Improving the Performance of the Molecular Similarity in Quantum Chemistry Fits

Alexander M. Cappiello

Department of Chemistry
Carnegie Mellon University
Pittsburgh, PA 15213

December 17, 2012

Abstract

This paper discusses a sub-task in the Molecular Similarity in Quantum Chemistry (MSQC) project. The goal of MSQC is to develop new semi-empirical methods for electron structure determination, taking advantage of molecular similarity and machine learning to improve the accuracy of calculations relative to time cost. The work is primarily done using MATLAB (1). At this point, fits have been successfully run for small hydrocarbons. The focus of this paper is on methods for improving the time cost of these fits. This falls into two primary categories: altering code to utilize MATLAB's parallel/distributed computing tools and identifying inefficient pieces of code that can be replaced. Improvements in both areas have been significant. Timed runs have been run both for small test cases and for long fits.

Introduction

Generally, electron structure calculations can be thought of as a compromise between the accuracy of the result and computer time required to produce it. Since this time tradeoff is significant and measurable, calculations on larger systems become increasingly unfeasible. The goal of the MSQC project is to develop new methods of calculation based on similarity between a given a fragment in changing environments (2). More specifically, this is done by taking a molecular fragment in a broad range of charged environments (the training set) and determining a set of parameters that describe the molecule. These parameters can then be applied to the fragment in a new environments (the test set). The parameters are used to adjust the data from the low level calculation on this environment to give a more accurate result.

However, these fits are still very time consuming. Therefore, multiple methods of decreasing the runtime has been considered. The first is to incorporate aspects of MATLAB's parallel/distributed computing tools (3). This enables us to take advantage of modern multicore CPUs or distributed clusters, such as Blacklight at the Pittsburgh Supercomputing Center. this does not decrease the overall computational cost, but spreading out the computation improves the overall runtime. The other strategy is to locate inefficient blocks of code that can be replaced. Extensive use of MATLAB's profiling tool was used to locate time consuming sections of code. The code is then replaced with either an equivalent MATLAB function or a small piece of code written in C. Both strategies have

been successful in significantly reducing the runtime of fits.

Code referenced in this report is publicly available in its most up to date version at `https://github.com/acappiello/msqc` and as a persistent version of the code that will not change from its current state `https://github.com/acappiello/msqc/tree/cce6b155a557908dde055fa552b3d5edb1c2944f`. For example, `@Model3/hartreeFock.m` refers to the file `hartreeFock.m` in the `@Model3` folder.

Method and Analysis

The adaptation of the code to run in parallel is done using MATLAB's Parallel Computing Toolbox. The toolbox provides a simple construct to convert serialized computation in a loop into a distributed computation over multiple processes. In the code, this is done simply by replacing a `for` loop with a `parfor` loop. Computation is then distributed across multiple MATLAB processes, which are started by a starting a `matlabpool`. While this may seem very simple, initial attempts did not achieve the desired performance increase. Theoretically, going from one core to two cores, for example, would halve the runtime. However, in practice there is always overhead associated with the computation that prevents this from being realized. Since MATLAB handles parallel computation by using separate processes, moving data between processes becomes the primary source of overhead. This was overcome by saving the data to a file in advance of the parallel computation and then loading the file as part of the parallel computation. At this point, a greater percentage time improvement was seen. The code that handles this can be found in `@Fitme/err.m`.

Rewriting code to be more efficient is significant because it actually decreases the amount of work done, as opposed to spreading it out over more processors. As mentioned previously, MATLAB's profiling tool is used to locate time consuming sections of the code. The profiler simply analyzes code that is run normally. Afterwards, it shows data how much time is spent in certain functions or on certain lines in the code. One cause of inefficent code is simply not knowing the cost of MATLAB's built in functions and matrix operations. MATLAB is a fairly high-level programming language, making it hard to predict how efficient code will be when writing it. Therefore, it may be possible to write equivalent code that runs faster. One example of this is where we need to convert a 4-dimensional matrix with two singleton dimensions to a 2-dimensional matrix. The old code used the `squeeze`

function, which needs to determine which dimensions to remove before doing so. Since the structure of the matrix is already known, this can be replaced with a call to `reshape` to have the same effect. This was implemented in `@Model3/hartreeFock.m`.

