University of Illinois at Urbana-Champaign

SPIN Project Report

Chin-Yu Cheng

August 4, 2016

1 Background

In evolutionary dynamics, we can calculate the fitness of organisms with specific genotypes. It measures how well a genotype can adapt to the environment and its ability to produce offsprings. Finding the fittest genotype of a virus can help the process of vaccine design. The project is divided into the following two parts.

1.1 Protein Analysis

The first project consists of analyzing three proteins produced by the flu virus (NS1, NS2, NP). The main goal is to use the existing OpenMP and CUDA programs to find the best fit of the virus proteins.

1.2 Accelerating Population Dynamics Simulation

The goal of the second project is to accelerate the Population Dynamics simulation program, which is a Monte Carlo simulation of virus evolving in the human body. Available APIs include OpenMP and CUDA.

2 Project Results

2.1 Protein Analysis

2.1.1 Results

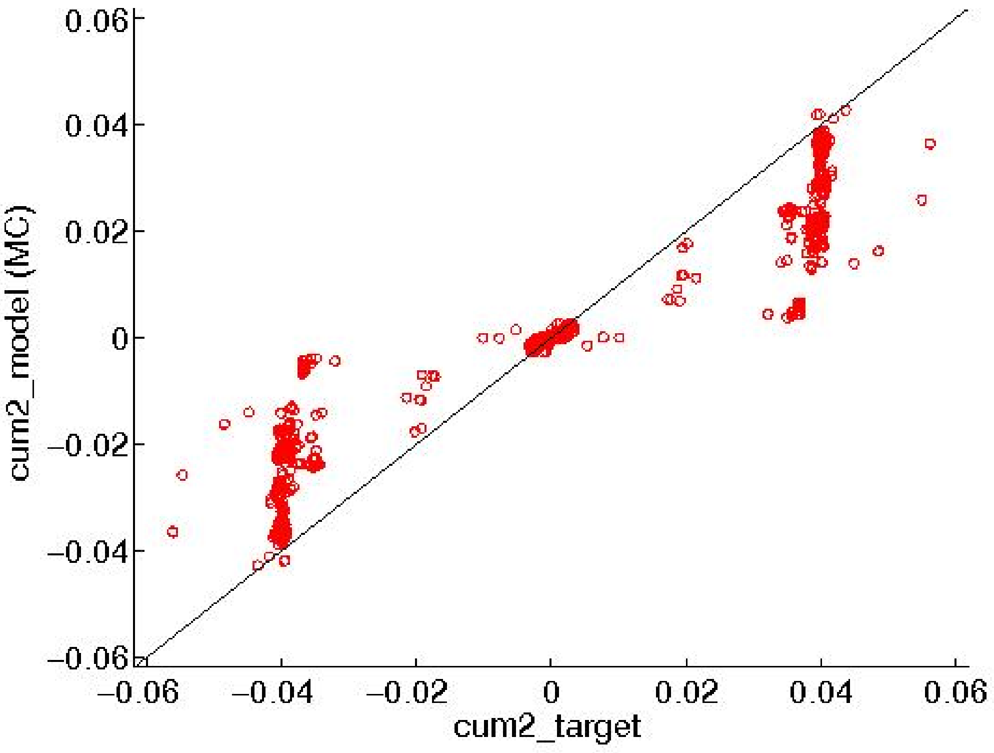


Figure 2.1: A good fit of the cum2 model should result in the samples aligning with the diagonal line. This figure shows a fit with 10 million cycles per iteration.

Currently we have very good results for protein NS2. We're able to get stable h and j parameters which provide a good fit when run for long cycles.

NP and NS1 protein analyzing still hasn't achived satisfying results yet. They are longer proteins that require longer cycles for a good fit. Their lengths are 340 and 210 amino acids respectively compared to NS2's 107. We'll keep running the analyzing programs for these prteins.

