Olympiads in Informatics, 2011, Vol. 5, 92–102
© 2011 *Vilnius University*

Fairness of Time Constraints

Martin MAREŠ
Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
e-mail: mares@kam.mff.cuni.cz

Abstract. Many computer science contests use automatic evaluation of contestants' submissions by running them on test data within given time and space limits. It turns out that the design of contemporary hardware and operating systems, which usually focuses on maximizing throughput, makes consistent implementation of such resource constraints very cumbersome. We discuss possible methods and their properties with emphasis on precision and repeatability of results.

Key words: automatic grading, time limits, precision.

1. Introduction

Many programming contests in the world employ automatic grading of programs submitted by contestants. This is usually accomplished by running the submissions on batches of input data and testing correctness of the output. To distinguish between correct solutions of different efficiency, the tested program must finish within given resource limits. In most cases, the only limited resources are execution time and memory, but there are other possibilities, e.g., communication complexity in case of interactive programs.

For the competition to be fair, all submissions must be treated the same by the grader and the grading process must be repeatable. Contest rules often forbid non-deterministic programs for this reason, or at least they disclaim repeatability in such cases. For deterministic programs, memory consumption and communication complexity are well-defined quantities which can be measured exactly. However, several concerns were raised recently about the precision of time measurements.

In this paper, we investigate this issue. We have performed multiple experiments with timing real contenstants' submissions from the IOI 2009 on different machines. We will present a summary of our results and we will try to explain them. As far as we know, this is the first study of this kind based on such broad data.

2. Possible Sources of Inaccuracy

Programming contests are typically run on commodity PC-class hardware with an OS based on the Linux kernel. However, unlike their ancestors from the 1980's, contemporary PC's are fairly complex and the operating systems used on them doubly so. A variety of optimization techniques are used in both software and hardware with the common goal of improving performance of frequent code patterns at the expense of decreased predictability of instruction timing.

Possible sources of inaccuracy include:

- *Context switches.* Although Linux is usually considered to be a monolithic kernel, it defers many activities (like some kinds of I/O) to system processes. Therefore even if the contestant's program is the only user process running, the scheduler occasionally switches context to a system process and back. The kernel tries to keep track of execution time of each process separately, but the precision of is mostly limited to a single timer tick (on the order of milliseconds). Therefore short-lived context switches are seldom accounted for properly.
- *Cache hierarchy.* Since processors are much faster than the main memory, the processor contains cache memory which remembers recently accessed data in a hope to speed up access to frequently used values. There is typically a hierarchy of several caches of different sizes and speeds. Unfortunately, the caches are not fully associative, so their efficiency can vary with exact placement of data in the main memory. The placement depends on too many external factors. Furthermore, a context switch to a different process leads to flushing data from the cache.
- *Multiple cores.* Parallel computation is no longer limited to expensive servers. Desktop processors contain several processing cores, together forming a small multi-processor system on a chip. Some caches are local to a single core, some are shared by more (but not necessarily all) cores. The process scheduler in the OS assigns running processes to cores, trying to exploit parallism as much as possible. On the other hand, when a process is moved to another core, it loses a part of the cached data, so it slows down for a short while. The scheduler therefore tries to balance inter-core movements, but the success varies.
- *Non-uniform memory architecture (NUMA).* On some machines, the main memory is not uniform. There are multiple memory controllers, each of them handling a different part of memory. Usually, the machine consists of several NUMA nodes, each of them containing a set of cores and a set of memory controllers connected to each other. Memory accesses within a node are then faster than inter-node ones. Again, the placement of processes and their data on the nodes is rather unpredictable.
- *Power management.* When the system load is small, the OS tries to save electric energy by putting functional blocks of the hardware to various power saving modes. E.g., an idle CPU core can have its operating frequency lowered (so it runs slower), or it can be even turned off completely. Transitions between these states are not immediate and they can incur flushing data from caches. Fortunately, most of the power management functions can be turned off and we recommend to do so on grading machines.
- *System management mode.* Machine firmware (the BIOS) can act behind the scenes. Most importantly, the hardware can ask the processor to interrupt execution of the program (and even of the OS kernel), to switch to a so-called system management mode and run a firmware routine. Significant amounts of time can be

spent in the SMM and since the OS is not notified, all this time is accounted to the currently running process. This is a common problem on laptops, but effects caused by the SMM have been observed on desktop and server computers as well.

