Implementing Astrophysical Modeling Pipelines with CPU and GPU Parallelization

Final Project Report for PHYS 244, UC San Diego, Spring 2021

Eltha Yu-Hsuan Teng

1 Introduction

Astrophysicists often compare observational data with theoretical models to study the physical properties of astronomical objects. For instance, theoretical models can be a large grid of predicted fluxes under various combinations of input physical parameters, and by comparing with the observed fluxes, the parameter sets with good fits can be found. These model grids are usually large, as it needs to cover a wide range of physical conditions and provide high enough resolution to distinguish between various conditions.

However, simply constructing the model grid and fitting with data includes multiple iterative steps: (1) entering all combinations of input conditions, (2) computing with equations and theories, (3) reading outputs and assigning modeled values to grid, and (4) least square fitting and/or calculating probabilities – all these steps had to be done iteratively. By parallelizing these steps, the computational time of modeling should be significantly reduced, and thus leaving more time for further scientific analyses.

In this project, I develop both serial and parallelized programs for the four steps mentioned above. Parallelized programs are implemented with multiple OpenMP threads and OpenACC. I analyze the performance and speedup among different implementations. To further decouple the contribution from each sub-procedure, the runtimes and speedup of the individual steps are evaluated. I also compare the results between different input grid sizes.

2 Methodology

2.1 Three-Dimensional Flux Model

As an example, I model the line fluxes in the 1-0 rotational level of CO under various combinations of H_2 density (n_{H_2}), kinetic temperature (T_k), and CO column density (N_{CO}). I utilize a public radiative transfer code, RADEX [1] to obtain the predicted CO 1-0 line fluxes under different physical conditions. The input of RADEX are text files specifying a molecular data file and the values of n_{H_2}, T_k, and molecular column density. The molecular data file I use (*co.dat*), which includes information of the energy levels, statistical weights, Einstein A-coefficients, and collisional rate coefficients of CO, are from the Leiden Atomic and Molecular Database [2].

As shown in the source codes, I construct a three-dimensional model grid with $\log(n_{H_2})$ varying from 2 to 5 in steps of 0.1, $\log(N_{CO})$ from 16 to 20 in steps of 0.1, and T_k from 10 to 100 K in steps of 5 K. This gives a grid size of $N = 19 \times 41 \times 31 = 24149$, which is actually a few orders of magnitudes smaller than a realistic astrophysical modeling with more free parameters or finer resolutions, but it should be sufficient for the purpose of this project. For comparison, I also generate a coarser grid with a size of $N = 19 \times 21 \times 16 = 6384$ by doubling the step sizes of $\log(N_{CO})$ and $\log(n_{H_2})$ to 0.2 dex. All source codes are written in C++. The serial, OpenMP, and OpenACC codes for a grid size of $N = 24149$ are *main.cpp*, *main_omp.cpp*, and *main_acc.cpp*, respectively. The corresponding codes for the coarser $N = 6384$ grid have an additional *_coarse* in the filenames.

2.2 Program Structure

As introduced in Section 1, this program combines the four essential steps that must be done iteratively. I wrap the procedures of each step into a function and run them with for loops. In other words, there is a corresponding function to each step, and these functions will be run iteratively in the main program. I will explain how each step works in the following paragraphs.

The first step requires creating multiple input files and writing in the values of varying parameters, as RADEX will read in the information given in the input files one by one, do the calculations, and then create corresponding output files that contain predicted values of several observables. The corresponding function for step 1 is named as *write_input*. It converts the index of the model grid to a specific combination of parameters over our 3D parameter space, generate a corresponding input file, and then write the required information into the file.

In the second step, RADEX is executed iteratively to obtain theoretical predictions for every parameter set. The corresponding function, *compute*, converts the grid indices to the corresponding input file names, and then outputs the required commands to run RADEX with each input file. The third step is mainly implemented by the function, *read_flux*. It reads in the output files from step 2, extracts and returns the predicted CO 1-0 line flux. The main program then assigns the returned values into an array, in order to construct the model grid for further analysis with data.

