Molecular dynamics calculations via distributed computing project Folding@Home: A case study

Andrew Canter
Weatherhead School of Management
Case Western Reserve University
Cleveland, Ohio
ajc340@case.edu

David Casente
Dept. of Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
dac153@case.edu

Ozan Dernek
Dept. of Phsyics
Case Western Reserve University
Cleveland, Ohio
oxd37@case.edu

Yibo Guo
Dept. of Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
yxg604@case.edu

***Abstract*—Folding@Home (FAH) has been one of the leading organizations conducting research on neurological and infectious diseases. In this paper we first provide an overall information about the program and the brief physics theory on protein folding. Later on, the primal components of FAH, namely the work unit, cores and clients are explained for general user. Finally we provide both CPU and GPU parallelization examples involving a toy model for CPUs and CUDA parallelization of GPUs, which utilizes an open source library OpenMM. While OpenMM is the interface between the GPU and the non-graphic task, we use this library as a stand alone software for molecular dynamics benchmark calculations.**

***Index Terms*—FAH, protein folding , parallel computing**

I. Introduction

Computer simulations has been used to search for a cure for many neurological and infectious diseases, as well as cancer treatments [1-3]. Although the computation power grow apace, conventional resources can be overwhelmed by the immense computation cost of simulations for disease treatments. Folding@Home (F@H or FAH) has been founded to provide the required resources for such research projects. FAH is an open source computing project with the computing power built by the volunteers from all around the world. Since October 1, 2001, more than 200 research papers have been published by Pande Lab as a direct result of FAH [4]. It was also the first supercomputer ever in history to surpass 1 exaflop, which happened in late March 2020, and as of April 12, 2020, FAH reached a computing power estimated around 2.34 exoflops [5].

In this paper, in section II we start with providing brief information about the purpose of FAH and the physical/biological concept behind it without getting too much into detail. In section III. we explain the general idea of FAH and how the system works for individual local machines. The FAH system runs behind a graphical user interface (GUI), hiding it's all complexity so that everyone - without any prior knowledge can participate. However, this prevents us to examine the algorithms behind it. For a better understanding, we explain the CPU parallelization using a toy model in section IV. Also, we used our HPC and installed the open source "OpenMM" library which uses application programming interface (API) to interface molecular simulation softwares to CPUs or GPUs. In this project, we use "OpenMM" compiled with CUDA, as a stand alone software for molecular simulations. The details of installation and the results of some protein folding benchmark calculations are given in section V.

II. Molecular simulations and Markov state model

A Markov state model (MSM) describes the entire dynamics of a system [6]. The system does not have to be consist of molecules, it can be any system that is evolving through time. That's why this formalism has been applied to a number of scientific fields [7]. In the frame of this project, the system is an atomic cluster, which is assumed to be in thermodynamic equilibrium. Then, the MSM is an $n \times n$ probability matrix, where n is the number of possible configurations of the cluster, i.e. every possible position of each atom in the system. In this matrix, the matrix elements a_{ij} represents the transition probability from state i to state j. From this definition one can guess that the diagonal elements are the probability of the system remaining in the same step. Once all these states are determined, one can map the dynamical evolution of the system by observing the state of the system at time τ (lag time). For this system to be feasible, τ should be small enough, i.e. "Markovian", so that the system lose it's "memory" of its previous states. Otherwise, the information of previous steps will accumulate as the evolution continues and will make the calculations virtually impossible. Once all the conditions are satisfied, one can calculate the free energies of each state and determine the possible pathways of evolution between any pair of states.

On top of the requirements, such as completeness of the configuration space and "Markovian" time step τ, there are other conditions that must be satisfied for MSM to be applicable:

- Each transition should occur equal times forward and backward ("Thermal equilibrium").
- Symmetric (So that MSM matrix will have mirror symmetry with respect to matrix diagonal).
- Given enough time, each state can be achievable ("Ergodicity").
- Final equilibrium distribution should be achieved, independent of the starting state or evolution pathway ("Aperiodicity").

