A Faster Parallel Algorithm for Analyzing Drug-Drug Interaction from MEDLINE Database

Sulav Malla, Kartik Anil Reddy, Song Yang

Department of Computer Science and Engineering
University of South Florida
Tampa, USA
{sulavmalla, kartikreddy, songyang}@mail.usf.edu

Abstract

Use of multiple medications can have hazardous effect on health due to Drug-Drug interaction (DDI). MEDLINE database contains many articles related to DDI and a random-sampling-based algorithm has been purposed to automatically identify DDI by reading the substance field of MEDLINE records. However, this single threaded algorithm cannot fully utilize the processing capability of multi-core processors that are common today. As the number of cores in a single processor continue to rise, a parallel program that can better utilize these resources at hand become increasingly important. In this paper, we introduce a parallel version of the existing algorithm that can run on multiple cores simultaneously, making the computation faster. Experimental results on same hardware show that calculations done using our parallel multi-threaded program is up to 19 times faster compared to the original single threaded one.

Keywords—Drug-Drug interaction, Parallel algorithm, Multi-threading, Text mining

I. INTRODUCTION

Drug-drug interaction (DDI) refers to the change in effect of a drug when consumed together with another drug. Since people are generally taking more than one kind of drugs together, DDI has become a major cause of morbidity and mortality, leading to increased health care costs [1]. Therefore, doctors and medical care personnel need to consider possible DDI cases. MEDLINE (Medical Literature Analysis and Retrieval System Online) is a wildly used database for life sciences and biomedical literature. DDIs are usually reported in medical or scientific articles, hence searching MEDLINE provides a useful way to identify drug-drug interactions. However, the sheer volume of results returned on a single search in MEDLINE makes it impossible to analysis them manually. A random-sampling-based algorithm has been proposed in [1] to automatically identify DDI by reading the substance field of MEDLINE records.

The proposed algorithm was implemented as a single threaded python program. However, most of the processors that we find today are multi-core processors. Such single threaded program cannot fully utilize the multiple cores that we have at our disposal. In this project, we re-implemented the algorithm as a multi-threaded C++ program.

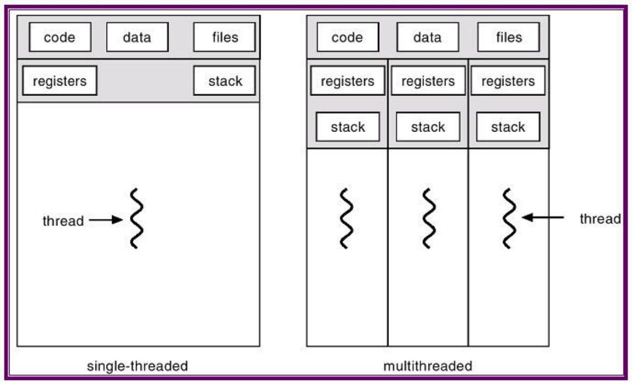


Figure 1: A single-threaded program vs a multi-threaded parallel program

Figure 1 shows the flow of a multi-threaded program. We can see that multiple threads of a program have separate individual stack and registers but share code and data region. We can exploit this to have data that is shared between the threads (in the data region) as well as data that is private to individual threads (in the stack region). The difference between implementing multiple threads instead of multiple programs is that threads are lightweight and switching between threads takes less time. As a practical example, if we want to paint 1000 pixels in the shape of an alphabet ‘A’, we will need to paint one pixel at each point and move to the next. In case of a multi-threaded environment with 1000 threads, each thread can paint the pixel at a separate point at the same time. The same operation in a multi-threaded environment can take 1/1000th the time it took (best case) in a single threaded environment.

The rest of the paper is organized as follows. Section II presents the algorithm and analysis of the existing program. Next, section III presents our results and we compare the single threaded program with our multi-threaded program for a sample drug. Finally, in section IV we draw some conclusions. We have also included measurements for all the drugs in appendix.

