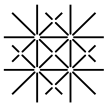

University of Basel

Impact of the Linux Operating System on the Performance of OpenMP Applications

Master Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
HPC Group
https://hpc.dmi.unibas.ch/

Examiner: Prof. Dr. Florina M. Ciorba
Supervisor: Jonas H. Müller Korndörfer

David Kuhn
david.kuhn@stud.unibas.ch
16-057-960

12.02.2021

Table of Contents

1 Introduction

The performance of parallel OpenMP applications is affected by numerous factors for example loss of data locality, improper usage of concurrency control mechanisms and load imbalance. The operating system (OS) scheduling, interrupts and thread migration can affect their performance. The goal of this project is to measure the degree of influence and performance loss caused by the OS.

We use the Linux kernel tool perf to measure switches and their influence on different parallel applications. Our results show that some applications are severely impacted by the scheduler, while other applications have no big overhead. Besides this, we investigate the influence of different OpenMP scheduling techniques and thread configurations.

We start this report with the background in chapter 2 we explain all necessary technologies for this project as well as some related work. In the following chapter 3, we explain how we performed our measurements, which results are presented in chapter 4. We analyse the results in chapter 5 and conclude this report in chapter 6.

2

Background and Related Work

In this chapter, we introduce the basics of the Linux scheduler and OpenMP scheduling techniques. Then we have a short explanation of how perf works. We end this chapter with the related work.

2.1 Linux Operating System Scheduler

The scheduler is an important part of a multitasking operating system [12][16], as Linux. If there was only one process for a system, then this process could execute on a processor until it is finished. This case is rare. Often there are a lot of different processes that compete for resources like execution time, memory, network bandwidth, etc. In today's processor architecture it is not possible to execute more than one process at a time. A multitasking operating system can interleave the execution of multiple processes on one processor to give the impression that several processes are executing simultaneously.

In the Linux operating system, the scheduler decides which process can execute next on the CPU. It also determines the execution time of each process. The goal of the scheduler is to give the user the impression that multiple processes are executing in parallel and that all processes can react to input in a short time. To achieve this, the scheduler can not only start a process when the processor is free, but it can also preempt an executing process. The Linux scheduler should also optimally utilize the processing time. No idle process should occupy the processor if other processes are waiting for execution time. The scheduler should preempt such a process and switch to another waiting process. This is the difference between cooperative multitasking and preemptive multitasking. In cooperative multitasking systems, all processes have to wait until the executing process frees the processor voluntarily. This method is rarely used in practice, by any operating system. The standard is preemptive multitasking where the scheduler can preempt an executing process. This preemption forces the process to wait until the scheduler allows it to execute again.

2.1.1 Scheduling Policy

The scheduling policy in Linux determines the behavior of the whole system. Depending on the use the scheduler can prefer one sort of process over others.

Different processes need different processing times. The design of the scheduler has to balance between latency and throughput. If the execution time for each process is short, then all processes can execute often. This leads to a shorter response time. Before a new process starts executing, it has to load its data to the cache. Therefore each switch results in a delay. If there are too many switches no process could execute because they spent all of their execution time loading data.

Input/ Output- (I/O) bounded processes often do not need a lot of processing time. Normally they wait for an I/O request. After this event, they have to react quickly but execute only for a short time. I/O-bounded processes have to switch often to get a reasonable response time. For example, a text editor that waits until the user presses a button on the keyboard. The editor should react fast and display the new character. Otherwise, the user experience is not great.

On the other hand, processor-bounded processes execute as long as they can. Normally they do not block for I/O events. Therefore the only breaks are preemptions from the scheduler. Because processes have to load the cache after each preemption their total execution times get longer for every switch. In contrast to I/O-bounded processes, processor-bounded processes suffer less if they have to wait for a long time until they can occupy the processor again.

The Linux scheduler has to balance between these two needs. For I/O-bounded processes, the scheduler has to guarantee low latency and for processor-bounded processes a high throughput. In UNIX systems the scheduling policy prefers the I/O-bounded processes for better interaction.

2.1.2 Timeslice

To determine how long a process should execute on a processor, some schedulers calculate a timeslice. This is the allocated time a process can execute. The timeslice has a lower bound because each swap has an overhead. Too many swaps would degenerate the performance of the system. When a process runs its remaining timeslice is reduced. When it is zero the scheduler preempts the process and another process can run. If a process waits for an event, it can also free the processor and allow the next process to run.

2.1.3 Priority

Often, scheduling algorithms use a priority ranking to schedule the next process. A process with higher priority should execute sooner or more often. A process in Linux has two different priority values. A *Nice value* between -20 and +19. A large Nice value corresponds to lower priority. The second priority range is the real-time priority, normally between 0 and 99. A large real-time priority value corresponds to a higher priority.

The Linux scheduling algorithm should prefer tasks with a higher priority but at the same time, low priority tasks should not starve. This means they should also get some

executing time.

2.1.4 Complete Fair Scheduler

Before the Linux kernel 2.6, there were several problems with the scheduler [11]. In the versions 2.4 - 2.6 several different scheduling algorithms were tried but had severe drawbacks. The O(n)-scheduler, the name is derived from its complexity O(n), has an overhead that grows too big, if there are too many tasks. At high load, a significant part of the computation power is spent only to schedule the next process. Also, the scheduling algorithm had a single runqueue for all processors. This allowed for simple load balance on multiprocessing systems, but a process can be scheduled at any processor. This is not optimal because the cache has to be reloaded.

The completely fair scheduler (CFS) was introduced to Linux by Ingo Molnar in 2007. It has a complexity of O(1). CFS simulates a real multitasking processor. The CFS does not allocate a hard timeslice to a process. Instead, each process receives a share of the processor. The scheduler allocates 1/n of the total processor time to a process. Where n the number of processes is, which need processing time. The processing time depends on the load of the system. CFS uses the Nice value as weight. The share of a low priority process is smaller than that of a higher priority process. To simulate a real multi-process system, this timeslice should be infinitely small. But to prevent too many switches this share of computing time has a lower bound.

To guarantee that in a given time interval each process can execute, CFS has a *target latency*. A smaller target latency results in better interactivity for I/O-bounded processes. If there are too many tasks, the allocated timeslice would become too small and the processes would switch too often. To prevent this, CFS has a *minimal granularity*. This is the minimal time that a process should execute to prevent that the switching costs affect the performance of the whole system. The default minimal granularity is 1 millisecond.

Each process has a *virtual runtime*. This is the execution time normalized by the number of runnable processes. To determine the next process, the CFS has a red-black-tree (rbtree) ordered to the runtime of each process. With this, the next process is found in the left-most leave of the rbtree. This is the process that had the least time on the processor.

CFS has a dynamic load balancing. In regular time intervals, CFS moves processes from CPU's with high loads to CPU's with less load.

The Linux scheduler is modular. This allows different scheduling algorithms to run in parallel. Different scheduler classes can schedule different processes. Each scheduling class has a priority. The one with the highest priority and a runnable process can schedule the next process. The complete fair scheduler is the standard scheduler for normal processes.

2.1.5 Context Switch

In a context switch, an executing process is interrupted to free the processor for another process [17] [12]. This is necessary that several processes can share one processor in a multitasking operating system. For this, the state of the executing process is stored. This process can be restored later. It can then continue the work where it was stopped.

A context switch can be triggered, when the scheduler preempts a process to let another process execute. It is also possible that a process voluntarily frees the processor. For example as a result of an interrupt for disc storage access or to synchronise with other processes.

Context switches are expensive. To switch from a process to another requires saving the register and memory map. In Linux, the switch between threads from the same process is not as expensive as switching between different processes. This is because threads from the same process share the same virtual memory map.

In our experiments, we measure the impact of context switches. We see the impact that switching has on the performance of different applications.

2.2 OpenMP

Open Multi-Processing (OpenMP) is an application programming interface (API) that supports the programming languages C, C++ and Fortran [6]. It consists of compiler directives and library routines to express shared-memory parallelism. OpenMP is supported on most operating systems, including Linux, Windows, macOS, Solaris, AIX and HP-UX. OpenMP is a simple interface to develop multi-threaded applications. In OpenMP a primary thread forks sub-threads. The system shares the work among these threads. OpenMP supports task parallelism and data parallelism. The first version of OpenMP was introduced by the OpenMP Architecture Review Board (ARB) in 1997. In November 2020 the current version 5.1 was released.

2.2.1 Scheduling Techniques

The OpenMP standard specifies three loop scheduling techniques: static, dynamic, and guided [13][14]. The static scheduling technique divides the work into equal parts before the execution. Each thread receives one part. So if there are t threads and n loop iterations, each thread receives about n/t of the work.

The dynamic scheduling technique divides the work into chunks. Each tread requests a chunk of work until there is no more work left to do. It is not deterministic which thread executes which chunk.

Guided scheduling is a self-scheduling method similar to dynamic. The difference is the size of the chunks. In the beginning, the chunk size is large. The chunk size is proportional to the numbers of unassigned chunks and the numbers of threads. The chunk size decreases exponentially during the execution.

2.3 Perf

Perf is part of the Linux kernel tools [7][15]. Like the Linux kernel, perf is open source. Perf can be used for monitoring the performance of applications, CPU performance counters, dynamic tracing, lightweight profiling and many more. Perf was introduced as a tool to use the performance counters subsystem in Linux. It had various enhancements to add tracing capabilities. Performance counters are CPU hardware registers that count hardware events.

For example, instructions executed, cache-misses suffered, or branches mispredicted.

On the command line, the usage of perf is similar to git. There exist the generic tool perf and a set of commands for different tasks, eg stat, record, report and many more. You can use `perf --help` to see a list of the most commonly used perf commands.

Good guides to start with perf are [9] and [1]. Different perf commands are documented and they provide good examples for many use cases. There are many tools that can analyse the output file of perf to analyse the performance of applications.

In this project, we use pref to analyse the influence scheduling has on parallel applications. We explain in section 3.2 how perf can be used to investigate the Linux scheduler behavior.

2.3.1 Events

Perf can measure events from different sources. You can see a list of supported measurable events with this command: perf list Events from the kernel and other parts of the OS are called *software events*. Examples of this are context-switches, CPU-migration or minor-faults.

The processor and its Performance Monitoring Unit (PMU) are another source for events called *PMU hardware events*. Examples of this are CPU-cycles, cache-misses or instructions. Hardware events vary, depending on the type of processor. There are also *tracepoint events*. They are implemented in the ftrace infrastructure.

Perf can be used in three ways. The simplest way is counting events. Perf creates just a summary of what it counted during its observation. No binary perf.data file is generated. An example of this is the perf stat command. The overhead of this measurement is comparatively small.

Perf can also sample events. This writes all event data to a buffer. Perf writes this buffer asynchronously to a binary file, by default called perf.data. With the perf report or perf shript commands, we can analyse this file after the measurements are finished. This reduces the overhead of measurement, which is significant for this type of recording. The perf.data file can become very large for applications with a longer execution time. It is also possible to use user defined programs. To filter and summarize the events.

2.4 Related Work

Betti et al. [3] designed an operating system for HPC clusters, that addresses the issue of temporal synchronization. To study their kernel improvements, they performed tests to analyse the system noise. They use Fixed Time Quanta (FTQ) to measure the amount of work done in a fixed time quantum. With the measured basic operations and the maximum number of basic operation, they calculate the influence of the OS.

Gioiosa et al. [8] present modification to Linux to optimize the performance of HPC applications on compute clusters. They use perf to count CPU migrations and context switches during parallel application executions. They focus on MPI applications and measure significant variation in execution time. They also report a correlation between the

number of CPU migration and process preemption and the variation of the execution time. According to Gioiosa et al. the scheduler is one of the main components that introduces overhead and performance variability.

Akkan et al. [2] evaluate the OS noise in the Linux kernel. They describe OS noise as everything that is not directly related to the application. For example scheduler load balancing, timekeeping and accounting. They use procfs file to get information about system interrupts and ftrace to identify events during ticks.

Differently from the work mentioned above, the goal of this project is to investigate the relationship between OS scheduling and thread level scheduling in OpenMP.

3

Evaluating Overhead of Linux Operating System

To record the Linux scheduling overhead of different OpenMP applications we use perf. The resulting output files are analysed with python scripts, which also generate the plots. We explain how we conducted our experiments and what the goals of them are, in this chapter.

3.1 Hypothesis

The goal of this project is to find out what influence OS scheduling has on parallel applications. We want to find out what the influence of the OS scheduler is on the performance of the application. The scheduler can switch or interrupt processes and thereby influence their performance. With perf, we can measure the execution time, the numbers of switches and how long a switch takes on average. With this method, we use different numbers of threads to see if the operating system can use idle CPUs. Furthermore, we compare different scheduling techniques introduced in Section 2.2.1.

We analyse the relationship between OS scheduling and thread level scheduling in OpenMP. Questions we want to answer with our experiments are, will fewer threads result in fewer interruptions, because there are free cores that can be used for the operating system? If this holds the scheduler can make use of the idle CPUs if the application does not use them all.

We also want to investigate the influence of the different OpenMP scheduling techniques static, guided and dynamic, which we explained in section 2.2.1, on the performance and numbers of interrupts. A scheduling technique that distributes the work at execution time has a bigger overhead because some work is deferred. But the work distribution at execution time addresses the problem of load imbalance. The threads that finish their work faster are less idle when waiting for the thread with the highest workload. So we investigate if there is a difference in the number of switches between those scheduling techniques. We want to find out if there are more switches if we use a scheduling technique with a bigger overhead.

3.2 Scheduling Overhead Analysis with Perf

In this work, we used perf to analyse the performance impact of OS scheduling on OpenMP applications. In this section we describe the tools perf provide to analyse the scheduler behavior.

The `perf sched` command provides several tools to analyse the Linux scheduler. It stores the recorded events in the file perf.data. Perf provides several tools to analyse this file.

`perf sched record` records the scheduling events of arbitrary workloads and saves it in the file perf.data.

`perf sched latency` Analyses the perf.data file. The output is a list of scheduling properties for each process. This includes the runtime of each process, the number of switches the average delay as well as the maximum delay for switching and when this was recorded.

`perf sched map` shows a summary of context-switch events from the file perf.data. This is printed so that each CPU has a column. It shows for every switching event, what each CPU was doing.

`perf sched replay` simulates the captured workload in perf.data by spawning processes with similar execution times. This was introduced to see how changes in the scheduler algorithm behave for similar workloads.

`perf sched script` dumps all scheduler events. This allows us to see a detailed trace of the workload that was recorded.

`perf sched timehist` provides an analysis of the latency for all scheduling events.

3.2.1 Recording Scheduling Overhead with Perf

To measure the overhead introduced by the scheduler, we record the workload of benchmarks with the following command:

```
sudo perf sched record -a -R ./application
```

We use perf sched record to save all events in a binary file. It has a noticeable overhead because it stores all recorded events. Because perf reads kernel counters we need to execute this command with sudo. There is also the option to give perf permanent access to the needed counters by setting the /proc/sys/kernel/perf_event_paranoid value to 1. The argument -a is that perf records events from all CPU's and -R collect raw sample records from all opened counters.

An example for the application Mandelbrot, which we will introduce in chapter 4. The output of perf sched record is:

```
perf record: Woken up 266 times to write data
perf record: Captured and wrote 1338.504 MB [...] (12518133 samples)
```

Perf recorded 12'518'133 samples. The file perf.data has a size of 1.25 GB. This depends on the workload and the system. Perf has only written 266 times to the file. This is to reduce the overhead of perf. Perf should use as little execution time as possible.

We analyse the binary file that was generated by the previous command with:

```
sudo /usr/bin/perf sched latency
```

We save this output in a text file to generate plots later. The output of this command is a list of all processes that were recorded at the time the application was executing. For each process, there is the runtime in milliseconds, the number of switches and the average duration of a switch. Additionally, there is the maximum duration of a switch, but we did not consider this for our analyse because this is often an extreme outlier.

In figure 3.1 we see an example of this output. This is a recording of the application Mandelbrot. The recorded execution time is 12214945.771 ms. This is the cost of the parallel execution on all used processors. We also see that perf recorded 46360 switches for Mandelbrot with an average delay of 0.058 ms. So, 2688.88 ms was spent on switching only the application itself. Besides the measurements of Mandelbrot, we see different other programs that executed on the node alongside our application.

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
date:(2)	2.010 ms	4	avg: 4.464 ms	max: 8.341 ms	max at: 2334836.177709 s
perf:39741	1149.086 ms	242	avg: 1.648 ms	max: 12.999 ms	max at: 2334870.438190 s
sadc:39783	9.453 ms	8	avg: 1.439 ms	max: 8.755 ms	max at: 2334836.162709 s
khugepaged:233	0.373 ms	31	avg: 0.999 ms	max: 1.000 ms	max at: 2334994.285534 s
systemd-cgroups:39786	0.994 ms	2	avg: 0.745 ms	max: 0.985 ms	max at: 2334836.198711 s
crond:(2)	6.561 ms	19	avg: 0.739 ms	max: 12.327 ms	max at: 2334836.152713 s
kthreadd:(8)	2.137 ms	58	avg: 0.151 ms	max: 0.983 ms	max at: 2335117.086833 s
mandel.o:(40)	12214945.771 ms	46360	avg: 0.058 ms	max: 19.138 ms	max at: 2334916.499317 s
dbus-daemon:1804	4.037 ms	44	avg: 0.048 ms	max: 0.218 ms	max at: 2334836.140213 s
slurmd:(2)	0.383 ms	8	avg: 0.025 ms	max: 0.066 ms	max at: 2334935.842305 s

Figure 3.1: This is an example output of the command perf sched latency. The input is the file perf.data. The recorded application is Mandelbrot.

3.3 How the Experiments Where Performed

All experiments were conducted on miniHPC. This is a small HPC cluster at the University of Basel. It has 22 Intel Xeon E5-2640 computing nodes on which the experiments were conducted. The operating system of minHPC is CentOS Linux 7 (Core).

To do the experiments efficiently, we have several python scripts for the single steps of the experiment. With

```
create\_test\_nodes\_scripts.py
```

we generate job scripts for each node and scheduling technique with the desired configurations. These job scripts can be submitted on miniHPC. The job script includes the performance measurements of a benchmark and the analysis of the resulting binary file. The output of perf sched latency is stored for later use. Parameters for this file are the application we want to analyse and its arguments, The compiler, the numbers of threads and to which CPUs they are pinned.

We used matplotlib version 3.3.3 to generate the plots in chapter 4. We generate them with the script:

```
generate\_latency\_plots.py
```

This script reads the output of perf sched latency and generates plots. If there are different executions, it calculates fitting y-axis scales for the plots. So that they are easier to compare.

4

Experiments

In the first part of this chapter, we explain our experiments. Then we show the results of these experiments in many graphs.

4.1 Design of Factorial Experiments

For our experiments, we used the applications LavaMD and Hotspot3D from the rodinia benchmark [4][5], the benchmark SPH_EXA [10] and Mandelbrot. All benchmarks use OpenMP to parallelise the main loop(s).

LavaMD calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into large boxes, that are allocated to individual cluster nodes. LavaMD is a non time-stepping algorithm. There is one loop that is iterated 64'000 times. We compiled LavaMD with `gcc -lpthread -lstdc++` and executed it with `./lavaMD -cores 16 -boxes1d 40`.

Hotspot3D is a widely used tool to estimate processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations for block. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. Hotspot3D is a time-stepping algorithm, we are executing eight time steps. It has one loop that is iterated 100 times. We compiled Hotspot3D with `gcc -lpthread -lstdc++` and executed it with this command `./3D 512 8 100 ../../data/hotspot3D/power_512x8 ../../data/hotspot3D/temp_512x8 output.out`. Hotspot3D is a special application. It has only eight loop iterations. Therefore only eight threads can execute in parallel. Every additional thread can not be used and is idle. We expect that the application has fewer switches if more than eight threads are used.

The smooth particle hydrodynamics (SPH) technique is a purely Lagrangian method. SPH discretizes a fluid in a series of interpolation points, whose distribution follows the mass density of the fluid and their evolution relies on a weighted interpolation over close neighboring particles. SPH_EXA is a time-stepping algorithm. We execute two time steps. It has two loops with 1'000'000 iterations. We executed SPH_EXA with this command `./bin/omp.app -s 2 --cinput ../checkpoints_600_to_2500/checkpoint`

`700.bin`.

The Mandelbrot algorithm generates a Mandelbrot set. We execute Mandelbrot with 200 time-steps. The algorithm has three loops with 262'144 iterations. We compiled the application with `gcc -O3 mandel.c -lstdc++ -fOpenMP -o mandel.o .` and executed it with the command `./mandel.o 10000 512 0 0 0.5`.

We chose these applications because they have different kinds of load imbalances. LavaMD has a relatively short execution time and has little load imbalance. The load imbalance is moderate for SPH_EXA and high for Mandelbrot. Hotspot3D is special because it has only eight iterations. Also SPH_EXA, Mandelbrot and Hotspot3D are time-stepping and algorithms. LavaMD is a non time-stepping algorithm.

We use three different scheduling techniques static, guided and dynamic,24. As described in 2.2.1 static distributes the work before the execution in equal chunks among the working threads. Guided divides the work in chunks and distributes them to the threads at runtime. The chunk size is big at the beginning and shrinks during the execution. The dynamic scheduler divides the work into chunks. Each tread requests a chunk of work until there is no more work left to do. The chunk size stays the same during the execution.

We also considered different thread configurations. The nodes we used for the experiments have two sockets with ten CPU's each. The first configuration is 20 threads that can execute on any CPU. They are not pinned, therefore the scheduler can migrate them to any CPU during the execution. The second configuration is again with 20 threads, but they are pinned to a CPU. The scheduler can not migrate a thread to another CPU during the execution. Then we used only 16 Threads, eight on each socket. This leaves two idle CPU's on each socket. We hope that the scheduler can use these free CPUs for everything that is not directly related to the application. In the last setup, we used only then threads on one socket, to leave a socket idle. In the third and fourth configuration, the threads are pinned to a CPU so that the scheduler can not migrate the threads.

Design of Factorial Experiments, total 59'400 experiments

Factors	Values	Properties
Applications	rodinia lavaMD	N = 64'000 T = 1 L0 = kernel_cpu;112;197;
	rodinia hotot3D	N = 100 T = 8 L0 = main;161;179;
	SPH-EXA Evard Collapse	N = 1,000,000 T = 2 L0 = (gravityTreeWalk, [194..194] L1 = (findNeighbors, [37..37])
	Mandelbrot	N = 262,144 T = 200 L0: main;178;178 L1: main;208;208 L2: main;240;240
Scheduling techniques OpenMP standard	static (STATIC)	Straightforward parallelization
	guided (GSS), dynamic,24	Dynamic and non-adaptive self-scheduling techniques
Computing nodes	miniHPC{cl-node001..cl-node022}	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each) P = 20 without hyperthreading
Pinning	Free	no pinning
	20 Threads	OMP_PROC_BIND=close", OMP_PLACES="cores"
	16 Threads	OMP_PLACES= "{0,1,2,3,4,5,6,7,10,11,12,13,14,15,16,17}"
	10 Threads	OMP_PLACES="{0,1,2,3,4,5,6,7,8,9}"

Table 4.1: The Variable N is the number of loop iterations. The variable T is the number of time steps. L0 to L2 is the location of the loop.

4.2 Results of the Experiments

In this section, we present the results of our experiments. The different graphs, that resulted from our experiments, are integrated into one figure. The graphs represent the same for every experiment.

On the X-axis there is always the node. This is the minHPC node on which the experiment was executed. In the first graph, from the top, we see the parallel execution cost for the application on the Y-axis. The values are in seconds. This is the combined execution time of each thread. This is not the execution time. This is the time the application executed on the different CPUs.

In the second graph, there is the time spent on switching only for the applications. This is calculated with the number of switches multiplied by the average delay of the switches. We print additional information on the x labels. There is the average delay in seconds. This is the average delay for the switches that perf reports. We also have the average switch time. This is the mean, which is also shown as the middle bar in the plot. The last information is the percentage of the execution time that is spent on switching. In the third graph, we see the time spent on switching during the whole execution time. This includes the time from the application of the second graph and every other program during the execution time. As in the previous graph, we show the average switch time on the X-axis. This makes it easier

to compare the two values. The values of the Y-axis of the second and third graph is the time in seconds.

In the fourth graph, we see the number of switches for each application on the Y-axis. And in the last graph, the total amount of switches during the execution time. This includes the switches for the application.

4.2.1 Measurements of LavaMD

In this section, we see the results for the measurements of the LavaMD application. In figures 4.1-4.3 we see the utilisation of all 20 CPU's with free threads. Then in figures 4.4-4.6 we see the results of 20 threads pinned to a CPU. The results for 16 threads are in figures 4.7-4.9. And in 4.10-4.12 we have the measurements for 10 threads on one socket. For each configuration, we have the results for the three different scheduling techniques static, guided and dynamic. Each experiment has 100 samples.
We observe that node eight has a slightly higher execution time than other nodes with 16 or ten threads (figures 4.7-4.12). This is not so, with 20 threads.

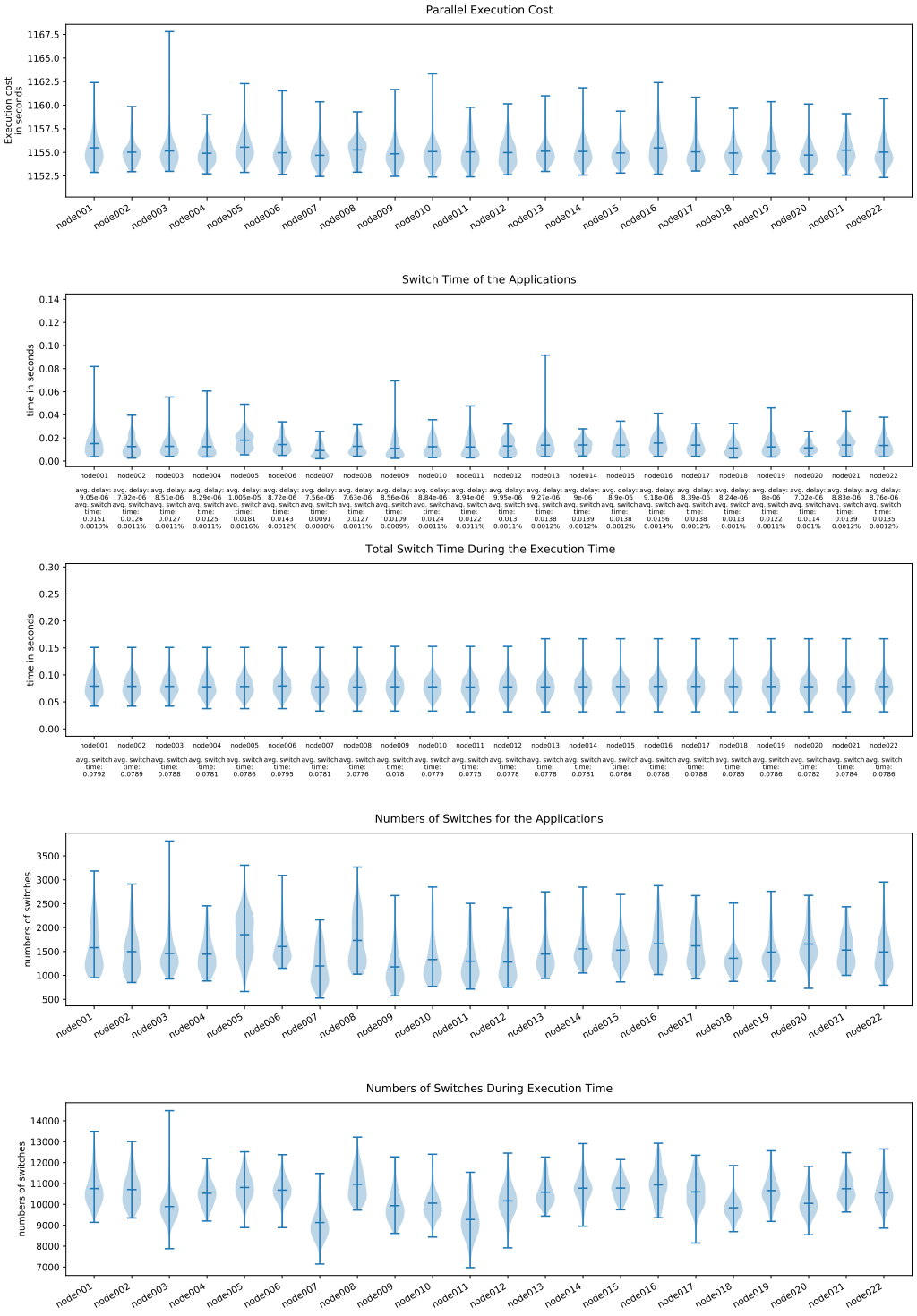

Figure 4.1: Measurements for the application LavaMD from the rodinia benchmark. scheduler static, 20 free threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. In this graphs, we have very few differences in the nodes. There are also only a few outliers. Only the total number of switches (5. graph) of nodes seven and 11 is less than on the other nodes.

