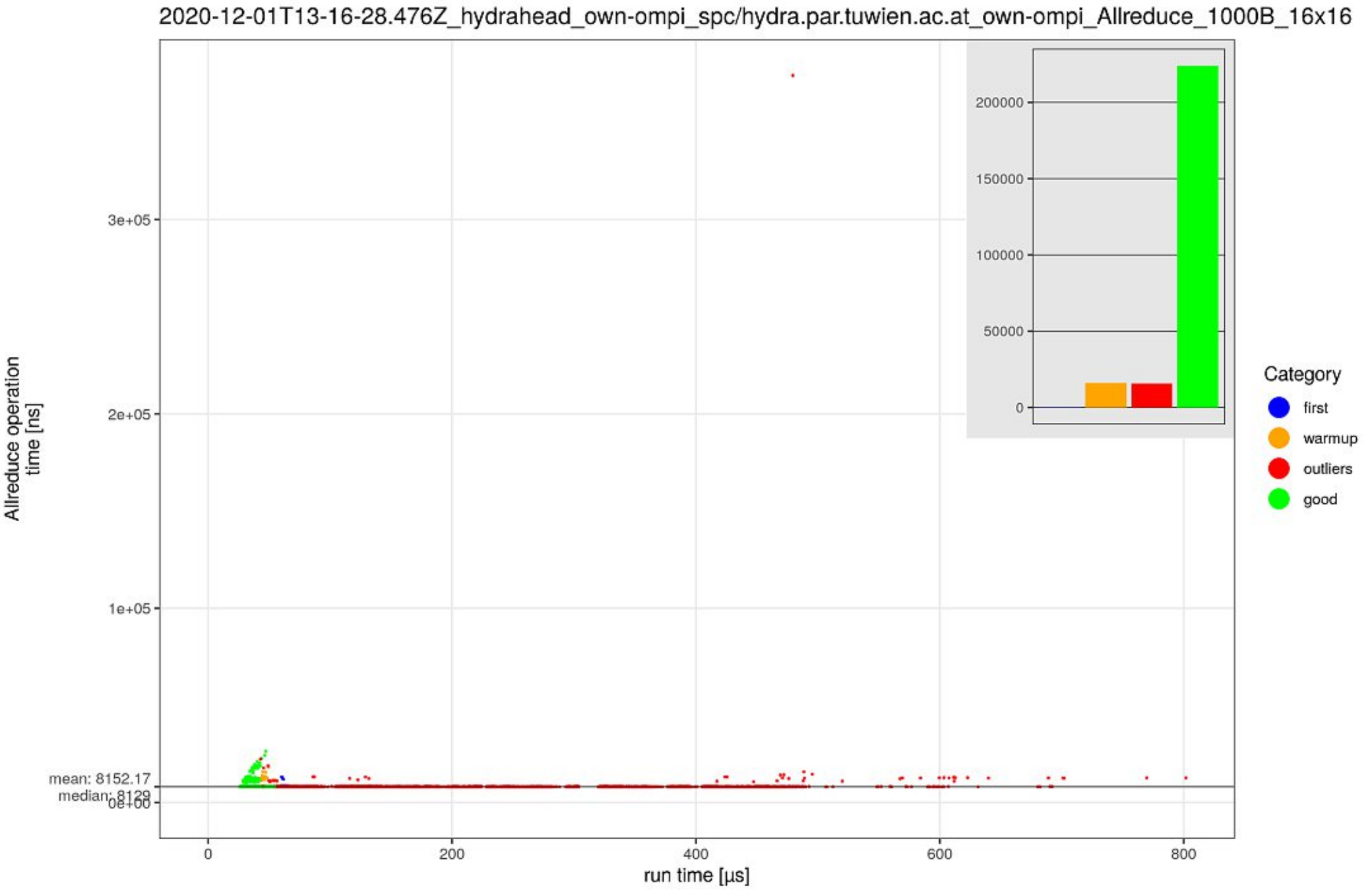
APPENDIX A

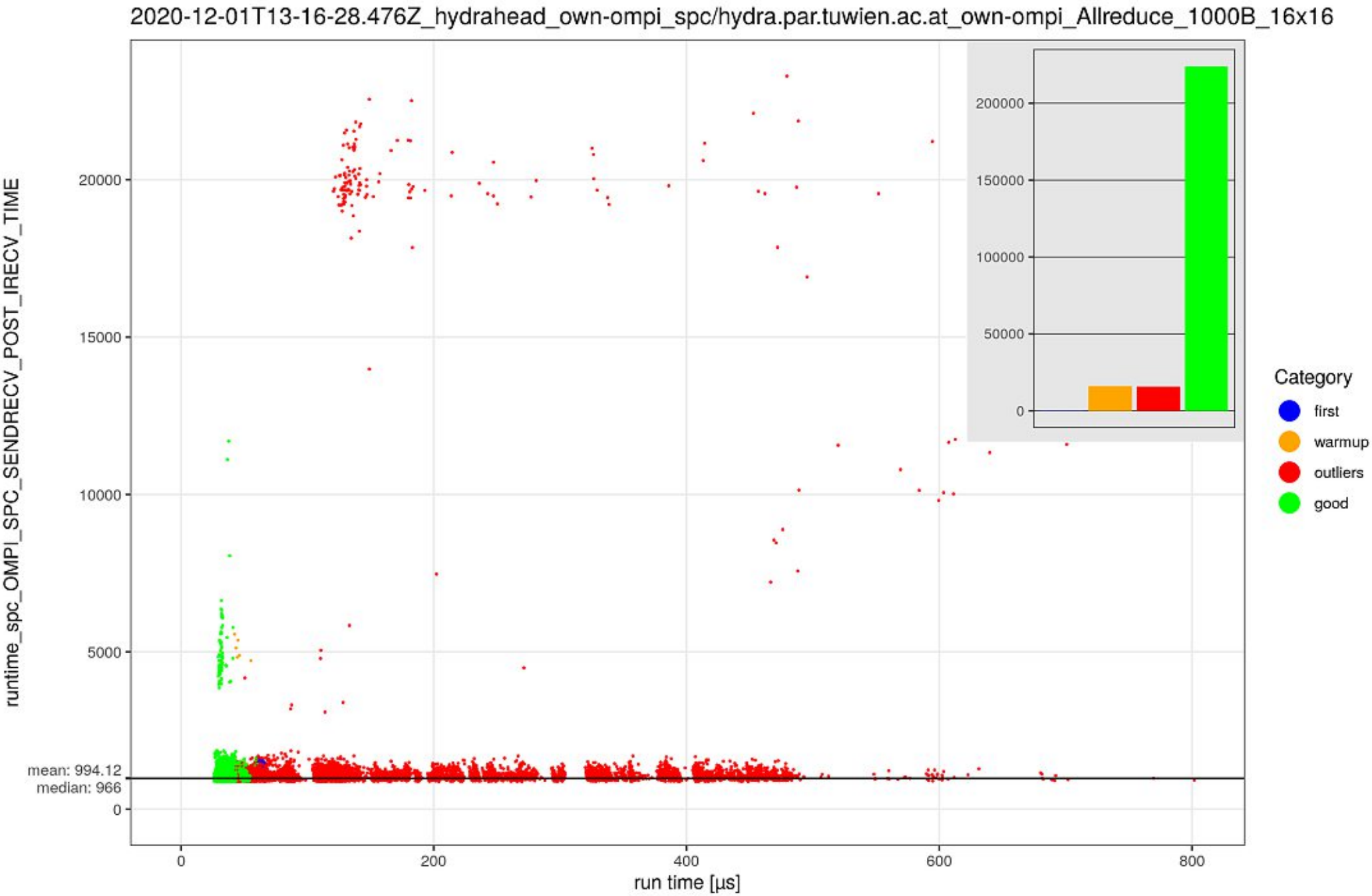
Software Performance Counters vs. Run Time Plots

