Dissertation on

"Automated Parallelization of Source Code using Program Comprehension"

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology
in
Computer Science & Engineering

UE18CS390B – Capstone Project Phase - 2

Submitted by:

Darshan D	**PES1201801456**
Karan Kumar G	**PES1201801883**
Manu M Bhat	**PES1201801452**
Mayur Peshve	**PES1201801439**

Under the guidance of

Prof. N S Kumar
Visiting Faculty
PES University

June - December 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

'Automated Parallelization of Source Code using Program Comprehension'

is a bonafide work carried out by

Darshan D	**PES1201801456**
Karan Kumar G	**PES1201801883**
Manu M Bhat	**PES1201801452**
Mayur P L	**PES1201801439**

in partial fulfilment for the completion of seventh semester Capstone Project Phase - 2 (UE18CS390B) in the Program of Study - Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period June - December 2021. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 7th semester academic requirements in respect of project work.

Signature	Signature	Signature
N S Kumar	Dr. Shylaja S S	Dr. B K Keshavan
Visiting Faculty	Chairperson	Dean of Faculty

External Viva

Name of the Examiners	**Signature with Date**
1. ____________________	____________________
2. ____________________	____________________

DECLARATION

We hereby declare that the Capstone Project Phase - 2 entitled **“Automated Parallelization of Source Code using Program Comprehension”** has been carried out by us under the guidance of Prof. N S Kumar, Visiting Professor and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester June - December 2021. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

PES1201801456	**Darshan D**	Darshan D
PES1201801883	**Karan Kumar G**	[illegible]
PES1201801452	**Manu M Bhat**	Manu. Bhat
PES1201801439	**Mayur P L**	Mayur

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. N S Kumar, Department of Computer Science and Engineering, PES University, for his continuous guidance, assistance, and encouragement throughout the development of this UE18CS390B - Capstone Project Phase – 2.

We are grateful to the project coordinator, Prof. Silviya Nancy J, for organizing, managing, and helping with the entire process.

We take this opportunity to thank Dr. Shylaja S S, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from the department. We would like to thank Dr. B.K. Keshavan, Dean of Faculty, PES University for his help.

We are deeply grateful to Dr. M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro Chancellor – PES University, Dr. Suryaprasad J, Vice-Chancellor, PES University for providing us various opportunities and enlightenment every step of the way. Finally, this project could not have been completed without the continual support and encouragement we have received from our family and friends.

Abstract

With the diminishing rise of computational capacity of hardware resources available in today's world, it is imperative to make efficient use of available computational power to achieve better optimisations and faster execution times to improve cost-effectiveness. There is a need to utilise multiple resources simultaneously to achieve parallel execution. This requires software to be written by developers in such a way as to run parallely on multiple resources. However, designing code to achieve parallelism is not an easy task and requires skilled developers for the same. Automating the process of converting sequential to its parallel equivalent source code helps overcome the dependency on human expertise and saves valuable time. We propose such an automated solution targeting all types of generic sequential code, which converts it to its accurate parallel version, capable of utilising all of the hardware resources available in the underlying system. We do so by integrating multiple techniques to cover various cases of source code. We analyse sections of source code to understand the intent and correspondingly replace it with an optimized version. We also employ a scheduling algorithm to achieve fine grained control over execution of multiple sections of code, thereby maximising efficient usage and minimising execution times.

Table of contents

List of Figures

1. Introduction

In the earlier days, there was very little computational power that could be provided by the processors on a computer system. Software was written sequentially, which meant every instruction was executed one by one. This meant large softwares took a very long time to execute.

But there was an exponential boom in the development of computer hardware, with faster and more powerful processors coming into the market. The clock speed drastically increased which enabled more instructions to be executed in very short durations of time. However, this continuous increase in power and computational capacity was cut short a few years ago. This meant that ever increasing need for more and more computational power could no longer rely on developing hardware. We had to shift focus to the software side, to make intelligent and efficient use of available hardware.

Parallel computing is a paradigm which enables us to make efficient use of available hardware to run multiple instructions on multiple hardware resources, simultaneously. This could mean running separate tasks on multiple cores/threads available on a single system, or running separate tasks on multiple systems connected in a cluster. This enables us to execute our software faster and saves valuable time in several use cases, thereby improving cost-effectiveness. Thus, we propose a novel approach to solve the problem of automated parallelisation of sequential code, by using multiple techniques integrated into one package to convert any general sequential source code to its parallel equivalent source code without any human intervention. We achieve this primarily by using the concepts of Program comprehension and Task-level parallelism. Program Comprehension enables us to identify intent and algorithm implemented in a code section and consequently replace the same with its optimised parallel version as is defined in a backend database. Task-level parallelism enables us to execute different functions in a given source code in parallel by use of a data-dependency driven scheduling algorithm.

2. Problem Statement

Parallel computing is a programming paradigm that enables efficient use of available hardware to run multiple instructions on multiple hardware resources simultaneously, allowing for faster software execution. This saves valuable time in several use cases and reduces the cost considerably.

However, a major challenge with developing programs that have parallelism is that there is a need for highly skilled programmers. There is also the problem that parallelizing pre-existing source code would take a large amount of manpower and time. This may not be feasible or even affordable, depending on the size and nature of the project. Auto parallelization techniques would help in mitigating the cost, manpower and time required for such a project. There has been significant work done in this area, especially in recent times, that has shown promising results and new ideas with interesting perspectives on how to go about parallelising source code.

We propose a solution that enables automated conversion of sequential source code to parallel source code with the aid of program comprehension and our bespoke scheduling algorithm, capable of task-level parallelization.

Existing auto-parallelisation techniques are either constrained to specific domains alone or do not try to perform parallelisation by doing thread scheduling explicitly that would allow for fine-grained control over the execution. We intend to use the concept of program comprehension to assimilate the source code and use that information to parallelise based on programming paradigms. We also intend to use a novel thread scheduling idea that would allow us to control the execution of the client program such that we can improve the performance by using multiple threads while reducing any overheads caused due to parallelism.

The implementation can therefore be broken down into the following:

1. Conversion of given sequential source code to an Enriched Abstract Syntax Tree representation, which enables us to query the nodes to retrieve useful information about the input code.
2. Data dependency, control and data flow analysis performed with the aid of the generated AST to identify independent code sections.
3. Program comprehension to understand the programming paradigm or the intent of each program section or function using the concept of Clustering based on program vector embeddings.
4. Parallelisation of the eligible sections of source code, based on the identified programming algorithm and the best fit parallel version for it.
5. Using the data dependency graph, to execute multiple sections of the entire program in separate threads with the help of our scheduling algorithm, to maximise CPU utilisation and minimise execution times.

3. Literature Survey

3.1. Program Comprehension for parallelisation

3.1.1. Pasquale Cantiello and Beniamino Di Martino, Automatic Source Code Transformation for GPUs Based on Program Comprehension, 2012.

3.1.1.1. Introduction

The authors use a previously established technique of program comprehension called the PAP Recognizer along with some modifications. This is used to improve the performance of a program, specifically for running on GPU. The methodology used is based on previously built upon ideas regarding program comprehension, using a static analyser. There is an “Extractor”, which works based on Prolog facts, which is used to recognise programming paradigms or patterns in the program. These patterns are then used to recognise the algorithm being used. Upon recognising the algorithm, the AST is then modified by a transformer module to transform the version which runs the same algorithm in a parallel manner.

3.1.1.2. Implementation

The implementation of both the static analyser and the transformer module is on an AST. This allows for the tool to abstract out the program a little more, hence makes it easier to process and modify. The static analyzer is first run on the AST, to identify algorithms described by different sub-trees in the AST. The algorithmic identification is done based on hierarchical parsing, which reduces the memory usage of the identifier. After identifying the sub-tree and the algorithm associated with it, the following steps are carried out:

- Removal of the subtree from the AST, and original code commented.
- Generation of a new tree with the modified code which may contain the necessary library calls and memory allocations needed for GPU processing.
- Addition of this newly generated tree at the same location as the removed subtree.

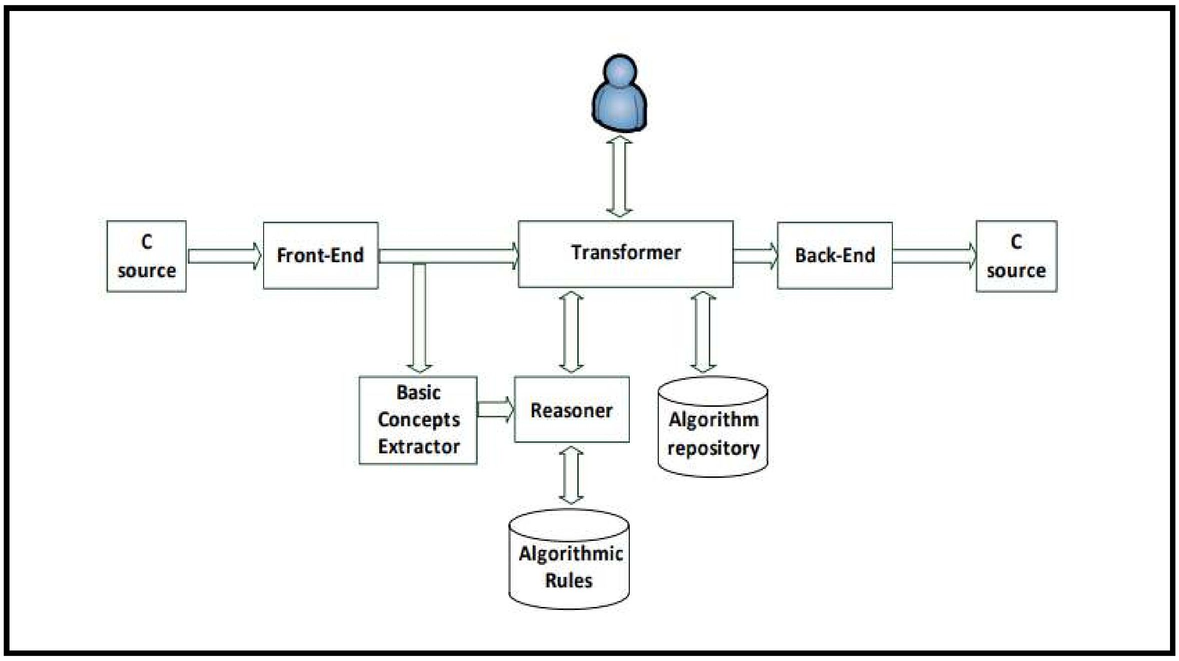


Figure 1: Working pipeline of Parallelisation tool

The transformer module makes the modifications to the AST based on "Algorithm repository" to make code better suited for running on GPU

The tool was implemented using SWI-Prolog for the analysis of AST based on rules, while the ROSE compiler was used to extract AST and to be able to query and modify the AST. The tool was capable of parsing C/C++ and FORTRAN 2003

3.1.1.3. Conclusion

The paper gives us insights into how program comprehension can be used to assimilate code, using a static analyser. It also provides for an interesting approach, by using AST to abstract out the program, allowing us to remove the personal choices of the programmers, and only deal with the necessary details of the program. This also lays the foundation for what can be done to perform parallelism within a small portion of the program, allowing for possible improvements in performance in a function or a small section of code.

Currently, the tool is only built to work on basic linear algebra algorithms, as the rules for algorithm recognition are written only for those. However, the recognition phase still takes a lot of time and hence has scalability issues. So a performance investigation on the transformer would possibly help improve that. The authors have ensured that there is an extension to OpenCL, allowing for usage on heterogeneous architectures, however, OpenCL is not used much in GPU programming anymore. They have also stated that they are working on increasing the algorithms that will be recognised.

3.1.2. Martino B. D. & Iannello G, Towards automated code parallelization through program comprehension, 1994.

3.1.2.1. Introduction

The authors talk about how existing tools simply try to identify loops that can be parallelized and loops that cannot be parallelized using data flow and dependency analysis. They discuss the "Concept Assigning Problem". In its general form, it is analyzing the programs to identify abstract concepts. However, the paper proposes that this problem can have complete automation if identification is applied on programming concepts and algorithms, such as searches, sorts, structure modification, numerical, etc. (which are essentially algorithmic concepts)

3.1.2.2. Implementation

Defines the parallel structure of a program as Programming Paradigms (PP). There are two PPs proposed:

- **Tree computation:** a set of processes that communicate with one another based on a binary tree structure. Every problem that needs to be solved is divided into sub-problems, which are treated as child processes. These child processes are executed and their results are moved upward to their parents
- **Master-Worker:** consists of a master process and a collection of worker processes. The worker processes carry out processes as defined by the master process.

Problems irrespective of their specific working and applications can be grouped under the above two mentioned paradigms. For eg: the "Tree computation" paradigm can be assigned to an iterative quicksort algorithm whereas the "Processor farm" paradigm can be assigned to a branch-and-bound binary search problem. To recognize the paradigm itself, the authors make use of a finite set of pattern templates (called cliches). PS defines the Parallel Skeleton code for the selected paradigm.

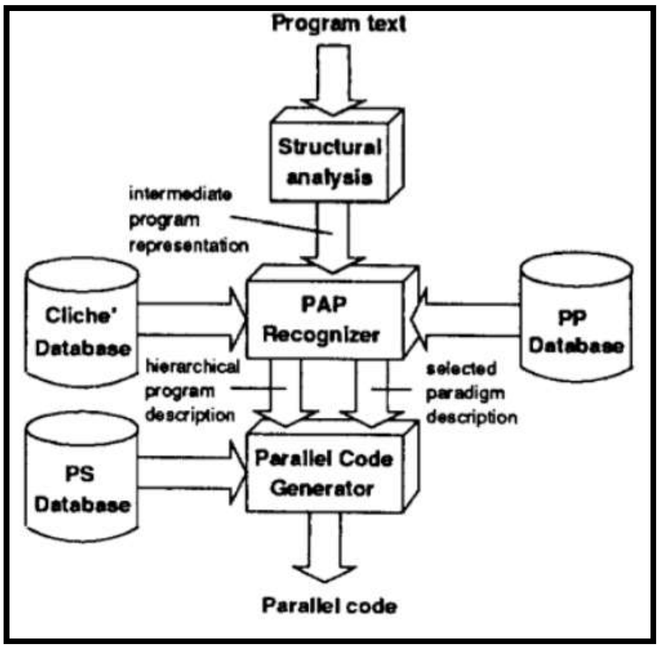


Figure 2: Working pipeline of PAP Recogniser

3.1.2.3. Conclusion

- This is only theoretical, a tool hasn't been devised.
- The approach is very ambitious and might not work on all generic programs
- Upon further research, the concept of cliches and paradigms as defined in the paper can be looked into, possibly expanding the databases to further improve the generality of the tool
- This paper was before OpenMP came into existence. With the use of OpenMP, it is possible to have better results using this same concept.

3.1.3. Di Martino B & Iannello G, PAP Recognizer: a tool for automatic recognition of parallelizable patterns, 1996.

3.1.3.1. Introduction

The authors present a new method of analysing programs to identify certain regions, which they call Parallelizable Algorithmic Patterns (PAP). They do so by using program recognition based processes. The output of the tool is a web application based representation showing the hierarchical description of the recognized patterns. The prototype tool has been implemented in the Vienna Fortran Compilation System.

3.1.3.2. Implementation

The Automatic parallelization problem is categorised into the following requirements:

- Identification of a parallel model
- Data distribution for the defined processes
- Selection of work categorisation, enabling code decomposition

The tool mentioned is mainly driven by paradigm recognition rules working on representation of concepts. These rules enable differentiation of multiple concepts based on intent and meaning. These rules can be represented as abstract functions contributing to building an abstract structure. Control and data dependency analysis are implemented as an abstract process.

The tool analyses the program by making use of control flow and data flow to develop some kind of abstract flow structure. All this information about underlying concepts are stored as dependency graphs in a datastore. This datastore is updated as the identifier parses the entire program.

3.1.3.3. Conclusion

The PAP Recognizer introduced in this paper is capable of recognizing some features related to numerical computation only. It would be challenging to introduce new concepts that are compatible with the design of this tool. Hence generalization would be an uphill task.

3.1.4. Di Martino B & Kessler C.W, Two program comprehension tools for automatic parallelization, 2000.

3.1.4.1. Introduction

- Program comprehension is identification of abstract concepts in a program
- This could mean finding sequences of code implementing some algorithmic concept
- This is challenging because of syntactic variation, algorithmic variation, delocalization, and overlapping implementations.
- Why do program comprehension for automatic parallelization?
  - It provides for an aggressive code transformation, without being dependent on the sequential structure in the program
  - The acquired knowledge allows for automatically selecting segments for optimization, and helps improve the working of performance prediction

  - The application domain considered mainly consists of numerical computations, linear algebra and partial differential equation codes (hence it is not exactly generalized)
- Non numerical examples - quicksort and branch-and-bound

3.1.4.2. Implementation

- PAP was implemented in the Vienna Fortran Compiler (VFC)
- PARAMAT tool provides a faster solution, whereas PAP Recognizer provides for a greater generalization with higher run times
- Recognition of subconcepts for a given concept to identify the hierarchical concepts is done.
- Rules to infer a concept are defined in terms of intermediate representation features like operator symbols, equality of program objects, control and data flow information and already computed subconcept information.
- The IR can be an annotated abstract syntax tree for which customized tree-pattern matching techniques will be used
- It uses Concept of Interactive Parallelization - helping users with useful information during their efforts to parallelize a program
- Another goal is to replace program parts with calls to already implemented parallel code

3.1.4.3. Conclusion

Applications are plenty with this approach of Program comprehension to parallelization. Some of these applications are:

- Automatic Code restructuring and Library use
- Template based code transformations
- Automatic data distribution
- Automatic performance prediction
- Speculative program restructuring
- Knowledge-based support to interactive parallelization (take live inputs from users during the parallelization process)

The authors say that they have indeed tried to integrate the two tools together to extract the best out of the two. Their conclusion is as follows:

If recognition time is of utmost importance, PARAMAT can be used, otherwise use the PAP tool since it offers greater flexibility.

3.2. Parallelisation techniques

3.2.1. Peter Kraft, Amos Waterland, Daniel Y Fu Anitha Gollamudi, Shai Szulanski, Margo Seltzer, Automatic Parallelisation of Sequential programs, 2018

3.2.1.1. Introduction

The paper deals with the implementation of a previously hypothesised idea known as ASC (Automatic Scalable Computation) architecture. In ASC, the memory state and registers are used to create a vector. This vector is then used to predict the possible trajectory of the program execution. This prediction is then used to continue execution. By having simultaneous executions, when the program comes to the point at which the prediction was made, the worker thread that was tasked to continue the execution of the correct prediction is then used. This method uses processor farms to improve performance.

3.2.1.2. Implementation

The vector for characterising the program state is made using the memory state of the program and registers involved, by the usage of ptrace and perf_events. This vector is then used as input to either a neural network or a decision tree. The prediction made by the neural net or decision tree is stored in a look-up table. Also the predicted execution is passed to a worker thread to continue execution from the predicted point onwards. Once the execution of the main program reaches the point of the prediction, we do a look-up on the predicted state and match with the actual state to see which worker had the task of executing the correct prediction. The result of the worker thread is then used, and the main program execution skips to the point at which worker thread is executing. The paper uses an Intel tool called "PIN", which on the basis of dynamic analysis identifies points at which predictions can be made.

3.2.1.3. Conclusion

The paper extensively uses PIN, however it is a very expensive tool, so improvements to PIN would improve the performance of the tool. The paper also discusses the potential use of other dynamic instrumentation tools like JIT with PIN and valgrind. There has been some related work, in the area of binary parallelisation (parallelising sequential binaries to parallel equivalent) and compiler parallelisation, including the usage of OpenMP and other specific compilers. The tool also fails with things such as accumulators and similar concepts.

3.2.2. Cristian Ramon-Cortes, Ramon Amela, Jorge Ejarque, Philippe Clauss, Rosa M. Badia, AutoParallel: A Python module for automatic parallelization and distributed execution of affine loop nests, 2018.

3.2.2.1. Introduction

The authors propose a python module for automated task-based parallelisation. This is implemented on affine loop nests to execute them in parallel in a distributed computing environment. This also involves construction of data blocks to identify and use task granularity for achieving better performance in terms of execution.

3.2.2.2. Implementation

- Uses existing tools to devise a pipeline to parallelize Python code on distributed systems
- Pluto is a tool which helps in parallelizing affine loops
- COMPs is a tool which helps in converting source code to run on a distributed cluster (similar to functionalities of MPI)
- A combination of these two is proposed (similar to a combination of OpenMP and MPI - however, this is manual)

3.2.2.3. Conclusion

- Difference between affine and non-affine programs:
- Affine loops are the loops in which the referenced array subscripts and loop bounds are a linear function of the loop index variables. This implies that the memory access sequence is already noted in the compile time itself.

  Example:

  ```
  for (int variable = 1; variable < 12424; ++variable)
          my_array[variable] = my_array[variable-1] + 9586
  ```

- In the case of non-affine loops, the memory access sequence cannot be pre-identified during the compile time itself.

Example:

```
for (iter=1;iter<50;++iter)
        my_z[my_e[iter]] = my_z[my_t[i]] + my_z[my_p[i]]
```

- Most of the research has been done with respect to affine transformations as the availability of memory access patterns at compile time helps in the development of techniques and methods that can be used to identify the parallelizable segments easily.
- Performing parallelization for non-affine program segments can be something that can be looked at as a possible area of research.

3.3. Automated tools for parallelisation

3.3.1. Pluto: Uday Bondhugula, J. Ramanujam, P. Sadayappan, PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System, 2007.

3.3.1.1. Introduction

The Pluto tool is an auto-parallelisation tool that uses the ideas of polyhedral modelling and analysis to parallelise programs using OpenMP pragma directives, along with other techniques such as loop optimizations, tiling, loop merging etc. It was developed as a PhD thesis by Uday Bhondugula, currently a professor at IISC Bangalore. It has been a benchmark tool since its inception in 2007. The ideas of polyhedral analysis are deep rooted in Integer Linear Programming, so as to be able to optimise the execution of programs.

3.3.1.2. Implementation

The tool uses the ideas of polyhedral modelling. In polyhedral modelling, each loop is considered a lattice point and a program forms a polyhedral. This polyhedral is then made to undergo an affine transformation by the usage of Integer Linear Programming. By performing these transformations, there are checks to make sure that the correctness of the program is maintained while improving the performance of the loops being transformed. The transformed polyhedral is then converted back into a program and made available to users. For affine loops, the compiler applies transformations based on dependencies. For non-affine loops, compilers may perform various other transformation techniques, such as tiling or unrolling of loops etc. Performing the affine and non-affine transformation converts the polyhedra into different optimised polyhedra.

3.3.1.3. Conclusion

The drawback of polyhedral analysis is the expensive nature of Integer Linear programming. This is hence translated into the usage of Pluto itself. The tool is also very cumbersome to install. However, with respect to loop parallelisation, the results of the tool were accurate. Following were the restrictions we found out about the tool:

- Placement of pragma scop and pragma endscop is user-defined. Hence this step is not fully automated

- If this section defined under pragma contains code that can all be parallelized, then it ends up parallelizing them. However, if there is even one section of code within this pragma section that cannot be parallelized, then even those code segments that can be parallelized do not get parallelized (can still perform loop jamming but no parallelization)
- Can only place one such pragma section in the entire program (otherwise issue of redeclaration of helper variables that is automatically used by the tool internally)
- Placement of pragma requires user intervention and knowledge about parallelism defeating the purpose of auto parallelization
- Not able to handle cases of reduction (divide and conquer cases) For example, accumulator to sum up elements of an array

3.3.2. ParaWise – Widening Accessibility to Efficient and Scalable Parallel Code (White Paper), 2004.

3.3.2.1. Introduction

It is a commercial paid tool, which has been in constant development for the past 20 years. It provides a lot of features to customize the type of parallelization, number of threads, etc., providing additional flexibility to the users.