3. Experiments

3.1. *Test Setup*

For our experiments, we have chosen three different tasks from the IOI 2009 in Plovdiv, Bulgaria:

- *Raisins* – this is a usual batch task (the program reads an input file and produces and output file) with small inputs (the largest input file has $10\,\mathrm{kB}$), but requiring a lot of time and memory. The model solution runs in time $\mathcal{O}(n^5)$ and memory $\mathcal{O}(n^4)$.
- *Mecho* – another batch task, but with larger inputs (the largest test case reaches ca. $600\,\mathrm{kB}$) and smaller resource requirements. The model solution runs in time $\mathcal{O}(n \log n)$.
- *Regions* – this is an interactive task requiring implementation of a data structure. The program interacts with a judge program, it receives queries from the judge and it has to answer them on-line. However, the judge's queries are not adaptive, so we can also run the task as a batch task for comparison. The test data contain about 105 queries per test, the model solution runs in time $\mathcal{O}(n^{0.5})$ per query and in linear space.

For the full description of the tasks, see Tzanov *et al.* (2009).

For all three tasks, we have used real contestants' submissions from the IOI, written in Pascal, C and C++, and also real test data. In some experiments, we had to reduce the submissions and the tests to a (hopefully representative) subset in order to run them within reasonable time. Also, we have dropped several C++ programs which failed to compile with a recent compiler.

We have run the tests on the following machines:

- *Camellia* – AMD Athlon64 X2 6000+ with 2 cores running at $1\,\mathrm{GHz}$, equipped with $2\,\mathrm{GB}$ of RAM.
- *Turing* – Intel Core2 Quad with 4 cores running at $2\,\mathrm{GHz}$, $4\,\mathrm{GB}$ of RAM.
- *Corbu* – Intel Core i7 with 4 cores at $2.66\,\mathrm{GHz}$, $6\,\mathrm{GB}$ of RAM.
- *Arcikam* – AMD Opteron 2218, 2 processors with 2 cores each, running at $2.5\,\mathrm{GHz}$. Arcikam's $8\,\mathrm{GB}$ of RAM are split to 2 NUMA nodes, each containing one processor.

All machines run Debian Linux 5.0 (Lenny) with kernel 2.6.37. The programs were compiled by GCC 4.3.2 and FreePascal 2.4.2. Effects of different versions of the kernel and of the compilers were also investigated, but they were negligible, so we do not discuss them here in detail. Also, where the behavior of several machines or of several tasks is sufficiently similar, we show only a single example. For grading, we have used the Moe contest environment developed by us; see Mares (2009) for details. The only part of Moe's grader, which can affect timing in a significant way, is the sandbox. It monitors all system calls for security reasons, passing them to a supervisor process for inspection. For some of the tests, we turned the sandbox off to check how much slowdown it causes.

3.2. *Error Distribution*

First of all, we try to establish statistical behavior of time measurement errors. We pick a single submission and run it repeatedly on a single input file. Histograms of two such experiments on batch tasks can be seen on Figs. 1 and 2, other experiments always resembled one of these two types. Our interactive task behaves in a more complex way, which will be discussed separately in Section 3.5.

In the first case (Arcikam), we got a distribution close to Gaussian, the second case (Turing) is more peculiar: most values are concentrated near the mean, but there is a couple of exceptional values much farther. Table 1 shows the fractions of samples within k-times the standard deviation from the mean.

Another interesting question is how much the standard deviation depends on the mean (that is, whether the error is primarily additive or multiplicative). To answer this, we have run all submissions of a single task 5 times on a single input and plotted standard deviation against mean. A typical plot looks like Fig. 3.

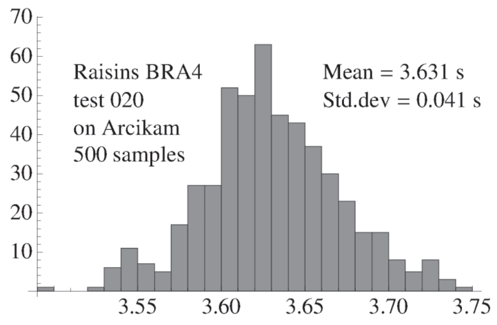

Fig. 1. Histogram of time [s] spent on a single test, type 1

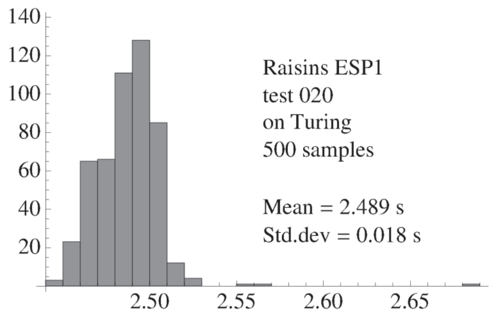

Fig. 2. Histogram of time [s] spent on a single test, type 2

Table 1
Probability of values within k-times standard deviation

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
Arcikam	0.694	0.938	0.998	1.000	1.000	1.000
Turing	0.740	0.982	0.994	0.996	0.998	0.998
Gaussian	0.683	0.955	0.997	0.999	0.999	1.000

