Pre-Parallelization Exercises in Budget-Constrained HPC Projects: A Case Study in CFD

Shamsheer Ahmed, Suma Bhat, Mohammed Isham, Waseem Ahmed, Ramis M. K.
P. A. College of Engineering, Mangalore, India

ABSTRACT
Projects associated with the Grand Challenge Applications (GCAs) often involve large multi-disciplinary teams, are well funded and have access to good computational resources. The code base used in these projects is mature and well maintained and may have gone through multiple revisions spanning decades. Parallelization of this serial code to enable execution on a distributed multi-computer architecture or a shared memory multi-processor system is the next immediate step.

Parallelization of serial code used by young researchers working on GCA-related applications in privately-funded institutions, on the other-hand, is not as straightforward. These researchers work under tight budget and resource constraints and do not have much access to funds or experienced programmers as their other counterparts. Initial findings from a case study are presented that show how such limitations can be alleviated by inter-departmental collaboration involving undergraduate students' final year projects. Code developed by a single programmer over a period of about three years for the Conjugate Heat Transfer problem in Computational Fluid Dynamics (CFD) has been used for the study.

Categories and Subject Descriptors
D.1.3 [**Software**]: Programming Techniques, Concurrent Programming

General Terms
Parallel Programming, Parallel and Distributed Computing

Keywords
Parallelization, Parallel Programming, CFD, HPC, Cluster Computing

1. INTRODUCTION
A category of applications, known as the Grand Challenge Applications (GCAs) [6], need extremely high performance and high memory computers for successful execution. Serial code written for such applications first needs to be parallelized before it can be executed on a multi-computer or a multi-processor system. These applications fall in the scientific domain, and area in which the knowledge of an average computer science graduate may be limited. This exercise in parallelizing the serial code, thus, becomes extremely challenging and time consuming as it requires the programmers to have more than a working knowledge of the domain.

Parallelization of code is a mature field of study and there have been various approaches presented in literature to address it in different ways [1, 2, 3, 4, 7, 9, 10]. A characteristic of teams using such approaches is that they use a fairly mature software base to initiate the parallelization process. The software, in most cases, is many decades old and has been developed by a large multi-disciplinary team [5]. The parallelization process can begin using this correct and, in most cases, compact, easy to comprehend, well written and well documented code. The research team also has substantially large funds available at their disposal to purchase or develop proprietary software like automatic parallelizing compilers, profilers and debuggers and for hiring experienced programmers to aid their parallelization process [5].

There is another category of researchers involved with GCAs who work on similar real world research problems. However, unlike their counterparts, they do not have access to large funds or other basic High Performance Computing (HPC) resources. Left with little alternatives, such researchers make use of free and open source software and COTS components for their research. Additionally, these researchers do not have access to funds that can enable them to hire new developers or purchase expensive software. Working in these tight resource constraints, their research accomplishments can be severely limited.

This paper takes the example of one such case at an undergraduate institution and shows how inter-departmental collaboration can help alleviate some of the problems. Code written for the Computational Fluid Dynamics (CFD) domain by a single developer with a non-Computer Science background has been used as the input to this study. Initial experiences and observations of a group of undergraduate Computer Science students attempting to parallelize the code for parallel execution on a Linux based cluster are presented. The rest of the paper is organized as follows. The next section introduces the reader to the Conjugate Heat Transfer problem in the field of CFD and gives details on the working of the code. Section 3 lists the initial observations. Suggestions for improvement of code and other pre-parallelization exercises are listed in section 4. Section 5 concludes the paper.

2. CONJUGATE HEAT TRANSFER

2.1 Introduction

The term Conjugate Heat Transfer refers to a heat transfer process involving an interaction of heat conduction within a solid body with either of the free, forced, and mixed convection from its surface to a fluid (or to its surface from a fluid)flowing over it. An accurate analysis of such heat transfer problems necessitates the coupling of the problem of conduction in the solid with that of convection in the fluid by satisfying the conditions of continuity in temperature and heat flux at the solid–fluid interface.