Unlike the first three steps being implemented for model construction, the final step performs basic analysis with observational data. For this step, a χ^2 fitting and probability analysis is implemented with the function *fitting*, which first calculates the χ^2 values for each grid point [3]

$$\chi^2(\theta) = \left(\frac{S^{\text{mod}}(\theta) - S^{\text{obs}}}{\sigma} \right)^2 \tag{1}$$

where $S^{\text{mod}}(\theta)$ is the modeled flux at $\theta = (N_{CO}, T_k, n_{H_2})$, S^{obs} is the observed flux from data (which I randomly assigned a value for this project), and σ is the uncertainty for which I simply assumed as 10% of the flux. Then, the *fitting* function converts χ^2 values into relative probabilities by assuming a multivariate Gaussian probability distribution [4]:

$$P(S^{\text{obs}}|\theta) = \exp\left[-\frac{1}{2}\chi^2(\theta) \right] \tag{2}$$

I record the run times of all four steps using the high-resolution clock defined in the *chrono* time library, having a precision in microseconds. All the recorded times are written into an output log file (i.e. *main.log*, *main_omp_x.log*, ..., etc., where x is the number of OpenMP threads used). Other information written into the log files include the number of processors and/or threads available.

3 Results

To verify the correctness of our paralleled programs using OpenMP and OpenACC, I printed out the values of the constructed model grid and probability grid returned from steps 3 and 4 and have confirmed that the values are consistent with those from the serial program. Section 3.1 shows the speedup and analysis for all the steps using the serial, OpenMP, and OpenACC codes with a model grid size of $N = 24149$. Results for the coarser $N = 6384$ grid and comparison with the $N = 24149$ grid are presented in Section 3.2.

3.1 Speedup and Performance

The OpenMP codes are run with 1, 2, 3, 5, 10, 20, 30, 40, and 50 threads, respectively. Figure 1 shows how the execution time for each step varies with the number of OpenMP threads used. The execution time of the serial code is overplotted as the blue dashed line, and that of the OpenACC

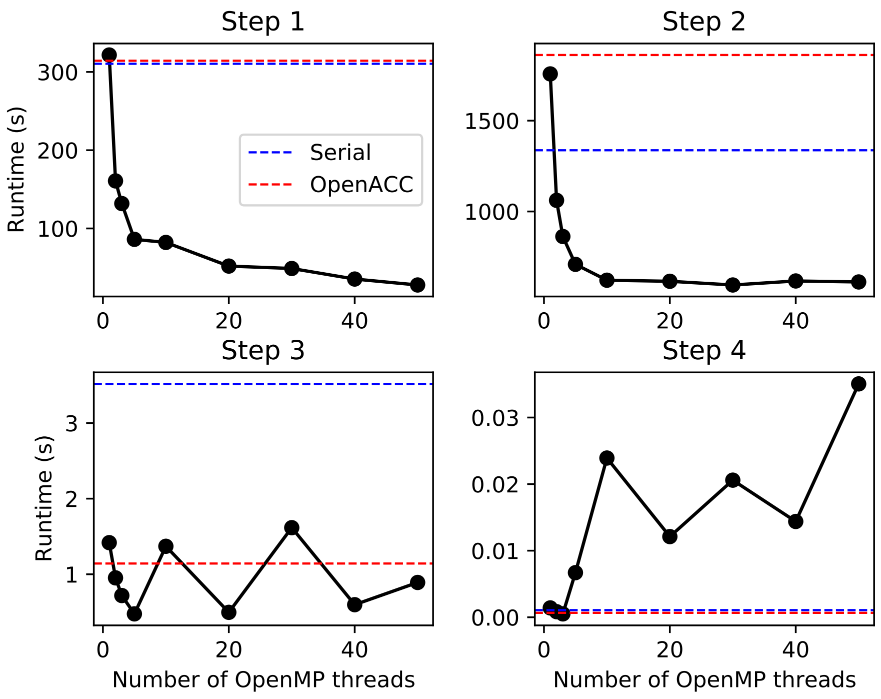

Figure 1: Relation between execution time and number of OpenMP threads for each step. The red and blue dashed lines indicate the execution time of the serial and OpenACC code, respectively.