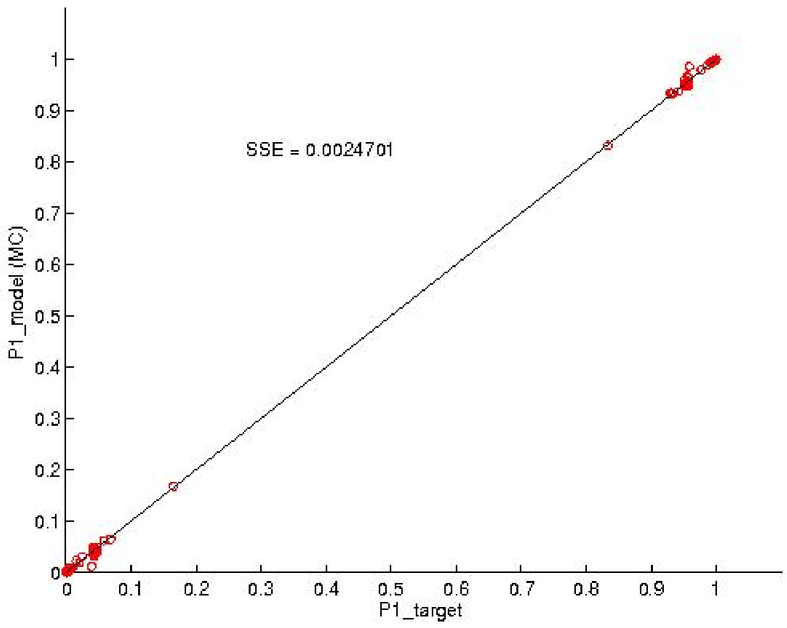


Figure 2.2: P1 fit at very large samples

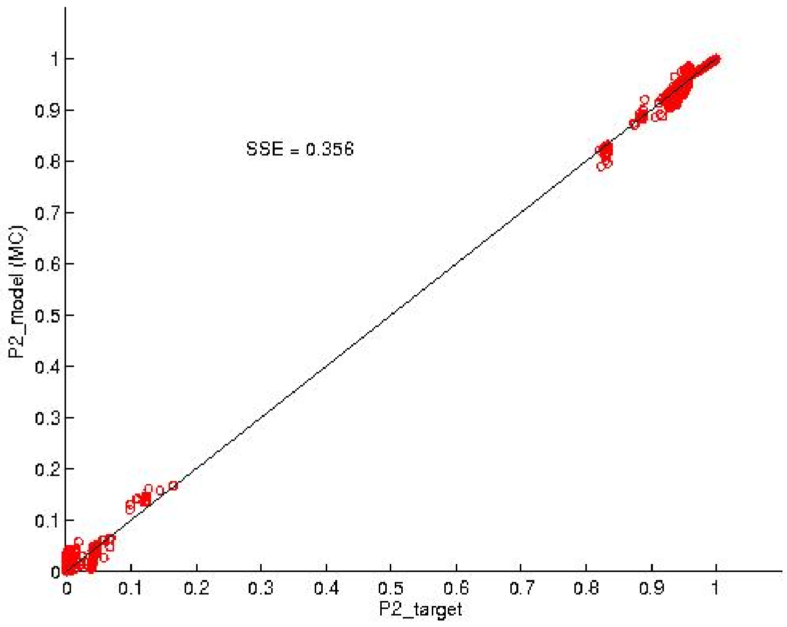


Figure 2.3: P2 fit at very large samples

2.2 Accelerating Population Dynamics Simulation

2.2.1 The problem

In the population dynamics simulation program, the most time consuming region of the algorithm involves selecting random sequences of proteins to mutate. This process involves looping over N populations($> 10^4$) and selecting different sequences in each iteration.

Listing 1: code snippet of *main*0

```
// original code (main0)
#pragma omp single
{
   while(...) // looping through N positions
   {
      // sequence selection
            .
            .
      // a time consuming region that is restricted to one thread
            .
            .
   }
}
```

The mutation selection process is originally implemented using a serial loop to ensure that a different sequence is selected in every iteration. Though some parts of the mutation process could be done in parallel, the whole loop is restricted to be run by only one thread. This results in poor utilization of the CPU despite using the OpenMP program to speed up the program.

The original version of the program will be refered to as *main*0.

2.2.2 Solution 1: Omit the `omp single` directive

Because we cannot launch parallel threads in an omp single region, replacing the `omp single` directive with `if else` control statements allows the rest of the loop to be parallelized. This technique is shown in Listing 2.

