Multiprocessing implementation for MCNP using Python

Muhammad Arif Sazali\textsuperscript{1}, Muhammad Syahir Sarkawi\textsuperscript{1}, Nur Syazwani Mohd Ali\textsuperscript{1}

\textsuperscript{1} School of Chemical and Energy Engineering, Faculty of Engineering, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia

Abstract. Monte Carlo N-Particle (MCNP) is a widely-used code in nuclear engineering, but it needs high computation times due to the tracking of every single particle and interaction event. This causes shielding optimization using trial-and-error to take long times to complete. It can be solved by using multiprocessing, but this requires the MCNP source code which can be difficult to obtain. Therefore, this paper aims to suggest a solution on how Python can be used to run multiple instances of the code in shielding optimization. Two hardware setups were tested: one with a dual-core CPU, and one with a six-core CPU. Each of them would run several repeated simulations with and without multiprocessing enabled. Comparisons were made between each case of the same setup to observe the improvements in completion time. A tutorial of the Python algorithm is also provided in the methodology.

1. Introduction

Monte Carlo N-Particle (MCNP) is a nuclear simulation code developed and maintained by Los Alamos National Laboratory \cite{1}. It is one of the nuclear modeling software that utilizes the Monte Carlo method. Other examples include Geant \cite{2}, Fluka \cite{3}, OpenMC \cite{4}, and SuperMC \cite{5}. These codes provide very accurate results for complex situations and geometries. However, they are known to be computationally expensive, needing more calculation and time as the complexity increases. This is because the software needs to track every particle from its birth to its death or escape from the simulation space. Several works had been done to improve the runtimes of Monte Carlo simulation for radiation transport. Rydén et al. (2018) \cite{6} utilized four graphics processing units (GPUs) to run an MC code, simulating 3.2 billion photons/s for single-photon emission computed tomography (SPECT) reconstruction. Liang et al. (2020) \cite{7} argued that there are some limitations to using GPUs in Monte Carlo simulations, thus they set out to test the implementation of inverse transform sampling technique on CPU, hoping that it could be used to solve the GPU limitations. Nevertheless, multiprocessing is also available natively for some versions of MCNP. However, they may not be easily obtained compared to the serial version. There is a trend of using MCNP in metaheuristic algorithm optimization which demands the improvement in MCNP runtimes. This is because in these algorithms, the MCNP will be executed multiple times until the design variables were optimized.

One of the earliest studies was by Hu et al. (2008) \cite{8} where they coupled the MCNP code to a genetic algorithm (GA) metaheuristic to optimize the design of a composite shield in a mixed radiation situation. The result was five compositions that showed comparable attenuation with lead oxide. Cai et al. (2018) \cite{9} used the same method to optimize a multilayer shield for uranium-235 radiation. The result was six arrangements of material that proved the reliability and consistency of the metaheuristic in shield optimization. Chen et al. (2019) \cite{10} had also implemented the same strategy in designing the shielding of a nuclear reactor in a passenger-cargo ship from a set of 15 materials. The same application was
investigated by Wilson (2019) [11] but it was for bismuth-borate glasses. They were found to be comparable to limonite concrete and barite concrete for mixed radiation conditions. In 2020, two studies were conducted: one had developed six shielding designs from polyethylene, lead, and boron [12], while the other produced a 16-layer shield for a mobile nuclear reactor system from epoxy resin, lead, boron carbide, and graphene [13].

However, most studies do not reveal the method of how the repetition or multiple instances of MCNP was done. Therefore, this article aims to demonstrate a way of coupling the MCNP code with a Python-based algorithm and to investigate the improvement that can be expected when utilizing Python multiprocessing. Two different computer setups are used to observe the increase in performance if the physical CPU cores are increased. A tutorial for the Python algorithm can be found in Section 2 and Appendices.

2. Methods and materials

2.1 Hardware setups

Two Windows 10 hardware machines are used to run the Python algorithm. The first one is a laptop from 2017 which has an Intel Core i5-6200U. It is a dual-core, four-thread processor running at 2.4 GHz and it can boost up to 2.7 GHz. The second is a desktop PC that was built in 2018, equipped with the six-core, twelve-thread AMD Ryzen 5 2600X processor. Running at a base clock speed of 3.6 GHz, it has the capability of running at 4.2 GHz on all cores, provided that adequate cooling is maintained. This desktop is run using two cases: (1) all cores capped at 2.7 GHz, and (2) all cores capped at 3.9 GHz. The former is implemented to estimate the effect in the laptop setup if its logical cores were to increase beyond 4.

