Published by :
http://www.ijert.org
International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 06, June-2023

Auto Parallelization

Abhishek Choudhary
Computer Department
Fr.C.R.I.T.
Vashi, India

Gabriel Rajendran
Computer Department
Fr.C.R.I.T.
Vashi, India

Loukik Raina
Computer Department
Fr.C.R.I.T.
Vashi, India

Yashraj Mane
Computer Department
Fr.C.R.I.T.
Vashi, India

Abstract—Dual-core and quad-core processors have become ubiquitous in modern computing. To exploit the capabilities of these multi-core processors, developers often write parallel programs. However, many legacy applications are designed for sequential execution and are unable to fully utilize the processing power of multiple cores. To optimize these applications, they must be either rewritten or parallelized. Manual parallelization is a complex and costly process, making automatic parallelization a desirable alternative. Development of parallel software has traditionally been thought of as time and effort-intensive work. Programs spend most of their execution time in nested loops therefore it will be optimal to execute this parallelly. The ability to perform complex loop restructuring is required for parallelizing programs. A program dependence graph can be used for identifying and distributing the parallel slices of a given sequential program.

The proposed system is an Automatic Code Parallelizer utilizing OpenMP that can automate the insertion of compiler directives to facilitate parallel processing on multi-core shared memory machines. This tool converts sequential C source code into multi-threaded parallel C source code, supporting multi-level parallelization with the generation of nested OpenMP constructs. The proposed scheme breaks down a sequential C program into coarse-grained tasks, analyzes the dependency among tasks, and generates MPI parallel code. This approach prioritizes coarse-grained task parallelism to achieve performance improvements beyond the limits of loop parallelism. Additionally, the generated MPI codes are compatible with a broad range of SMP machines, which may lead to performance gains.

Keywords: OpenMP, MPI, Program dependence graph, Parallelization.

I. Introduction

A. Background

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time after that instruction is finished, the next one is executed.

Parallel computing, on the other hand, uses multiple pro-cessing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent

Identify applicable funding agency here. If none, delete this.

parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked comput-ers, specialized hardware, or any combination of the above. Historically parallel computing was used for scientific com-puting and the simulation of scientific problems, particularly in the natural and engineering sciences, such as meteorology. This led to the design of parallel hardware and software, as well as high-performance computing. However, loop-level par-allelism has gained importance only in the past few years. Due to the ever-growing trend of multi-core architecture, parallel programming is important as well as interesting. Automatic parallelization is a mechanism of automatically converting a sequential program to a version that can directly run on multiple processing elements without changing the meaning of the program. Automatic parallelization is typically performed in a compiler, at a high level where most of the information needed is available. Computing power can be used effectively if the programmers write only the sequential codes and leave the task of parallelization to the compiler.

B. Motivation

It is seen that parallel programming is a difficult chore requiring great effort from the programmer. One conclusive elucidation of this problem is automatic parallelization.

Currently well accepted methods of parallel programming, such as MPI, are essentially extensions to existing languages, like C or FORTRAN. On one hand, it allows the reuse of an existing code base while on the other hand, it requires both the compiler and programmers to deal with languages that were not originally designed for parallelism. The major challenges involved in the design and implementation of such a tool include side-effects of function calls, finding alias variables, the dependency between statements, etc. Additionally, the tool has to deal with a variety of coding styles, length and number of files. It is also important to take into consideration the amount of inherent parallelism the application provides.

C. Aim and Objective

The output of an auto-parallelizer is a race-free deterministic program that obtains the same results as the original sequential

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

program. This dissertation deals with compile-time automatic parallelization and primarily targets shared memory parallel architectures for which auto-parallelization is significantly easier. The programming control structures on which auto parallelization places the most focus are loops, because, in general, most of the execution time of a program takes place inside some form of loop.

Automatic parallelization eliminates the need for program-mers to manually identify sections of code that are suitable for parallel execution. It also relieves them of the burden of performing data dependency analysis to ensure program correctness and inserting parallel code or directives at relevant locations. However, there is some overhead associated with parallelization. While automatic parallelization may result in some performance gain, the extent of the improvement depends on the inherent nature of the program and the degree of parallelization. Programs with many dependencies will still be executed serially and will not see any performance improvement.

Loop-level parallelism, where loop iterations are paral-lelized, is a common approach to parallelization. However, this method requires prior knowledge of loop bounds and cannot be applied to unstructured programs. As a result, there is a need for systems that can automatically leverage task-level parallelism in sequential programs.