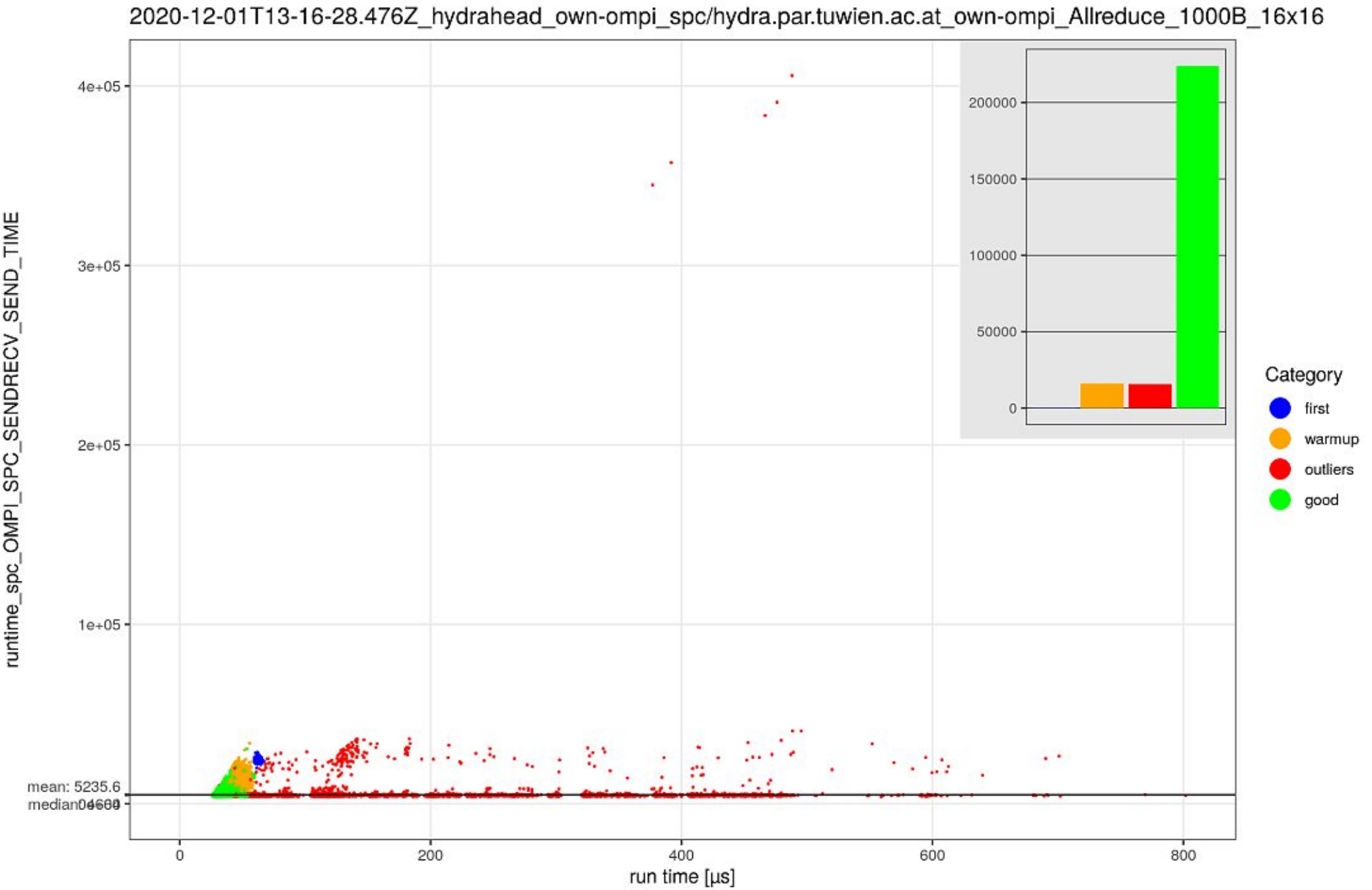
This appendix holds the scatter plots of all SPCs from Table 4.3 which had a value which was not 