3.3.2.2. Implementation

The white paper talks about current problems of High Performance Computing (HPC). They analyse the requirements of users in the domain of HPC and accordingly design their product. It also talks about the final end user and major market for HPC and categorizes them into expert, non-expert and serial code users.

ParaWise makes use of efficient code analysis, enabling them to use OpenMP directives to be inserted into appropriate positions, thus introducing parallelization into a serial source code. They also experiment with Message Passing optimizations to further improve the benefits of parallelization.

3.3.2.3. Conclusion

The tool is quite exhaustive providing state of the art features to enable parallelization. The tool being interactive, requires users to provide valuable inputs during the process of parallelization. Thus, it is not completely automated. It also doesn't cover all possibilities which can be exploited to introduce parallelization.

3.3.3. Idan Mosseri, Lee-or Alon, Re’em Harel, and Gal Oren, ComPar: Optimized Multi-Compiler for Automatic OpenMP S2S Parallelization, 2020.

3.3.3.1. Introduction

- Talks about other tools: AutoPar, Par4All, and Cetus.
- ComPar is an innovative approach to parallelization and uses a source to source multi compiler. It utilizes code segmentation along with fusion, with the use of hyper parameters for tuning.
- Performance is improved with no manual intervention by tuning the hyper parameters, so as to obtain the best possible parallelised program. This is done while ensuring the validity of the input source code.
- The results obtained for analysis are on the NAS and PolyBench benchmarks.

3.3.3.2. Implementation

- The S2S compiler working is as follows:
    - The source code is parsed into an Abstract Syntax Tree (AST)
    - The AST is analysed, so as to obtain data dependencies; which is then used to find code segments to parallelise and the required directives for parallelization are inserted. This is done to optimise the code. This process is repeated until convergence is reached.
    - After convergence, the AST is then reverted back to the source code language as necessitated.
- As of now, no automatic parallelization tool or compiler is capable of replacing programmer insights. Human programmers still outperform compilers in this regard. This is due to information gathering required for parallelization from an AST is difficult in generalised scenarios. This is a major disadvantage for automatic parallelising tools.

- An example for the above being, function side effects. Also, relevant information that plays a major role in parallelization such as the load of computation, scheduling optimization, and available threads etc.
- The tuning of OpenMP directives to optimise performance automatically is well established. We might not be able to work here.
- Parallelising directives of OpenMP target sections of the code separately, unlike MPI. The sections may have different working fashions hence no unified compilation of an entire source code with one unique Source to Source compiler can yield optimised result. Thus, by using code segmentation, and usage of different Source to Source compilers will result in better parallelization of the entire input source code.

3.3.3.3. Conclusion

- Despite the fact that resources required for ComPar are greater, it is compensated by better results compared to other S2S compilers, especially ones that need tuning of hyperparameters by an external user.
- ComPar can be accessed and viewed: https://github.com/Scientific-Computing-Lab-NRCN/compar
- After testing several S2S compilers, the conclusion that can be drawn is that despite individual advantages and drawbacks, none of them are superior in results compared to the rest in all scenarios.
- The combination of all the above compilers yields the best possible optimised result, under hyperparameters tuned for the specific input. This combination of compilers is expensive computationally due to the sweep of the entire space of possibilities of the hyperparameters, for each possibility the performance is estimated so as to make the best possible optimisation.

3.3.4. Hamid Arabnejad, João Bispo, Jorge G. Barbosa, João M.P. Cardoso, AutoPar-Clava: An Automatic Parallelization source-to-source tool for C code applications, 2018.

- Performs parallelization of loops only
- Performs Static analysis without any runtime info and any additional info from the user.
- Checks for dependencies within the candidate loops
- Checks the dependencies for static variables - Performs liveness analysis and determines how the variables are referenced. Finds the Read, Write pattern that exists and determines if any dependency exists
- Checks dependencies for arrays - Using array subscripts to determine if loop iterations are independent which is done using existing methods such as GCD and Extended GCD.
- If no dependency exists, the OpenMP directive that it matches is found. Also, it categorizes the variables according to the OpenMP classes.
- Generates code annotated with OpenMP directives.

3.3.5. A Review of Parallelization Tools and Introduction to Easypar

3.3.5.1. Introduction:

The paper is a survey of existing tools for auto-parallelization. They have classified these tools into various categories based on the degree of automation and on the parallel programming language. The classification is also on the following criteria:

3.3.5.1.1. Based on Parallelization stage:

"Parallelization process is a systematic process, especially automatic parallelization. First stage of the parallelization process is parallelization identification. The code is parsed and analyzed (static or dynamic dependency analysis) to search for the code sections that can be executed concurrently."

3.3.5.1.2. Based on era:

The tools are classified as either First Generation Tools(FTG) or Second Generation Tools (STG) based on when they were developed

3.3.5.1.3. Based on Graphical Assistance Tools:

This classification is based on if the tool provides a visualisation of the parallelization phase or not.This information could reduce burden on programmers in understanding the generated program.

3.3.5.2. Tools Described:

3.3.5.2.1. First Generation Tools:

- Automatic and Interactive parallelization
- Object Based parallel Programming assist
- Animation Choreographer
- DEEP Development Environment
- PTP-PLDT by IBM
- GRED
- VISO
- The SUIF compiler

3.3.5.2.2. Second Generation Tools:

- Alchemist
- DProf
- Prospector
- Coarse Grain Parallelization
- LoopSampler
- iPAT/OMP
- Capo
- Kremlin
- Kismet
- Cilk++
- Holistic approach for auto-parallelization
- Polaris
- SD3
- Prism
- AutoFuture
- Vector Fabrics

- Pluto
- Par4All
- Cetus
- S2P
- EasyPar

3.3.5.3. Conclusion:

The paper provides a brief overview of the work done in the past in the area of auto-parallelization. By providing robust classifications, it renders future literature surveys simpler. The paper also presents EasyPar as a parallelization tool that can be used to not only generate parallel code but also to generate code to run on GPU's. The extensive literature survey done by the authors eases the work for future work in the area, until such a time there has been significant changes to the domain.

3.4. AST Generation and Querying

3.4.1. Rose

Rose is an open-source compiler-based tool that is compatible with multiple programming languages such as C, C++, OpenMP, etc.

Rose tool is used for generating data dependencies, along with intermediate representations such as Abstract Syntax Trees. It provides for compilation and decompilation processes between high-level code and intermediary code. This tool has been used in the paper mentioned in section 3.1.1.

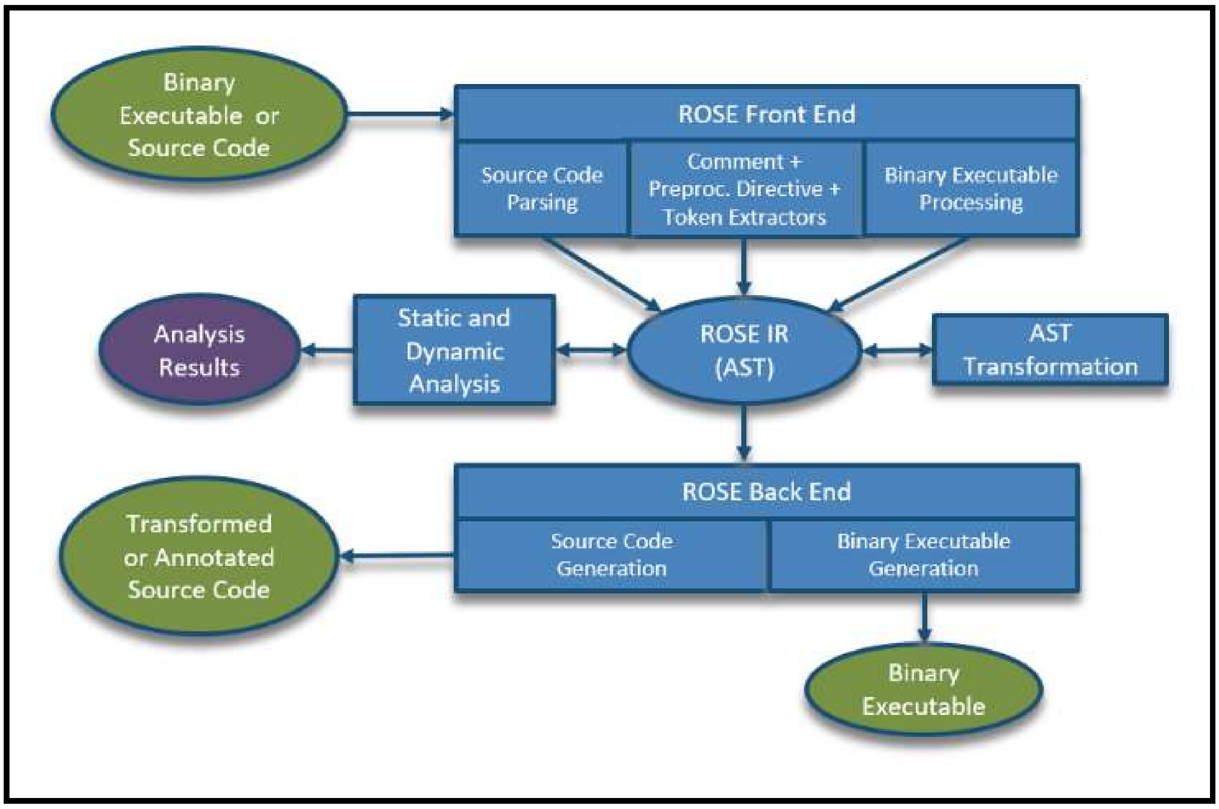

Binary Executable or Source Code
ROSE Front End
Source Code Parsing
Comment + Preproc. Directive + Token Extractors
Binary Executable Processing
Analysis Results
Static and Dynamic Analysis
ROSE IR (AST)
AST Transformation
ROSE Back End
Transformed or Annotated Source Code
Source Code Generation
Binary Executable Generation
Binary Executable

Figure 3: Working of ROSE Compiler

3.4.2. Clava/Lara - João Bispo, João M.P. Cardoso, Clava: C/C++ Source-to-Source compilation using LARA, 2020.

3.4.2.1. Introduction

Clava is a comprehensive compiler-based tool, which aims to extend the functionalities provided by Clang. It is based on a Javascript-based language called Lara. It works on an enriched Abstract Syntax Tree, with multiple attributes associated with each of the nodes. It provides useful information about data and control flow mechanisms involved in a program. It works for the programming languages of C and C++. In addition to supporting compilation and execution on Linux based systems, it also provides an online IDE.

3.4.2.2. Implementation

Following is the working of Clava:

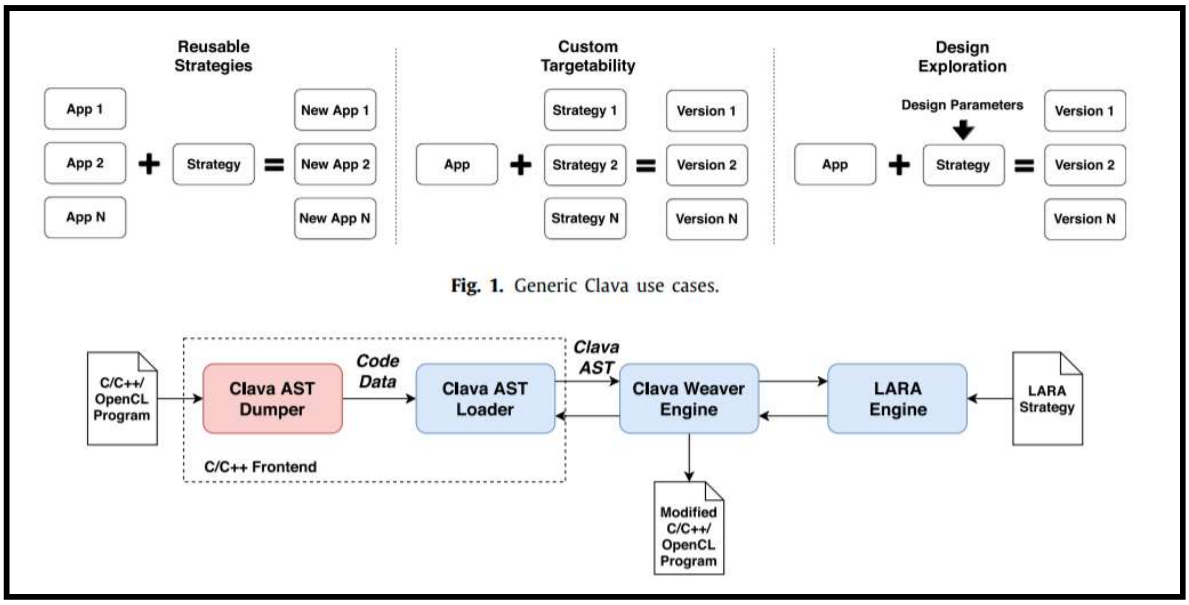


Figure 4: Working of CLAVA

3.4.2.3. Conclusion

The Clava tool is a very useful tool to identify data and control flow information from the input program. It has elaborate attribute information for each node in the Abstract Syntax Tree, which enables multiple functionalities. It also provides us with a means of modifying the program to enable functional parallelism (both intra and inter-functional).

3.5. Functional parallelism

3.5.1. Sean Rul, Hans Vandierendonck, Koen De Bosschere, Function Level Parallelism Driven by Data Dependencies, 2007.

3.5.1.1. Introduction

This paper proposes a method for acquiring and assimilating potential parallelism in programs. The proposed method is said to reveal the amount of parallelism present in the sequential programs and also suggest an appropriate parallel construct for the program such as a pipeline architecture, master-slave design and so on.

The method is not an exact analysis but rather a profile-based approach and hence, being dependent on the input, is not safe in terms of accuracy and correctness. It focuses more on measuring memory dependencies at a functional level and constructs two graph representations of the profile data: the interprocedural data flow graph - showing the data flow between functions and the data sharing graph - denoting the data structures used to share data. The visualisation of these graphs helps in finding the sections of data that could be modified and parallelised and revealing the data structures that may need synchronization for ensuring thread safety.

3.5.1.2. Implementation

- To construct a call graph, they record and form a caller/callee relations among functions, keeping a track of the number of times a function is executed, different callees of a function and the execution time it consumes

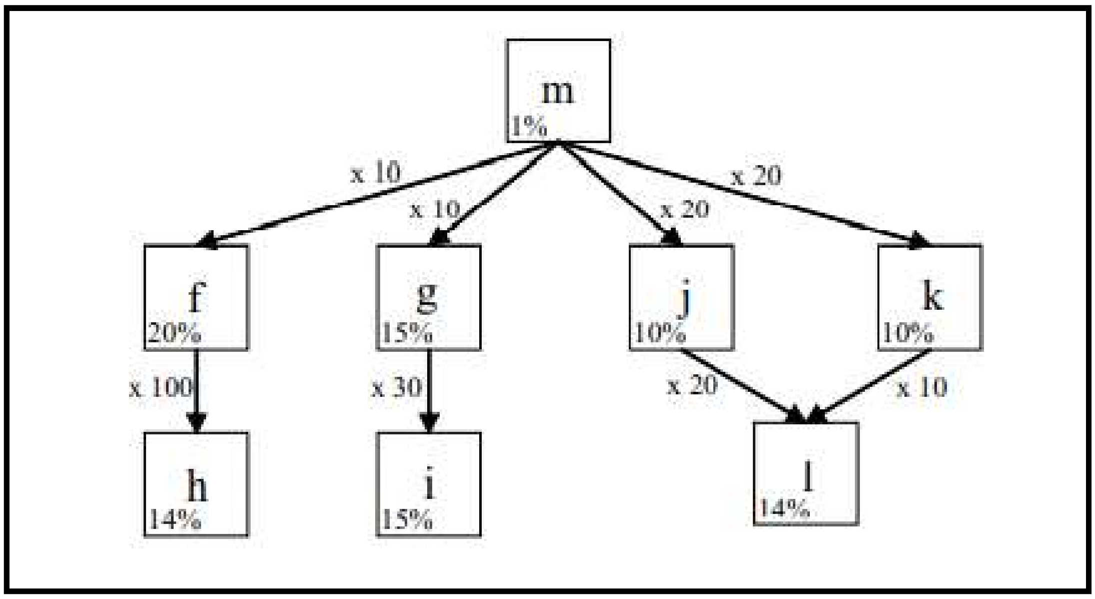
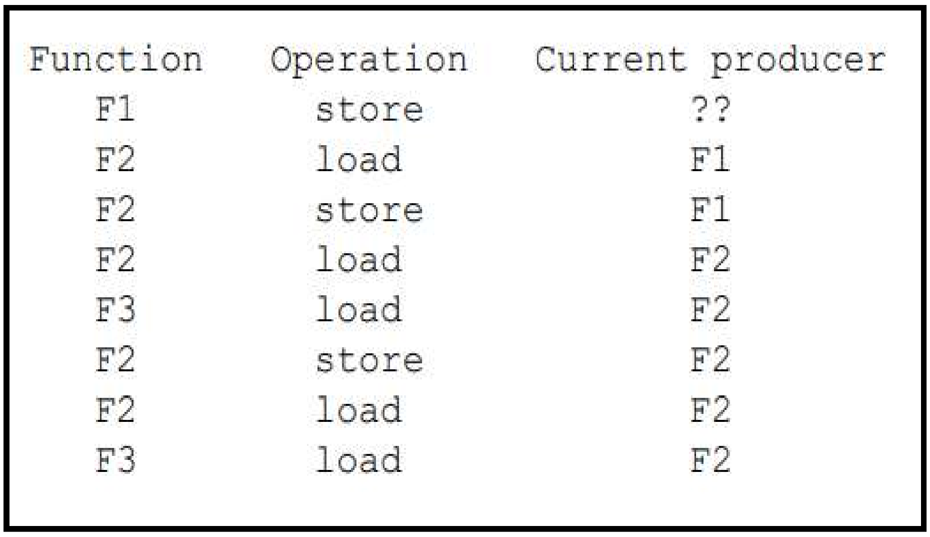


Figure 5: A Call Graph from data-dependency analysis

- Data Dependencies: Identifies and records in a matrix, inter function dependencies of data.



Function	Operation	Current producer
F1	store	??
F2	load	F1
F2	store	F1
F2	load	F2
F3	load	F2
F2	store	F2
F2	load	F2
F3	load	F2

Figure 6: Memory Dependencies

- The above figure shows a trace of memory operations for a specific variable.
- A matrix is built for each variable modified.
- Using these matrices, the functions are clustered based on the strength of connection among them and inter and intra cluster data streams are represented as a directed graph.

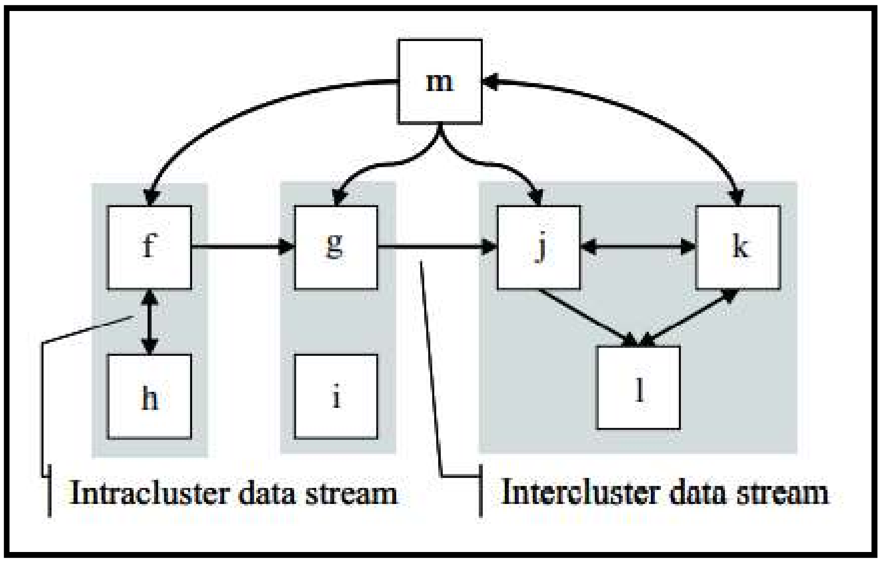


Figure 7: Interfunctional data flow graph

- To show how data sharing is done, a data-sharing graph is constructed consisting of function nodes and data nodes and the relation among them.
- A consumer is a function that reads data modified by a different function. The function writing such data will be the producer. Private consumption is the modification of local data. Constant consumption is the untraceable modification of data.

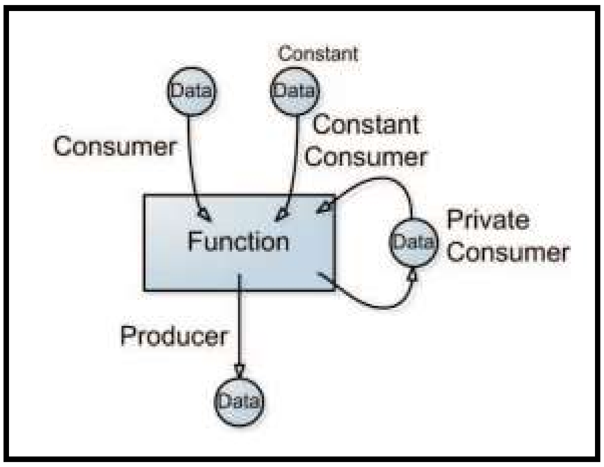


Figure 8: Classification of data dependencies

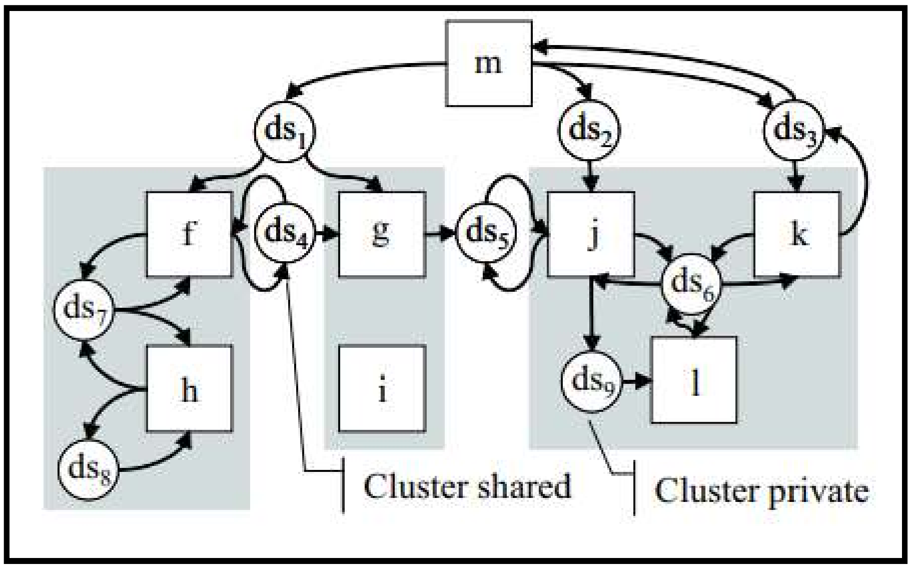


Figure 9: Data sharing graph

- The paradigms discussed:
  - **Master-Slave paradigm:** a Master thread creates several slave threads and assigns a part of the work to each. Synchronization is maintained by the use of barriers.
  - **Workpile paradigm:** Each thread requests a part of the work from a queue called the workpile. Threads can also push work to the pile.
  - **Pipeline paradigm:** Follows a simple producer-consumer relation, each stage in the pipeline produces data for the next stage of the pipeline

3.5.1.3. Conclusion

For generating results, an attempt was made to parallelise a compression procedure in bzip2. Based on the graphical analysis discussed, 4 functional clusters were discovered and the pipeline paradigm was implemented. A similar analysis was made for the decompression process. The results:

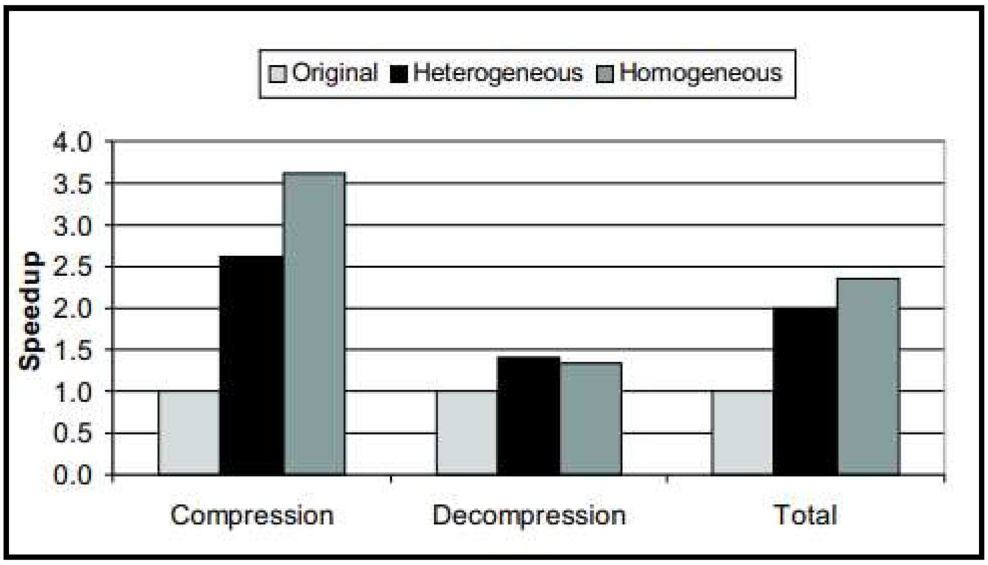


Figure 10: Speedup results of parallelized bzip2

A parallel programmer can use this as a tool to detect and automate, to some extent, the parallelisation of code, but he will still need to validate its correctness.