2.2 MCNP simulation model

Figure 1 shows the model of MCNP simulation that is run by the algorithm. Its code can be found in Appendix A. The model is a simple radiation attenuation experiment setup with a single monoenergetic source, a slab of shielding, and a detector volume. The source emits neutrons of 1-MeV in the direction of the detector (source direction biasing). The shield is made up of polyethylene and its dimension is set to 3 cm (W) × 3 cm (H) × 3 cm (L). The detector volume is a boron trifluoride-filled cylinder of 2 cm in diameter and 5 cm in length. F4 tally for neutrons is specified in the volume, while secondary photons are neglected. All 20 runs are performed at 2×10^6 nps and they are made to be independent of one another by having distinct seed numbers. A good summary of all the functions in MCNP can be found from the work of Shultis and Faw (2011) [14].

![Figure 1. MCNP simulation model. The red cross represents the point source which is placed close to the surface of the polyethylene shield (yellow box). The purple cylinder is the detector volume. Both bodies are surrounded by air (dashed line black box).](image)
2.3 Python multiprocessing for MCNP

The algorithm is developed using Anaconda, a distribution package of Python for scientific computing. It is user-friendly and the results of the code can be seen automatically in the same interface as the code. This means that the code can be written, run, and debugged conveniently. The algorithm is written in a Jupyter notebook (.ipynb) which is opened through the Anaconda Prompt. If done otherwise, it cannot access the MCNP cross-section file "xsdir". Figure 2 shows the flowchart of the algorithm, while the code can be found in Appendix B.

![Flowchart of the algorithm](image)

**Figure 2.** Python algorithm flowchart

Firstly, the multiprocessing function is called. For clarification, this function tells each logical core (thread) to run a single MCNP input at a time (Figure 3) rather than all of them run a single input. Therefore, the maximum number for the "Pool" depends on the number of logical cores. For example, the highest number for the desktop setup is 12 since it has 12 threads. After that, the "os" function is imported to enable the algorithm to run MCNP code from the command prompt. The mcnp.exe and the .ipynb file also need to be in the same folder.

Then, the MCNP input files are created using the algorithm so that each of them has a distinct seed number. This is done by defining a new function named "mcnp_in". The code for this function can be found in Appendix C. After the MCNP input lines have been added, the input is set to be run by using the command "os.system()". After the run is completed, the output file is read to extract the desired output such as tallies and relative errors. At the end of the function, all dump files ("runtpe", "runtpf", etc.) are deleted so that the next MCNP runs will not be exhausted of letters for their dump file name.

The multiple instances of MCNP runs are executed using the command "pool.starmap(mcnp_in, zip(input))". For example, if the Pool is set to 12 and there are 20 MCNP runs, the algorithm will write
12 MCNP input files and run them simultaneously. It does not wait until all 12 runs are completed to start on the remaining 8 runs. Instead, it creates and runs new ones as the current 12 runs finish. The time taken for all runs to be completed is recorded using the "time" function.

3. Results and discussion

Table 1 shows the results of the experiment for logical cores, while Figure 4 plots the completion time against the number of logical cores. Note that when there is only one logical core, it means that the MCNP runs are executed in series using one logical core. Based on Table 1, there are no significant changes to the average tally and the average relative error when the number of logical cores increases. The average relative errors for all cases are also constant at 0.0038 which is less than 0.05. Although this does not guarantee the acceptability of the tally, they are reliable based on the tally interpretation [12].