There are many engineering and practical applications in which conjugate heat transfer occurs. One such area of application is in the thermal design of a fuel element of a nuclear reactor. The energy released due to fission in the fuel element is first conducted to its lateral surface, which in turn is dissipated to the coolant flowing over it so as to maintain the temperature anywhere in the fuel element well within its allowable limit. If this energy generated is not removed fast enough, the fuel elements and other components may heat up so much that eventually a part of the core may melt. In fact, the limit to the power at which a reactor can be operated is set by the heat transfer capacity of the coolant. Therefore, the knowledge of the temperature field in the fuel element and the flow and thermal fields in the coolant is needed in order to predict its thermal performance.

2.2 Computational Details

Software was developed that deals with the study of conjugate heat transfer problem associated with a rectangular nuclear fuel element washed by upward moving coolant. Accordingly, employing stream function-vorticity formulation, equations governing the steady, two-dimensional flow and thermal fields in the coolant are solved simultaneously with the steady, two-dimensional heat conduction equation for the fuel element using second-order accurate finite difference schemes. Keeping Prandtl number Pr = 0.005 for liquid sodium as coolant, numerical results thus obtained are presented for a wide range of the involved parameters aspect ratio A_r in the range 5 to 30; conduction-convection parameter, N_{CC} in the range 0.2 to 0.4; total energy generation parameter Q_t in the range 0.2 to 0.8 and Reynolds number, Re_H in the range 100 to 10000.

Complete code for the software was developed in-house over a period of about 3 years. Coding involved a single developer with a Mechanical Engineering background who had limited programming and Software Engineering experience. The code was initially coded using Microsoft's Visual C++ IDE and consisted of about 1800 LOC with 31 functions.

3. CODE ANALYSIS

3.1 Parallelization Team

The team undertaking the parallelization task was undergraduate students in the Computer Science department with limited or no knowledge of Computational Fluid Dynamics (CFD) or the code. The pre-parallelization tasks involved porting the code to Linux, code comprehension, profiling, analysis and optimization. While one student worked on the optimization part, two others were involved in profiling, code comprehension and analysis. The subsections below describe the tasks in more detail.

3.2 Porting

As the cluster used was a Linux based cluster, the first step involved porting of the code written in Visual C++ to the Linux platform. This did not present a problem as no Visual C++ specific libraries were used in the initial code and the code compiled and executed without change on Linux.

3.3 Profiling and code comprehension

The students used gprof, Gnu's open source profiler, to profile the code. The output of gprof, which includes a flat profile and a call graph, was also used for code comprehension. The following observations were made by the students

1. Values for the different input parameters were hard-coded into the program. Each change of parameter value necessitated a recompilation of the code.
2. For the given set of parameters present in the code the observed run time was about 23 minutes. The run time could get much larger (ranging from a few hours to days) with changed parameters. The large system run time discouraged experimenting with a range of values of the computational grid system that may have resulted in values with a finer resolution. Additionally, the range of values of parameters that may compute results giving more insight into the physics of the problem could not be studied for the same reasons.
3. Multiple output files were used and data was being written to a large set of output files. File handling could have been more efficiently done to positively impact the total execution time.
4. As the program was serially executed, the execution of a few functions was delayed, even though parameters or values required to compute that function were available. A similar case was of functions being called after the complete execution of loops although no data or control dependencies existed between the loops and/or the functions.
5. It was observed that there was an excessive and sometimes unnecessary use of global variables.
6. Some loops were identified that could have been combined to bring down the size of the code.
7. None of the functions used any input parameters nor did they return any value. Global variables were used as a replacement for both.
8. Some functions that have been defined exhibit identical functionality with few differing statements.