3.6. Parallel programming libraries

3.6.1. OpenMP

- OpenMP provides support for parallel programming in C, C++, Fortran in shared memory environments.
- The programmer has to explicitly mention the regions that can be parallelized. This is done using the OpenMP directives which are used to annotate the sequential code. We can define the number of threads to be used as well. OpenMP initiates those many threads and runs the parallel region across these threads.
- Usage: Works using pragma directives
  #pragma omp directive_name more_options
- Effectively used to parallelize code by hand (programmer needs to be aware of what can be parallelized, does not need to know how to write code for parallelizing it)
- For example, a loop can be fully parallelised, or partially parallelised, if you identify it, OpenMP can parallelize this part, you can customize by mentioning thread counts, etc.
- OpenMP provides for synchronization inherently.
- For task-level parallelism, OpenMP provides a “parallel” construct using which we can manually define the regions of code that can be executed independently of the rest of the program.

3.6.2. Pthreads

- Library to spawn and manage POSIX threads
- The library is considered to be very effective for multi-processor and/or multi-core systems. They are useful when there is the scheduling of processes on specific processors/threads so as to gain improvements in both distributed and parallel computing.
- The overhead of using “fork” and spawning new processes is higher than the usage of threads. This is due to the system not needing to initialize the new system virtual memory space and environment for every thread usage.
- The effectiveness of threads is higher on multiprocessor systems, but still, gains can be found on uniprocessors as well. This is due to the exploitation of latency in I/O and other possible interrupts.

- C++ thread library is built on top of Pthreads

GNU Parallel Algorithms:

- C++ provides a few parallel algorithms for a few standard sequential algorithms
- These are provided under different header files
- These are provided in a separate namespace as GNU extensions
- The following table shows some of the sequential algorithms and their equivalent parallel algorithms:

Algorithm	**Header**	**Parallel algorithm**	**Parallel header**
std::accumulate	numeric	__gnu_parallel::accumulate	parallel/numeric
std::adjacent_difference	numeric	__gnu_parallel::adjacent_difference	parallel/numeric
std::inner_product	numeric	__gnu_parallel::inner_product	parallel/numeric
std::partial_sum	numeric	__gnu_parallel::partial_sum	parallel/numeric
std::adjacent_find	algorithm	__gnu_parallel::adjacent_find	parallel/algorithm
std::count	algorithm	__gnu_parallel::count	parallel/algorithm
std::count_if	algorithm	__gnu_parallel::count_if	parallel/algorithm
std::equal	algorithm	__gnu_parallel::equal	parallel/algorithm
std::find	algorithm	__gnu_parallel::find	parallel/algorithm
std::find_if	algorithm	__gnu_parallel::find_if	parallel/algorithm
std::find_first_of	algorithm	__gnu_parallel::find_first_of	parallel/algorithm
std::for_each	algorithm	__gnu_parallel::for_each	parallel/algorith

			m
std::generate	algorithm	__gnu_parallel::generate	parallel/algorithm
std::generate_n	algorithm	__gnu_parallel::generate_n	parallel/algorithm
std::lexicographical_compare	algorithm	__gnu_parallel::lexicographical_compare	parallel/algorithm
std::mismatch	algorithm	__gnu_parallel::mismatch	parallel/algorithm
std::search	algorithm	__gnu_parallel::search	parallel/algorithm
std::search_n	algorithm	__gnu_parallel::search_n	parallel/algorithm
std::transform	algorithm	__gnu_parallel::transform	parallel/algorithm
std::replace	algorithm	__gnu_parallel::replace	parallel/algorithm
std::replace_if	algorithm	__gnu_parallel::replace_if	parallel/algorithm
std::max_element	algorithm	__gnu_parallel::max_element	parallel/algorithm
std::merge	algorithm	__gnu_parallel::merge	parallel/algorithm
std::min_element	algorithm	__gnu_parallel::min_element	parallel/algorithm
std::nth_element	algorithm	__gnu_parallel::nth_element	parallel/algorithm
std::partial_sort	algorithm	__gnu_parallel::partial_sort	parallel/algorithm
std::partition	algorithm	__gnu_parallel::partition	parallel/algorith

			m
std::random_shuffle	algorithm	__gnu_parallel::random_shuffle	parallel/algorithm
std::set_union	algorithm	__gnu_parallel::set_union	parallel/algorithm
std::set_intersection	algorithm	__gnu_parallel::set_intersection	parallel/algorithm
std::set_symmetric_difference	algorithm	__gnu_parallel::set_symmetric_difference	parallel/algorithm
std::set_difference	algorithm	__gnu_parallel::set_difference	parallel/algorithm
std::sort	algorithm	__gnu_parallel::sort	parallel/algorithm
std::stable_sort	algorithm	__gnu_parallel::stable_sort	parallel/algorithm
std::unique_copy	algorithm	__gnu_parallel::unique_copy	parallel/algorithm

3.7. Vector Representation of Source Code

3.7.1. code2vec: Learning Distributed Representations of Code

3.7.1.1. Introduction

- This paper enlists a method which uses a neural model to represent sections of source code as vector embeddings, which are essentially low-dimensional numerical vectors .
- These vector embeddings act as semantic representations of source code, capturing the meaning, intent and structure in the given section of code.
- The paper explains a method to first convert the code to its abstract syntax tree, then analyze different paths in the generated tree to produce individual vectors, which is finally aggregated into the final vector representation.
- Using these numerical representations of source code snippets, it is possible to use this in multiple applications such as code labelling, code captioning, clone detection and so on.

3.7.1.2. Implementation

- To capture the varying inherent importance of different sections of code within the given source program, it is necessary to identify the relative importance of these sections in influencing the final vector embeddings. The paper recognises this requirement and uses a path-based neural attention model.
- The paper also enlists a method to produce different vector embeddings for similar programs (not identical) to capture the subtle differences between the two programs. The attention model enables the calculation of a weighted average value on the attentions produced for different sections of code, extracted by the structure of an Abstract Syntax Tree.
- To represent the code in the form of an AST, numerical values are attached in a randomised manner in a bottom up approach to capture sub-trees. These numerical representations of the AST are then fed into the attention model which generates the final numerical representation of the input source code.

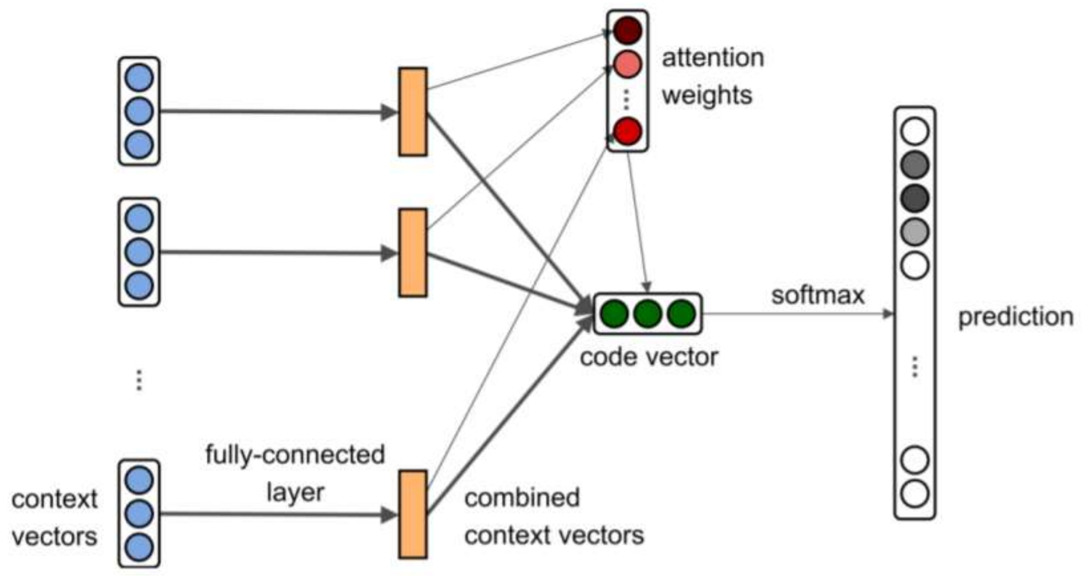


Figure 11: Architecture of Path-attention neural model used in Code2Vec

3.7.1.3. Conclusion

- The model requires an annotated training data set.
- The model is able to perform better than other popular vector representation techniques available due to the nature of the attention model concept.
- Vector embeddings capture subtle differences and can thus be used to perform algorithm recognition (program comprehension).

3.7.2. A Novel Neural Source Code Representation based on Abstract Syntax Tree

3.7.2.1. Introduction

- The paper proposes a method meant for representing source code, which performs better than other existing state-of-the-art models.
- This method uses an Abstract Syntax Tree based Neural Network model.
- Other existing techniques use the entire AST. However, there are issues related to the large size of the generated AST which could prove to be detrimental to the performance of the model. The proposed method chooses to break down the full AST into smaller statement trees.
- These smaller statement trees are then converted to vectors by capturing the lexical, as well as the syntactical information, present inherently in the source code.

3.7.2.2. Implementation

- The given source code is parsed into an Abstract Syntax Tree. It is split into smaller statement trees using the preorder traversal algorithm.
- Each of these smaller statement trees are encoded to vectors using “Statement Encoders”.
- After this, Bidirectional Gated Recurrent Units are used to capture and model the inherent structure of statements. Multiple hidden states of the Recurrent units are combined into one single vector, which is the final representation of the given source code.

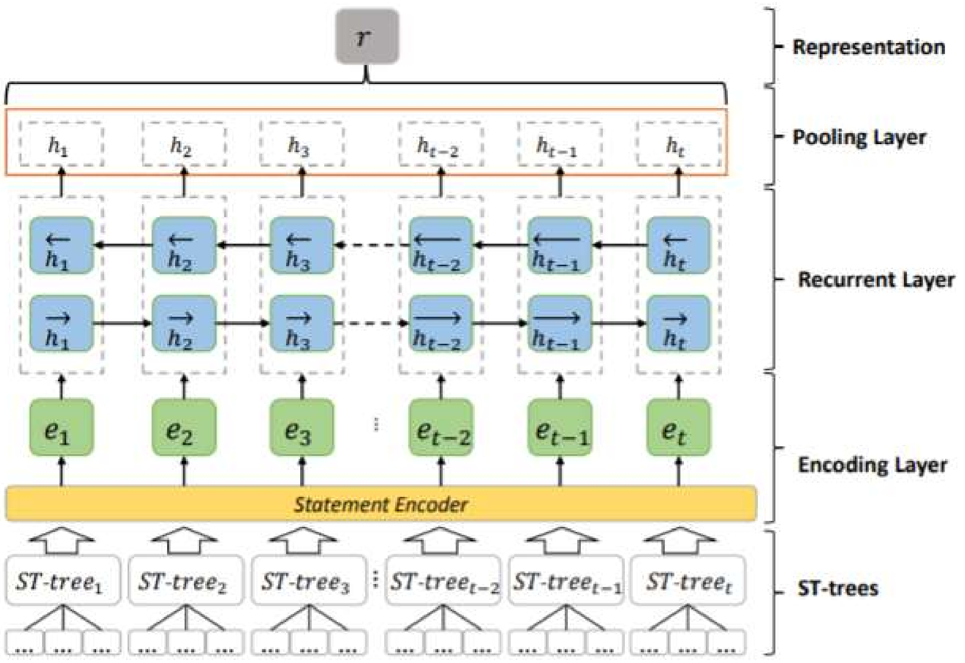

r
Representation
Pooling Layer
Recurrent Layer
Encoding Layer
Statement Encoder
ST-trees
h_1 h_2 h_3 h_{t-2} h_{t-1} h_t
e_1 e_2 e_3 e_{t-2} e_{t-1} e_t
$ST\text{-}tree_1$ $ST\text{-}tree_2$ $ST\text{-}tree_3$ $ST\text{-}tree_{t-2}$ $ST\text{-}tree_{t-1}$ $ST\text{-}tree_t$

Figure 12: Model architecture implemented in the paper

3.7.2.3. Conclusion

- The paper proposes an efficient approach to learn and represent vector embeddings of source code using “AST-based Neural Network (ASTNN)”.
- The model is successful in capturing both the lexical and syntactical information in the given code, in addition to identifying the code structure.
- Evaluation of the model is done on two popular program comprehension applications - source code classification and code clone detection.
- The model requires large-scale datasets to perform effectively.
- The model is not language-agnostic. It doesn’t perform well on multi-variety datasets.

4. Product Requirements Specification

4.1. Introduction

The purpose of this document is to elucidate the requirements of an Auto Parallelisation Software, which uses Program Comprehension, at both a functional and non-functional level. The document is intended to provide a brief description of the intricacies involved in building such a software, present an in-depth description of the functionalities included, and finally illuminate the non-functional requirements of the software.

4.2. Project Scope

The proposed project is an auto parallelisation tool that performs source to source compilation. It takes in the user code as input, brings it to an Intermediate Representation, analyses and segments the code into multiple parallel paradigms based on data and control dependencies and parallelizes the aforementioned segments efficiently.

Purpose:

Tool to automate the generation of parallel code covering a variety of problem types.

Benefits:

- Parallel programming helps in executing code efficiently.
- It saves time as it executes the applications in a shorter wall clock time.
- Larger, complex problems can be solved due to the ability to parallelize code.
- It reduces cost as sequential code leads to the under utilization of available hardware resources. Whereas parallel code tries to extract the best possible performance from the underlying hardware.
- Manual parallelization of code is a difficult and error-prone process. Automating the process of parallelization makes the entire process faster, easier and accurate, thereby saving valuable time and cost.
- The proposed tool should be able to cover a wide range of problem types. This generalization makes it indispensable to the development of efficient software.

Objectives:

- Comprehend the source code.
- Analyse data flow and data dependencies.
- Identify parallelizable segments.
- Identify hardware-dependent code optimizations.
- Generate and test equivalent parallel code.
- Check for correctness of the generated parallel code.

Goals:

- Automate the generation of parallel code.
- Cover a wide range of possibilities and variations with respect to the problem types.
- Generate equivalently correct parallel code without introducing additional vulnerabilities and issues.
- Parallel code generated should extract the best possible performance from the underlying hardware.
- Provide an efficient and accurate alternative to manually parallelizing code.

Coverage of the System:

- Ideally, the proposed software should be generalized enough to cover a wide range of possibilities with respect to the problem types that it can handle.
- The possible applications of the proposed software can be broadly divided into 2 categories:
    - **Parallelization of legacy software** - Softwares that have already been developed in a sequential manner can be parallelized by using our software, thereby resulting in considerable improvement in performance.
    - **As a tool to aid in the development of new software** - The number of developers skilled to write accurate parallel code is alarmingly low as most of them have been taught to develop code in a sequential manner. Thus, our proposed software can be used to develop new software that can be parallelized for best possible performance. This will end up saving valuable time and cost

for the software developers.

Limitations of the system:

- Generalizing the software to cover all possible types of problems could be a challenge.
- Effectively handling the issues that arise due to parallelization such as deadlocks, race conditions, starvation etc determines the success of the generated parallel code.

4.3. Product Perspective

Parallel computing has played a major role in a variety of areas such as computational simulations for scientific, engineering and commercial applications. The cost benefits that one gains along with the increase in performance provides compelling arguments in favour of parallel computing.

The need for parallelizing source code is ever increasing with the improvements in hardware and the advent of multicore and multithreaded processors. Without parallelizing source code, it is not possible to exploit the available hardware resources to extract the best possible performance.

However, it is difficult to find skilful developers capable of writing parallel code. It is also a difficult task to migrate and manually parallelise applications, the process being more error prone.

Automating the process of parallelizing source code would help overcome these problems, saving valuable time which would have otherwise been spent in studying, analysing, identifying and parallelizing the code manually.

We propose to design a novel pipeline to find sections of code that can be parallelised in the input program, through analyzing the algorithms used, integrating them and extracting common paradigms or cliches from them. The recognized regions of code are then modified into the corresponding parallelized equivalent, by using OpenMP directives and/or making calls to parallel implementations in C++ STL or other equivalent libraries.

4.4. Product Features

- Convert the input sequential source code to an intermediate representation (an Abstract Syntax Tree).
- Parse the tree to gather a basic understanding of the code to obtain predicates or rules.
- Identify and analyze the data and control dependencies.
- After aggregating all the information obtained until this point, the code is segmented into sections that can be parallelized.
- The parallelized equivalent of the source code is generated.
- Optimizations are done to fully utilize the underlying hardware resources.
- The generated code is tested and checked for correctness.
- Performance metrics are then used to evaluate the generated code.

4.5. Operating Environment

There are no specific requirements with respect to the environment per se.

But the system running the generated parallel code should have the necessary features to support multiprogramming. This is with respect to support for multiple threads, multiple processors and/or multiple cores.

4.6. General Constraints, Assumptions and Dependencies

- **Legal implications:** The source code being transformed by our pipeline should have the necessary permissions to access and update it. The tools, methods, approaches we are proposing to use should not infringe any existing copyrights, patents etc.

- **Usage limitations:**
  - Generalizing the parallelization process for any type of sequential code.
  - Effectively handling the issues that would arise due to parallelization such as deadlocks, race conditions, starvation etc.

- Handling the overheads involved in parallelizing the source code to gain sufficient speed up.

- It is assumed that the source code being processed is free from semantic and run-time errors that could manifest during or after the parallelization process.
- It is assumed that the end-user has a system that provides support for multiprogramming in terms of multiple threads, multiple processors, multiple cores.

4.7. Risks

- Failure of the dependency analysis of the code implies no guarantee to the correctness of the final transformation.
- Failure of a thread or starvation due to the creation of more threads or execution of other processes alongside would reduce the time efficiency or may even lead to a crash.
- If issues such as race conditions, deadlocks etc. are not handled properly, it could lead to serious issues.
- Hardware failures in terms of the failure of handling multiple threads, multiple processors, multiple cores could defeat the very purpose of parallelizing code.
- Version Compatibility problems: If hardware-specific optimizations are made, the generated code cannot be reused as is, on other systems. The code will have to be sent through our pipeline again and the corresponding parallel code should be generated specific to that hardware.

4.8. Functional Requirements

Validity Test on inputs:

- The inputs are the same as the ones being processed by the original sequential source code.
- So, as long as the input is valid for the original source code, it will be valid input for our generated parallel code as well.

The sequence of operations:

Intra-Function Parallelism:

- Convert source code to an Intermediate Representation (like an enriched Abstract Syntax Tree).
- Parse the tree with a top-down and requirement-driven approach to identify basic concepts. These concepts are represented either as rules or embeddings.
- Compose the basic concepts to form bigger known algorithms.
- If any algorithm is found, match the corresponding region of code and identify the data and control dependencies for the region.
- Choose from one of the parallel alternatives from the algorithm repository to modify and replace the corresponding region in the AST.
- Verify if the modified region fits in accurately by analysing the data and control dependencies around the region.
- After the AST is modified, we use our compiler to decompile the AST back to the original source code language.

Inter-Function Parallelism:

- We perform a similar analysis as before to retrieve the data and control flow and dependencies from the Abstract Syntax Tree of the source code.
- Based on the results of the analysis, an appropriate parallel paradigm is chosen
- This parallel paradigm is applied to the source code and an attempt is made to parallelise the same at a functional level
- A generator program generates a new source code that complies with the parallel paradigm suggested.
- Verification is then performed to ensure the correctness of the code.

Error handling and recovery:

Various errors and issues might arise due to parallelization. These include:

- Deadlocks
- Race conditions
- Starvation

It might also be difficult to effectively manage multiple threads. It may result in coherency issues and so on.

There is a requirement to handle these error conditions in the right manner.

<u>Consequences of parameters:</u>

- The information about the underlying hardware configuration is significant to generate the most optimized version of the parallel code possible, that extracts every bit of performance from the underlying hardware.
- Information needed to evaluate the performance metrics:
    - The wall clock execution time of the original sequential source code
    - The wall clock execution time of the generated parallel code
    - Number and type of hardware resources available
- The sequential source code being fed into our pipeline must be free from semantic and run-time errors as these will propagate to the generated parallel code and there is nothing that our software can do about it.

<u>Relation of output to input:</u>

Output is just the parallelized version of the input sequential source code.

The difference could be in terms of OpenMP pragma directives added or replacement of segments of code with the most efficient parallel version of it etc.

4.9. External Interface Requirements

4.9.1. User Interfaces

- The proposed pipeline, implemented as a tool, should have an easy to use interface - a GUI that lets the user upload the input sequential source code.
- The tool performs the different stages in the pipeline and outputs the generated parallel code if applicable.
- The performance metrics are also shown with the evaluated values.
- The relative timing of inputs and outputs: The generation of output should not take a considerable amount of time from the moment the input is fed into the pipeline.

4.9.2. Hardware Requirements

- Input is fed from standard input devices like keyboard, mouse etc.
- Output devices include standard output devices like display monitor, printer etc.
- The software product should be able to perform the required I/O through these devices.

- It should have the required permissions to spawn multiple threads and access all the available cores or processors.

4.9.3. Software Requirements

4.9.3.1. GCC/G++

Since the proposed software would be generating parallel code for the C++ language, we would need the gcc/g++ utility to compile and generate the required executables.

Version Requirements: GCC 5.0 onwards

4.9.3.2. CLAVA/LARA

The LARA language facilitates querying on the Abstract Syntax Tree of the input code. Clava is an S2S compiler for C/C++, built with LARA as its foundation. It provides better ways to modify and transform C/C++ code, statically and dynamically.

4.9.3.3. Python

Python utility is needed as some of the helper functions and code required to perform some functionalities will be written in python.

Version Requirements: Python 3.0 onwards

4.9.3.4. OpenMP

OpenMP provides support for parallel programming in C, C++, Fortran in shared memory environments. It was introduced in 1997. The programmer has to explicitly mention the regions that can be parallelized. This is done using the OpenMP directives which are used to annotate the sequential code. OpenMP initiates the required number of threads (can be set by the user) and runs the parallel region across these threads.

Version Requirements: OpenMP version 4.0 onwards

4.9.3.5. Pthreads

Pthreads library will be required to create, spawn and manage threads as the threads are the basic units of execution and the means by which parallelism can be achieved.