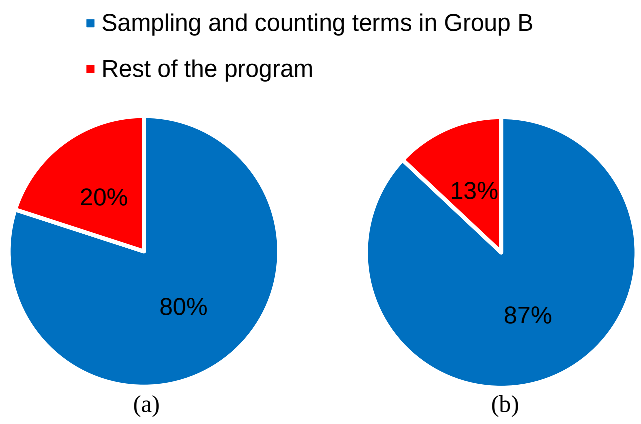


Figure 2: Percentage of time single threaded C++ program spends its time on. (a) for sampling frequency of 100 (b) for sampling frequency of 1000

II. METHOD

First, we describe the algorithm very briefly here, interested readers are encouraged to see [1] for further details. PubMed is a search engine that facilitates searching the MEDLINE database. We search PubMed with the drug name (e.g. ibuprofen) which returns the matching articles. These articles have various fields like title, abstract, substance, MeSH (Medical Subject Heading) terms etc. We then separate them into two groups. Group A consists of all the articles that have DDI related MeSH terms like 'drug agonism', 'drug interactions' etc. while the rest of the articles are placed in Group B. After that, we make a list of compounds and proteins that occurred in Group A along with their count of occurrence.

In the next step, we randomly sample articles from Group B, same number of articles that were present in Group A, and create a similar list of compounds and proteins with their count. We only include those terms that were found in Group A. This process of random sampling articles and counting terms in Group B is repeated n times with replacement (we refer to this as sampling frequency). Now, we have a distribution of all the compounds and proteins, found in Group A, in Group B.

Next step is to find the probability that count of a term found in Group A is from the distribution in Group B. If this probability is low, we can say that this term is related to DDI since it occurs in Group A very often but rarely in Group B. In this way, calculating the p-value for each term in Group A, we have them sorted in descending order. Compounds and proteins that are related to DDI will surface on top.

The program that was written in Python downloaded the articles, created the count table by sampling and calculated the p-values (and z-values). Since the downloading of articles from PubMed happens serially, multi-threading this part has little improvement. Hence, we split the program into two, one to download the articles and write them to files and other to read these files and do all the calculations. We used the same first program for downloading the articles (to avoid reinventing the wheel).

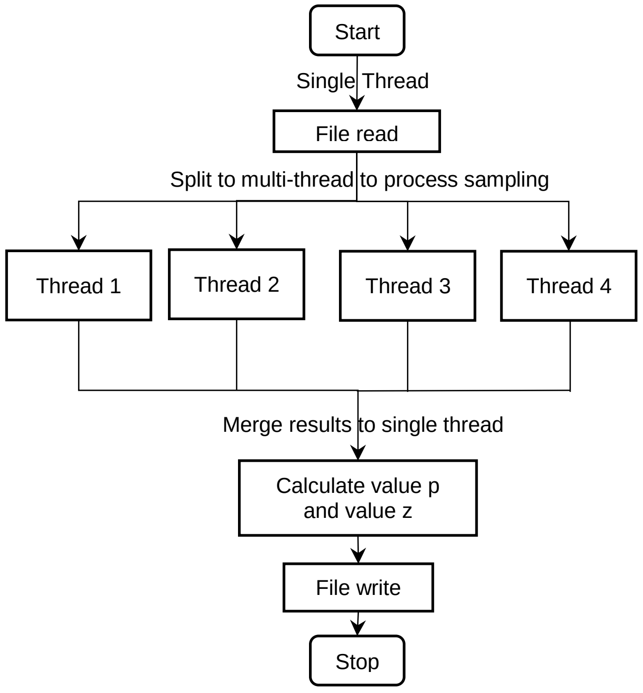

Start
Single Thread
File read
Split to multi-thread to process sampling
Thread 1
Thread 2
Thread 3
Thread 4
Merge results to single thread
Calculate value p and value z
File write
Stop