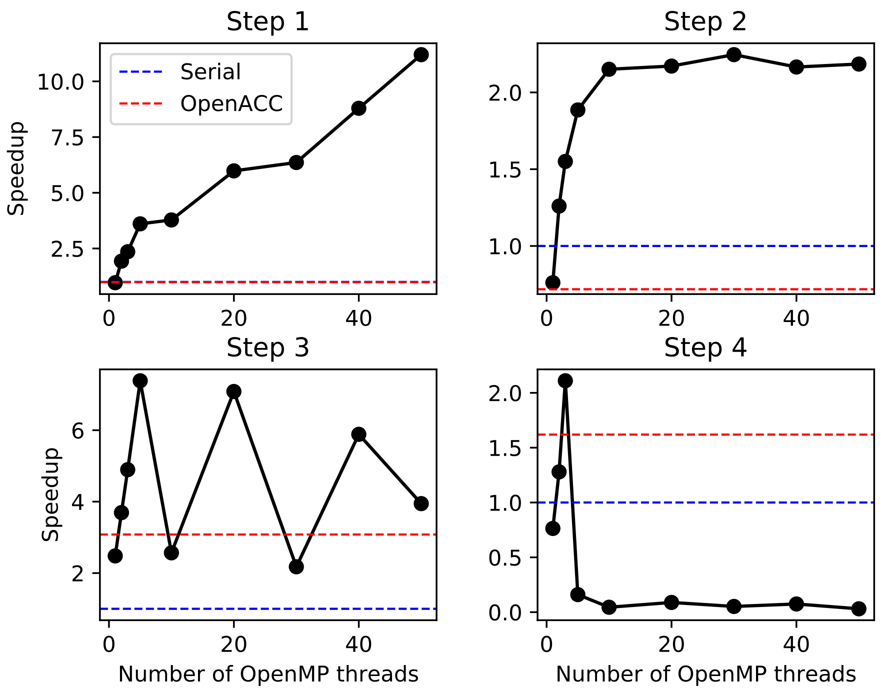

Figure 2: Relative speedup to the serial code varied with the number of processors used.

code using 1 GPU is shown by the red dashed line. Similar plots showing the speedup relative to the serial code (defined as speedup = 1) are presented in Figure 2. In general, we can see that running with only 1 OpenMP thread takes more time than running with the serial code. This is expected, because working on a single CPU thread is equivalent to a serial code, while it needs additional time to initiate OpenMP.

In steps 1 and 2, it is clear that the speedup increases with the increase of working processors. However, the speedup becomes slower or saturates after reaching 10 threads, which is likely due to resource redundancy. For steps 3 and 4, we do not see clear speedup when using more threads, and the maximum speedup are both at $\sim$5 threads, which may indicate that resource redundancy happens earlier due to rather simple tasks. As seen from the very short runtimes, steps 3 and 4 are very simple tasks, so the waiting time for initiating concurrent workers could be dominant in such a short timescale. Also, the tasks could be bandwidth-limited rather than CPU-limited. These are all possible reasons for why the speedup is not evident.

Another interesting thing to notice is that in the first three steps the OpenACC code works even worse than both serial and OpenMP codes. It was beyond my expectation at first, but then I realized that the first three steps include intensive input/output (i/o) processes that require frequent communication with the disk memory. Therefore, there has to be frequent data transfer between CPU and GPU, making the whole process inefficient. On the other hand, step 4 is purely computational, namely, no i/o processes such as accessing files on the disk are needed. In such case, OpenACC does outperform both serial and OpenMP implementations, although the curve was not perfect due to more time spent on data transfer compared to the very short computational time.

3.2 Comparison with Smaller Grid Size

In addition to the model grid with size $N = 24149$, I did another run with a smaller grid with size of $N = 6384$ by halving the resolution of $\log(N_{CO})$ and $\log(n_{H_2})$. The input dimension is therefore $\sim$3.8 times smaller than the $N = 24149$ grid. Since the time for running step 4 is already short with the $N = 24149$ model, only the first three steps are run and compared using the smaller model with size $N = 6384$.

