Simanfor Parallel Implementation

1 Overview

The original plans were to use Apache Dask (ref: https://docs.dask.org/en/latest/) but they have been abandoned. The simulator has proven to be embarrassingly parallel. This is explained in the next section. However, parts of the code which use Apache Dask should be left in the code for:

a) demonstrative purposes
b) to serve as a guide for implementing Dask in the simulator if the need arises: if it is required to process plots with thousands of trees, in which case the simulator will most likely cease to be embarrassingly parallel.

In Section 4 we discuss Dask's use and potential use in Simanfor.

2 The simulator is embarrassingly parallel

An algorithm or section of code, or, as is the case in Simanfor, a whole program is considered embarrassingly parallel (ref: https://en.wikipedia.org/wiki/Embarrassingly_parallel) when the code can be executed on various independent cores which don't communicate with each other, without any loss of information. Typical examples of embarrassingly parallel problems are rendering of computer graphics (where each pixel can be rendered independently of all others), discrete Fourier transforms, or multiple independent text searches.

In Simanfor's case, the input is a forest inventory, in the form of an Excel file with two sheets:

- the first one is a list of plots and their attributes, with one plot per row, and
- the second sheet is a list of trees and their attributes, with one tree per row, linked to a plot via a plot ID, which corresponds to a row in the first sheet.

The output is one Excel file per plot, which contains the results of executing the simulator on various scenarios. The key here is that there is exactly one output file per plot, with no interaction between plots. Thus, the program could conceivably execute on a separate core for each plot, and would produce the desired output. In practice, as is discussed below, it is more efficient to run the sequential program on more than one plots per core. Somewhere between 16 and 32 seems to be very efficient, if not optimal.

Thus, neither Apache Dask nor any other parallel framework or library is needed. Instead, splitting the data input into smaller chunks of input (if and when needed), and submitting separate jobs for these chunks on the supercomputer, appears to be the ideal solution. This is the approach taken in this case. Next we describe how to install and run Simanfor on caléndula, and discuss performance issues and test runs.

3 The simulator on caléndula

The simulator code is on github: https://github.com/simanfor-dask/simulator

All that is needed to install on caléndula is to clone or download the code, make sure Python 3 is loaded, and that all the dependencies in **`.../simulator/requirements.txt`** are also installed.

The script **`simanfor.sh`** submits four jobs, on four different input files (inventories) that were created by manually splitting an example provided by UVA with 100 plots with 100 trees each. It simply calls the four scripts that individually submit the jobs:

```
CALENDULA[ sngular_aia_1_3@frontend2 scripts]$ cat simanfor.sh
#!/bin/bash

sbatch basic_engine_n4_1-25.sh
sbatch basic_engine_n4_26-50.sh
sbatch basic_engine_n4_51-75.sh
sbatch basic_engine_n4_76-100.sh

CALENDULA[ sngular_aia_1_3@frontend2 scripts]$
```

The Python code requires a scenario configuration file, where the input in the form of an inventory file described above is specified, together with the output directory, the models being used, forestry scenarios such as cuts and (tree growth???), and other configuration options.

Each script defines the Slurm directives, loads Python 3.7, and executes the python code, passing two arguments: the scenario configuration file and a logging configuration file path, e.g.:

```
CALENDULA[ sngular_aia_1_3@frontend2 scripts]$ cat basic_engine_n4_1-25.sh
#!/bin/bash

# numero de cores que serán reservados
#SBATCH -n 4
# particion en donde se ejecutara el trabajo
#SBATCH -p haswell
# limites que se aplicaran al trabajo
#SBATCH -q normal
# nombre
#SBATCH -J python_sngular
# tiempo maximo de ejecucion (p.e. 2 dias). Maximo permitido: 5 dias
#SBATCH --time=01:00:00
# archivos de salida y de error
#SBATCH -o ./salida/basic_n4-%j.o
#SBATCH -e ./salida/basic_n4-%j.e
# directorio de trabajo por defecto
#SBATCH -D .
# notificaciones por email relacionadas con la ejecucion del trabajo
#SBATCH --mail-user=spiros.michalakopoulos@sngular.com
##SBATCH --mail-type=ALL

ROOT=/home/sngular_aia_1/sngular_aia_1_3/dev/simulator

# carga de las variables necesarias para usar Python 3.7.7
module load python_3.7.7

python $ROOT/src/main.py -s $ROOT/files/scenario_claras_1-25.json -
logging_config_file $ROOT/config_files/logging.conf

CALENDULA[ sngular_aia_1_3@frontend2 scripts]$
```

4 Performance

Tests have been carried out on inventories of various sizes, and on various scenarios. Here we show the performance on the above mentioned inventory of 100 plots by 100 trees, split into 4 smaller input files, each with 25 plots by 100 trees.

