Intent-based Compilation for Heterogeneous Parallel Architectures

Waseem Ahmed
King Khalid University
Abha, Saudi Arabia
waseem@computer.org

Shamsheer Ahmed
Manipal Center for Information Science
Manipal 576104, India
meetshamsheer11@gmail.com

Yajnesh Talapady
P. A. College of Engineering
Mangalore 574153, India
yajnesh_t@yahoo.com

Jickson Periya
P. A. College of Engineering
Mangalore 574153, India
jicksonstephen@gmail.com

ABSTRACT

Although a lot of research has been done in the field of automatic parallelization, the extent of parallelization that can be identified and extracted is still minimal compared to what can be potentially achieved. Thus, the gains in terms of performance improvement are minimum.

Intent-based compilation, an approach presented in this paper to address this shortcoming, is based on identifying the intent of the programmer from the given serial code and generating the best possible implementation for a given architecture. Based on a collection of algorithmic variants the most parallelizable variant for the given architecture is then substituted for the pertinent slices in the identified code to achieve a higher performance gain than can be achieved by using just automatic parallelization. Three very contrasting studies have been used to illustrate the challenges involved in the process. A work-in-progress tool and results obtained are also described. Initial findings gave speedups close to manual parallelization.

KEYWORDS

Software Development Tools and Support, Libraries and Programming Environments, Operating Systems and Compilers

1. INTRODUCTION

With the rapid proliferation of multi-cores and multiprocessors in embedded systems (MPSoCs), PCs, GPUs, game consoles and in the HPC domain, the need for sophisticated software development tools that support these parallel architectures becomes extremely important [21]. As the existing programmer community, in general, has been educated in sequential programming paradigms a revolutionary shift to parallel paradigms and parallel thinking would be needed to cater to the new need of parallel applications for these platforms.

Although this would theoretically be the ideal case, it would require time as many topics in the undergraduate curricula may need to be changed [14] for parallel thinking to become mainstream. The other, albeit temporary, solution that researchers agree on is to allow the programmers to continue to code sequentially as currently been done but support them with sophisticated tools to help parallelize the sequentially written code [12]. Indeed, over the past few decades or more there has been a large volume of work done on automatic parallelization. Although mature, the progress in this field is only minimal [6]. The amount of speedup obtained using existing commercially available parallelization tools as compared to what can be potentially obtained with manual parallelization reflects this.

There are two main reasons that may be attributed to this. The first is that the sequential code is assumed to be coded for the purpose of parallelization. To clarify, an algorithm used in the code that cannot be parallelized or cannot be easily parallelized is not replaced with another more parallelizable algorithm that solves the same problem. Rather, the bottleneck is retained. This is the case with almost all existing parallelization approaches including [17, 13, 10, 6], to name a few. Additionally, if the code is subjected to restructuring (different from just refactoring [9]) before parallelization, it may reduce coupling between functions and components [3] and may aid in exposing more parallelization.

The second reason is that majority of the automatic parallelizers and compilers convert the sequential program into a binary executable format. This leaves very little for the parallel developer to work on in case additional parallelization could be manually extracted.

Intent-based compilation is a methodology presented in this paper that addresses the above shortcomings. The next section briefly describes the methodology. The third section illustrates the challenges faced when using intent-based compilation with the help of three case studies. The work-in-progress tool is described in the following section. The last section gives the conclusion and briefly highlights the future work.

2. INTENT-BASED COMPILER
2.1 Introduction

To parallelize high-performance systems, both conventional data-flow optimizations and high-level transformations are necessary to improve parallelism and memory hierarchy performance. Currently, there are two main choices for a developer to choose from – automatic parallelization and manual parallelization. As mentioned earlier, automatic parallelization has thus far not been successful at extracting scalable parallelism from general programs [6]. The other option, manual parallelization, involves time consuming hand optimizations performed for a specific number of cores and for a specific platform. Intent-based compilation (IBC) is a meet-in-the-middle strategy. It is not a radically new compiler methodology but rather involves the introduction of two additional, but essential steps, to the automatic compilation technique. The main premise of intent-based parallelization is that a given code or algorithm is not assumed to be 'the perfect fit' for the given architecture. To illustrate, consider the following case. Conventional approaches attempt to parallelize a given algorithm by extracting explicit parallelism, efficiently utilizing the memory hierarchy and various other techniques. However, if the given algorithm is embarrassingly serial, even the best of the available parallelizing compilers would fail to achieve any speedup. An intent-based compiler instead seeks to understand the intent of the given algorithm - what is the program attempting to do or what is it trying to solve? Once the intent is identified and with a given library of algorithms that can achieve the same goal, a more parallelizable alternative is substituted for the existing one. The next sub-sections explain the methodology in more detail.

