Dillon Tidgewell

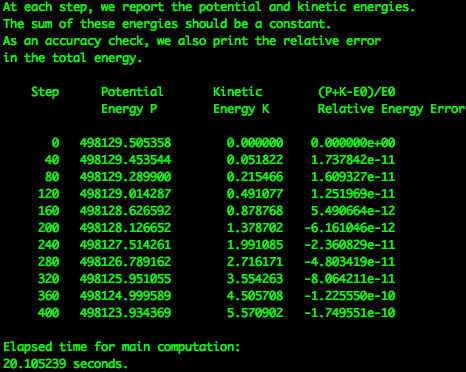
Professor Rene German

CPSC 445

18 January 2018

Molecular Dynamics in OpenMP

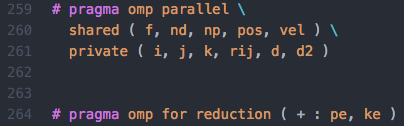
Some of the most complex programs that we run today are simulations of real-world phenomena. These programs are complex because they require a vast amount of data and an incredible amount of computing power. One way that we can increase the amount of computing power available is to make more efficient use of the hardware that we already have available. OpenMP, when wielded properly, can help developers achieve a significant increase in efficiency. OpenMP is an Application Programming Interface used to implement multi-threaded, shared-memory parallelism through the use of compiler directives, runtime library functions, and environment variables.

To display and analyze the use of OpenMP to increase efficiency through parallelism, I will use a relatively simple simulation of molecular dynamics. This program, *md\_omp.c*, which I will refer to as MD, comes from a list of examples of parallelized programs on John Burkhardt’s website, which can be found here: <https://people.sc.fsu.edu/~jburkardt/c_src/md_openmp/md_openmp.html>. MD simulates particles interacting with a “central pair potential”, or function that described potential energy, and outputs the potential energy, kinetic energy, and relative chance of error at a specified number of steps through the simulation. To compile MD, I enter “g++-7 –std=c++11 –fopenmp md\_omp.c –o md\_omp”. The “-fopenmp” flag is what tells the compiler to be aware of OpenMP compiler **directives and library functions within the code. To run, I enter “./md\_omp”. A sample of the output of MD (Fig. 1) can be seen below.

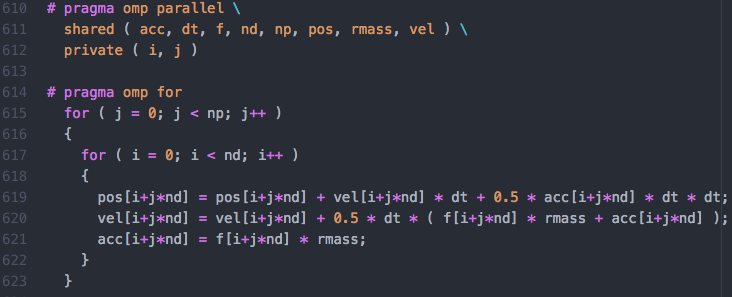
The majority of this program is run serially. MD consists of the methods *main*, *compute*, *dist*, *initialize*, *r8\_uniform\_01*, *timestamp*, and *update*. The two methods that use parallelism are compute and update. The rest are run serially. I will briefly discuss those serial methods first.

**Figure 1**

In *main*, all relevant variables are declared and output is handled. The line “proc\_num = omp\_get\_num\_procs ( );” is used to determine how many processors are available and store them in the variable *proc\_num*. This line is an example of a runtime library function in the OpenMP API. OpenMP will use this to create an appropriate number of threads for the parallel blocks of code. In *dist*, the distance between the two particles is calculated. In *initialize*, variables like position, velocity, and acceleration are initialized. In *r8\_uniform\_01*, a pseudorandom number is generated and returned. Finally, *timestamp* is used to measure the computation time of the program.