Figure 3 shows the relation between speedup (relative to serial code) and number of threads for both the $N = 24149$ and $N = 6384$ models. The dashed lines indicate the speedup of OpenACC implementation. In all three steps, we find the trends of speedup curves being very similar. The relation with total time is dominated by the curve of step 2 as it is the most time-consuming step. In step 1, the speedup curve for $N = 6384$ model begins to flatten at $N_{\text{threads}} = 30$, while the curve for $N = 24149$ model continues to increase till at least $N_{\text{threads}} = 50$. Note that I am only showing the results for first three steps which involve intensive i/o processes, so OpenACC does

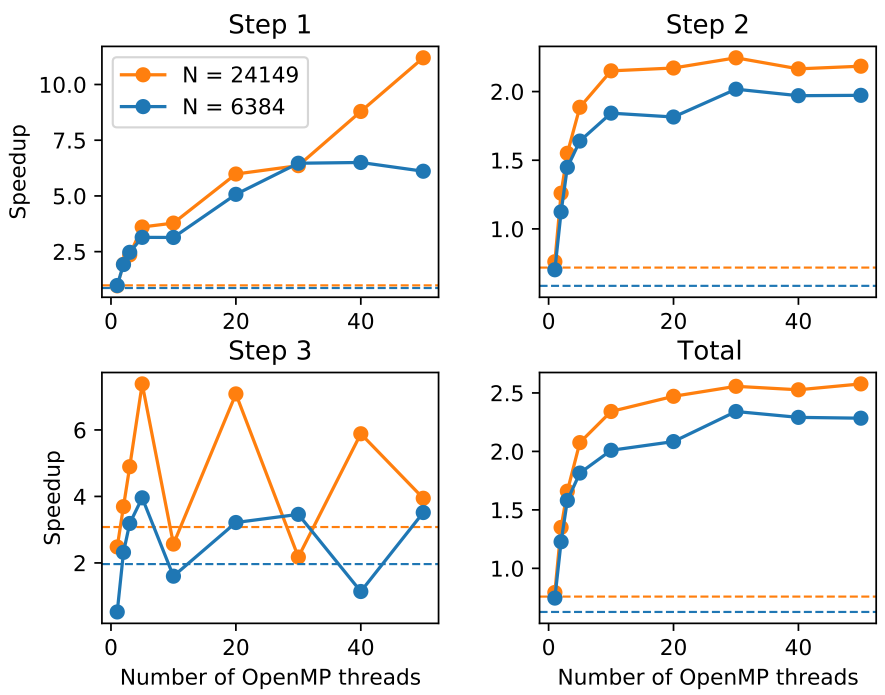

Figure 3: Comparison between the relative speedup (compared to its serial code) for two different grid sizes. Dashed lines show the OpenACC results.

not perform better in these cases (see Section 3.1 for discussions).

On the other hand, by comparing the absolute factor of speedup with the $N = 24149$ model, we can see the speedup for both parallelization methods reduces when run on a smaller grid, possibly because of resource redundancy. Figure 4 presents a comparison between the runtimes with two different grid sizes. It is clear that grid size is proportional to runtime. Since $N = 24149$ is ~4 times the size of $N = 6384$, we also observe similar factor in the growth of their runtimes.

4 Conclusions

I have developed serial, OpenMP, and OpenACC programs of an astrophysical modeling pipeline. Performance and speedup of each sub-procedure are analyzed and compared among different implementations. The main results are summarized as the following:

1. Generally, speedup increases with the number of processors, but may saturate or decline beyond a certain number due to resource redundancy. The execution may also becomes bandwidth-limited rather than thread-limited when there are too many threads. For the pipeline implemented in this project, overall saturation happens at ~10 threads.