0 during the run correlated to the run time for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The plots are colored using the categories introduced in Section 4.5.

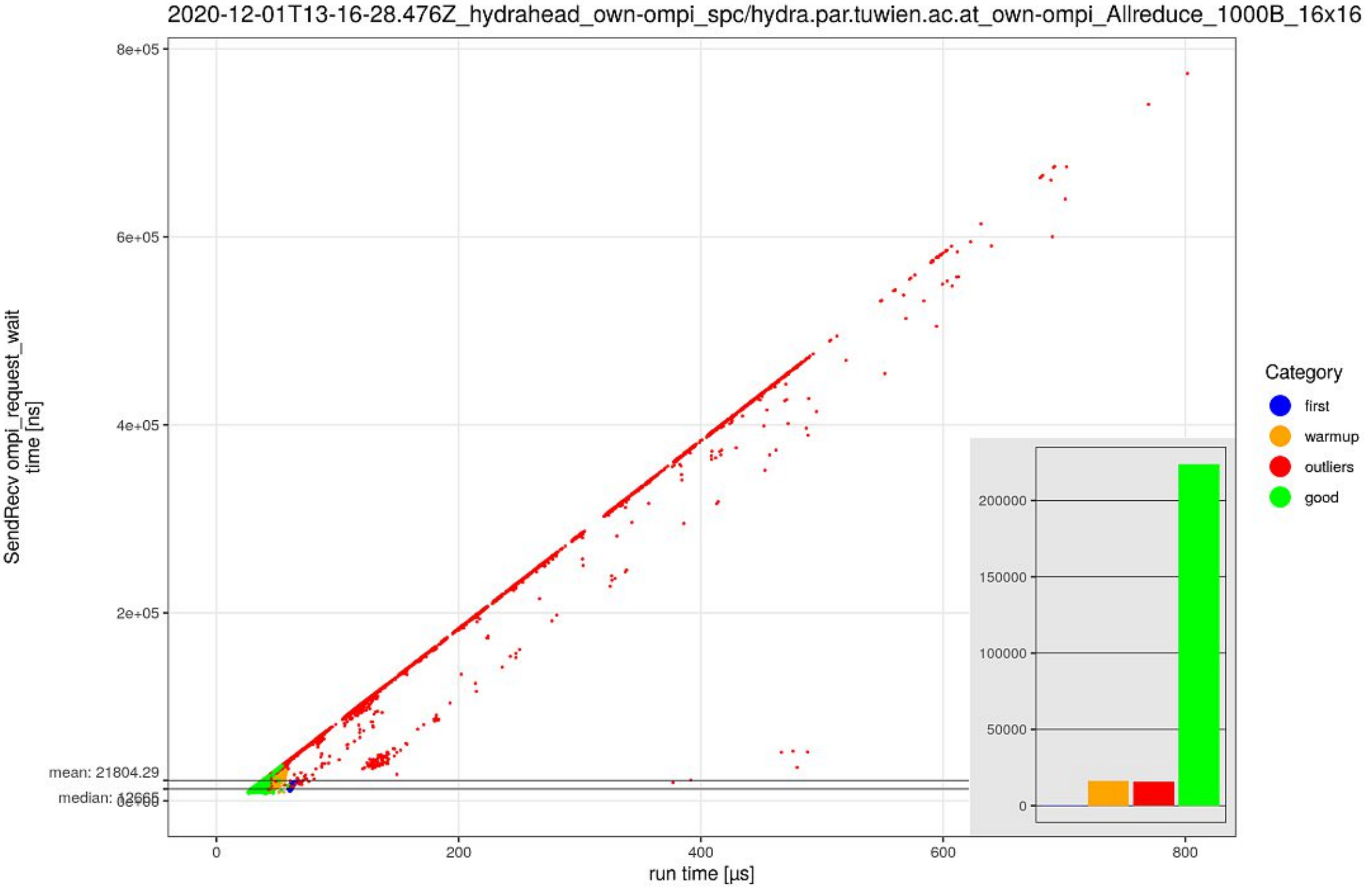


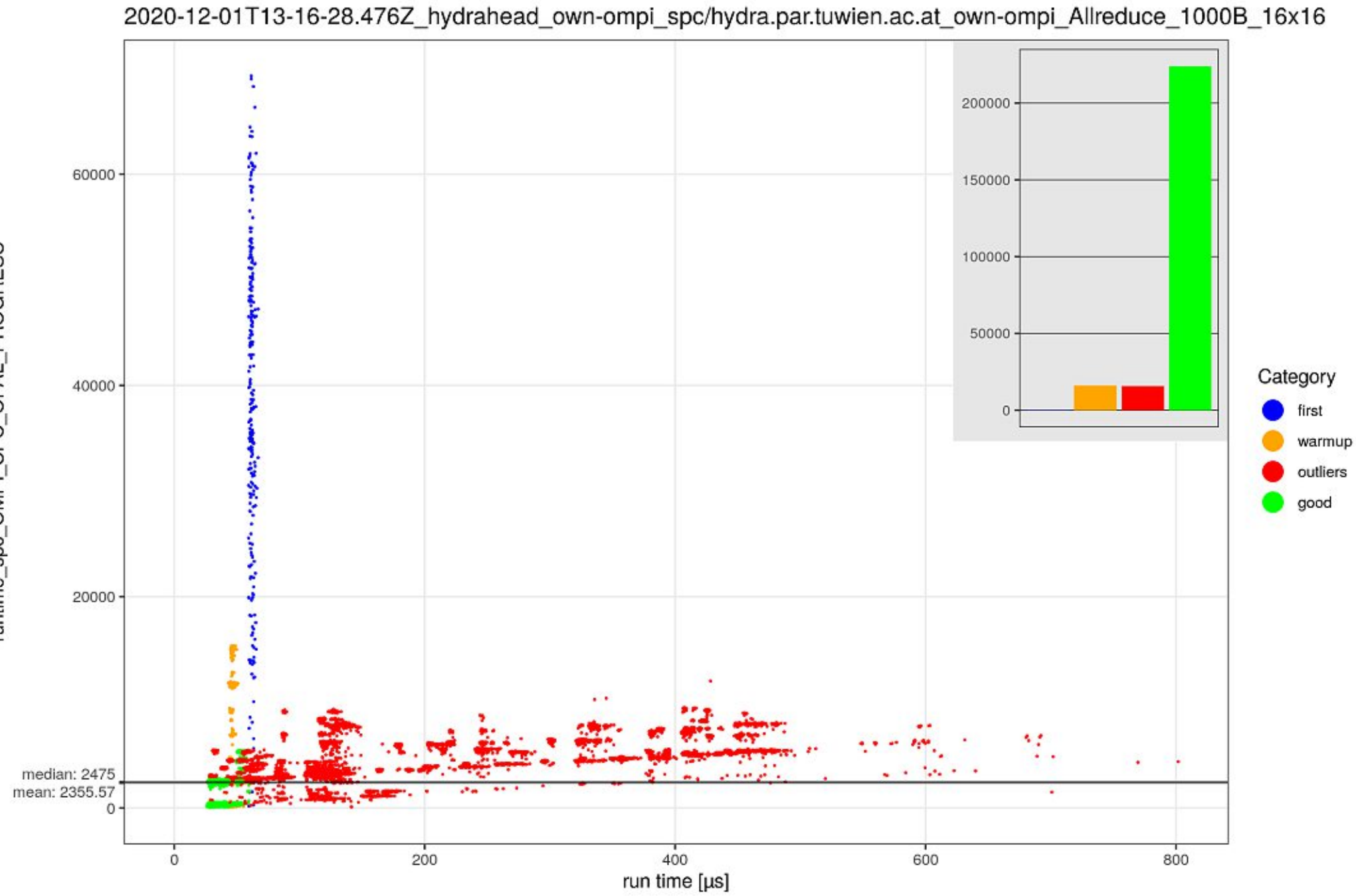


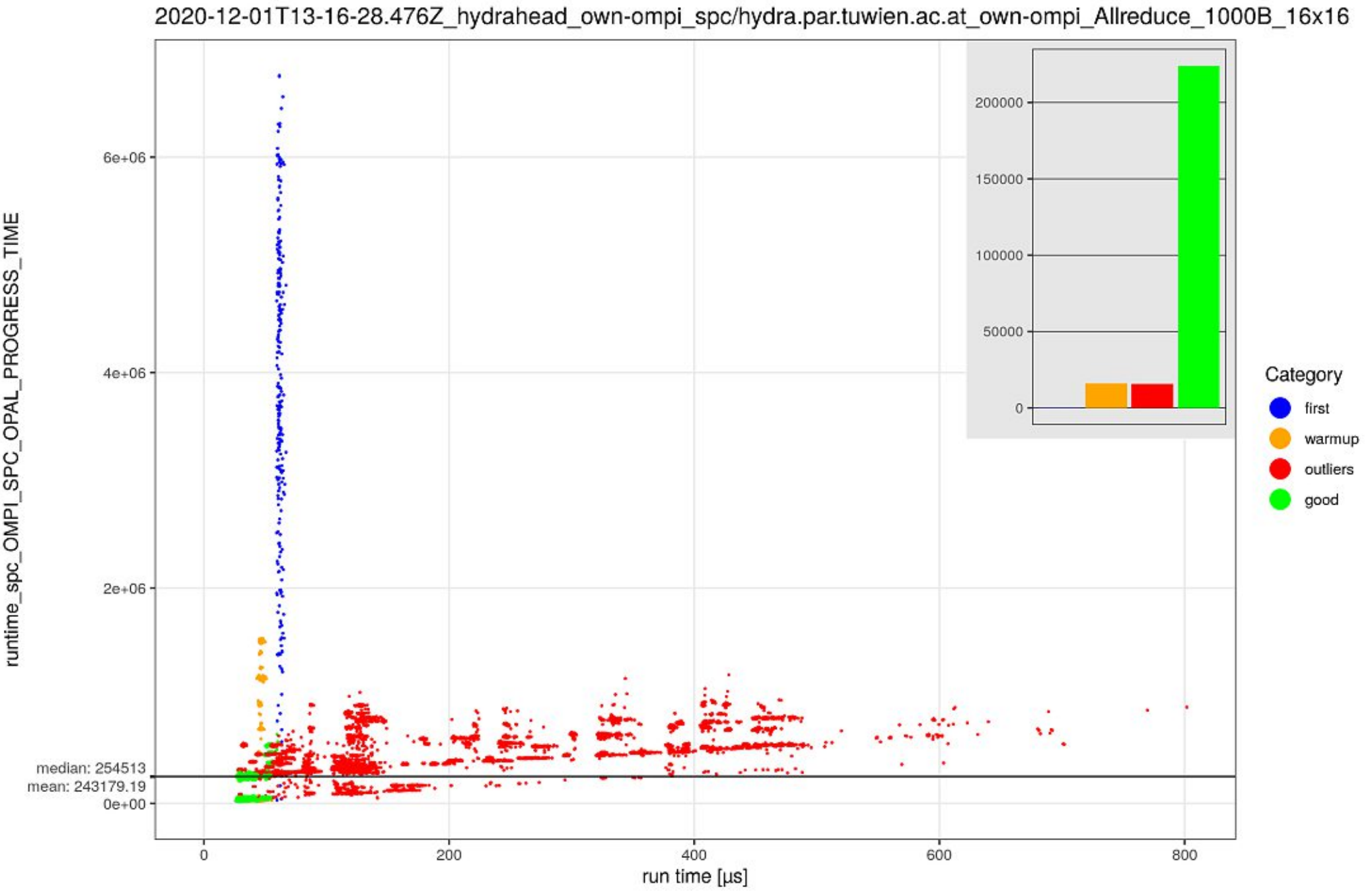