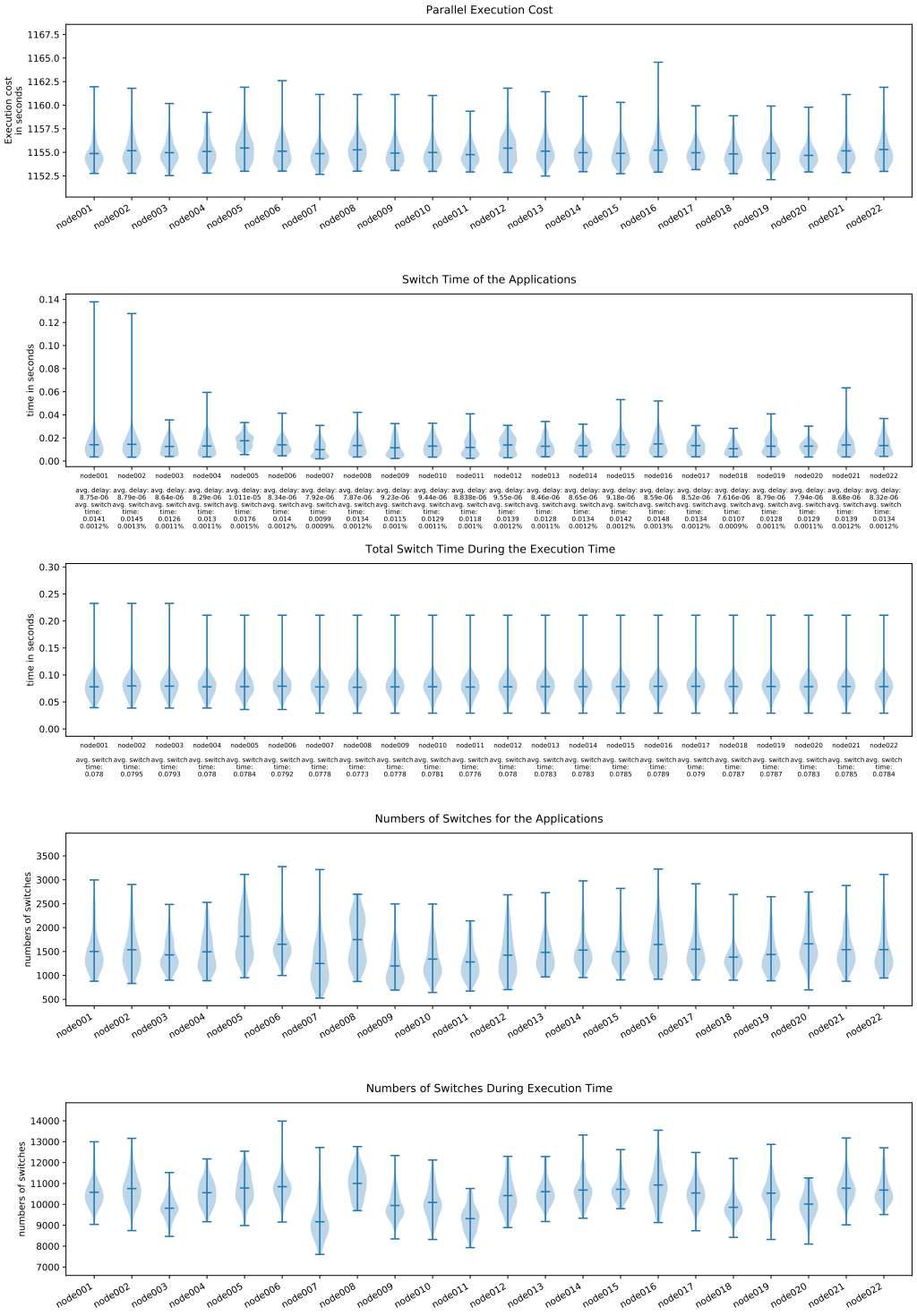

Figure 4.2: The measurements for the application LavaMD from the rodinia benchmark. scheduler guided, 20 free threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application,, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Here we see in the switch time of the application (2. graph) some outliers in nodes one and two. Also, the deviation of the total switch time during the execution is higher than with the static scheduler, but it is similar for all nodes. Again the number of switches during the execution (5. graph) of node seven is lower than that of other nodes.

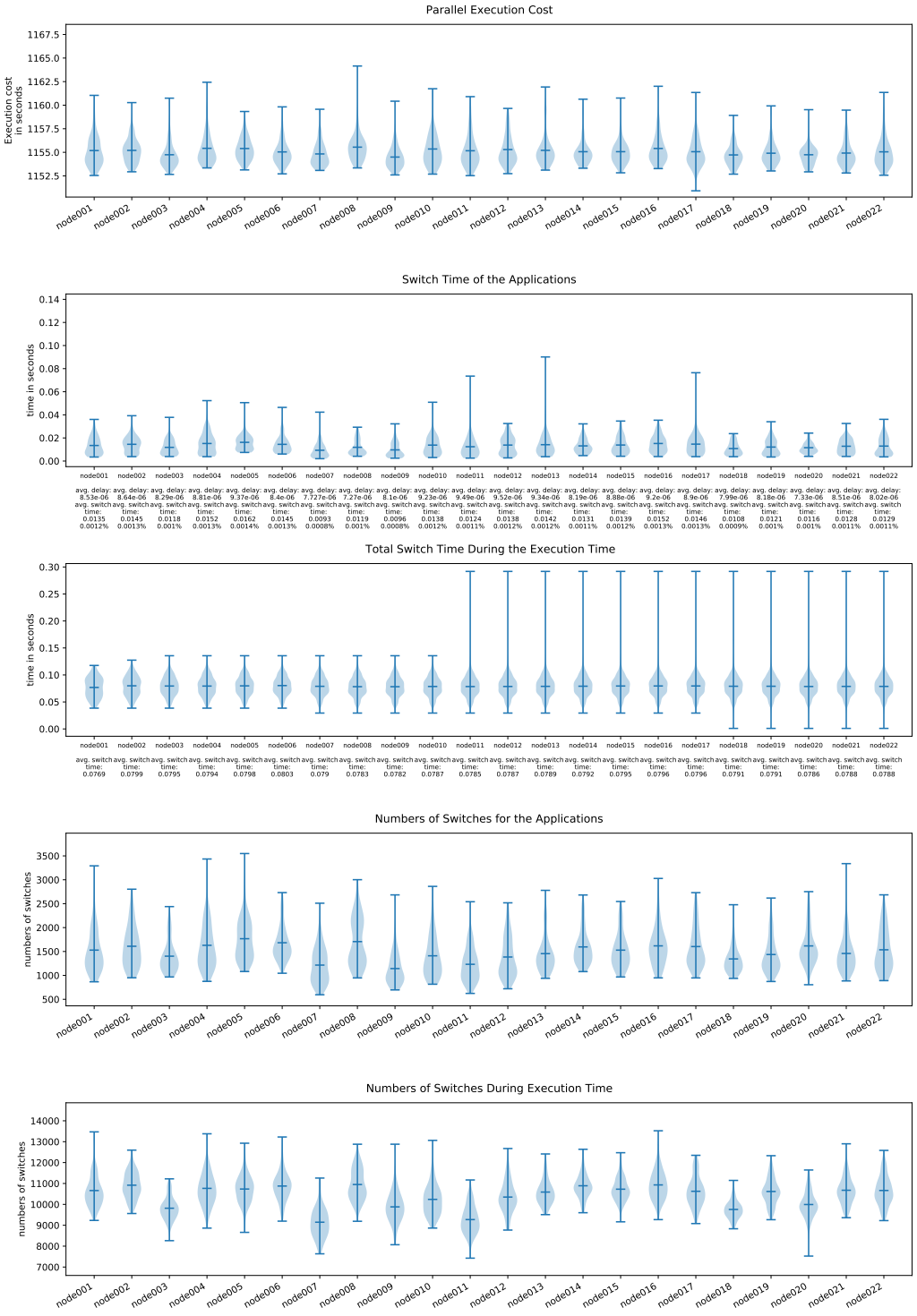

Figure 4.3: The measurements for the application LavaMD from the rodinia benchmark. scheduler dynamic,24, 20 free threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application,, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We see outliers on nodes 11 - 22 in the total switch time during the execution (3. graph). These outliers seem very similar on each of those nodes. As with the static and guided scheduler, the total number of switches (5. graph) of nodes seven and 11 is less than on the other nodes.

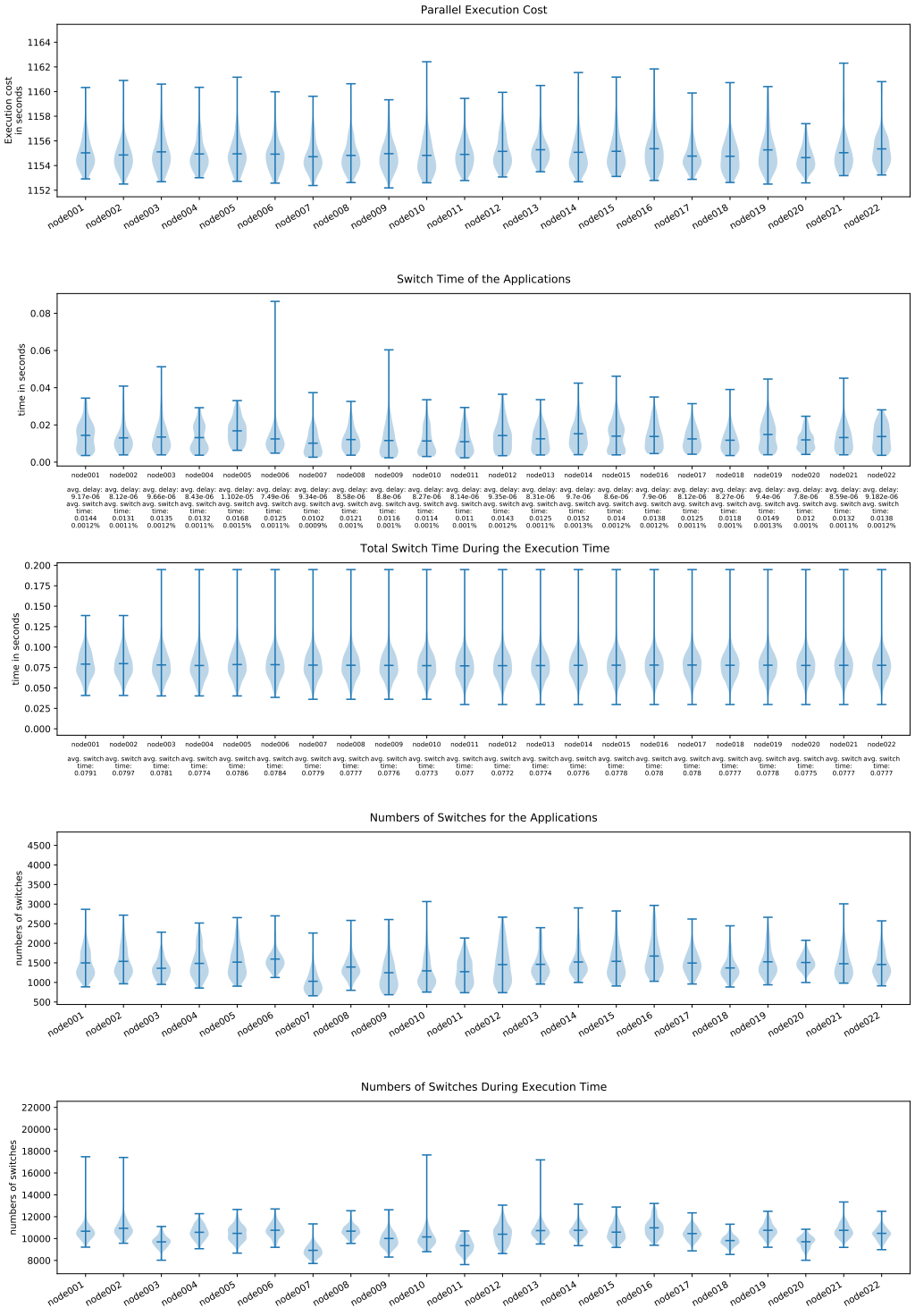

Figure 4.4: The measurements for the application LavaMD from the rodinia benchmark. scheduler static, 20 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Here node one and two are interesting. Both have less deviation in the total switch time during the execution (3. graph) and both have outliers in the number of switches during the execution (5. graph), in contrast to most other nodes.

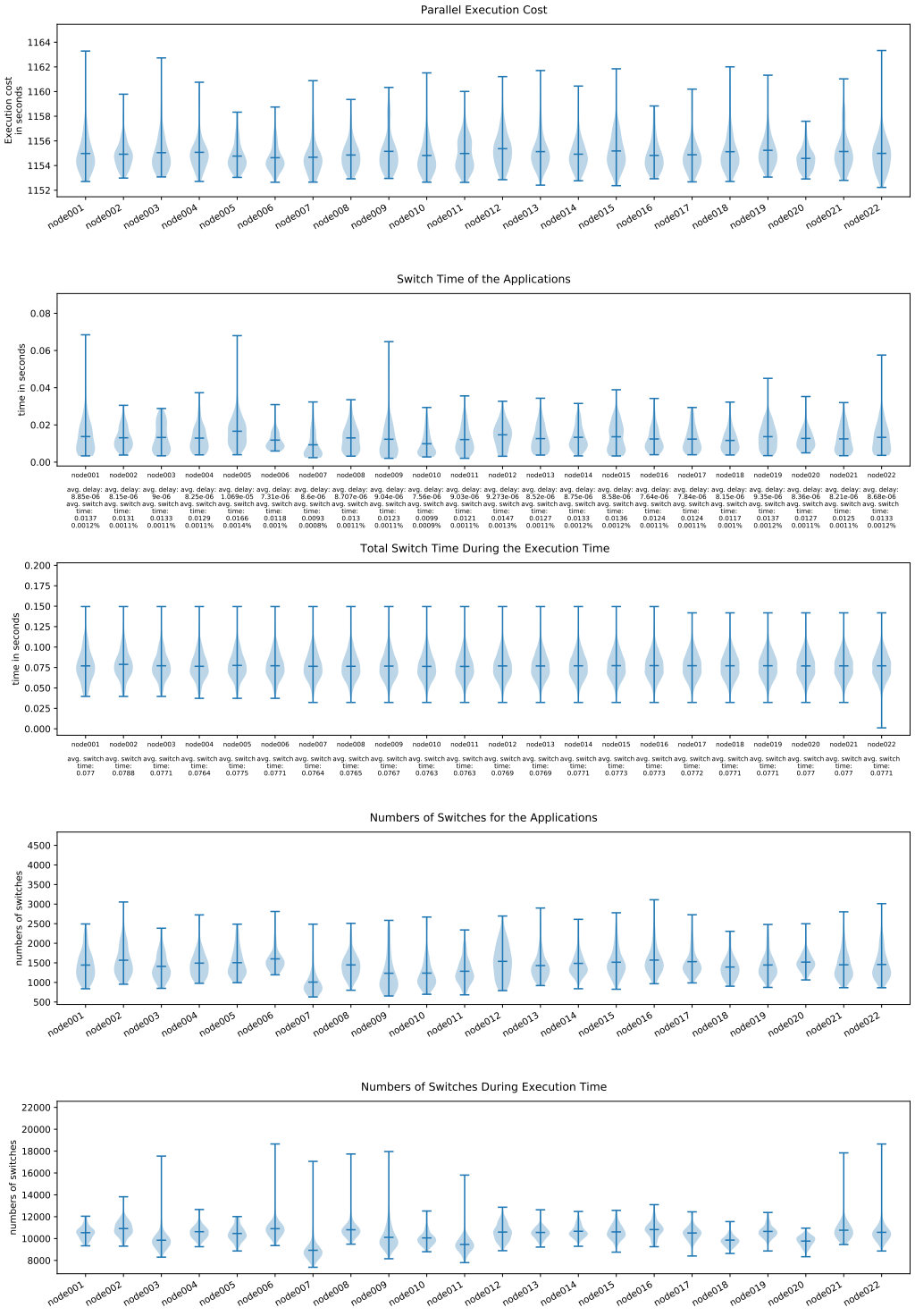

Figure 4.5: The measurements for the application LavaMD from the rodinia benchmark. scheduler guided, 20 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We observe that only some nodes have outliers in switch time of the application (2. graph) and numbers of switches during the execution (5. graph). Node seven has again fewer switches, both for the application and the whole system (graph four and fife).

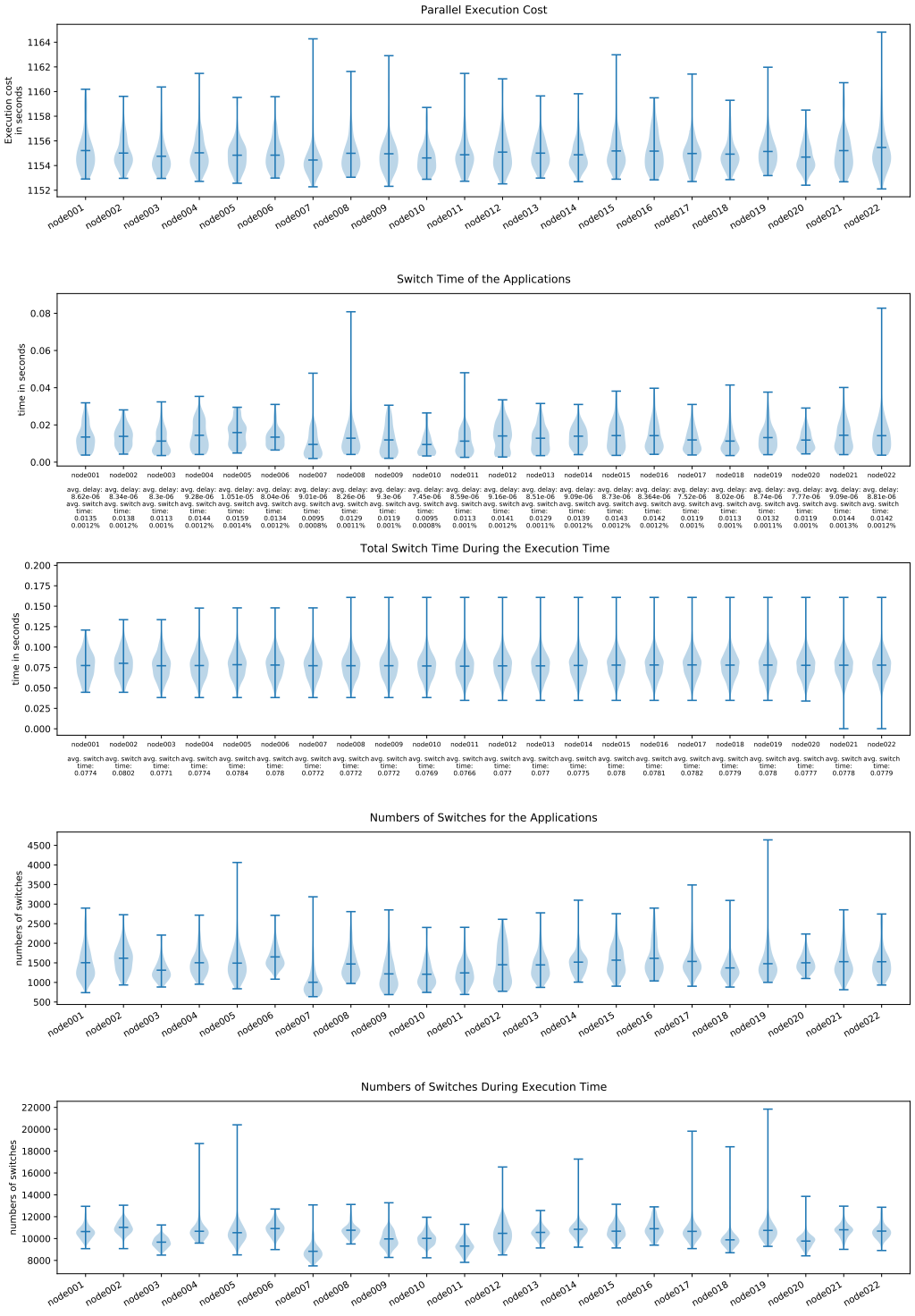

Figure 4.6: The measurements for the application LavaMD from the rodinia benchmark. scheduler dynamic,24, 20 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Interesting is that node five and 19 have outliers in the numbers of switches both for the application and during the execution time (4. and 5. graph), but for both, there are no outliers in the switch times. Again node seven has fewer switches than other nodes.

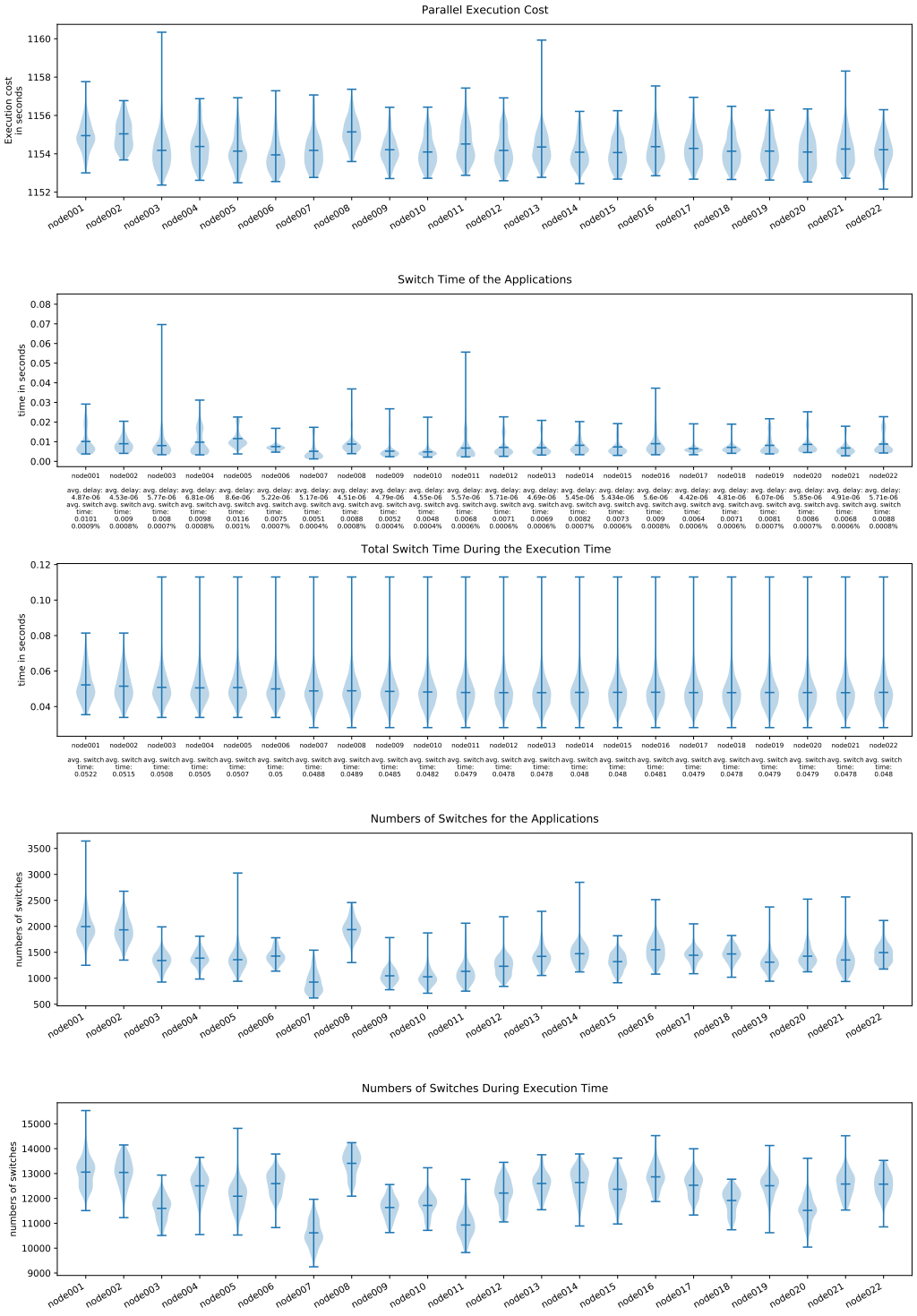

Figure 4.7: The measurements for the application LavaMD from the rodinia benchmark. scheduler static, 16 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. As in figure 4.4 node one and two have less deviation in the total switch time during the execution. All other nodes lock very similarly. Nodes one, two and eight have more switches and a slightly higher execution time.

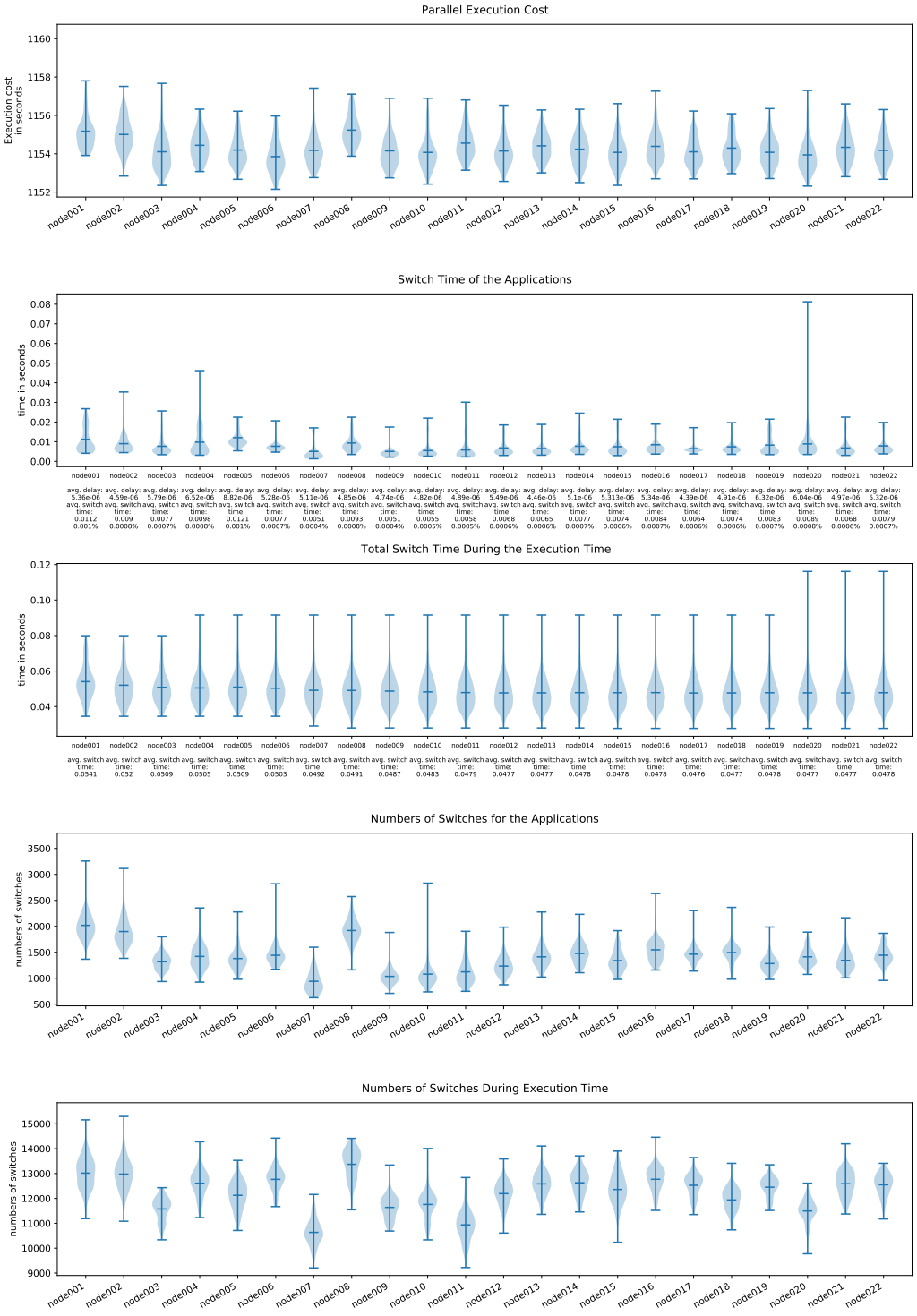

Figure 4.8: The measurements for the application LavaMD from the rodinia benchmark. scheduler guided, 16 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We see again an interesting pattern in the total switch time during the execution. Certain nodes have similar outliers. Also, there are noticeable differences in the number of switches on different nodes.