4.10. Non-Functional Requirements

4.10.1. Performance Requirement

- The performance at the very minimum should match that of the original sequential source code.
- The speedup gained from parallelizing should be substantial enough to overlook the overheads involved in generating the parallel code.
- Should ensure speedup or efficiency boost in a particular or all metrics as per user requirements
- The performance is evaluated against various metrics as follows. For the generated parallel code to be acceptable as per the user requirements, the calculated performance metrics should be greater than a certain threshold as specified by the user.

Performance Metrics :

- **Speedup:** ratio of sequential to parallel execution times
- **Efficiency:** ratio of performance to the computational resources used to gain that performance
- **Redundancy:** ratio of the number of instructions executed by the sequential to the parallel version of the code
- **Utilization:** the ratio between the computational resources used and the resources available
- **Restrictions:** limitations as defined by Amdahl’s law and other related laws

Software Quality Attributes:

- **Correctness:** The generated parallel code should compare with the original sequential source code in terms of correctness. The newly generated parallel code should successfully pass all the test cases that the original sequential source code passes. The outputs generated must be similar in content and format. The only considerable difference must be with respect to the speed up.
- **Maintainability:** The pipeline developed should have clean code and should be well documented to ensure that maintenance is easy.
- **Reliability:** It should reliably generate parallelized code without adding any issues.

- **Robustness:** It should be able to handle a wide spectrum of problem types in terms of generating the correct parallel code for it.
- **Testability:** It should be testable for different scenarios.
- **Usability:** It should have an easy and convenient to use interface that anyone can use with minimum additional knowledge.

4.10.2. Safety Requirements

- There should not be any loss or damage to the original source code that is being parallelized.
- The generated code should not damage or corrupt the data that it works on.

4.10.3. Security Requirements

- The proposed software should not copy or store any temporary files that may be generated.
- The data generated during the process of parallelization should be completely discarded after the required parallel code is generated. If not, this will lead to issues such as violation of security and privacy rules of the original software whose source code is being parallelized. This is very significant as the software might include sensitive data, patented and copyrighted technology.
- We should ensure that no security breaches are introduced in the modified source code
- The security standards of the original sequential source code should be maintained as such. Security vulnerabilities should not be introduced while generating the parallel code.

5. <u>System Design</u>

5.1. Introduction

In the earlier days, there was very little computational power that could be provided by the processors on a computer system. Software was written sequentially, which meant every instruction was executed one by one. This meant large software took a very long time to execute.

But there was an exponential boom in the development of computer hardware, with faster and more powerful processors coming into the market. The clock speed drastically increased which enabled more instructions to be executed in very short durations of time. However, this continuous increase in power and computational capacity was cut short a few years ago. This meant that the ever-increasing need for more and more computational power could no longer rely on developing hardware. We had to shift focus to the software side, to make intelligent and efficient use of available hardware.

Parallel computing is a paradigm that enables us to make efficient use of available hardware to run multiple instructions on multiple hardware resources, simultaneously. This could mean running separate tasks on multiple cores/threads available on a single system or running separate tasks on multiple systems connected in a cluster. This enables us to execute our software faster and saves valuable time in several use cases. Thus, we propose a novel approach to solve the problem of Parallelisation of sequential code, by the use of multiple techniques integrated into one package to convert any general sequential source code to its parallel equivalent source code. We achieve this primarily by using the concepts of Program comprehension, Task and Functional level parallelism.

The purpose of this document is to elucidate the high-level design of an Auto Parallelisation Software, that takes in sequential source code as its input and generates the parallelized equivalent. The document is intended to provide details about the design of the software.

The high-level design of the proposed software includes the following:

- A component or module capable of generating an enriched Abstract Syntax Tree (AST).

- The generated AST should be such that it should enable us to query the nodes to retrieve useful information about the input sequential source code.
- A component or module that queries the AST and analyzes the input code to generate data dependency graphs, control flow graphs, data flow graphs.
- A component or module that assimilates the information from the various graphs generated. The information extracted from this is used to determine the segments of code that are candidates for parallelization.
- A component or module that takes in these candidate segments of code, parallelizes them using different techniques proposed and developed by us to generate the equivalent parallel code.

5.2. Current System

Present-day tools designed to convert sequential to parallel code focus mainly on loop-level parallelization. Several tools implementing different techniques to achieve loop level parallelisation exist such as Pluto, Parawise, Compar, etc. These are also called Parallelizing Compilers. Some of the most popular ones and their approach is mentioned below in brief:

- **Pluto**

  This tool makes use of a concept called the Polyhedral model. This tool focuses on achieving loop-level parallelization and achieves a good degree of optimisation.

- **ParaWise**

  This is a commercially available paid tool designed to convert sequential code to parallel code pertaining to loops and some level of task parallelization. It provides ample customization options to users to control the level of parallelization.

- **ComPar**

  This is a semi-automatic tool designed to optimize code by fusing other source-to-source compilers' outputs that can be achieved from auto parallelizing compilers which need no manual intervention. It makes use of the outputs obtained from other S2S compilers such as AutoPar, Cetus and Par4All.

The approach we present is focused on achieving more generalised parallelization. We do so by trying to understand the intent and logic of code, termed as Program comprehension, which enables us to replace such code with its parallel equivalent, thus achieving task level parallelization. Our design also includes a Master-Worker scheduling algorithm which enables us to execute independent functions on multiple threads, thus achieving functional parallelization.

5.3. Design Details

The platforms, systems and processes that the proposed software depends on are:

- A process that generates an enriched AST: The Abstract Syntax Tree corresponding to the input sequential source code must be generated. Ubiquitous tools and compilers like gcc, clang generate AST. The generation of the AST should be modified to contain more information about the input sequential source code so that we can query the nodes of the enriched AST effectively.

- A process that generates the Data Dependency Graphs, Control Flow Graphs and Data Flow graphs: The generation of accurate Data Dependency Graphs, Control Flow Graphs and Data Flow Graphs is significant to ensure the success of the forthcoming stages in the proposed pipeline. Ubiquitous tools and compilers like gcc, clang generate some of these graphs. They will have to be modified to better suit our requirements in terms of the analysis that needs to be performed on it to extract meaningful insights about the input sequential source code.

- **GCC/g++**

  Since the proposed software would be generating parallel code for the C language, we would need the GCC utility to compile and generate the required executables.

  g++ will be required as the development of some of the functionality in our proposed software is done in C++.

  Version Requirements: GCC 5.0 onwards

- **CLAVA / LARA**

  The LARA language facilitates querying on the AST of the input code

  Clava is a Source to Source compiler for C/C++, built with LARA as its foundation. It provides better ways to modify and transform, statically and dynamically, C/C++ code.

- **Python**

  Python utility is needed as some of the helper functions and code required to perform some functionalities are written in python.

  Version Requirements: Python 3.0 onwards

- **OpenMP**

  One of the approaches we are considering involves generating parallel code that consists of OpenMP directives inserted at the right place in the input sequential source code. This is the reason the proposed software requires the OpenMP utility.

  Version Requirements: OpenMP version 4.0 onwards

- **Pthreads**

  Pthreads library will be required to create, spawn and manage threads as the threads are the basic units of execution and the means by which parallelism can be achieved.

- **Thread Pool:**

  The proposed software involves the implementation of thread pools to enable us to gain fine grained control over the executing threads. We use an open source implementation of thread pool, which is a C++ header only library which provides a definition of a simple thread pool, capable of accepting and executing tasks and provides a means to capture the return value of functions using the C++ concept of future and promises.

5.3.1. Novelty

Our proposed design integrates multiple techniques to convert sequential code to its parallel equivalent. We make use of program comprehension to understand the nature and intent of code, and replace relevant sections of code with the parallelised equivalent using mappings. Such sections/functions of code are then executed in parallel by identifying the underlying data and control dependencies, enabling us to execute independent portions of code simultaneously. To achieve fine grain control over efficient execution of such multiple functions, we design a scheduling algorithm based on the dependencies, allowing for better control and improving performance in an adaptive manner. This proposed methodology is novel based on our literature survey.

5.3.2. Innovativeness

The proposed methodology of scheduling threads after checking for data dependencies, and using program comprehension for improving individual function performance by replacing them with optimised parallel program is a technique that has not been tried before. The method allows for a new approach that may shed new perspective and also open avenues for more work on the front, especially due to the fine control gained by the way we have implemented thread scheduling.

5.3.3. Interoperability

The input so far is C/C++ code, and the resultant program is in C++. This allows for the program to be run on any machine after compilation. However, the performance will depend on the machine, the number of threads and any related machine dependent optimizations.

5.3.4. Performance

The performance at the very minimum should match that of the original sequential source code.

● The speedup gained from parallelizing should be substantial enough to overlook the overheads involved in generating the parallel code.
● Should ensure speedup or efficiency boost in a particular or all metrics as per user requirements

The performance is evaluated against various metrics applicable to our proposed software as follows:

- **Speedup:** ratio of sequential to parallel execution times.
- **Efficiency:** ratio of the performance to the computational resources used to gain that performance.
- **Redundancy:** ratio of the number of instructions executed by the sequential to the parallel version of the code.
- **Utilization:** ratio between the computational resources used and the resources available.

5.3.5. Security

The proposed methodology should not copy or store any temporary files that may be generated. Also, the data generated during the process of parallelization should be completely discarded after the required parallel code is generated. If not, this will lead to issues such as violation of security and privacy rules of the original software whose source code is being parallelized. This is very significant as the software might include sensitive data, patented and copyrighted technology.

We will need to ensure that no security breaches are introduced in the modified source code. That is, the security standards of the original sequential source code should be maintained as such.

5.3.6. Reliability

As the resultant code is in C++, there is no issue with the code working on different machines. However, we have to ensure that the function runs as it used to previously and produces the same results as before parallelization. To ensure this, we make sure that all the data dependencies are satisfied before the usage of any data by any thread. This is done by a thorough analysis of the data dependency graph and using that information in scheduling in a manner ensuring the satisfaction of data-dependencies.

5.3.7. Maintainability

All our source code is well documented with comments. Additionally, our report carries a detailed explanation of our ideas, and implementation. This is useful in understanding the purpose of our code and the design thought behind it. We have ensured from the start to keep the project in an organised manner, allowing for incremental improvements, while keeping the structure easy to understand.

5.3.8. Legacy to modernization

By using our methodology, upon scaling, it would be possible to convert legacy code written in a sequential manner into one that runs on parallel architecture. This would help improve the performance of the legacy code, with the improvement depending on the program, it’s domain and the design of the program.

5.3.9. Reusability

The progress made by our work could find potential reusability in GPU programming and possibility of automating conversion of programs into GPU specific architectures, like into CUDA or Vulkan programs. This could be an area of research where our methodology could have a possible impact, with some necessary modifications in the idea to suit those exact system requirements.

5.3.10. Application compatibility

Our methodology tries to be as system independent as possible, with regards to compatibility. We have tried to maximize the number of threads utilized, while not setting any minimum threshold. Since all our code is written in languages that have a wide range of support across systems and OS’s, we suspect no specific compatibility issues to arise. However, we do require support for OpenMP, gcc (along with g++) and python, as dependencies. We have assumed the underlying architecture to have random access memory, and also to have multiple-cores and threads.

5.3.11. Resource utilization

The very core intent of our research has been to improve the performance of programs by the utilisation of existing resources that are otherwise not being used. Our design so far uses as many threads as possible in it’s thread pool during scheduling of the client program, to run in parallel.

6. Implementation and Pseudo Code

In our research, we have set our goal to find the limit of parallelism we can achieve for the given sequential source code. The objective is to gain the maximum possible speedup that can be achieved by parallelizing a particular program. In this regard, we have experimented with different approaches that work on achieving parallelism at different levels of abstraction and hence providing different degrees of control over the execution of the program itself.

Some Assumptions made while trying out the different approaches during Capstone Phase 1:

- The input sequential source program consists of atomic functions that perform only one functionality.
- Our research on Program comprehension techniques to convert sequential to parallel code has provided us with valuable insights on analysing a program to obtain its intent(s). This enables us to predict and define a paradigm (or cliche or algorithm) associated with every function/region of code. The different paradigms are pre-defined and stored in a database, and the corresponding mappings to one or more optimised versions for the paradigm are defined and stored as well.
- In Phase 1, we have assumed we have the paradigm identifier and database set up, which enables us to modify and replace code within functions to achieve intra-function parallelisation. By assuming the availability of this functionality, we have been able to work on Intra and Inter level functional parallelism and show meaningful results at the end of Phase 1.
- We plan to work on the paradigm identifier and populate the corresponding database at the beginning of Phase 2.
- The cases we are trying to handle now predominantly involve programs with functions (procedural programming).
- The function calls are sequential, i.e there are no selection, looping statements encapsulating these function calls. However these can be present within the functions themselves.

6.1. Parallelization Phase

6.1.1 Method 1: Inter and Intra-Function Parallelism by AST Querying and replacement with OpenMP Directives

6.1.1.1 Details about the approach:

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is carried out using a tool called Clava.
- Next, querying of the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript and is compatible with the Clava tool, thereby allowing us to query the AST.
- The following two functionalities have been implemented:
  - **Functional parallelism**
    - Here, the primary objective is to find functions that are independent of each other.
    - Functions are said to be independent of each other if they are performing operations on different sets of data. If they are working on the same data, none of the functions involved should be modifying this data. However, they are allowed to read the data simultaneously.
    - To identify the candidate functions that can be run parallely (functional parallelism), data dependencies that exist are found.
    - This is done by performing read-write dependency analysis.
    - Two functions are inherently said to be candidates for functional parallelism if they work on different data. Hence these functions are independent of each other and can be parallelized.
    - If the functions being considered work on the same data, the following cases arise:
      - If the argument(s) used by the function(s) is/are passed by using the mechanism of pass by value, then any modification being made to the argument(s) is local to the function only. Hence the functions being considered are independent.

- If the argument(s) used by the function(s) is/are passed as constant parameters, then it means that these functions cannot modify these argument(s), hence the functions being considered are independent.
- If the argument(s) used by the function(s) is/are passed by using the mechanism of pass by reference, then any modifications being made to the argument(s) within a function will propagate across the entire program. Therefore read-write dependency analysis will have to be performed to find if there is indeed a write to such an argument happening within the function.
  - If a write is being performed to such an argument, then the functions cannot be parallelized, they have to be run in the same sequential order they are being called by the user, to maintain the intended semantics.
  - If there exists only read dependencies and there is no write being performed, then the functions are independent and hence can be run in parallel.
- It is not enough to only consider arguments being passed to the functions under consideration to decide whether they are independent. There can also be cases where the functions use and modify global variables. Read-write dependencies are analyzed for such global variables as well.
  - If the functions under consideration only read these global variables, then the functions are independent and can be run in parallel
  - If the functions under consideration write into these global variables, then the functions using these global variables cannot be parallelized, they have to be run in the same sequential order they are being called by the user, to maintain the intended semantics

■ Once the independent functions are established, they are parallelized by inserting OpenMP directives at accurate points in the code. This is done by creating OpenMP sections and associating each parallelizable function to one OpenMP section.

○ **Intra Function Parallelism (Parallelism within the function)**

■ Here, considering that the functions are atomic having only one purpose and that we know what each function is intended to do (we obtain this information from the defined paradigm identifier and pre-defined database, which we have assumed is available in Phase 1, on which we intend to work at beginning of Phase 2), we try to map the functions to the parallelized equivalent of it.

  - For example, if the user has a function to perform sort where the user has written his own logic, we can map it to the gnu_parallel:sort algorithm available under the parallel algorithms library in C++.
  - There are obviously going to be cases where the function cannot be mapped to an algorithm in the parallel algorithms library. In such a scenario, we would like to map the user's sequential code to the parallelized equivalent that we have written. Hence a database needs to be created to map different possible sequential algorithms to their parallelized equivalents. This has to be made comprehensive with respect to the number of different cases that can arise.
  - Once the mapping is identified, the user's code within the function is replaced by the mapped parallelized equivalent.
- The parallelized equivalent of the input sequential source code is generated by applying the above-mentioned functionalities.
- The speedup achieved was considerable as can be seen in the following graph:

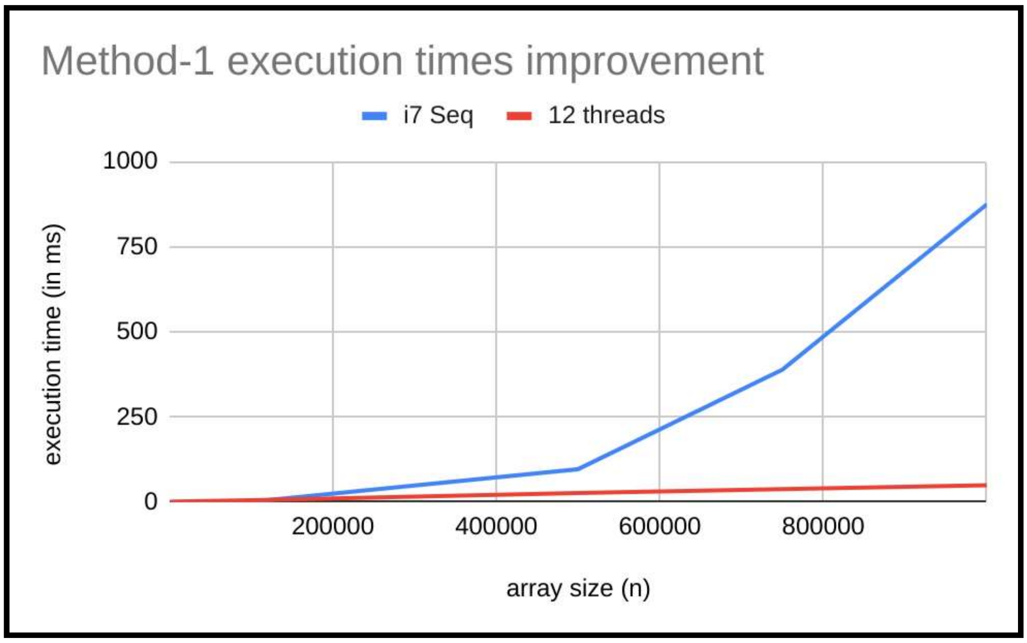


Figure 13 : Results of Method-1 on i7 9th gen

Input Code:

```
void sort(int* arr_,const int arr_n)
{
      for ( int x = 0;x < arr_n-1;++x)
      {
            int min_ = i;
            for(int y = x+1;y < arr_n;++y)
            {
                  if(arr_[y] < arr_[min_])
                  {
                        min_ = y;
                  }
            }
            int temp_ = arr_[x];
            arr_[x] = arr_[min_];
            arr_[min] = temp_;
      }
}

int my_max(int* a4, int n4)
{
      int max = a4[0];
      for ( int x = 1; i < n4; ++x)
      {
            if(a4[x]>max)
            {
                  max = a4[x];
            }
      }
      return max;
}

int my_min(int* const a4, const int n4)
{
      int min = a4[0];
      for ( int x = 1; x<n4; ++x)
      {
            if(a4[x]<min)
            {
                  min = a4[x];
            }
      }
      return min;
}
```

```
int main()
{
      int arr1[] = {7, 34, 5, 3, 1};
      int n = sizeof(arr1) / sizeof(arr1[0]);

      my_sort(arr1, n);
      int min1 = my_min(arr1, n);
      int max1 = my_max(arr1, n);
}
```

Output Code:

```
void my_sort(int *arr, int const arr_n) {
   __gnu_parallel::sort(arr, arr + arr_n );
}

int my_max(int *a4, int n4) {
   return *__gnu_parallel::max_element(a4, a4 + n4 );
}

int my_min(int * const a4, int const n4) {
   return *__gnu_parallel::min_element(a4, a4 + n4 );
}

int main() {
   int arr1[5] = {7, 34, 5, 3, 1};
   int n = sizeof(((arr1))) / sizeof(((arr1[0])));
   my_sort(arr1, n);
   #pragma omp parallel sections
   {
      #pragma omp section
      int min1 = my_min(arr1, n);
      #pragma omp section
      int max1 = my_max(arr1, n);
   }
}
```

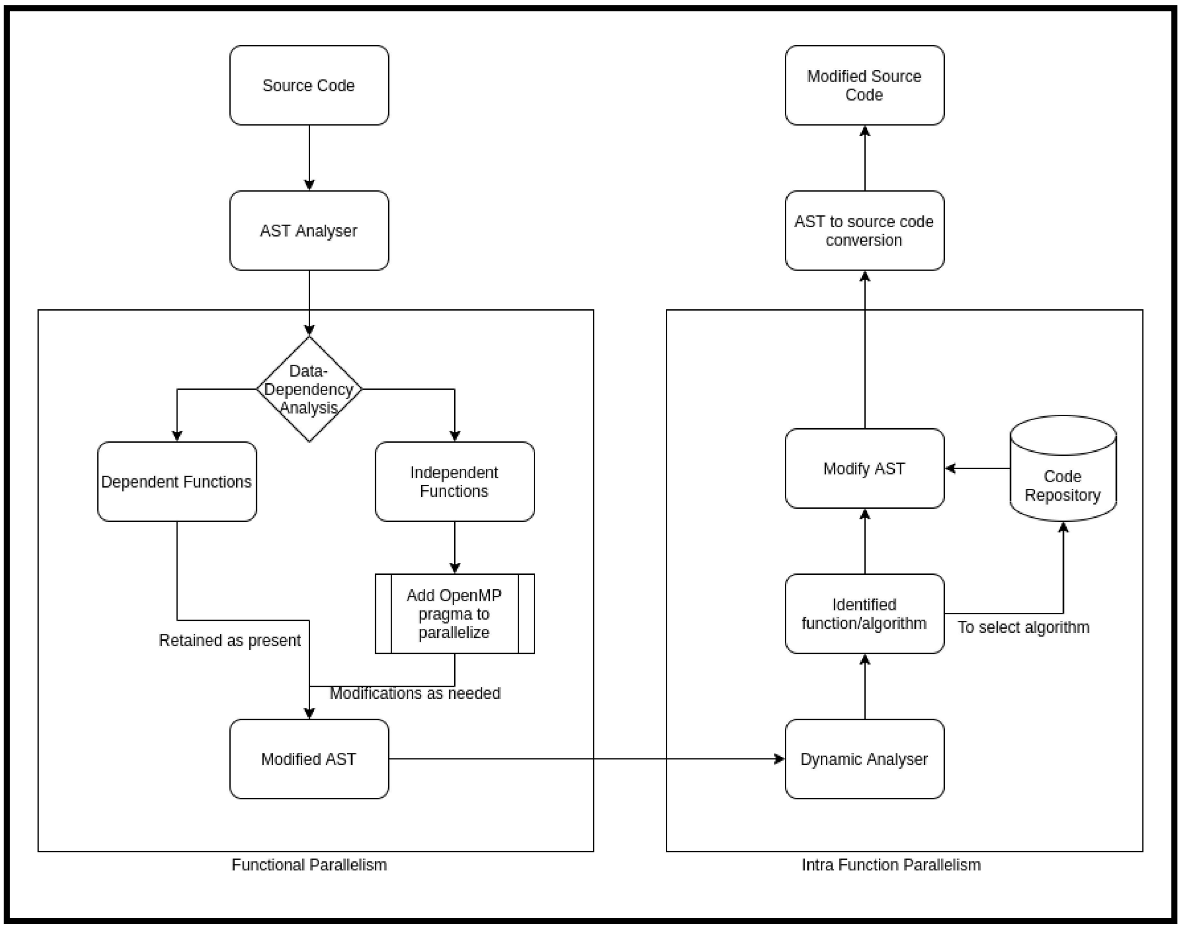


Figure 14: Method-1 implementation

6.1.1.2. Inferences from Method 1

- The degree of parallelism that could be achieved was still limited. This is because we add OpenMP directives and the rest of the details are abstracted. Parallelizing of the code based on the directives inserted, creating, spawning threads, assigning functions to the threads, scheduling and managing these threads are all abstracted as this is managed by the OpenMP library based on the directives we have inserted.
- Hence we don’t have the much-needed flexibility with respect to the above parameters, thereby limiting the parallelism that can be achieved.
- This method could also not support the grouping of independent functions together due to limitations with respect to the OpenMP directives.