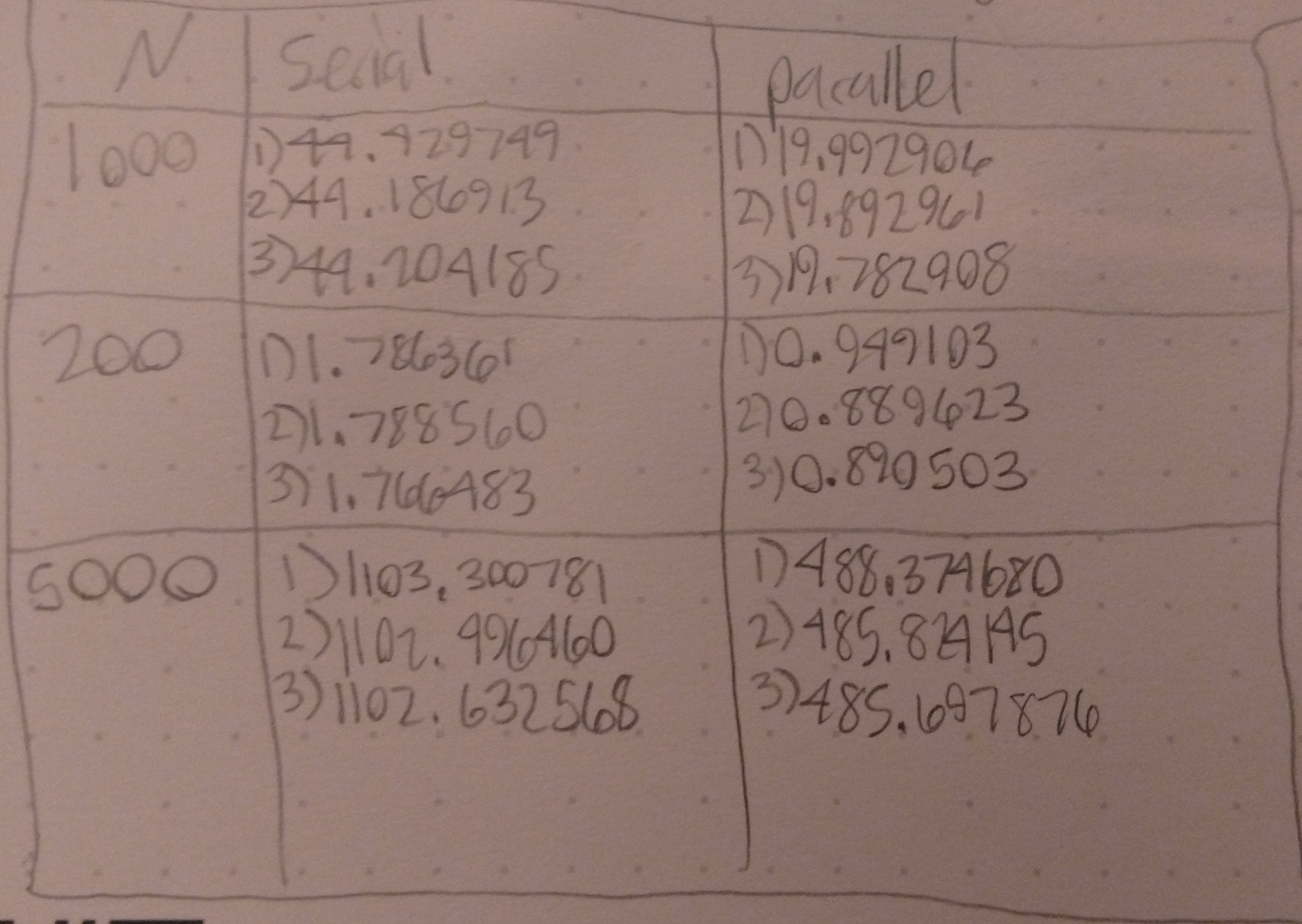
The first method that utilizes parallelism that I will discuss is *compute*. This method is composed of a series of computationally expensive nested for-loops which computes the potential and kinetic energy according to the “potential function V(X)…= ( sin ( min ( x, PI2 ) ) )\*\*2” and other math magic that we do not need to fully understand for the purpose of this paper. Figure 2 shows the compiler directives that are used to parallelize this method. These four lines are added just before the beginning of the for-loop section. The line “# pragma omp parallel” (Line 259 in Figure 2) is the basic way to declare that the upcoming section of code should be run in parallel. The # symbol denotes that this is a compiler directive. The term “pragma” indicates that the compiler should use “pragmatic or implementation-dependent features”. The phrase “omp parallel” indicates that OpenMP will be used to run the block in parallel. The variables listed in parentheses following “shared” will be shared by all of the threads that are created to run the parallel block. The variables listed after “private” will be private to each thread that is created. Line 264 shows the directive used to run a for loop in parallel. This works by splitting iterations of the for loop between the threads that were created by Line 259. The “reduction” part indicates that the variables in parentheses, pe and ke, will have a local version created for each thread that is running, and these local variables will be added together at the end of the block to give the value of pe and ke, respectively. This is done in order to avoid a race condition, which is a risk when multiple threads are accessing the same variable. I have not included the code that follows in the image because it would take up a whole page and is not that relevant.

**Figure 2**

The second method that utilizes parallelism is *update*. The contents of *update* can be seen in Figure 3. This method updates the position, velocity, and acceleration of each of the particles in the simulation. The OpenMP directives used in this method are not much different from those used in the *compute* method, save for the fact that this method does not use the reduction flag in the for-loop directive.

**Figure 3**

According to Amdahl’s Law, the theoretic gains in performance, referred to as speedup, should be where f is the fraction of instructions that are run serially and p is the number of processors that are used. Speedup in this case ranges between 1 and p. There are 626 lines of code in this program, 62 of which are run in parallel, 207 of which are run serially, and 357 of which are comments or blank lines. Dividing the serial portion of instructions by the total number of non-comment lines, 269, we find that f is equal to .7695. The number of processors I am using is 4. So, Amdahl’s Law provides a theoretical speedup of .

Theoretical gains are great, but I want to know the actual speedup that parallelism is enabling. To find this, I ran the program and measured the average running times for the simulation using 200 particles, 1000 particles, and 5000 particles. Then, I removed the OpenMP directives and ran the program serially, again measuring the average running times for the previous particle numbers listed. My measurements can be seen in Figure 4 on the next page. It is important to note that speedup can be measured using the equation

. Speedup in this case ranges between 0 and p. When N, the number of particles, is 200, the average serial running time was 1.7805 seconds compared to an average parallel running time of 0.9097 seconds. When N is 1000, the average serial running time was 44.2736 seconds compared to an average parallel running time of 19.8896 seconds. When N is 5000, the average serial running time was 1102.8099 seconds compared to an average parallel running time of 486.6322 seconds. When N is 200, . When N is 1000, . When N is 5000,

Figure

From the above tests, we can see that the benefits of using OpenMP to parallelize our code are clear. Even at a relatively low level of computations, the program ran almost twice as fast. It is interesting to note that as N grew, the speedup actually increased as well. The implications this has for larger-scale programs is immense. OpenMP is wonderful for its ability to allow programmers to easily run computationally expensive programs.