Under these conditions MSM matrix can be decomposed into "eigenvectors" (column matrices) and their corresponding "eigenvalues". Each eigenvector will have n elements corresponding to n states. The magnitude and sign of every element will give the information about how individual states contribute to the process. Highest eigenvalue can be 1, corresponding to the equilibrium distribution, while all the other eigenvalues must me smaller than 1. Upon these eigenvalues, negative values are not physical, but positive values give the information about how the system evolves through its final distribution.

The key feature FAH exploits in this model is that the system is "memoriless", so that different pathways between a pair of states can be simulated in different computers.

III. FAH in Personal Machines

After a straightforward installation of the client, users can participate in calculations for a treatment of their choice. The GUI provides a few options for users to determine how much time and computation power will be donated. Whole process consists of three main process, namely work units, cores and client. Basic scheme of the process visualized in Fig. 1.

A. Work Units

A work unit is a partial calculation of a molecular simulation for a pair of states in a Markov model. Once the user determines the project, work unit is downloaded to run in the local machine automatically [8]. Each work unit serially runs for a pre-determined time in different volunteer's machine. Completed calculations gathered back in FAH database to derive a conclusion. If the calculation exceeds the runtime, the unit is re-assigned to another volunteer. To prevent time loss due to problematic work units, each unit goes through testing before being issued to participants [9].

B. Cores

Cores are the molecular dynamics softwares that calculate the work units. Within FAH there are several cores, each with specific hardware optimizations for different architecures and/or physical models. As the hardwares advance, these cores are optimized for the new system, while retiring the old ones. Majority of the cores are based on a molecular dynamics code suite called GROMACS. Other active cores are GPU3, the third generation GPU cores, based on OpenMM.[10]

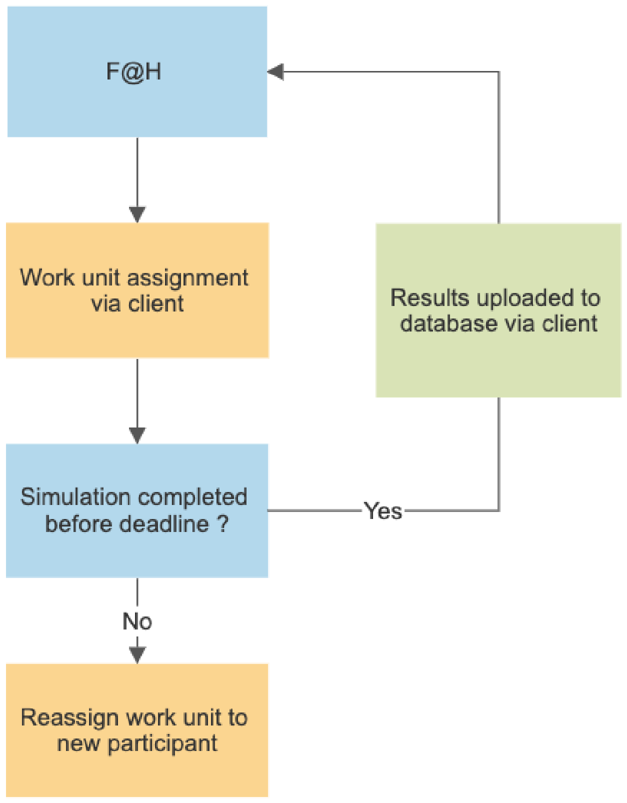
F@H
Work unit assignment via client
Results uploaded to database via client
Simulation completed before deadline ?
Yes
No
Reassign work unit to new participant

Fig. 1. Workflow of FAH. Client is responsible for downloading the assigned task and uploading the results to the database. If the given task is not done before deadline, it will be reassigned to a new participant.

C. Client

Client is the software that provides the GUI to user to interact with the other components of the system. Users can keep track of the progress using the client, such as viewing the user log and personal statistics. Moreover, participants can allocate the processing power using the "Power Slider". There are three levels enabled through the slider:

- **Light:** Folding is performed on CPU at half speed, GPU folding is disabled.
- **Medium:** Folding is performed on CPU at three-quarter speed, GPU folding is on. (Default)
- **Full:** Folding is performed on CPU at full speed, GPU folding is on.

Additionally, users are also provided "While I'm working" option, which enables folding at all times; and "Only when idle" option, which enables folding when the system remains idle for a pre-determined time.