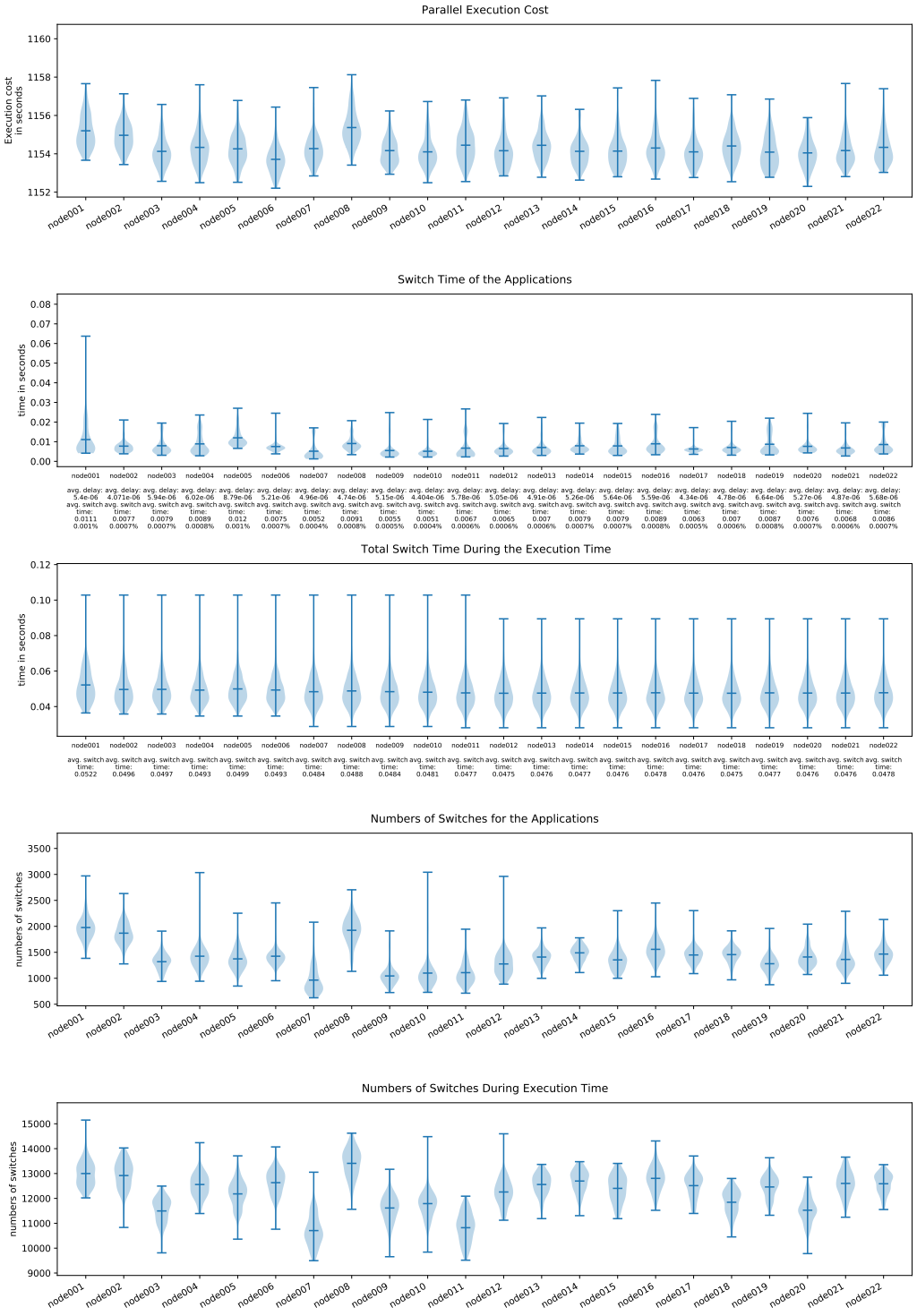

Figure 4.9: The measurements for the application LavaMD from the rodinia benchmark. scheduler dynamic,24, 16 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We observe again that groups of nodes have similar outliers in the total switch time during the execution.

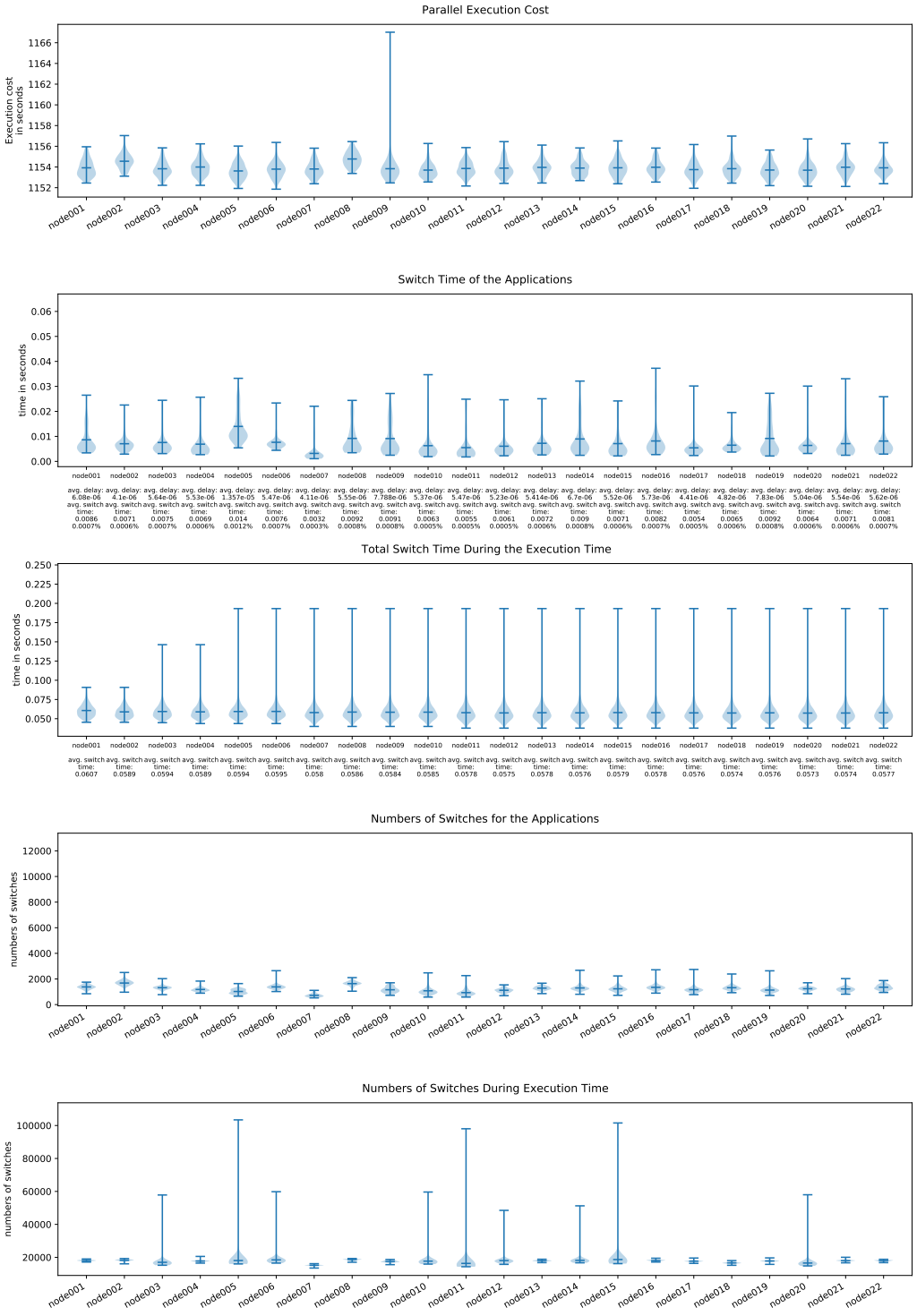

Figure 4.10: The measurements for the application LavaMD from the rodinia benchmark. scheduler static, 10 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. There is an extreme outlier in the execution time on node nine. Also, we have several outliers on different nodes in the total number of switches during the execution.

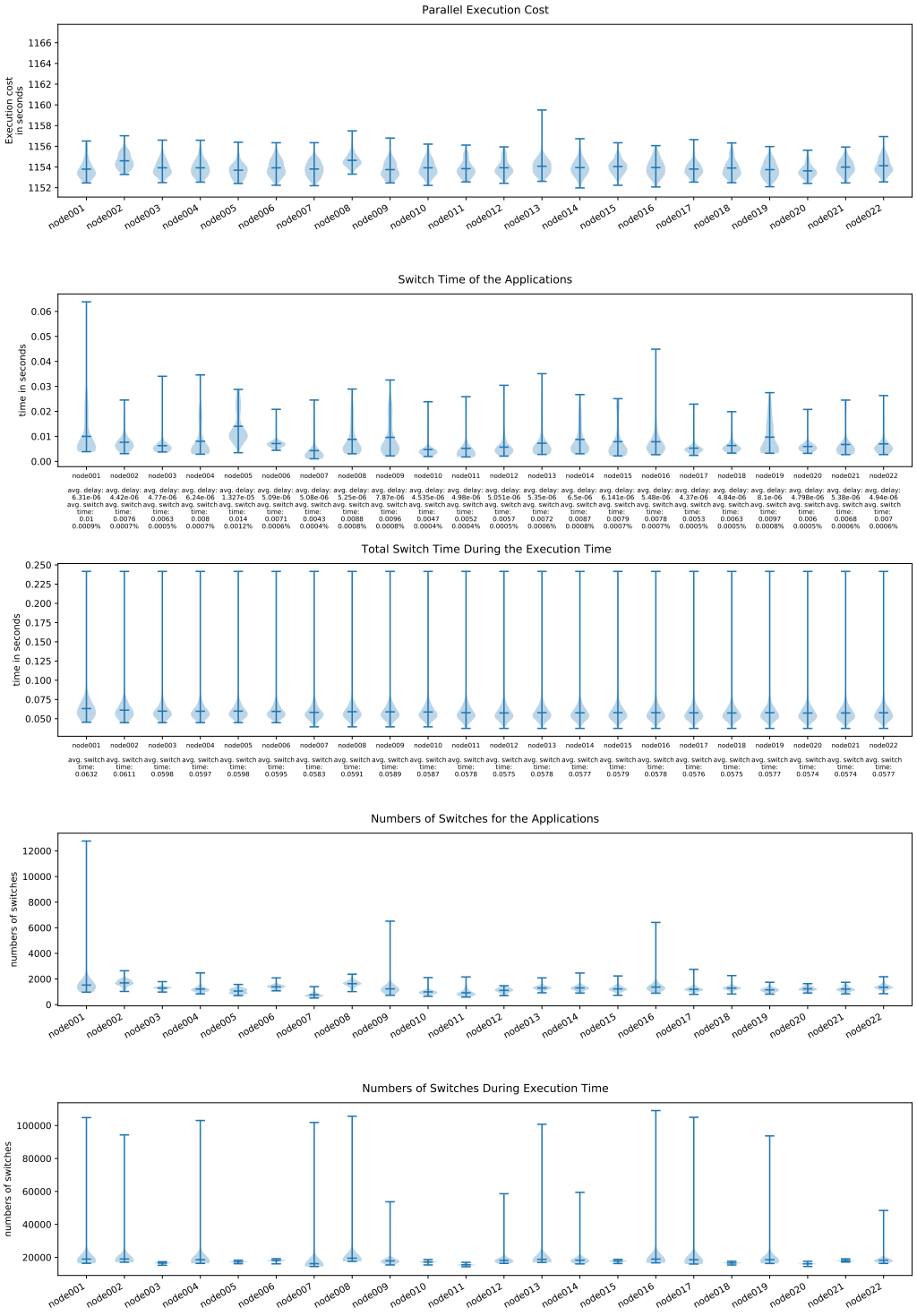

Figure 4.11: The measurements for the application LavaMD from the rodinia benchmark. scheduler guided, 10 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. The average switch time of the application differs between nodes. For node fife, it is 0.014 seconds and node ten has only 0.0047 seconds. Also, there are many outliers in the number of switches.

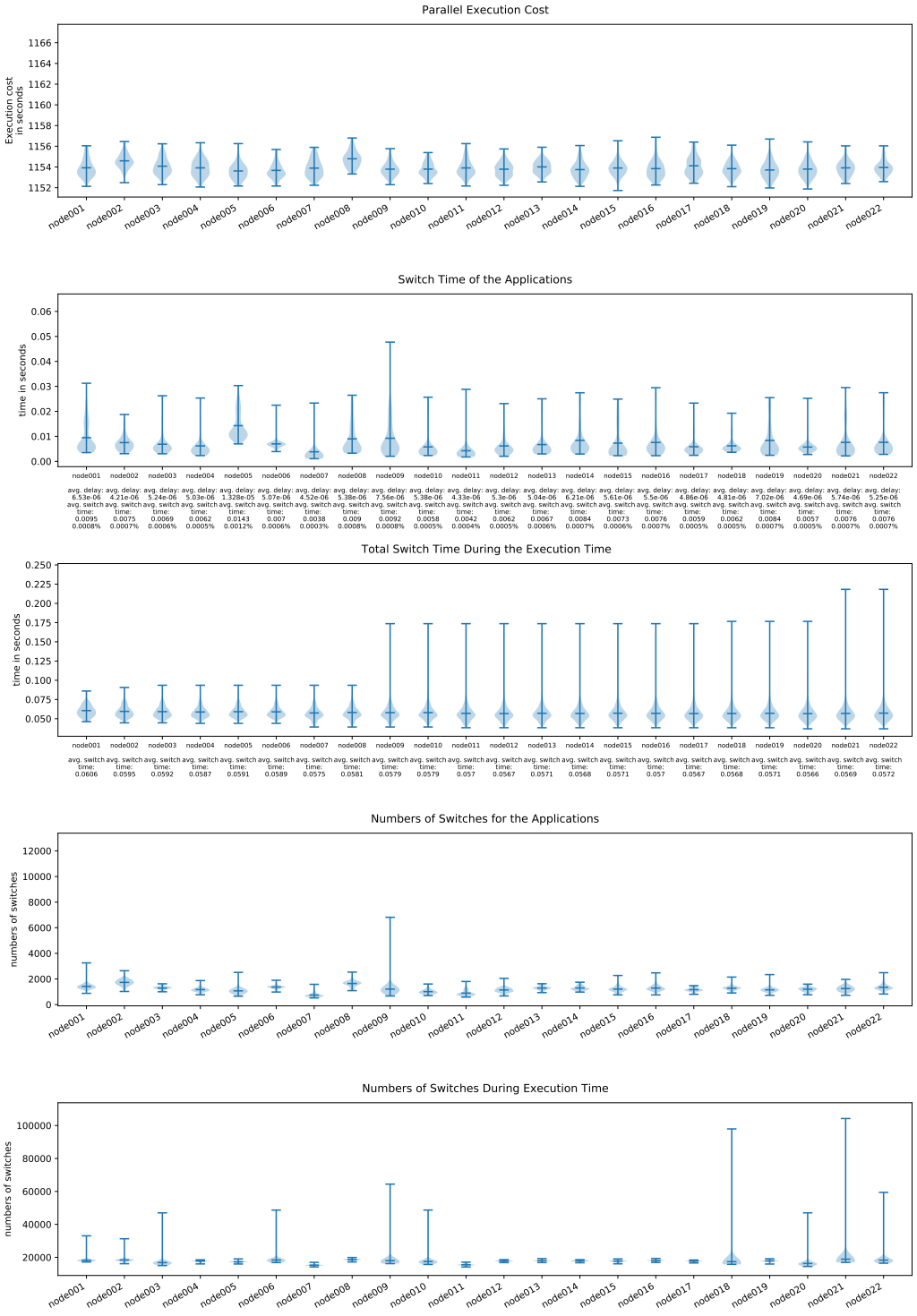

Figure 4.12: The measurements for the application LavaMD from the rodinia benchmark. scheduler dynamic,24, 10 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We have similar differences in the switch time of the application as in figure 4.11 Also, there is a similar pattern as in previous graphs, in the outliers of the total switch time during the execution.

4.2.2 Measurements of Hotspot3D

In this section we have the results of Hotspot3D. For each experiment there are 100 samples. In figures 4.13-4.15 we have the results for 20 free threads. The result for 20 pinned threads are in 4.16-4.18. In the figures 4.19-4.21 there are the measurements for 16 threads and in 4.22-4.24 there are the results for eight threads. Hotspot3D is different from the other applications. It has only eight loop iterations. Therefore it has eight threads that execute in parallel. Every thread more can not be used and is idle. This is the reason why we have only eight threads in the last experiments instead of ten.

We observe that in the parallel execution cost there are two groups. One around 13.9 seconds and the other at 13.6 seconds. This is the case for all scheduling techniques and all thread configurations. A similar grouping exists for the number of switches, both for the application and the whole system. But we do not see this pattern for the switch time.

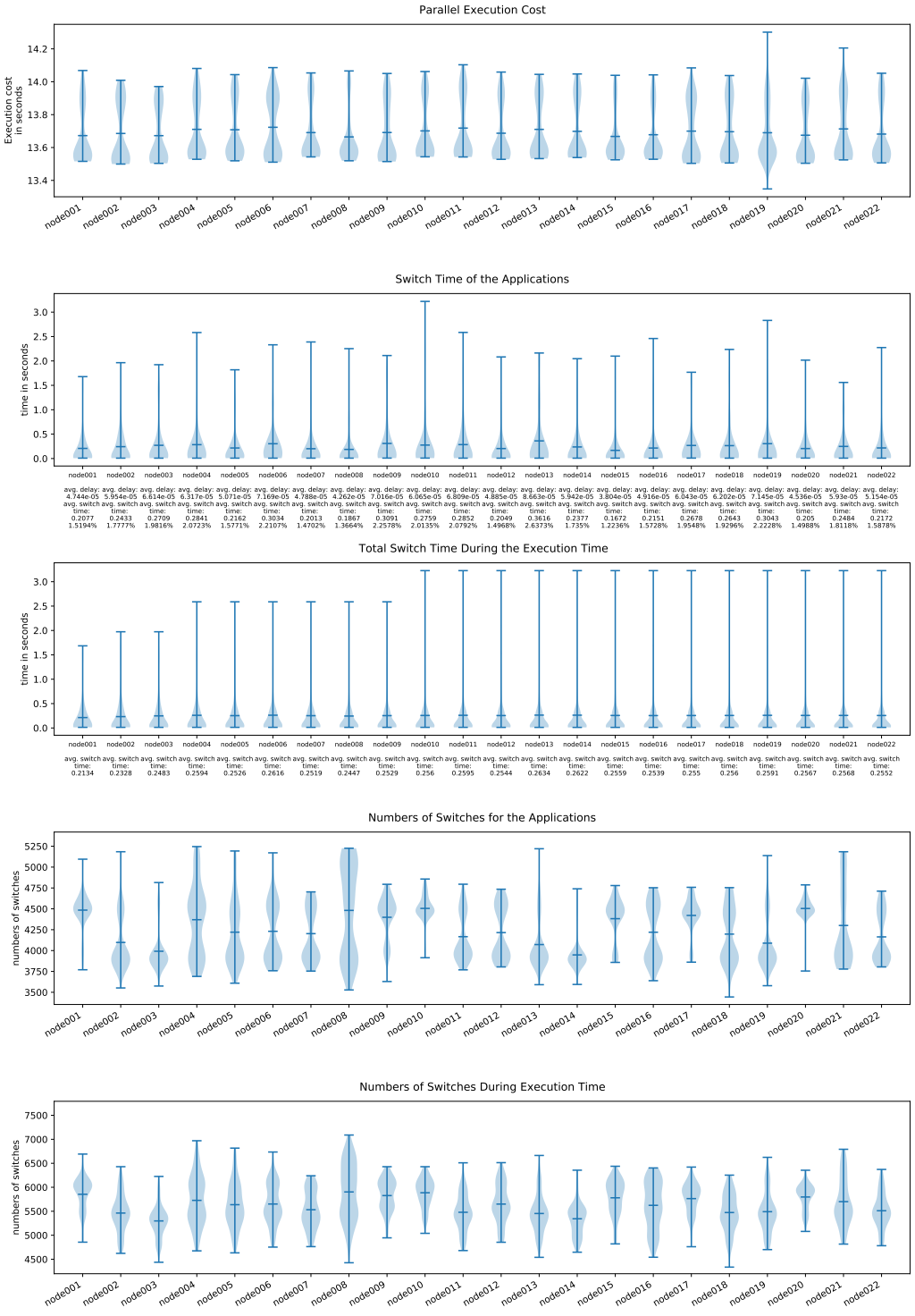

Figure 4.13: The measurements for the benchmark Hotspot3D, scheduler static, 20 free threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. The average switch time is similar between the nodes. But all nodes have extreme outliers. Also interesting is the lower outlier of the execution time on node 19.

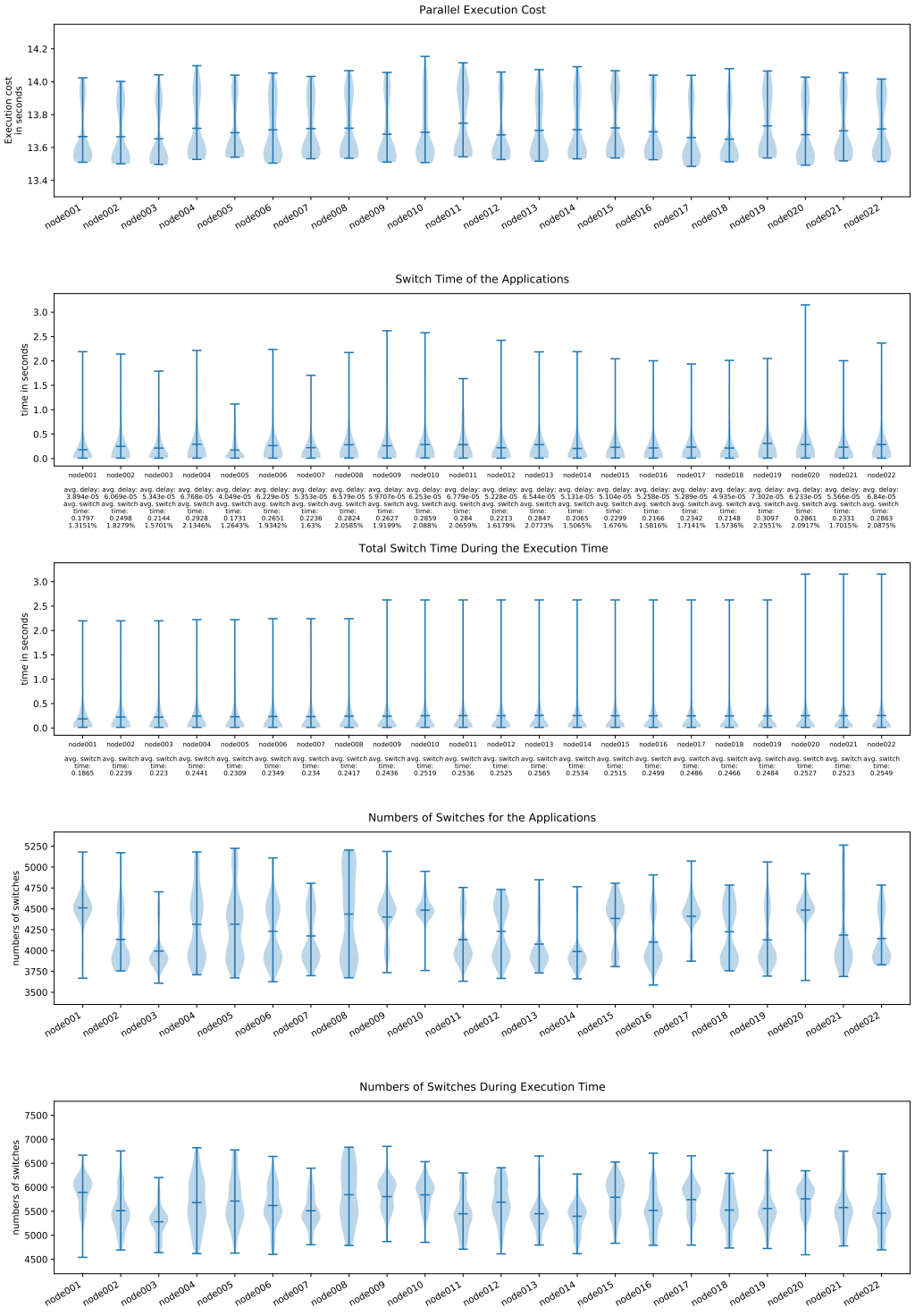

Figure 4.14: The measurements for the benchmark Hotspot3D, scheduler guided, 20 free threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

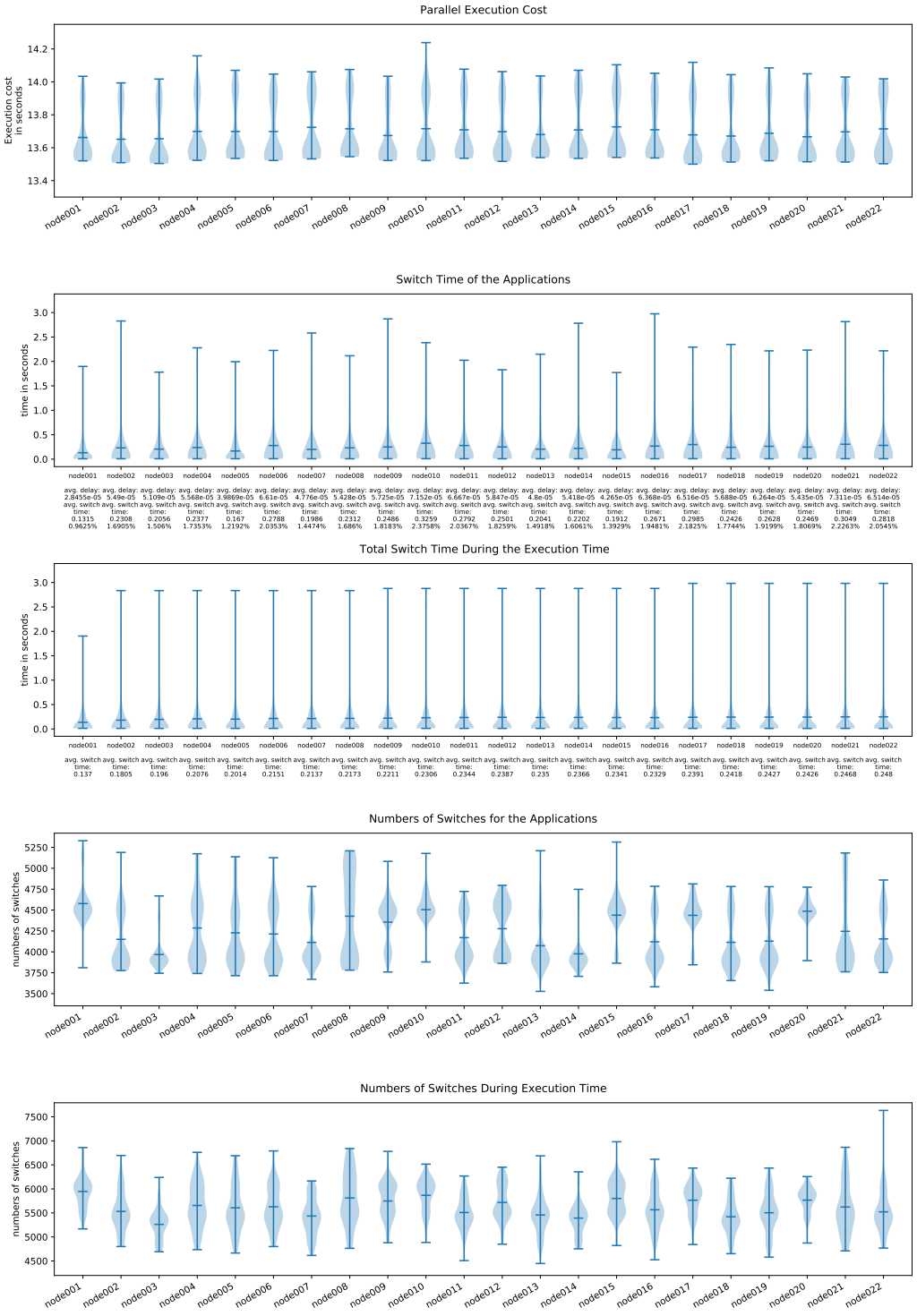

Figure 4.15: The measurements for the benchmark Hotspot3D, scheduler dynamic, 20 free threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. All nodes have similar outliers for the total switch time during the execution except node one. This is also noticeable in the average. Node 1 has an average switch time during the execution of 0.137 seconds. This is higher on other nodes.