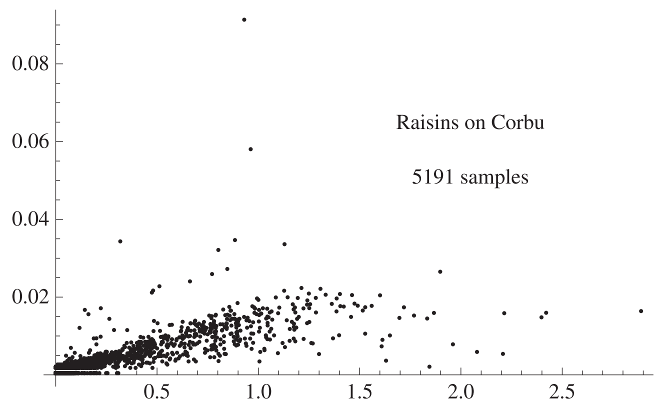

Fig. 3. Dependence between mean and standard deviation [both in s]

The data look roughly linear and indeed, linear regression gives a good fit with small variance. Furthermore, the linear approximation has an almost zero constant term, so we can conjecture that the error consists of many small parts well distributed over the whole run-time of the program. Indeed, Gaussian distribution is typical for such processes.

We also note that when we run the test on Camellia with power management turned on, there is a significant constant term, which corresponds to the time needed for an idle core to gain speed. On newer machines, the transition delay is much smaller, so power management causes only the standard deviation to increase.

3.3. *Tuning the Scheduler*

We have already explained several effects, which contribute to measurement errors. Most of them are related to the way how the OS schedules processes. We however need not blindly follow the default settings of the scheduler – many parameters affecting the scheduling decisions can be indeed configured. For example, a set of processes can be pinned to a specific core or NUMA node. In this section, we will consider several tricks of this type and quantify their effects.

We have observed that different submissions (or even different tests of a single submissions) are affected in different ways. It even happens that a single change speeds some submissions up, while slowing down the others. In order to average out these effects, we want to test several different submissions on several different inputs.

For each task, we have manually selected 5 submissions such that they finish within a reasonable time (larger than the time limit originally set for the task) without any run-time errors or security violations. We have also picked 3 input files such that the execution times of the selected submissions are not too small, so that they lie in the approximately linear region of Fig. 3. Each pair of a submission and an input (we will call it an *instance*) was run 50 times, alternating different submissions, so that the caches are flushed like in a real contest environment.

To make it possible to combine the results, we normalize them. For each instance, we calculate a reference time by dropping the smallest 2 and the largest 2 values and taking a mean of the remaining values. We then divide each measurement by the reference time for that particular instance. (Under our hypothesis on the source of the errors, this normalization is correct.) When we run the tests again under different conditions, we re-use the reference times from the base experiment.

The graphs in Figs. 4 and 5 show results for 5 experiments with the task Raisins run on Corbu. Results for other tasks and machines are similar. The left graph shows order statistics: the gray box shows the range between quantiles 0.25 and 0.75, the horizontal line inside the box is the median and the "whiskers" show the full range of the values. The boxes in the right graph are centered on the mean and their height is twice the standard deviation.

The five experiments are as follows:

- **K** – the base exeriment.
- **C** – the scheduler was told to handle the submission as a real-time task (FIFO class) with priority higher than all other user processes. This was intended to decrease effects of contexts switches. Surprisingly, while it has decreased the mean, it has increased variance.
- **R** – again with real-time scheduling, but this time we have pinned the process to a specific core, hoping to eliminate inter-core movement and cache pollution. The variance is better than **C**, but nowhere near **K**.