Client is the component that downloads the work unit and uploads the result to FAH database once the simulation is completed. Client also downloads the appropriate cores depending on the architecture of the local machine. Computer clients are optimized for three different classes, namely uniprocessor and multi-core processors, and GPUs. Although GPUs provide the highest computation power, uniprocessors and multi-core processors can be used for relatively smaller tasks for efficiency.

1) Multi-core processing client: Costly calculations can be performed in significantly shorter runtimes, by running the same task on multiple CPU cores simultaneously. This can be achieved by using a parallelization method, such as OpenMP. Although In the following section, we will discuss a small example of OpenMP parallelization.

2) GPUs: GPUs has ben used for non-graphic tasks since they provide greater processing power than CPUs. Since these units are intended for graphics applications, it requires a suitable interface between the unit and the non-graphic task. FAH uses OpenMM for this purpose, which will be discussed in detail in the following section.

3) PlayStation 3: For almost six years, PS3s had been used for folding, with a successful calculation rate compared to PCs. Nevertheless, Sony product no longer supports folding [11].

IV. CPU Paralleling

CPU is a major location where programs are generally been executed, therefore we want to find out how paralleling in CPU could help us to execute several jobs at the same time, and eventually, overall speedup the job execution. This section in the paper does not consider the system resource management issue, and therefore cannot guarantee each single job will be accelerated on itself. In this section, we will provide two different ways of doing such CPU job execution paralleling. We will then conduct a short experiment to see how it actually works. We will provide comparison, analysis, and theoretical program behavior at the end of the section.

A. Public shell and common designing logic

Given our solution was primarily intended to serve OpenMM framework and its applications, we assume users will use Python to write their script, unless they have it otherwise declared. The program will provide user two options of CPU paralleling, multi-core or multi-thread. In general, the program will attempt to allocate each execution script a core, or a thread to run, depending on the user selection, but we will discuss the selection advice in a later sub-section. The multi-core practice is based on the OpenMP framework, and multithread approach is simply based on the Unix multi-thread library. Both approaches are based on the Unix come along multi-process service.

The program, upon finish processing user command line input, will start all up a new process to perform the requested user services by making a unix shell call. For example, if user requested to run file test.py in the default environment, the program will call

```
python3 test.py
```

for user internally. Another example is, assume user selected the ”shell” option to run ./test.sh, it will call

```
./test.sh
```

and wait until the end of such process. After the service was performed, the program will either merge the result to main program from each core, or simply exit the thread.

B. No parallel

First let’s see how long it would take if no paralleling is used, with our simple python example.

```
$ time python hello1.py > /dev/null

real   0m2.015s
user   0m0.008s
sys 0m0.005s
$ time python hello2.py > /dev/null

real   0m3.016s
user   0m0.008s
sys 0m0.005s
```

We see, in total, it goes to about 5 seconds. The simply python file used ”time.sleep()” to sleep in order to best simulate the true execution of some meaningful real script.

C. Multi-core

1) Introduction: The multicore approach of this CPU paralleling solution is based on the OpenMP. A OpenMP macro will be inserted into the code before the service sill start to perform the requested actions. Such macro is some special mark which can be recognized by OpenMP, and such framework will attempt to assign each true job with a CPU core. Each assignment is expected to go to different core, but the real assignment result depends on the OpenMP design and the availability of CPU.

2) Implementation: Here let’s see the source code of the implementation and a sample run result.

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "parallelizing.h"

int main (int argc, char ** argv) {
    int ret = 0;

    char * argvSelection = calloc(sizeof(char), argc
);
    memset(argvSelection, 0, sizeof(char) * argc);
    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-t") || !strcmp(argv[i
], "--type")) {
            i++;
            if (!strcmp(argv[i], "shell") || !(
strcmp(argv[i], "sh"))) scriptType = 1;
            else if (!strcmp(argv[i], "py") || !(
strcmp(argv[i], "py2"))) scriptType = 2;
            else if (!(strcmp(argv[i], "py3")))
scriptType = 3;
            else scriptType = 0;
            continue;
        } argvSelection[i] = 1;
    }

    #pragma omp parallel
    #pragma omp for reduction (+ : ret)
    for (int i = 1; i < argc; i++) if (argvSelection
     [i]) ret += worker(argv[i]);
    free(argvSelection);
    return ret;
}
```

Listing 1. multi-core main.c