2.2 Methodology

The intent-based compilation process comprises the following main steps

1. Identification of programmer intent in code.
2. Program restructuring.
3. Processing compiler directives (if any).
4. Identification of explicit parallelism.

The third and fourth steps are invariably used in existing automatic parallelization techniques and not unique to intent based compilation and, thus, will not be described in this paper. Intent-based compilation, in addition, introduces the first two steps to the compilation process. Again, these steps are not totally unique and may be implicitly performed manually during hand-parallelization of code. Intent-based compilation explicitly include these steps in the compilation process and attempts to automate them. The first step, i.e. identification of programmer intent in code, comprises three sub-steps. The first is the identification of programmer intent in the sequential code. Identification of programmer's intent is an exercise in pattern matching based on a collection of sequential design patterns, or dwarfs as used in [4], and its variants that commonly occur in HPC applications. Correctly identifying intent in code by matching it against a library of algorithmic variants is a very challenging and an important step. Once a match is identified, the next sub-step involves the selection of the most optimal algorithm from a set of *architectural variants* depending on the available hardware architecture. The last sub-step involves the substitution of the algorithm or portion of code for the existing portion. This involves the identification of the pertinent *slices* [16] in code related to that algorithm that need to be replaced and or modified. This is further explained in section 3.1 using a case study.

The second step is the program restructuring done and is performed considering the entire code. This is an involved step which includes different operations ranging from basic operations like the removal of dependencies and refactoring to more complex ones like re-arrangement of code for effective cache utilization. As completely automating this step is challenging, human decision making and intervention may be required in this step. Two options exist at the end of this step - either the modified code can be parallelized using existing automatic compilers or a code-to-code transformation applied to convert it into one using OpenMP, Message Passing Interface or a mix of both libraries for possible further manual parallelization. The challenges and variations faced during this step are highlighted in sections 3.2 and 3.3 using two separate case studies.

2.3 Pattern Matching

To store, match, capture and understand the intent of the developer or algorithms present in code, a collection of patterns in the form of regular expressions is defined. This *pattern tree* is categorized into a hierarchy with each branch from the root denoting a separate dwarf ([4]) or design pattern commonly used in the HPC domain. Branches from the root may have different depths and breadths depending on their *algorithmic* and *architectural* variants. For example, the dense matrix-matrix or matrix vector multiplication has very few algorithmic variants and will have a smaller depth and breadth while the branch denoting the TSP problem will comprise a large sub-tree with a relatively larger height and breadth as the algorithmic variants in this case are much higher. This is further explained in the next section. Having a well defined and comprehensive collection of both algorithmic and architectural variants for commonly occurring patterns is essential to the IBC tool (described in Section 4)

.

3. CASE STUDY

To illustrate the intent-based compilation process, three very contrasting examples are used as case studies. The first is code used in the study of a problem in Computational Fluid Dynamics. The code used for the study represents a category of applications that have been developed by domain engineers and scientists with little programming and software engineering experience. The developed code, from the software engineering perspective, is 'unfit' for immediate automatic parallelization as-is and needs to be polished and restructured before subjecting it to profiling, bottleneck identification and the rest of the parallelization process. The second is dense matrix-matrix multiplication, an operation that is a basic kernel used in many Grand Challenge Applications (GCA) [11]. It is one of the simplest operations in HPC that can be manually parallelized and different approaches to parallelize it have been presented in literature. The third example used as a case study is the traditional Travelling Salesman Problem (TSP). Although many approaches exist in literature to solve the TSP, it presents a difficult challenge from the automatic compilation point of view. The next three sub sections describe these challenges in more detail.