First the jobs are submitted via the **`simanfor.sh`** script:

```
CALENDULA[ sngular_aia_1_3@frontend2 scripts]$ ./simanfor.sh
Submitted batch job 350695
Submitted batch job 350696
Submitted batch job 350697
Submitted batch job 350698
CALENDULA[ sngular_aia_1_3@frontend2 scripts]$
```

While they are running, we check the queue:

```
CALENDULA[ sngular_aia_1_3@frontend2 scripts]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            350695   haswell python_s sngular_  R       1:13      1 cn3034
            350696   haswell python_s sngular_  R       1:13      1 cn3034
            350697   haswell python_s sngular_  R       1:13      1 cn3034
            350698   haswell python_s sngular_  R       1:13      1 cn3034
CALENDULA[ sngular_aia_1_3@frontend2 scripts]$
```

Script output after execution:

```
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ ls -lrt
total 24
drwxr-xr-x 2 sngular_aia_1_3 sngular_aia_1 4096 May 14 18:32 errors_old
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1    0 May 19 13:37 basic_n4-350698.e
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1    0 May 19 13:37 basic_n4-350697.e
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1    0 May 19 13:37 basic_n4-350696.e
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1    0 May 19 13:37 basic_n4-350695.e
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 1727 May 19 13:39 basic_n4-350697.o
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 1726 May 19 13:39 basic_n4-350698.o
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 1727 May 19 13:39 basic_n4-350695.o
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 1727 May 19 13:39 basic_n4-350696.o
drwxr-xr-x 2 sngular_aia_1_3 sngular_aia_1 4096 May 19 13:41 old
CALENDULA[ sngular_aia_1_3@frontend2 salida]$
```

Timings of each script:

```
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ tail -2 basic_n4-350695.o
Models executions finished after 43.656333446502686 seconds.
Program finished after 67.93810319900513 seconds.
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ tail -2 basic_n4-350696.o
Models executions finished after 47.366323709487915 seconds.
Program finished after 73.33272171020508 seconds.
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ tail -2 basic_n4-350697.o
Models executions finished after 39.95999836921692 seconds.
Program finished after 60.887235164642334 seconds.
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ tail -2 basic_n4-350698.o
Models executions finished after 42.91468381881714 seconds.
Program finished after 66.82035899162292 seconds.
CALENDULA[ sngular_aia_1_3@frontend2 salida]$
```

Note that the time between the end of the models executions and the program finishing is the generation and printing of the output Excel files.

Those times don’t include scheduler and queue tasks. Here are the full times of each job:

```
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ sacct -j 350695,350696,350697,350698 --
format='JobID,JobName,Elapsed,State'
       JobID    JobName    Elapsed      State
------------ ---------- ---------- ----------
350695       python_sn+   00:02:03  COMPLETED
350695.batch      batch   00:02:03  COMPLETED
350695.exte+     extern   00:02:04  COMPLETED
350696       python_sn+   00:02:08  COMPLETED
```

```
350696.batch      batch   00:02:08  COMPLETED
350696.exte+     extern   00:02:08  COMPLETED
350697       python_sn+   00:01:56  COMPLETED
350697.batch      batch   00:01:56  COMPLETED
350697.exte+     extern   00:01:56  COMPLETED
350698       python_sn+   00:02:02  COMPLETED
350698.batch      batch   00:02:02  COMPLETED
350698.exte+     extern   00:02:02  COMPLETED
CALENDULA[ sngular_aia_1_3@frontend2 salida]$
```

Finally, here is a part of the output directory after execution:

```
CALENDULA[ sngular_aia_1_3@frontend2 salida]$ ll
/scratch/sngular_aia_1/sngular_aia_1_3/output_claras/
total 28572
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 328318 May 19 13:38 Output_Plot_1.xlsx
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 329631 May 19 13:38 Output_Plot_10.xlsx
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1  98246 May 19 13:39 Output_Plot_100.xlsx
…
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 374410 May 19 13:39 Output_Plot_97.xlsx
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1 377562 May 19 13:39 Output_Plot_98.xlsx
-rw-r--r-- 1 sngular_aia_1_3 sngular_aia_1  98145 May 19 13:39 Output_Plot_99.xlsx
CALENDULA[ sngular_aia_1_3@frontend2 salida]$
```

Note that the small output file size for some plots is not an error in the execution but due to the trees in the plots not fulfilling certain criteria which permits them to be included in the simulation. This results in empty sheets in the output file, sheets where the tree details are normally printed in the cases where the trees in the plot are taken into account for the simulation.