3.4 Analyzing Data Dependencies

The call graph was used to understand the initial working of the

code after which the data dependencies at the coarse function-level granularity were analyzed. More specifically, the following data dependencies were identified.

1. Flow dependence - If the variables modified in one function are passed to another function then the execution must follow the same path.
2. Anti-dependence - If a changed variable in one function is being used by a previously called function. The order of these functions cannot be interchanged.
3. Output dependence - If two functions produce or write to the same output variable they are said to be output dependent; thus their order cannot be changed.
4. I/O dependence - This dependence between two functions occurs when a file is being read and written by both these functions.

Based on the identified dependencies, the code was statically restructured for a theoretical execution on a multi-processor system. Initial analysis indicated a reduction of the execution time to about 13 seconds indicating a theoretical speedup of 1.7 ignoring the communication overhead.

4. PRE-PARALLELIZATION EXERCISES

Based on the analysis, it was noted that the following need to be completed before the start of the parallelization step:

1. Code needs to be changed to read in parameters from the command-prompt or from an externally available input file. This would allow the code to be executed unchanged for different values of the parameters without the need for a recompilation.
2. Reduction in the number of output files. If the files are genuinely required, the output sequence needs to be further analyzed; else the data that is written to these files can be combined into a reduced set.
3. Functions that have been identified with no data dependencies between them are good candidates for parallel execution. Their execution time profiles and the computation-communication ratio needs to be further studied to see if parallelization will indeed produce a speedup.
4. The code needs to be rewritten to reduce the usage of global variables. This may involve changing all or most of the function signatures to read in input parameters and return results. This exercise may also involve the creation of more efficient data structures for parameter passing between functions.
5. Many functions can be eliminated by rewriting functions to combine the functionality of two or more functions. This would considerably reduce the code size and will result in more compact and well written code. However, as noted in [8], code repeated in different programs offers the advantage that it can be customized to execute that part of the program where it lies in a unique manner; combining similar portions of code into a generalized single function, while offering other advantages, removes this advantage [8]. The trade-offs need to be deliberated before performing this exercise.
6. Loops that are temporally close need to be studied along with their indices to see if they can be successfully combined. In addition to reduce the code size, this would reduce the effort of parallelization as only a single loop needs to analyzed.

5. CONCLUSION

A case study was presented that analyzed code for the Conjugate Heat Transfer problem. As the code was developed by a person with a non-Computer Science background with limited programming and Software Engineering experience, modification of the code for parallel execution on a multi-computer or a multi-processor system cannot begin immediately. It was observed that the pre-parallelization exercise involves substantial rewriting of code. Some initial findings by a team of under-graduate Computer Science students have been reported. Static restructuring of the code revealed a theoretical speedup of 1.71

6. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L.Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A View from Berkeley. Technical Report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. Parallel Computing, 24(3-4), 1988.

[3] D. E. Culler, J. P. Singh, and A. Gupta. Parallel Computer Architecture. Morgan Kauffman, 1999.

[4] T. R. Halfhill. Parallel processing with CUDA. Microprocessor Report, Reed Electronics Group,January 2008.

[5] L. Hochstein and V. R. Basili. The ASC-Alliance projects: A case study of large-scale parallel scientific code development. IEEE Computer, 41(3), 2008.

[6] K. Hwang. Advanced Computer Architecture. McGraw Hill, 1993.

[7] W.-M. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm,I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta,M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In DAC '07: Proceedings of the 44th annual conference on Design automation, pages 754–759, New York, NY, USA, 2007. ACM.

[8] D. E. Knuth. The Art of Computer Programming, Volume 1. Pearson Education, Third Edition.

[9] P. Lee and Z. M. Kedem. Automatic data and computation decomposition on distributed memory parallel computers. ACM Transactions on Programming Languages and Systems, 24(1), January 2002.

[10] S.-W. Liao. SUIF Explorer: An Interactive and Inter-procedural Parallelizer. PhD thesis, Stanford, 2000.