```

#ifndef CSE438FINPROJ_PARALLELIZING_H
#define CSE438FINPROJ_PARALLELIZING_H

char scriptType; // 0 = undefined, 1 = bash / Shell
     script, 2 = Default Py or Py2 Script, 3 = Py3
     Script
// This type by default goes to Python3

int worker(char *);

#endif //CSE438FINPROJ_PARALLELIZING_H
```

Listing 2. multi-core parallelizing.h

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "parallelizing.h"

int worker(char * scripts) {
    char * cmd = calloc(sizeof(char), 9 + strlen(
     scripts));
    if (scriptType == 3 || !scriptType) sprintf(cmd,
     "python3 %s", (char *) scripts);
    else if (scriptType == 2) sprintf(cmd, "python %
     s", (char *) scripts);
    else (sprintf(cmd, "%s", (char *) scripts));
    system((char *)cmd);
    free(cmd);
    return 0;
}
```

Listing 3. multi-core parallelizing.c

3) Execution and result: To make and execute the program, use

```
make clean all
./a.out hello1.py hello2.py // Using default Py3
     environment
./a.out -t sh ./echo.sh // Specifying to use bash
     shell as interpreter
```

Listing 4. Make and run the multi-core approach

```
$ make clean all
rm -rf main.o parallelizing.o
rm -rf main.s parallelizing.s
rm -rf a.out
gcc -fopenmp -Werror -Wall -Wextra -Wpedantic -
     Wshadow -std=c99 -O2 -g -c main.c -o main.o
gcc -fopenmp -Werror -Wall -Wextra -Wpedantic -
     Wshadow -std=c99 -O2 -g -c parallelizing.c -o
     parallelizing.o
gcc -fopenmp -Werror -Wall -Wextra -Wpedantic -
     Wshadow -std=c99 -O2 -g main.o parallelizing.o -
     o a.out
$ ./a.out hello1.py hello2.py
Hello Python 1!
Hello Python 0!
0 woke up!
Hello 1 slept 3 sec
$ ./a.out -t sh ./echo.sh
Hello Echo!
$ time ./a.out hello1.py hello2.py > /dev/null

real   0m3.029s
user   0m0.236s
sys 0m0.018s
```

Listing 5. Multi core result

D. Multi-thread

1) Introduction: Different from the multi-core implementation, multi-threading allows all scripts to run on the same core, but different threads. Instead of assigning jobs with core before entering the worker function, it assigns thread to each script inside the worker function. However, it may requires to have one extra function, compare with multi-core and as required by Unix API, as the functionality inside each thread. Such thread assignment guarantees that each script goes into a different thread.

2) Implementation: Here let's see the source code of the implementation and a sample run result of the multithreading approach.

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "parallelizing.h"

int main (int argc, char ** argv) {
    int ret = 0;

    char * argvSelection = calloc(sizeof(char), argc
    );
    memset(argvSelection, 0, sizeof(char) * argc);
    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-t") || !strcmp(argv[i
    ], "--type")) {
            i++;
            if (!strcmp(argv[i], "shell") || !(
    strcmp(argv[i], "sh"))) scriptType = 1;
            else if (!strcmp(argv[i], "py") || !(
    strcmp(argv[i], "py2"))) scriptType = 2;
            else if (!(strcmp(argv[i], "py3")))
    scriptType = 3;
            else scriptType = 0;
            continue;
        } argvSelection[i] = 1;
    }

    worker(argc, argv, argvSelection);
    free(argvSelection);
    return ret;
}
```

Listing 6. Multi-thread main.c

```

#ifndef CSE438FINPROJ_PARALLELIZING_H
#define CSE438FINPROJ_PARALLELIZING_H

char scriptType; // 0 = undefined, 1 = bash / Shell
     script, 2 = Default Py or Py2 Script, 3 = Py3
     Script
// This type by default goes to Python3

int worker(int, char **, char *);

#endif //CSE438FINPROJ_PARALLELIZING_H
```

Listing 7. Multi-thread parallelizing.h