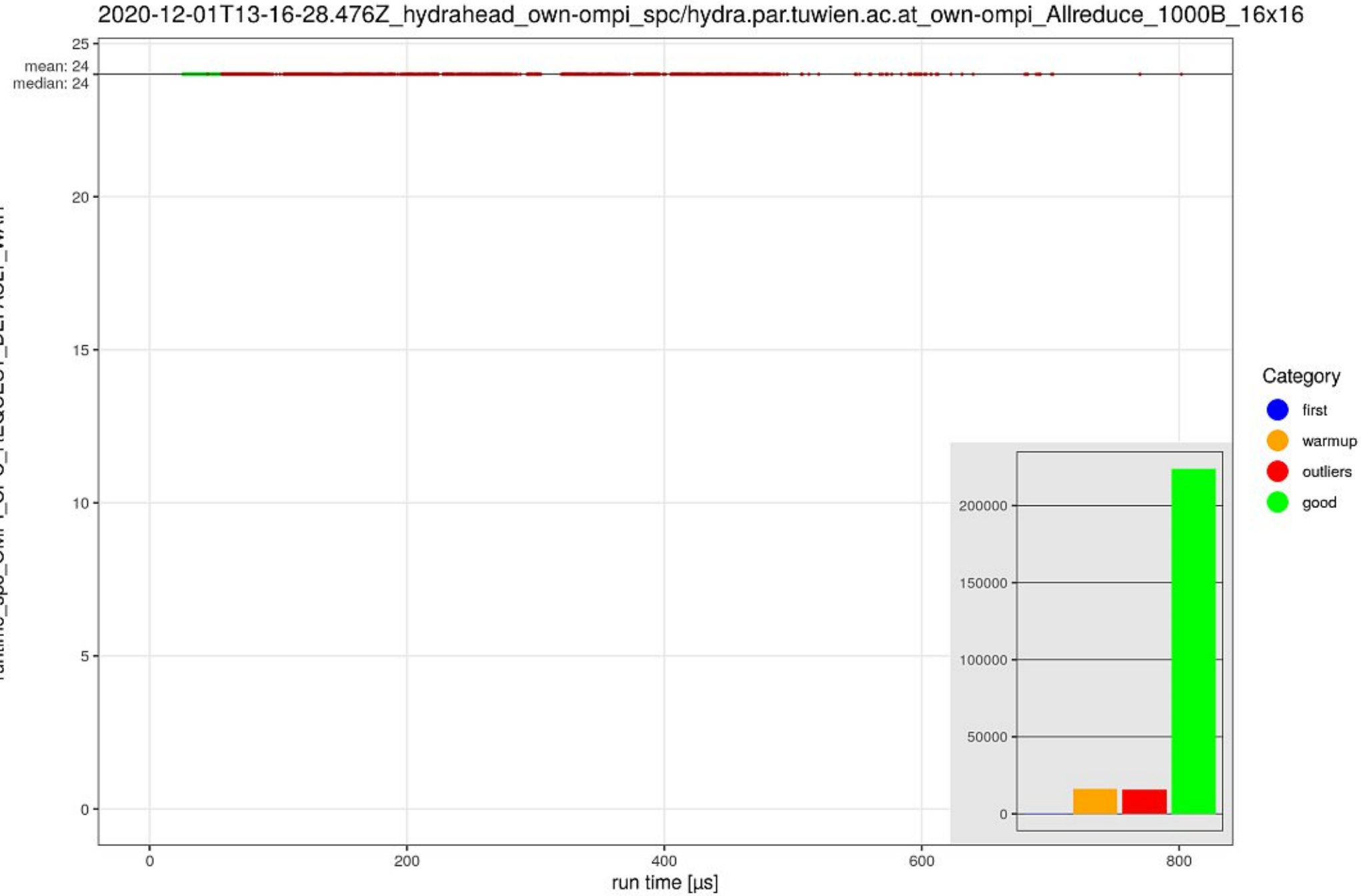


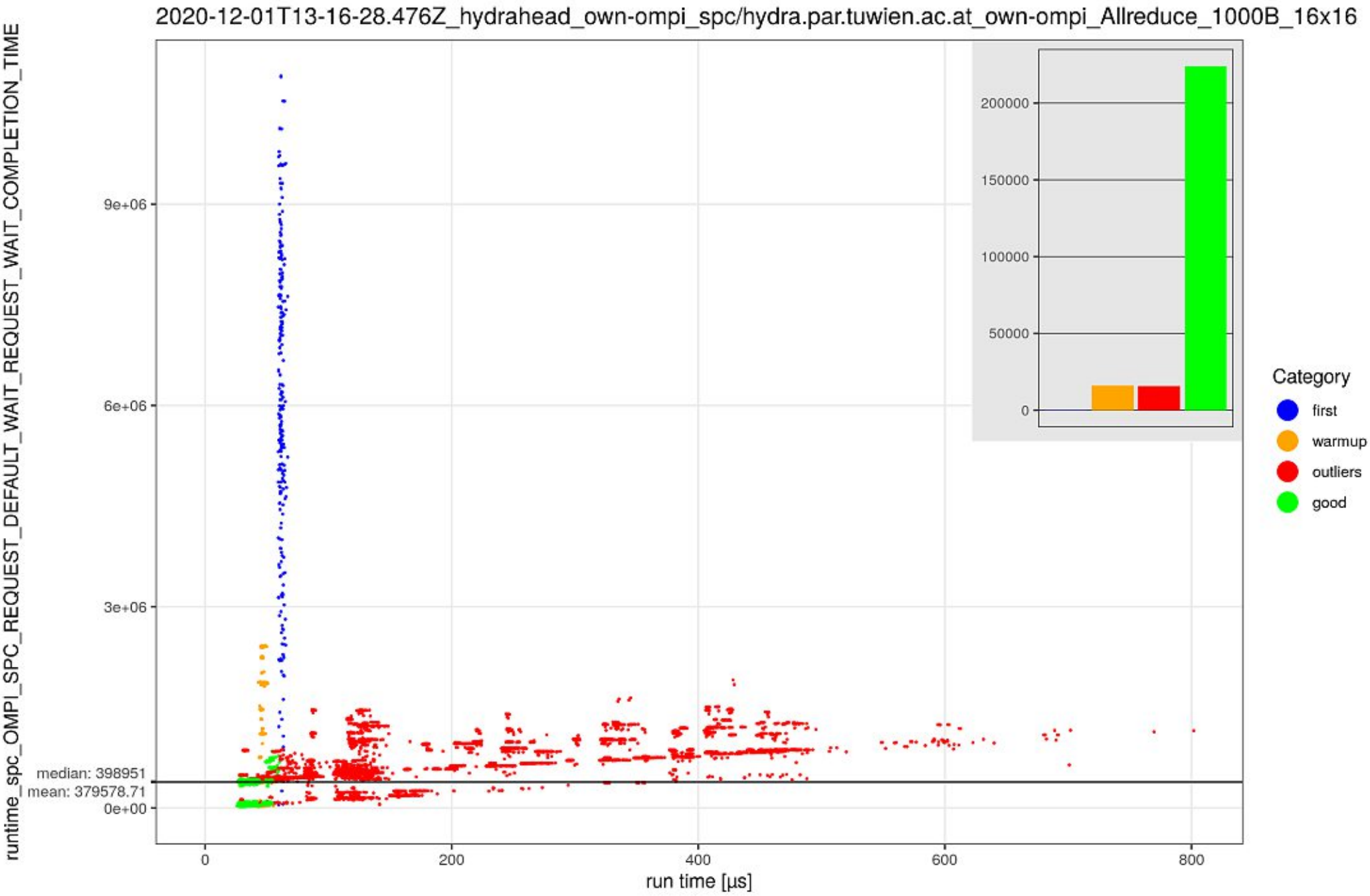


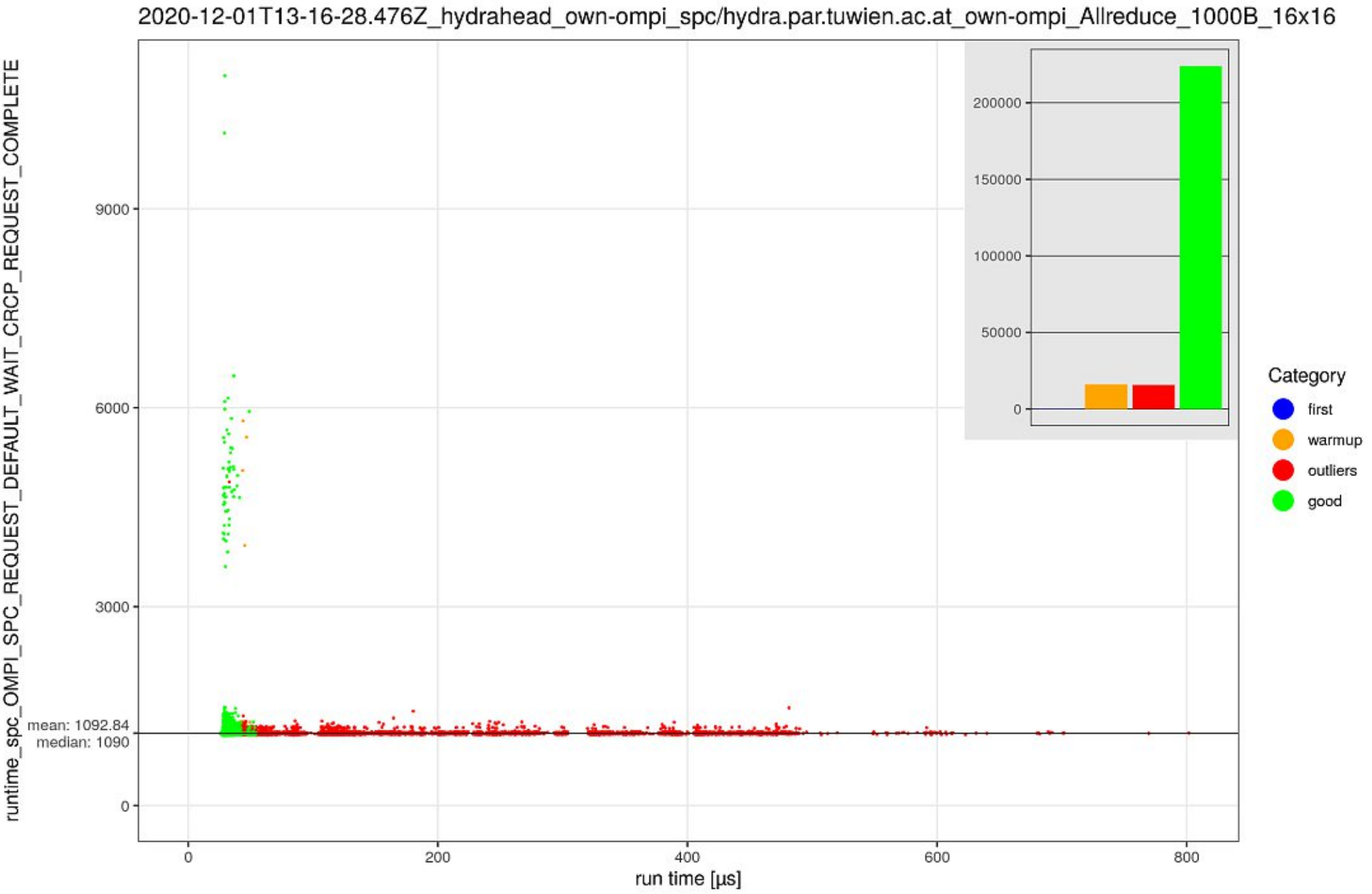