The proposed approach for achieving this is based on coarse-grained task-level parallelization. In this approach, a sequential program is statically decomposed into tasks, and the parallelism among tasks is analyzed. Independent tasks are then parallelized using OpenMP task constructs, while ap-propriate synchronization constructs are introduced to ensure program correctness in the presence of dependencies. Further-more, the proposed system considers multi-level parallelism by considering nested loops within the outer loop.

II. Literature Survey

A. Survey of Research Papers

Hamid Arabnejad et al. [1] presented the AutoPar-Clava compiler, which provides a versatile automatic paralleliza-tion approach for Clava, a C source-to-source compiler. The compiler is currently focused on parallelizing C programs by adding OpenMP directives. The proposed source-to-source compiler deals with the original code, and inserts OpenMP directives (mainly parallel-for and atomic directives) and the necessary clauses.

Vladmir Beletsky et al. [2] created a package that provides effective parallelization of the programs that are written in the C language and guarantees the correctness in executing them on transputer-systems of various configurations.

Amit G Bhat et al. [3] proposed an algorithm which is simple and efficient for automatic parallelization of “for” loops using OpenMP APIs. It uses compile time cost estimation in order to determine whether parallelization of the program is profitable or not, since the overhead involved in creation of multiple threads sometimes degrades the performance of loops on par-allelization. The algorithm also performs minor modification possible, such as multiple initialization elimination, in order make the loop compatible for using OpenMP APIs.

Manju Mathews et al. [4] proposed an algorithm whose focus is on multilevel coarse grained task parallelism and does not consider loop parallelism. The observations on sample test codes indicate that there is a performance gain with parallelization.

B. Existing Systems

- YUCCA : YUCCA is a Sequential to Parallel automatic code conversion tool developed by KPIT Technologies Ltd. Pune. It takes input as C source code which may have multiple source and header files. It gives output as transformed multi-threaded parallel code using pthreads functions and OpenMP constructs. The YUCCA tool does task and loop level paral-lelization.
- Par4All : Par4All is an automatic parallelizing and opti-mizing compiler (workbench) for C and FORTRAN sequential programs. The purpose of this source-to-source compiler is to adapt existing applications to various hardware targets such as multi-core systems, high performance computers and GPUs. It creates a new source code and thus allows the original source code of the application to remain unchanged.
- Cetus : Cetus is a compiler infrastructure for the source-to-source transformation of software programs. This project is developed by Purdue University. Cetus is written in Java. It provides basic infrastructure for writing automatic paral-lelization tools or compilers. The basic parallelizing techniques Cetus currently implements are privatization, reduction vari-ables recognition and induction variable substitution. A new graphic user interface (GUI) was added in Feb 2013. Speedup calculations and graph display were added in May 2013. A Cetus remote server in a client–server model was added in May 2013 and users can optionally transform C Code through the server. This is especially useful when users run Cetus on a non-Linux platform. An experimental Hubzero version of Cetus was also implemented in May 2013 and users can also run Cetus through a web browser.
- PLUTO : PLUTO is an automatic parallelization tool based on the polyhedral model. The polyhedral model for compiler optimization is a representation for programs that makes it convenient to perform high-level transformations such as loop nest optimizations and loop parallelization. Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but not lim-ited to those. OpenMP parallel code for multi-cores can be automatically generated from sequential C program sections.

III. Proposed System

Our proposed system is to have a transpiler that can parallelize the execution of sequential code as input. This transpiler will act on certain subset of programming languages and will optimize the execution by maximizing the throughput

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

and making maximum utilization of available resources, and worker nodes.

It will make use of extracted section parallelization that aims to identify sections of code independent of data flow within the program and do not rely on execution of previous code blocks and do not have any dependents further in the program. The extent of parallelization relies on the level of coupling within program, a program with lot of interdependence and shared memory usage will have poor parallelization compared to sequential code with independent code blocks with minimal coupling. Independent blocks with non-overlapping resource access can be separated for parallelization, while these non-overlapping sections might require sequence, they can be parallelized independently.

Automatic parallelization aims to minimize human inter-vention, it has stages which are analogous to working of compilers, and involves code parsing into an intermediate form which is independent of source program. The stage that's not analogous to compiler is the schedule phase where we define the order of execution of non-overlapping sections and the distribution of tasks to worker nodes in an optimal way and ensuring redundancy for fault tolerance. The parallelized ver-sion of our code will be able to work on certain architectures of CUDA and use MPI for message passing between worker nodes.