```
#include <pthread.h>
#include <zconf.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "parallelizing.h"

static void * exec(void * argv) {
    char * cmd = calloc(sizeof(char), 9 + strlen(
    argv));
    if (scriptType == 3 || !scriptType) sprintf(cmd,
     "python3 %s", (char *) argv);
    else if (scriptType == 2) sprintf(cmd, "python %
    s", (char *) argv);
    else (sprintf(cmd, "%s", (char *) argv));
    system((char *)cmd);
    free(cmd);
    return 0;
}

int worker(int argc, char ** argv, char * filter) {
    /*pthread_t tid;
    return pthread_create(&tid, 0, &exec, scripts) +
     pthread_join(tid, 0);*/
    pthread_t tid[argc];
    for (int i = 0; i < argc; i++) if (filter[i])
    pthread_create(&tid[i], 0, &exec, argv[i]);
    for (int i = 0; i < argc; i++) if (filter[i])
    pthread_join(tid[i], 0);
    return 0;
}
```

Listing 8. Multi-thread parallelizing.c

3) Execution and result: To make and execute the program, use

```
make clean all
./a.out hello1.py hello2.py // Using default Py3
    environment
./a.out -t sh ./echo.sh // Specifying to use bash
    shell as interpreter
```

Listing 9. Make and run the multi-thread approach

```
$ make clean all
rm -rf main.o parallelizing.o
rm -rf main.s parallelizing.s
rm -rf a.out
gcc -fopenmp -Werror -Wall -Wextra -Wpedantic -
    Wshadow -std=c99 -O2 -g -c main.c -o main.o
gcc -fopenmp -Werror -Wall -Wextra -Wpedantic -
    Wshadow -std=c99 -O2 -g -c parallelizing.c -o
    parallelizing.o
gcc -fopenmp -Werror -Wall -Wextra -Wpedantic -
    Wshadow -std=c99 -O2 -g main.o parallelizing.o -
    o a.out -pthread
$ ./a.out hello1.py hello2.py
Hello Python 0!
Hello Python 1!
0 woke up!
Hello 1 slept 3 sec
$ ./a.out -t sh ./echo.sh
Hello Echo!
$ time ./a.out hello1.py hello2.py > /dev/null

real  0m3.028s
user  0m0.029s
sys 0m0.017s
```

Listing 10. Multi thread result

We can see the performance and how it looks like are almost the same, which because there are two variants of a same CPU paralleling structure, and the program is not complex enough to see the difference. Theoretically, given CPU is somewhere used to perform computations, and such process will be idled while waiting for I/O. Therefore, with a program which is complicated enough, we expect I/O heavy scripts may be more benefited from the multi-thread approach, which is also saving CPU resources: Computing heavy program may prefer multi-core approach, as multi-core is truly adding more computing power toward the requested scripts.

E. Hybrid

For some cases, combining both models can be benficial, as we have some I/O heavy program and some computational heavy program. Therefore, our solution do support a hybrid mode which allow people be benefited from multithreading for I/Os and multi-core for computations. Here is how it works: Assume ”mt.out” is the multithreading executable, and ”omp.out” is the multi-core executable.