Figure 3: Flow of multi-threads program

The global interpreter lock (GIL) in Python does not allow multiple threads to execute in parallel as it provides the CPython compiler to only one thread at a time (this is important because multiple threads that run serially take longer than a single thread due to context switching). Hence, we reimplemented the second part of the program (that reads in files and does calculation) in C++. We profile the single threaded version of this C++ program to identify bottlenecks. We can see in figure 2 that most of the time is spent on sampling and counting terms in Group B. This portion of the program takes 80% of the total time for sampling frequency of 100 where as it grows to 87% for sampling frequency of 1000. This motivated us to focus on parallel execution of this portion of the program. Moreover, each sampling and counting of terms are independent and can run simultaneously without affecting the result.

In the C++ program, first, we create a Hash table to store terms (compound and protein) and the number times each term shows up in Group A. We also initialize a vector for each term to store the result from sampling (number of occurrences in the sample). This is the distribution of that term while sampling in Group B. For Group B, we store each record (all keywords from a medical article as a single record) in a vector. Next, we sample Group B n (sampling frequency) times by selecting records randomly. For each sample, we pick the same number of records as there are records in Group A. From the random records selected, we count the number of occurrences of each term and update its respective vector. Once this update is completed for n samples, we calculate a 'z-score' for each record that determines the probability of the compound or protein being involved in DDI.

We observed that implementation with C++ has a better performance than python. This may be because python is an interpreted language while with C++ we get a complied program. Next, to fully utilize the CPU, we want to change this into a

```
serial code;
// set the number of threads
omp_set_num_threads(NUM_THREADS);
// define a parallel region
#pragma omp parallel private(variables)
{
    parallel code;
    // parallelized for loop by breaking apart
    // iterations between threads
    #pragma omp for
    for(init;condition;increment)
    {
        body of for loop
    }
}
// end of parallel region
serial code;
```

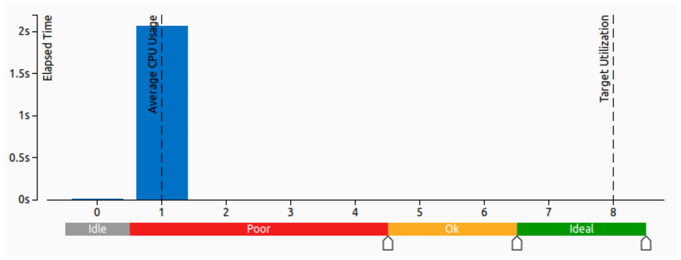
Figure 4: OpenMP program pseudo code

multi-threaded parallel program. The most important thing in parallel programming is that the parallel sections should be independent from each other. The single threaded program can be divided into three parts: I/O part (file read and write), sampling part and other calculation (z-score and p-value). As explained earlier, we parallelize the sampling part. Since we initialize the vector that stores the count to the size of n with all zeros, each thread accesses a different part of the vector for each sampling and there will not be any conflicts. Figure 3 shows the flow of our multi-threaded program.

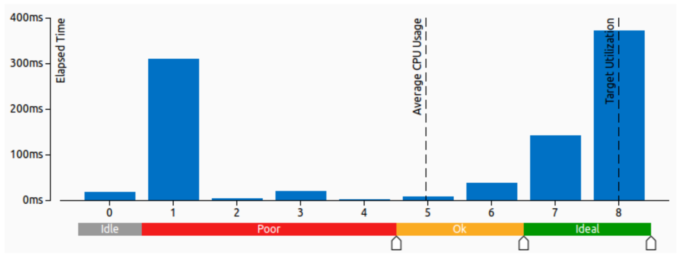
We use OpenMP API to implement the multi-threaded program. OpenMP is a very popular and widely used parallel programming model that is supported by C/C++/Fortran compiler from many vendors like GCC, Intel, IBM etc. [2]. Multi-threaded parallel programming in OpenMP is done through a set of complier directives *(#pragma)* and library functions. This makes it easy for us to define parallel regions in the code, the number of threads that we want, which variables to share between the threads and which ones to make private. For example, the complier directive *#pragma omp parallel* is used to define section of the program that is to run in parallel. Similarly, we can use the function *omp_set_num_threads()* to set the number of threads that we want to use. Figure 4 shows a snippet of pseudo code of the program that we used.