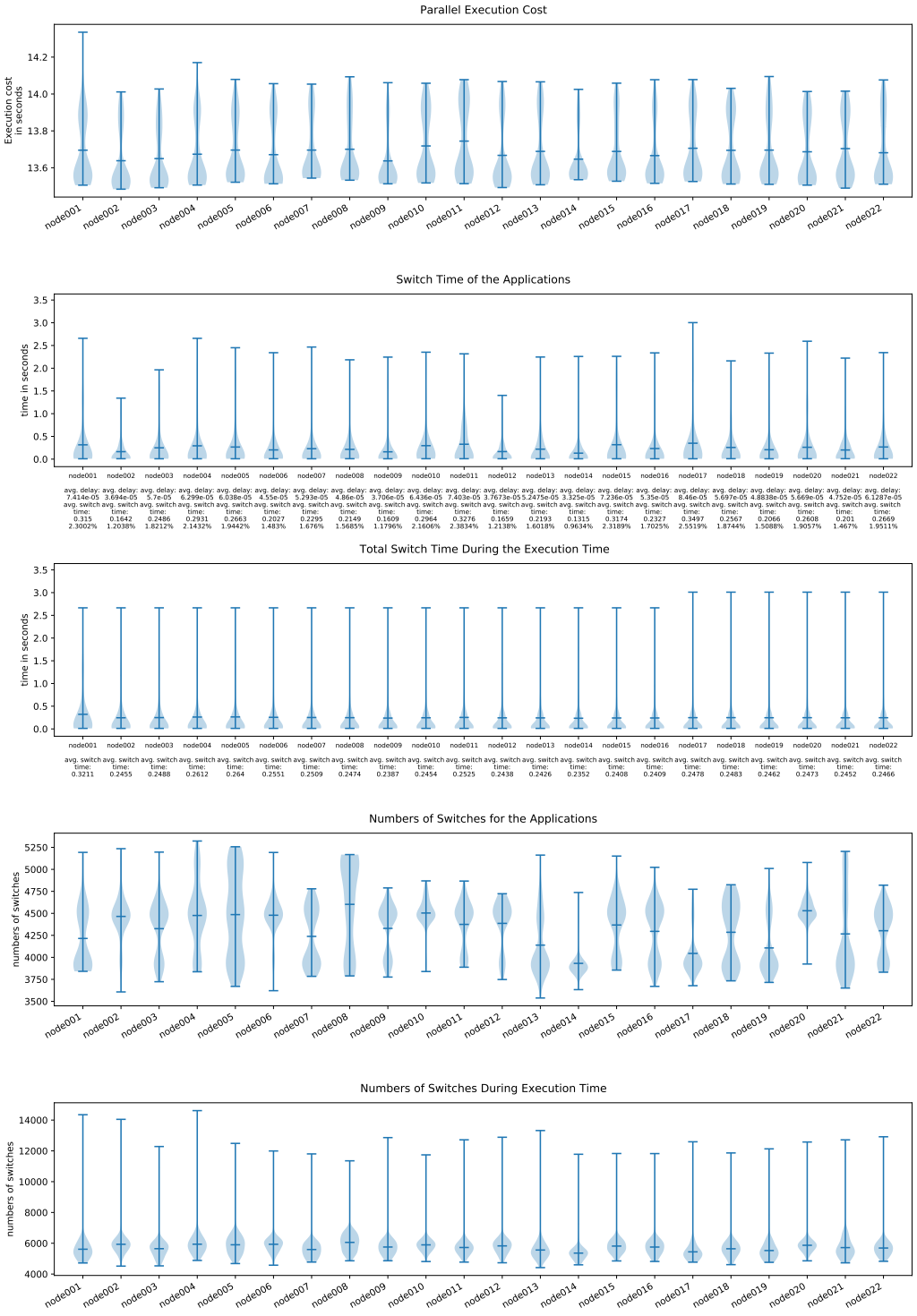

Figure 4.16: The measurements for the benchmark Hotspot3D, scheduler static, 20 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Here we do not have the grouping, that we observed in the previous plots, for the total number of switches during the execution. Although we see this grouping for the number of switches of the application. But there are extreme outliers with more than double the numbers of switches than the average.

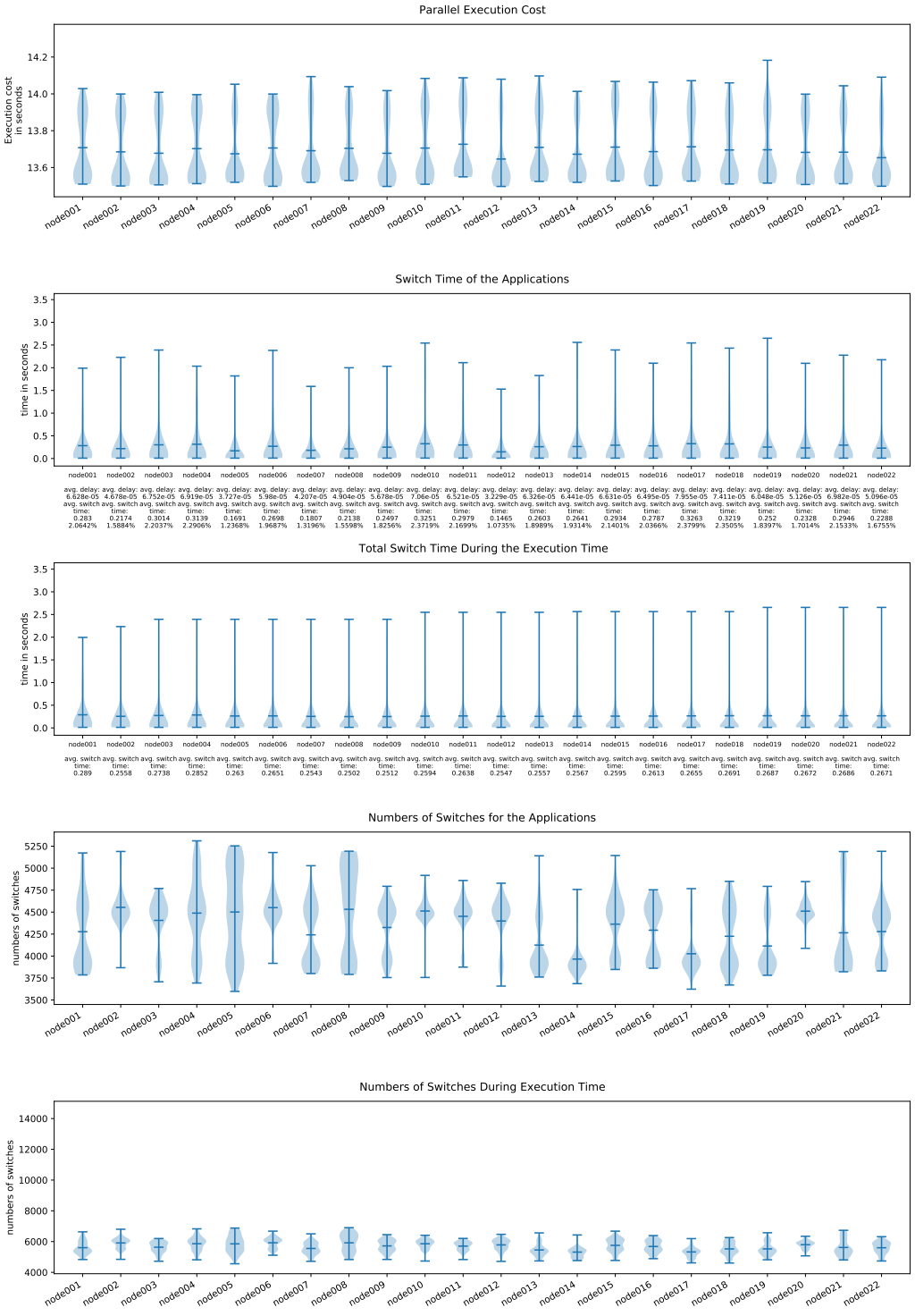

Figure 4.17: The measurements for the benchmark Hotspot3D, scheduler guided, 20 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. There are no such extreme outliers in the numbers of switches during the execution as in figure 4.16.

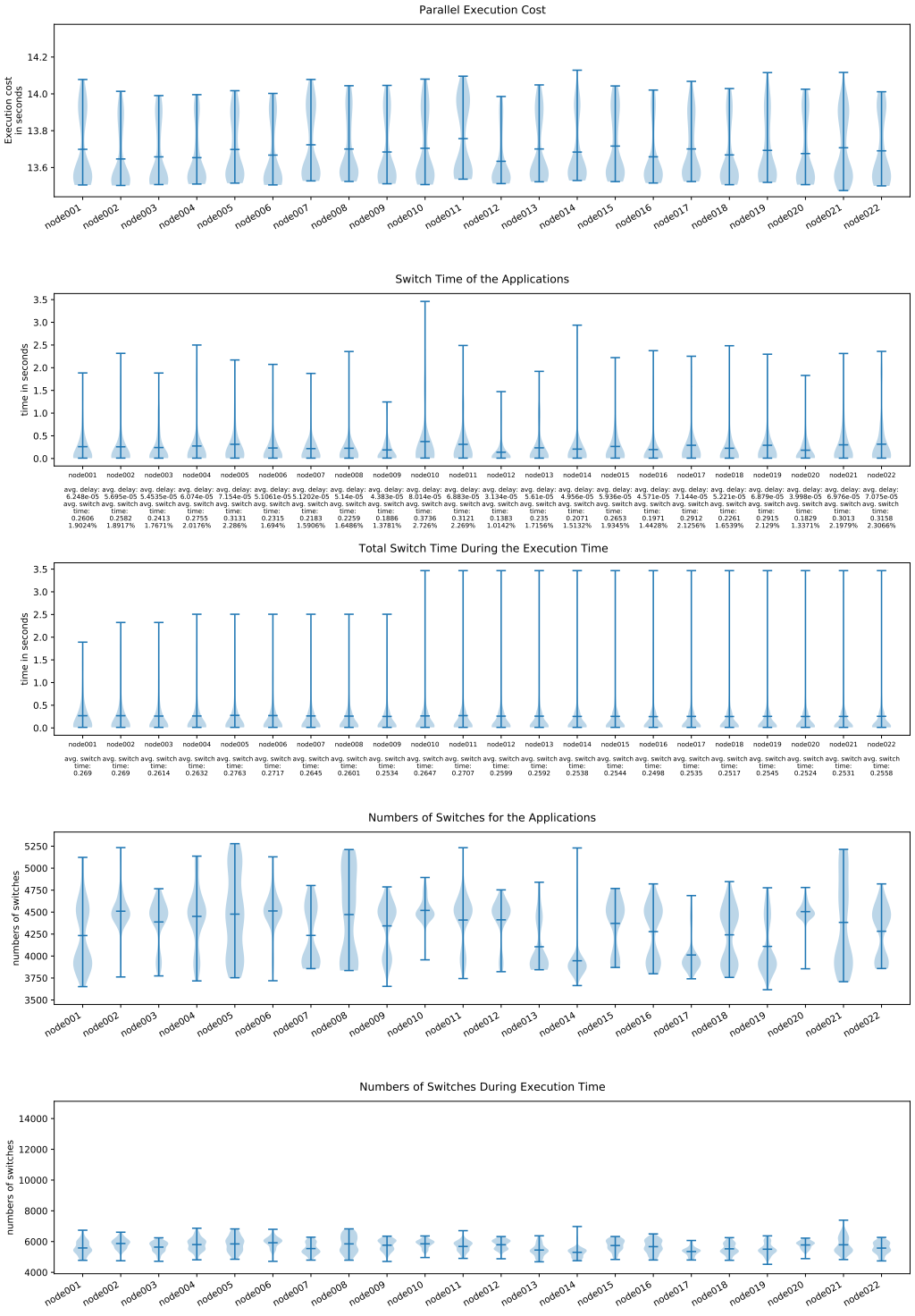

Figure 4.18: The measurements for the benchmark Hotspot3D, scheduler dynamic, 20 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We see again the similar outliers in the total switch time during the execution.

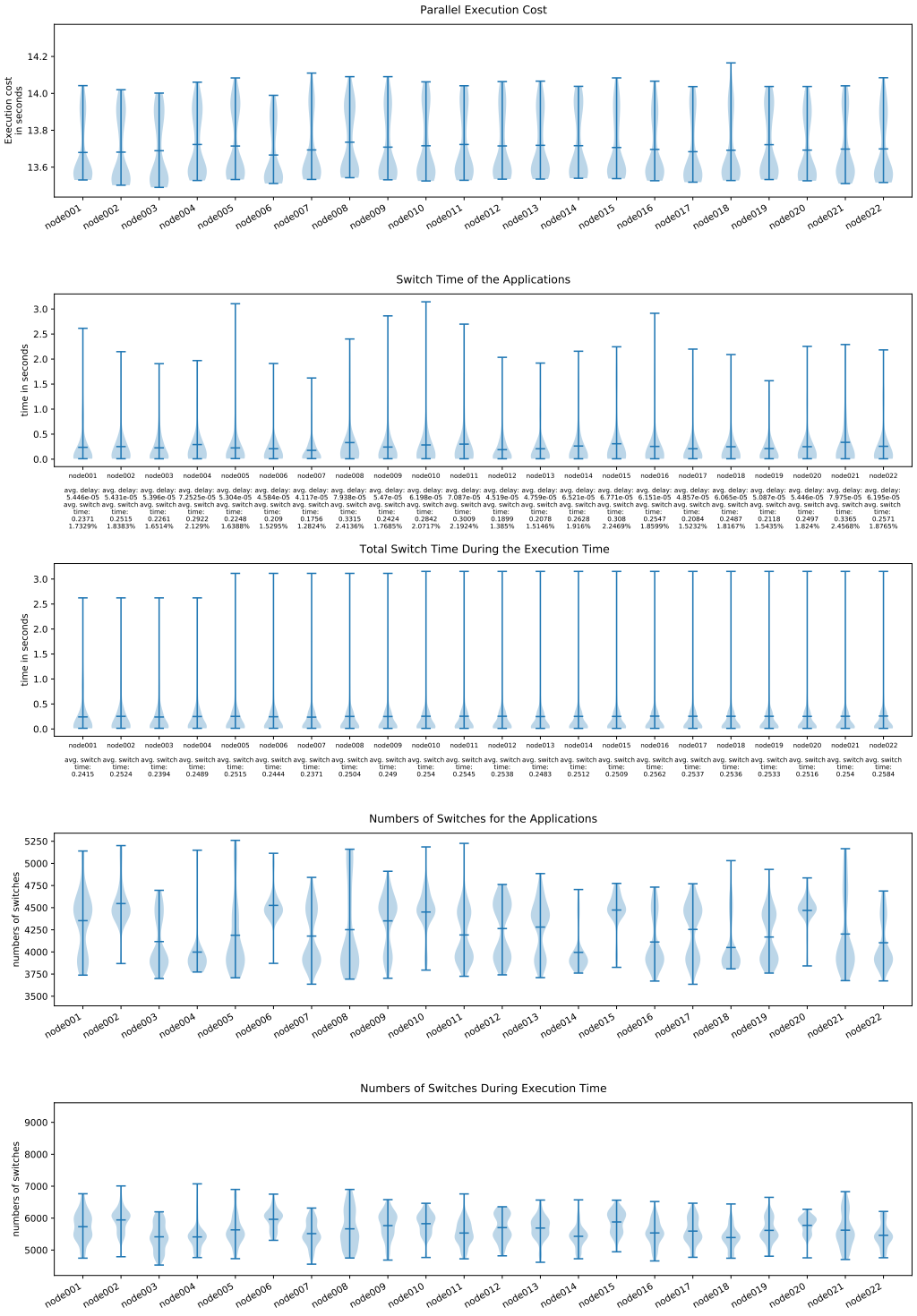

Figure 4.19: The measurements for the benchmark Hotspot3D, scheduler static, 16 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

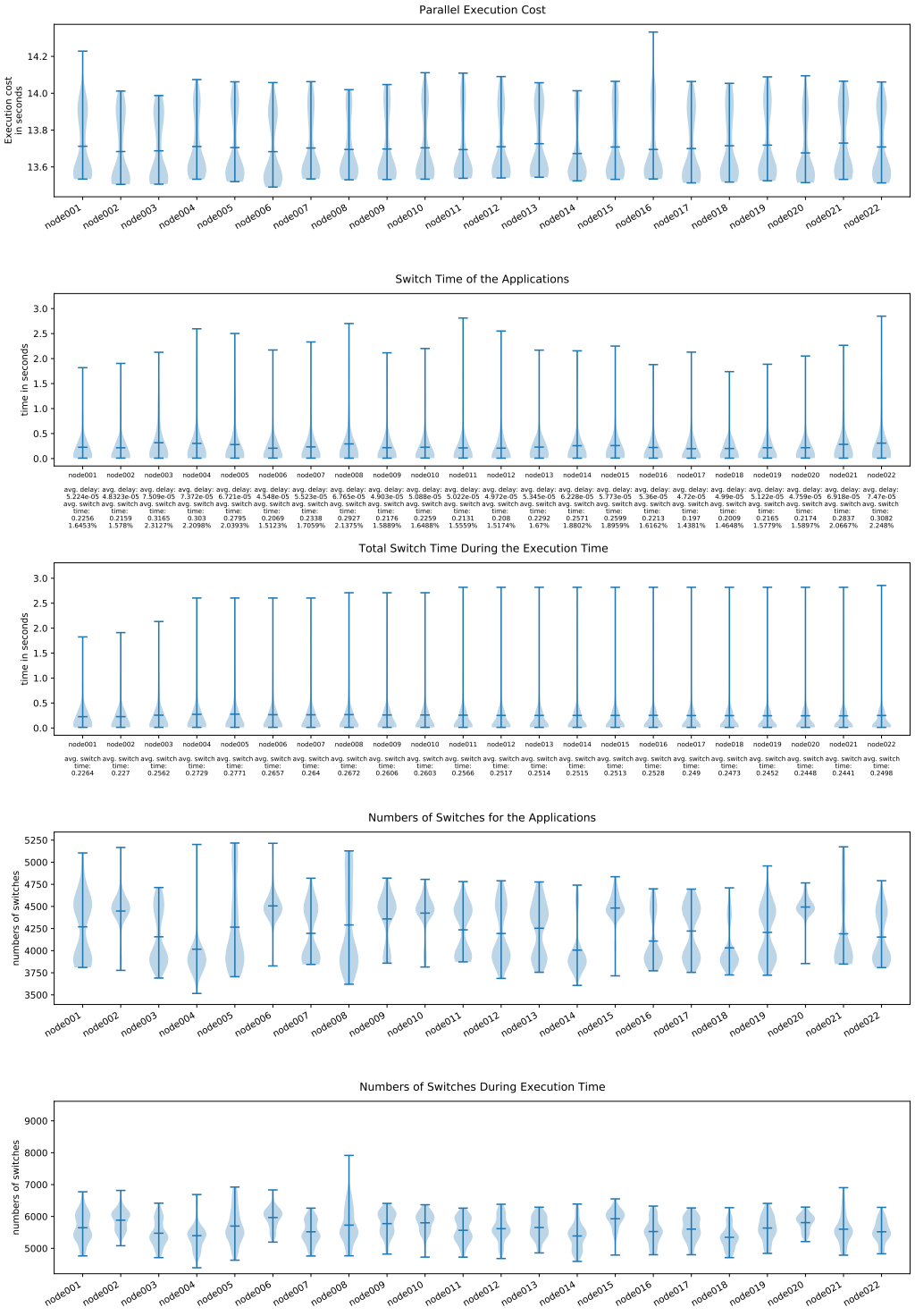

Figure 4.20: The measurements for the benchmark Hotspot3D, scheduler guided, 16 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

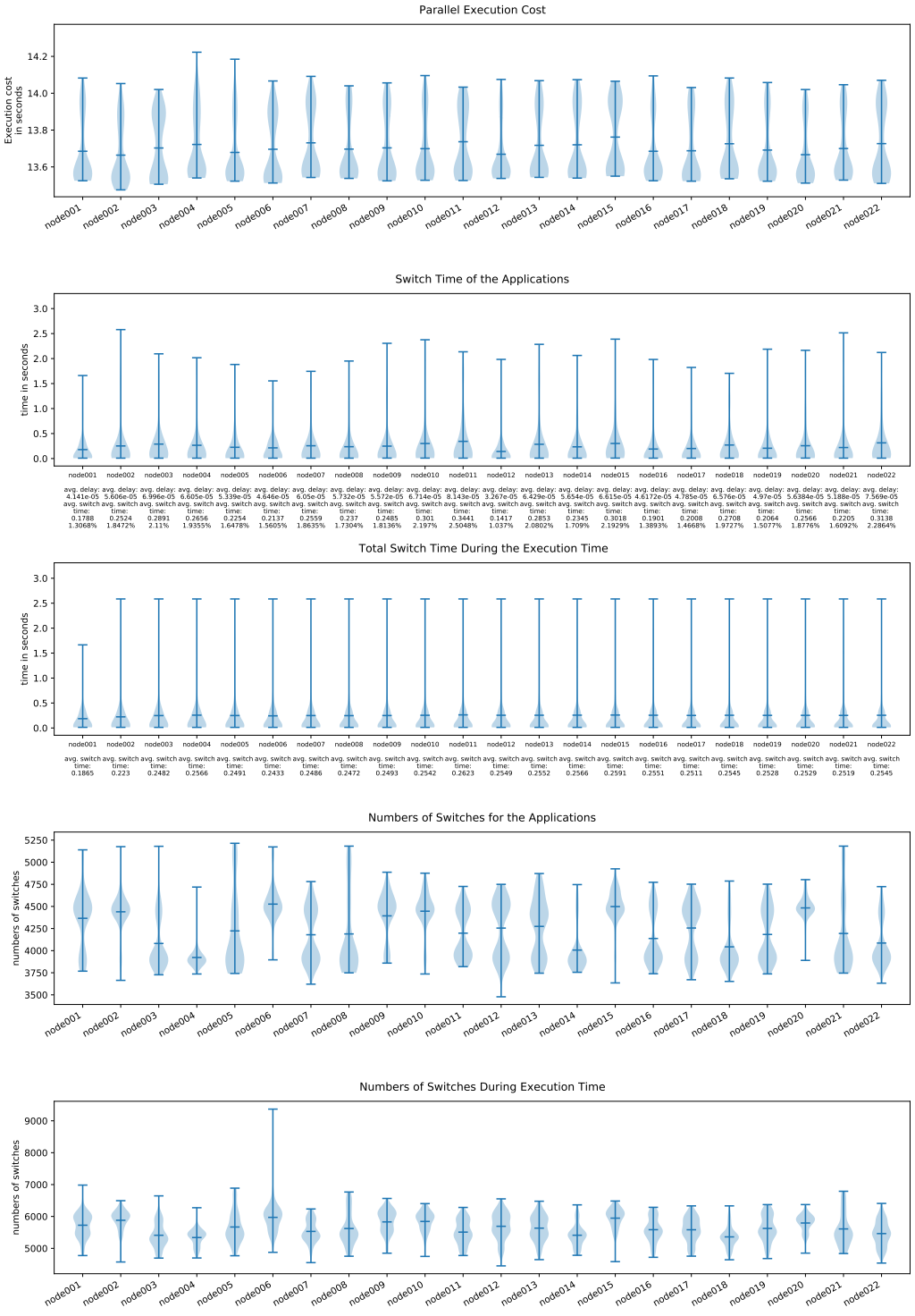

Figure 4.21: The measurements for the benchmark Hotspot3D, scheduler dynamic, 16 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We have one outlier in the numbers of switches during the execution on node six.

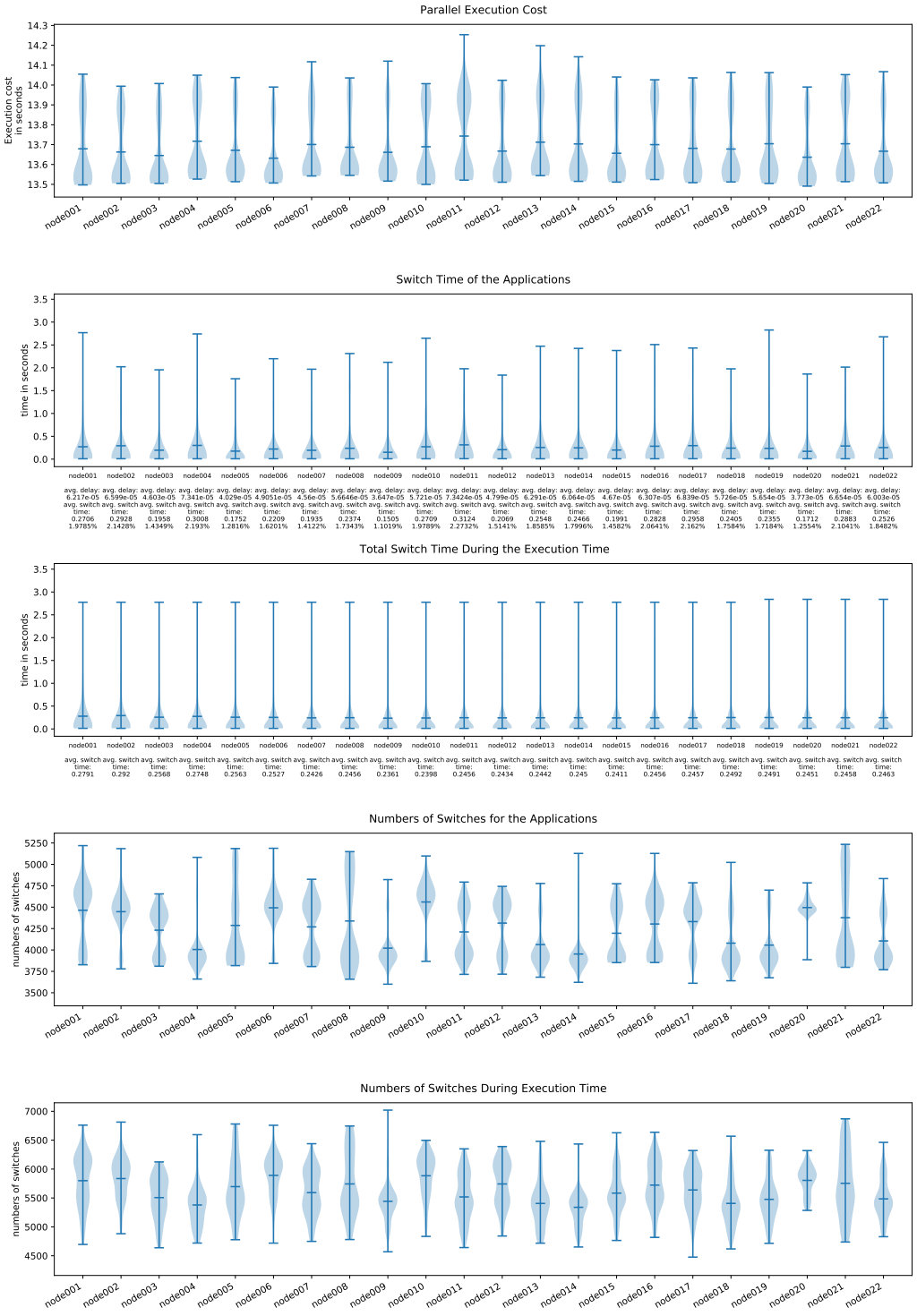

Figure 4.22: The measurements for the benchmark Hotspot3D, scheduler static, 8 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

