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: May 23, 2020

Abstract

Denne skrives til sidst :-)

Contents

1 Introduction

With the rise of high level programming languages like `python`, as well as the increased availability of high performance computers, long running programs, such as simulations, are becoming more and more common. These often need to run for weeks, months, or even years in extreme cases, but as with anything related to computers, backups should be made often, to avoid data loss in case of a system failure. This, however, is not a trivial task, and people who specialize in a subject other than computer science do not necessarily have the knowledge required to successfully implement it.

1.1 Motivation

The motivation behind this project is primarily to aid researchers and scientists in their studies, as most would currently need to create and implement their own checkpointing functionality in a program. As mentioned in the Introduction, this is not a trivial task, and furthermore, it is not meant to be part of their work. Therefore, a dynamic `python` library, which can checkpoint every `python` application, is highly desirable. This is primarily relevant in terms of research and work, that uses simulations, such as molecular dynamics or climate, which can run for weeks, months or even years.

These long running programs often run on dedicated and powerful servers, however, it is not assured that these servers will not run into problems, resulting in downtime. If the long running programs do not have a checkpointing functionality in the event of server downtime, all progress is lost, and the program will have to start over.

Having an easy, dynamic and efficient way to checkpoint an arbitrary program written in `python` would make their - researchers and scientists - lives much easier and increase efficiency of the their work, as they now have more time to focus on the actual functionality of the program, rather than creating fail-safes or restarting the program in case of system failure.

1.1.1 Supporting data

We have received data about the jobs currently running on the University of Copenhagen's cluster. Filtering the currently running jobs to only show processes that have been running for 40 or more days return 88 results [A.1], with the longest having been running for 230 days as of May the 7th.

These data emphasize the need of a checkpointing functionality, since if the server were to run into a problem, and these programs would have to start over, the researchers and scientists would have wasted a lot of time ranging from one month to seven months of progress.

Furthermore we have received some data showing the uptime of some of the servers [A.2] which are running on the NBI cluster at University of Copenhagen. The average uptime of these machines is 102.4 days, and seeing as they are only restarted upon encountering a critical error, crashing or for security updates, scheduling a program which needs more than 102.4 days to finish without checkpointing is risky.

As evidenced by a research report released in 2018, even the most experienced host providers experience some downtime due to unforeseen circumstances [1]. Furthermore, the following report - while only covering the period from 2007-2013 - details several incidents where even the biggest internet companies have been completely down [2].

Those sources prove that 100% uptime is almost impossible and unexpected downtime occurs for a number of reasons.
Any machine is subject to unexpected downtime, but with a checkpointing functionality, this downtime will be a minor inconvenience - resuming from a checkpoint - instead of a disaster - starting over.

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 [3]. 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. This format could be 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. Later again the object or data structure can 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 [4].

2.2 Python

This project is specifically for programs written in the language `python`, as that is frequently used due to it's simplicity and accessibility, which allows even people without advanced coding capabilities to read and write programs. As a result of these properties, `python` is often used by scientists who don't necessarily have a lot of coding experience.

In addition to being simple, `python` is an interpreted language, which adds to its accessibility and has a series of other upsides, but it has one rather big downside in relation to this project. Compiled languages have a program counter [21] - also known as an instruction pointer - which is a register pointing to the next instruction in the program sequence. This is very useful when checkpointing as it allows, in theory at least, to simply save the program counter as part of the checkpointed program. When restoring the program from a checkpoint, it would then know exactly from which instruction to continue the execution. Without the program counter, this is not going to be possible, at least not in the same sense. What `python` does instead if keep a stack of every call to functions, methods, classes, etc.. This stack consists of PyFrames [22] which contain not only an individual program counter for each callable, it also contains the code and local variables. Interacting with these frames is the key to creating a seamless checkpointing solution. This is discussed more in-depth in Section 8.2.

An interesting part of `python` is that it is actually a semi-compiled language in the sense that source code is compiled to `python` bytecode, which is platform independent, meaning that a 'compiled' `python` program will behave the exact same across operating systems [23]. This bytecode is then interpreted by compiled `C`-code, at which point the program execution becomes platform- and operating system specific. Furthermore, `python` bytecodes generated by one version of python will likely not be interpreted correctly by a different version of python as they are often subject to change and optimization [24].

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.

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 [5].

Pickle is written in pure python, but has an optimized version which is written in pure C called cPickle [6], and is up to 1000 times faster [7]. 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 [8] 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. Thus, the user has to be aware, if the data they are trying to serialize, is trustworthy, as dill does not protect against maliciously constructed data.
The limitations of this module - and cloudpickle - are discussed further in Sections 3.1.1 and 6.1.

2.3.3 cloudpickle

Another version of the pickle module is the cloudpickle [9] module. This module seemingly supports serialization to the same degree as dill, but upon researching for this project, dill had a lot more documentation and resources to help understand the module and its inner workings.

Both `cloudpickle` and `dill` are written entirely in `python` extending the `python`-level `pickle` API to serialize objects.

3 Implementation

In order to have a proper, iteratively structured program for use and benchmarking of different implementations of checkpointing throughout this project, we were advised to use a heat equation simulation program written specifically for benchmarking [10]. 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.

We have chosen to use the `dill` module instead of `pickle` as it supports more types, e.g. `numpy` types. `cloudpickle` seemingly support the same types as dill, but we only discovered this module later in the process. Another reason for choosing `dill` over `cloudpickle` is that the `dill` module has a lot more documentation than `cloudpickle`, and thus it seemed a more secure choice.

3.1 Benchpress - Heat equation

This program solves the heat equation [11] using the Jacobi method [12].

3.1.1 `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 checkpoint at an arbitrary point in the program, but due to limitations in the `python`-level API, the context of function calls - i.e. frames on the underlying `C`-stack - and their local variables cannot be saved [25]. 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)

    grid = jacobi(grid, max_iterations=I)
    ...
```

to

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

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

    grid = jacobi(grid, max_iterations=I)
    ...
```

This was done to allow resuming the program from a checkpoint by simply calling `main()`. Prior to this change, calling `main` after deserializing 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`:
The variables saved by `dill` have to be part of the **__main__** state of the `python` program in order to be serialized. This means that they have to be defined in the top-level code of the program, meaning they have to be unindented and not defined inside of a function. In short, variables which are local to a function are not saved, only global variables are saved [26] [27].

3.1.2 Implementing the Checkpoints

In this heat equation simulation program, we decided to dump the session every 100th iteration with the `dill` function `dump_session`, and by using a global variable `counter`. We decided to checkpoint rather often to easily be able to spot inconsistencies and to create clear and manageable graphs of the results. 100 was an arbitrary number, and there are also graphs and analyses of checkpointing with less frequency later in this report.

```
if counter % 100 == 0:
    dill.dump_session('dump_file.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` [13], which measures actual time elapsed, not CPU time.
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. The graphs can be seen in Section 4.2.2.

3.2 VEROS

After having a thorough look an the Heat Equation and the `dill` module for checkpointing, we would like to look at a larger simulation, to represent that scientists often have simulations running for days, months or even years. For this, we look at `VEROS` [14], which is a versatile ocean simulator written in pure `python`. Another reason for picking `VEROS` is that attempting to implement checkpointing in an already existing code base would be a good indicator of how easy or difficult it would be to implement in similarly sized, established projects.

To run `VEROS`, we needed to set up a model, and the model we used was a pre-implemented model called the `ACC model`.

3.2.1 VEROS' own chekpointing functionality

Upon looking into `VEROS`, we discovered that `VEROS` has its own checkpointing functionality [15]. We would like to compare this implementation and its performance with that of the `dill` module.

The checkpointing functionality in `VEROS` is seen in the model setting `VerosState.restart_frequency`.
By setting `VerosState.restart_frequency` to a finite value, restart data is written in the time interval given. This is `VEROS`' own serialization functionality.

In the `ACC model`, we have added the checkpointing functionality in the part where output data is written as follows:

```
...
@veros_method
def set_diagnostics(self, vs):
    vs.diagnostics['snapshot'].output_frequency = 86400 * 10
    vs.diagnostics['averages'].output_variables = (
        'salt', 'temp', 'u', 'v', 'w', 'psi', 'surface_taux', 'surface_tauy'
    )
    vs.diagnostics['averages'].output_frequency = 365 * 86400.
    vs.diagnostics['averages'].sampling_frequency = vs.dt_tracer * 10
    vs.diagnostics['overturning'].output_frequency = 365 * 86400. / 48.
    vs.diagnostics['overturning'].sampling_frequency = vs.dt_tracer * 10
    vs.diagnostics['tracer_monitor'].output_frequency = 365 * 86400. / 12.
    vs.diagnostics['energy'].output_frequency = 365 * 86400. / 48
    vs.diagnostics['energy'].sampling_frequency = vs.dt_tracer * 10
    # Checkpointing functionality
    vs.restart_frequency = 86400 * 100
    vs.restart_output_filename = "restart_data.h5"
...
```

We have set the restart frequency to be 86400×100, meaning that restart data will be written every 100th day in the simulation, as 86400 represents a day in seconds.
To resume from a checkpoint, and use the restart file as initial conditions, the `VEROS` command `VerosState.restart_input_filename` should be given in the terminal, with the restart file as input. In the terminal, this could be written as

```
python acc.py -s restart_input_filename restart_data.h5
```

We observe that the new simulation starts with the restart data from the previous run, but we also notice that the simulation progress is not saved. Thus, it continues from the checkpoint, but since the total runtime, the parameter `vs.runlen`, is set to one year, it runs for additionally one year from the point at which a checkpoint was made. This is the result of the variables being reset upon starting the program

- even from a checkpoint. This was a minor thing and the program variables could be altered in the program source code to account for a specific checkpoint before reloading, but it was notable nonetheless.

3.2.2 Implementing `dill`

To implement the `dill` module's checkpointing functionality in `VEROS`, we navigated to the main loop and utilized an already existing counter function used to count number of simulation iterations to checkpoint. This check was inserted in the very end of the main loop.