```
$ ./omp.out -t sh './mt.out hello1.py hello2.py' '
    multithread/echo.sh'
Hello Echo!
Hello Python 0!
Hello Python 1!
0 woke up!
Hello 1 slept 3 sec
```

Listing 11. Hybrid mode - multithreading in multi-core execution

Note that in the hybrid mode, the ”-t” type option is required to be sh, or there should have no difference with pure multi-core mode. Requires user to figure out the core grouping before execution. In the above example, ideally, ”./mt.out hello1.py hello2.py” will take one core, and ”multithread/echo.sh” will take another core.

F. Comparison and general OEM advises

As we discussed above, we do suggest users to use the hybrid mode to achieve the best of the performance. Due to the example limitation, we are unable to verify this assumption with some visible instances, but manually grouping to-be-executed script into cores by averaging I/O need would be the best option.

V. Demonstration on HPC

For the demonstration, we wanted to look further into how FAH is able to harness the power of parallelization on user's machines in order to complete complex protein folding that is based on simulated physics and chemistry.The implementation of FAH's parallelization is proprietary and the application itself is quite rigid, so we can't learn and demonstrate exactly how FAH uses multi level parallelization to use the CPU and GPU's processing power. Instead we focused on attempting to run some simpler chemistry simulations, that the FAH simulations are built up of, on the CWRU HPC. The goal of the demonstration was to analyze parallelization techniques including MPI, and OpenMP on the time to run Gromacs simulations built on the OpenMM library. Unfortunately, due to some dependency errors, the full version of OpenMM could not be installed on the HPC and the full experiment planned could not be completed. In this demonstration section, there will be a documentation of the smaller analysis that was able to be completed on the HPC and the planned analysis, where it went wrong, the planned setup, and expected results. With more time and collaboration, we believe that we can get the Gromacs simulations running and complete the full analysis soon.

The main components we researched in order to conduct the parallelization analysis were the OpenMM library, the Gromacs library, GPU acceleration using CUDA, MPI, and OpenMP. Below are general summaries of the OpenMM and Gromacs Libraries:

1) OpenMM: OpenMM is a library that models molecules and simulates interactions between them. There are 5 basic components to a OpenMM simulation. A system defines the general attributes of the model such as number of molecules, the properties of each molecule and the interactions between them. A force defines interactions between molecules or any force on a molecule in general. The context stores all of the above information for a specific simulation. An integrator implements a specific algorithm that actually progresses the simulation, calculating forces and molecule positions over time. Lastly, a slate stores the output of an integrator at a specific time in the simulation, capturing the state of the simulation.

2) Gromacs: Gromacs expands upon OpenMM and the package is mainly used to simulate millions of particles to represent larger scale biomedical objects such as proteins or polymers. Gromacs has integrated multi-level parallelization to run the simulation efficiently.

A. GPU parallelization analysis of openmm Python simulation

First we need to install MiniConda on our HPC environment by following the guide on CWRU HPC site. With MiniConda installed, we can request a GPU, load the CUDA module, and create a Python environment, called "python3" in this implementation, which we can install Conda's version of OpenMM on. Conda's OpenMM library installation is only able to compile Python code and does not have the ability to compile C code.

```
$ srun -p class -A sxg125_csds438 --gres=gpu:2 --
  pty bash
$ module load cuda/10.0
$ eval ''\$(/home/dac153/miniconda3/bin/conda
  shell.bash hook)''
$ conda create --name python3 python=3.5
$ conda activate python3
$ conda install -c omnia/label/cuda100 -c conda-
  forge openmm
$ conda list
```

Listing 12. MiniConda installation and activation of python3 environment. Note that last line is only used to make sure that the OpenMM library is installed. Please see the attached image "condaListCode" to see the packages in this environment.

Now that OpenMM is installed, we can take a look at our Python OpenMM script, which creates a biomolecular system from the file "input.pdb", uses models to parametrize the system, and runs the simulation for 10,000 steps. The lines of code we are focusing on are the platform instantiation and property definition. For our analysis, there are four different variants we are running of this simple biomolecular simulation. In all variants, the platform used is OpenCL, which runs the code on the GPU. The two parameters are the number of GPU's in use and the precision type. OpenCL is able to parallelize a simulation across multiple GPU's and that is how we are parallelizing the simulation in this analysis. One GPU will be used for the Serial implementation and two will be used for the Parallel ones. In addition, we will be running the simulations using different precision options to see if there are any timing differences. There are two precision options: single, which is the quickest and least accurate, and double, which computes forces and integrates slower due to the increased threshold. Four different variants of the simulation is run using one GPU-single precision, one GPU-double prescision and two GPUs-single precision. See the results respectively at attached images "serialSingleCode", "serialDoubleCode", "parallelSingleCode" and "parallelDoubleCode" respectively.

Now that we've covered the necessary simulation code, we can run each of the variants and analyze the time differences. The batch submission file outlines the code necessary to run these simulations and reiterates some of the setup code. Note that the time function is used to measure the timing of the simulations. While the batch submission code is displayed, the simulation outputs were run in an interactive session. Execution time of each code is given in Table I