- We realized that we would need to gain more fine-grained control over the actual execution of threads than what we could achieve using OpenMP.
- The above-mentioned reasons were the motive to consider a different approach - namely Method 2.

6.1.2. Method 2: Naive Thread Scheduling using C++ concepts of Promises and Futures

6.1.2.1. Details about the approach

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is done using a tool called Clava.
- Next, querying the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript that is compatible with the Clava tool, thereby allowing us to query the AST.
- We then perform a read-write dependency analysis to get significant information about the input sequential source code.
- The information assimilated from the read-write dependency analysis performed above combined with the ability to query the enriched AST is used to populate text files that will be used by the next phase in our pipeline.
- The information about the functions that are modifying the arguments and the arguments being modified by the respective functions are written into one text file.
- The order of function calls along with information about the function name, the arguments being passed to these functions are written into another text file.
- The next phase in this approach includes the naive thread scheduling algorithm written using C++ promises and futures.
- The text files generated are used by the thread scheduling algorithm.
- The information gathered from these text files is used to establish the order in which the functions can be called to exploit the possible parallelism.

- Functions that have no dependencies are executed in parallel.
- Functions that have dependencies are reordered in such a way that these functions are executed only after the dependencies are satisfied as mentioned in the original sequential code, thereby maintaining the semantics of the program.
- This is achieved by using the concept of C++ promises and futures.
- The use of C++ promises and futures ensures that no function that has a dependency is executed until all its dependencies are fulfilled.
- C++ thread libraries are used to create, spawn and schedule the threads.
- One function is executed as part of a thread.
- The promises and futures are associated with the threads based on the dependencies

Following are some test cases and obtained results:

Example 1: (Works with this method)

Input program (only the function calls inside main):

```
fn_A(arr1,n)
fn_B(arr1,n)
fn_C(arr1,n)
fn_D(arr2,n)
fn_E(arr2,n)
```

Constraints:

fn_A function modifies arr1, which implies fn_B and fn_C dependent on arr1, is in turn dependent on the execution of fn_A

Output:

```
std::promise<void> p_arr1_0
thread ti(fn_A, params);
thread ti(fn_D, params);
thread ti(fn_E, params);
std::future<void> f_arr1_1= p_arr1_0.get_future();
thread ti(fn_B, params);
thread ti(fn_C, params);
```

Analysis:

Function calls fn_B and fn_C with arguments arr1 need to be executed only after fn_A on arr1 finishes its execution.

However, fn_D and fn_E are two function calls on arr2, which can be independently executed.

Hence the reordering of function calls to schedule fn_A, fn_D, fn_E happen as expected. Future is called on a promise set on fn_A(arr1, n). This implies, there is a wait on fn_A(arr1, n) to finish its execution, only then the consecutive statements are executed. Thus fn_B and fn_C on arr1 are correctly executed on a possibly modified arr1 variable.

This is a test case that behaves as expected and can be scheduled using this method.

Example 2: (Not optimised with this method)

Input program (only the function calls inside main):

```
fn_A(arr1,n)
fn_B(arr1,n)
fn_C(arr1,n)
fn_A(arr2,n)
fn_B(arr2,n)
fn_C(arr2,n)
```

Constraints:

fn_A function modifies arr1

fn_A function modifies arr2

fn_B and fn_C is dependent of execution of fn_A

Output:

```
std::promise<void> p_arr1_0
thread ti(fn_A, params);
std::promise<void> p_arr2_0
thread ti(fn_A, params);
std::future<void> f_arr1_1= p_arr1_0.get_future();
thread ti(fn_B, params);
thread ti(fn_C, params);
std::future<void> f_arr2_1= p_arr2_0.get_future();
thread ti(fn_B, params);
thread ti(fn_C, params);
```

Analysis:

There are two function calls - fn_A(arr1, n) and fn_A(arr2, n), which are modifying their respective arguments (in this case, arr1 and arr2 respectively). This means that fn_A and fn_C function calls on arr1 and arr2 need to be executed only after the execution of fn_A on arr1 and arr2 respectively.

The fn_B and fn_C calls can independently execute as long as their dependency on corresponding fn_A calls is satisfied. However, in the output generated by this method, there is an unnecessary wait on completion of fn_A(arr1, n), which is affecting the execution of fn_B and fn_C on arr2, something which is completely independent of this particular fn_A call. Hence such cases cannot be handled using this method.

The future approach is to obtain fine grained control through our own thread scheduling algorithm.

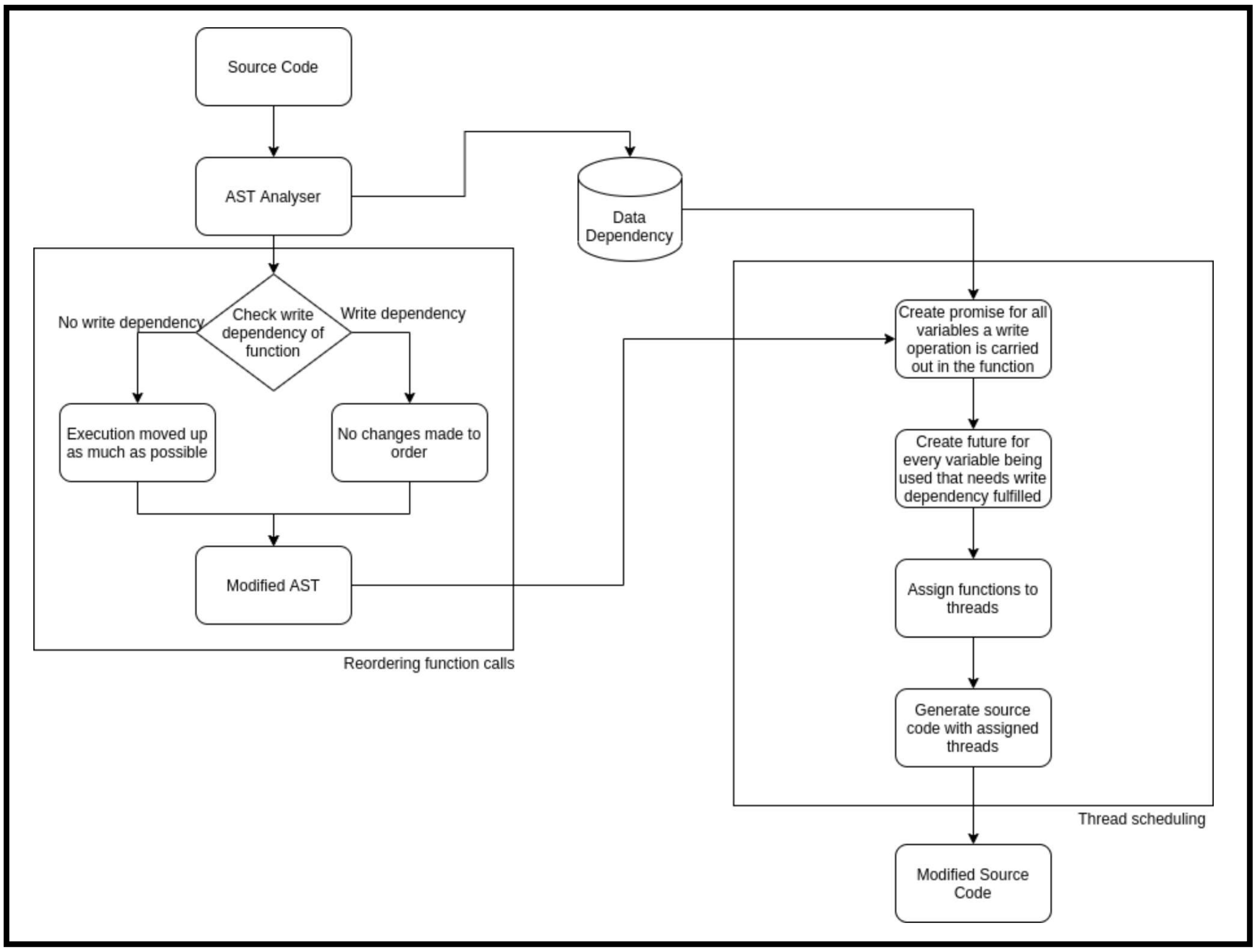

Source Code
AST Analyser
Data Dependency
No write dependency
Check write dependency of function
Write dependency
Execution moved up as much as possible
No changes made to order
Modified AST
Reordering function calls
Create promise for all variables a write operation is carried out in the function
Create future for every variable being used that needs write dependency fulfilled
Assign functions to threads
Generate source code with assigned threads
Thread scheduling
Modified Source Code

Figure 15: Method-2 implementation

6.1.2.2. Inferences from Method 2:

- This method provided a more fine grained control over thread creation, management and scheduling giving more flexibility and opportunity to maximize the parallelism that can be achieved.
- This method allowed us to go a level down in terms of the abstraction as we are dealing with threads directly in this approach.
- One of the major drawbacks of this approach was that grouping of function calls could not be performed properly.
- The reordering of functions had limitations to the cases where it was accurate and provided sufficient speed up.

- As grouping of functions could not be done, there were cases where certain functions ended up waiting for the completion of execution of certain other functions they didn't even depend on, even when the required dependencies for the execution of this particular function had already been satisfied.
- This is not desirable and thereby we needed a more fine grained control over thread scheduling to be able to achieve the maximum possible speedup by parallelism.
- This was the intent behind our next approach - namely Method 3

6.1.3. Method 3: Optimised Thread Scheduling for Functions using Master-Worker based approach to achieve Functional Parallelism

6.1.3.1. Details about the approach:

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is done using a tool called Clava.
- Next, querying the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript that is compatible with the Clava tool, thereby allowing us to query the AST.
- We then perform a read-write dependency analysis to get significant information about the input sequential source code.
- The information assimilated from the read-write dependency analysis performed above combined with the ability to query the enriched AST is used to populate text files that will be used by the next phase in our pipeline.
- The information about the functions that are modifying the arguments along with the information about the arguments being modified by these functions is written into one text file.
- The order of function calls along with information about the function name, the return type of the function, the arguments being passed to these functions, the parameters of the

functions, the data types of both the arguments and the parameters are written into another text file.

- The information about the functions, their return types, the line number where the function is being called and the variable to which the returned value is being assigned (in case of a non - void return type).
- The next phase in this approach is the thread scheduling algorithm.
- The text files populated by the previous phase are used by the thread scheduling algorithm.
- The information from these text files is assimilated and stored in appropriate data structures.
- The main idea behind this approach is to follow a master worker based method to ensure the execution of processes that are independent of each other simultaneously and ensure the right order of execution of dependent functions, to maintain the correctness of execution.
- Here, there are two master threads that are always running.
    - One master thread that schedules the functions to the worker threads
    - One master thread that tracks the worker threads to know their status
- We make use of 2 separate task queues - ready and wait queues.
- The wait queue is intended for those functions which are dependent on a previously called function(s). This implies that such functions need to wait until its dependency functions finish their execution. Such functions get appended into the wait queue.
- Once the dependency functions finish their execution, the functions waiting in the wait queue are moved into the ready queue, implying they are ready to be executed. Functions inside the ready queue are assigned separate threads for execution. Once assigned, such functions are dequeued from the ready queue.
- To ensure the synchronization is maintained in common data structures used to carry out the scheduling process, such as the ready and wait queues, other lists used to maintain dependent functions and corresponding arguments, mutex locks have been used to prevent any race conditions.
- To make our scheduling algorithm generalized for any sorts of client programs, we make use of a program generating technique, which produces the parallelised version of the client code, with appropriate thread allocation and mutex locks in place.
- To avoid excessive overheads generated due to the allocation of new threads and the deallocation of existing threads when the processes finish their execution, we make use of a thread pool. A thread pool uses a fixed number of threads, which remain allocated until all

the processes finish their execution. It makes use of the C++ concepts of ‘future’ and ‘promise’ to obtain the return value of functions that are assigned as processes to threads. This avoids repeated cycles of allocation and deallocation of threads and improves efficiency.

- Processes are assigned to the thread pool, which maintains and coordinates the assignment of the process to one of the available threads. To exploit the full hardware potential available on the system, we assign all available hardware threads to the thread pool, thereby increasing performance.

Input program (only the function calls inside main):

```
fn_A(arr1,n)
fn_B(arr1,n)
fn_C(arr1,n)
fn_A(arr2,n)
fn_B(arr2,n)
fn_C(arr2,n)
```

Constraints:

fn_A function modifies arr1

fn_A function modifies arr2

Output:

In the generated parallel program, following is the sequence of events:

```
// declare "special" array which holds all arguments of function calls
presently being executed
// define two master thread functions to update special array and move
functions from wait queue to ready queue when there are no conflicting
dependencies
// Push function call to either ready or wait queue depending on
whether any of its arguments is present in the special array presently
push_to_ready_queue(fn_A, arr1, n)
```

```
push_to_wait_queue(fn_B, arr1, n);
push_to_wait_queue(fn_C, arr1, n);
push_to_ready_queue(fn_A, arr2, n);
push_to_wait_queue(fn_A, arr2, n);
push_to_wait_queue(fn_A, arr2, n);
```

Analysis:

The input program was the same as the one used in the previous method.

Previously in Method 2, the fn_A(arr2, n) was waiting unnecessarily until fn_B and fn_C finished their respective executions (which were indirectly waiting for fn_A to finish its execution). However, with the use of dynamic wait and ready queues, functions are pushed to execution queues purely based on their arguments dependency on currently executing functions. In this case, fn_A(arr2, n) wouldn’t need to wait on anything since none of its arguments (both arr2 and n) are presently not being modified by any previous methods. Thus, there is no unnecessary waiting and execution flow is continuous.

Thus, in this method we were able to ensure independence among the two dependent function clusters through fine grained control over the scheduling of threads. Execution of the first cluster is independent of the execution of the second cluster. Intra-cluster dependency is still maintained implying that execution of fn_B and fn_C on arr1 would still wait on the execution of fn_A on arr1. By having fine grained control over the thread scheduling algorithm, we were able to fix the inter-cluster dependency issue that was present in the previous method.

Comparison of execution times between sequential program and the generated parallel program:

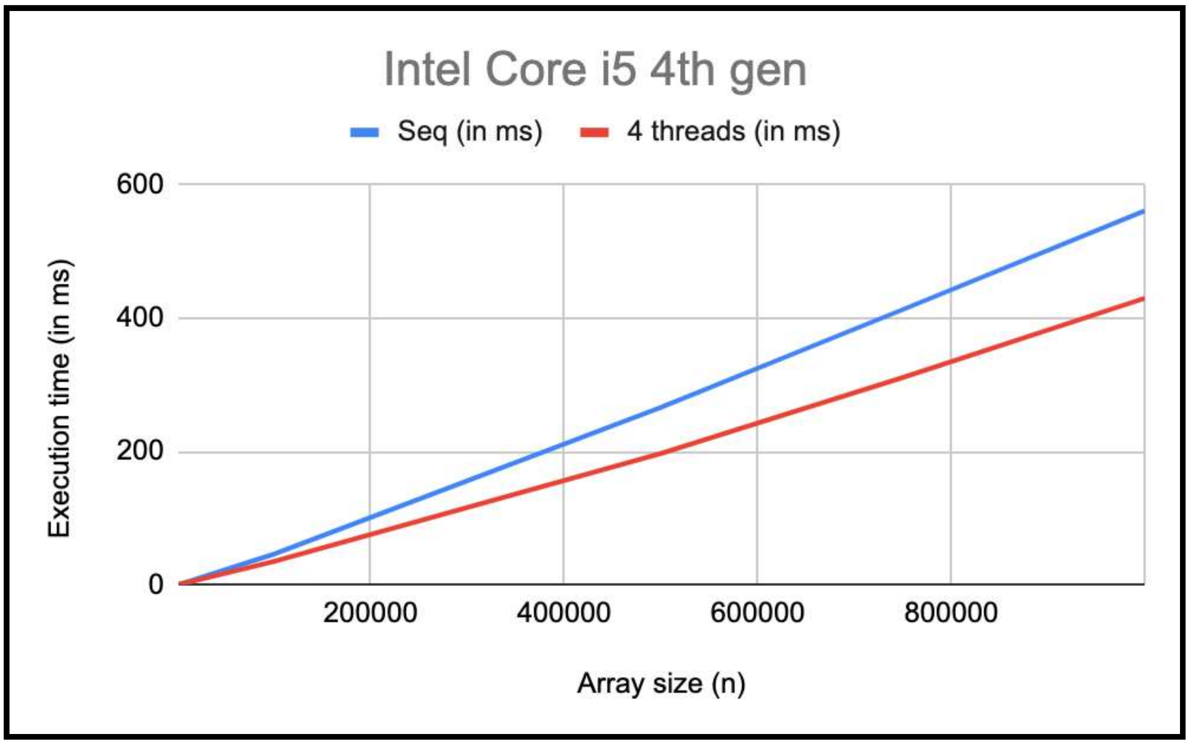


Figure 16: Results of Method-3 on i5 4th gen

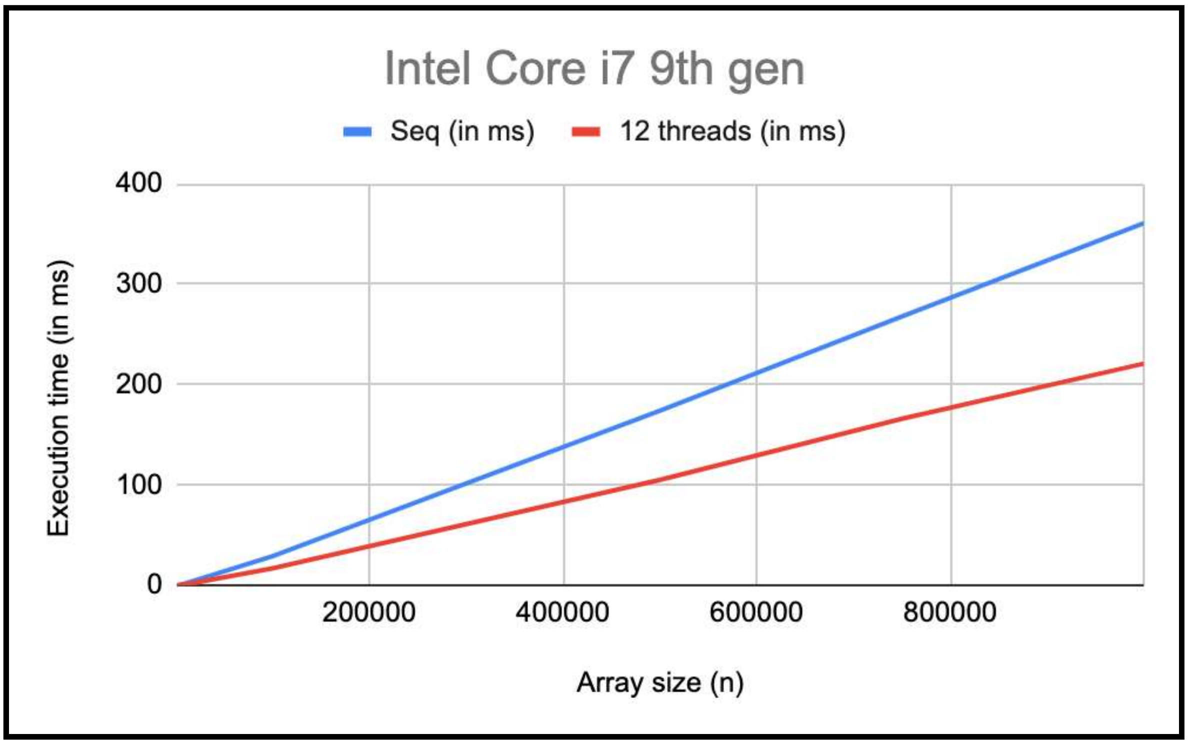


Figure 17: Results of Method-3 on i7 9th gen

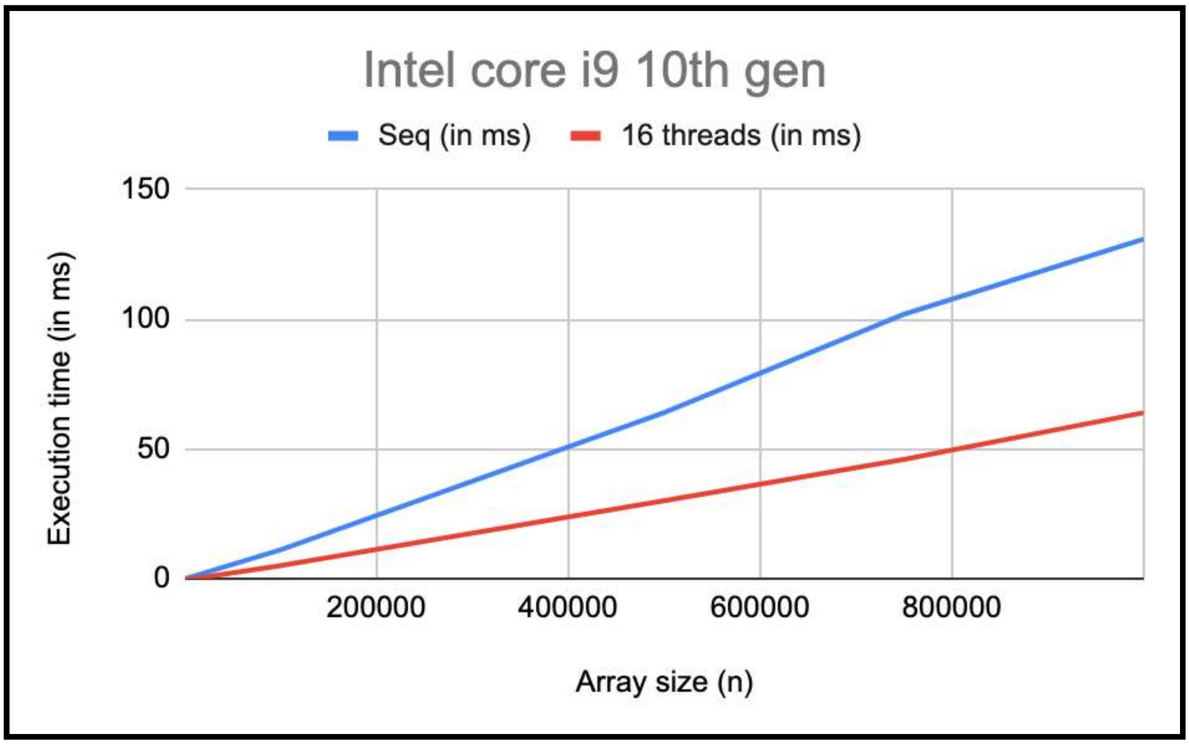


Figure 18 : Results of Method-3 on i9 10th gen

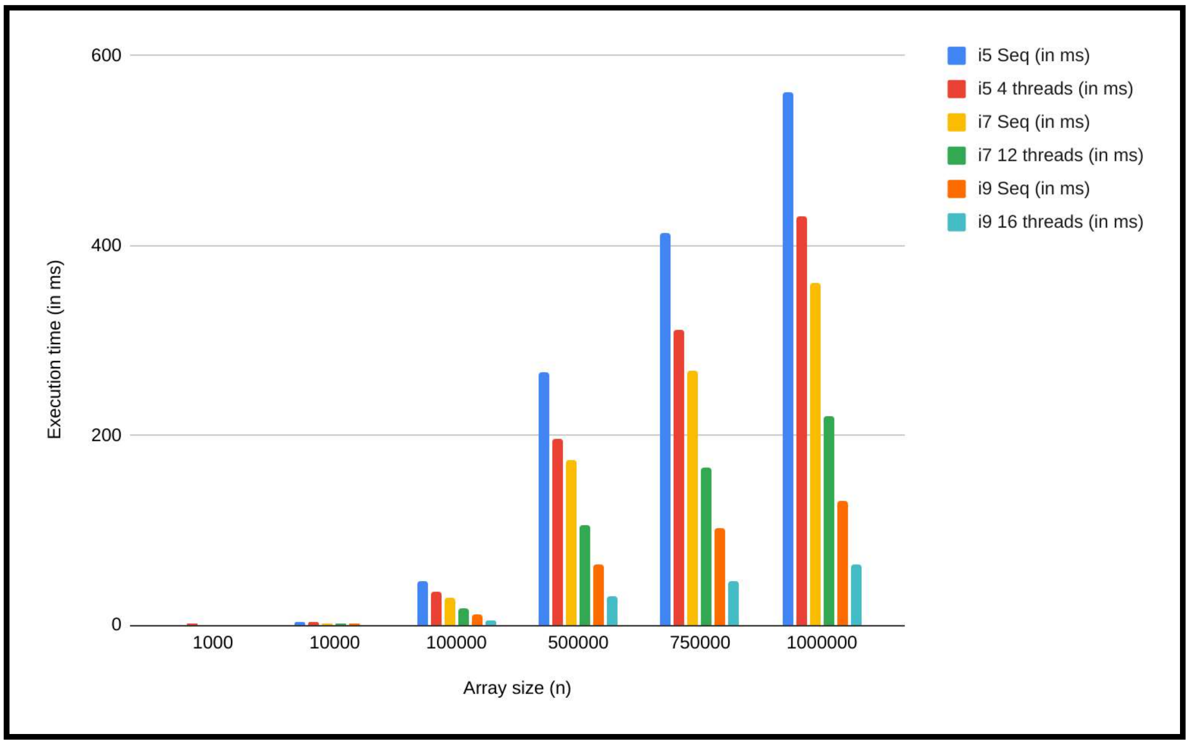


Figure 19 : Comparison of execution times of Sequential vs Parallel program across multiple CPU Architectures