Thus we see that the above 100 plot by 100 trees simulation can be executed in just over 2 minutes on caléndula, without any sophisticated parallelization beyond splitting the input file into smaller input files. It should also be noted that in our tests on the full 100 by 100 input file, the sequential code produces the correct results in just over 4 minutes.

5 Website to submit the jobs

In the above examples, the input file and scripts used were generated manually. It is suggested that if a web site is developed to allow users to perform simulations on their data, that scripts are written that will behind the scenes:

- check the inventory input file and if it contains above a certain numbers of plots, split the file accordingly. This can very easily be done using the Python library Pandas.
- If the inventory has been split into multiple input files, then the same number of jobs should be submitted. This is achieved by generating multiple scenario configuration files and submitting jobs using the same code, but the different scenario configuration files.

6 Apache Dask

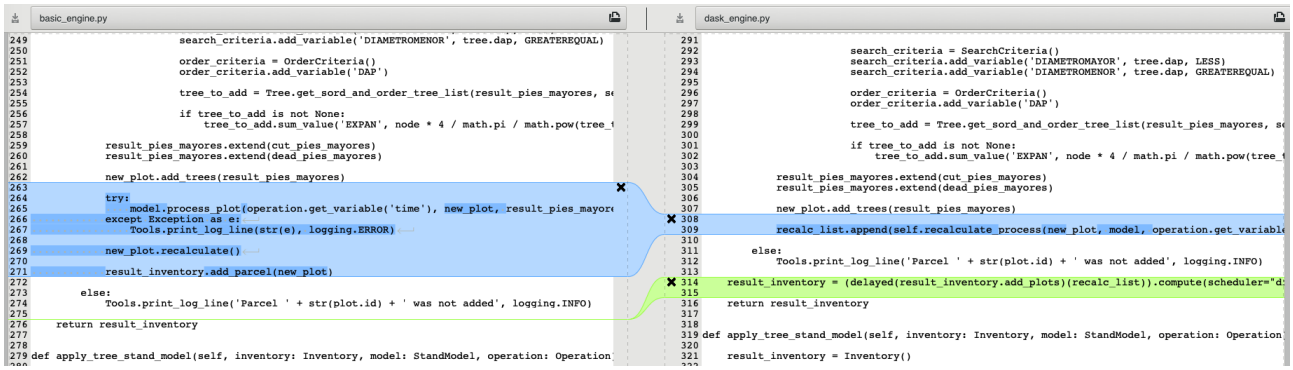
As we mentioned in the beginning of this document Apache Dask was planned to be used, but was considered surplus to requirements. It was also stated though that in the future, if the requirements for simulating tree growth on plots with a much larger number of trees is required, Dask could be a good option for speeding up the simulators execution. For that reason and because some interesting work has already been done in the code using Dask, we present this work here.

Dask has various ways of executing parallel code (ref: https://github.com/dask/dask-tutorial), including Numpy-like Dask Arrays and Pandas-like Dask DataFrames. In this work we used Dask Delayed (ref: https://docs.dask.org/en/latest/delayed.html).

To execute the simulator with the Dask parallel version, you pass in the "**`-e 2`**" flag. This evokes the **`dask_engine.py`** code instead of the sequential **`basic_engine.py`** code. In **`dask_engine.py`** you need to import delayed:

```
from dask import delayed
```

A side by side comparison between **`basic_engine.py`** and **`dask_engine.py`** illustrates the differences:



This code is executed for each plot in the inventory. In **`basic_engine.py`**, on the left hand side, the **`model.process_plot()`** function is called (passing in **`new_plot`** as a parameter), which performs calculations on some of **`new_plot`**'s variables, followed by **`new_plot.recalculate()`**, which performs further calculations on **`new_plot`**'s variables. Finally, the modified **`new_plot`** is added to the list of plots in **`result_inventory`**.