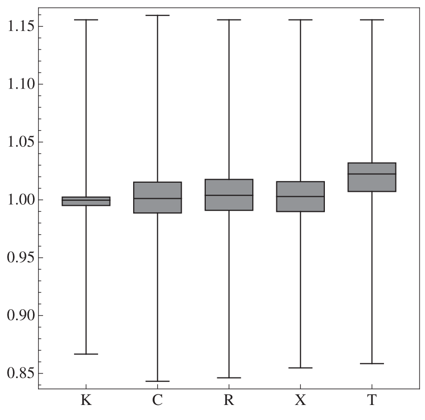

Fig. 4. Raisins on Corbu: quantiles

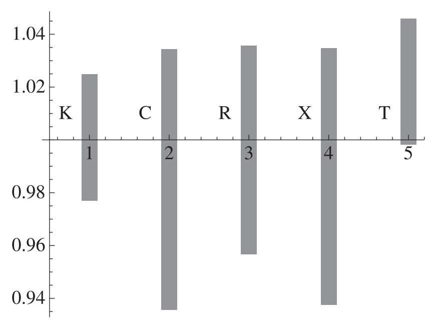

Fig. 5. Raisins on Corbu: mean and deviation

- **X** – in addition to **C**, we have turned off syscall interception in the sandbox. It is clear that the overhead of the sandbox is negligible. (In Mecho, it is slightly visible, because we need much more system calls for I/O, but it is still insignificant.)
- **T** – the base exeriment performed with an older kernel (2.6.32, approximately 13 months old). It is well visible that the scheduler was significantly improved since that version.

We can conclude that the process scheduler in the kernel is doing its job very well and that our attempts to reduce the variance by tweaking scheduler parameters are in fact making the results worse.

3.4. *Parallel Execution*

With a multiple-core machine, it is tempting to evaluate multiple submissions in parallel on different cores. Given the possibilities of interference between the submissions (e.g., competing for access to a single common memory controller or for space in caches), one should be very careful about that. We will use the method of normalized measurements from the previous section to estimate the effect of parallelization.

The results can be seen in Figs. 6 and 7: **R1**, **R2** and **R4** is the task Raisins run on Corbu (a 4-core machine) with 1, 2 or 4 processes running in parallel (**R1** is used as a base case for our normalization). **M1**, **M2** and **M4** are the same for Mecho.

For both tasks, parallel evaluation gives both larger mean (by almost 20% in **R4**) and much higher variance. Mecho is harmed much less, we conjecture that because it has much smaller memory footprint, most of the time it runs from the cache, which shields it from the other processes.

We also tried to make use of the NUMA at Arcikam: we have run two instances of the grader, each pinned to a single NUMA node. Figures 8 and 9 show the results: **R1** is the base case with a single process, **R2** uses 2 processes with default scheduling, **R2’** uses node pinning. **M1**, **M2** and **M2’** are analogous tests for Mecho. We can see that even

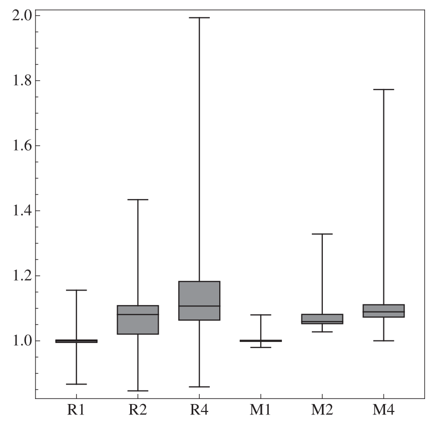
Fig. 6. Parallal grading: quantiles

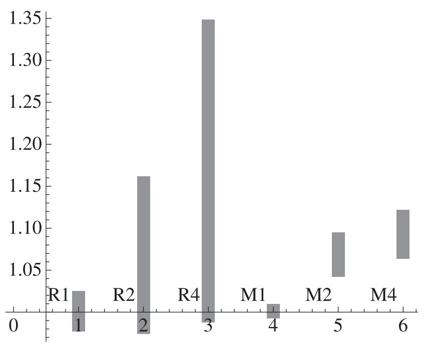
Fig. 7. Parallal grading: mean and deviation

