Associate Prof. Dr. Sascha Hunold
Klaus Kraßnitzer, BSc
TU Wien
Faculty of Informatics
Research Group for Parallel Computing

Parallel Algorithms - Scheduling

2022W
Assignment 2

date: 2022-12-14
due date: 2023-01-11
register for a group until: 2023-01-04

1. General Rules

- You may use a programming language and libraries of your choice, as long as they are freely available. The documentation of Assignment 1 lists the languages we currently support. Please consult us if you want to use a programming language that is not listed.
- Use *exactly* the input and output formats that are defined in Section 6. Your submission is tested automatically on ParJudge and is deemed incorrect if this is not the case.
- The input to your program is provided via standard in (`stdin`), use standard out (`stdout`) for printing the resulting schedule. Make sure that you do not use `stdout` for debug output (use `stderr` instead).
- You may work in **groups of two persons**. Please register for a group on TUWEL before the group registration deadline, **also if you want to work alone**. A group registration is needed for uploading a solution.
- We provide code templates for Python, Julia, C++ and Java. The use of these templates is *entirely optional* but they may help you to get a better understanding of the input and submission format.

2. What to hand in?

Your submission consists of two parts: A submission on ParJudge and a report on TUWEL.

2.1. ParJudge Submission

The source code of your implementation has to be compatible with ParJudge and is to be handed in on our ParJudge instance `https://oj.par.tuwien.ac.at` (reachable from within the TU VPN) before the submission deadline. Each submission is assigned an ID, please include the ID of the submission you want to be graded in the report you submit on TUWEL. Otherwise, we will grade you based on the *last submission before the deadline* your group made on ParJudge.

2.2. Report

On TUWEL, you are only required to submit a *single PDF file* that has to contain the following:

1. Group name, group member names and matriculation numbers
2. A description of your algorithm.
3. The ParJudge submission ID of your code that you want to be graded.

We provide an *optional* report template that you can use to make sure all the above information is included.

3. Provided Files

This section lists the files we provide on TUWEL to assist you in solving this assignment. In the *Assignment 2* submission element, you will find the following files, note that each zip is timestamped with the current date in case any of the files need to be updated (`DATE`).

- `cpp_template_DATE.zip`: C++ project template
- `instances_DATE.zip`: Sample instances, this folder is also contained in the project templates
- `java_template_DATE.zip`: Java project template
- `julia_template_DATE.zip`: Julia project template
- `python_template_DATE.zip`: Python project template
- `report_template_DATE.zip`: LaTeX report template
- `sched_assignment2.pdf`: This file
- `schedule_util_DATE.zip`: Schedule utilities, see Section A.1.

The project templates are described in more detail in Section C of the problem description for Assignment 1 (`energyscheduling`).

4. The Problem

In this assignment, we will take the role of an online batch scheduler, e.g., SLURM (c.f., Lecture 8).[1] The batch scheduler's task is to allow for a high utilization of the parallel system, by running as many parallel tasks as possible.

4.1. Machine Model

We are considering a system with m parallel, identical machines.

4.2. Application Model

In our online setting, a sequence of independent, ridged, parallel jobs arrives at the scheduler over time, one after another. The total number of jobs is denoted as n. Each job j has a release date r_j, an associated number of machines $m_j \leq m$, and a requested running time p_j. In addition, each job j also has an actual running time $\widetilde{p}_j$. Each job needs exactly m_j available machines to be executed. No two jobs can run on the same machine concurrently and the execution of each job is non-preemptive.

4.3. Online Setting

Your program has to *simulate* the functionality of an online scheduler. In the input file, you are provided with the requested *and* actual running times for each job j (p_j and $\widetilde{p}_j$, respectively). Your simulator necessarily has an inherent timestamp t at every point during the simulation. It is vital that your scheduler does not use the actual running time $\widetilde{p}_j$ until job j has actually terminated. In particular, your scheduler is not allowed to use $\widetilde{p}_j$ for any scheduling decision while $t < s_j + \widetilde{p}_j$. As soon as a job has actually terminated (i.e., $t \geq s_j + \widetilde{p}_j$), you can assume that the machines the job was scheduled on are available again.

4.4. Optimization Problem

Your task is to implement a job scheduler that maps the m jobs onto the n machines such that criterion C is minimized (see Section 4.5). In particular, the scheduler needs to assign a start time s_j to each job j such that at no time, the number of occupied machines of currently running jobs exceeds the number of available machines. Additionally, every job j must be scheduled no earlier than its release date r_j.

More formally, let J_t be the set of jobs that are running at time t. Then, the following two conditions must hold for a schedule to be feasible:

$$\forall t : \sum_{j \in J_t} m_j \leq m \quad \wedge \quad \forall j : r_j \leq s_j \quad .$$

[1] `https://slurm.schedmd.com/`

4.5. Optimization Criterion C

Our goal is to minimize the average stretch of all jobs. The stretch S_j of job j is defined as

$$S_j = \frac{s_j - r_j + \widetilde{p}_j}{\widetilde{p}_j} \quad . \tag{1}$$

Therefore, the average stretch is computed as follows:

$$S_{\text{avg}} = \frac{1}{n} \sum_{j=1}^{n} S_j \quad . \tag{2}$$

4.6. Scoring

Based on the optimization criterion C, for each test case, we assign the following score to a schedule σ your program produces:

$$\text{Score}(\sigma) = \frac{100}{S_{\text{avg}}} = \frac{100 \cdot n}{\sum_{j=1}^{n} S_j}$$

Since $\forall j : 1 \leq S_j$ holds per definition and there are 10 scored test cases, the score will never exceed 100 points per test case or 1000 points in total. Similarly to Assignment 1, the first two test cases are provided to you (`student_instance_{1, 5}.dat`) and thus worth 0 points.