3.1 Conjugate Heat Transfer

3.1.1 Introduction

A category of researchers working on real world Grand Challenge Applications have limited programming and Software Engineering experience and do not have access to funds that can enable them to hire developers to help them parallelize their software [2]. The paper takes one such example to illustrate why restructuring of code is important in such cases in order to achieve the maximum possible speedup using an automatic parallelizer.

The code used for this study was developed to understand the *Conjugate Heat Transfer* problem associated with a rectangular nuclear fuel element washed by upward moving coolant. The code developed over a period of 3 years is fairly mature and results obtained using the code have been published in various journals in the CFD field. The code had about 1800 LOC with 31 functions and was developed in-house by a single developer with a Mechanical Engineering background and with limited programming and Software Engineering experience.

3.1.2 Parallelization Process

Automatically parallelizing this code was not a good option as the code violated many basic software engineering principles as described in [2]. A few examples are given below for illustration

- Excessive and sometimes unnecessary use of global variables
- Unnecessary separation of loops adding to the size of code
- Functions (methods) without input parameters or return value. Global variables were used as substitute for both purposes
- Multiple functions (methods) with similar functionality and few differing statements
- Excessive use of file input-output operations.

As the code had multiple loops spread across multiple functions the bottlenecks were spread out. Even manual parallelization of such code, without a deep understanding of the domain, would prove to be difficult. Also, the potential to expose maximum parallelism in such an application with an automatic parallelizer would not be much. Such code needs to be first restructured before any attempt is made to parallelize it.

3.1.3 Challenge Factor

Medium to High. As the number of ways an application can be coded is large, even manually restructuring a program to obtain optimized code can become difficult. Automating this for a moderately sized code will indeed be challenging.

3.2 Matrix multiplication

3.2.1 Introduction

Dense matrix-matrix multiplication is an important operation in the field of scientific computation and has been a topic of great interest to computer scientists for over forty years. There has been a lot of work presented in literature to reduce the time complexity of $O(n^3)$ of the traditional brute force matrix-matrix multiplication. Strassen's [20] work initially brought down the complexity to $O(n^{2.807})$. This was followed by Coppersmith and Winograd's algorithm [8] which brought it further down to from $O(n^{2.376})$. For large matrices, however, this still takes an extremely large time when run on single-core processor desktop PCs. Moreover, these algorithms are not trivial to implement and remain obscure among the mainstream programmers who have been educated with the traditional and simple three nested-loop approach. Besides these, there are very few other popular variants for implementing matrix multiplication sequentially.

3.2.2 Parallelization Process

Consider the automatic parallelization of a typical, sequentially coded simple dense $n \times n$ matrix-matrix multiplication algorithm. This algorithm requires n^3 multiplication and n^3 addition, leading to a sequential time complexity of $O(n^3)$. The independence between iterations of the two outer loops makes parallelization simple. Theoretically, with $p \geq n^2$ processors, a parallel time complexity of $O(n^2)$ is easily achievable. For this simple case and given a large number of processors, the theoretical lower bound of $\Omega(n^2)$ can be achieved by any average automatic compiler.

However, when $p < n^2$, an automatic compiler may not be able to achieve the maximum potential speedup. Consider for example, a distributed network of workstations. Given a sequential code, automatically dividing the matrices into appropriately sized blocks or stripes based on the matrix size, the cluster size, cluster topology and processor types available within the cluster while optimally utilizing the multi-level cache hierarchy and optimally balancing the loads between processors and cores is certainly non-trivial. Moreover, for a GPU (NVIDIA) implementation, automatic parallelization of the sequential code would be far from ideal. This is because code optimized for a GPU involves a manual translation that would involve replacing simple C statements present in sequential code with CUDA specific code including the creation of new kernels, including statements for data transfers between host and device memory hierarchies of the GPU or introducing explicit calls to functions present in the CUBLAS libraries [19] for matrix multiplication. Although there are approaches that translate OpenMP code to a CUDA implementation [5], there is limited literature that explains how this can be achieved using sequentially written code.

Intent-based compilation, on the other hand, identifies the intent of the programmer - matrix multiplication in this case and appropriately replaces the pertinent slides [16] with the most appropriate implementation for that particular architecture using appropriate OpenMP, MPI and CUDA calls.