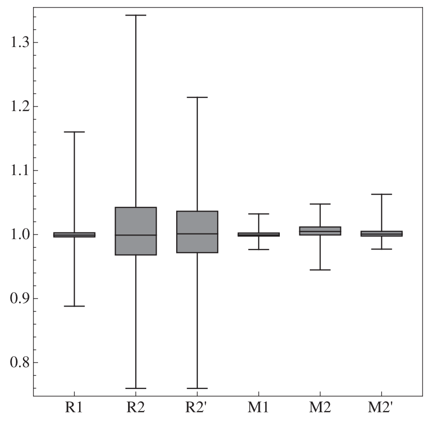

Fig. 8. Parallel grading on NUMA: quantiles

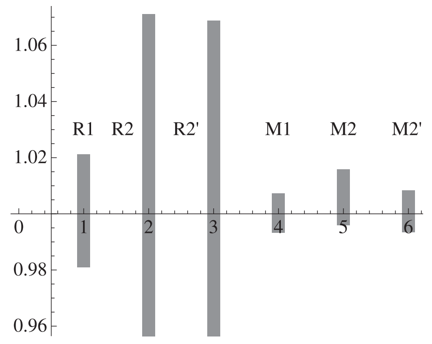

Fig. 9. Parallel grading on NUMA: mean and deviation

with default scheduling, the NUMA exhibits a much smaller variance and explicit node pinning helps even more. Again, Mecho is affected significantly less than Raisins.

3.5. *Interactive Tasks*

Let us turn our attention to Regions, our interactive task. We can expect serious problems here, because of huge amount of context switching between the submission, the judge (that is, the program generating and checking the queries) and the system call monitor of the sandbox. This was already noted by an earlier paper by Merry (2010), who studied the same task and compared performance in different types of sandboxes. We will try to gain more insight into the related issues.

Figures 10 and 11 summarize results of several experiments with Regions (we again use our normalization technique, but with only 5 submissions and 2 test inputs to make it

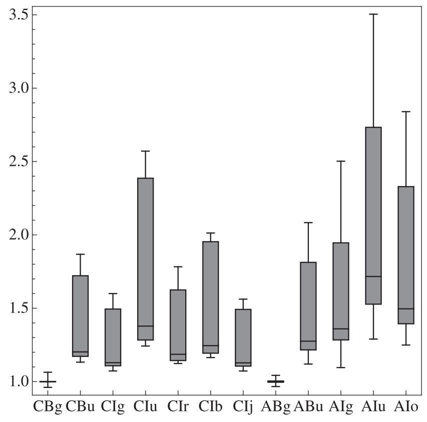

Fig. 10. Regions: quantiles

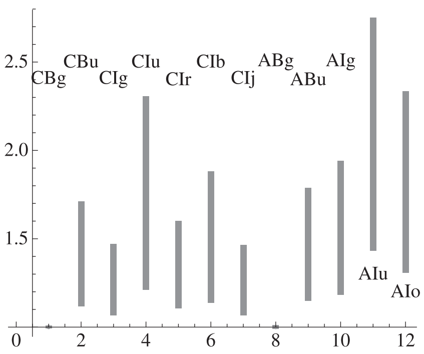

Fig. 11. Regions: mean and deviation

finish within reasonable time). The experiments are assigned three-letter codes: the first letter encodes the machine (**C** for Camorra, **A** for Arcikam), then comes **B** for the batch version of the task or **I** for the interactive version with the judge, and finally:

- **g** when running with no system call monitoring,
- **u** for a normal run (with system call monitoring),
- **b** for **u** with pinning to a specific core,
- **r** for **b** with real-time scheduling,
- **j** for **g** with pinning the submission to one core and the judge to another,
- **o** for **u** with pinning to a specific NUMA node.

As the base case for normalization, we have chosen the most straightforward version: batch mode with no system call monitoring.

It is clearly visible that system call monitoring has very significant overhead, even in batch mode (this has a simple reason: in order to work with the interactive judge, all submissions flush their output after every line, so they need at least one system call per query in batch mode). Not only that – the difference between interactive and batch runs is also immense regardless of the sandbox. Explicit pinning of processes to cores or nodes helps, but the slowdown is still very high.

A histogram of normalized execution times (e.g., of the **CIu** test) in Fig. 12 reveals a surprise: there are two clusters where the values are concentrated, located far from each other. Careful inspection of the raw data confirms that it is indeed the case – for some instances, the slowdown is much higher than for others. We do not have a good theoretical explanation of this phenomenon yet.

We get a completely different picture when we calibrate each test run against itself (that is, we do not compare with the non-sandboxed batch-mode base case and we only check consistency of each test run per se). Normalized standard deviations drop to values around 0.02, which is similar to what we got for batch tasks. Pinning still gives a small improvement. A histogram of values, again for the **CIu** test, can be seen on Fig. 13.