IV. Design

A. Data Flow Diagram

1) Data Flow Diagram (Level 0):

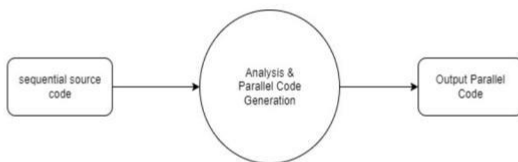

Fig. 1. Data Flow diagram: Level 0

2) Data Flow Diagram (Level 1):

• Read Input : Read sequential code in C as input from storage.

• Lexical Analysis : This component is responsible for parsing the input file and identify all tokens from the file. Each line in the file is checked for predefined keyword to separate them into tokens.This token is then used later for analysis.

• Analyze : The analyzer is used to identify sections of code that can be executed concurrently. The analyzer uses the static data information provided by the scanner-parser. The analyzer will first find all the totally independent functions and mark them as individual tasks. The analyzer then finds which tasks have dependencies.

• Identify blocks that can be parallalized : Independent blocks with non-overlapping resource access can be separated for parallelization, while these non-overlapping sections might require sequence, they can be parallelized independently.

• Code Generation : The scheduler will generate a list of all the tasks and the details of the cores on which they will execute along with the time that they will execute for. The code Generator will insert special constructs in the code that will be read during execution by the scheduler. These constructs will instruct the scheduler on which core a particular task will execute along with the start and end times.

• Write Output : Write parallel code in C as output to the storage.

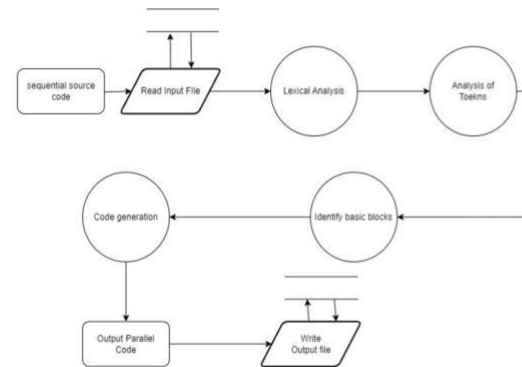

Fig. 2. Data Flow diagram: Level 1

3) Data Flow Diagram (Level 2):

• Read Input : Read sequential code in C as input from storage.

• Lexical Analysis : This component is responsible for parsing the input file and identify all tokens from the file. Each line in the file is checked for predefined keyword to separate them into tokens.This token is then used later for analysis.

• Dependency Analysis : Dependence analysis is the basis for the parallelizer to decide whether a loop is parallelizable. AutoPar invokes the dependence analysis from the loop opti-mizer, which implements algorithms to effectively transform both perfectly nested loops and non-perfectly nested loops. An extended direction matrix (EDM) dependence representation is used to cover non-common loop nests that surround only one of the two statements in order to handle non-perfectly nested loops.

• Data Flow Analysis : Data-flow analysis is a technique for gathering information about the possible set of values cal-culated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

• Identify blocks that can be parallalized : Independent blocks with non-overlapping resource access can be separated for parallelization, while these non-overlapping sections might require sequence, they can be parallelized independently.

• Determine order of blocks to be executed : List all the tasks and their dependencies on each other in terms of execution and

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

start times. The scheduler will produce the optimal schedule in terms of number of processors to be used or the total execution time for the application.

• Code generation : Code generation is done after analysis phase is complete. Here we can choose to generate code for different architecture such as CUDA or MPI based on user choice. Code generation is done based on the previous analysis and techniques like polyhedral models for transforming code into different architectures.

• Write Output : Write parallel code in C as output to the storage.