In the case of **`dask_engine.py`**, on the right, the function **`recalculate_process()`** is added to a list called **`recalc_list`**. This is what **`recalculate_process()`** looks like:

```
    def recalculate_process(self, plot: Parcel, model: TreeModel, time, trees: list, result_inventory: Inventory):

        plot = delayed(model.process_plot)(time, plot, trees)
        plot = delayed(plot.recalculate)()

        return plot
```

What this does is take the functions **`model.process_plot()`** and **`recalculate()`** and "delays" them. This tells the Dask kernel that you don't want to execute these functions right now, but just associate them with the specific plot. So, the code, loops through all the plots and places their delayed functions **`model.process_plot()`** and **`recalculate()`** into the list called **`recalc_list`**.

Then, after all the plots are processed, **`basic_engine.py`** simply returns the **`result_inventory`** with all the plots and their modified values, whereas **`dask_engine.py`** does this:

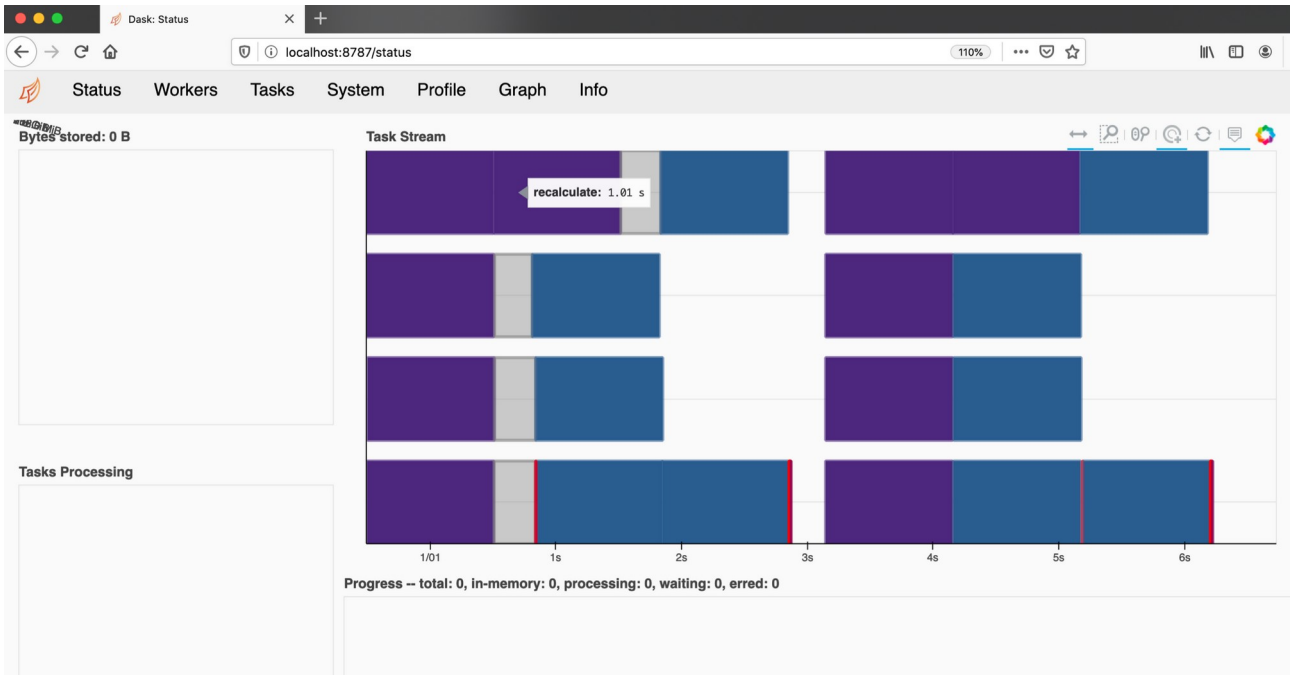
```
        result_inventory = (delayed(result_inventory.add_plots)(recalc_list)).compute(scheduler="distributed")

        return result_inventory
```

This code, tells the dask kernel to now go and `compute()` what is on the `recalc_list`, in parallel (using dask's "`distributed`" scheduler – there are other types of schedulers such as "`single-threaded`" which is very useful for debugging, see ???).