```
if pbar._iteration % 200 == 0:
    dill.dump_session('checkpoint.pkl')
```

While this worked and we were able to resume from the created checkpoint, `VEROS` is structured in such a way, that all of the custom variables are reset upon starting the program, even if the main loop is called directly. Thus, we encountered the same issue we had encountered upon initially attempting to checkpoint the heat equation program. Setting out to fix this with if-statements checking if variables were already initialized proved to be insufficient, and several other files and functions had to also be modified to accommodate resuming from our checkpoint. After working on this for a while, this approach was dropped, as it was more of a general refactoring of a large part of the program, than it was an implementation of checkpointing.

3.3 Lattice Boltzmann

3.4 Shallow Water

3.5 Limitations

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

Due to structural changes in `python` over the course of the langauge's lifetime, any program that implements this version of checkpointing must use `python` 3.2 or later [8]. Furthermore, the way a program is compiled to python bytecode and interpreted has changed numerous times, and a checkpoint must therefore be restored using the same version of `python` as it was created with [24].

4 Analysis

Upon discovering the `dill` module [16], we decided to analyze it prior to implementing our own version, as it might be favourable to simply extend or improve this existing module, rather than re-invent the wheel from scratch.

4.1 Specifications

Every graph or profiling of a program was run on a desktop machine running an almost fresh install of Manjaro Linux with only the necessary modules and packages included. The machine has the following specifications:

Module:	Version:
CPU:	Intel i7 6700 4 core 3.40 GHZ
RAM:	16 GB DDR4 2133 MHz
Disk:	480 GB SSD 560/530 (read/write MB/s)
OS:	Manjaro 19.0.2
Python version:	3.8.2

Table 1: Specifications of the machine

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

When analyzing the cost of the checkpoints, we have used the data that was generated by the code from Section 3.1.2.

We have written a `bash` script which executes the `benchmark` program 2000 times. In the following, when we refer to an execution, it is an execution of the whole program via the command '`python3 benchmark.py`'. In the `benchmark` program, we limit the number of iterations of the `loop_body` to 3000 iterations. Thus, when we refer to an iteration, it is one of the iterations of the `loop_body`, where the calculations are performed and the variables are altered.

4.2.1 RAM disk

The following graph illustrates our initial attempt to benchmark the heat equation while checkpointing every 100th iteration of the program to the disk. This resulted in the program running into I/O blocking around every 10th execution of the program, in which the program would halt and wait for the disk to finish writing. This could total up to a second of wasted time, and produced, as seen below, very irregular results.

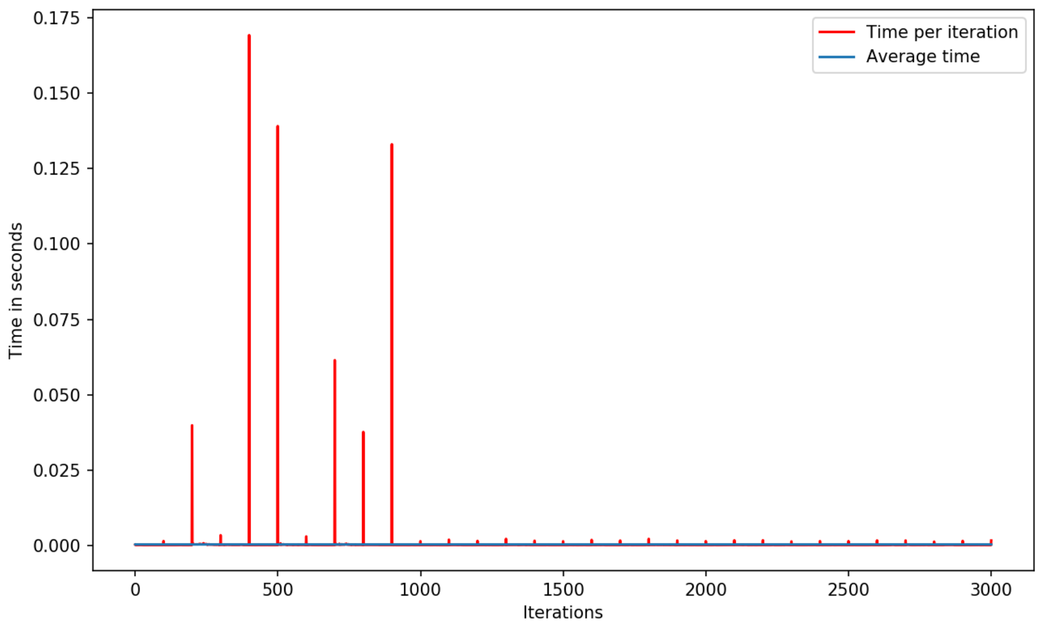

Figure 1: I/O blocking during 12th execution

The solution was using a RAM disk [17], in which we allocated 1024MB of RAM memory and wrote the checkpoint to that, instead of the disk itself. This minimized the time spent waiting for I/O blocking and also removed the irregular results.

The irregularities seen on the graph likely happened because the disk ran out of buffer space due to the constant writing.

Thus, we will be writing every checkpoint in the following checks to a RAM disk, in order to get consistent results, which are based on the performance of the program and not the limitations of the hardware.

This is, of course, a temporary solution, as saving checkpoints to volatile memory is no more useful than simply not checkpointing at all, as in the case of total system failure, the volatile memory is wiped. We therefore need to find a solution to the disk speed. For now, however, we use a RAM disk to illustrate performance difference between the different program versions.

4.2.2 Graphing

We have run four different versions of `benchmark.py`, each capped at 3000 iterations, 2000 times. Out of these 2000 executions of the program, we have discarded the 10 first of them, to warm up the system, and get more consistent results. In each graph the blue line indicates the average time spent on each iteration, the red bars indicate the variance in each iteration, ranging from the worst to the best case of the 1990 executions, and the green line shows the total average of all the executions of the 3000 iterations. Note that the green line is only really visible on the graph in which a checkpoint was created every 100 iteration.

The first version was a stripped version, where no modules were loaded and no checkpointing was done. The following is graph of the first version.

Extra modules loaded	No
Checkpointing	No
RAM disk	No
Normal disk	No
Objective	Show that the iterations of the original program are consistent

Table 2: The first version: The stripped version

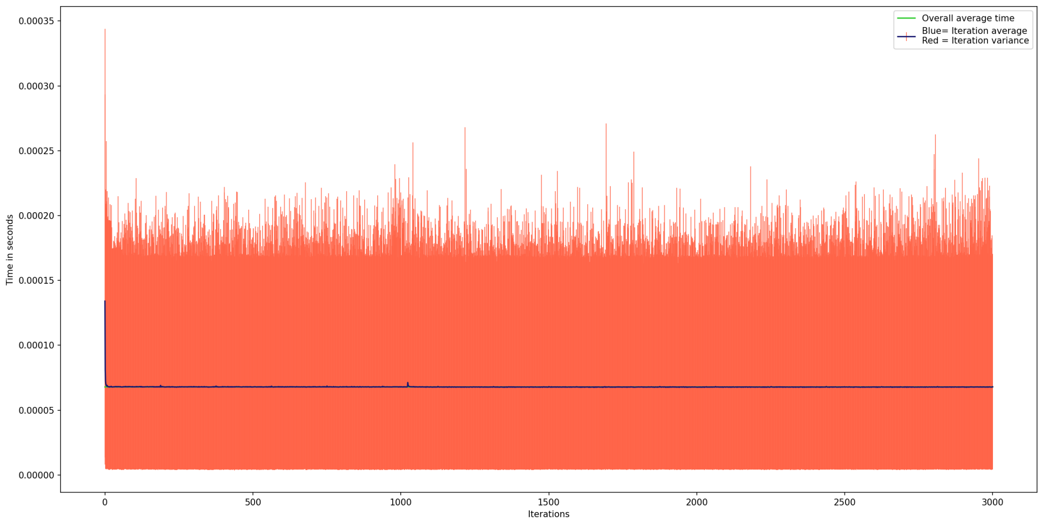

Figure 2: Stripped benchmark

The second version was where we only loaded the modules needed for checkpointing, but without actually checkpointing.

Extra modules loaded	Yes
Checkpointing	No
RAM disk	No
Normal disk	No
Objective	Show that the iterations of the program are consistent, except the first couple of iterations, where the modules are loaded

Table 3: The second version: The version where we load the modules

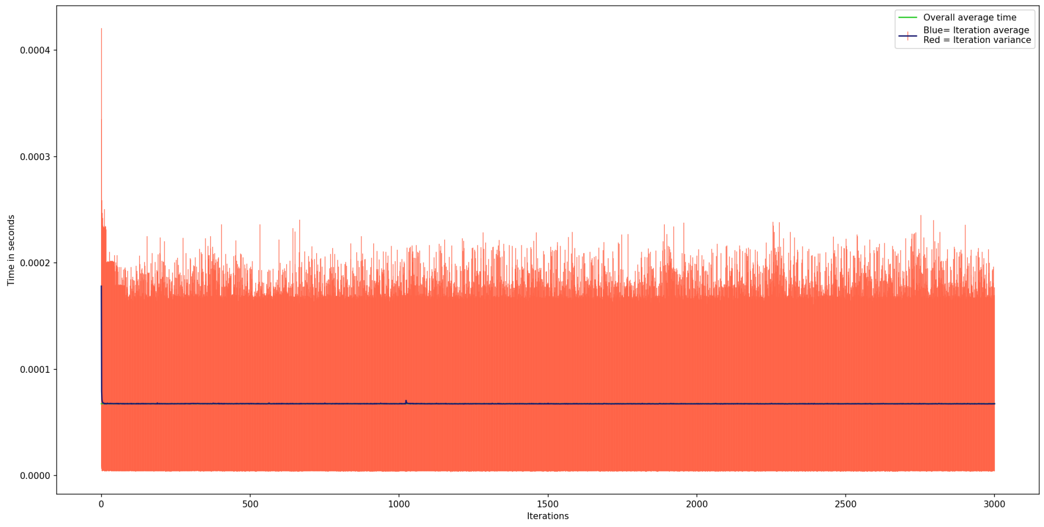