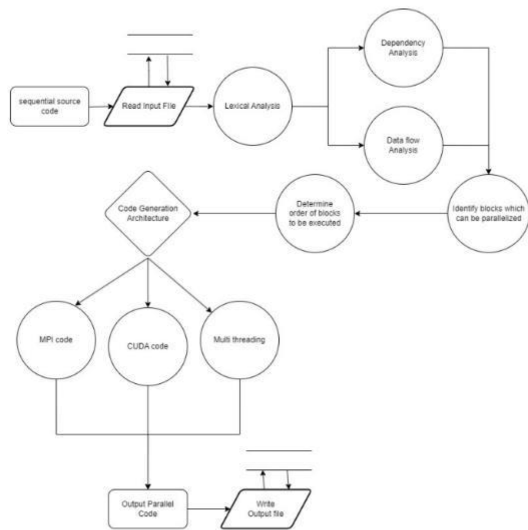

Fig. 3. Data Flow diagram: Level 2

V. Results and Performance Analysis

For testing purpose, Intel Core i3 is used which has two cores with Hyper Threading (HT) technology. Operating Sys-tem is Ubuntu 14.04. The compiler version is gcc 4.8.2 which supports OpenMP version 3.1. A dual core machine with HT has 2 physical cores but scheduler treats them as 4 logical cores. For the logical processors in a HT enabled machine, the architectural state of the processor is duplicated. We used OpenMPI to connect up to 6 computers having the same processing power and used them for testing various algorithms.

Fig. 4, shows the execution time on different numbers of processors connected with OpenMPI, executing sum of array program for different array sizes. It can be observed that sequential execution time is lesser than execution time on one processor with MPI, this is because of the network overhead and communication overhead caused due to MPI. Also it is observed that for increase in number of processors the execution time is drastically reduced.

Fig. 5, shows the execution time for matrix multiplication [13] for different dimension of matrix using sequential and MPI parallel code (no. of processors used = 3 here). It can be seen for higher dimension matrix MPI parallel code is working much better than sequential one.

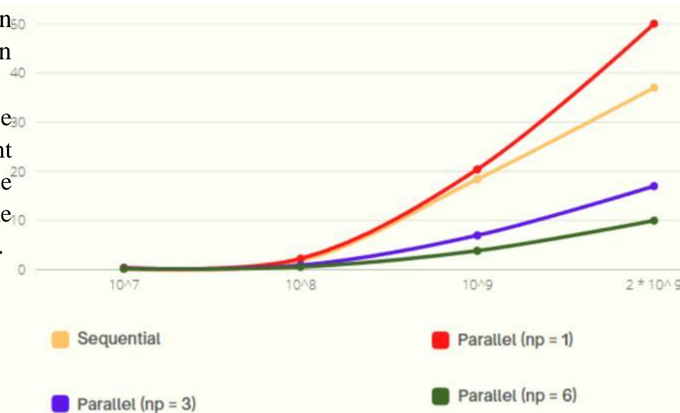

Fig. 4. Time Comparison: Sum Of Array

Fig. 5. Execution time for matrix multiplication

VI. Conclusion

In this paper, we aim to design an auto parallelization compiler, which converts the source program such that its compatible with parallel systems and ensure maximum con-currency. We analyze various techniques to parallelize certain sections of source program. It allows us to maximize the throughput and effective utilization of parallel systems without manually writing parallel code.

Automation of parallelization tasks allows us to make more effective use of multi-core processors, with development of tools such as OpenMP and CUDA, it is now possible to have true concurrency, and an effective way to parallelize programs with minimal human intervention will be very useful and allow complex computation on commodity hardware, thereby increasing it's reach to researchers who do not have access to high-end supercomputers.

Data Flow analysis, and dependency analysis helps in identifying which sections of code can be parallelized, and also helps us in understanding limitations of such Auto-parallelization mechanisms, Having shared memory access poses complexity if the parallel sections of code have a non-atomic read-modify-write operations which might result in inconsistent data flow. While we are limited by this, it is

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

possible to have lock-free techniques but are infeasible to automate with current advancements in auto-parallelization techniques.

The development of an auto-parallelization compiler will enhance the performance of sequential codes which can run on hardware with no parallelization support and also hardware which support parallelization and it would be automatic so we won't have any development overhead. This paper aims to describe and propose ideas which can help in development in this domain.

VII. Future Scope

The future scope for our project, which is focused on auto parallelization of C programs to MPI programs, can be expanded in a number of ways to make it more powerful and versatile.

Currently, our project supports a limited number of algo-rithms for auto parallelization. We can expand the scope of our project by adding support for additional algorithms for auto parallelization. We can also improve the effectiveness of auto parallelization by incorporating state-of-the-art algorithms that are designed to handle a wide range of code patterns and architectures. In addition to supporting MPI, we could add support for CUDA, which is a parallel computing platform developed by NVIDIA. We also have to improve the accuracy of the auto parallelization so that output of our project may always be optimal. While our project is currently focused on C, we could consider adding support for other programming languages as well.

