UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Check-pointing Long-running Applications in Python

Bachelor project

Jacob Olesen (slc458)
Leeann Quynh Do (hlp848)

Kenneth Skovhede

Handed in: April 13, 2020

Abstract

Contents

1 Introduction

WELCOME TO A BACHELOR PROJECT WITH JACOB AND LEEANN!!

TODO: INSERT ADDITIONAL TEXT HERE AS INTRODUCTION

1.1 Motivation

The motivation behind this project is primarily to aid researchers and scientists in their studies, as most would currently need to write their own checkpointing function in a program. This is not easy, and is not meant to be part of their work, so a dynamic `python` library which can checkpoint every python application is highly desirable. This is primarily relevant in terms of research and work which uses simulations such as molecular dynamics or climate, which can run for weeks, months or even years. Having an easy, dynamic, and efficient way to checkpoint these programs would make their lives much easier.

1.2 Checkpointing

Checkpointing is a technique for saving and restoring the state of a partially executed program to ensure that a program can continue operating properly in the event error or failure. The technique consists of saving a snapshot of the state of the program, so that the program can restart from that point in case of failure [1]. Checkpointing is a good method for preventing eventual loss of time, in the sense that it is not necessary to rerun the program from the beginning, but only from the last checkpointed state.

2 Theory

2.1 Serialization and deserialization

Serialization of data is the process of translating a state of an object or a data structure into a format, e.g. a byte stream, that can be stored in a file or transferred between computing system, e.g. across a network connection or via a storage unit, such as a USB stick, and can later again be reconstructed to the exact same state, even on another machine. This opposite operation of translating a byte stream into an object or a data structure, is called deserialization [2].

2.2 Python

Interpreted vs compiled
no program counter.

2.3 Existing solutions

`python` already have some existing solutions to serializing and deserializing objects or data structures. For `python`, these are the `python` modules `pickle` and `dill`. There are different solutions for different languages, `C#` for example has a `[Serializable()]` keyword [3].

TILFØJ FLERE MODULER ELLER LØSNINGER SOM OGSÅ KAN BRUGES TIL DETTE.

2.3.1 `pickle`

The `pickle` module is a module in `python` for serialization and deserialization. The module uses the terms `pickling` and `unpickling` for `serialization` and `deserialization`, respectively.
The format that the `pickle` module translate data to, is specific to `python`. This means that programs written in anything else than `python`, are not guaranteed to be able to reconstruct pickled `python` objects [4].

Pickle is written in pure `python`, but has an optimized version which is written in pure `C` called `cPickle` [5], and is up to 1000 times faster [6]. `pickle` attempts to import and use `cPickle`, but falls back to the pure `python` version in case of errors. `cPickle` does not support quite as many features and types as `pickle`.

2.3.2 `dill`

`dill` is an extension to the `pickle`-module. In addition to pickling and unpickling `python` objects, `dill` also allows the user to save the state of the interpreter session, so that if the program were to continue on another machine, it is possible to continue from the saved state of the ‘original’ interpreter.
`dill` is very flexible, as it allows arbitrary user defined functions and classes to be serialized. The user thus has to be aware if the data they are trying to serialize is trustworthy, as `dill` does not protect against maliciously constructed data.

Some types that `dill` does not support are `frame`, `generator` and `traceback` [7].

3 Setup

In order to have a proper, iteratively structured program for use and benchmarking throughout this project, we were advised to use a heat equation simulation program written specifically for benchmarking [8]. This is an ideal program for this project, as it is structured the same way as the simulation programs which we aim to be able to checkpoint.

3.1 `dill`

Our initial implementation efforts were made in collaboration with our initial research in order to gain a proper understanding of exactly what was happening under the hood of the program, and, more specifically, how `dill` works. `dill` is initially able to make a perfect checkpoint at an arbitrary point in the program, but due to `python` having no program counter, the program is unable to continue from a specific instruction. Thus, we had to rewrite the program to be able to properly continue from a checkpoint by calling a function. What this meant in practice was that the `main()` function was rewritten from

```
def main():
    H = bench.args.size[0]
    W = bench.args.size[0]
    I = 50000

    grid = bench.load_data()
    if grid is None:
            grid = init_grid(H, W, dtype=bench.dtype)
    ...
```

to

```
def main():
    global H, W, I, grid

    if grid is None:
            grid = init_grid(H, W, dtype=bench.dtype)
    ...
```

This was done to allow resuming the program from a checkpoint by simply calling `main()`. Prior to this change, calling `main` after de-serializing a state would simply re-initialize the variables used, which in practice means that the program starts over.