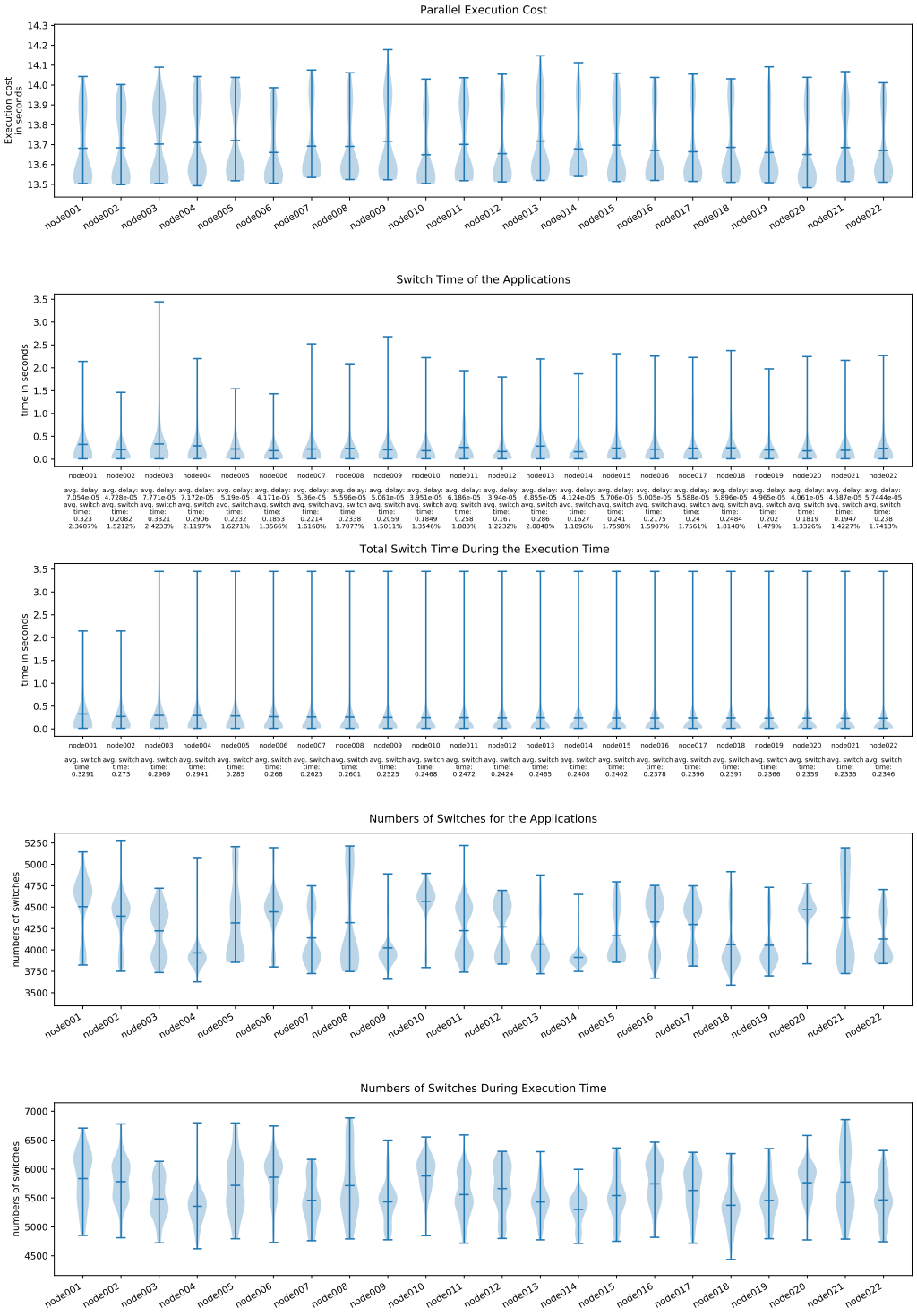

Figure 4.23: The measurements for the benchmark Hotspot3D, scheduler guided, 8 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

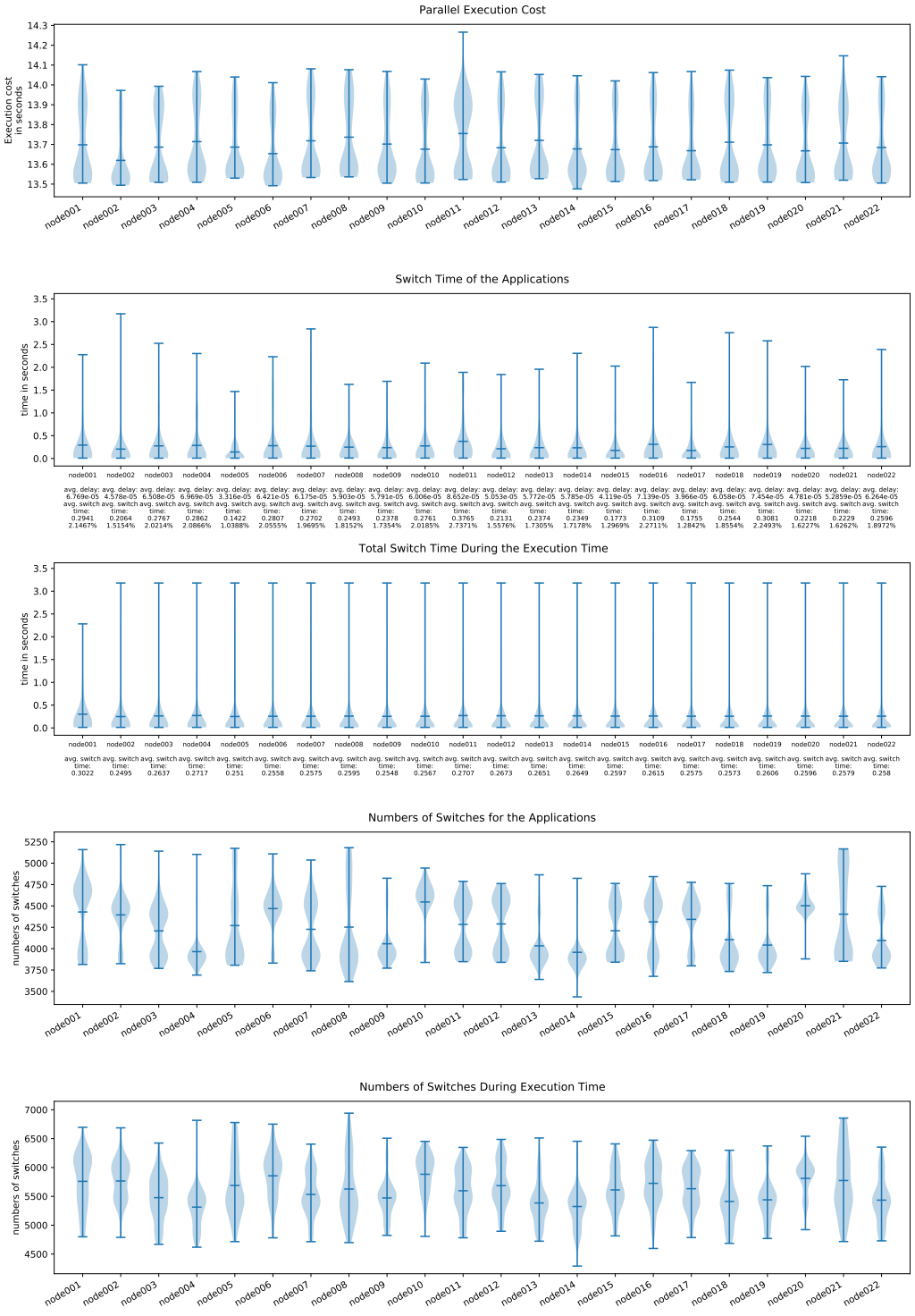

Figure 4.24: The measurements for the benchmark Hotspot3D, scheduler dynamic, 8 pinned threads, 100 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Node 11 has a higher parallel execution cost than the other nodes. But there is not a difference in the numbers of switches nor the switch time.

4.2.3 Measurements of SPH_EXA

Here are the results of SPH_EXA. There are only five samples for each experiment because this application takes much longer than LavaMD or Hotspot3D. This is a reason why the measurements differ more than those for LavaMD and Hotspot3D. In the figures 4.25-4.27 we have the measurements for 20 free threads. In 4.28-4.30 there are the results for 20 pinned threads, 4.31-4.33 the results for 16 threads and in 4.34-4.36 the results for 10 threads on a single socket.

It is striking that with 16 and ten threads, node1 has much less deviation and a much shorter total switching time than other nodes. This is also observable for node2, but there are more outliers.

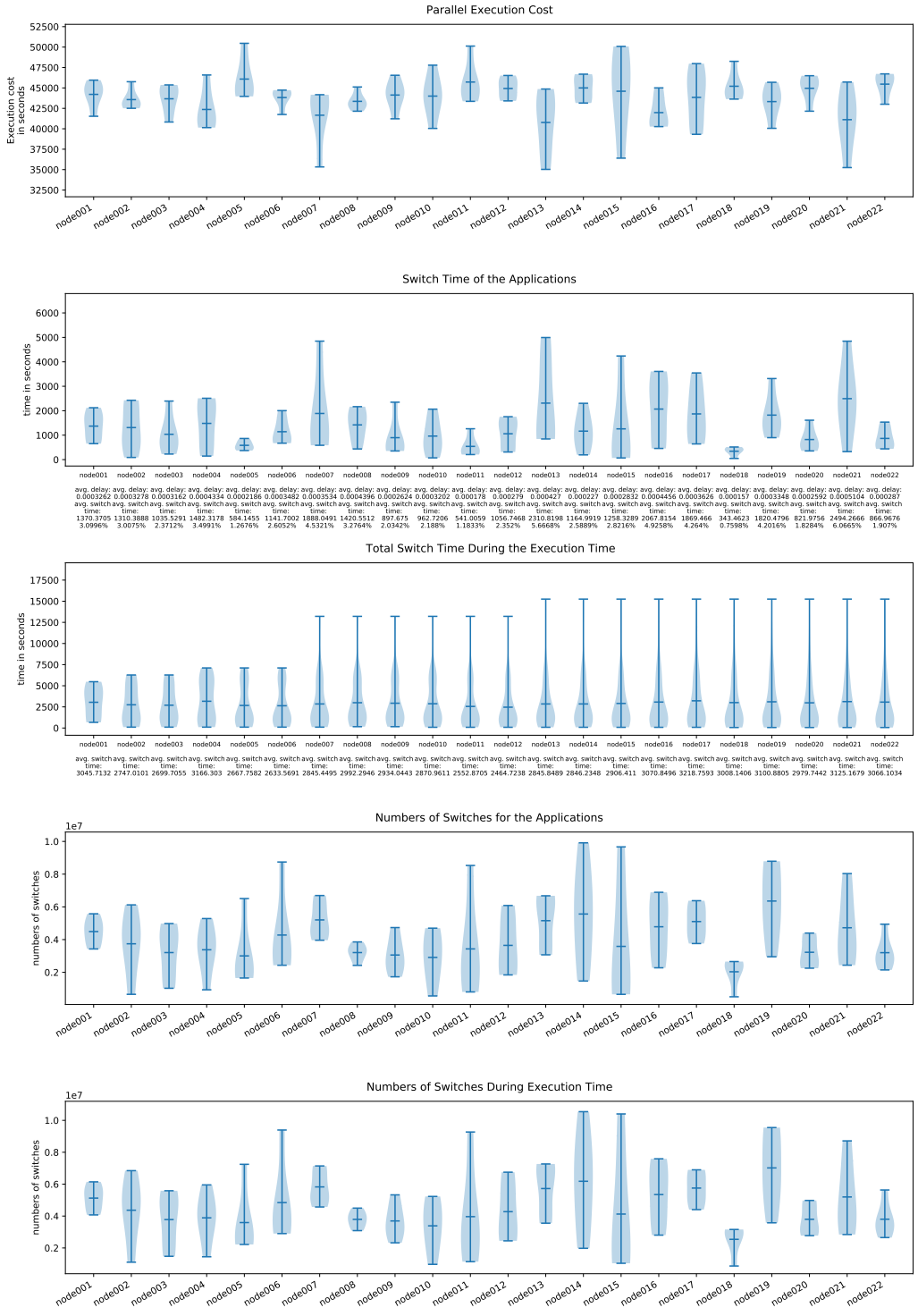

Figure 4.25: The measurements for the benchmark SPH_EXA, scheduler static, 20 free threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. The parallel execution cost varies much between the different nodes. Only the average switch time during the execution has similar results for all nodes.

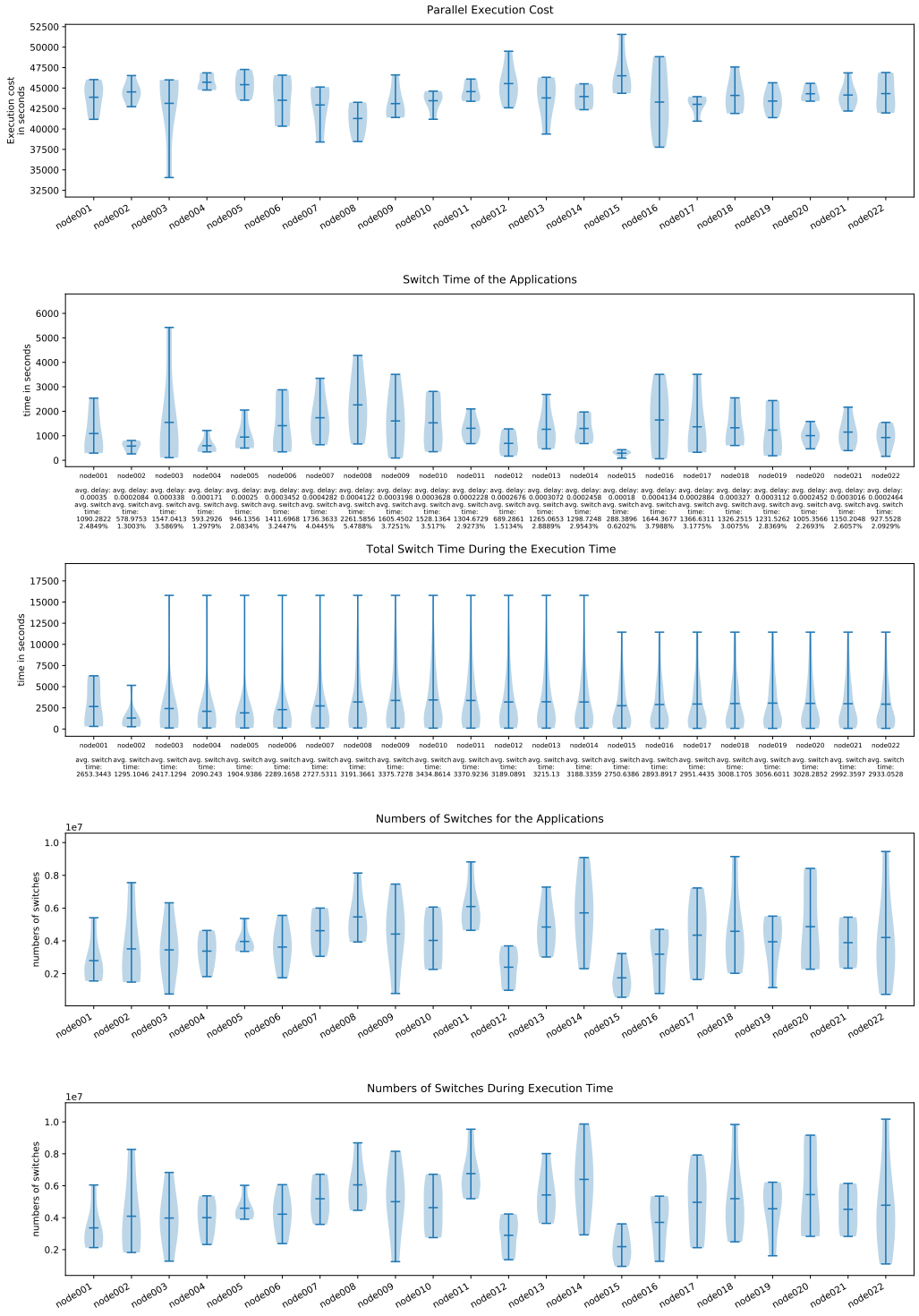

Figure 4.26: The measurements for the benchmark SPH_EXA, scheduler guided, 20 free threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

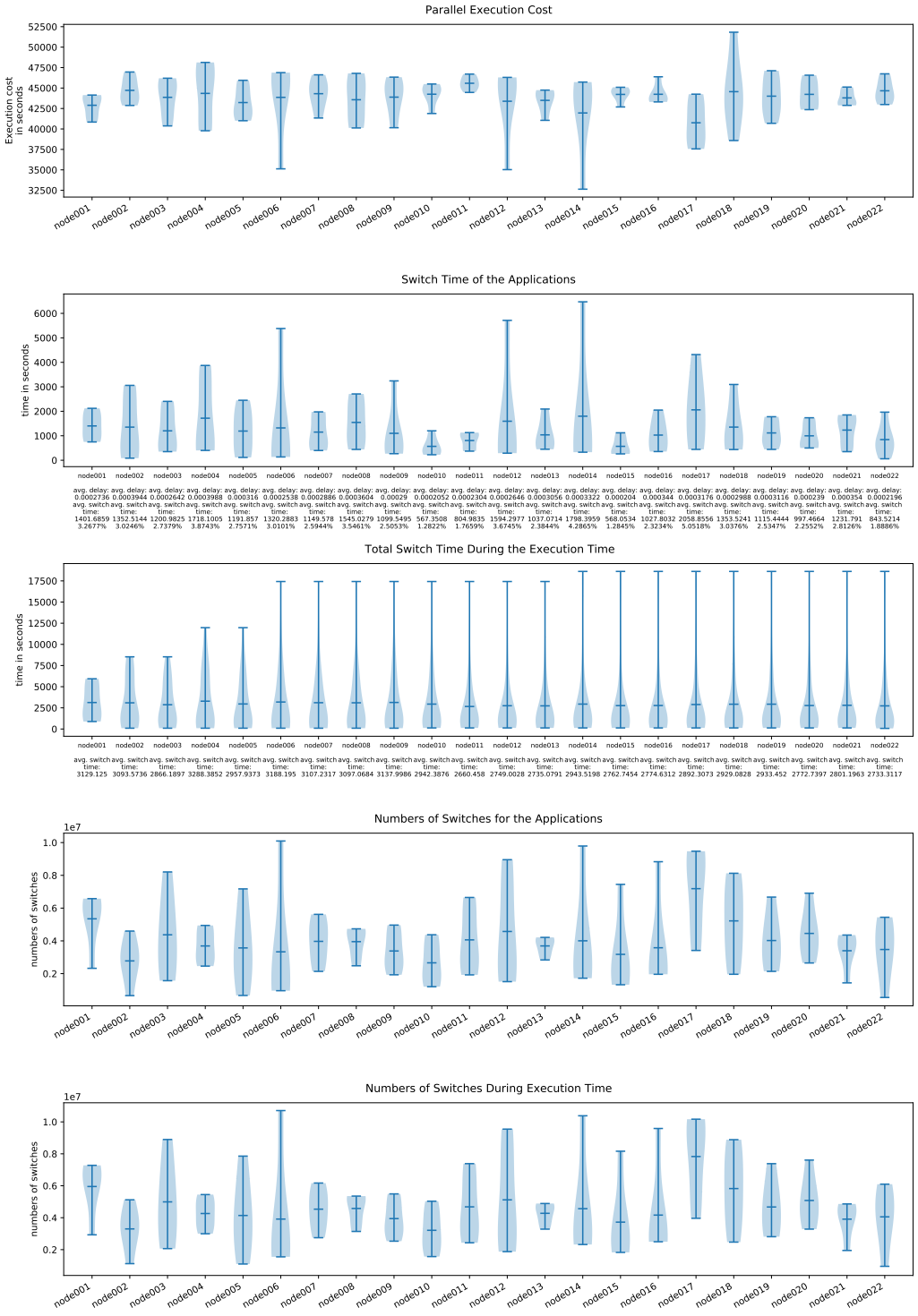

Figure 4.27: The measurements for the benchmark SPH_EXA, scheduler dynamic,24, 20 free threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. With the dynamic scheduler the means of the parallel execution cost are much closer together than with the previous scheduling techniques. But the number of switches differs just the same.

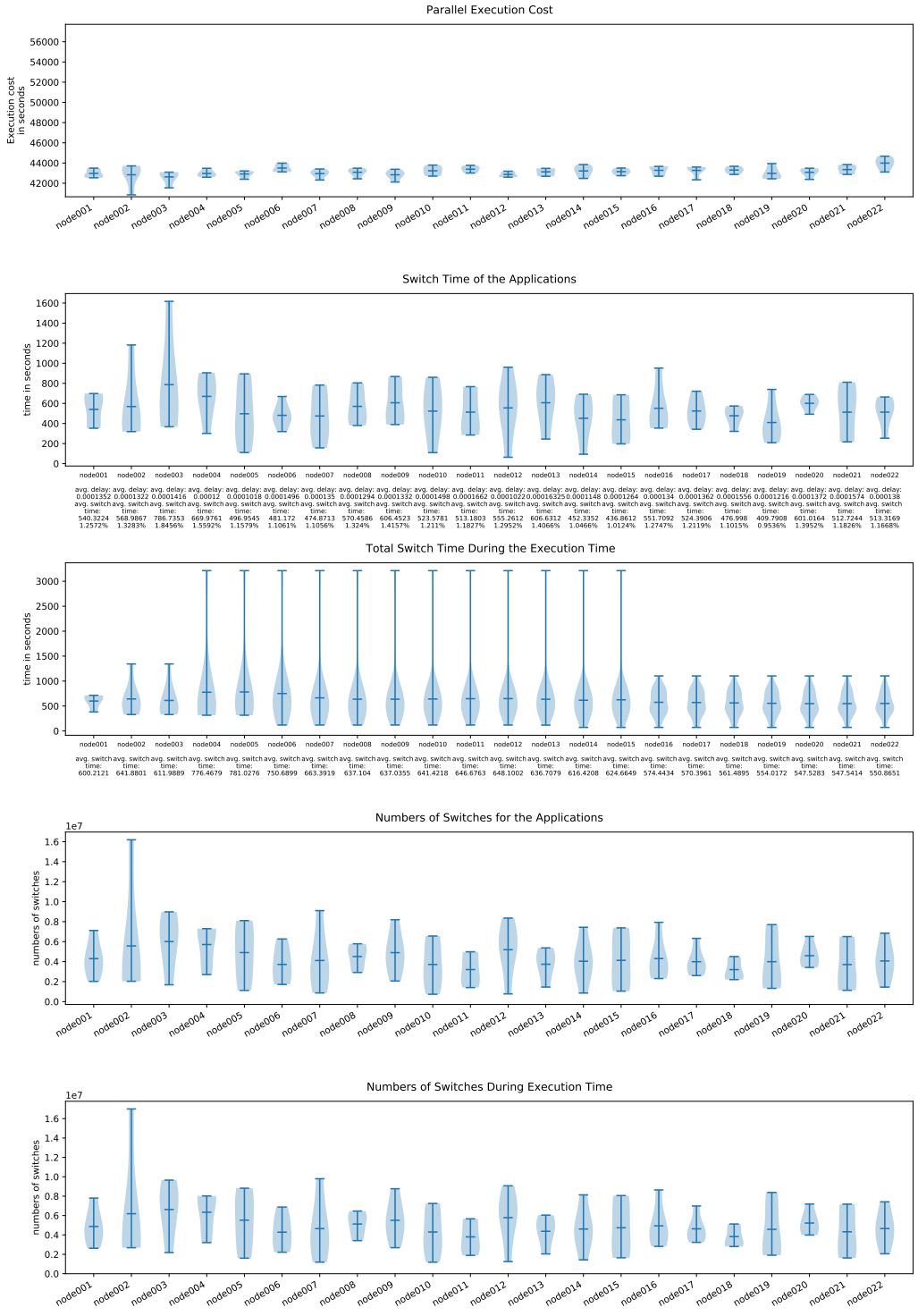

Figure 4.28: The measurements for the benchmark SPH_EXA, scheduler static, 20 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. We observe again a pattern of similar outliers in the total switch time during the execution on node four to 15.

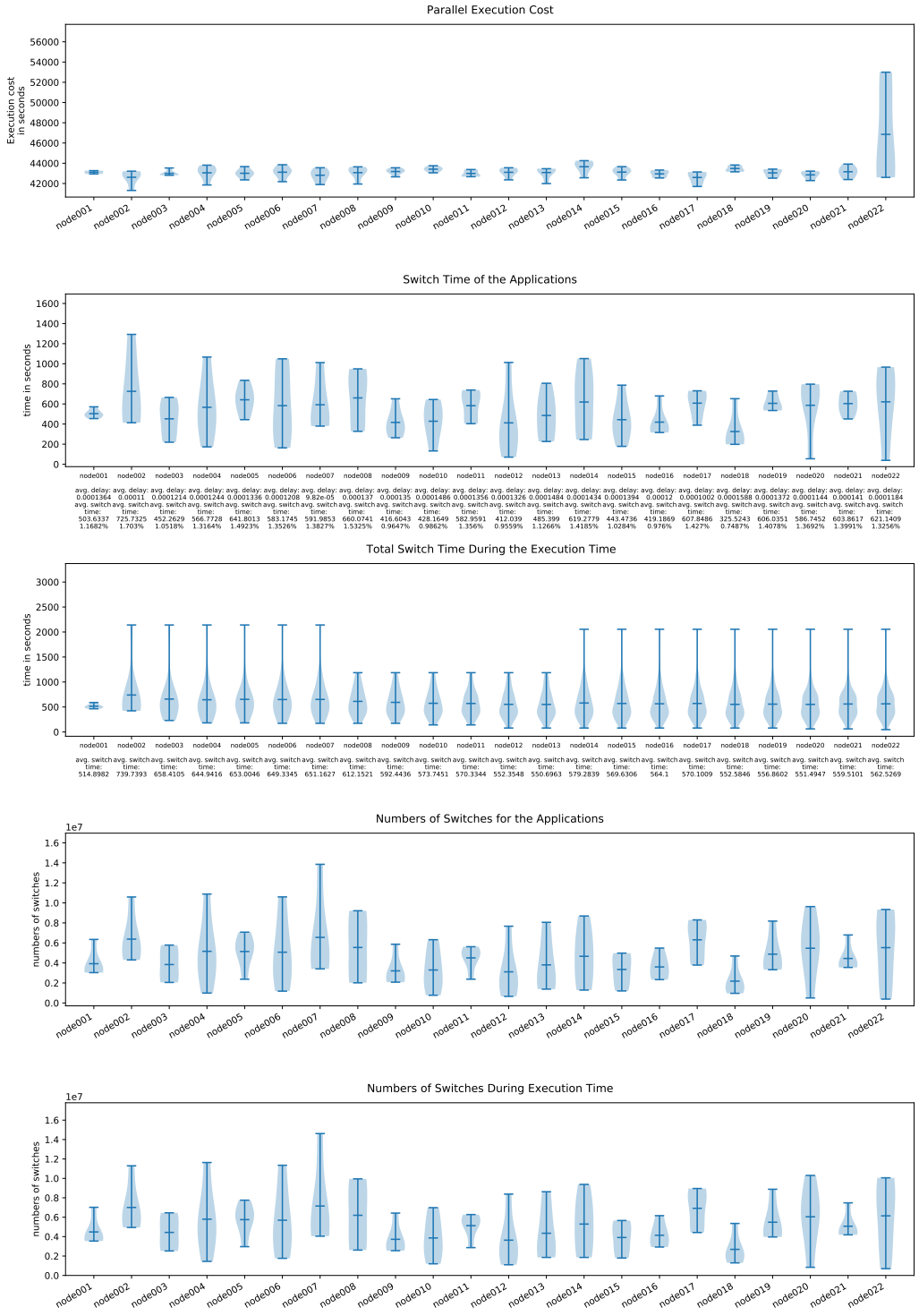

Figure 4.29: The measurements for the benchmark SPH_EXA, scheduler guided, 20 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Node 22 has several measurements with a much higher execution time. To check that these are not some random outliers we repeated these experiments on node 22 with the same results. Some executions took a similar time as on other nodes, but some took about 20% longer. The rest of the results look similar to other experiments.