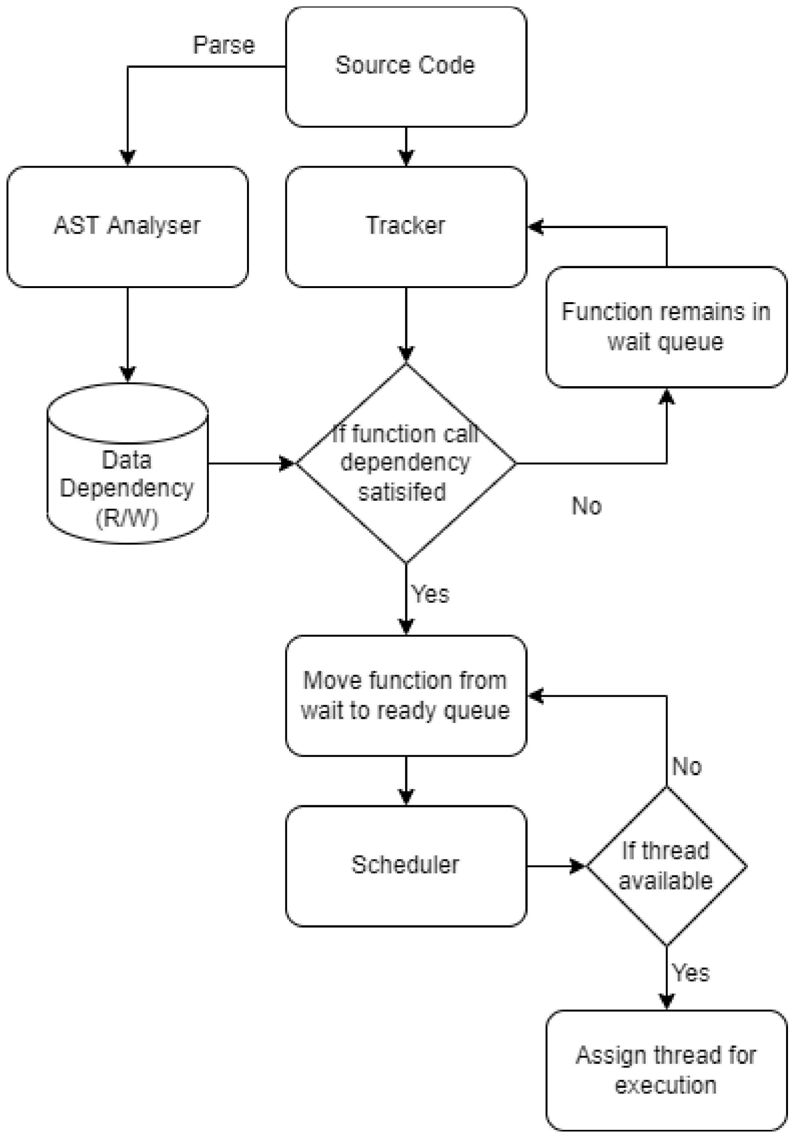

Parse
Source Code
AST Analyser
Tracker
Function remains in wait queue
Data Dependency (R/W)
If function call dependency satisifed
No
Yes
Move function from wait to ready queue
No
Scheduler
If thread available
Yes
Assign thread for execution

Figure 20: Method 3 implementation

6.1.3.2. Inferences from Method 3:

- This method provided a more fine grained control over thread scheduling allowing more flexibility and opportunity to maximize the parallelism that can be achieved.
- This method allowed us to go a level down in terms of the abstraction as we are dealing with threads directly in this approach.

- By controlling the scheduling of threads, the reordering of function calls could be done in a more accurate manner so as to gain the best possible speedup through parallelism.
- Through the use of semaphores and locks, we ensure there is synchronization among shared data

6.1.4. Method-4: Optimised Thread Scheduling for Functions using Non Master-Worker based approach to achieve Functional Parallelism

6.1.4.1 Implementation Details

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is done using a tool called Clava.
- Next, querying the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript that is compatible with the Clava tool, thereby allowing us to query the AST.
- We then perform a read-write dependency analysis to get significant information about the input sequential source code.
- The information assimilated from the read-write dependency analysis performed above combined with the ability to query the enriched AST is used to populate text files that will be used by the next phase in our pipeline.
- The information about the functions that are modifying the arguments along with the information about the arguments being modified by these functions is written into one text file.
- The order of function calls along with information about the function name, the return type of the function, the arguments being passed to these functions, the parameters of the functions, the data types of both the arguments and the parameters are written into another text file.

- The information about the functions, their return types, the line number where the function is being called and the variable to which the returned value is being assigned (in case of a non - void return type) is written into another text file.
- The information about the occurrences of function calls and the line numbers before which the respective functions should finish execution is provided and written into a separate input file.
- The next phase in this approach is the thread scheduling algorithm.
- The text files populated by the previous phase are used by the thread scheduling algorithm.
- The information from these text files is assimilated and stored in appropriate data structures.
- Variables passed as arguments to functions and return value variables contribute as variable types whose dependency information needs to be collected and analyzed.
- Arguments to a function call: Variables passed as arguments to function calls from inside main are taken into consideration. The read/write dependency analysis on these arguments inside the function body is analyzed.
- If variables are passed by value, then the original variables in the main scope remain unchanged.
- If variables are passed by reference, then further checks are made to find out if variables passed as arguments are changed in any way inside the function body. If the variables are merely accessed, then it is known that they remain unchanged.
- Functions can return values of any type as well. These return values can be stored in a local variable inside the main scope.
- The next step is to find out the next point of usage of the above two types of variables. Since we perform inter-functional parallelism, care should be taken to ensure the variables have fully updated values, i.e., the functions should have completely finished execution. Thus, the function calls modifying any variables or returning any variables should be fully executed before the next point of usage of the aforementioned impacted variables.
- A point of usage of variables is referenced by the relative line number with respect to the beginning of the main body.
- Execution of functions in Method 4 is similar to how it works in Method-3. Functions are pushed into a thread pool, where it is executed on individual threads. In this method, we have eliminated the use of additional master threads to keep track of changing variables (referred to as special variables in Method 3). Instead, we use the additional information

gathered from extended data-dependency performed, giving us adequate information to execute functions before a specified point inside our program.

- In the previous method, we ensured the completion of all functions pushed into the thread pool by calling the "wait" or "get" method on all thread pool functions at the end of the program. However, in this present method, we preemptively call the "wait" or "get" method on the relevant function just before its designated point of execution (as determined and decided by the data dependency analysis). This way, all variables meant to be updated before their next point of usage are fully updated by a preemptive call to the corresponding function.
- To demonstrate, consider the following example:

Input Program:

```
transform_A(int *a, int n);
transform_B(int *b, int m);
transform_C(int *a, int n);
```

Output:

In the generated parallel program, following is the sequence of events:

```
// no special array required to keep track of modified arrays
// no ready and wait queues required to keep track of executing
functions
// definition of find_future function which ensures execution of the
mentioned function at that point of program
futures.emplace_back(transform_A, a, n)
futures.emplace_back(transform_B, b, m)
// transform_A needs to be complete before executing transform_C (which
uses array a)
find_future(transform_A, a, n)
futures.emplace_back(transform, a, n)
```

Analysis:

We see that pointer "a" is used again in transform_C after transform_A makes a modification on it. So when processing the data-dependencies, we keep a map, where we store that transform_A needs to complete before the execution of transform_C. This can be seen by the inclusion of the “find_future” function just before pushing the transform_C function into the thread pool.

By using this technique, we allow the threads to be able to decide the deadline for execution, hence eliminating the need for a master thread to monitor the threads and their execution.

- We use a map structure to store consecutive functions and their ordering local to the main body. This way, we have a reference to all the functions. This enables us to pick the right functions and preemptively finish its execution whenever needed.
- We handle code in the main function that are statements other than function calls. We use the data-dependency analysis to see what sections of these statements are dependent on any functions. If they are not dependent on any previous functions, then we align them for execution. If they do have a dependency, we add a barrier as deemed appropriate before this segment of code, to await completion of its dependencies.
- If a variable is being assigned the return value of a function, it is then dependent on that function. During our data-dependency analysis, we check for such instances, and in case of them we also deduce when the next usage of this variable happens at. We use this information, so that there is a barrier introduced before the next usage. This ensures that return values can be used in programming, while we continue to improve parallelization for function calls that return any type and not only void. The implementation of return type and return value handling is done using async programming concepts of promise and future.

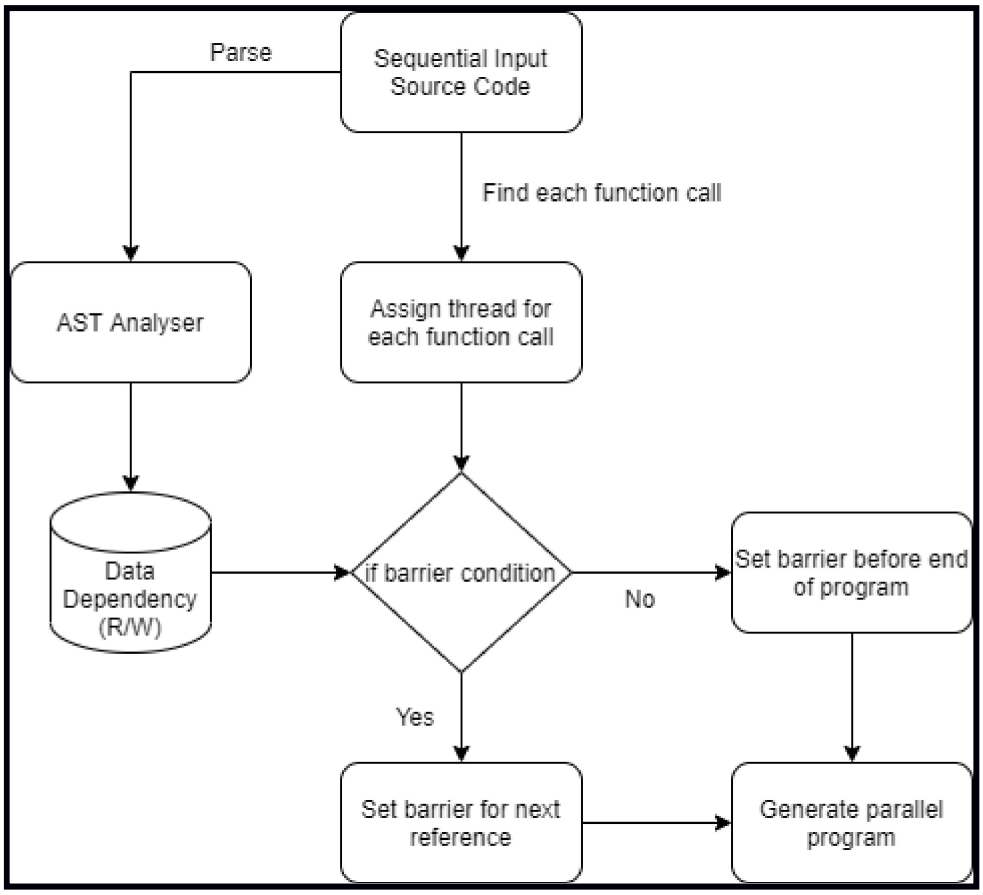

Parse
Sequential Input Source Code
Find each function call
AST Analyser
Assign thread for each function call
Data Dependency (R/W)
if barrier condition
No
Set barrier before end of program
Yes
Set barrier for next reference
Generate parallel program

Figure 21: Method 4 Implementation

- Consider the following example:

Input program:

```
int transform_A(int a)
{
    return a + 2;
}
void transform_B(int *a, int n)
{
    a[0] = 4;
    return;
}
int main(int argc, const char** argv)
{
    int res1;
    int *arr = {1,2,3,4,5,6};
    int b1 = 4;
    res1 = transform_A(b1);
    int res2 = 0;
    transform_B(arr, b1);
    for(int i = 0; i < res1; i++)
    {
      res2 += arr[i] + 2;
    }
    res1 = res2 + 2;
    return 0;
}
```

Output:

In the generated parallel program, following is the sequence of events:

```
// no special array required to keep track of modified arrays
// no ready and wait queues required to keep track of executing
functions
// definition of find_future function which ensures execution of the
mentioned function at that point of program
int main(int argc, const char** argv)
{
    int res1;
    int *arr = {1,2,3,4,5,6};
    int b1 = 4;
    futures.emplace_back(res1, transform_A, b1)
    int res2 = 0;
    futures.emplace_back(transform_B, arr, b1)
    for(int i = 0; i < res1; i++)
    {
     find_future(transform_B, arr, b1)
      res2 += arr[i] + 2;
    }
    res1 = res2 + 2;
    return 0;
}
```

Analysis:

- The function transform_A makes a write operation on the variable passed, but it is a pass by value, so this has no implications on the argument passed. transform_B has passed by reference for the array, and it makes a write on this array. So for every call of tranform_B, we will need to set a barrier for the next usage of array passed. So in the main function, we set a barrier before the next usage of arr inside the loop, which is shown by the call to the find_future function. Note that, find_future called from within the loop will execute only once at the beginning of the loop if it identifies any redundant operations. This is handled in the internal implementation for find_future. For handling loops we check if all barriers set for it are cleared, if they are, we assign thread and execute the loops. In the above example, we notice that res1 used res2, so we set a barrier for that assignment, to wait for the loop to complete execution.
- We handle selection statements, like if-else and loops, in a similar fashion. For selection statements that have function call(s) within them, we apply the same function parallelization technique described before. This takes care of the selection statements, and even ensures that there are no race conditions as it is handled when the functions are being parallelized. Every other statement inside selection statements are taken care of as stated previously.

6.1.4.2. Results

Our main aim has been to improve resource utilization of underlying hardware to the maximum potential when executing a program, and thereby reduce the execution times. We have measured this by comparing the execution time of the sequential program and its parallel equivalents, generated by both the aforementioned Method 3 and Method 4. Although execution time of a program can be used to indicate speed-up, as our goal is also to improve the utilisation of available resources, we can not directly use only comparison of execution time between the original sequential source code and the generated parallel source code. In order to better understand the impact of resource utilisation, we have chosen to run the generation of parallel code and its execution on different hardware with varying numbers of cores and threads. We have specifically chosen an Intel i5 processor with 2 cores and an

Intel i7 processor with 12 cores. These architectures are commonly available today to the average consumer and hence can be used as a reflection of the impact in the real world. This allows us to showcase that the parallel version of the program has increased the utilisation of the underlying available computational resource.

The Fig. 22 showcases the comparison of the sequential program with that of the parallel program generated by Method 4, Master Worker based approach, on two different architectures. The Y-axis is the ratio of execution time of sequential to parallel as we increase the number of computations carried out in a program. We have used ratios to represent the increase in this figure, as absolute comparisons are nearly impossible. The sequential program execution time increases non-linearly, while the increase in parallel time is seemingly more linear, Fig 24 shows for both methods. Fig 22 and Fig 23 can be compared to each other. They are the result of running on two different architectures. We notice that while in Fig 22 the speed up is 500 times almost, in Fig 23 it is 550 times. To generate Fig 23, we used an Intel i7 architecture with 12 cores as compared to the i5 architectures with 2 cores used to generate Fig 22. This can be used to infer that, with increased availability of hardware, we can improve the execution. This is possible as the underlying parallel program tries to maximise the resource utilisation, and on a machine with more cores and threads, it can achieve much higher parallelism.

The implementation of Method 4 is an evolution of some of the ideas established in Method 3 and additional design features. These changes streamline the resultant parallel code by reducing the number of mutex locks, increasing available threads as workers and removing all scheduling overheads. These changes along with the other described in previous sections justify the improvement in execution time. Fig 24 indicates the impact of these changes. Method 4 makes significant gains on Method 3 parallel code execution. As the number of computations are increased, Method 4 continues to perform much better as a result of more available threads and having no busy waiting for thread scheduling.

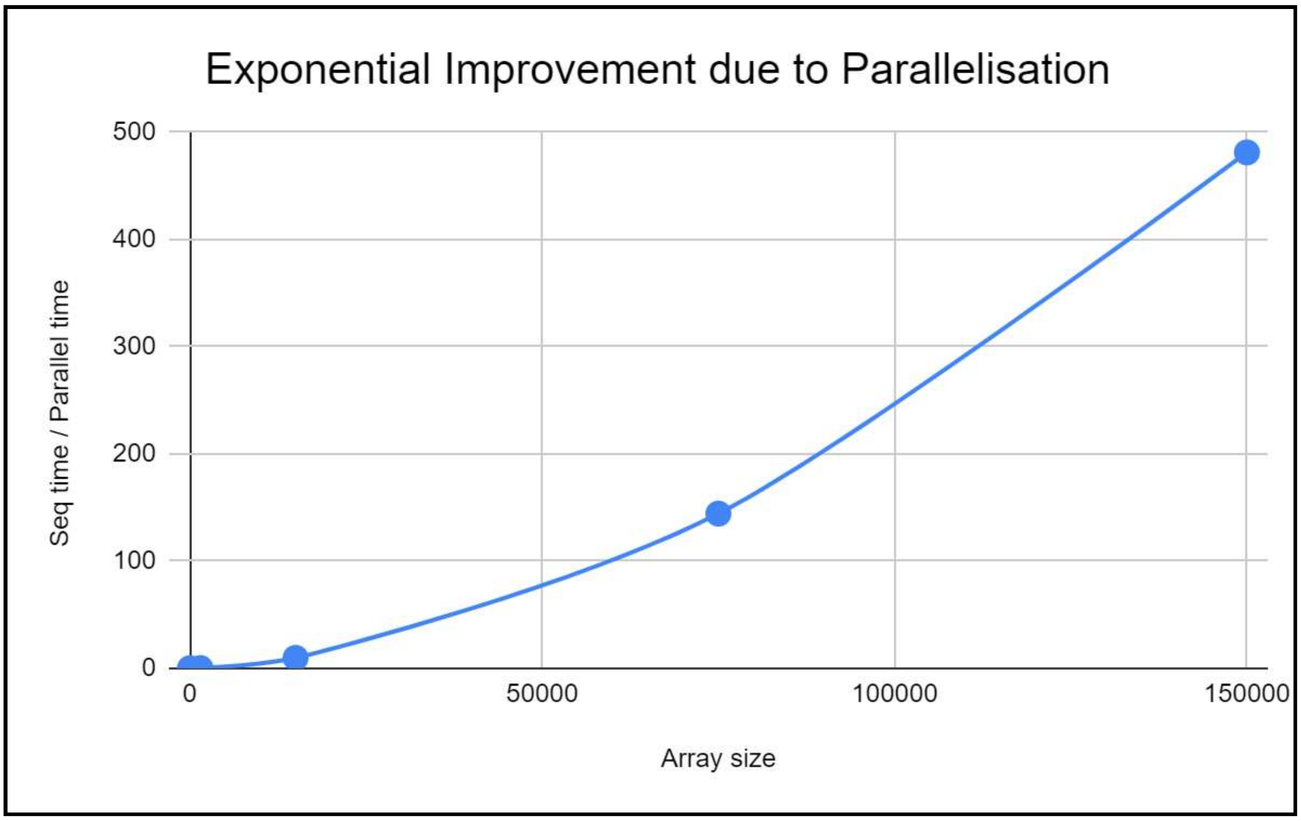


Figure 22: Ratio of Sequential execution times to Parallel execution times (Hardware setup : Core i5 - 2nd gen - 2 core machine)

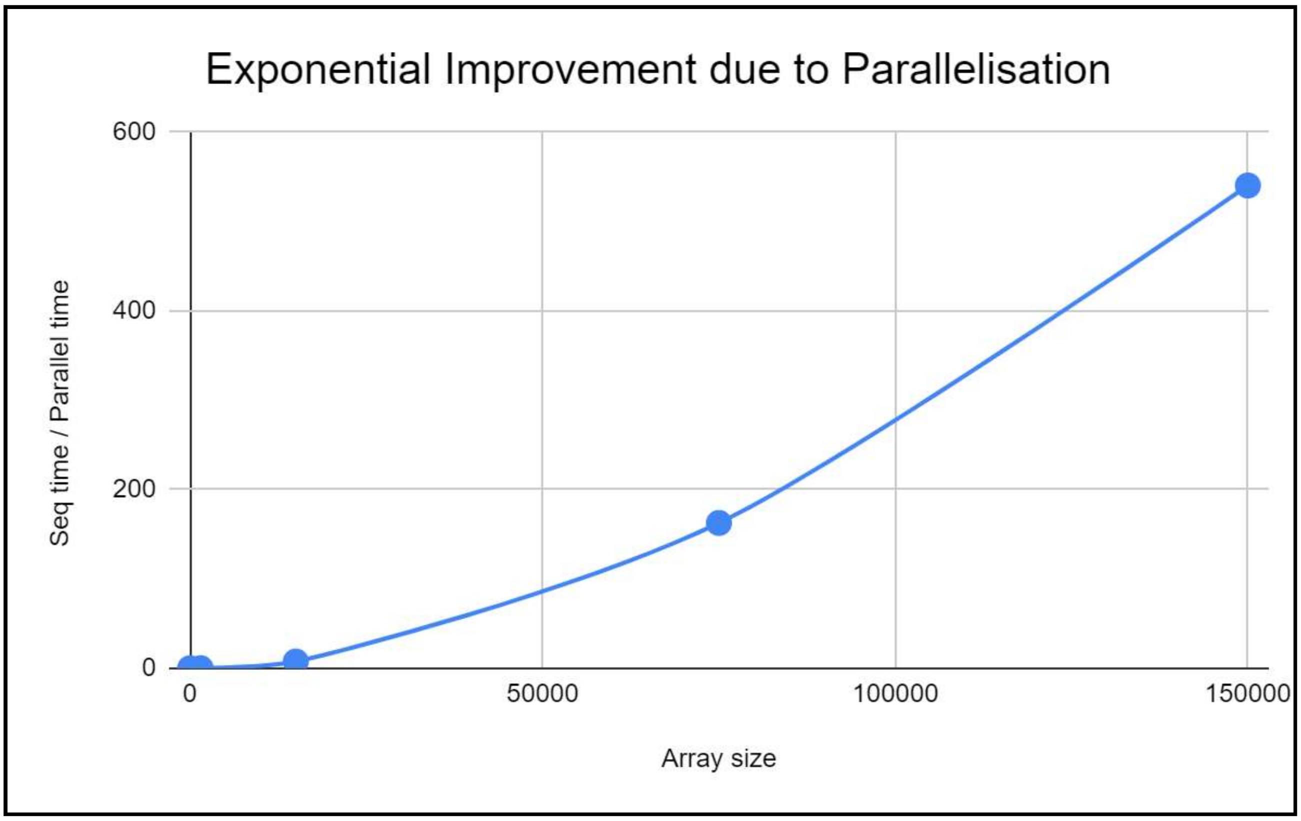


Figure 23: Ratio of Sequential execution times to Parallel execution times (Hardware setup : Core i7 - 9th gen - 6 core machine)