| No. of logical cores | Laptop (2.7 GHz) | Desktop (2.7 GHz) | Desktop (3.9 GHz) |
|----------------------|------------------|-------------------|------------------|
|                      | Time taken (s)   | Ave. tally (10^3 cm^2) | Ave. rel. error (10^-3) | Time taken (s) | Ave. tally (10^3 cm^2) | Ave. rel. error (10^-3) | Time taken (s) | Ave. tally (10^3 cm^2) | Ave. rel. error (10^-3) |
| 1                    | 395.79           | 1.06282            | 3.8               | 426.26           | 1.06322            | 3.8               | 283.93           | 1.06301            | 3.8               |
| 2                    | 247.34           | 1.06325            | 3.8               | 234.63           | 1.06262            | 3.8               | 159.18           | 1.06262            | 3.8               |
| 4                    | 180.58           | 1.06376            | 3.8               | 128.53           | 1.06160            | 3.8               | 88.27            | 1.06164            | 3.8               |
| 6                    |                  |                    |                   | 86.94            | 1.06208            | 3.8               | 60.18            | 1.06122            | 3.8               |
| 8                    |                  |                    |                   | 72.98            | 1.06212            | 3.8               | 50.89            | 1.06390            | 3.8               |
| 10                   |                  |                    |                   | 57.27            | 1.06020            | 3.8               | 40.22            | 1.06181            | 3.8               |
| 12                   |                  |                    |                   | 56.67            | 1.06091            | 3.8               | 39.57            | 1.06091            | 3.8               |

Figure 4. The completion times per number of logical cores

From Figure 4, both laptop and desktop (27 GHz) setups show a similar trend. When doubling the logical cores from 1 to 2 (or from series to parallel), the time taken for the laptop and the desktop decreases by 37.5% and 45%, respectively. The noticeable discrepancy in improvement might be due to
other differences in hardware (CPU cache, RAM amount and speed, storage speed, etc.). When the logical cores are increased to 4, the desktop shows a 70% decrease in time, while the laptop shows a 54% decrease. At 6 and 8 logical cores, the time taken for the desktop decreases by 80% and 83%, respectively. The remaining 10 and 12 logical cores only decrease the MCNP runtime at very similar rates of 86.6% and 86.7%, respectively. These results suggest that the improvement per increase in logical cores is not linear and there seems to be a diminishing return after a certain number of logical cores. The reason might be because at 1 to 6 logical cores, the MCNP runs are each assigned to one thread of the 6 physical cores. Further increase in the logical cores means that the physical cores are now being shared by two threads, reducing the performance gain. Other processes that are running in the background might also cause a decrease in improvement. Nevertheless, it is clear that by having more logical cores, the MCNP runs can be finished in significantly less runtimes.

Table 2 shows the results when the CPU clock speed increases. Similar to Table 1, the average tally and relative error show no significant changes with the clock speed. Figure 5 shows the details of improvement per GHz increase on the desktop setup. It is observed that increasing the clock speed reduces the MCNP completion times. The trends suggest that their relationship is linear. However, the gradient decreases with the number of logical cores. For 1 logical core, the time taken decreases by 121.35 seconds per GHz. However, increasing the logical core by 1, the gradient decreases to 66.95 seconds per GHz (45% less than the previous case). At the maximum number of logical cores, the completion time only decreases by 17.73 seconds per GHz. This trend shows the same diminishing returns as seen in Figure 4. There might be other factors that caused these results such as the way that instructions per core (IPC) work, a limited amount of cache memory, memory latency, and other hardware or software limitations/bottlenecks.

### Table 2. Experimental results for the clock speed

| Clock speed (GHz) | Number of logical cores | 1 | 2 | 12 |
|-------------------|-------------------------|---|---|----|
|                   | Time taken (s)          |   |   |    |
|                   | Ave. tally (10³ cm⁻²)   |   |   |    |
|                   | Ave. rel. error (10³)   |   |   |    |
| 2.4               | 480.54                  | 1.06137 | 3.8 | 264.79 | 1.06316 | 3.8 | 67.57 | 1.06091 | 3.8 |
| 2.7               | 426.26                  | 1.06322 | 3.8 | 234.63 | 1.06262 | 3.8 | 56.67 | 1.06117 | 3.8 |
| 3.0               | 384.87                  | 1.06382 | 3.8 | 211.62 | 1.06202 | 3.8 | 51.88 | 1.06215 | 3.8 |
| 3.3               | 349.99                  | 1.06213 | 3.8 | 192.49 | 1.06336 | 3.8 | 46.81 | 1.06433 | 3.8 |
| 3.6               | 320.56                  | 1.06221 | 3.8 | 176.35 | 1.06120 | 3.8 | 42.97 | 1.06404 | 3.8 |
| 3.9               | 296.10                  | 1.06267 | 3.8 | 162.99 | 1.06255 | 3.8 | 39.57 | 1.06391 | 3.8 |