III. RESULT

We used our own laptop to run the programs and time it. Three different laptops were used to time execution for 6 different drugs, Aspirin, Cyclosporine, Ibuprofen, Rifampin, Simvastatin and Valproic Acid. In this section, we discuss results for the Ibuprofen drug. However, similar results hold for other drugs and hardware, as can be seen in the tables in the appendix. The hardware configuration of the laptop used for measurements of the Ibuprofen drug is as follows, CPU: Intel i7 2.2 GHz quad core processor with 8 logical threads (Hyper-Threading), memory: 8 GB. We used Intel compiler [3] to compile the program as it produces faster programs compared to GCC compilers. As a first step, we execute the program with 1 thread and 8 threads to see how they perform. The number of threads can be passed as an argument to our program. We choose sampling frequency of 1000. To profile the CPU utilization of



(a)



(b)

Figure 5: CPU usage histogram that shows the time the specific number of logical CPUs were being utilized simultaneously by our program (a) Simultaneous CPU running our program with 1 thread (b) Simultaneous CPU running our program with 8 threads

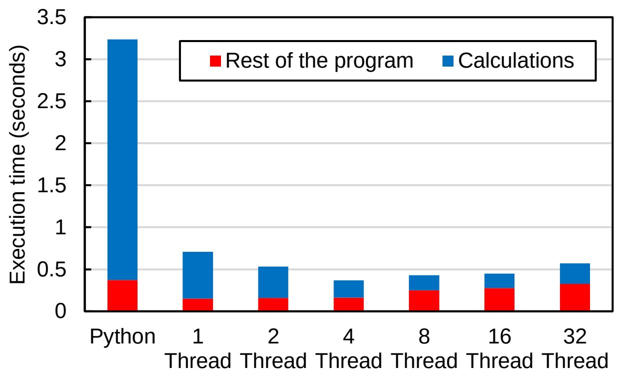


Figure 6: Total and calculation execution time for processing Ibuprofen drug with sampling frequency of 1000

our program we use the Intel VTune Amplifier [4] performance profiler.

Figure 5 shows, the time that our program was running simultaneously on the specific number of logical CPUs. As we can observe, for the program with 1 thread, almost all the time, only one CPU is being utilized with the rest 7 CPU remaining idle. On the other hand, for the program with 8 threads, all 8 CPU were utilized for more than half of the time. We can also note that about 30% of the time, our program is running on only one CPU. This may be because we have some serial portions in the program to read and write files, create multiple threads and combine the results.

Next, we compare the execution time for different number of threads and compare it with the original python program. To compare between them, we measured the total time as well as calculation time (excluding file I/O time and initializations). The

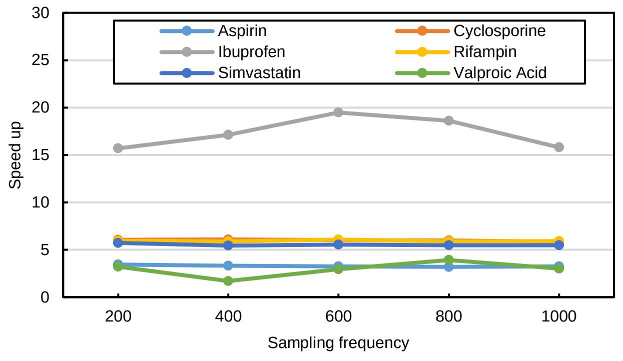


Figure 7: Calculation speed up for different drugs with varying sampling frequency

results are shown in figure 6. Our program significantly out performs the python program. It improves as we increase the number of threads up to a point. Performance is maximum when the number of threads is roughly equal to the number of available local CPUs. We can see that calculation time is minimum for 16 threads which is 16.5 times faster than the python program, while the total time is minimum for 4 threads, with 7.5x improvement. We can also see a trend on time taken to do rest of the calculation. It gradually increases as we increase the number of thread. This may be due to the overhead of creating multiple threads and combining back the results.