Figure 3: Benchmark with modules loaded

When we compare the two first graphs, the stripped graph and the graph where we load the modules, we notice that they are almost identical. The most noticeable difference is that in Figure 3, we see that the time used in the first couple of iterations, more specifically the first two take longer than in Figure 2. As we need to load the modules in this version, it naturally takes longer than if we did not. Although we have this difference, we see that the total average is approximately the same.

The third version was where we checkpointed every 100 iteration.

Extra modules loaded	Yes
Checkpointing	Yes
RAM disk	Yes
Normal disk	No
Objective	Show that the iterations in which the program is checkpointed take a consistent amount of time, and that even with an unreasonable high frequency of checkpointing, the overall average time does not increase that much

Table 4: The third version: The version where we checkpoint every 100th iteration

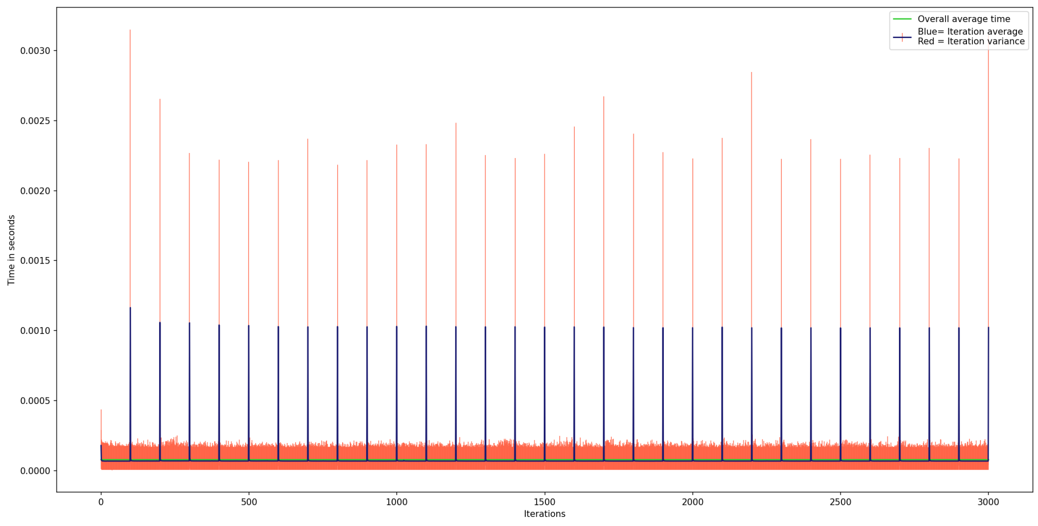

Figure 4: Benchmark with checkpoint every 100th iteration

The fourth version was where we checkpointed every 500 iteration.

Extra modules loaded	Yes
Checkpointing	Yes
RAM disk	Yes
Normal disk	No
Objective	Show that the chekcpointing iterations of the program are still consistent, and that the average mean time decreases compared to checkpoints every 100th iteration

Table 5: The fourth version: The version where we checkpoint every 500th iteration

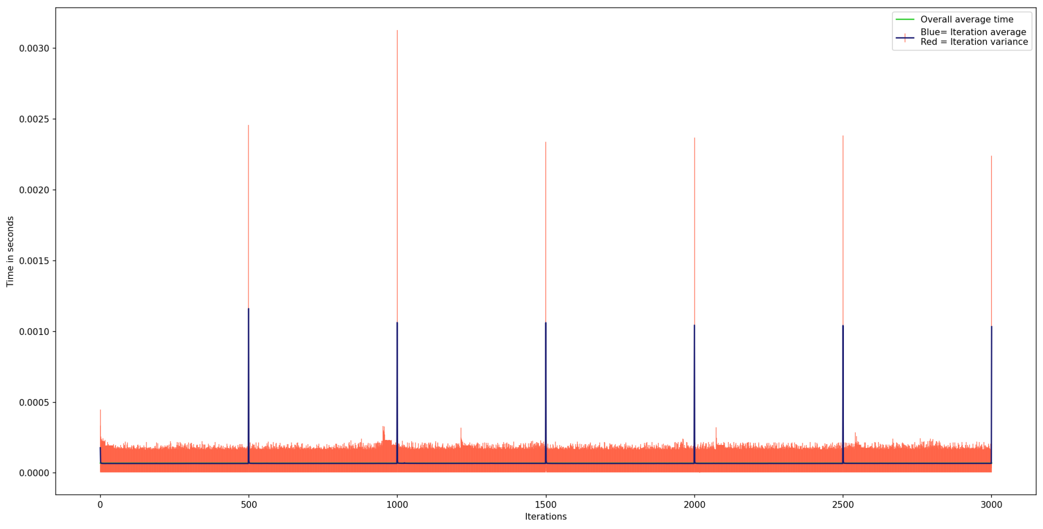

Figure 5: Benchmark with checkpoint every 500th iteration

Comparing the two different checkpointing graphs, it can be seen that there is no significant difference between the average time used to create a checkpoint. It can also be seen that the first checkpoint takes slightly longer, which is consistent across all executions, no matter the frequency of checkpoints. This does not matter in a practical setting though, as checkpointed programs would likely run for so long that the first checkpoint would have almost no impact on the average.

The error bar of each iteration, seen in the red error bars, is neither consistent across checkpointing iterations nor normal iterations. This is likely due to the operating system switching context or allocating resources to a different process during one of the 2000 executions. The important part is that it does not noticeably affect the average of all the iterations.

The error bars seem to create a solid line at the bottom, a certain lower bound if you will. From the graphs above, this seemingly also happens below the checkpointing iterations, but upon increasing the resolution of the graph and zooming in, it can be seen that this does not in fact, happen. In the checkpointing iterations the lower bound is not as even as it seems. In Figure 6 below is a zoomed in picture of the 4 graph illustrating that this lower bound does in fact change in iterations in which a checkpoint is made.

Worth mentioning, however, is that the best-case of each iteration is still very consistent, so consistent, that even if the bars were seperated and zoomed in on, they almost form a consistent lower bound, which is likely the actual running time when the operating system does not interrupt the process.

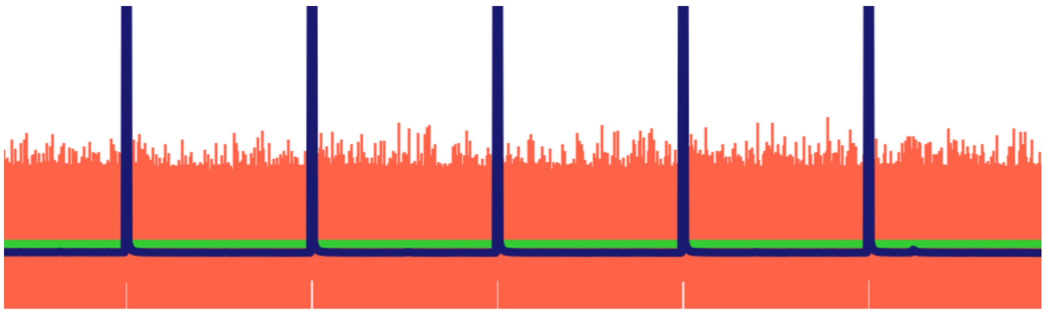

Figure 6: Zoomed in on lower bound at checkpoints

In all of the graphs above, we see that the first few iterations spike quite a lot compared to the rest of the iterations. Every time the `bash` script has finished an execution, it frees the memory, and thus has to be reallocated at the beginning of the next execution. Using a `python` script instead of a `bash` script to perform the benchmarking executions would likely normalize this part of every graph.

4.2.3 Performance Difference

From the graphs above, it can be seen that the iterations in which the program is checkpointed, take considerably longer than the rest of the iterations. They do, however, not take an absurd amount of time, and due to the checkpoint being written to a RAM disk instead of an actual disk, a minimal amount of this added time is spent waiting for I/O limitations. Thus, the result seen is as close to straight program performance difference as possible.
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 resulting numbers:

• With checkpointing:	0.001002280 seconds
• Without checkpointing:	0.000066738 seconds

Comparing these show that the checkpointing iterations take 15.01 times longer, or 1501% longer to run. While this is a lot, it is very possible that programs which take longer to perform an iteration would see a smaller percentage wise cost of checkpointing, as it is possible that a lot of the time used to checkpoint is used to initialize the process of checkpointing. Furthermore, an increase in the time between checkpoints would reduce the overall runtime of the program significantly.

4.3 Analyzing for optimizations

To see the statistics that describe how often and how long various parts of the programs are executed, we have used `python`'s own profiler module `cProfile` [18].

We have used the profiler in order to be able to see which function calls in the `dill` module that took an unusually long amount of time, in order to see which parts of the program that made the most sense to try optimizing. The `cProfile` command allowed to see some statistic of the heat equation program, where we checkpoint every 100th iteration for 50.000 iterations.

The following is a snippet of the output, by running the following command in the terminal: `p3 -m cProfile -s cumtime benchpress.py`