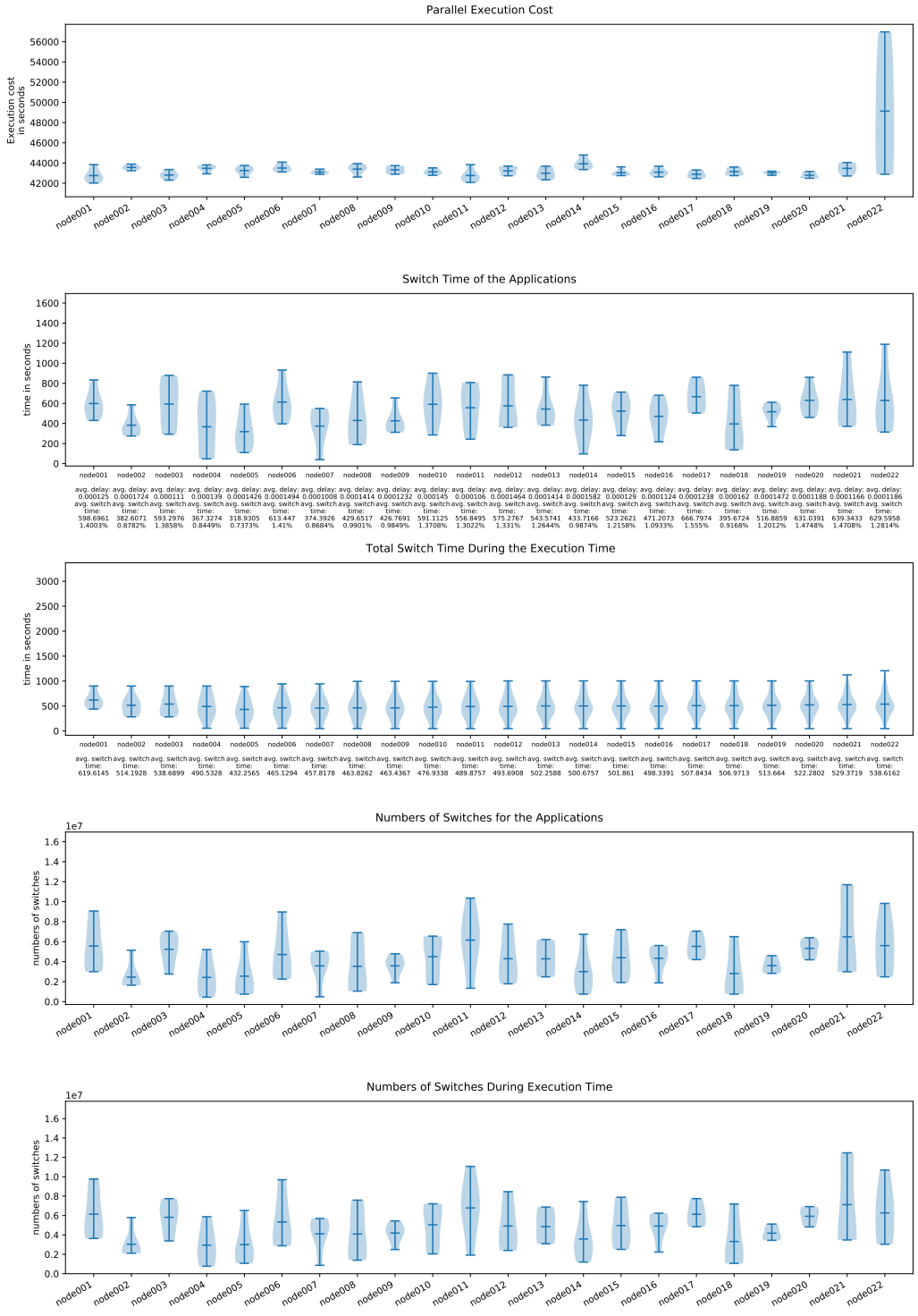

Figure 4.30: The measurements for the benchmark SPH_EXA, scheduler dynamic,24, 20 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. See the comment on Node 22 in the previous figure 4.29. There are no outliers in the total switch time during the execution.

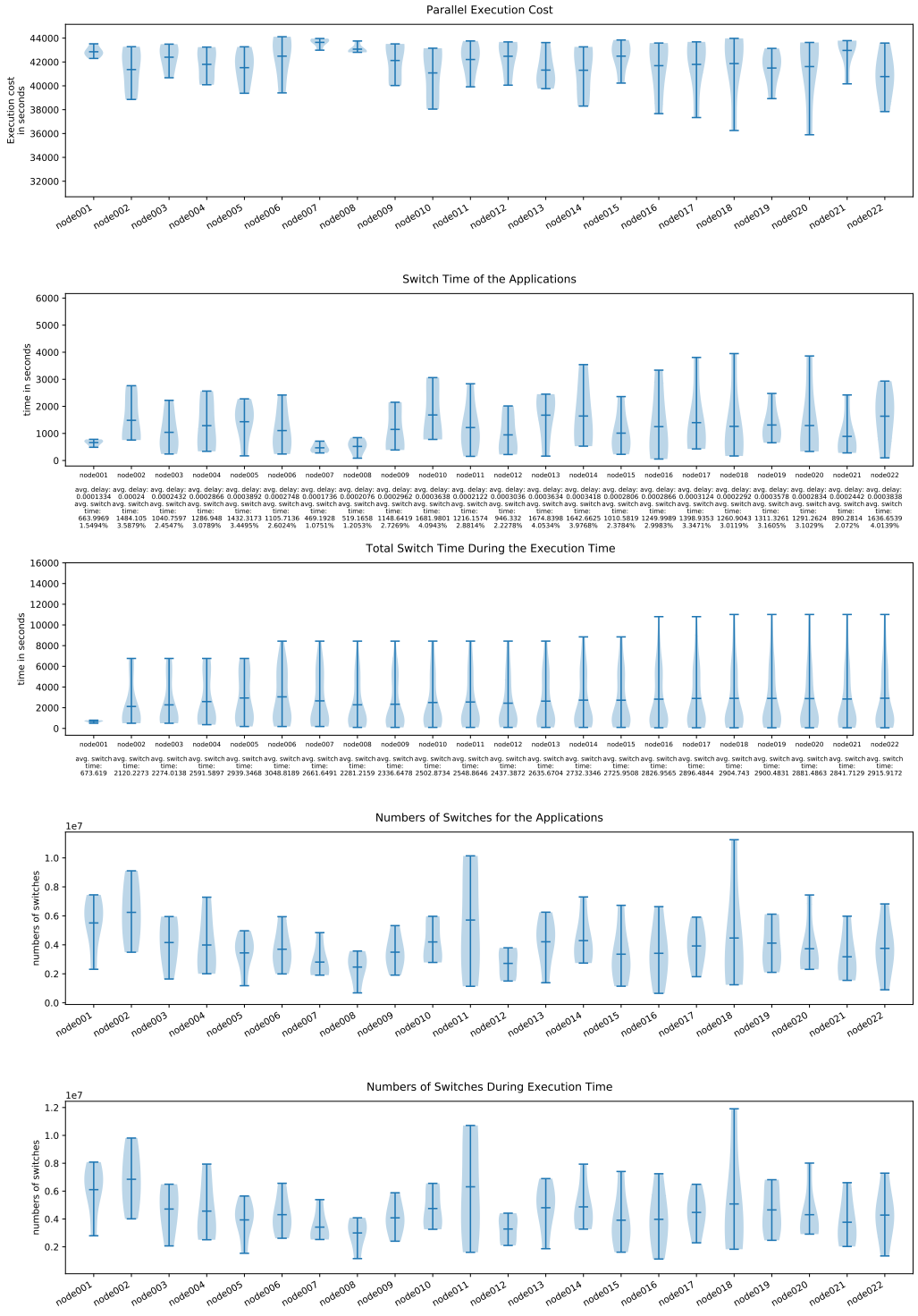

Figure 4.31: The measurements for the benchmark SPH_EXA, scheduler static, 16 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Remarkable is the lack of deviation in parallel execution cost and switch time of the application on nodes one, seven and eight.

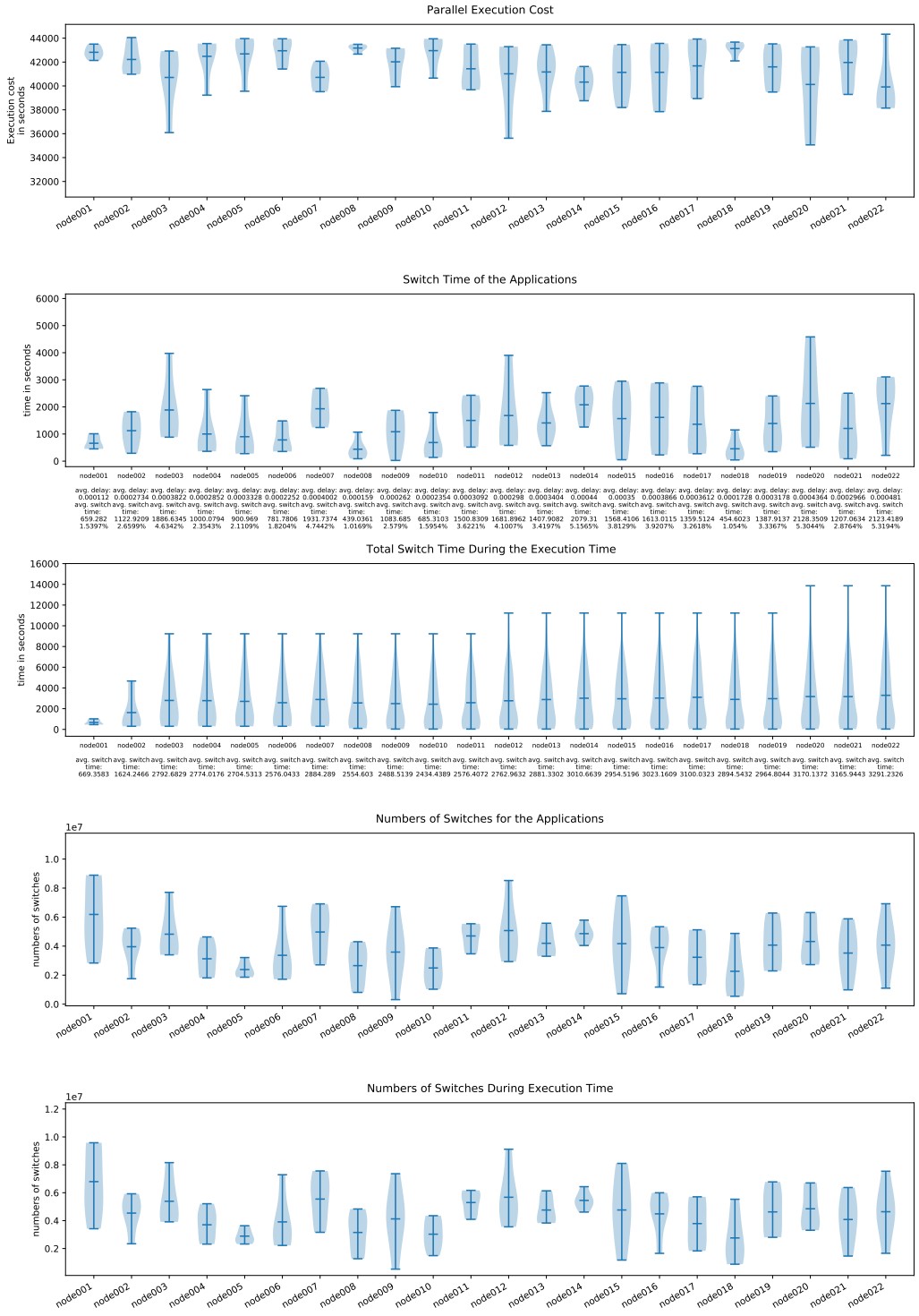

Figure 4.32: The measurements for the benchmark SPH_EXA, scheduler guided, 16 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Node 1 has a much less average switching time than the other nodes.

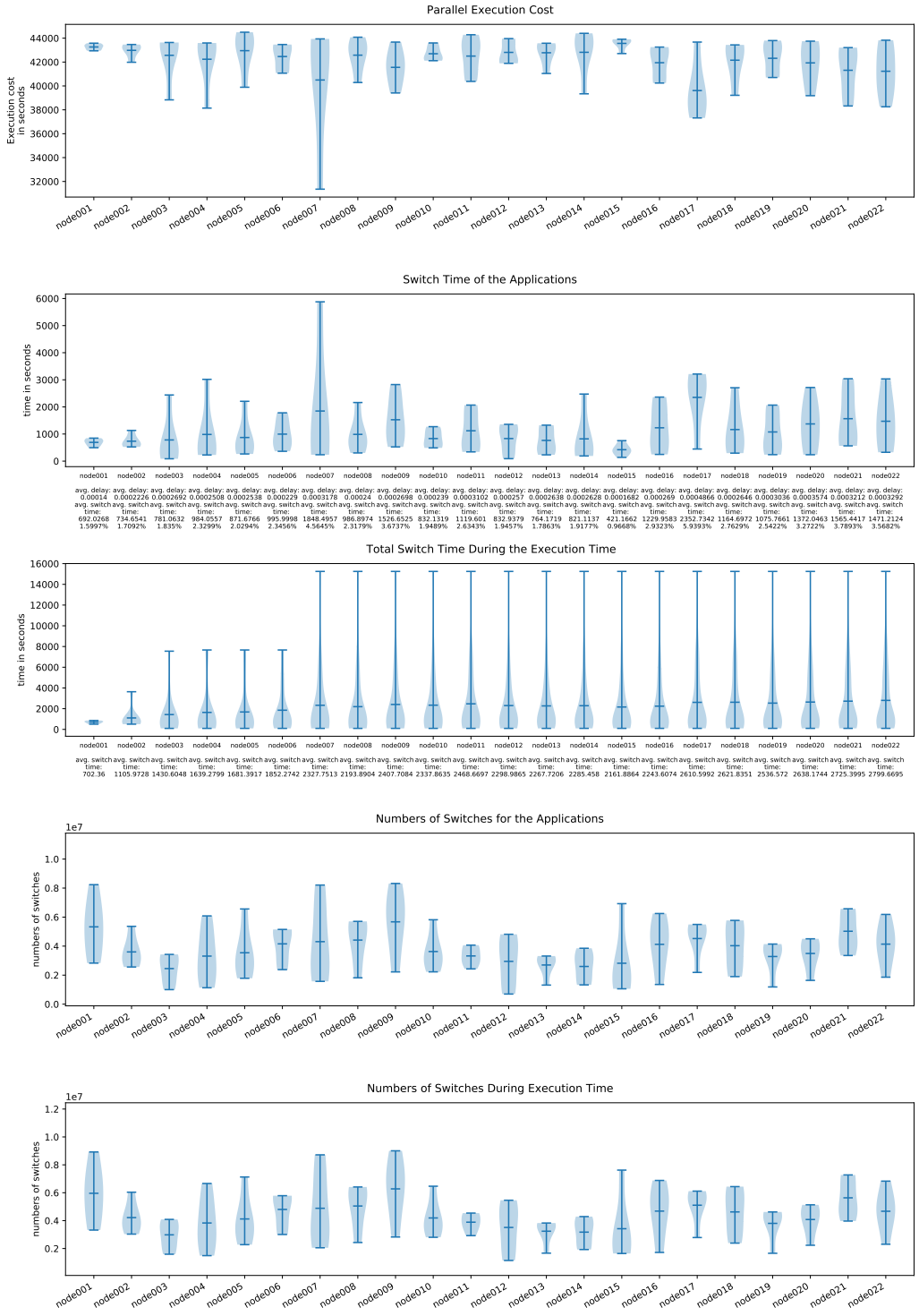

Figure 4.33: The measurements for the benchmark SPH_EXA, scheduler dynamic,24, 16 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Node seven has an extreme outlier in the parallel execution cost and the switch time of the application. Perf reports that the execution took 25% less time than all other executions. There might be an error in that execution.

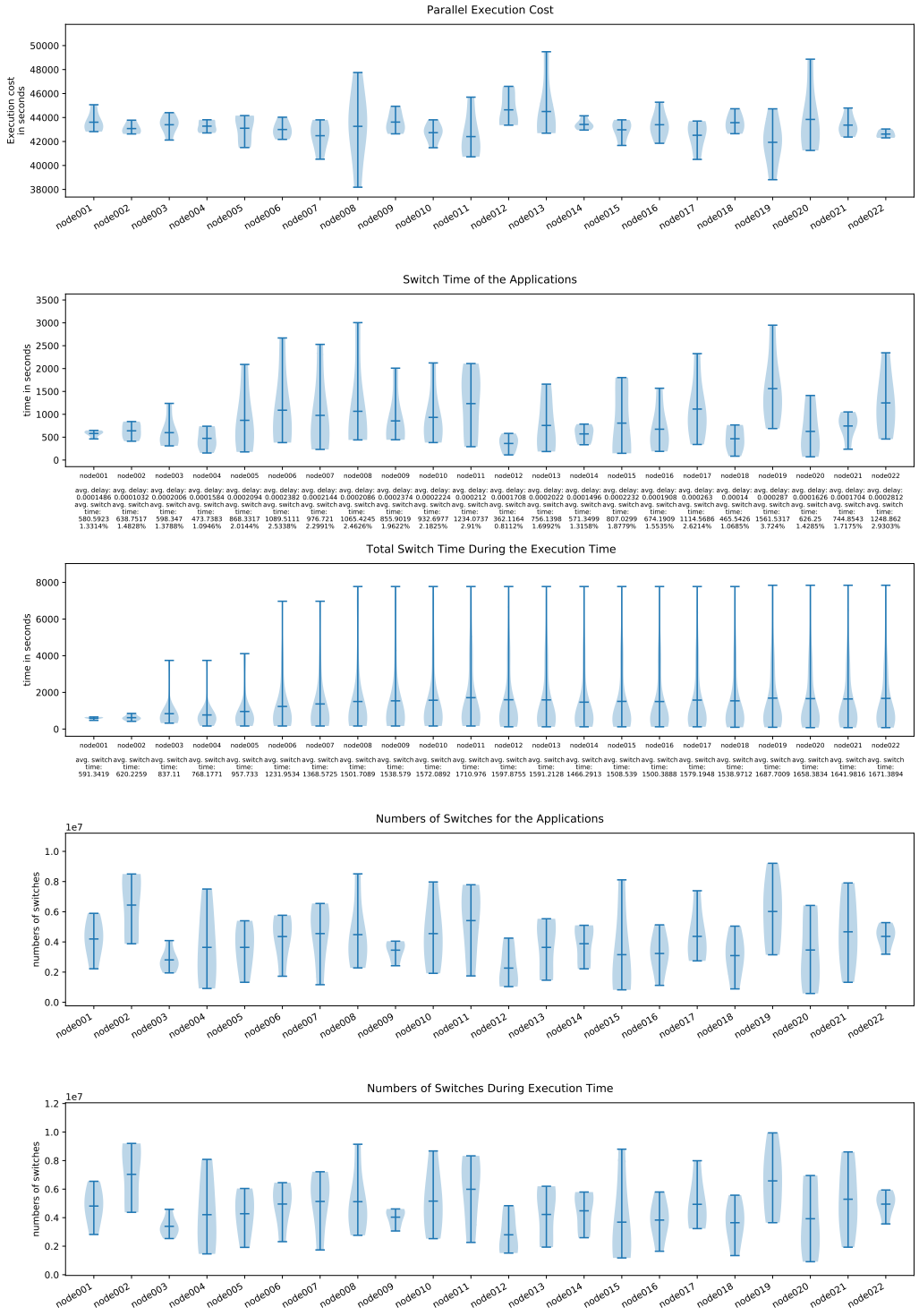

Figure 4.34: The measurements for the benchmark SPH_EXA, scheduler static, 10 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Again the total switching time of node one and two are much lower than the rest.

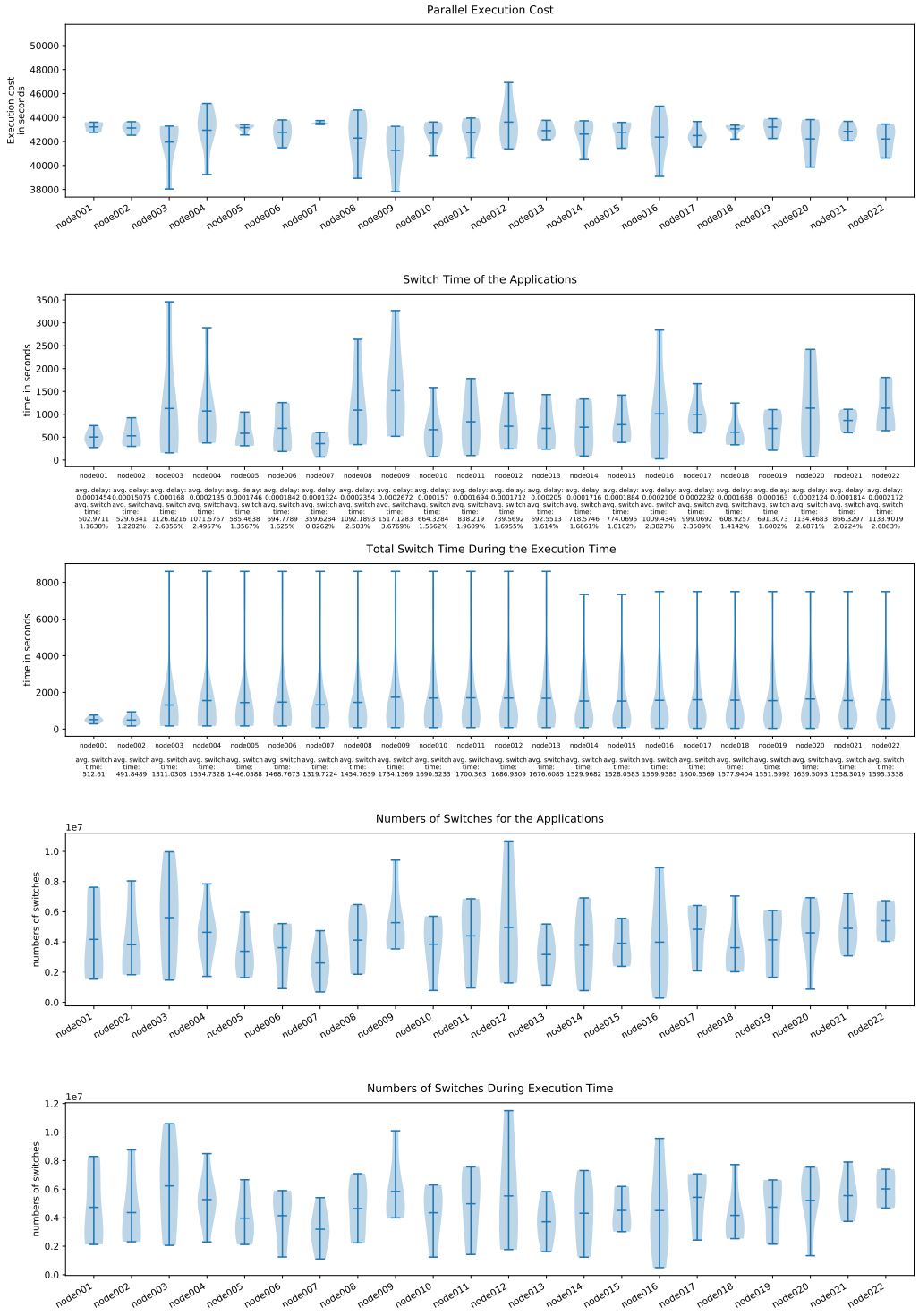

Figure 4.35: The measurements for the benchmark SPH_EXA, scheduler guided, 10 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Several nodes have very little deviation in the parallel execution cost. And node 1 and 2 have again a shorter total switch time during the execution.

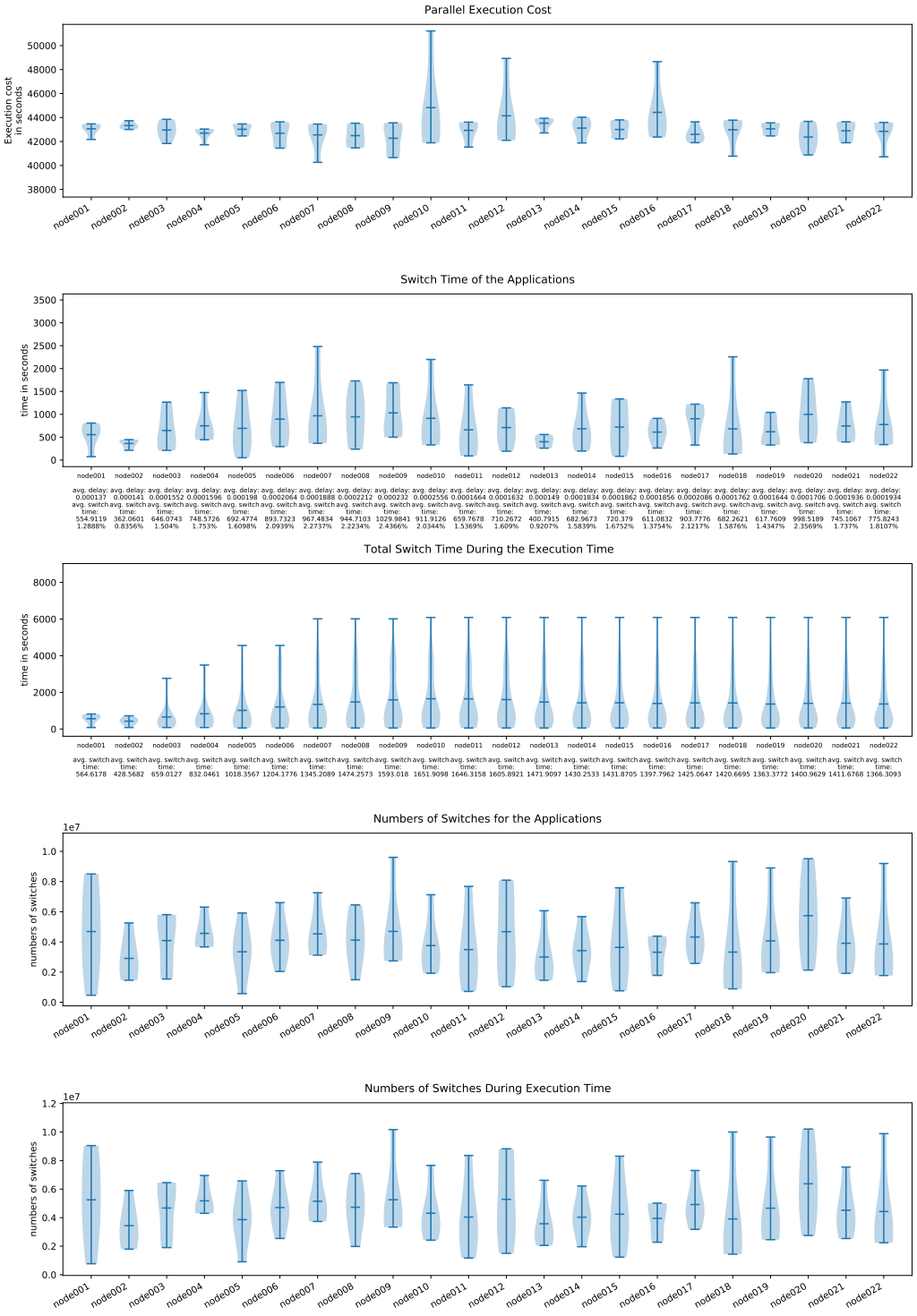

Figure 4.36: The measurements for the benchmark SPH_EXA, scheduler dynamic,24, 10 pinned threads, 5 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Nodes 10, 12 and 16 have outliers in the parallel execution cost.

4.2.4 Measurements of Mandelbrot

In this section, we have the result of the last application. There are 20 samples for each experiment. In figures 4.37-4.39 are the plots for 20 free threads. In 4.40-4.42 the plots for 20 pinned threads. We see the results of 16 threads in 4.43-4.45 and in 4.46-4.48 the results of ten threads.