3.2.3 Challenge Factor

Easy to medium. As the algorithmic variants for dense matrix-matrix multiplication are small and the problem has been well studied, parallelizing sequential code in this case is not very difficult. However, optimizing code to achieve maximum performance for a given architecture can pose certain difficulties considering the different choices available. For example, the documentation available for programming the NVIDIA GPU using CUDA lists about eleven variations, including calls to the CUBLAS library.

3.3 Travelling Salesman Problem

3.3.1 Introduction

This case study, in general, represents the optimization class of problems used in many domains. One such well studied optimization problem is the Travelling Salesman Problem (TSP). The last century has seen various approaches presented towards solving the TSP. These range from exact algorithms (brute force) for solving the problem for around 30 cities (nodes), to branch-and-bound approaches [15] for over 60 cities to genetic algorithms [18], progressive improvement and branch-and-cut for a much higher number of cities. The Concorde TSP Solver [7], for example, recently solved an instance with 85,900 points taking over 136 CPU years. Other approaches to solve the TSP are as Nearest Neighbor, Insertion algorithms, Match Twice and Stitch (MTS), Convex Hull, Lin-Kernighan, k-opt heuristic, Ant colony optimization, Tabu Search and Simulated Annealing, to name a few.

The important point to note is that all these *algorithmic variants* seek to solve the same optimization problem – that of having a salesman visit all the given cities once and return to the starting point covering the least possible distance. For a given architecture, the number of cities and the application domain, one approach might be more suitable than the other

.

3.3.2 Parallelization Process

When compiling code for the TSP (and other similar problems), there exist two very interesting alternatives from the intent-based compilation point of view. Having identified the intent in sequential code, should the compiler replace the algorithm with its *algorithmic-variant* or replace the algorithm with its *architectural-equivalent*? To clarify, if the compiler identifies the intent as the TSP problem being solved using the Genetic Algorithm approach should it replace the affected code with the Concorde equivalent (algorithmic-variant), for example, or replace it with code also using a Genetic Algorithm approach but optimized and parallelized specifically for a 64-processor shared memory supercomputer (*architectural-variant*)? Problems that fall in this class of application and other similar ones are hierarchical problems consisting of solution alternatives at two levels or more. Additionally, these sub-problems also have algorithmic variants. For example, there are various ways of solving the TSP problem using genetic algorithm-based approaches – using different crossover and mutation operators, fitness functions, population size, etc. Moreover, for parallelizing any genetic algorithm-based approach unique decisions need to be taken before parallelization. For example, the following questions need to be precisely answered during algorithm formation as even slight variations can result in large deviations between results.

1. Should each processor operate independently on isolated sub-populations or should each processor execute a portion of the algorithm?
2. If the former, should inter-processor crossover or migration be used? If yes, which model [22]?
3. Which crossover operation [18] should be used?
4. How to decide on the mutation rates, termination conditions, and other genetic algorithm specific intricacies?

Similar variations exist when solving using the Ant Colony Optimization and other methods.

3.3.3 Challenge Factor

High to Extremely High. As large algorithmic variants exist for this category of problems, deciding on which one to substitute from among the large available set of *algorithmic variants* is challenging. The first step when applying the intent-based compilation approach to this and other similar hierarchical problems would be to invite human intervention to some extent. Additionally, a comprehensive collection of algorithmic variants and architectural variants for each algorithm would need to be created and maintained. The Pattern Tree creation and maintenance is an intense exercise and its description is beyond the scope of this paper.

4. DESCRIPTION OF TOOL

The tool used for intent-based compilation has been developed on the Linux platform and is based on Perl, the GNU toolkit and MPICH libraries. The compilation, as described earlier and as shown in Figure 1 is a two-step process. In the first-step, the tool parses a correct sequential C program, identifies the developer intent in code, maps the intent to the appropriate algorithmic variant present in the Pattern Tree, identifies the pertinent slices in code that need to be modified and, based on this knowledge, translates the sequential code into code consisting of OpenMP directives and MPI calls. In the second-step, the intermediate code is compiled into an executable for a particular hardware architecture as specified in the user-supplied hardware architecture configuration file. This additional step of intermediate code generation provides additional flexibility for further manual hand optimizations, if needed. This work-in-progress tool, currently, is limited to the translation of sequential code containing the dense matrix-matrix multiplication kernel and has been tested for different values of `m x n`. The parallelized code was executed on a Linux cluster of dual-core PCs. Processor/core utilization achieved was similar to that obtained with hand-optimized code. Initial findings and results obtained using the tool have been published earlier in [1]. The following sections describe the methodology followed in finding the hot spot of the program particular to the IBC tool shown in figure 1.