Similarly, we conducted experiment for 5 other drugs with different sampling frequency on other laptops. This time we used GCC compiler to compile the C++ program. One of the laptop was an older one with a dual core processor. We computed the calculation speed up time of 8 threaded program over the original python program. Figure 7 shows our result for all the 6 drugs with sampling frequency of 200, 400, 600, 800 and 1000. It can be observed that two drugs, Aspirin and Valproic Acid, which were measured on the old dual core laptop has the minimum speed up of about 3. Similarly, 3 drugs, Cyclosporine, Rifampin and Simvastatin, measured on another laptop had roughly the same speed up of 6. Finally, Ibuprofen, which was measured on laptop with comparatively better configuration and using Intel compiler gives the maximum speed up of over 15.

IV. Conclusion

Based on the results, we conclude that our multi-threaded program out performs the existing program. Performance increases as we increase the number of threads and reaches maximum when the number of threads is roughly equal to the number of CPUs present. Program compiled with Intel compiler perform better than the one compiled with GCC. We were able to achieve calculation times that were more than 15 times faster than the original single thread program. Moreover, our parallel program utilized all the available CPUs whenever possible. In this way, we were able to implement a faster, parallel, multi-threaded version of the drug-drug interaction algorithm.

Acknowledgment

We would like to thank Dr. Yicheng Tu for providing this opportunity to work on a database project and his constant support throughout the process. We would also like to thank Aditya Chandra Vothgod Ramachandra for his guideline, continued assistance and providing his previous work on this DDI project.

References

[1] Lu, Y. et al. A novel algorithm for analyzing drug-drug interactions from MEDLINE literature. Sci. Rep. 5, 17357; doi: 10.1038/srep17357 (2015)
[2] http://www.openmp.org/resources/openmp-compilers/
[3] https://software.intel.com/en-us/intel-compilers
[4] https://software.intel.com/en-us/intel-vtune-amplifier-xe

Appendix

Aspirin			
Python(single thread)			
Number of samples	Total	All calculation	Sampling
200 samples	2.393	2.051	
400 samples	4.26	3.902	
600 samples	6.158	5.769	
800 samples	7.946	7.547	
1000 samples	9.8	9.376	
C++ with 1 thread			
Number of samples	Total	All calculation	Sampling
200 samples	1.413	1.247	1.24
400 samples	2.77	2.467	2.453
600 samples	4.141	3.703	3.683
800 samples	5.414	4.953	4.907
1000 samples	7.025	6.318	6.284
C++ with 2 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.865	0.703	0.696
400 samples	1.737	1.428	1.414
600 samples	2.603	2.161	2.139
800 samples	3.371	2.792	2.765
1000 samples	4.421	3.712	3.678
C++ with 4 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.745	0.577	0.568
400 samples	1.636	1.335	1.319
600 samples	2.212	1.775	1.75
800 samples	2.906	2.48	2.291
1000 samples	3.571	2.885	2.846
C++ with 8 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.772	0.596	0.588
400 samples	1.485	1.181	1.161
600 samples	2.221	1.773	1.75
800 samples	2.923	2.361	2.326
1000 samples	3.562	2.89	2.853
C++ with 16 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.789	0.614	0.606
400 samples	1.457	1.173	1.158
600 samples	2.169	1.735	1.715
800 samples	2.941	2.378	2.352
1000 samples	3.616	2.92	2.886
C++ with 32 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.809	0.639	0.632
400 samples	1.518	1.209	1.194
600 samples	2.26	1.836	1.815
800 samples	2.938	2.371	2.343
1000 samples	3.707	3.033	2.999