This version of the program will be refered to as *main*1

Listing 2: code snippet of *main*1

```
// modified version 1 (main1)
#pragma omp parallel
{
   while(...) // looping through N positions
   {
      if(rank == 0)
      {
         // sequence selection
      }
      else
      {
         // other threads wait for main thread to finish
      }
      #pragma omp barrier
      // The rest of the loop can benefit from omp parallelism
   }
}
```

One drawback of this solution arises in the strong dependency among the threads. Using `if else` statements and `omp barrier` synchronization results in idle threads waiting for each other to finish, causing low CPU Utilization (figure 2.4). Despite the thread dependencies, main1 is still able to run three to four times faster than main0 at 16 threads (figure 2.5).

2.2.3 Solution 2: Selecting multiple sequences in parallel

In order to reduce thread dependency, we made a change to the algorithm to parallelize the sequence selection process. We'll refer to this version as *main*2.

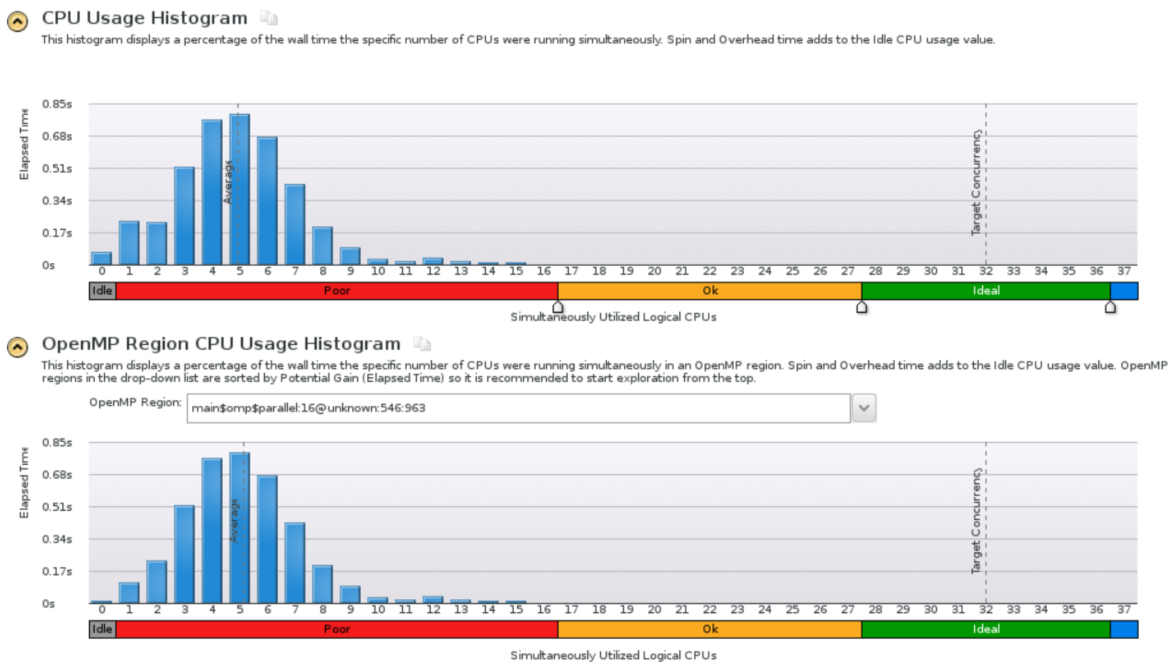
Listing 3: code snippet of *main*2

```
// modified version 2 (main2)
#pragma omp parallel
{
   while(...) // Loop can be divided among the threads
   {
      // selecting sequences according to the number of threads
      #pragma omp single
      {
         // making sure selected sequences are distinct
      }
      // The rest can be done in parallel
   }
}
```

One crucial part of this algorithm is making sure that the selected sequences are distinct. We can achieve this by sorting the array of generated sequences and checking whether adjacent elements are equal.

*main*2's performance (figure 2.3) is slightly worse compared to *main*1 when running at fewer threads due to the overhead of generating distinct sequence. With higher utilization of the CPU, *main*2 is able to close the performance gap as more threads are available to the program.