Grading interactive tasks using online communication with a judge process therefore does not provide realistic timing, when compared to the behavior outside the grader. This

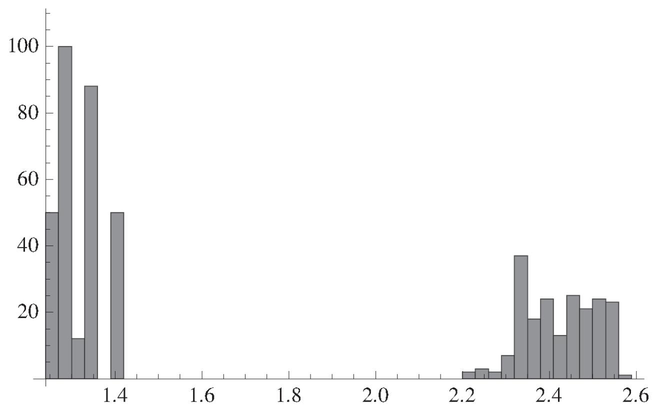

Fig. 12. Surprising behavior of the test **CIu**

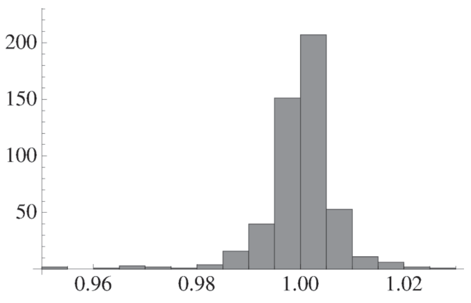

Fig. 13. Test **CIu** normalized with respect to itself

holds regardless of the sandbox. Despite of this, the timing can be still self-consistent and fair.

Compared to the Merry's paper, we have observed much more reliable results. We suspect that the difference is caused by improvements in the Linux kernel since version 2.6.30 used by Merry.

4. Conclusions

We have studied the problem of measuring program execution time in programming contests. We have performed multiple experiments with data from a real contest on different machines. We have also given theoretical explanation of some of the observed phenomena.

For batch tasks, we have found that the variance of measured values is reasonably low and that the process scheduler in recent Linux kernels utilizes the hardware fairly well. All attempts to override the scheduling decisions manually by pinning the processes to specific CPU cores or NUMA nodes made the variance worse. Also, the overhead of system call checking by our sandbox turned out to be very small.

For interactive tasks, the situation is much more complicated. We considered the typical setup with an interactive judge process communicating with the graded program over pipes. In this case, context switches between these two programs have drastic effects on timing, regardless of the sandbox used. When compared with execution outside the grader, the timings are not representative; when compared with each other, they can still be considered fair in some cases.

We recommend grading system authors take into account that unlike memory use or communication complexity, execution time behaves as a random variable encumbered by measurement errors. There are several ways how to cope with this fact. One of them is to repeat the measurements and use either minimum or mean of the results. This however makes grading much slower. Another possibility is to repeat the measurement only if the value is near the time limit (according to our data, the distribution of errors is concentrated around the mean). Or we can replace the binary time limit (passed/failed) by a continuous function transforming time to points – this is already used in the Polish olympiad in informatics for several years.

We also recommend avoiding the type of interactive tasks with a separate judge process in all cases where the number of queries is large. Uses with only a few queries (e.g., when playing combinatorial games) should be safe. When the number of queries has to be large, integrating the judge with the submission to a single process will have much better performance, but it is hard to do in a secure way. One possibility applicable to judges with non-adaptive queries could be reading the queries from a file, decrypting each query using the answers to the previous queries as a key (this enforces on-line execution). More possibilities of grading interactive tasks should be examined, for example communication through shared memory guarded by spin-locks, which might avoid scheduling altogether.

Finally, we have examined the possibility of grading multiple submissions simultaneously when a multiple-core machine is available. The variance turned out to be high, but in cases where the number of cores exceeds the number of submissions, it seems to be acceptable. Assigning processes to cores or nodes manually seems to improve the situation somewhat. Nevertheless, parallel grading should be done with a great care and checked carefully.

References

Mareš, M. (2009). Moe – design of a modular grading system. *Olympiads in Informatics*, 3, 60–66.
Merry, B. (2010). Performance analysis of sandboxes for reactive tasks. *Olympiads in Informatics*, 4, 87–94.
Tzanov, V. *et al.* (2009). Tasks and Solutions. In: *21st International Olympiad in Informatics*, Plovdiv, Bulgaria.

M. Mareš is as an assistant professor at the Department of Applied Mathematics of Faculty of Mathematics and Physics of the Charles University in Prague, a researcher at the Institute for Theoretical Computer Science of the same faculty, organizer of several Czech programming contests, member of the IOI Scientific Committee and a Linux hacker.