```
7139308 function calls (6692348 primitive calls) in 6.741 sec

Ordered by: cumulative time

ncalls  tottime percall cumtime percall filename:lineno(func)
570/1     0.003   0.000   6.742   6.742 {built-in method ...
    1     0.000   0.000   6.742   6.742 ramdisk.py:1(<module>)
    1     0.000   0.000   6.512   6.512 ramdisk.py:64(main)
    1     0.568   0.568   6.512   6.512 ramdisk.py:28(jacobi)
50000     2.950   0.000   5.209   0.000 ramdisk.py:31(loop_body)
  500     0.003   0.000   1.708   0.003 _dill.py:334(dump_session)
  500     0.001   0.000   1.695   0.003 _dill.py:414(dump)
  500     0.001   0.000   1.690   0.003 pickle.py:474(dump)
342k/500 0.458   0.000   1.681   0.003 pickle.py:533(save)
1500/500 0.008   0.000   1.679   0.003 _dill.py:1258(save_module)
17k/500  0.034   0.000   1.662   0.003 pickle.py:619(save_reduce)
15k/500  0.050   0.000   1.633   0.003 _dill.py:882(save_module_dict)
9000/500 0.011   0.000   1.627   0.003 pickle.py:962(save_dict)
      ...
```

As can be seen in the above extract from the profiler, `dill` almost does not add any overhead to `pickle`. The difference between the time spent in **dill.dump_session** and **pickle.dump** is only 1.01%, or 0.018 seconds.
It can be seen that the **jacobi** function call takes 6.512 seconds, but this is the function in which the timings are done and saved to a file, so this is not relevant to this analysis. What is notable, however, is that the **loop_body**, the function in which all the calculations are made and the checkpoints saved, takes 5.209 seconds to run. Of those 5.209 seconds, 1.708 seconds are spent creating and saving the checkpoints - as evident in the **_dill.py:334(dump_session)** line. While this

is a 48.786% overhead, it is not very significant, considering a checkpoint is made every 100th iteration of the **loop_body**, as a real world application would not need to checkpoint anywhere near as often.

Our initial purpose behind the profiler was to find areas in which `dill` was lacking, but the result was seeing that `dill` practically imports the `pickle` module but with added type support. This meant that in order to optimize the performance of the serialization process, we would have to compete with, and outperform, `python`'s own serialization and deserialization module.

4.4 Checkpointing to non-volatile memory

In the above analysis, our result were based on writing to volatile memory in the form of a RAM disk, in order to avoid inconsistencies due to I/O limitations. Though, as checkpointing only makes sense if you write to non-volatile memory in the form of a disk, we wish to figure out the source of the I/O blocking to determine if it is a general issue, or if it is solvable.

In an attempt to find the source of the I/O blocking, the following options were considered and put into motion one by one:

OS Problem	Rename checkpoint files instead of overwriting same file.
Hardware problem	Increase workload so disk has more time to 'recover' before the next write
Python solution	Convert library to use `python`'s `asyncio` [19] to allow the program to start working on the next loop iteration instead of waiting for the disk to finish writing

4.4.1 OS Problem

The problem
Manjaro's way of handling memory might somehow introduce inconsistency and blocking, upon writing to the same file name several times per second.

Solution attempt
Testing this was as simple as ensuring that each checkpoint file had a new, unique name. However, this was not the cause of the problem, and there was still the occasional spike in checkpointing due to I/O blocking. The graph can be seen in appendix [A.3]

4.4.2 Hardware problem

The problem

The SSD used in the test computer might be unable to keep up with the frequency and amount of writing done. This would likely be a problem with the SSD buffer.

Solution attempt

There are several ways to test whether this is the issue, ranging from acquiring a newer SSD with a higher writing speed and a bigger buffer, creating a raid 0 setup using multiple disks, or modifying the test to not be as write-intensive.

We decided to modify our test to be less write-intensive, as introducing new hardware to the testing environment would likely produce incomparable results in relation to previous tests.
Increasing the dimensions of the grid being iterated over in the heat equation [11] from 100×100 to 1000×1000, would increase the amount of computation in each iteration significantly, which would subsequently increase the time each iteration takes. Furthermore, we only checkpoint every 500 iterations. The objective of this, is that the disk has more time to 'recover' before each checkpoint needs to be written so it doesn't have to clear its buffer during the creation of a checkpoint.

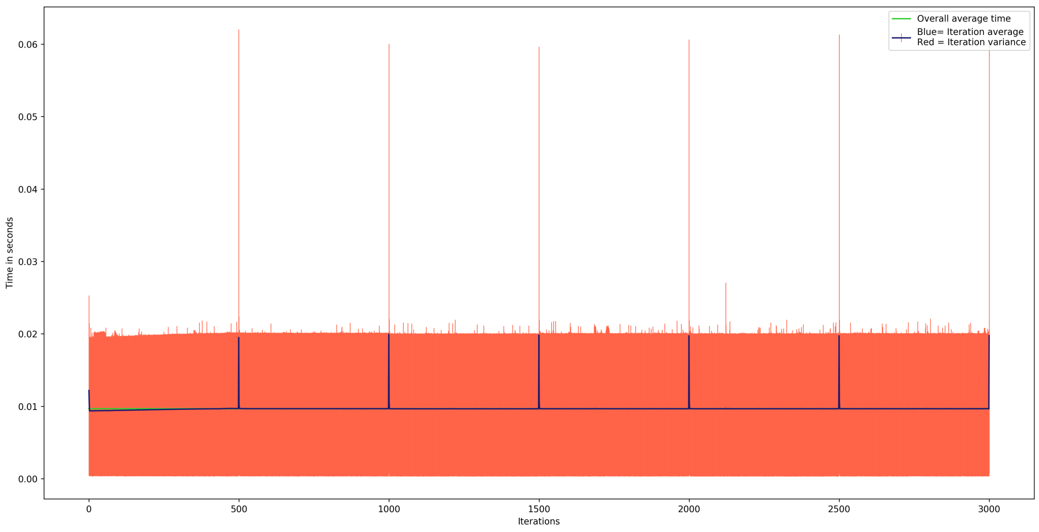

Figure 7: Heat equation using a 1000×1000 grid with checkpointing every 1000 iterations

The result was that the size of the checkpoint file increased from 83.6 KiB ($85,627$ bytes) to 7.7 MiB ($8,034,690$ bytes), and overall runtime of the program increased from around 0.32 seconds to 30 seconds. Something notable here is, that both the checkpointing size and total time to run each execution increased by almost exactly 2 orders of magnitude, as we increased the size of the grid by 2 orders of magnitude. This also completely solved the I/O blocking issue of saving checkpoints to disk, which means that it was likely an issue with the disk buffer being unable to keep up with the frequent checkpointing of the smaller scale tests.

This also suggests that as computational complexity and the time between each checkpoint increases, I/O blocking will become a non-issue. This is good, as computational complexity and checkpoint size is vastly bigger in large scale applications, which this checkpointing functinality is ultimately intended for. The efficiency of checkpointing will therefore be almost entirely dependent on the efficiency of the checkpointing module, and not external factors such as hardware.

4.4.3 Python solution

This solution did not get explored much, as it was a very extensive modification to the original `dill` library, and it was unnecessary, as the problem was solved with increased computational complexity.

4.5 Comparing checkpointing benchmarks

Selvom vi ikke har undersøgt disse moduler endnu antager vi at vi vil få lignende resultater og baserer vores delkonklusion derpå. Dette vil selvfølgelig ændre sig hvis det viser sig at performance er et område hvor dill modulet mangler.

4.5.1 Lattice Boltzmann

4.5.2 Shallow water

4.6 Areas of improvement

From the results presented throughout this section, it can be concluded that performance is not a fundamental issue of this module. It can also be concluded, however that ease of use and seamlessness are both areas in which a lot of improvements can be made and will therefore be the primary focus of the remainder of this report.

5 Testing

5.1 Benchpress - Heat equation

To make sure that `python`'s checkpointing module `dill` worked as intended, we wanted to test if the heat equation program yielded the same result whether we checkpointed or not. To do this, we started by simply running the program without any checkpoint commands, and saved the result of the execution of the program in a file - we called it `no_check.txt`. Afterwards we ran the program again, but this time we checkpointed and exit the program to start it up again from the checkpointed state. The result of this execution was saved to another file called `yes_check.txt`. We then compared the two files with the following bash command, and observed, that the result of the two different executions were identical:
`diff no_check.txt yes_check.txt` [28]

5.2 **`VEROS`**

Since we were not able to implement a checkpointing functionality, without having to extensively modify varying parts of `VEROS`, we were not able to test if the program would yield the same result with and without the checkpointing.

Part of the reason for why we were not able to implement it, has to do with local variables and the `C` call stack. We will get further into this in Section [8.2]

5.3 Lattice Boltzmann & Shallow water

Dette kommer til at ligne testing af Heat Equation næsten fuldstændig - igen antager vi at resultaterne vil være lignende dem vi har fået indtil videre.

6 Discussion

6.1 Current limitations

From the implementations - or attempt thereof - and subsequent analyses of checkpointing in the four programs, several limitations of the currently available checkpointing functionalities have been identified:

- All variables which need to persist across checkpoints need to be global in the scope of the **__main__** module. Furthermore, these variables need to be initialized outside of the main loop function or the function calling the looping function to prevent re-initializing the variables upon loading a checkpoint.

- Variables not needed for the actual result of the program but which are still needed in the loop logic - such as a counter in a for-loop - also need to be global such that the loop does not simply start over with the restored variables upon restoring a checkpoint. This was one of the issues faced when trying to make `VEROS` compatible with `dill` checkpointing.

6.2 Implementation options

The three main ways to get around these limitations are:

- Use a different interpreter than `cPython`. One like `stackless python` or `PyPy`, which do not rely on the `C` stack or have their own wrappers which provide more information at "`python`-level". These are both discussed in depth in Section 8.2.3.

- Use a checkpointing program for the `C`-code responsible for evaluating and interpreting the python bytecodes. This does have the downside of both removing the platform independence of `pickle` and also `C` is a lot more difficult to checkpoint due to the low-level nature of the language.
  Note that it might actually be possible to access, save, and rebuild the call stack using the **sys._getframe([depth])** [29] funtion introduced in `python 3.8`, but this is primarily for debugging and might therefore not work as expected in the case of rebuilding the call stack. There is also no `python` level API that allows access to the `C`-stack, which is necessary for rebuilding the saved call stack. This is discussed further in Section 8.2.

- Building the program around `dill` or `cloudpickle` such that variables are not re-initialized upon reloading a checkpoint and calling the main loop function. Furthermore, every variable which needs to be persistent across checkpoints needs to be top-level, also known as global.
  This is the option explored in this report, the least extensive, and the one discussed further in the following section.

6.3 Program implementation for checkpointing to be trivial

After looking at different programs and trying to implement a checkpointing functionality, where some were a success and others were not, we propose a structure for how a program should be implemented in the first place, to later be able to trivially implement checkpoints.

Dette afsnit færdiggøres in-depth efter Latize Boltzmann og Shallow water implementationerne.

6.4 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.

In the case of confidential code, checkpointing in this fashion is suboptimal, as a checkpoint does not only save the state of a program, but can also be used to recreate the entire source code of the program.

7 Conclusion

Dette skrives efter inklusion af Lattice Boltzmann og Shallow water checkpointing resultater.

8 Future Work

8.1 Optimization of `dill` performance

We have ealier in this report, in Section 2.3.1, mentioned the `python` module `pickle` written in `python`, but also the optimized version `cPickle`, which is written in `C`. The main difference between the two modules are, that `cPickle` is much faster than `pickle`. This is because the `cPickle` module is written entirely in `C`, and the `pickle` module is only used as a fail-safe in case of errors during pickling using the `cPickle` module. The downside of this is that the `cPickle` functions and methods can neither be extended nor customized, as opposed to `pickle`, where the classes are easily extended [20].

Equivalently to the optimization of `pickle`, some future work could include optimizing the `dill` module to be written in pure `C`, as it is currently written in `python`. This would yield a faster version of `dill`, and as `pickle` and cPickle, the normal dill module could attempt to import the C-version of dill, but fall back to normal dill again, in case of errors. In the best case, one could obtain the same performance improvement, as can be seen between pickle and cPickle. Furthermore, a much more extensive checkpointing functionality could be created using the python C-level API, as explained below.

8.2 Low level checkpointing

As mentioned in Section 6.2, a method of implementing complete checkpointing functionality would be to use a checkpointing solution developed for C programs, as the code responsible for interpreting the compiled python bytecode is written in C. In order to explain this further, a little knowledge of how python is interpreted is needed: Whenever a callable [30] object is called - i.e. a function, method, class, etc. - a PyFrame object is pushed onto the C stack. The struct for the PyFrame [22] object looks like this:

```
struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;   /* previous frame, or NULL */
    PyCodeObject *f_code;    /* code segment */
    PyObject *f_builtins;    /* builtin symbol table (PyDictObject) */
    PyObject *f_globals;     /* global symbol table (PyDictObject) */
    PyObject *f_locals;      /* local symbol table (any mapping) */
    PyObject **f_valuestack; /* points after the last local */
    ...
    int f_lineno;            /* Current line number */
    int f_iblock;            /* index in f_blockstack */
    char f_executing;        /* whether the frame is still executing */
    ...
};
```

There is a lot of info here which needs to be restored in order to resume execution at the exact point in which a checkpoint was made. Correctly saving the stack of frames such that they could be re-allocated and restored would allow a program to resume execution at the exact instruction following the call to checkpoint. This would also remove the need for variables to be defined in the global scope, and instead allow for local variables to be saved and allow any type of program structure to easily implement checkpointing.

Not only is this a very non-trivial task to create, none of the python-level API functions are low-level enough to access this information in its entirety, which means that the checkpointing functionality would have to be written entirely in C.

Note that we mention in Section 3 that we use an iteratively structured program for checkpointing. Should a low-level solution be implemented, this would not be a limitation either, and even recursive programs could be checkpointed due to the ability to save every function call and its state.

8.2.1 Compared to `dill`

A short comparison can be made to `dill` - and `cloudpickle` for that matter - as they are written entirely in `python`, which is likely why they only have the capability to save and restore top-level variable declarations and objects.

8.2.2 Generators, frames, and `pickle`

Neither `dill` nor `cloudpickle` can save **generator**, **frame**, or **traceback** objects. This is due to the above mentioned limitations of the `python`-level API in relation to accessing the low-level interpreter's handling of frames. As both **traceback** and **generator** type objects rely on a **frame** object to handle their logic, they are therefore seemingly out of scope of a checkpointing module written in `python` only. Even the creator of `dill` has commented on the need for `C` code in order to checkpoint these types [25]. Furthermore, several developers of the `python` language have expressed a disinterest to support 'pickling' (serializing) of **generator** type objects - and by proxy frame objects - due to security concerns [31]. Note that this was in late 2008, and their stance on it might have changed, but as the `pickle` module still does not support generator types, it is unlikely to have changed.

8.2.3 `PyPy` and `stackless python`

`stackless python` [32] is an alternative `Python` interpreter which focuses on optimizing and improving performance through the introduction of improved concurrency to `python` and removing reliance on the `C` call stack that `cPython` uses. The interesting part here is the improved concurrency, as this is done through something called **tasklets** [33]. These run concurrently within a single or a few threads and supposedly improve performance considerable. Not only do these improve performance, they are also fully serializable to the point of being able to resume at the exact instruction a checkpoint was made [34]. This is done by the developer of `stackless python` having extended the `pickle` module to be able to save **frame** type objects [35]. Note that this is an extension written entirely in `C`, and is several thousands of lines of code, so it is not trivial to port over to support the standard `python` interpreter `cPython`. This is made more difficult by the fact that `stackless python` uses a modified version of python, which includes a modified version of the most essential object, the **PyFrame** object [36].

PyPy [37], an arguably more popular third party interpreter for python includes objects similar to **tasklets**, called **greenlets** which are also savable using pickle, again due to extended support provided by the interpreter [38]. Note, however, that while the stackless python interpreter is rewritten entirely in C, PyPy is written entirely in python aswell as a limited version, called rPython - or restricted python.

Adding support for **frame** type objects also adds support for the two object types that neither dill nor cloudpickle currently support, namely **generator** and **traceback** type objects.

8.2.4 Additional downsides to C-level checkpointing

While testing VEROS we ran into an issue with dill not being able to save **abstract base classes**. Upon researching the issue, we discovered that this was because the implementation of **abstract base classes** was completely rewritten in C with python 3.7, which meant that we had to downgrade to python 3.6.10 in order to even attempt to checkpoint VEROS. This was due to dill not yet supporting the new API provided for interacting with **abstract base classes** [39]. Similiar issues will arise with new updates to the python language, but a C-level third party extension of the language will be significantly harder to update and fix than a python-level third party extension.

A Appendix

A.1 Cluster data