However, *main*2 provides an algorithm that is more suitable for GPU acceleration APIs such as CUDA and OpenCL. In addition, we have to design a more reliable algorithm for generating N distinct sequences. The algorithm described above could make the program stall when generating a large number of sequences ($> 10^4$) for each GPU thread.



CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

OpenMP Region CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously in an OpenMP region. Spin and Overhead time adds to the Idle CPU usage value. OpenMP regions in the drop-down list are sorted by Potential Gain (Elapsed Time) so it is recommended to start exploration from the top.

OpenMP Region: main$omp$parallel:16@unknown:546:963

Figure 2.4: A graph showing CPU utilization of *main*1 when 16 threads are available. The average usage of CPU is about 5 threads.

2.2.4 speedup Results

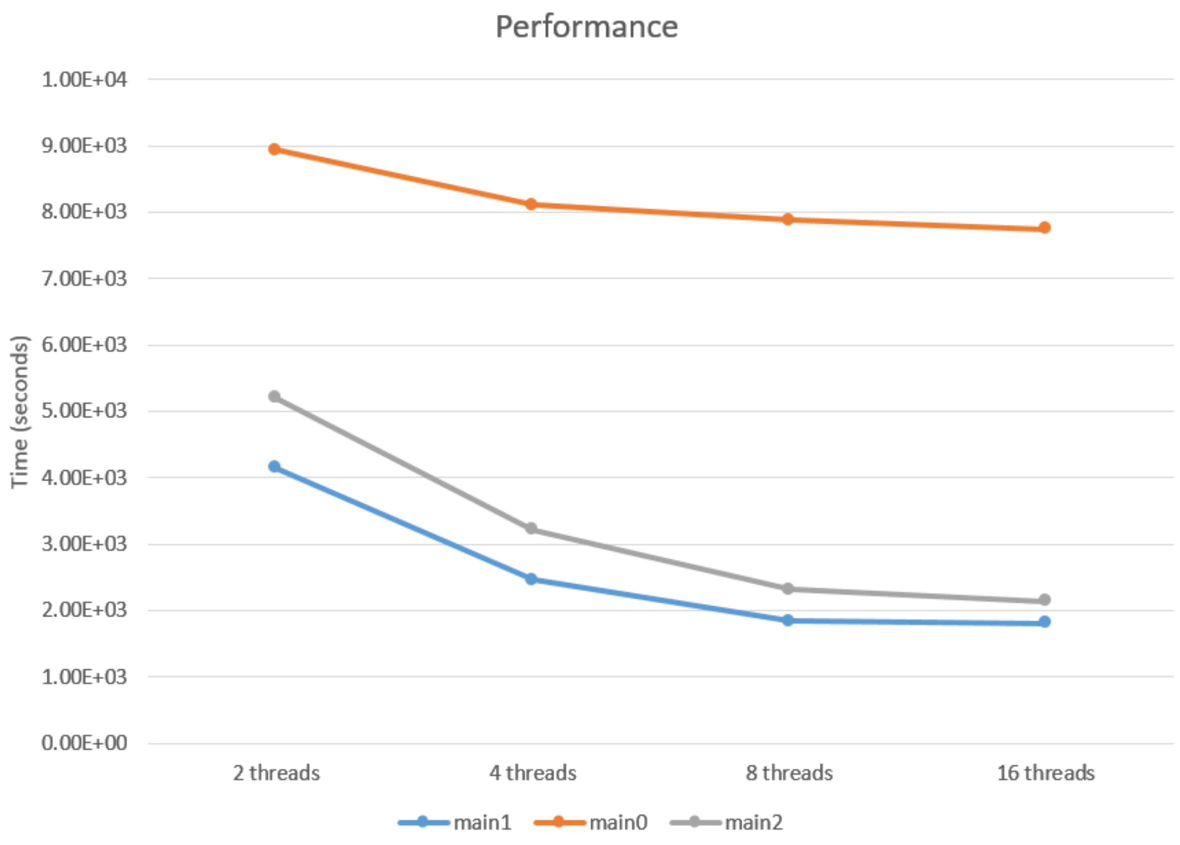


Figure 2.5: Performance comparison of three different solutions.

We're able to achieve three to four times speedup using only CPU optimization. With the new algorithm demonstrated in *main*2, we expect a GPU port to reach even better performance compared to the CPU version.