Cyclosporine			
Python(single thread)			
Number of samples	Total	All calculation	Sampling
200 samples	0.824	0.552	0.483
400 samples	1.31	1.028	0.953
600 samples	1.803	1.508	1.423
800 samples	2.323	2.017	1.924
1000 samples	2.788	2.478	2.377
C++ with 1 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.254	0.201	0.198
400 samples	0.486	0.392	0.386
600 samples	0.726	0.593	0.584
800 samples	0.959	0.79	0.779
1000 samples	1.214	0.989	0.984
C++ with 2 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.169	0.117	0.113
400 samples	0.324	0.23	0.224
600 samples	0.483	0.348	0.339
800 samples	0.632	0.46	0.448
1000 samples	0.792	0.58	0.565
C++ with 4 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.153	0.1	0.096
400 samples	0.296	0.199	0.192
600 samples	0.42	0.286	0.276
800 samples	0.552	0.378	0.365
1000 samples	0.724	0.496	0.48
C++ with 8 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.153	0.1	0.097
400 samples	0.289	0.196	0.189
600 samples	0.424	0.289	0.281
800 samples	0.569	0.393	0.381
1000 samples	0.685	0.474	0.458
C++ with 16 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.161	0.107	0.104
400 samples	0.303	0.211	0.205
600 samples	0.459	0.326	0.316
800 samples	0.6	0.429	0.418
1000 samples	0.737	0.526	0.511
C++ with 32 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.185	0.133	0.13
400 samples	0.323	0.224	0.218
600 samples	0.518	0.387	0.372
800 samples	0.61	0.438	0.426
1000 samples	0.78	0.566	0.55

Ibuprofen			
Python(single thread)			
Number of samples	Total	All calculation	Sampling
200 samples	0.984	0.628	
400 samples	1.573	1.217	
600 samples	2.12	1.734	
800 samples	2.626	2.271	
1000 samples	3.238	2.864	
C++ with 1 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.234	0.188	0.188
400 samples	0.355	0.282	0.282
600 samples	0.484	0.381	0.381
800 samples	0.612	0.462	0.462
1000 samples	0.708	0.555	0.554
C++ with 2 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.158	0.108	0.108
400 samples	0.234	0.159	0.159
600 samples	0.343	0.239	0.239
800 samples	0.43	0.299	0.298
1000 samples	0.533	0.374	0.373
C++ with 4 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.112	0.064	0.064
400 samples	0.172	0.086	0.086
600 samples	0.281	0.153	0.152
800 samples	0.308	0.171	0.17
1000 samples	0.367	0.203	0.203
C++ with 8 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.104	0.04	0.04
400 samples	0.186	0.071	0.07
600 samples	0.249	0.089	0.088
800 samples	0.332	0.122	0.121
1000 samples	0.43	0.181	0.18
C++ with 16 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.146	0.065	0.064
400 samples	0.234	0.091	0.09
600 samples	0.357	0.105	0.105
800 samples	0.404	0.161	0.16
1000 samples	0.45	0.174	0.173
C++ with 32 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.295	0.135	0.134
400 samples	0.423	0.241	0.241
600 samples	0.497	0.233	0.232
800 samples	0.522	0.221	0.221
1000 samples	0.572	0.246	0.245

Rifampin			
Python(single thread)			
Number of samples	Total	All calculation	Sampling
200 samples	0.821	0.546	0.479
400 samples	1.308	1.025	0.949
600 samples	1.829	1.537	1.452
800 samples	2.299	1.997	1.903
1000 samples	2.816	2.502	2.579
C++ with 1 thread			
Number of samples	Total	All calculation	Sampling
200 samples	1.5	0.197	0.193
400 samples	0.496	0.405	0.399
600 samples	0.73	0.594	0.586
800 samples	0.971	0.8	0.788
1000 samples	1.211	0.993	0.978
C++ with 2 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.164	0.115	0.112
400 samples	0.323	0.231	0.225
600 samples	0.477	0.345	0.337
800 samples	0.639	0.464	0.453
1000 samples	0.791	0.58	0.566
C++ with 4 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.158	0.099	0.097
400 samples	0.285	0.192	0.185
600 samples	0.423	0.29	0.28
800 samples	0.551	0.379	0.366
1000 samples	0.687	0.473	0.456
C++ with 8 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.153	0.099	0.096
400 samples	0.293	0.198	0.191
600 samples	0.424	0.287	0.28
800 samples	0.547	0.377	0.365
1000 samples	0.685	0.472	0.458
C++ with 16 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.167	0.112	0.109
400 samples	0.328	0.233	0.227
600 samples	0.465	0.33	0.321
800 samples	0.586	0.412	0.399
1000 samples	0.739	0.528	0.513
C++ with 32 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.185	0.127	0.124
400 samples	0.341	0.248	0.242
600 samples	0.468	0.336	0.326
800 samples	0.622	0.449	0.437
1000 samples	0.789	0.579	0.56