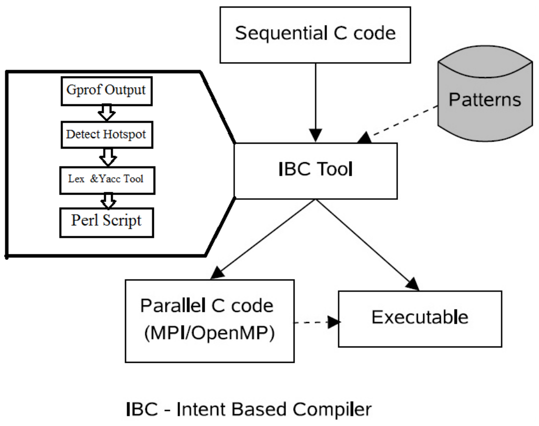

Figure 1: Tool for Intent-Based Compilation

5. CONCLUSION

Research on automatic parallelizers for parallel architectures is a mature field but one with little progress. Limitations of using automatic compilers are known and have been well documented [6]. Intent-based compilation, a novel approach, was presented in this paper that addresses these limitations. Further, the paper, using three very different case studies, described the various challenges faced when adopting the approach. A work-in-progress tool developed for intent-based compilation was presented.

A lot of ground work still remains to be done before the methodology can be applied to any sequentially coded application and for any domain. As a first step in this direction, the creation of a comprehensive pattern tree of algorithmic and architectural variants for the thirteen dwarfs [4] recurring in HPC applications is needed.

REFERENCES

[1] S. Ahmed and W. Ahmed. Code-to-code translator for heterogenous parallel computing environments. In *Proceedings of the International Conference on On-Demand Computing (ICODC'10)*, Bangalore, India, November 2010.

[2] S. Ahmed, S. Bhat, M. Isham, W. Ahmed, and M. K. Ramis. Pre-parallelization exercises in budget - constrained hpc projects: A case study in cfd. *International Journal of Computer Applications*, 1(26), February 2010.

[3] W. Ahmed and D. Myers. Design refinement for efficient cluste ing of objects in embedded systems. In *Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 718–719, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[4] 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 No. UCB/EECS-2006-183, University of California at Berkeley, 2006.

[5] H. Bae, L. Bachega, C. Dave, S.-I. Lee, S. Lee, S.-J. Min, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. In *Proceedings of the 14th Int'l Workshop on Compilers for Parallel Computing*, 2009.

[6] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, pages 12–20, Jan-Feb 2008.

[7] Concorde. Concorde tsp solver. *http://www.tsp.gatech.edu/concorde.html*.

[8] S. Coppersmith, Don; Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[9] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29:84–89, December 1996.

[11] K. Hwang. *Advanced Computer Architecture (Parallelism, Scalability, Programability)*. Tata Mc Grawhill, 1993.

[12] 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, 2007.

[13] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek. A retargetable parallel-programming framework for mpsoc. *ACM Trans. Des. Autom. Electron. Syst.*, 13:39:1–39:18, July 2008.

[14] E. A. Lee. The problem with threads. *IEEE Computer*, May 2006.

[15] A. Levitin. *Introduction to The Design and Analysis of Algorithms*. Pearson Education, 2nd edition, 2007.

[16] S.-W. Liao. *SUIF Explorer: An Interactive and Interprocedural Parallelizer*. PhD thesis, Stanford University, 2000.

[17] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 37–48, New York, NY, USA, 1999. ACM.

[18] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.

[19] NVIDIA. *CUDA CUBLAS Library*, June 2007.

[20] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[21] J. Turley. Survey says: Software tools more important than chips. *www.embedded.com*, November 2005.

[22] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Education, 2004.