```
#!/bin/bash
#SBATCH -p classc --gres=gpu:2 -A sxg125_csds438

module load cuda/10.0
eval ''\$(/home/<dac153>/miniconda3/bin/conda
  shell.bash hook)''
conda activate py3
conda list
time python simulationSerialSingle.py
time python simulationSerialDouble.py
time python simulationParallelSingle.py
time python simulationParallelDouble.py
```

Listing 13. Batch file for the python code submission of four variants.

TABLE I
THE EXECUTION TIME OF EACH VARIANT IS GIVEN. DOUBLE PRECISION CALCULATIONS TAKE MORE TIME, WHILE RUNNING THE SIMULATION ON PARALLEL GPUS REDUCE THE CALCULATION TIME SIGNIFICANTLY.

	SerialSingle	SerialDouble	ParallelSingle	ParallelDouble
real	0m32.447s	0m50.475s	0m5.369s	0m21.881s
user	0m18.623s	0m29.124s	0m3.057s	0m12.756s
system	0m13.448s	0m20.877s	0m2.035s	0m8.881s

We can notice a few insights from the outputs of the four variants. First of all, it makes sense that the simulations using double precision take much longer than the single precisions, given that the double precisions are much more computationally intensive due to the higher precision threshold.

Second, the outputs confirm that the parallel simulations run significantly faster than their serial counterparts. The parallel simulations have access to double the processing power and can run multiple lines of code simultaneously, all of which contribute to the time decrease. Lasly, we can see that there is a discrepancy between the increase in time from single to double precision for the Serial and Parallel simulations. For the parallel simulation, the system time required to run the simulation quadruples when switching to double precision. On the other hand, for the Serial implementation, the time increase is around fifty percent. This difference in time increase is likely due to the fact that there is a large portion of time dedicated to initializing the simulation in the Serial Simulation, which is not affected by the precision designation.

B. CUDA parallelization on openmm C simulation

After requesting a GPU, in order to install the full OpenMM library and not just the application layer (previous part), we need to load the latest cmake module and the accompanying gcc module. Next step is downloading and unzipping the latest OpenMM library release, we can use ccmake, the visual cmake configuration tool to complete configuration and fix any incorrect file paths.