VIII. Acknowledgements

We would like to express our sincere gratitude and appre-ciation to Prof. Amroz Siddiqui for his invaluable guidance, support, and encouragement throughout the course of our project. His expertise, insight, and willingness to provide constructive feedback have been instrumental in shaping our work and pushing us to achieve our best.

We extend our heartfelt appreciation to Prof. Shashi R Dugad for his unwavering support, exceptional guidance, and invaluable contributions, which have been instrumental in making this project a resounding success. His remarkable mentor ship has left an indelible mark on our academic and personal lives, and we will always be grateful for his unwavering support and guidance. His exceptional leadership, unwavering dedication, and profound expertise have been a source of inspiration for us throughout this endeavor.

We would also like to thank the computer department of our college for their assistance and resources, which have been essential in completing this project. Their commitment to providing a high-quality learning environment and state-of-the-art facilities has been instrumental in our success.

Without their support and guidance, we would not have been able to complete this project successfully. We are truly grateful for their contribution and look forward to continuing our work with their continued support and guidance.

We would also like to express our heartfelt gratitude to Prof. Lata Ragha, Head of the Computer Department, for her constant support and guidance throughout the project.

IX. References

[1] M. Mathews and J. P. Abraham, ”Automatic Code Par-allelization with OpenMP task constructs,” 2016 International Conference on Information Science (ICIS), Kochi, India, 2016, pp. 233-238, doi: 10.1109/INFOSCI.2016.7845333.

[2] V. Beletsky, A. Bagaterenco and A. Chemeris, ”A package for automatic parallelization of serial C-programs for distributed systems,” Programming Models for Massively Parallel Computers, Berlin, Germany, 1995, pp. 184-188, doi: 10.1109/PMMPC.1995.504357.

[3] A. G. Bhat, M. N. Babu and Anala M R, ”Towards automatic parallelization of “for” loops,” 2015 IEEE Inter-national Advance Computing Conference (IACC), Banglore, India, 2015, pp. 136-142, doi: 10.1109/IADCC.2015.7154686.

[4] M. Mathews and J. P. Abraham, ”Automatic Code Par-allelization with OpenMP task constructs,” 2016 International Conference on Information Science (ICIS), Kochi, India, 2016, pp. 233-238, doi: 10.1109/INFOSCI.2016.7845333.

[5] A. Beletska, W. Bielecki, and P. S. Pietro, “Extracting coarse-grained parallelism in program loops with the slicing framework, in parallel and distributed computing.,” tech. rep., ISPDC ’07. Sixth International Symposium, July 2007.

[6] C. Bastoul, “Code generation in the polyhedral model is easier than you think,” tech. rep., Laboratoire PRiSM Universit´e de Versailles Saint Quentin 45 avenue des Etats-Unis, 78035 Versailles Cedex, France.

[7] N. Patankar, A. P. Khanbu, H. Bhoir, and A. Siddiqui, “Automated c code polyhedral parallelizer,” tech. rep., FCRIT Vashi,Maharashtra,India, 2015.

[8] R. Rugina and M. Rinard, “Automatic parallelization of divide and conquer algorithms,” tech. rep., Laboratory for Computer Science Massachusetts Institute of Technology Cambridge, MA 02139, 1999.

[9] B. Hertzberg and K. Olukotun, “Runtime automatic speculative parallelization in code generation and optimization (cgo), 2011,” tech. rep., 9th Annual IEEEIACM International Symposium, April 2011.

[10] J.-F. Collard, “Automatic parallelization of while-loops using speculative execution,” tech. rep., 1995.

[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Brown-ing, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Freder-ickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, “The nas parallel benchmarks—summary and preliminary results,” in Proceed-ings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing ’91, (New York, NY, USA), p. 158–165, Association for Computing Machinery, 1991.

[12] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The splash2 programs: characterization and methodological considerations,” in Proceedings 22nd Annual International Symposium on Computer Architecture, pp. 24–36, 1995.]

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 06, June-2023

[13] Ziad A.A. Alqadi, Musbah Aqel and Ibrahiem M. M.El Emary, " Performance analysis and evaluation of paral-lel matrix multiplication algorithms, World Applied Sciences Journal, 5 (2): 211-214, 2008.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)