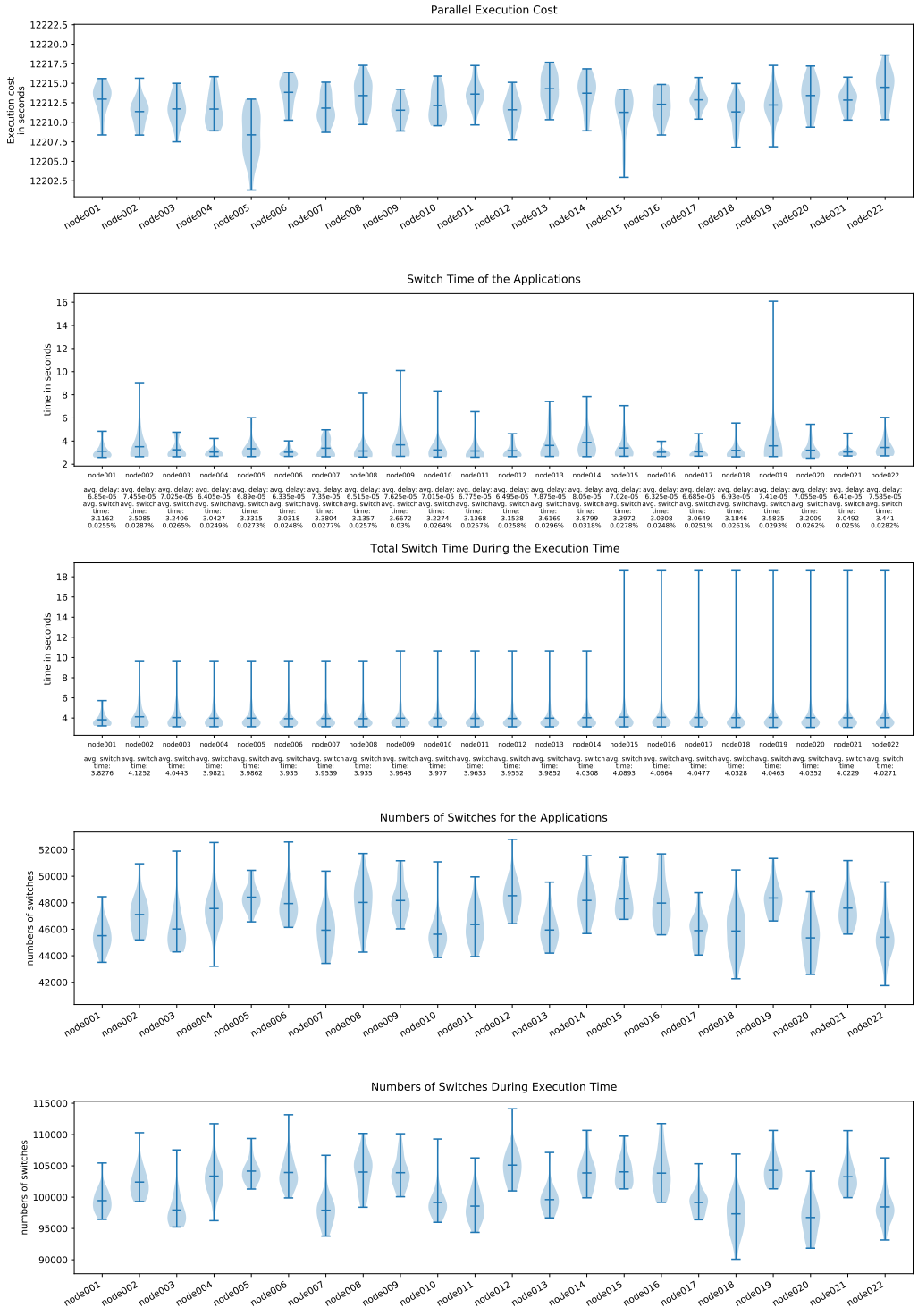

Figure 4.37: The measurements for the benchmark Mandelbrot, scheduler static, 20 free threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Node 5 has a much less average execution cost then the other nodes.

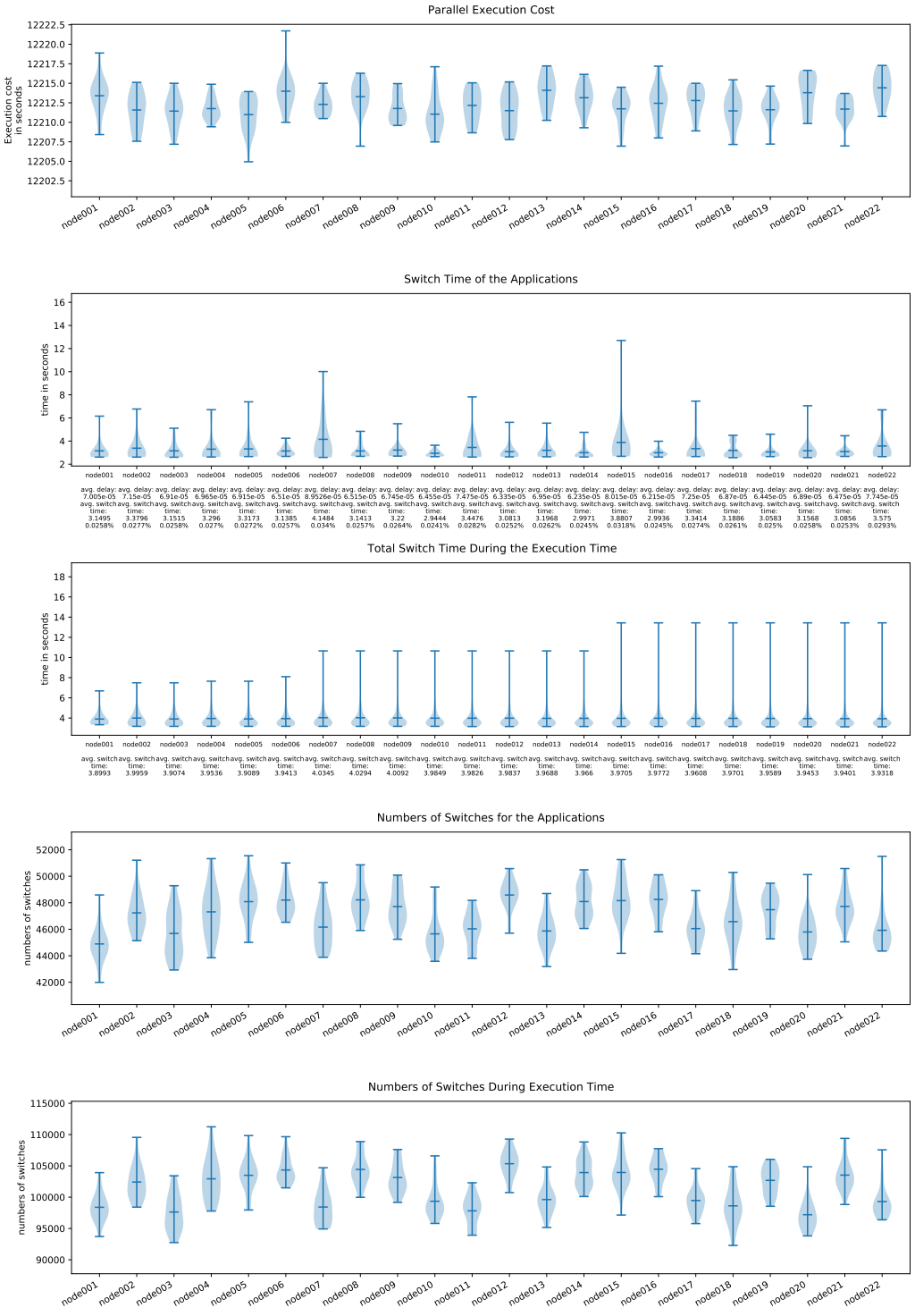

Figure 4.38: The measurements for the benchmark Mandelbrot, scheduler guided, 20 free threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

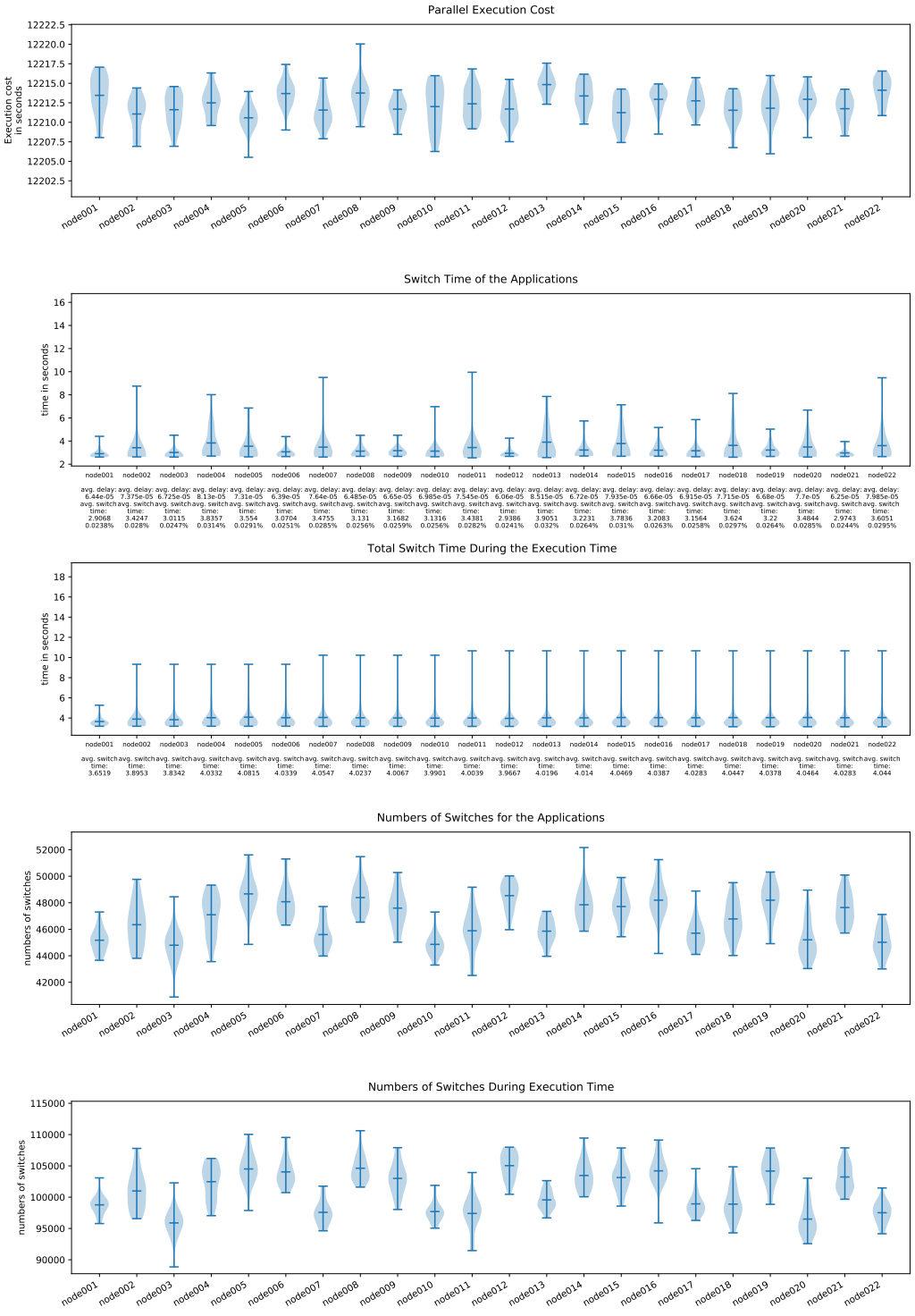

Figure 4.39: The measurements for the benchmark Mandelbrot, scheduler dynamic,24, 20 free threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

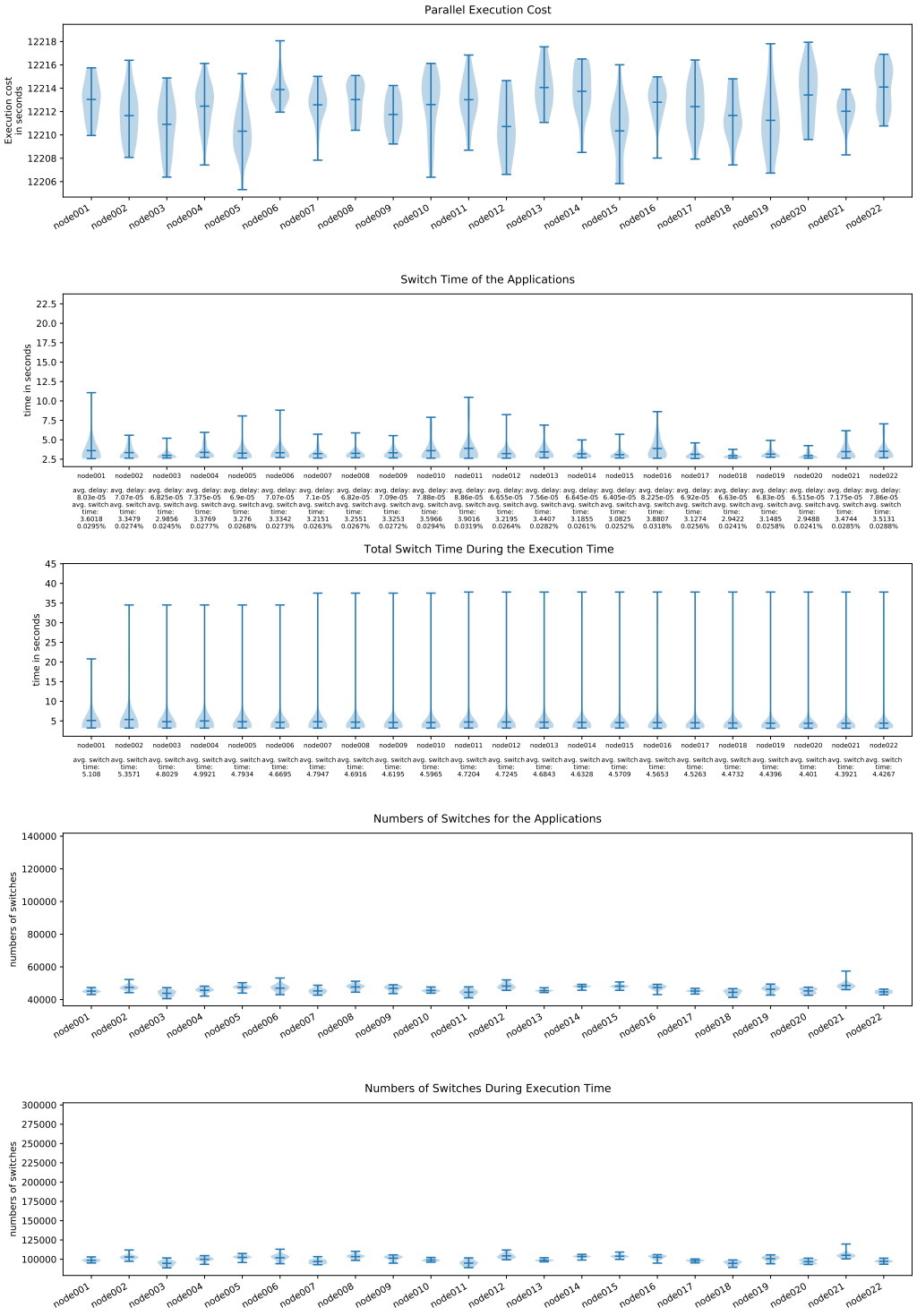

Figure 4.40: The measurements for the benchmark Mandelbrot, scheduler static, 20 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

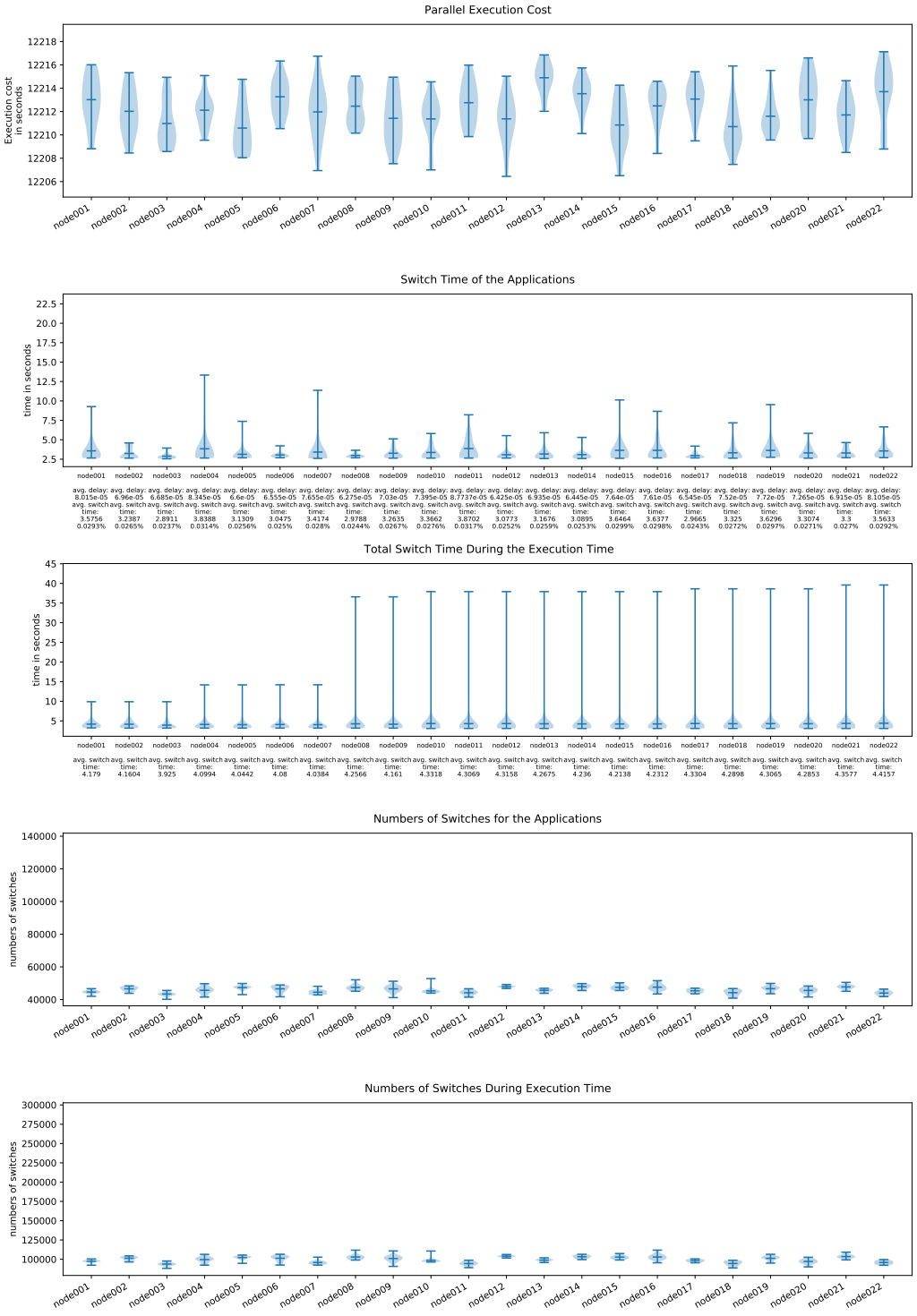

Figure 4.41: The measurements for the benchmark Mandelbrot, scheduler guided, 20 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Nodes 1, 4, 7, 11, 15, 16 and 19 and have some extreme outliers in the switch time of the application.

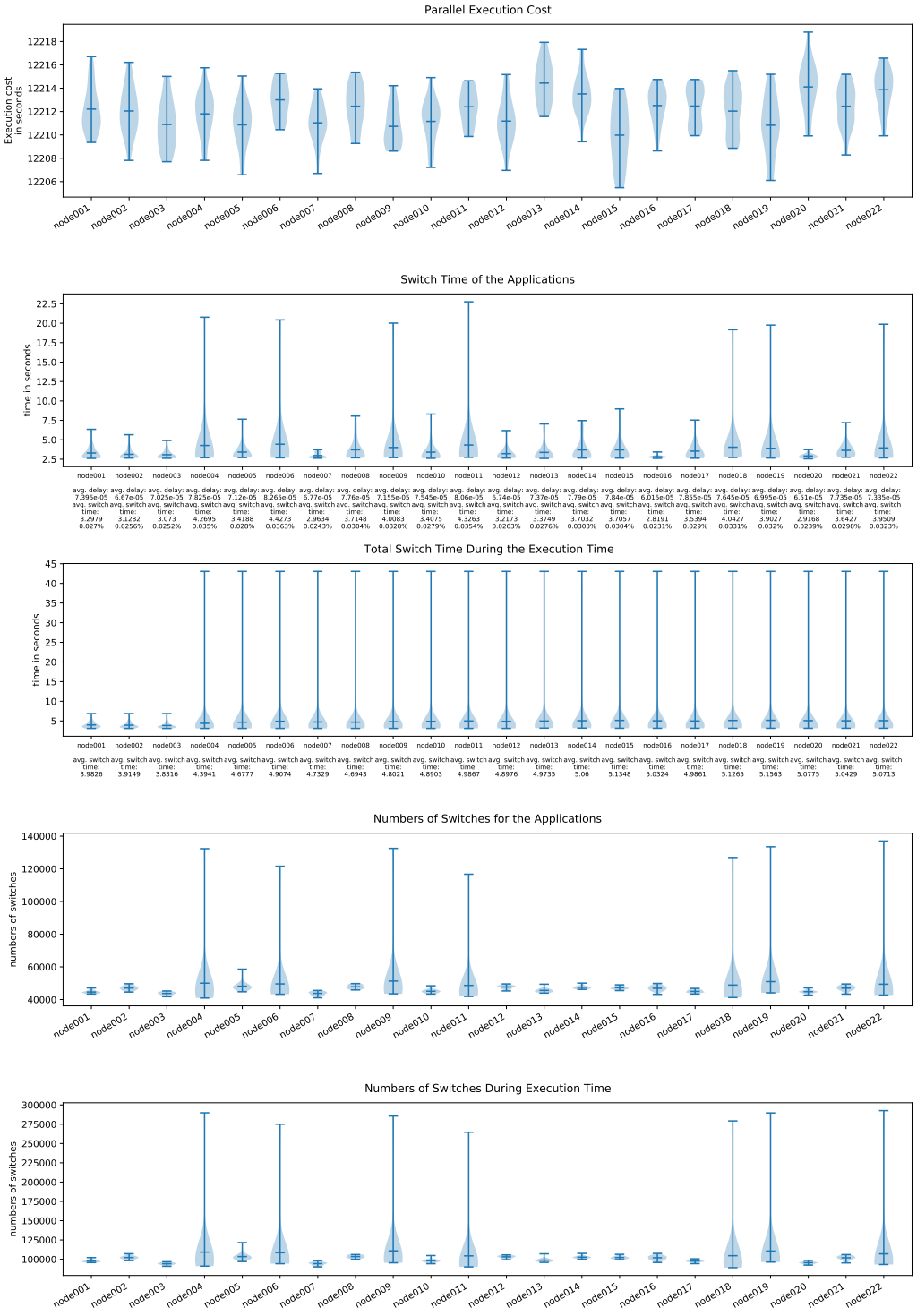

Figure 4.42: The measurements for the benchmark Mandelbrot, scheduler dynamic,24, 20 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. Nodes 4, 6, 9, 11, 18, 19 and 22 had outliers in the numbers of switches and therefore in the total number of switches. Interestingly they also have outliers in the switch time of the application.

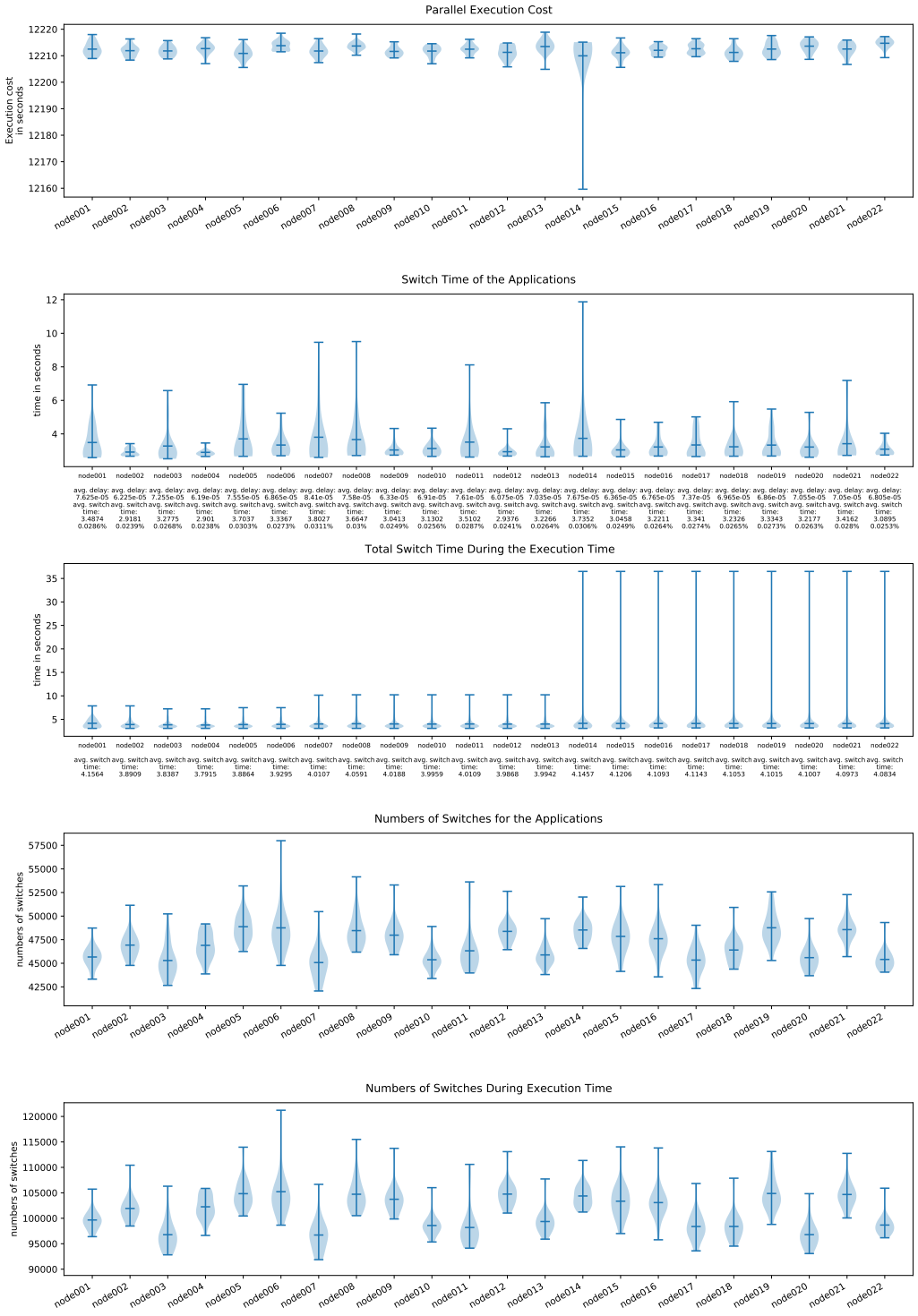

Figure 4.43: The measurements for the benchmark Mandelbrot, scheduler static, 16 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches. One outlier on node 14 has a shorter parallel execution cost.