```
$ module load gcc/7.3.0
$ module load cmake/3.10.2
$ ccmake -i /home/dac153/openmm/openmm/
```

Listing 14. Loading gcc and cmake modules

Upon the first configuration attempt, it fails due to HPC's version of Swig being outdated. So, we can install our own updated version of swig in our HPC envirronment by following the installation instructions. With the updated Swig executable in hand, we can enter the advanced mode of ccmake and manually enter in the file path to point to our newly installed Swig (Please refer to attached image "cmakeInterfaceCode").

Unfortunately, we couldn't proceed due to an error occured while loading shared libraries. This is the critical error that is preventing the installation of the full OpenMM library. In this case, the configuration fails because cmake cannot locate libimf.so. The best guess as to why this error occurs is the faulty installation of Swig due to lack of sudo access. If the Swig version on HPC is updated, we believe the installation would be possible. Note that we have gotten the configuration step to complete and even the make step to complete in previous iterations, but the make install step failed with the same error that libimf.so could not be located.

Since there is no more implementation to walkthrough, the rest of this part explains the planned analysis. With the full OpenMM library installed on out HPC environment, we can now compile and run simulations using c code, which allow us to use the CUDA parallel computing platform. A simple simulation C program simulates a few Argon molecules moving back and forth (Codes are available in our GitHub repository). Important lines of code to focus on are the platform instantiation and property definition. The code is similar to part one, but now we are choosing the CUDA platform. For the analysis, we would run the simulation with the same set up as part one, comparing the timing of simulations run on one and two GPU's. In addition, we plan on augmenting the C program to include CUDA parallelization code. This would allow us to parallelize on the thread level of the GPU, not just on the overall GPUs. We plan on parallelizing the creation of the Argon atoms and the simulation step. Then we would compare the timing of the serial simulation, and different levels of parallelism based on the thread count to see the performance speed up based on parallelization.

C. Multi-level parallelization analysis of Gromacs simulation

This part describes the planned (but not been able to achieve) final analysis of parallelization techniques. The goal of the planned last analysis was to install the Gromacs library on top of the OpenMM library and run Gromacs simulations serially and parallelized using a combination of MPI and OpenMP. The advantage of the Gromacs demo is that the Gromacs simulations are much larger with significantly more objects and interactions than the previous parts. We would be able to display more accurately how FAH harnesses the power of multi-level parallelism. With Gromacs installed, we plan on compiling a simple large scale simulation and compare different variants. We would have a serial implementation, various implementations using different numbers of MPI ranks and OpenMP threads. This variation would allow us to analyze the performance increase as processing power increases. Also, we could have variations using a single GPU and two GPUs to tie in the previous parts.

VI. Conclusion

Folding@Home (FAH) is the biggest computational source that provides an immense support to scientists for disease treatment. By the contribution of individuals from all around the world, FAH exceeded 2 exaflops. Utilizing the Markov state model, FAH distributes work units to volunteers' local machines through a client program and collect the results in the same manner to conclude a result. Once the client is installed, the "folding" can be done in different levels, each uses a parallelization model, namely multi-core CPU and GPU parallelization. To investigate these models, we started this work with the idea of compiling a core (a molecular dynamics program) on CPU and GPU on our school's HPC. We chose OpenMM library as a stand alone program for this task. Due to time limitation and technical difficulties, we did not apply CPU parallelization, but rather explained different approaches to this, using a toy model. We managed to achive our goal partially and run a python code using OpenMM library in serial and parallel GPUs. The problems we have faced was mainly caused by the lack of sudo access and not being able to modify/update the necessary components. Nevertheless, we had the glimpse how FAH simulations are run on different levels, and provided with the updated components, we can achieve the rest of our goals. To reproduce the results given in section IV and V reader can refer to our GitHub repository .

REFERENCES

[1] Kuzmanic A, Bowman GR, Juarez-Jimenez J, Michel J, Gervasio FL, " Investigating Cryptic Binding Sites by Molecular Dynamics Simulations," Acc Chem Res. 2020 Mar 05.

[2] Porter JR, Meller A, Zimmerman MI, Greenberg MJ, Bowman GR, "Conformational distributions of isolated myosin motor domains encode their mechanochemical properties," Elife. 2020 May 29;9

[3] Sigg D, Voelz VA, Carnevale V, "Microcanonical coarse-graining of the kinetic Ising model," J Chem Phys. 2020 Feb 28;152(8):084104

[4] Folding@Home: Paper Results, https://foldingathome.org/papers-results/ .

[5] Folding@Home: Active CPUs and GPUs by OS, https://stats.foldingathome.org/os .

[6] Husic B E, Pande V S, "Markov State Models: From an Art to a Science," J. Am. Chem. Soc. 2018, 140, 2386−2396

[7] Schütte C, Fischer A, Huisinga W, Deuflhard P J, "A Direct Approach to Conformational Dynamics based on Hybrid Monte Carlo ," Comput. Phys. 1999, 151, 146−168.

[8] Beberg A L, Ensign D L, Jayachandran G, Khaliq S, Pande V S, "Folding@home: Lessons from eight years of volunteer distributed computing," 2009 IEEE International Symposium on Parallel and Distributed Processing, Rome, 2009, pp. 1-8, doi: 10.1109/IPDPS.2009.5160922.

[9] OpenMM work unit testing, https://folding.typepad.com/news/2011/04/more-transparency-in-testing.html .

[10] Folding@Home: Active Projects, https://apps.foldingathome.org/psummary .

[11] Folding@Home: Client Statistics by OS, https://www.webcitation.org/6AqqwVmpj?url=http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats .

[12] Nickolls J, Buck I, Garland M, Skadron K, "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?," Association for Computing Machinery, vol.6, num.2, 2008 Mar.

[13] Abraham M J, Murtola T, Schulz R, Páll S, Smith J C, Hess B, Lindahl E, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers,", SoftwareX, vol. 1-2, p. 19-25, 2005.

[14] Eastman P, Swails J, Chodera JD, McGibbon RT, Zhao Y, Beauchamp KA, et al., "OpenMM 7: Rapid development of high performance algorithms for molecular dynamics," PLoS Computational Biology Software, vol. 13, 2007.

[15] Openmm user guide: http://docs.openmm.org/latest/userguide/application.html .

[16] Gromacs parallelization documentation: http://www.gromacs.org/Documentation/Acceleration_and_parallelization#GPU_acceleration .