Figure 5. The completion times per CPU clock speed

\[ y = -121.35x + 758.64 \quad R^2 = 0.98 \]
\[ y = -66.95x + 418.04 \quad R^2 = 0.979 \]
\[ y = -17.73x + 106.76 \quad R^2 = 0.952 \]
4. Conclusion
A Python algorithm is developed to investigate the improvement in MCNP runtimes that can be expected when multiprocessing is enabled. 20 MCNP simulations are run using two hardware setups: a laptop with a dual-core, 4-thread processor and a desktop with a 6-core, 12-thread processor. The tutorial for writing the algorithm is also provided and all relevant codes can be found in the appendices.

By enabling multiprocessing or running MCNP in parallel, the runtimes can be decreased by at least 37.5%. Both setups show a similar trend when the number of logical cores is increased. However, the improvement per increase in logical cores is not linear and diminishing returns are observed when the logical cores exceed the physical core count. Increasing the clock speed also decreases the time taken, but the gradient reduces as the logical cores increase. Therefore, it is recommended to prioritize the number of logical cores over the CPU clock speed when running multiple instances of MCNP for metaheuristic optimizations.

Future works can be directed towards expanding the algorithm to check for the acceptability of the tally results automatically. Photons can be added into the MCNP input so that mixed radiation can be simulated which is the usual case for metaheuristic optimizations. An investigation can be done on running the Python-MCNP algorithm using discrete graphics processing units (GPU) especially Nvidia GPUs. They typically have several hundred to thousands of cores. It is interesting to observe the improvements that these GPUs can provide to parallel MCNP simulations.

5. Appendices

Appendix A. MCNP input file

c 3 cm PE, 1 MeV neutrons
c cell card
    1 1  -0.001205 -1 2 3 imp:n=1
    2 2  -0.93 -2 imp:n=1
    3 3  -0.002831 -3 imp:n=1
999 0 1 imp:np=0

c surface card
    1 rpp 0 11 -2.1 2.1 -2.1 2.1 $air
    2 rpp 1 4 -1.5 1.5 -1.5 1.5 $shield
    3 rcc 6 0 0 5 0 0 2 $detector

c data card
mode n
nps 2e6
rand seed = 39360440332897
m1 6012 -0.00012 7014 -0.75527 8016 -0.23178 18000 -0.01283 $air
m2 1001 -0.1437 60000 -0.8563 $polyethylene
m3 5011 -0.15944 9019 -0.84056 $BF3
sdef pos=0.99 0 0 par=1 erg=1 vec=1 0 0 dir=d2 wgt=0.5
si2 0 1
sp2 01
f4:n 3
**Appendix B.** Python multiprocessing algorithm

```python
import time
start_time = time.time()
from multiprocessing import Pool as ThreadPool
import os
import numpy as np
pool = ThreadPool(4)
run = np.arange(1, 21)
tally = error = []
resultn = pool.starmap(mcnp_in, zip(run))
print(resultn)
for i in range(0, len(resultn), 1):
    tally = tally + [resultn[i][1][0][0]]
    error = error + [resultn[i][1][0][1]]
print("--- %s seconds ---" % (time.time() - start_time))
```

**Appendix C.** MCNP input function

```python
def mcnp_in(x):
    rseed = int(np.random.random() * 1e14)
    a = open("%sn" % x, "a+")
    a.write('c 3 cm PE, 1 MeV neutrons \n')
    a.write('c cell card \n')
    a.write(' 1 1 -0.001205 -1 2 3 imp:n=1 \n')
    a.write(' 2 2 -0.93 -2 imp:n=1 \n')
    a.write(' 3 3 -0.002831 -3 imp:n=1 \n')
    a.write(' 999 0 1 imp:np=0 \n')
    a.write('  \n')
    a.write('c surface card \n')
    a.write(' 1 rpp 0 11 -2.1 2.1 -2.1 2.1 $air \n')
    a.write(' 2 rpp 1 4 -1.5 1.5 -1.5 1.5 $shield \n')
    a.write(' 3 rcc 6 0 0 5 0 0 2 $detector \n')
    a.write('  \n')
    a.write('c data card \n')
    a.write('mode n \n')
    a.write('nps 2e6 \n')
    a.write('rand seed = 39360440332897 \n')
    a.write('m1 6012 -0.00012 7014 -0.75527 8016 -0.23178 18000 -0.01283 $air \n')
    a.write('m2 1001 -0.1437 60000 -0.8563 $polyethylene \n')
    a.write('m3 5011 -0.15944 9019 -0.84056 $BF3 \n')
    a.write('sdef pos=0.99 0 0 par=1 erg=1 vec=1 0 0 dir=d2 wgt=0.5 \n')
    a.write('si2 0 1 \n')
    a.write('sp2 0 1 \n')
    a.write('f4:n 3 \n')
    a.close()
```
os.system("mcnp5.exe i=%sn o=%so"%(x,x))

