TU WIEN Informatics Parallel Computing

Research Group for Parallel Computing
Faculty of Informatics
TU Wien

# Advanced Multiprocessor Programming

2023S

Mandatory assignment: ssh key for system access

Issue date:: 2023-04-16
Due date:: 2023-06-19 (23:59)

For this project you have to implement the results of *one* of the papers presented in the lecture in groups of up to three students. The deadline is Monday, the *19.06.2023 at 23:59:59*.

## 1 Requirements

Your submission has to fulfill the following three requirements.

- The chosen algorithm is implemented correctly. Use assertions to verify your assumptions. This implies that it runs as specified on any legal input and does not crash.

  *Note:* dynamically allocated, distributed memory may be leaked. We do not expect you to implement concurrent memory management.

- Your submission provides a *Makefile* which at least knows the two targets `small-bench` and `small-plot`.

`small-bench` Builds your submission and executes some short representative benchmark of your choice, running for *at most 1 minute* on the cluster.

*Note:* The important part here is the 1 minute. You probably cannot pack all interesting benchmarks into this time restriction and we are curious about the justification of your choices.

`small-plot` Produces a plot or multiple plots from the data gathered of the previously run target `small-bench`. This plot or these plots are representative for the results you show in your report and can be used to quickly verify them.

*Note:* The plots should be created in the folder `plots/`. Print their filenames after creation for easier localisation.

- Provide a short report discussing the problem and your results. You do not need to answer each of the following questions individually, they should just guide you through your report.
  - Can you verify the claims made in the paper? Did you find some other realistic benchmark which produces very different results compared to the results from paper?
  - Measure at least the *throughput* (successful operations in some given timeslot) and the *latency* (typical time per operation) of your implementation.

Is your mix of operations representative of how the implementation would be used? What assumptions did you make for the 'typical use case'? Here it is probably infeasible to benchmark all relevant use cases, why are your choices interesting and relevant to look at?

What is your baseline? Does the parallel implementation offer any benefits compare to the best known baseline?

- State properties of your implemented algorithm and give worst-case bounds if possible. Proofs are not required.
- Think of the global interpreter lock in python which makes concurrent *python* applications impossible, does it impair the test results obtained via the provided framework? Why not?
- You may have to take some design decisions, like which benchmark you use in the target `small-bench`. Document them here. Why did you take that decision? Are there trade-offs?

Some of the projects are based on research papers with pretty advanced results and many algorithms. It is *not required* that you implement all of the proposed solutions in the time available. Your task is to also distill out and understand the basic core algorithms and implement these in a sensible way, but do not trivialize your assignment. When in doubt, write us a short email!

## 2 Framework

We provide you with a framework to benchmark your submission with, *but you are not required to use it*. It consists of the following files.

`src/library.c` Is an example implementation of a concurrent data structure, a library for threads. It contains two books which can be lent out and returned via the respective functions.

`plots/avg_plot.tex` Contains the tikz source for a plot of some data in the `data/` folder. It depends on the `\DATAPATH` definition which is set in the `Makefile` and allows to vary the data source, i.e., the benchmark run.

`report/report.tex` Is a short latex template for your report. It includes the plot generated from the `make small-plot` target so it will throw an error if that was not run before.

`Makefile` Is an exemplary `Makefile` providing, amongst others, the two required targets. Running `make all` compiles `library.c` into an executable and a shared object. Running `make small-bench` runs a small benchmark and `make small-plot` compiles one LaTeX-plot in the folder `plot/` with the newly generated data.

`benchmark.py` Showcases how data could be collected. It contains the `class Benchmark` which holds all results from a certain benchmark run and writes those results timestamped back into the `data/` folder.

Communication between python and C is relatively pain-less. The C file has to be compiled as a shared library which can be directly imported into a python file as an external library. As python does not know the return types of the

benchmark functions in C we have set them explicitly. After that we can just call them directly. This allows the tedious data collection and post-processing to be offloaded onto python, with very little additional efforts. For further reference see the official documentation.

To get started you can run `make small-bench`, `make small-plot`, followed by `make report`. This will build the sample algorithm, run its small benchmark, plot the results and assemble a short sample report.

## 3 Submission

Upload your source code, together with the report and a `Makefile` in a in `.zip` archive onto Tuwel until *18.06.2023 23:59:59*. It is sufficient that only one member of you group submits your solution.