Simvastatin			
Python(single thread)			
Number of samples	Total	All calculation	Sampling
200 samples	0.816	0.543	0.477
400 samples	1.31	1.028	0.952
600 samples	1.829	1.515	1.43
800 samples	2.316	2.012	1.919
1000 samples	2.79	2.474	2.374
C++ with 1 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.247	0.198	0.195
400 samples	0.49	0.392	0.387
600 samples	0.732	0.596	0.587
800 samples	0.961	0.787	0.519
1000 samples	1.199	0.989	0.975
C++ with 2 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.17	0.116	0.113
400 samples	0.324	0.232	0.226
600 samples	0.485	0.347	0.338
800 samples	0.636	0.461	0.452
1000 samples	0.783	0.573	0.559
C++ with 4 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.153	0.101	0.097
400 samples	0.284	0.192	0.185
600 samples	0.42	0.286	0.276
800 samples	0.555	0.38	0.367
1000 samples	0.703	0.496	0.479
C++ with 8 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.151	0.01	0.096
400 samples	0.289	0.453	0.189
600 samples	0.424	0.291	0.282
800 samples	0.572	0.396	0.384
1000 samples	0.696	0.485	0.47
C++ with 16 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.162	0.112	0.109
400 samples	0.311	0.214	0.208
600 samples	0.462	0.327	0.319
800 samples	0.581	0.407	0.395
1000 samples	0.731	0.518	0.503
C++ with 32 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.172	0.12	0.117
400 samples	0.343	0.25	0.244
600 samples	0.464	0.332	0.323
800 samples	0.636	0.464	0.451
1000 samples	0.784	0.569	0.554

Valproic Acid			
Python(single thread)			
Number of samples	Total	All calculation	Sampling
200 samples	1.393	1.117	
400 samples	2.37	1.136	
600 samples	3.423	3.115	
800 samples	4.445	4.133	
1000 samples	5.424	5.091	
C++ with 1 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.862	0.726	0.721
400 samples	1.642	1.449	1.441
600 samples	2.433	2.152	2.138
800 samples	3.311	2.944	2.927
1000 samples	4.187	3.724	3.703
C++ with 2 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.536	0.412	0.407
400 samples	1.108	0.803	0.794
600 samples	1.546	4.242	1.229
800 samples	2.281	1.909	1.886
1000 samples	2.534	2.074	2.052
C++ with 4 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.453	0.338	0.332
400 samples	0.871	0.663	0.652
600 samples	1.27	1.005	0.99
800 samples	1.718	1.347	1.333
1000 samples	2.169	1.74	1.71
C++ with 8 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.476	0.348	0.342
400 samples	0.849	0.666	0.658
600 samples	1.357	1.055	1.039
800 samples	1.753	1.055	1.039
1000 samples	2.16	1.689	1.67
C++ with 16 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.458	0.345	0.34
400 samples	1.172	0.961	0.949
600 samples	1.301	1.029	1.017
800 samples	1.772	1.399	1.381
1000 samples	2.143	1.719	1.694
C++ with 32 thread			
Number of samples	Total	All calculation	Sampling
200 samples	0.471	0.364	0.359
400 samples	1.232	1.015	1.003
600 samples	1.31	1.048	1.035
800 samples	1.771	1.423	1.411
1000 samples	2.149	1.711	1.689