If a MATLAB-based alternative cannot be found, MATLAB allows for the integration of code written in C, C++, or FORTRAN to be integrated with MATLAB code. Specifically, C has been used to replace code that could not be otherwise improved. In contrast, C is a low-level language, meaning that it can be written with more informed knowledge of CPU and cache efficiency. This strategy was used in our code where Hartree-Fock is solved. Specifically, in the calculation of the 2-electron components of the Fock matrix, where a large number of matrix sums are taken. The code for this is in `twoElecFock.c`, which is called from `@Model3/hartreeFock.m`. By optimizing the code for cache efficient array accesses and symmetry of the matrices, the speed has been significantly improved.

Measuring success is very simple and can be done by running either a short piece of test code or a full fit and timing how long it takes to complete. Tests are run both in parallel and nonparallel, because changes may behave differently in each case.

Results and Discussion

The first set of tests were run on a short test, written specifically to test runtime, `parallelTime2.m`. These tests were run on a Linux timeshare, `unix12.andrew.cmu.edu`, with an Intel X5460[1]. Results are shown in Table 1. This set of times were not run on the most recent code. The parallel code has been improved significantly since. However, it shows how effective replacing inefficient code has been.

[1]Datasheet: `http://ark.intel.com/products/33087/Intel-Xeon-Processor-X5460`

Table 1: Runtime for parallelTime2.m

Pool Size	Time (s)	Ratio[1]	Old Time (s)	ΔT	Speedup (%)[2]
1[3]	53.16	1.00	81.58	28.42	35
2	43.93	1.21	58.56	14.63	25
3	39.50	1.35	49.97	10.47	21
4	37.22	1.43	47.52	10.30	21
6	34.95	1.52	46.41	11.46	25
8	34.12	1.56	46.06	11.94	26

[1] Calculated as $53.16/T_{new}$.
[2] Calculated as $(T_{old} - T_{new})/T_{old} \times 100\%$.
[3] A pool size of 1 means that no matlabpool is started and the code is running nonparallel.

These results reflect our early struggles to write parallel code with a significant speed improvement. As shown, at best the code only runs up to 56% faster with many cores. It is important to note that adding more to the pool gets diminishing returns. These both indicate the high communication overhead that wastes time. However, the improvements to inefficient code are significant, with up to a 35% speed improvement. Given the time that fits take to run, such an improvement is significant.

Another set of tests was run on a long series of fits set up in `multihybrid1.m`. These were run on a computer with an AMD Phenom 9950 processor[2]. Results are shown in Table 2. These runs reflect the most up-to-date parallel code and efficiency improvements.

Table 2: Runtime for multiHybrid1.m

Pool Size	Efficiency Improvements	Time (s)	Parallel Ratio[1]	Efficiency Ratio[2]
1[3]	no	181192	–	–
4	no	47156	3.84	–
1[3]	yes	105281	–	1.72
4	yes	29783	3.53	1.58

[1] Over nonparallel run with same efficiency settings.
[2] Over prior calculation of same pool size.
[3] A pool size of 1 means that no matlabpool is started and the code is running nonparallel.

These results show that the current parallel code runs very well, running more than 3.5 times faster with a pool of size 4. This is about as good as one can hope for in practice.

[2]Datasheet: `http://products.amd.com/pages/DesktopCPUDetail.aspx?id=447`

The other efficiency improvements remain substantial. It makes sense that the ratios are somewhat smaller in the run with both parallel code and efficiency improvements compared to where only one or the other is running. It turns out that these major improvements to inefficient code appear inside the parallel code, so less time is spent in the parallel code. This means that the remaining communication overhead is magnified. If both ratios are combined, the combined improvement from both techniques is over 5.5.

Summary and Conclusion

In summary, efforts to improve the efficiency of the MSQC code have been very successful. The current version of the parallel code shows a runtime improvement that shows that any communication overhead has been minimized and scales well on a 4-core CPU. The next step will be to ensure that this continues to scale well in a larger distributed environment with more CPUs. The current target for this is the Blacklight Supercomputer. The data presented previously suggests that the parallel code is as effective as it can be, so work here may be more or less complete. Other improvements to inefficient code has been very successful, but also leaves room for improvement. This is because there are other sections of code with a high runtime that have not been investigated yet. Based on the success so far, it is likely that the efficiency of the code can still be significantly improved. Having learned from these improvements, it may be worthwhile to put more thought into writing efficient code from the start when adding new features, rather than going back afterwards to make changes.

References

[1] MATLAB, version 7.12.0.635 (R2011a). MathWorks: Natick, MA 2012; (accessed 12/16/2012).

[2] David Yaron. Overview [Word Document]. 2011.

[3] MATLAB Parallel Computing Toolbox. http://www.mathworks.com/products/parallel-computing/. MathWorks: Natick, MA 2012; (accessed 12/17/2012).