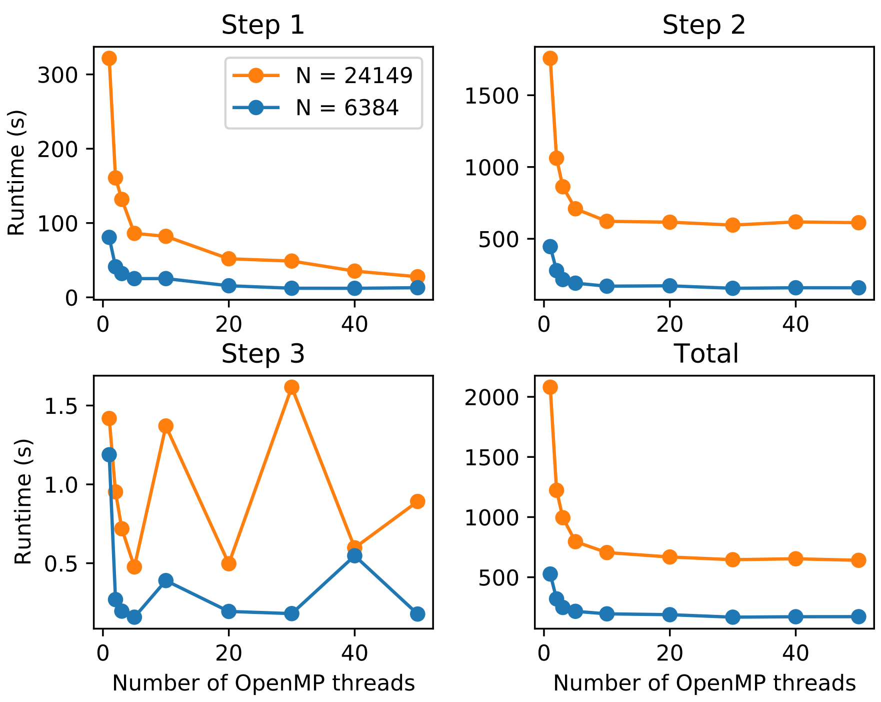

Figure 4: Comparison between the execution time using model grids with different sizes.

2. For simple tasks that can be done within few seconds, we may not observe clear relation between speedup and number of processors, as the time taken by data transfer or requests for workers could be dominant in such a short timescale.

3. The first three steps in our pipeline requires iterative i/o processes for accessing files in disk memory, and thus the OpenACC code using GPU does not show better performance than the OpenMP code due to continuous data transfer between CPU and GPU. On the other hand, OpenACC does perform better in the final step, which is a purely computational task without any i/o processes involved. **This implies that CPU parallelization may be the optimal method for loading/processing data, while GPU parallelization are most suitable for pure computational tasks.** This is consistent with modern Machine Learning pipelines, where data are usually processed via CPU and model training done on GPU (e.g. *Dataloader* in PyTorch or *Dataset* in Tensorflow).

4. By testing with two different sizes of model grid, we observe proportionality between the grid size and execution time. In addition, the relative speedup on smaller grids is also found to be lower compared to that on larger grids. This is likely caused by resource redundancy due to reduced amount of tasks.

References

[1] F. F. S. van der Tak, J. H. Black, F. L. Schöier, D. J. Jansen, and E. F. van Dishoeck. A computer program for fast non-LTE analysis of interstellar line spectra. With diagnostic plots to interpret observed line intensity ratios. *Astronomy & Astrophysics*, 468(2):627–635, June 2007.

[2] F. L. Schöier, F. F. S. van der Tak, E. F. van Dishoeck, and J. H. Black. An atomic and molecular database for analysis of submillimetre line observations. *Astronomy & Astrophysics*, 432(1):369–379, March 2005.

[3] J. Kamenetzky, N. Rangwala, J. Glenn, P. R. Maloney, and A. Conley. A Survey of the Molecular ISM Properties of Nearby Galaxies Using the Herschel FTS. *The Astrophysical Journal*, 795(2):174, November 2014.

[4] Karl D. Gordon, Julia Roman-Duval, Caroline Bot, Margaret Meixner, Brian Babler, Jean-Philippe Bernard, Alberto Bolatto, Martha L. Boyer, Geoffrey C. Clayton, Charles Engelbracht, Yasuo Fukui, Maud Galametz, Frederic Galliano, Sacha Hony, Annie Hughes, Remy Indebetouw, Frank P. Israel, Katherine Jameson, Akiko Kawamura, Vianney Lebouteiller, Aigen Li, Suzanne C. Madden, Mikako Matsuura, Karl Misselt, Edward Montiel, K. Okumura, Toshikazu Onishi, Pasquale Panuzzo, Deborah Paradis, Monica Rubio, Karin Sandstrom, Marc Sauvage, Jonathan Seale, Marta Sewiło, Kirill Tchernyshyov, and Ramin Skibba. Dust and Gas in the Magellanic Clouds from the HERITAGE Herschel Key Project. I. Dust Properties and Insights into the Origin of the Submillimeter Excess Emission. *The Astrophysical Journal*, 797(2):85, December 2014.