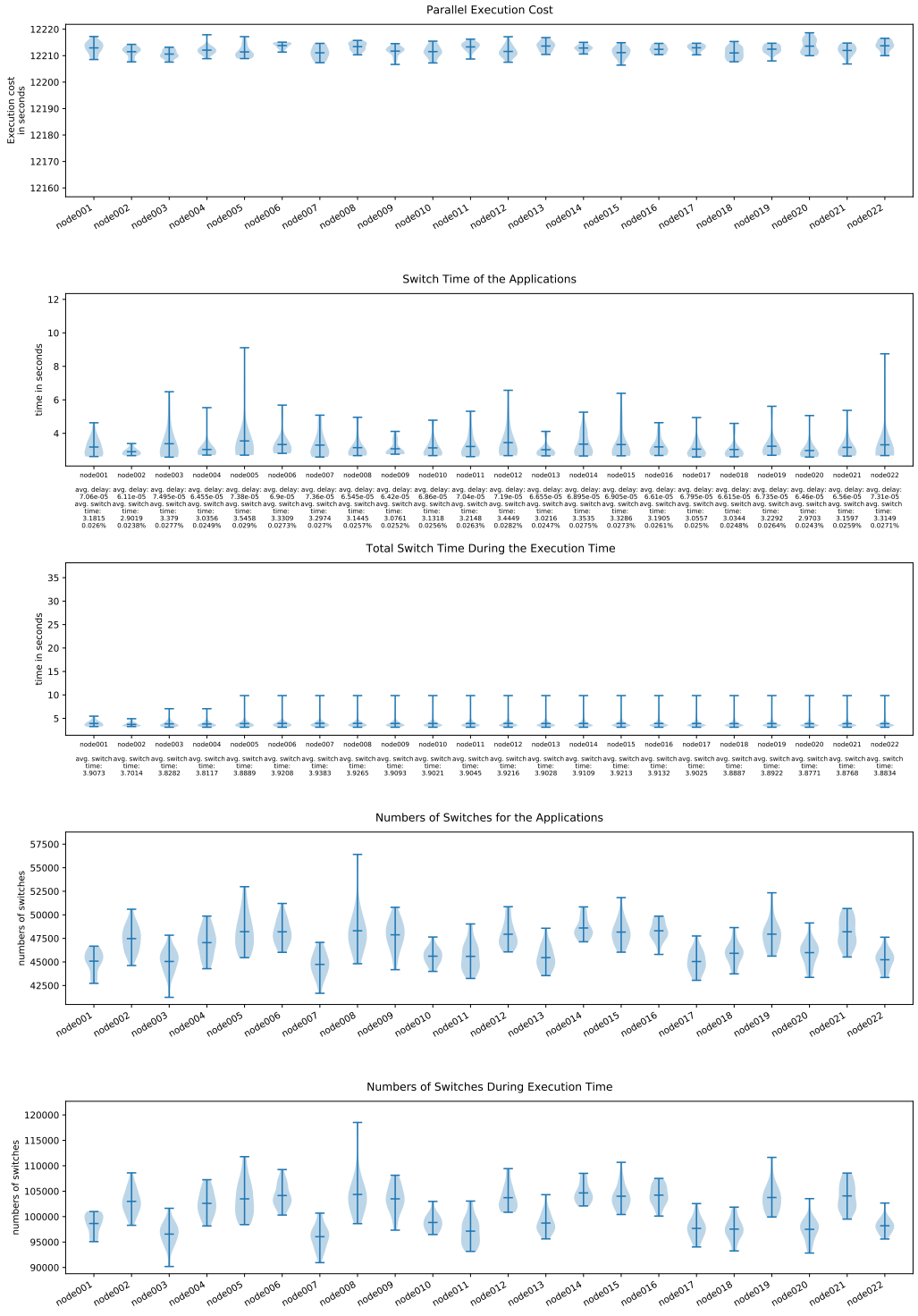

Figure 4.44: The measurements for the benchmark Mandelbrot, scheduler guided, 16 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

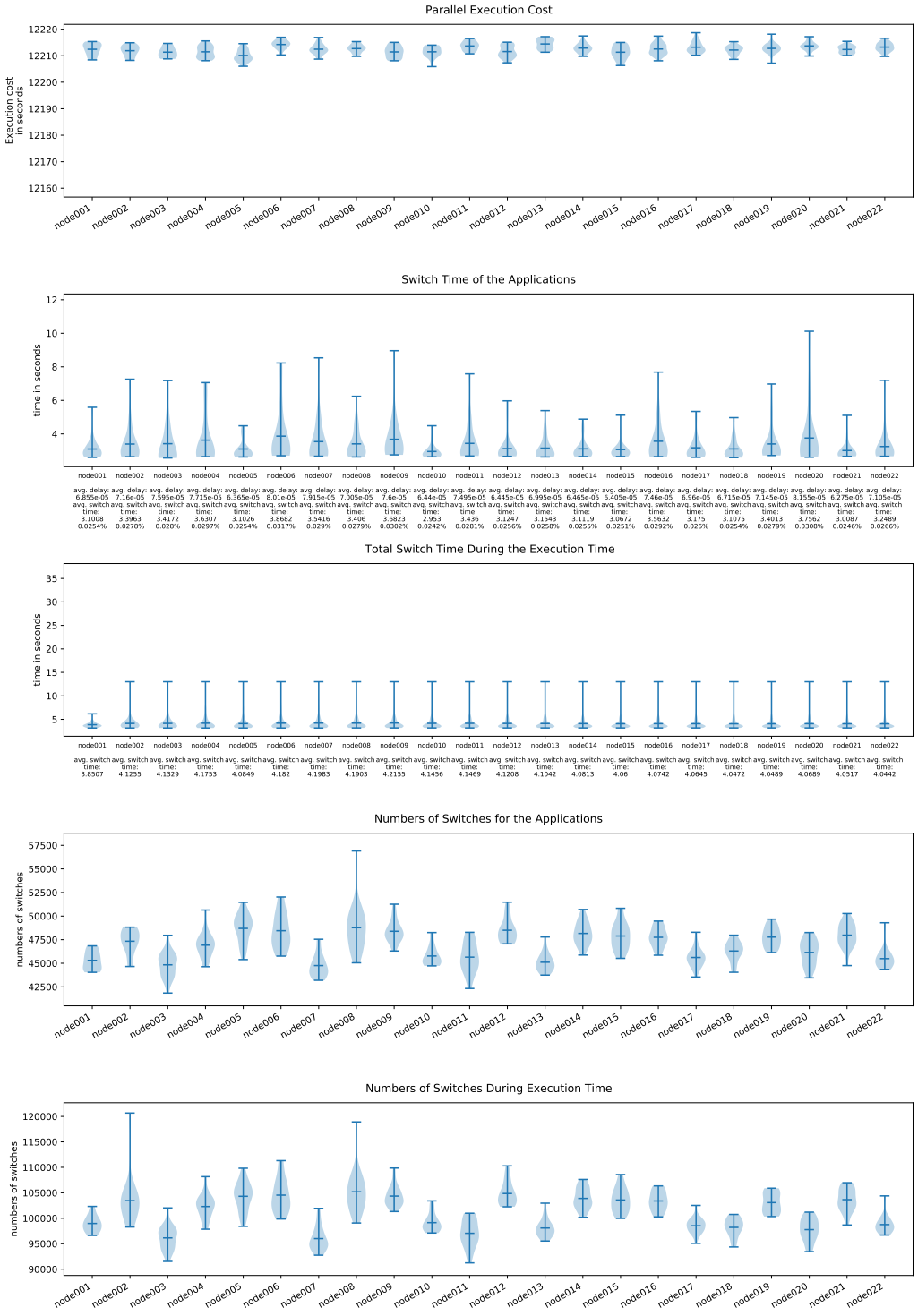

Figure 4.45: The measurements for the benchmark Mandelbrot, scheduler dynamic,24, 16 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

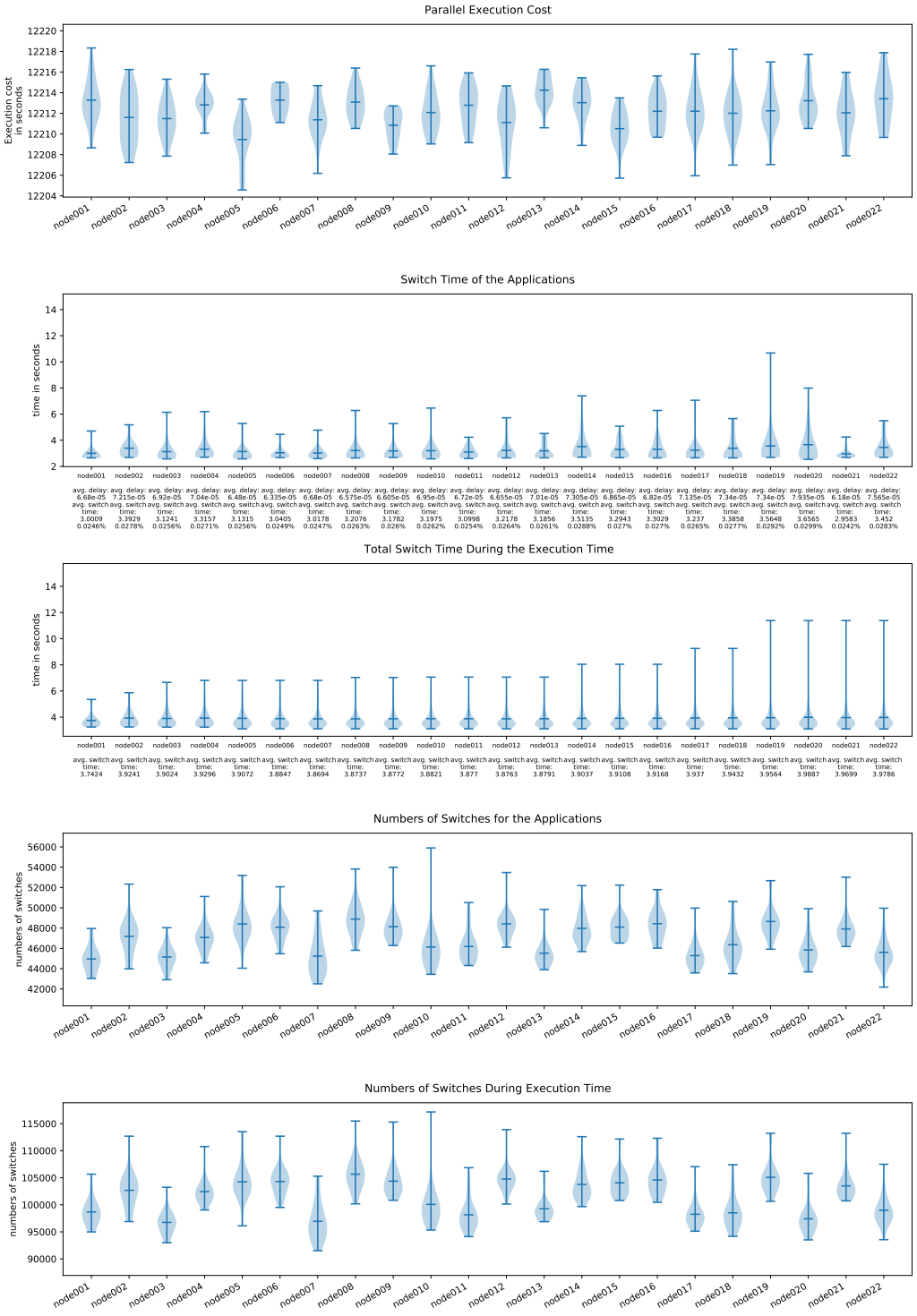

Figure 4.46: The measurements for the benchmark Mandelbrot, scheduler static, 10 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

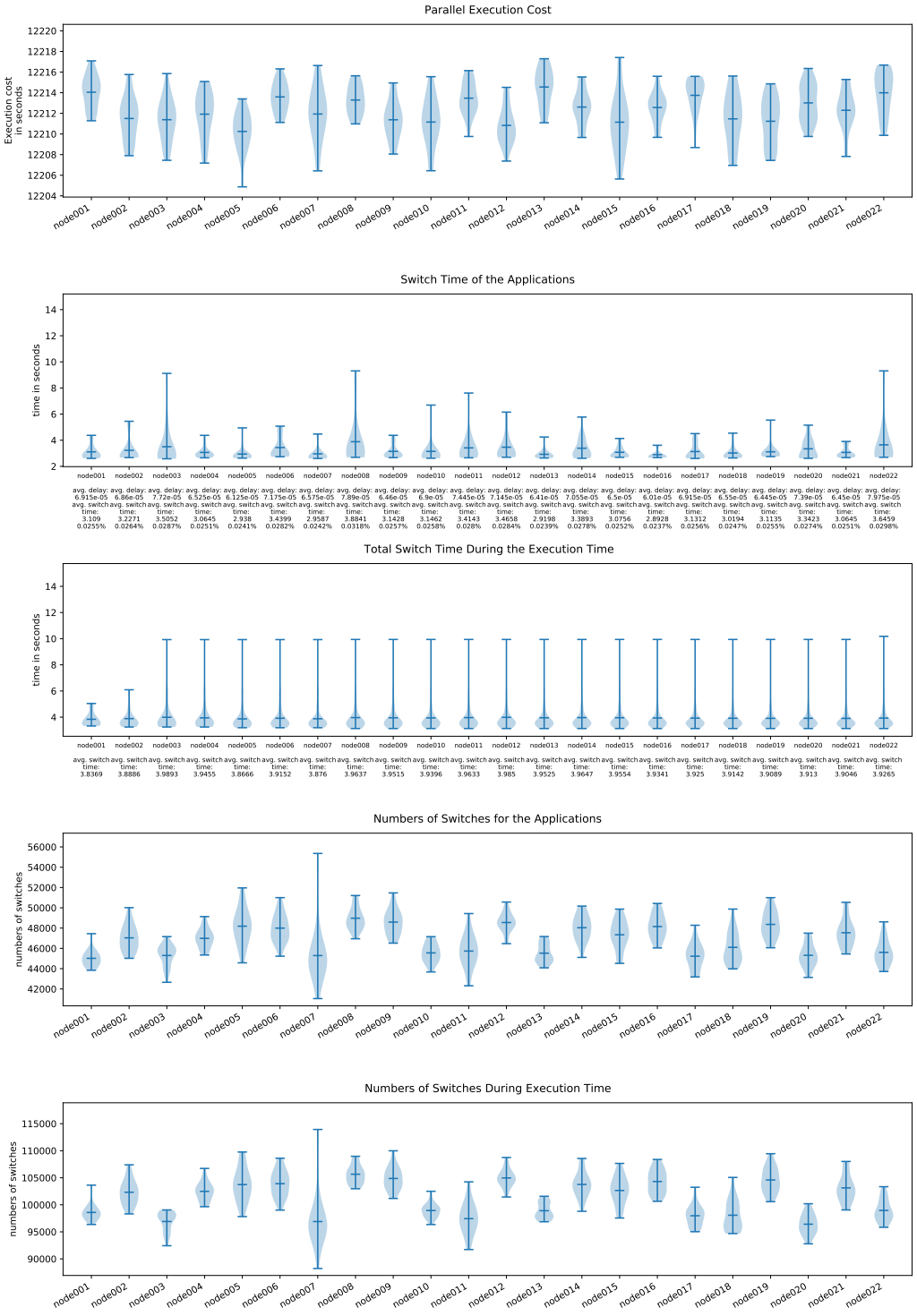

Figure 4.47: The measurements for the benchmark Mandelbrot, scheduler guided, 10 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

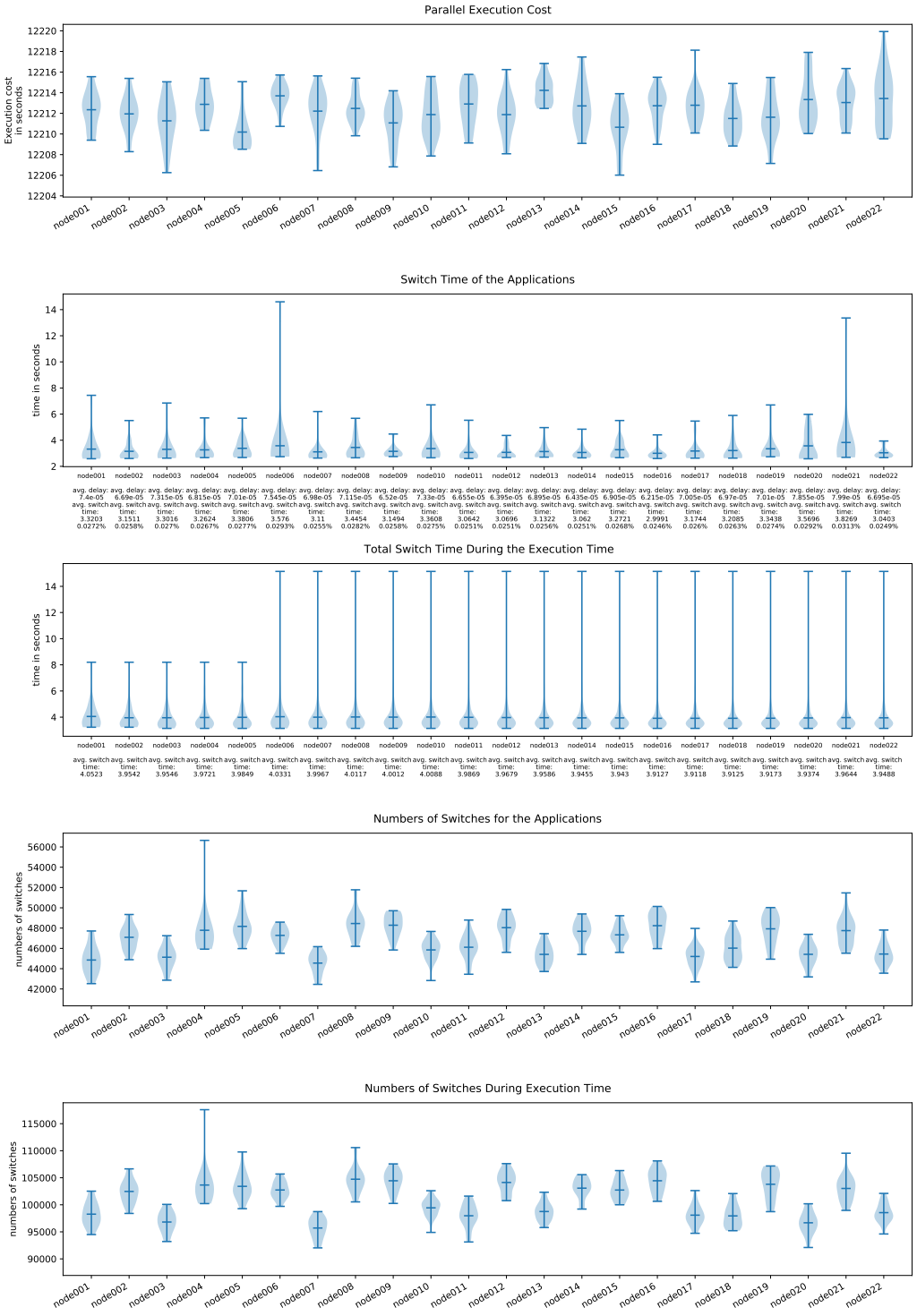

Figure 4.48: The measurements for the benchmark Mandelbrot, scheduler dynamic,24, 10 pinned threads, 20 samples. 1. graph: parallel execution cost of the application, 2. graph: overhead for switching the application, 3. graph: total switch time, 4. graph: numbers of switches for the application, 5. graph: total number of switches.

5 Discussion

In this report, we show our measurements of the Linux scheduler. We have the plots for our measurements in chapter 4. We observe that the parallel execution cost does not change much with different thread configurations. This is what we expected because the work does not change. The work is only done by a different number of threads. The parallel execution cost is not the same as the execution time. The execution time is much longer if we use fewer threads. Therefore we focus the discussion more on the time spend on switching and the numbers of switches.

First, we have a look at the percentage of the parallel execution cost that is spent on switching. Later we analyse the number of switches. Then we show some results we did not expect and end with the discussion about the limitations of the experiments.

5.1 Scheduling Overhead of the Application

The percentage of the parallel execution cost that is spent on switching varies between the application. LavaMD spends on average 0.0008% of the parallel execution cost on switching. For Hotspot3D this value is 1.7963%, for SPH_EXA it is 2.2189% and Mandelbrot spends 0.0270% of its time switching. These values are the averages that are printed on the x-axis of the second graphs in the figures in chapter 4.

Table 5.1: Percentage of the Parallel Execution Cost that is spent on Switching for each Application

LavaMD	Hotspot3D	SPH_EXA	Mandelbrot
0.0008%	1.7963%	2.2189%	0.0270%

The average percent spent switching the applications for the different scheduling techniques is static 0.9796&, guided 0.9889% and dynamic,24 0.9425%.

Table 5.2: Percentage of the Parallel Execution Cost that is spent on Switching for the different Scheduling Techniques

static	guided	dynamic,24
0.9796%	0.9889%	0.9425%

The averages of the percentage spent on switching the application for the different thread configurations are 1.1745% for 20 free threads, 0.7733% for 20 pinned threads, 1.1828% for 16 pinned threads and 0.9125% for ten threads (hotspot3D only eight threads).

Table 5.3: Percentage of the Parallel Execution Cost that is spent on Switching for the different Scheduling Techniques

20 Free Threads	20 Pinned Threads	16 Pinned Threads	10 Pinned Threads
1.1745%	0.7733%	1.1828%	0.9125%

So the most important factor for how much time is spent on switching is the application. LavaMD has little load imbalance, so it makes sense that this application has very little scheduling overhead. The difference between SPH_EXA and Mandelbrot is surprising because Mandelbrot has a higher load imbalance than SPH_EXA. We do not know why SPH_EXA has such high overhead. Hotspot3D is special because it has only eight loop iterations. Therefore we can not directly compare it with other applications.

The difference between the thread configuration is surprising. We expected that pinning the threads would result in fewer switch overhead. This is supported by our experiments. 20 free threads have a higher percentage spent on switching, than 20 pinned threads. But 16 and ten pinned threads spent more time switching than 20 pinned threads. We do not know the reason for this behavior. Since we thought that leaving CPUs idle would lead to fewer interrupts because the OS can run on the other processors.

The three scheduling techniques do not differ much regarding the scheduling overhead for the application. Guided has the highest overhead. This is probably the case because guided distributes the work at runtime, but we do not know why the difference between guided and dynamic is so big, compared to static.

5.2 Numbers of Switches

In this section, we have a look at the numbers of switches. In chapter 4 we have shown them in the 4. and 5. graph of each figure. The number of switches for the application is the value that perf recorded. The number of switches during the execution is the total number of switches. This includes the switches for the application and every other program that executed alongside the application.

Table 5.4: Average Numbers of Switches for each Application

	LavaMD	Hotspot3D	SPH_EXA	Mandelbrot
average switches for the applications	1'410	4'269	4'092'096	46'800
average switches during the execution time	12'227	5'641	4'672'929	101'079

The average number of switches varies between the different applications. The main reason for this is that the execution time is not the same. With our parameters, Hotspot3D executes for about 10 seconds and SPH_EXA more than 15 minutes. LavaMD and Mandelbrot are between those two extremes. It is obvious that an application that executes longer experiences more switches.

Table 5.5: Average Numbers of Switches for each Scheduling Technique

	static	guided	dynamic,24
average switches for the applications	1'379'687	1'410'949	1'353'939
average switches during the execution time	1'596'399	1'625'328	1'572'149

We wanted to observe which influence the different OpenMP scheduling techniques have on the parameters we measure. As in the last section about the overhead, guided is again the worst scheduling technique. It has more switches than static or dynamic. The difference between static and dynamic is not so big. The number of switches shows the same ranking as the scheduling overhead. Dynamic has the fewest switches and shortest overhead. Static is not as good and guided shows the worst results.

Table 5.6: Average Numbers of Switches for each Thread Configuration

	20 Free Threads	20 Pinned Threads	16 Pinned Threads	10 Pinned Threads
average switches for the applications	1'014'028	1'110'349	983'664	1'036'533
average switches during the execution time	1'176'007	1'274'131	1'142'666	1'199'072

These results are interesting. Although 20 and 10 pinned threads showed less overhead in the last section perf reports more switches with this thread configurations. 20 free threads and 16 pinned threads have fewer switches while having a higher switching overhead. This means that either 20 and 10 pinned threads have more switches which are on average faster or 20 free and 16 pinned threads have fewer switches which takes more time on average. We do not know the reason for this behavior.

In our hypothesis (see section 3.1) we stated that we want to find out if we can observe a difference in the scheduling behavior, if we do not use all CPUs. Regarding the numbers of switches, we do not see more or fewer switches for fewer used CPUs.

5.3 Other Observations

In this section, we state some observations that are interesting but not very important. We have no explanation for them.

We see that many graphs of the total switching time during the execution have an interesting pattern. Most outliers have roughly the same value for a group of nodes. In most cases, the means are close to each other, which is normal. Surprisingly is that the outliers have similar values. An example for this is in figure 4.7

Also node 1 and 2 have often much lover values than the other nodes. Some examples for this are in figures 4.32-4.36.

Another observation is that the number of switches for the application are similar on a individual node for different schedulers. An example for this is in figures 4.22 and 4.24 or a bit less clear in figures 4.7 and 4.8. If you look at one graph the distribution on the different nodes seems random. But if you compare both graphs, the violin plots for each node are very similar to each other. We have them close to each other in figure 5.1 for easier comparison. The intuition would be that the number of switches does not depend on the node.

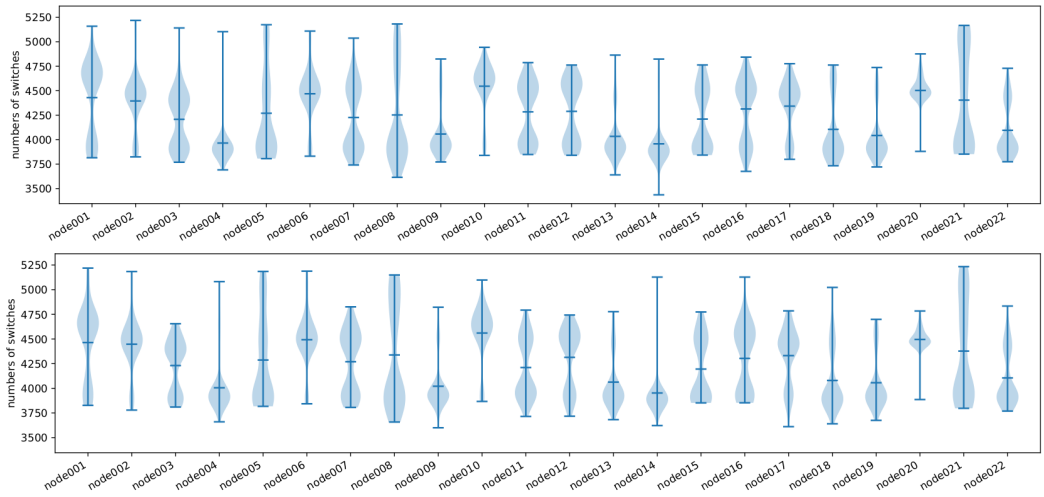

Figure 5.1: These are the numbers of switches for the applications of Hotspot3D with 8 threads.

In the results for hotsopt3D, the number of switches has sometimes two groups. Perhaps it depends on OS. Depending on where the OS executes. If the OS is scheduled on the CPU where the application is running then there are more switches. This would also explain the two groups in the parallel execution cost.

5.4 Limitations of our Measurements

Measurements with perf introduce an overhead to the execution of an application. It is hard to estimate how big this influence relay is. Perf stores every scheduling event and wakes up from time to time to write the recorded events in the perf.data file. At the end of the execution of an application, this file can become large in a short time.

We see that the number of switches varies highly for SPH_EXA. The reason for this is probably, that there are only five samples each, in contrast, LavaMD and Hotspot3D have 100 samples for each experiment. Also for Mandelbrot, where we have 20 samples, the number of switches varies widely. We did fewer experiments for SPH_EXA and Mandelbrot because this application takes much longer to execute.

6

Conclusion and Future Work

In this report, we showed our results of performance measurements of parallel OpenMP applications with perf. We have experimented with different applications, scheduling techniques and thread configurations.

Our results show that of the three scheduling techniques we used in our experiments, dynamic has the least overhead hand the fewest number of switches. Closely followed by static. The worst scheduling technique is guided. The thread configuration seems not to have a big impact on the scheduling behavior.

We also showed that the scheduler can have a big influence on some applications. One application, LavaMD, spends basically no time on switching SPH_EXA spend up to 2.2% of the parallel execution cost on switching.

6.1 Future Work

A topic for future work could be to experiment with different operating systems. These experiments were all done on the minHPC. All tests report from one operating system. It would be interesting to compare different operating systems with each other. Is there a big difference between Windows, macOS and different Linux distributions?

Bibliography

[1] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, (12.02.2021).

[2] Hakan Akkan, Michael Lang, and Lorie Liebrock. Understanding and isolating the noise in the linux kernel. *The International journal of high performance computing applications*, 27(2):136–146, 2013.

[3] Emiliano Betti, Marco Cesati, Roberto Gioiosa, and Francesco Piermaria. A global operating system for hpc clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[5] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11. IEEE, 2010.

[6] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi: 10.1109/99.660313.

[7] Urs Fässler and Andrzej Nowak. perf file format. Technical report, Technical report, CERN Openlab, 2011.

[8] Roberto Gioiosa, Sally A McKee, and Mateo Valero. Designing os for hpc applications: Scheduling. In *2010 IEEE International conference on cluster computing*, pages 78–87. IEEE, 2010.

[9] Brendan D. Gregg. perf examples. http://www.brendangregg.com/perf.html, (12.02.2021).

[10] Danilo Guerrera Aurelien Cavelan Michal Grabarczyk jg piccinali David Imbert Ruben Cabezon Darren Reed Lucio Mayer Ali Mohammed Florina Ciorba Tom Quinn. Github repository of the miniapp application SPH-EXA. https://github.com/unibas-dmi-hpc/SPH-EXA_mini-app, (12.02.2021). Commit #26.

[11] M Tim Jones. Inside the linux scheduler. *IBM Developer Works*, 2006.

[12] Robert Love. *Linux Kernel Development*. Pearson Education, 2010.

[13] Tang Peiyi and Yew Pen-Chung. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the International Conference on Parallel Processing*, pages 528–535, August 1986.

[14] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987. ISSN 0018-9340.

[15] Aman Singh, Anup Buchke, and Yann-Hang Lee. A study of performance monitoring unit, perf and perf_events subsystem, 2012.

[16] Konstantin Tabere and Dipl-Inf Michael Roitzsch. Migration sweetspots. 2013.

[17] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor

David Kuhn

Matriculation number — Matrikelnummer

16-057-960

Title of work — Titel der Arbeit

Impact of the Linux Operating System on the Performance of OpenMP Applications

Type of work — Typ der Arbeit

Master Project

Declaration — Erklärung

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 12.02.2021

Signature — Unterschrift