5. Tasks

5.1. Feasible Schedule (7 points)

Write a program that takes a scheduling instance as an input (see Section 6.1) and that outputs a feasible schedule for this instance in the given output format. The time limit for producing the output file is *10 seconds*.

5.2. Finding a Better Strategy (4 points)

Find a better strategy to minimize the optimization criterion C. You will be rewarded the points for this task if your solution achieves over 250 points on ParJudge.

5.3. Description of Algorithm (2+1+1 points)

Write a summary on how your algorithm works. Make sure to include the following:

1. Explain the strategy of your algorithm to minimize the criterion. As in Assignment 1, it is sufficient to explain the strategy for your solution of Section 5.2 if it reliably produces feasible schedules.
2. Explain in detail why your simulator fulfills the conditions from Section 4.3. In particular, describe the (inherent) timestamp of your simulator and how it influences scheduling decisions.
3. Analyze the computational complexity of your algorithm when a new task is released, i.e., how many operations are needed asymptotically to make a decision.

6. File Formats

6.1. Input

An input file contains the properties of the n jobs, which have to be scheduled onto the m parallel machines. Therefore, each input file contains exactly $n + 2$ lines.

- The first line defines the number of machines m.
- The second line denotes the number of jobs n.

Then, each of the n remaining lines define the following job properties (all in one line):

- r_j (integer), describing the release time of the job, the file is always sorted by increasing release dates;
- j (integer), the id of this job;
- p_j (integer), the requested runtime of this job;
- $\widetilde{p}_j$ (integer), the actual runtime of this job; note that always $\widetilde{p}_j \leq p_j$ holds; and
- m_j (integer), the requested number of machines to run this job.

Input limits

You can assume that all job properties can be represented with unsigned 32-bit integers. Furthermore, for all test cases, the following bounds apply

- $1 \leq m \leq 2^{10}$,
- $1 \leq p_j \leq 10^5$, and
- $1 \leq n \leq 10^6$.

6.2. Output

The output first contains one line specifying the number of jobs (n) in the instance, which also defines the number of lines that follow, i.e., the output file has exactly $n + 1$ lines. Each line, starting at line 2 (if we start counting at 1), contains exactly two values:

- j (integer), the job id of the current job, and
- s_j (integer), the assigned start time of job j.

The output file should be sorted by increasing job id.

6.3. Sample Input

You can find this file (`student_instance_1.dat`) in the `instances` folder that is contained in all project templates and is also found separately on TUWEL.

```
32
64
273 1 1200 8 8
1579 2 25200 5 8
1601 3 36000 7 4
1729 4 10800 2 6
1785 5 36000 61 8
1809 6 300 13 4
1840 7 36000 12 4
..
```

In this case, we have a system with $m = 32$ machines and $n = 64$ jobs to be scheduled. The third line states that the first job $j = 1$ arrives at time $r_1 = 273$. The requested running time p_j of this job is 1200 time units, but it has an actual running time $\widetilde{p}$ of only 8 time units. In order for job j to run, $m_1 = 8$ machines have to be available.

6.4. Sample Output

```
64
1 273
2 1579
3 1601
4 1729
5 1785
6 1809
7 1840
..
```

The output is straight forward. The scheduler has mapped $n = 64$ jobs to the machine, where job $j = 1$ starts at time 273, job $j = 2$ at time 1579, and so forth.

A. Provided Code and Templates

A.1. Schedule Validation

The archive `schedule_util_DATE.zip` provided on TUWEL might aid you in verifying the correctness of your generated schedules. To run the script, a working Python 3 installation is required.

The script `validate_batch_schedule.py` takes an instance (first positional argument) in the given input format as well as a schedule for that instance that can either be supplied using the `-s` argument (file path) or piped to the standard input of the script.

Example Usage

Assuming you wrote a schedule for `instances/student_instance_1.dat` to the file `schedule.out` you can use the script as follows:

```
$ python3 validate_schedule.py instances/student_instance_1.dat -s schedule.out
```

If the schedule is valid, `validate_schedule.py` will print "`Schedule is VALID`" and exit with exit code 0, otherwise "`INVALID Schedule`" as well as an error message indicating in which way your schedule is invalid is printed, and the verification script exits with exit code 1.

Alternatively, if supported by the system you are using, you can "pipe" the output of your program to the schedule verifier (this assumes your program is run using `./run` like in the provided templates):

```
$ ./run | python3 validate_schedule.py instances/student_instance_1.dat
```

A.2. Solution Templates

Like in Assignment 1, we provide solution templates for C++, Python, Java and Julia. Since the templates are in a very similar format to those for Assignment 1, please consult the documentation for Assignment 1 for more details on template structure and usage.