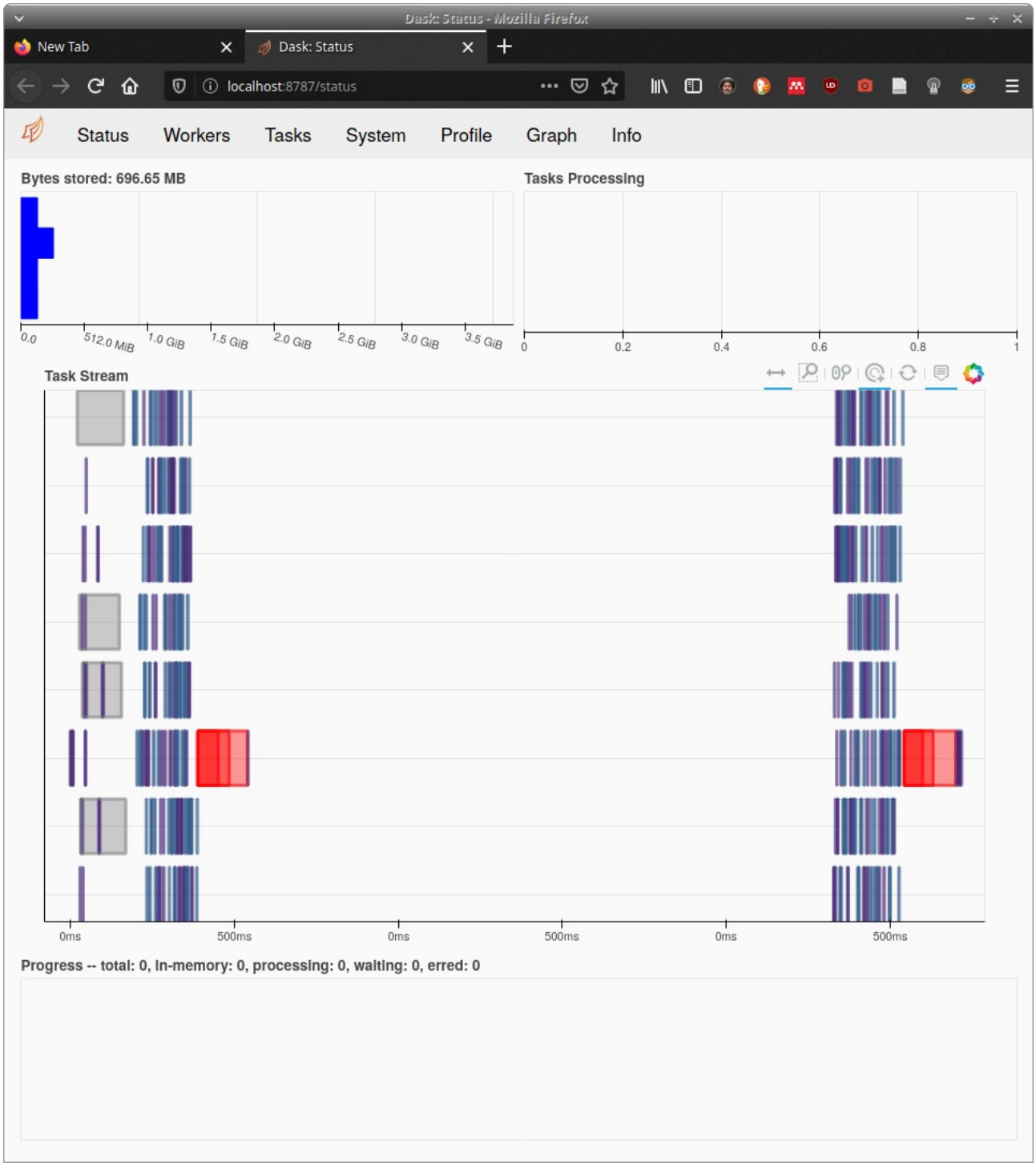
Dask comes with a useful diagnostics and visualization tool for the "`distributed`" scheduler, that can be seen at `localhost:8787`

In the following screen shot, the code was run on an example inventory with 5 plots, on a machine with 4 cores:



Note however that `sleep(1)` statements had been placed in the `recalculate()` and `process_plot()` functions, to be able to view the parallel execution better.

Here is another screenshot, this time without the sleep statements, on an 8 core machine, executing the 100 plot by 100 trees inventory:



Parallelizing those two functions however didn't show any increase in performance. On the contrary, in most cases, the execution times were slower, because of the overhead of dask. It showed though, that the vast majority of time consumed was in generating the output file. So, the output file generation code was also parallelized with **`dask.delayed`**. On small inventories, this showed a very slight (seconds or microseconds) improvement on runtimes, but on larger inventories resulted in memory problems.[1]

1 Because Python doesn't allow dynamic allocation and deallocation of memory, it wasn't clear to me (Spiros), how to solve this problem, and since the tests on the supercomputer and even my local machines showed that the splitting of the inventory into smaller ones was a better solution, I didn't investigate any further and have left the code commented out.