This allowed us to define the first limitation of `dill`.

3.1.1 Limitations

To make checkpointing work properly, the program needs to have any consistently used variable initialized independently of the looped function, or the function that initializes/resumes the loop. This was not the case in the above example, and the variables was therefore made global and initialized independently, outside of any functions, such that they would not be overwritten upon resuming the program by calling the `main()` function again.

Furthermore, `python` 3.0 was intentionally made backwards incompatible [9], and therefore any program which wishes to use checkpointing must be written in `python` 3.0 or later.

4 Analysis

Upon discovering the `dill` [7] module, we decided to analyze it prior to implementing our own version, as it might be be favourable to simply extend or improve this existing module rather than re-invent the wheel from scratch.

4.1 Specifications

Every graph or profiler run was run on a desktop machine running an almost fresh install of manjaro linux with only the necessary modules and packages installed. The machine has an i7 6770 processor and 16GB of DDR4 ram. Every time a test, graphing, or profiling of a program was done, no other programs were running on the computer, and the internet was turned off to ensure that no background applications were doing anything inconsistent or unexpected.

4.2 Analyzing the Cost of Checkpointing

In this heat equation simulation program, we decided to dump the session every 100th iteration with the `dill` function `dump_session`, and by using the global variable `counter`.

```
if counter % 100 == 0:
    dill.dump_session('bch.pkl')
```

To see how much time each iteration is running for, we have added a start time, and an end time by using the `timeit.default_timer()` function call, which on linux uses the `perf_counter` [10] – *TIL VEJLEDER: Should this be CPU time instead??.*
By doing this, we get an overview of exactly how long each iterations runs for, and we can easily separate the iterations in which we create a checkpoint from the iterations in which we don't. Note that the timers are started and stopped immediately before and after the `loop_body` function call, which is the part of the program actually performing the calculations, simulations, and checkpointing.

```
while True:
    if counter < max_iterations:
        start_time = timeit.default_timer()
        loop_body(grid)
        stop_time = timeit.default_timer()
        with open('time.txt', 'a') as f:
            f.write(str(stop_time - start_time) + '\n')
    else:
        break
```

Each measurement is then saved to a text file which we used to create a graph and calculate the difference between the normal iterations and the iterations in which we checkpoint the program.

4.2.1 Graphing

We have run the `benchmark.py` program with 3000 iterations, and this has given us a file `time.txt` with timings for each iteration. We have chosen 3000 iterations so that the graph would become easier to visualize, as the data points are further from each other, compared to graphing many iterations, where the data points will be cramped together. With less iterations, it is also a lot more manageable to see the difference between the two types of iterations - when we checkpoint and when we do not.