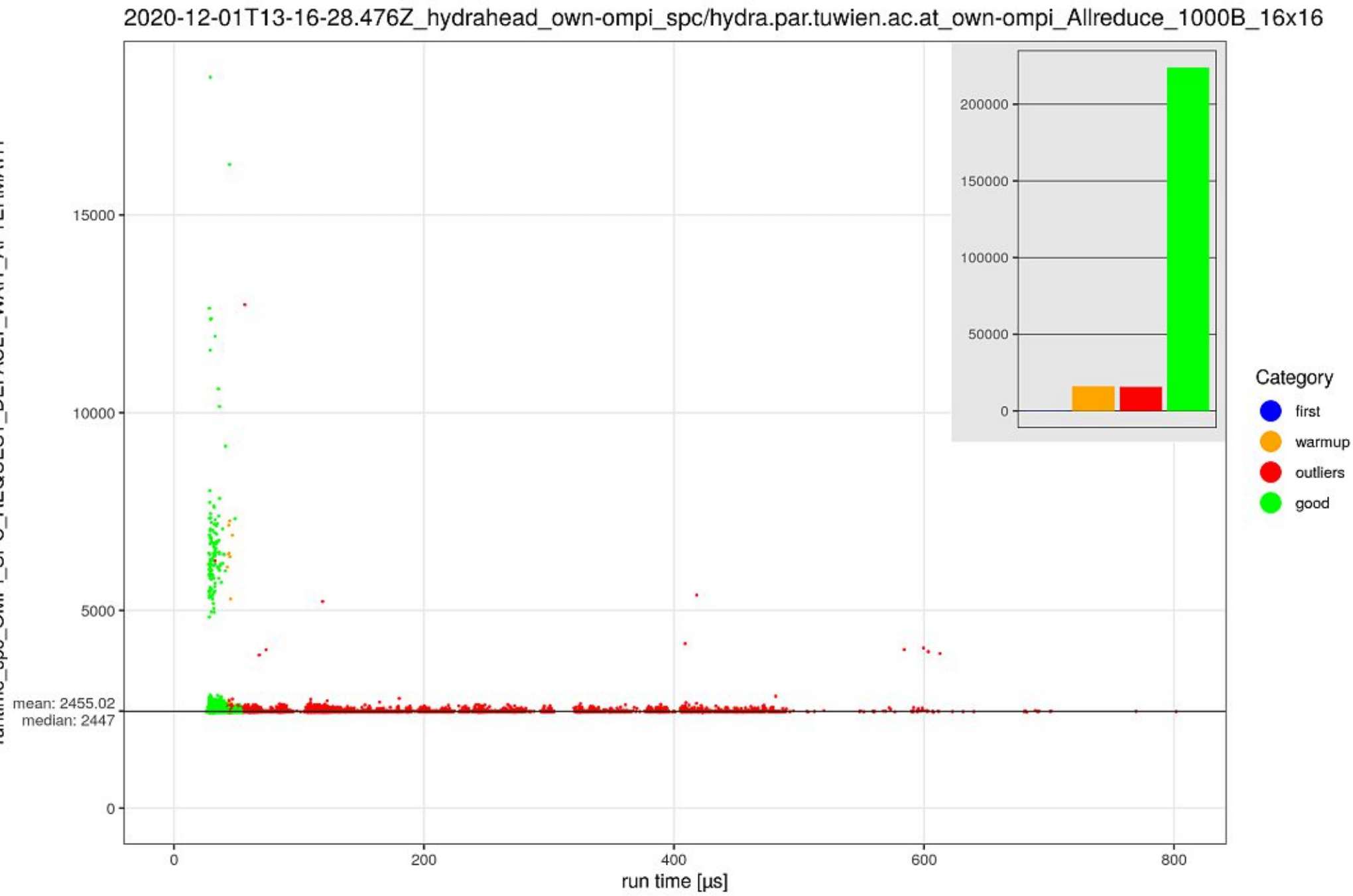












APPENDIX B

Software Performance Counters vs. Repetition ID Plots

This appendix holds the scatter plots of all SPCs from Table 4.3 which had a value which was not 0 during the run for each repetition for 1000 repetitions of 16×16 processes, 1000 B, `MPI_Allreduce` (Recursive Doubling) on *Hydra*. The plots are colored using the categories introduced in Section 4.5.