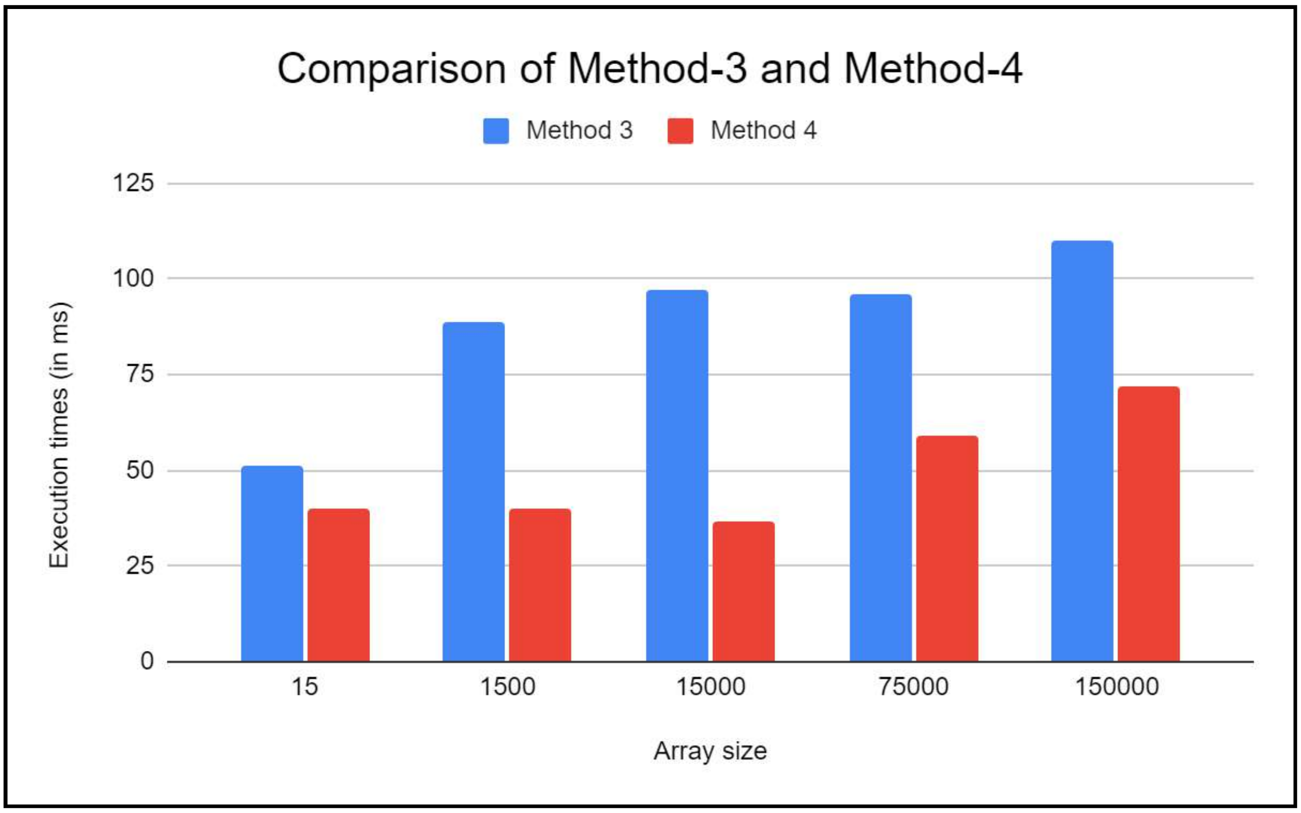


Figure 24: Comparison of execution times of Method-3 vs Method-4 (Hardware setup : Core i5 - 2nd gen - 2 core machine)

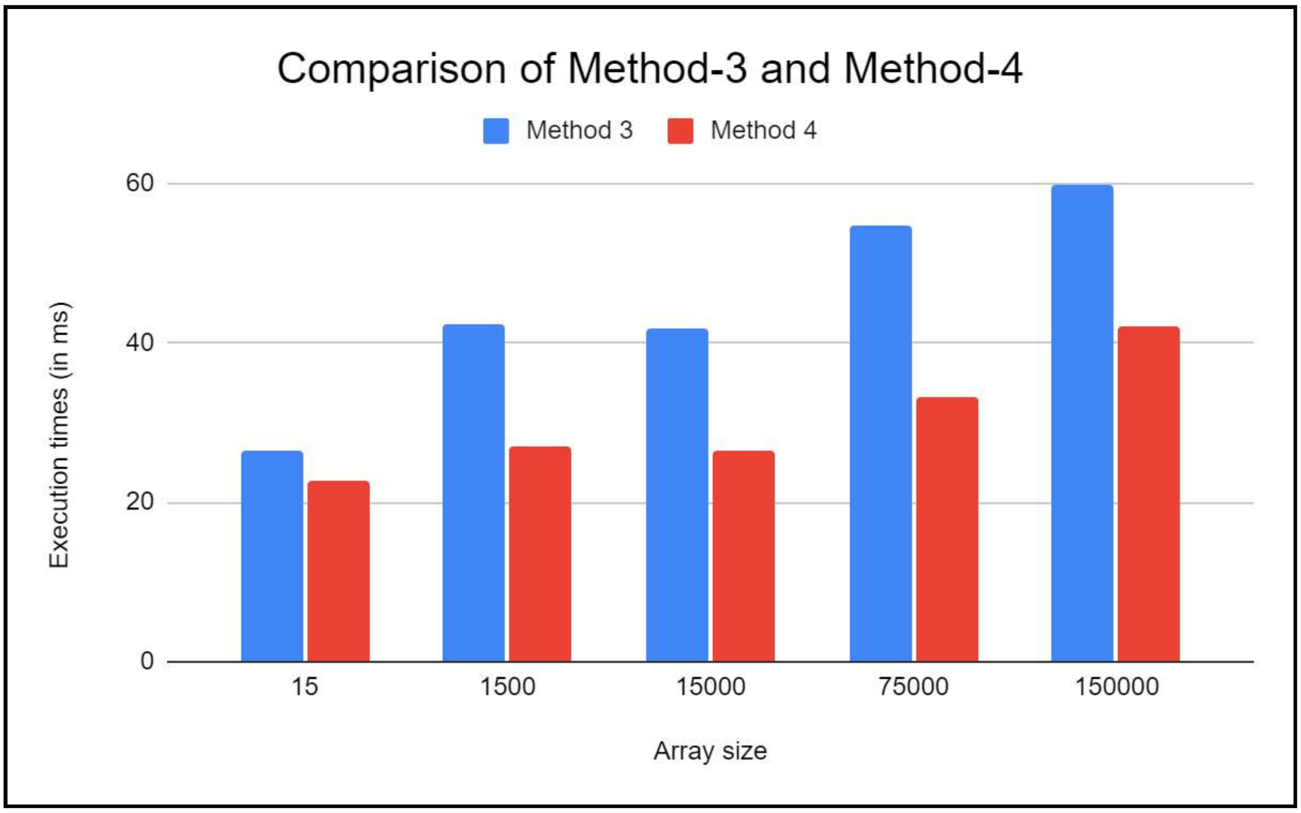


Figure 25: Comparison of execution times of Method-3 vs Method-4 (Hardware setup : Core i7 - 9th gen - 6 core machine)

6.1.4.3 Inferences of Method 4:

- Elimination of two additional master threads frees up available threads in the system, which can in turn be used for other tasks. This ensures more efficient usage of underlying hardware without any overheads.
- Simplified generated parallel code, with the elimination of if-else constructs, ready and wait queues, and a number of mutex locks required to keep certain data structures safe from race conditions.
- Due to the above two improvements mentioned, the generated program is now much shorter than the previous Method-3 approach.
- Thus, we have a significant improvement in the execution times of the newly generated parallel program when compared to that of the previous Method-3 approach.

6.2. Program Comprehension Phase

- The technique of program comprehension we employ enables us to identify the algorithm implemented in a specific function. If the program consists of multiple functions, then the requirement is to identify the algorithm tag for each of these functions.

  For example consider the following program:

```
void function_A(int* arr,const int arr_n)
{
    for(int i = 0; i < arr_n-1; ++i)
    {
        int min = i;
        for(int j = i+1;j < arr_n;++j)
        {
            if(arr[j] < arr[min])
            {
                min = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}


int function_B(int* a, int n)
{
    int max = a[0];
    for (int i = 1; i < n; ++i)
    {
        if(a[i]>max)
        {
            max = a[i];
        }
    }

    return max;
}
```

```
int function_C(int* a, int n)
{
    int min = a[0];
    for (int i = 1; i < n; ++i)
    {
        if(a[i]<min)
        {
            min = a[i];
        }
    }
    return min;

}
```

- The program comprehension stage of our pipeline will map each of the above functions to the corresponding algorithm as identified. In this case, function_A will be mapped to “sort”, function_B will be mapped to “max” and function_C will be mapped to “min”.
- The entire program comprehension pipeline is represented in the below flow chart

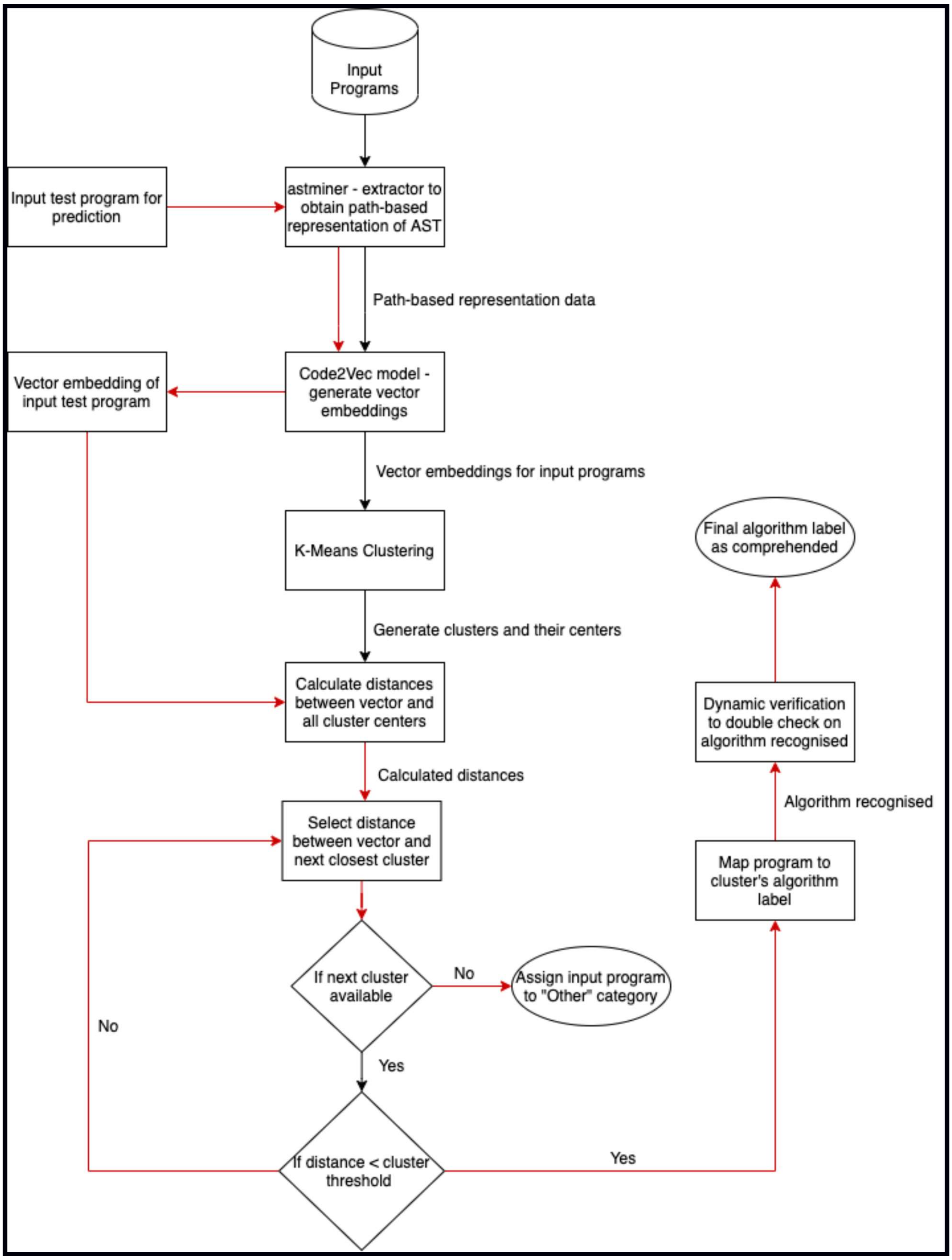

Input Programs
Input test program for prediction
astminer - extractor to obtain path-based representation of AST
Path-based representation data
Vector embedding of input test program
Code2Vec model - generate vector embeddings
Vector embeddings for input programs
K-Means Clustering
Final algorithm label as comprehended
Generate clusters and their centers
Calculate distances between vector and all cluster centers
Dynamic verification to double check on algorithm recognised
Calculated distances
Algorithm recognised
Select distance between vector and next closest cluster
Map program to cluster's algorithm label
If next cluster available
No
Assign input program to "Other" category
No
Yes
If distance < cluster threshold
Yes

Figure 26: Program Comprehension Implementation

- Our technique of program comprehension is to represent the input source code as vector embeddings, find similarities between these embeddings to group them into similar clusters, and additionally verify if the predicted label is accurate using a dynamic verification process.
- To represent the programs in the form of vectors, we employ a technique of embedding. It is a three-step process:
  - Use an extractor specifically designed for C/C++ programs to extract an Abstract Syntax Tree of the program and break it down into smaller sections based on nodes of the tree.
  - The nodes are then combined together to represent bigger sections, like a function definition or a selection construct body, to create so-called path-based context vectors.
  - These path based context vectors are then used to train a neural attention model to better represent individual vectors for each of the smaller sections, and then we concatenate these vectors based on weighted attention as computed by the model to finally form the single vector representation for the function or the program.
- The vectors thus obtained give a very good representation of each of the functions in the input program. Similar functions designed to implement the same algorithm would have very similar vector embeddings as well. To calculate the similarity between the vector embeddings, a simple technique such as cosine similarity can be employed.
- Based on these calculated similarity scores, it is now possible to group together similar functions. This technique is similar to the initial stages of the K-Means clustering algorithm. However, we do not use the concept of nearest cluster centroid to allot the cluster label to our functions.
- We have developed a means to calculate a threshold value for each of the clusters, which helps us in determining if a function belongs to the cluster or not. If the computed distance between the cluster centroid and the function is greater than the pre-computed threshold, then we skip the present cluster and perform the same comparison on the next closest cluster, since each cluster could essentially have different pre-computed threshold values. If a function doesn't belong to any of the clusters based on the said comparison and allotment technique, then the function is classified into an "Others" category.

- There is always a degree of uncertainty in the probabilistic design of the models we employ. The “Others” category is an important aspect of our Program Comprehension pipeline. It is important not to misclassify any of the functions, to ensure correctness and retain semantics of the original input source code. For instance, if we classify a function sorting an array, as a function reversing an array, this wrong label will affect the consecutive steps in our complete pipeline of parallelisation, and generate a semantically wrong parallel program in the end. However, it is still acceptable, if the same function sorting an array is classified into the “Others” category. In such a case, we will not be performing intra-function parallelisation on this particular function, however, it is safe from the wrong substitution of a parallel version with a different algorithm.
- Thus it is important to set strict threshold values of every cluster to ensure that we retain the semantics of the input sequential program.
- To further verify the allotted label through the clustering technique as mentioned, we make use of a dynamic verification process. Based on the identified algorithm label, we perform dynamic analysis on that function, verifying the output of that function for a predefined set of inputs.
- For example, if the clustering technique returns that the label for a function is sort, we can send in an unsorted array and verify if that function returns back a sorted array finally. This way, we double check on the identified label and further improve the accuracy of our complete program comprehension pipeline.
- Thus, through the use of multiple verification steps, we can be certain that this implementation returns the correct label for each of the functions in the input program, either the correctly identified specific label meant for an algorithm, or the “Others” label if there was any discrepancy in the prediction.

6.3. Additional Steps

- To ensure better parallelisation decisions and ensure correctness, we can make additional tweaks to our pipeline. Some of them are as follow:
    - If the execution times of the generated parallel program have increased, compared to the execution times of the input sequential program, due to context switching overheads, then we can skip parallelisation.

- If the outputs of the parallel and sequential programs are not identical, then this implies correctness of the original program has not been maintained. In such scenarios, we can skip parallelisation.

7. Conclusion

- Proposed Pipeline has been implemented in its entirety
- Assumptions made in Phase 1 have been eliminated

7.1. Parallelization:

- In the parallelizing phase, we have tried out 4 different approaches, namely:
    - Inter and Intra-Function Parallelism by AST Querying and replacement with OpenMP Directives
    - Naive Thread Scheduling using C++ concepts of Promises and Futures.
    - Optimised Thread Scheduling for Functions using Master-Worker based approach to achieve Functional Parallelism
    - Masterless Thread scheduling for functional parallelism
- With each approach we have tried to improve the parallel code being generated with respect to the different cases it can handle and with respect to the performance improvement that can be achieved.
- We have made improvements to Method 3 and arrived at a new approach, Method 4.
- Method 4 removes the Ready queue and Wait queue used in Method 3, thereby nullifying the requirement for the track and schedule Master threads.
- Method 4 handles a wider variety of cases , functions which return a value, selection and iterative statements.
- Method 4 simplifies the generated parallel code as compared to previous methods.
- With each approach, we have been moving down a level with respect to the level of abstraction that we are dealing with, thereby gaining a fine grained control over the different aspects concerning parallelization.

7.2. Program Comprehension:

- We have implemented the Program Comprehension Phase of our pipeline and integrated it with the Parallelization Phase.
- The input source code is represented as Vector Embeddings.
- The similarities between the vector embeddings of different programs is found to group them into clusters.

- Each of the clusters correspond to a previously defined category that has a parallel version in the backend database.
- It is tried to associate the new test program into one such cluster by using the appropriate thresholds.
- Dynamic Verification is used to additionally verify if the predicted label is accurate.
- "Others" category is introduced to ensure correctness of the program by avoiding any misclassification.

8. Future Work

- Refine and refactor the implemented code to gain possible improvements.
- Make the current implementations more efficient to reduce pre-processing time.
- Current implementation is a generalized technique, a possible area of future work would be to allow for domain specific optimisations as required.
- Explore possible improvements to increase the speedup in performance of the generated parallel code.
- Extend support to more cases wrt Program Comprehension by training on larger datasets and defining more categories with a parallel mapping.
- Experiment with any new improvements in Program Comprehension techniques.

Appendix A: Definitions, Acronyms and Abbreviations

Definitions

Parallelization - Parallelization is the act of designing a computer program or system to process data in parallel. Normally, computer programs compute data serially: they solve one problem, and then the next, then the next.

Program comprehension - Program comprehension is a domain of computer science concerned with the ways software engineers maintain existing source code. The cognitive and other processes involved are identified and studied. The results are used to develop tools and training.

Automatic Parallelization - Automatic parallelization refers to converting sequential code into multi-threaded and/or vectorized code in order to use multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine.

Intermediate representation - An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation.

Parallel paradigms - Defined categories to which different algorithms in the user's source code are matched to. Each paradigm represents a set of algorithms following a common notion of execution. A specific technique of parallelization is employed for each paradigm, providing more accurate results and achieving generalization.

Data Dependencies - A data dependency in computer science is a situation in which a program statement (instruction) refers to the data of a preceding statement. In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis.

Control Dependencies - Control dependency is a situation in which a program instruction executes if the previous instruction evaluates in a way that allows its execution.

MultiThreading - In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing.

Multi Threaded Processor - A multithreaded processor is a processor capable of running several software threads simultaneously.

Multicore Processors - A multi-core processor is a computer processor on a single integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run instructions on separate cores at the same time, increasing overall speed for programs that support multithreading or other parallel computing techniques. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP) or onto multiple dies in a single chip package. The microprocessors currently used in almost all personal computers are multi-core.

Multiprocessing - Multiprocessing is the use of two or more central processing units within a single computer system. The term also refers to the ability of a system to support more than one processor or the ability to allocate tasks between them.

Deadlocks - Deadlock is a situation where a set of processes are blocked off because each process is holding a resource and waiting for another resource acquired by some other process.

Race Conditions - A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Starvation - Starvation occurs when one or more threads in our program are blocked from gaining access to a resource and, as a result, cannot make progress.

Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer systems, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Overhead - In computer science, overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task. Overhead can be a deciding factor in software design, with regard to structure, error correction, and feature inclusion.

Speedup - In computer architecture, speedup is a number that measures the relative performance of two systems processing the same problem. More technically, it is the improvement in speed of execution of a task executed on two similar architectures with different resources.

Semantic Errors - In computer programming, a semantic error is a bug in a program that causes it to operate incorrectly, but not to terminate abnormally. A semantic error produces unintended or undesired output or other behaviour, although it may not immediately be recognized as such.

Run Time Errors - A runtime error is an application error that occurs during program execution. Runtime errors are usually a category of exception that encompasses a variety of more specific error types such as logic errors , IO errors , encoding errors , undefined object errors , division by zero errors , and many more.

Coherency Issues - In computer architecture, cache coherence is the uniformity of shared resource data that ends up being stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

Amdahl’s Law - In computer architecture, Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

Acronyms and Abbreviations

AST - AST stands for Abstract Syntax Tree

In computer science, an abstract syntax tree, or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

API - API stands for Application Program Interface

An application programming interface is a computing interface that defines interactions between multiple software or mixed hardware-software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc.

MPI - MPI stands for Message Passing Interface

Message Passing Interface is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures.

PVM - PVM stands for Parallel Virtual Machine

Parallel Virtual Machine is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor.

POSIX - POSIX stands for Portable Operating System Interface.

It is an IEEE standard designed to facilitate application portability. POSIX is an attempt by a consortium of vendors to create a single standard version of UNIX. If they are successful, it will make it easier to port applications between hardware platforms.

STL - STL stands for Standard Template Library

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.

GUI - GUI stands for Graphical User Interface

The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons and audio indicators such as primary notation, instead of text-based user interfaces, typed command labels or text navigation.

Report

by Karan Kumar G

Submission date: 07-Dec-2021 10:56AM (UTC+0530)
Submission ID: 1723098349
File name: Report_3.pdf (1.55M)
Word count: 20361
Character count: 116564

Abstract

With the diminishing rise of computational capacity of hardware resources available in today's world, it is imperative to make efficient use of available computational power to achieve better optimisations and faster execution times to improve cost-effectiveness. There is a need to utilise multiple resources simultaneously to achieve parallel execution. This requires software to be written by developers in such a way as to run parallely on multiple resources. However, designing code to achieve parallelism is not an easy task and requires skilled developers for the same. Automating the process of converting sequential to its parallel equivalent source code helps overcome the dependency on human expertise and saves valuable time. We propose such an automated solution targeting all types of generic sequential code, which converts it to its accurate parallel version, capable of utilising all of the hardware resources available in the underlying system. We do so by integrating multiple techniques to cover various cases of source code. We analyse sections of source code to understand the intent and correspondingly replace it with an optimized version. We also employ a scheduling algorithm to achieve fine grained control over execution of multiple sections of code, thereby maximising efficient usage and minimising execution times.