```
squeue | grep node | grep ' R [4-9][0-9]-'
jobid, group, program, user, status, time, cores, machine".
16519679 chem 02_dd_CN USER_PLACEHOLDER R 69-23:03:59 1 node218
16712435 kemi4 RPA_davi USER_PLACEHOLDER R 43-22:00:53 1 node041
16707533 kemi4 RPA_davi USER_PLACEHOLDER R 44-20:48:44 1 node049
16707528 kemi4 RPA_lanc USER_PLACEHOLDER R 44-20:49:26 1 node049
16683759 chem 28-TS-QS USER_PLACEHOLDER R 51-23:38:38 1 node225
16593467 chem MO-OR-SO USER_PLACEHOLDER R 53-20:10:22 1 node218
16593326 chem MO-OR-SO USER_PLACEHOLDER R 62-18:50:31 1 node212
16663389 cms-gpu bash USER_PLACEHOLDER R 56-22:07:59 1 node258
16698581 kemi6 2_coord_ USER_PLACEHOLDER R 47-23:40:59 1 node371
16678872 chem hrpd_sd_ USER_PLACEHOLDER R 40-19:10:00 1 node234
16725859 chem ho14 USER_PLACEHOLDER R 41-00:49:34 1 node214
16721689 chem thr_16_s USER_PLACEHOLDER R 41-07:50:45 1 node346
16721688 chem his_16_s USER_PLACEHOLDER R 41-08:10:09 1 node342
16721687 chem ile_16_s USER_PLACEHOLDER R 41-08:46:50 1 node338
16718454 chem 2yet_Rub USER_PLACEHOLDER R 41-09:52:35 1 node219
16718452 chem 2yet_Rub USER_PLACEHOLDER R 41-13:00:12 1 node227
16721686 chem val_16_s USER_PLACEHOLDER R 42-02:13:17 1 node338
16717577 chem met_24_s USER_PLACEHOLDER R 42-23:44:28 1 node343
16717370 chem line4a USER_PLACEHOLDER R 43-01:34:24 1 node347
16717371 chem line4b USER_PLACEHOLDER R 43-01:34:24 1 node235
16717316 chem gln_24_s USER_PLACEHOLDER R 43-02:12:26 1 node223
16717224 chem asn_16_s USER_PLACEHOLDER R 43-02:16:24 1 node234
16717228 chem glu_20_s USER_PLACEHOLDER R 43-02:16:24 1 node342
16678869 chem aosop_sd USER_PLACEHOLDER R 44-07:59:23 1 node221
16678868 chem aosop_no USER_PLACEHOLDER R 45-04:41:31 1 node227
16512458 chem rpd_sd_2 USER_PLACEHOLDER R 45-09:53:07 1 node232
16511746 chem tyr_92_f USER_PLACEHOLDER R 47-09:48:45 1 node215
16517327 chem cub05 USER_PLACEHOLDER R 68-01:05:16 1 node235
16517328 chem cub06 USER_PLACEHOLDER R 68-01:05:16 1 node335
15574869 kemi4 CrNacac2 USER_PLACEHOLDER R 72-16:43:52 1 node039
15574870 kemi4 VOacac2_ USER_PLACEHOLDER R 72-16:43:52 1 node047
16476224 chem MO-OR-HR USER_PLACEHOLDER R 72-20:39:08 1 node213
16513114 chem Cuacac2_ USER_PLACEHOLDER R 71-20:47:28 1 node214
16512896 chem Cuen2_B USER_PLACEHOLDER R 71-21:00:22 1 node218
16453875 chem VOacac2_ USER_PLACEHOLDER R 72-20:40:14 1 node212
15321413 chem CC_VDZ_T USER_PLACEHOLDER R 72-20:39:57 1 node212
16201201 kemi4 ile_64_s USER_PLACEHOLDER R 72-22:26:48 1 node033
16185780 kemi4 glu_80_s USER_PLACEHOLDER R 72-22:27:17 1 node040
16185781 kemi4 glu_80_s USER_PLACEHOLDER R 72-22:27:17 1 node043
16185813 kemi4 ser_64_s USER_PLACEHOLDER R 72-22:27:17 1 node047
16513252 kemi1 VOacac2_ USER_PLACEHOLDER R 71-20:34:00 1 node319
16513254 kemi1 Cuacac2_ USER_PLACEHOLDER R 71-20:34:00 1 node320
16513214 kemi1 VOacac2_ USER_PLACEHOLDER R 71-20:43:00 1 node312
```

```
16513219 kemi1 Cuacac2_ USER_PLACEHOLDER R 71-20:43:00 1 node314
16508506 chem ile_64_s USER_PLACEHOLDER R 72-11:44:15 1 node233
15717030 chem line5g USER_PLACEHOLDER R 72-18:36:59 1 node235
15719991 chem cub04 USER_PLACEHOLDER R 72-18:36:59 1 node215
16374728 chem ho12 USER_PLACEHOLDER R 72-18:36:59 1 node215
15716922 kemi1 CuCO3 USER_PLACEHOLDER R 72-19:56:05 1 node305
15716927 kemi1 FeCO5 USER_PLACEHOLDER R 72-19:56:05 1 node305
15717279 kemi1 MnNCN4 USER_PLACEHOLDER R 72-19:56:05 1 node305
15720787 kemi1 NiCO3H USER_PLACEHOLDER R 72-19:56:05 1 node306
16245997 kemi1 Cuacac2_ USER_PLACEHOLDER R 72-19:56:05 1 node305
16246007 kemi1 VOacac2_ USER_PLACEHOLDER R 72-19:56:05 1 node307
16246008 kemi1 VOacac2_ USER_PLACEHOLDER R 72-19:56:05 1 node307
16246015 kemi1 Cuacac2_ USER_PLACEHOLDER R 72-19:56:05 1 node308
16246018 kemi1 Cuacac2_ USER_PLACEHOLDER R 72-19:56:05 1 node308
16322772 kemi1 CrNacac2 USER_PLACEHOLDER R 72-19:56:05 1 node309
16322773 kemi1 CrNacac2 USER_PLACEHOLDER R 72-19:56:05 1 node309
16322774 kemi1 CrNacac2 USER_PLACEHOLDER R 72-19:56:05 1 node310
16358204 kemi1 CrNacac2 USER_PLACEHOLDER R 72-19:56:05 1 node311
16358205 kemi1 CrNacac2 USER_PLACEHOLDER R 72-19:56:05 1 node311
16261235 chem gln_92_s USER_PLACEHOLDER R 72-20:39:07 1 node230
15716727 chem line7d USER_PLACEHOLDER R 72-20:39:08 1 node217
15717029 chem line5f USER_PLACEHOLDER R 72-20:39:08 1 node218
16080756 chem line5i USER_PLACEHOLDER R 72-20:39:08 1 node218
16099835 chem aosop_sd USER_PLACEHOLDER R 72-20:39:08 1 node213
16185844 chem cys_64_s USER_PLACEHOLDER R 72-20:39:08 1 node214
16192390 chem thr_64_s USER_PLACEHOLDER R 72-20:39:08 1 node234
16216179 chem met_92_s USER_PLACEHOLDER R 72-20:39:08 1 node235
16219156 chem asp_64_s USER_PLACEHOLDER R 72-20:39:08 1 node227
16219484 chem lys_92_s USER_PLACEHOLDER R 72-20:39:08 1 node228
16219485 chem lys_92_s USER_PLACEHOLDER R 72-20:39:08 1 node229
16216019 chem pro_64_s USER_PLACEHOLDER R 72-20:40:09 1 node212
16219137 chem tyr_92_s USER_PLACEHOLDER R 72-20:40:09 1 node217
16219155 chem asp_64_s USER_PLACEHOLDER R 72-20:40:09 1 node220
16252185 chem hrpd_sd_ USER_PLACEHOLDER R 72-20:40:09 1 node220
16365526 chem his_48_f USER_PLACEHOLDER R 72-20:40:09 1 node217
16365528 chem his_48_s USER_PLACEHOLDER R 72-20:40:09 1 node220
16348368 kemi6 TS_H5_op USER_PLACEHOLDER R 83-02:22:55 1 node373
16219136 chem tyr_92_s USER_PLACEHOLDER R 96-06:28:23 1 node335


squeue | grep node | grep ' R [1-9][0-9][0-9]-'
16136279 kemi6 H1_band- USER_PLACEHOLDER R 109-21:41:25 1 node360
16136284 kemi6 H5_min_b USER_PLACEHOLDER R 109-21:41:25 1 node362
16136285 kemi6 H6_min_b USER_PLACEHOLDER R 109-21:41:25 1 node371
16136287 kemi6 C4_min_b USER_PLACEHOLDER R 109-21:41:25 1 node370
15785905 kemi6 Red_pc-1 USER_PLACEHOLDER R 159-20:17:05 1 node372
15716726 chem line7c USER_PLACEHOLDER R 176-15:41:55 1 node343
15351706 chem c4-cc3-A USER_PLACEHOLDER R 230-07:41:03 1 node344
```

A.2 Server uptimes