b.open("%so"%(x),"r")
data = b.readlines()
d = []
for i in range(0,len(data),1):
    ta1 = data[i].find("1tally 4")
    if ta1 != -1:
        d = d + [np.fromstring(data[i+9],dtype=float,sep=' ')]
b.close()

os.system("del %sn"%(x))
with open("%so"%(x), mode='w') as f:
    f.write('')

os.system("del runtp,runtpf,runtpg,runtp,runtpi,runtpk,runtpl")
return x,d

6. References

[1] Brown F B, Barrett R F, Booth T E, Bull J S, Cox L J, Forster R A, ... and Sweezy J 2002 MCNP version 5. Trans. Am. Nucl. Soc 87(273) 02-3935.
[2] Brun R, Urban L, Carminati F, Giani S, Maire M, McPherson A, ... and Patrick G 1993 GEANT: detector description and simulation tool CERN-W-5013
[3] Battistoni G, Cerutti F, Fasso A, Ferrari A, Muraro S, Ranft J, ... and Sala P R 2007 The FLUKA code: Description and benchmarking AIP Conf. proc. 896 31-49
[4] Romano P K and Forget B 2013 The OpenMC monte carlo particle transport code Ann. of Nuc. Ene. 51 274-281
[5] Wu Y, Song J, Zheng H, Sun G, Hao L, Long P, ... and FDS Team 2015 CAD-based Monte Carlo program for integrated simulation of nuclear system SuperMC Ann. of Nuc. Ene. 82 161-168
[6] Rydén T, Lagerlöf J H, Hemmingsson J, Marin I, Svensson J, Båth M, ... and Bernhardt P 2018 Fast GPU-based Monte Carlo code for SPECT/CT reconstructions generates improved 177Lu images EJNMMI phy. 5(1) 1-12
[7] Liang Y, Muhammad W, Hart G R, Nartowt B J, Chen Z J, James B Y, ... and Deng J 2020 A general-purpose Monte Carlo particle transport code based on inverse transform sampling for radiotherapy dose calculation Sci. rep. 10(1) 1-18
[8] Hu H, Wang Q, Qin J, Wu Y, Zhang T, Xie Z, ... and Zhan Y 2008 Study on Composite Material for Shielding Mixed Neutron and γ-Rays IEEE Tran. on Nuc. Sci. 55(4) 2376-2384
[9] Cai Y, Hu H, Pan Z, Hu G and Zhang T 2018 A method to optimize the shield compact and lightweight combing the structure with components together by genetic algorithm and MCNP code. Appl. Rad. & Isot. 139 169-174.
[10] Chen Z, Zhang Z, Xie J, Guo Q and Yu T 2019 Metaheuristic optimization method for compact reactor radiation shielding design based on genetic algorithm Ann. of Nuc. Ene. 134 318-329
[11] Wilson M 2019 Optimization of the radiation shielding capabilities of bismuth-borate glasses using the genetic algorithm Mat. Che. and Phy. 224 238-245
[12] Hu G, Hu H, Yang Q, Yu B and Sun W 2020 Study on the design and experimental verification of multilayer radiation shield against mixed neutrons and γ-rays Nuc. Eng. & Tec. 52(1) 178-184
[13] Hu G, Shi G, Hu H, Yang Q, Yu B and Sun W 2020 Development of gradient composite shielding material for shielding neutrons and gamma rays Nuc. Eng. & Tec. 52(10) 2387-2393
[14] Shultis J K and Faw R E 2011 *An MCNP primer* 1-42