The iterations and the time for each are plotted in the following graph

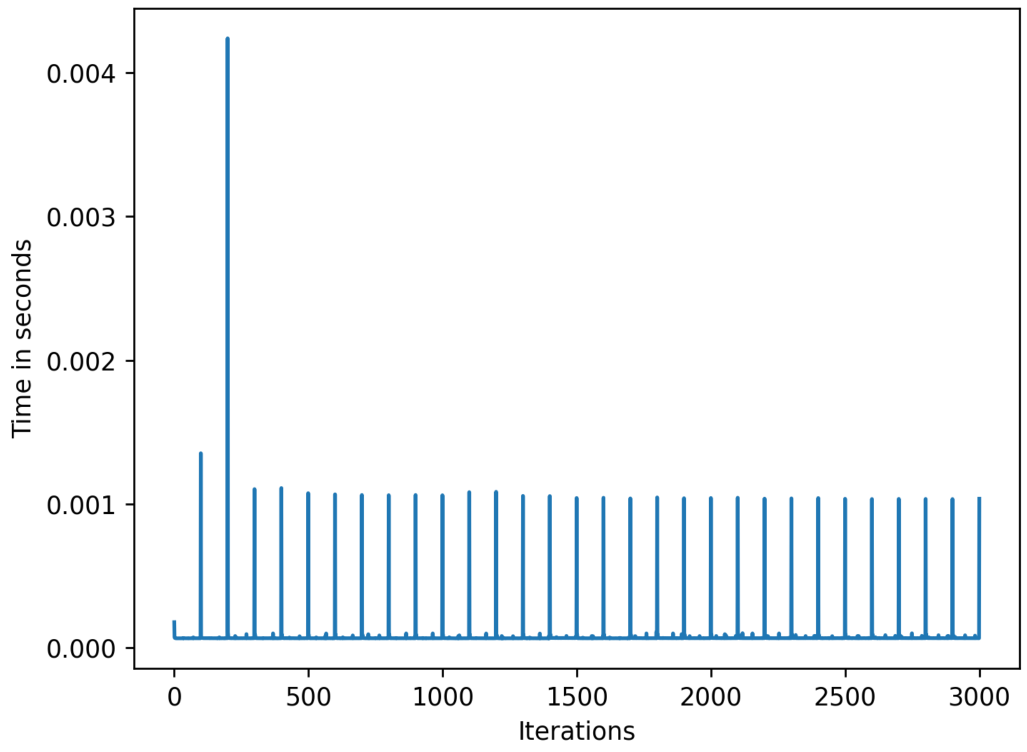

Figure 1: Graph with 3.000 iterations

In this graph, we see that every 100th iteration makes a spike in time, which is to be expected, since we this is where we to a checkpoint.

We notice that the two first checkpoints take longer time than the rest of the checkpoints.

4.2.2 Performance Difference

From the graph above, it can be seen that the iterations in which the program is checkpointed take considerably longer than the rest of the iterations. It can also be seen that the amount of time needed to checkpoint iterations is consistent, except for the two first checkpoints that take longer and are inconsistent compared to the rest. These 'inconsistencies' are consistent across different systems and different amounts of iterations, as the first two checkpoints always take longer than the rest, with the second being the most costly by a lot.

To counteract this and get a more reliable result with less variance, we ran the program again, this time increasing the number of iterations to 50.000. Afterwards, we took the average of every iteration in which a checkpoint was made and compared it to the average of every normal iteration.
The result:

- With checkpointing: 0.0054479 seconds
- Without checkpointing: 0.0000690 seconds

Comparing these show that the checkpointing iterations take $78,94$ times longer, or 7894% longer to run. While this is a lot, it is very possible that programs which take longer to perform an iteration would see a significantly smaller percentage wise cost of checkpointing, as it is possible that a lot of the time used to checkpoint is used simply writing to the file or loading and initializing the code required to checkpoint.

5 Implementation

A section where we can talk about our optimization/implementation of a checkpointing feature/solution.

6 Testing

7 Discussion

7.1 Program counter

How did it affect us, that there was no program counter?
Ja, men AST og man kan måske gøre noget hacket med det, men det er væsentligt sværere fordi det er et interpreted sprog.

7.2 Security

Most efficient serialization and deserialization modules in `python` are vulnerable to arbitrary code execution as they store the data as bytes and load them without any 'validation'. This can, of course, be a major point of concern in certain environments and use cases. However, we do not consider this to be a cause of concern in our project, as the purpose of this program is for users to only serialize and deserialize their own programs in case of errors during runtime, which should never lead to any dangerous code getting executed

7.3 Limitations

Are the limitations presented in **Setup** fair? Are they too much, should they be handled, what can be done about them, is it worth the effort etc.

8 Conclusion

9 Future Work

Snak om at omskrive dill til et andet sprog som C for at opnå samme performance improvement som forskellen mellem pickle og cPickle.

SNAK OM AST: https://realpython.com/cpython-source-code-guide/part-3-the-cpython-compiler-and-execution-loop
https://docs.python.org/3.9/whatsnew/3.9.htmlimproved-modules

10 Appendix

References

[1] April 6, 2020. URL: https://en.wikipedia.org/wiki/Application_checkpointing.

[2] March 4, 2020. URL: https://en.wikipedia.org/wiki/Serialization.

[3] April 6, 2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/.

[4] March 4, 2020. URL: https://docs.python.org/3/library/pickle.html.

[5] April 6, 2020. URL: https://docs.python.org/3/library/pickle.html#performance.

[6] April 6, 2020. URL: https://docs.python.org/2/library/pickle.html#relationship-to-other-python-modules.

[7] March 4, 2020. URL: https://pypi.org/project/dill/.

[8] February 7, 2020. URL: https://benchpress.readthedocs.io/autodoc_benchmarks/heat_equation.html.

[9] April 6, 2020. URL: https://docs.python.org/release/3.1.5/whatsnew/3.0.html.

[10] April 6, 2020. URL: https://docs.python.org/3/library/time.html#time.perf_counter.