```
1 server(s)   have been running for 1 day
2 server(s)   have been running for 3 days
2 server(s)   have been running for 6 days
3 server(s)   have been running for 7 days
20 server(s)  have been running for 9 days
2 server(s)   have been running for 10 days
1 server(s)   have been running for 12 days
1 server(s)   have been running for 13 days
42 server(s)  have been running for 14 days
2 server(s)   have been running for 15 days
79 server(s)  have been running for 16 days
24 server(s)  have been running for 17 days
4 server(s)   have been running for 21 days
1 server(s)   have been running for 22 days
3 server(s)   have been running for 30 days
1 server(s)   have been running for 33 days
1 server(s)   have been running for 37 days
1 server(s)   have been running for 44 days
1 server(s)   have been running for 48 days
1 server(s)   have been running for 56 days
1 server(s)   have been running for 60 days
1 server(s)   have been running for 62 days
1 server(s)   have been running for 63 days
1 server(s)   have been running for 65 days
1 server(s)   have been running for 66 days
1 server(s)   have been running for 71 days
205 server(s) have been running for 72 days
2 server(s)   have been running for 77 days
1 server(s)   have been running for 82 days
6 server(s)   have been running for 96 days
3 server(s)   have been running for 97 days
9 server(s)   have been running for 103 days
1 server(s)   have been running for 104 days
1 server(s)   have been running for 106 days
2 server(s)   have been running for 113 days
12 server(s)  have been running for 121 days
5 server(s)   have been running for 131 days
6 server(s)   have been running for 132 days
17 server(s)  have been running for 133 days
27 server(s)  have been running for 134 days
1 server(s)   have been running for 142 days
1 server(s)   have been running for 145 days
4 server(s)   have been running for 162 days
7 server(s)   have been running for 163 days
1 server(s)   have been running for 168 days
9 server(s)   have been running for 184 days
1 server(s)   have been running for 199 days
```

```
1 server(s)   have been running for 224 days
1 server(s)   have been running for 226 days
1 server(s)   have been running for 251 days
1 server(s)   have been running for 264 days
1 server(s)   have been running for 266 days
1 server(s)   have been running for 268 days
34 server(s)  have been running for 270 days
1 server(s)   have been running for 279 days
2 server(s)   have been running for 280 days
18 server(s)  have been running for 281 days
42 server(s)  have been running for 282 days
10 server(s)  have been running for 284 days
```

A.3 Saving to disk using unique file names

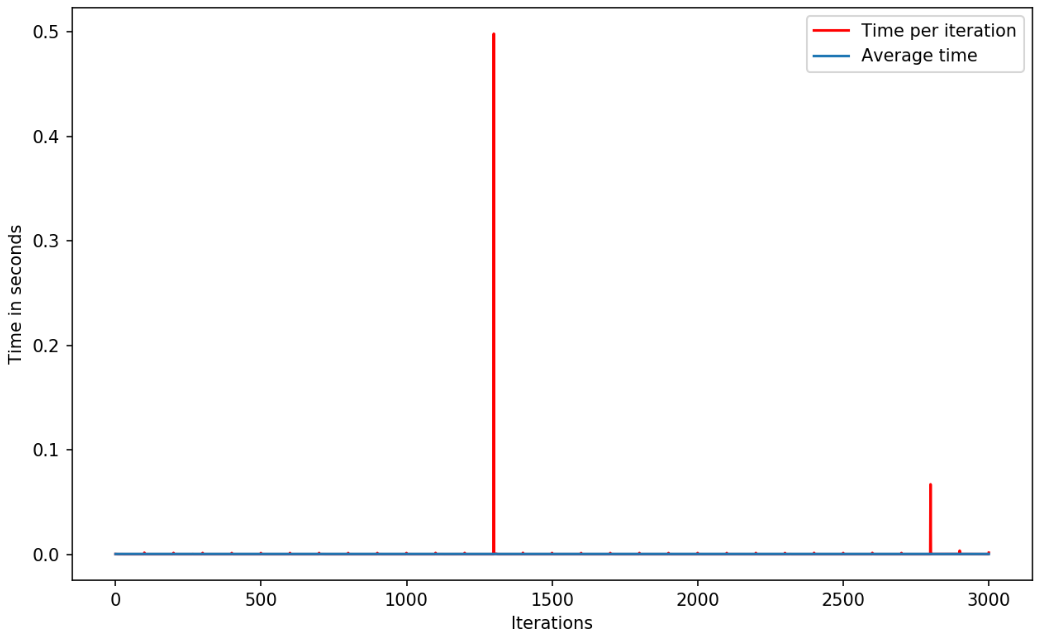

Figure 8: I/O blocking when saving checkpoints to disk using unique file names for each checkpoint

References

[1] May 8, 2020. URL: https://hostingfacts.com/average-hosting-uptime-study/.

[2] May 7, 2020. URL: http://iwgcr.org/wp-content/uploads/2014/03/downtime-statistics-current-1.3.pdf.

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

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

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

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

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

[8] April 6, 2020. URL: https://readthedocs.org/projects/dill/downloads/pdf/latest/.

[9] May 22, 2020. URL: https://github.com/cloudpipe/cloudpickle.

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

[11] April 24, 2020. URL: https://en.wikipedia.org/wiki/Heat_equation.

[12] April 24, 2020. URL: https://en.wikipedia.org/wiki/Jacobi_method.

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

[14] April 24, 2020. URL: https://veros.readthedocs.io/en/latest/.

[15] May 16, 2020. URL: https://veros.readthedocs.io/en/latest/quickstart/get-started.html#re-starting-from-a-previous-run.

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

[17] April 24, 2020. URL: https://www.linuxbabe.com/command-line/create-ramdisk-linux?fbclid=IwAR3gNUixmXLCOAIZ_DeMRRBm56dIIceh_VkKQ8Z09YrurKA5GbYOK9YLApo.

[18] April 24, 2020. URL: https://docs.python.org/3/library/profile.html.

[19] May 2, 2020. URL: https://docs.python.org/3.6/library/asyncio.html.

[20] May 7, 2020. URL: https://www.techrepublic.com/article/get-yourself-into-a-python-cpickle/.

[21] URL: https://en.wikipedia.org/wiki/Program_counter.

[22] . URL: https://github.com/python/cpython/blob/master/Include/cpython/frameobject.h.

[23] . URL: https://docs.python.org/3/library/dis.html#bytecode-analysis.

[24] . URL: https://docs.python.org/3/whatsnew/3.8.html#cpython-bytecode-changes.

[25] . URL: https://github.com/uqfoundation/dill/issues/10.

[26] . URL: https://dill.readthedocs.io/en/latest/dill.html#dill._dill.load_session.

[27] . URL: https://docs.python.org/3/library/__main__.html.

[28] . URL: https://ss64.com/bash/diff.html.

[29] . URL: https://docs.python.org/3/library/sys.html#sys._getframe.

[30] . URL: https://docs.python.org/3/reference/datamodel.html#object.__call__.

[31] . URL: https://bugs.python.org/issue1092962.

[32] . URL: https://stackless.readthedocs.io/en/3.7-slp/stackless-python.html.

[33] . URL: https://stackless.readthedocs.io/en/3.7-slp/library/stackless/tasklets.html.

[34] . URL: https://stackless.readthedocs.io/en/3.7-slp/library/stackless/pickling.html.

[35] . URL: https://github.com/stackless-dev/stackless/tree/master-slp/Stackless/pickling.

[36] . URL: https://github.com/stackless-dev/stackless/wiki/Portable-usage-of-PyFrameObject.

[37] . URL: https://doc.pypy.org/en/latest/index.html.

[38] . URL: https://bitbucket.org/pypy/pypy/src/default/pypy/module/_pickle_support/.

[39] . URL: https://github.com/cloudpipe/cloudpickle/pull/189.