1. Introduction

In the earlier days, there was very little computational power that could be provided by the processors on a computer system. Software was written sequentially, which meant every instruction was executed one by one. This meant large softwares took a very long time to execute.

But there was an exponential boom in the development of computer hardware, with faster and more powerful processors coming into the market. The clock speed drastically increased which enabled more instructions to be executed in very short durations of time. However, this continuous increase in power and computational capacity was cut short a few years ago. This meant that ever increasing need for more and more computational power could no longer rely on developing hardware. We had to shift focus to the software side, to make intelligent and efficient use of available hardware.

Parallel computing is a paradigm which enables us to make efficient use of available hardware to run multiple instructions on multiple hardware resources, simultaneously. This could mean running separate tasks on multiple cores/threads available on a single system, or running separate tasks on multiple systems connected in a cluster. This enables us to execute our software faster and saves valuable time in several use cases, thereby improving cost-effectiveness. Thus, we propose a novel approach to solve the problem of automated parallelisation of sequential code, by using multiple techniques integrated into one package to convert any general sequential source code to its parallel equivalent source code without any human intervention. We achieve this primarily by using the concepts of Program comprehension and Task-level parallelism. Program Comprehension enables us to identify intent and algorithm implemented in a code section and consequently replace the same with its optimised parallel version as is defined in a backend database. Task-level parallelism enables us to execute different functions in a given source code in parallel by use of a data-dependency driven scheduling algorithm.

2. Problem Statement

Parallel computing is a programming paradigm that enables efficient use of available hardware to run multiple instructions on multiple hardware resources simultaneously, allowing for faster software execution. This saves valuable time in several use cases and reduces the cost considerably.

However, a major challenge with developing programs that have parallelism is that there is a need for highly skilled programmers. There is also the problem that parallelizing pre-existing source code would take a large amount of manpower and time. This may not be feasible or even affordable, depending on the size and nature of the project. Auto parallelization techniques would help in mitigating the cost, manpower and time required for such a project. There has been significant work done in this area, especially in recent times, that has shown promising results and new ideas with interesting perspectives on how to go about parallelising source code.

We propose a solution that enables automated conversion of sequential source code to parallel source code with the aid of program comprehension and our bespoke scheduling algorithm, capable of task-level parallelization.

Existing auto-parallelisation techniques are either constrained to specific domains alone or do not try to perform parallelisation by doing thread scheduling explicitly that would allow for fine-grained control over the execution. We intend to use the concept of program comprehension to assimilate the source code and use that information to parallelise based on programming paradigms. We also intend to use a novel thread scheduling idea that would allow us to control the execution of the client program such that we can improve the performance by using multiple threads while reducing any overheads caused due to parallelism.

The implementation can therefore be broken down into the following:

1. Conversion of given sequential source code to an Enriched Abstract Syntax Tree representation, which enables us to query the nodes to retrieve useful information about the input code.
2. Data dependency, control and data flow analysis performed with the aid of the generated AST to identify independent code sections.
3. Program comprehension to understand the programming paradigm or the intent of each program section or function using the concept of Clustering based on program vector embeddings.
4. Parallelisation of the eligible sections of source code, based on the identified programming algorithm and the best fit parallel version for it.
5. Using the data dependency graph, to execute multiple sections of the entire program in separate threads with the help of our scheduling algorithm, to maximise CPU utilisation and minimise execution times.

3. Literature Survey

3.1. Program Comprehension for parallelisation

14

3.1.1. Pasquale Cantiello and Beniamino Di Martino, Automatic Source Code Transformation for GPUs Based on Program Comprehension, 2012.

3.1.1.1. Introduction

The authors use a previously established technique of program comprehension called the PAP Recognizer along with some modifications. This is used to improve the performance of a program, specifically for running on GPU. The methodology used is based on previously built upon ideas regarding program comprehension, using a static analyser. There is an "Extractor", which works based on Prolog facts, which is used to recognise programming paradigms or patterns in the program. These patterns are then used to recognise the algorithm being used. Upon recognising the algorithm, the AST is then modified by a transformer module to transform the version which runs the same algorithm in a parallel manner.

3.1.1.2. Implementation

The implementation of both the static analyser and the transformer module is on an AST. This allows for the tool to abstract out the program a little more, hence makes it easier to process and modify. The static analyzer is first run on the AST, to identify algorithms described by different sub-trees in the AST. The algorithmic identification is done based on hierarchical parsing, which reduces the memory usage of the identifier. After identifying the sub-tree and the algorithm associated with it, the following steps are carried out:

- Removal of the subtree from the AST, and original code commented.
- Generation of a new tree with the modified code which may contain the necessary library calls and memory allocations needed for GPU processing.
- Addition of this newly generated tree at the same location as the removed subtree.

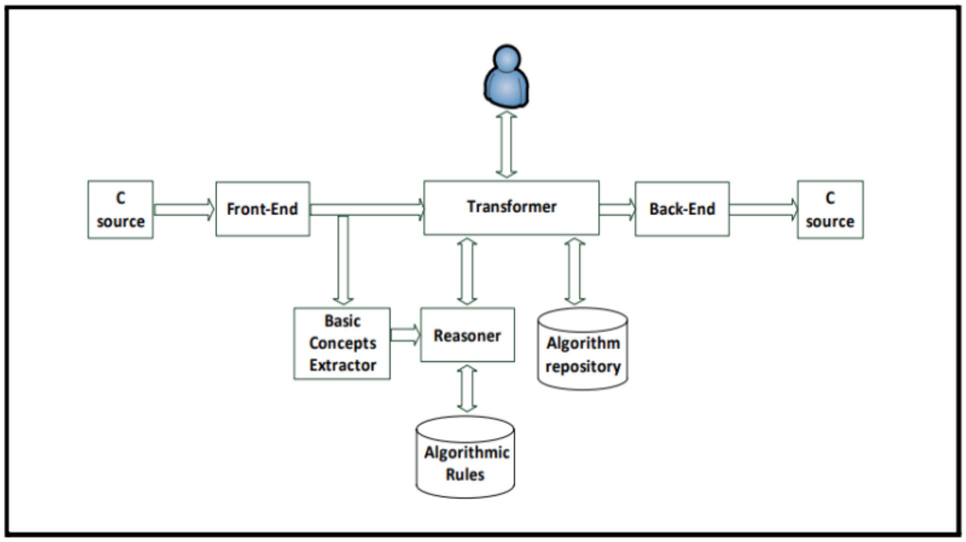


Figure 1: Working pipeline of Parallelisation tool

The transformer module makes the modifications to the AST based on "Algorithm repository" to make code better suited for running on GPU

The tool was implemented using SWI-Prolog for the analysis of AST based on rules, while the ROSE compiler was used to extract AST and to be able to query and modify the AST. The tool was capable of parsing C/C++ and FORTRAN 2003

3.1.1.3. Conclusion

The paper gives us insights into how program comprehension can be used to assimilate code, using a static analyser. It also provides for an interesting approach, by using AST to abstract out the program, allowing us to remove the personal choices of the programmers, and only deal with the necessary details of the program. This also lays the foundation for what can be done to perform parallelism within a small portion of the program, allowing for possible improvements in performance in a function or a small section of code.

Currently, the tool is only built to work on basic linear algebra algorithms, as the rules for algorithm recognition are written only for those. However, the recognition phase still takes a lot of time and hence has scalability issues. So a performance investigation on the transformer would possibly help improve that. The authors have ensured that there is an extension to OpenCL, allowing for usage on heterogeneous architectures, however, OpenCL is not used much in GPU programming anymore. They have also stated that they are working on increasing the algorithms that will be recognised.

32

3.1.2. Martino B. D. & Iannello G, Towards automated code parallelization through program comprehension, 1994.

3.1.2.1. Introduction

The authors talk about how existing tools simply try to identify loops that can be parallelized and loops that cannot be parallelized using data flow and dependency analysis. They discuss the "Concept Assigning Problem". In its general form, it is analyzing the programs to identify abstract concepts. However, the paper proposes that this problem can have complete automation if identification is applied on programming concepts and algorithms, such as searches, sorts, structure modification, numerical, etc. (which are essentially algorithmic concepts)

3.1.2.2. Implementation

Defines the parallel structure of a program as Programming Paradigms (PP). There are two PPs proposed:

- **Tree computation:** a set of processes that communicate with one another based on a binary tree structure. Every problem that needs to be solved is divided into sub-problems, which are treated as child processes. These child processes are executed and their results are moved upward to their parents
- **Master-Worker:** consists of a master process and a collection of worker processes. The worker processes carry out processes as defined by the master process.

Problems irrespective of their specific working and applications can be grouped under the above two mentioned paradigms. For eg: the "Tree computation" paradigm can be assigned to an iterative quicksort algorithm whereas the "Processor farm" paradigm can be assigned to a branch-and-bound binary search problem. To recognize the paradigm itself, the authors make use of a finite set of pattern templates (called cliches). PS defines the Parallel Skeleton code for the selected paradigm.

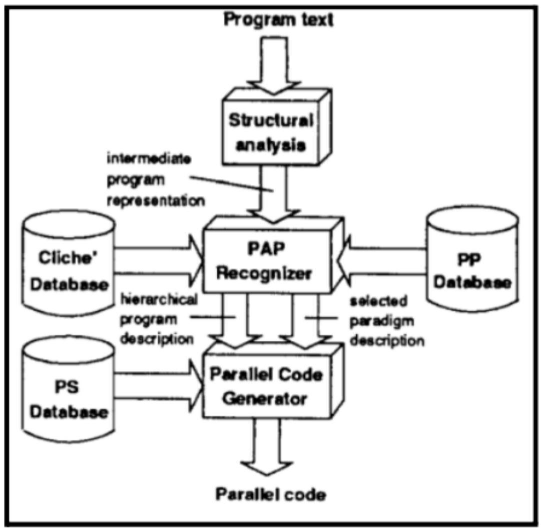


Figure 2: Working pipeline of PAP Recogniser

3.1.2.3. Conclusion

- This is only theoretical, a tool hasn't been devised.
- The approach is very ambitious and might not work on all generic programs
- Upon further research, the concept of cliches and paradigms as defined in the paper can be looked into, possibly expanding the databases to further improve the generality of the tool
- This paper was before OpenMP came into existence. With the use of OpenMP, it is possible to have better results using this same concept.

3.1.3. Di Martino B & Iannello G, PAP Recognizer: a tool for automatic recognition of parallelizable patterns, 1996.

3.1.3.1. Introduction

The authors present a new method of analysing programs to identify certain regions, which they call Parallelizable Algorithmic Patterns (PAP). They do so by using program recognition based processes. The output of the tool is a web application based representation showing the hierarchical description of the recognized patterns. The prototype tool has been implemented in the Vienna Fortran Compilation System.

3.1.3.2. Implementation

The Automatic parallelization problem is categorised into the following requirements:

- Identification of a parallel model
- Data distribution for the defined processes
- Selection of work categorisation, enabling code decomposition

The tool mentioned is mainly driven by paradigm recognition rules working on representation of concepts. These rules enable differentiation of multiple concepts based on intent and meaning. These rules can be represented as abstract functions contributing to building an abstract structure. Control and data dependency analysis are implemented as an abstract process.

The tool analyses the program by making use of control flow and data flow to develop some kind of abstract flow structure. All this information about underlying concepts are stored as dependency graphs in a datastore. This datastore is updated as the identifier parses the entire program.

3.1.3.3. Conclusion

The PAP Recognizer introduced in this paper is capable of recognizing some features related to numerical computation only. It would be challenging to introduce new concepts that are compatible with the design of this tool. Hence generalization would be an uphill task.

3.1.4. 6 Di Martino B & Kessler C.W, Two program comprehension tools for automatic parallelization, 2000.

3.1.4.1. Introduction

- Program comprehension is identification of abstract concepts in a program
- This could mean finding sequences of code implementing some algorithmic concept
- This is challenging because of syntactic variation, algorithmic variation, delocalization, and overlapping implementations.
- Why do program comprehension for automatic parallelization?
    - It provides for an aggressive code transformation, without being dependent on the sequential structure in the program

  - The acquired knowledge allows for automatically selecting segments for optimization, and helps improve the working of performance prediction
  - The application domain considered mainly consists of numerical computations, linear algebra and partial differential equation codes (hence it is not exactly generalized)
- Non numerical examples - quicksort and branch-and-bound

3.1.4.2. Implementation

- PAP was implemented in the Vienna Fortran Compiler (VFC)
- PARAMAT tool provides a faster solution, whereas PAP Recognizer provides for a greater generalization with higher run times
- Recognition of subconcepts for a given concept to identify the hierarchical concepts is done.
- Rules to infer a concept are defined in terms of intermediate representation features like operator symbols, equality of program objects, control and data flow information and already computed subconcept information.
- The IR can be an annotated abstract syntax tree for which customized tree-pattern matching techniques will be used
- It uses Concept of Interactive Parallelization - helping users with useful information during their efforts to parallelize a program
- Another goal is to replace program parts with calls to already implemented parallel code

3.1.4.3. Conclusion

Applications are plenty with this approach of Program comprehension to parallelization. Some of these applications are:

- Automatic Code restructuring and Library use
- Template based code transformations
- Automatic data distribution
- Automatic performance prediction
- Speculative program restructuring
- Knowledge-based support to interactive parallelization (take live inputs from users during the parallelization process)

The authors say that they have indeed tried to integrate the two tools together to extract the best out of the two. Their conclusion is as follows:

If recognition time is of utmost importance, PARAMAT can be used, otherwise use the PAP tool since it offers greater flexibility.

3.2. Parallelisation techniques

3.2.1. Peter Kraft, Amos Waterland, Daniel Y Fu Anitha Gollamudi, Shai Szulanski, Margo Seltzer, Automatic Parallelisation of Sequential programs, 2018

3.2.1.1. Introduction

The paper deals with the implementation of a previously hypothesised idea known as ASC (Automatic Scalable Computation) architecture. In ASC, the memory state and registers are used to create a vector. This vector is then used to predict the possible trajectory of the program execution. This prediction is then used to continue execution. By having simultaneous executions, when the program comes to the point at which the prediction was made, the worker thread that was tasked to continue the execution of the correct prediction is then used. This method uses processor farms to improve performance.

3.2.1.2. Implementation

The vector for characterising the program state is made using the memory state of the program and registers involved, by the usage of ptrace and perf_events. This vector is then used as input to either a neural network or a decision tree. The prediction made by the neural net or decision tree is stored in a look-up table. Also the predicted execution is passed to a worker thread to continue execution from the predicted point onwards. Once the execution of the main program reaches the point of the prediction, we do a look-up on the predicted state and match with the actual state to see which worker had the task of executing the correct prediction. The result of the worker thread is then used, and the main program execution skips to the point at which worker thread is executing. The paper uses an Intel tool called "PIN", which on the basis of dynamic analysis identifies points at which predictions can be made.

3.2.1.3. Conclusion

The paper extensively uses PIN, however it is a very expensive tool, so improvements to PIN would improve the performance of the tool. The paper also discusses the potential use of other dynamic instrumentation tools like JIT with PIN and valgrind. There has been some related work, in the area of binary parallelisation (parallelising sequential binaries to parallel equivalent) and compiler parallelisation, including the usage of OpenMP and other specific compilers. The tool also fails with things such as accumulators and similar concepts.

3.2.2. Cristian Ramon-Cortes, Ramon Amela, Jorge Ejarque, Philippe Clauss, Rosa M. Badia, AutoParallel: A Python module for automatic parallelization and distributed execution of affine loop nests, 2018.

3.2.2.1. Introduction

The authors propose a python module for automated task-based parallelisation. This is implemented on affine loop nests to execute them in parallel in a distributed computing environment. This also involves construction of data blocks to identify and use task granularity for achieving better performance in terms of execution.

3.2.2.2. Implementation

- Uses existing tools to devise a pipeline to parallelize Python code on distributed systems
- Pluto is a tool which helps in parallelizing affine loops
- COMPs is a tool which helps in converting source code to run on a distributed cluster (similar to functionalities of MPI)
- A combination of these two is proposed (similar to a combination of OpenMP and MPI - however, this is manual)

3.2.2.3. Conclusion

- Difference between affine and non-affine programs:
- Affine loops are the loops in which the referenced array subscripts and loop bounds are a linear function of the loop index variables. This implies that the memory access sequence is already noted in the compile time itself.

  Example:

```
      for (int variable = 1; variable < 12424; ++variable)
              my_array[variable] = my_array[variable-1] + 9586
```

- In the case of non-affine loops, the memory access sequence cannot be pre-identified during the compile time itself.

  Example:

  ```
  for (iter=1;iter<50;++iter)
          my_z[my_e[iter]] = my_z[my_t[i]] + my_z[my_p[i]]
  ```

- Most of the research has been done with respect to affine transformations as the availability of memory access patterns at compile time helps in the development of techniques and methods that can be used to identify the parallelizable segments easily.
- Performing parallelization for non-affine program segments can be something that can be looked at as a possible area of research.

3.3. Automated tools for parallelisation

3.3.1. Pluto: Uday Bondhugula, J. Ramanujam, P. Sadayappan, PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System, 2007.

3.3.1.1. Introduction

The Pluto tool is an auto-parallelisation tool that uses the ideas of polyhedral modelling and analysis to parallelise programs using OpenMP pragma directives, along with other techniques such as loop optimizations, tiling, loop merging etc. It was developed as a PhD thesis by Uday Bhondugula, currently a professor at IISC Bangalore. It has been a benchmark tool since its inception in 2007. The ideas of polyhedral analysis are deep rooted in Integer Linear Programming, so as to be able to optimise the execution of programs.

3.3.1.2. Implementation

The tool uses the ideas of polyhedral modelling. In polyhedral modelling, each loop is considered a lattice point and a program forms a polyhedral. This polyhedral is then made to undergo an affine transformation by the usage of Integer Linear Programming. By performing these transformations, there are checks to make sure that the correctness of the program is maintained while improving the performance of the loops being transformed. The transformed polyhedral is then converted back into a program and made available to users. For affine loops, the compiler applies transformations based on dependencies. For non-affine loops, compilers may perform various other transformation techniques, such as tiling or unrolling of loops etc. Performing the affine and non-affine transformation converts the polyhedra into different optimised polyhedra.

3.3.1.3. Conclusion

The drawback of polyhedral analysis is the expensive nature of Integer Linear programming. This is hence translated into the usage of Pluto itself. The tool is also very cumbersome to install. However, with respect to loop parallelisation, the results of the tool were accurate. Following were the restrictions we found out about the tool:

- Placement of pragma scop and pragma endscop is user-defined. Hence this step is not fully automated
- If this section defined under pragma contains code that can all be parallelized, then it ends up parallelizing them. However, if there is even one section of code within this pragma section that cannot be parallelized, then even those code segments that can be parallelized do not get parallelized (can still perform loop jamming but no parallelization)
- Can only place one such pragma section in the entire program (otherwise issue of redeclaration of helper variables that is automatically used by the tool internally)
- Placement of pragma requires user intervention and knowledge about parallelism defeating the purpose of auto parallelization
- Not able to handle cases of reduction (divide and conquer cases) For example, accumulator to sum up elements of an array

3.3.2. ParaWise – Widening Accessibility to Efficient and Scalable Parallel Code (White Paper), 2004.

3.3.2.1. Introduction

It is a commercial paid tool, which has been in constant development for the past 20 years. It provides a lot of features to customize the type of parallelization, number of threads, etc., providing additional flexibility to the users.

3.3.2.2. Implementation

The white paper talks about current problems of High Performance Computing (HPC). They analyse the requirements of users in the domain of HPC and accordingly design their product. It also talks about the final end user and major market for HPC and categorizes them into expert, non-expert and serial code users.

ParaWise makes use of efficient code analysis, enabling them to use OpenMP directives to be inserted into appropriate positions, thus introducing parallelization into a serial source code. They also experiment with Message Passing optimizations to further improve the benefits of parallelization.

3.3.2.3. Conclusion

The tool is quite exhaustive providing state of the art features to enable parallelization. The tool being interactive, requires users to provide valuable inputs during the process of parallelization. Thus, it is not completely automated. It also doesn't cover all possibilities which can be exploited to introduce parallelization.

3.3.3. Idan Mosseri, Lee-or Alon, Re’em Harel, and Gal Oren, ComPar: Optimized Multi-Compiler for Automatic OpenMP S2S Parallelization, 2020.

3.3.3.1. Introduction

- Talks about other tools: AutoPar, Par4All, and Cetus.
- ComPar is an innovative approach to parallelization and uses a source to source multi compiler. It utilizes code segmentation along with fusion, with the use of hyper parameters for tuning.
- Performance is improved with no manual intervention by tuning the hyper parameters, so as to obtain the best possible parallelised program. This is done while ensuring the validity of the input source code.
- The results obtained for analysis are on the NAS and PolyBench benchmarks.

3.3.3.2. Implementation

- The S2S compiler working is as follows:
  - The source code is parsed into an Abstract Syntax Tree (AST)
  - The AST is analysed, so as to obtain data dependencies; which is then used to find code segments to parallelise and the required directives for parallelization are inserted. This is done to optimise the code. This process is repeated until convergence is reached.
  - After convergence, the AST is then reverted back to the source code language as necessitated.
- As of now, no automatic parallelization tool or compiler is capable of replacing programmer insights. Human programmers still outperform compilers in this regard. This is due to information gathering required for parallelization from an AST is difficult in generalised scenarios. This is a major disadvantage for automatic parallelising tools.

- An example for the above being, function side effects. Also, relevant information that plays a major role in parallelization such as the load of computation, scheduling optimization, and available threads etc.
- The tuning of OpenMP directives to optimise performance automatically is well established. We might not be able to work here.
- Parallelising directives of OpenMP target sections of the code separately, unlike MPI. The sections may have different working fashions hence no unified compilation of an entire source code with one unique Source to Source compiler can yield optimised result. Thus, by using code segmentation, and usage of different Source to Source compilers will result in better parallelization of the entire input source code.

3.3.3.3. Conclusion

- Despite the fact that resources required for ComPar are greater, it is compensated by better results compared to other S2S compilers, especially ones that need tuning of hyperparameters by an external user.
- ComPar can be accessed and viewed: https://github.com/Scientific-Computing-Lab-NRCN/compar
- After testing several S2S compilers, the conclusion that can be drawn is that despite individual advantages and drawbacks, none of them are superior in results compared to the rest in all scenarios.
- The combination of all the above compilers yields the best possible optimised result, under hyperparameters tuned for the specific input. This combination of compilers is expensive computationally due to the sweep of the entire space of possibilities of the hyperparameters, for each possibility the performance is estimated so as to make the best possible optimisation.

3.3.4. Hamid Arabnejad, João Bispo, Jorge G. Barbosa, João M.P. Cardoso, AutoPar-Clava: An Automatic Parallelization source-to-source tool for C code applications, 2018.

- Performs parallelization of loops only
- Performs Static analysis without any runtime info and any additional info from the user.
- Checks for dependencies within the candidate loops
- Checks the dependencies for static variables - Performs liveness analysis and determines how the variables are referenced. Finds the Read, Write pattern that exists and determines if any dependency exists
- Checks dependencies for arrays - Using array subscripts to determine if loop iterations are independent which is done using existing methods such as GCD and Extended GCD.
- If no dependency exists, the OpenMP directive that it matches is found. Also, it categorizes the variables according to the OpenMP classes.
- Generates code annotated with OpenMP directives.

3.3.5. A Review of Parallelization Tools and Introduction to Easypar

3.3.5.1. Introduction:

The paper is a survey of existing tools for auto-parallelization. They have classified these tools into various categories based on the degree of automation and on the parallel programming language. The classification is also on the following criteria:

3.3.5.1.1. Based on Parallelization stage:

"Parallelization process is a systematic process, especially automatic parallelization. First stage of the parallelization process is parallelization identification. The code is parsed and analyzed (static or dynamic dependency analysis) to search for the code sections that can be executed concurrently."

3.3.5.1.2. Based on era:

The tools are classified as either First Generation Tools(FTG) or Second Generation Tools (STG) based on when they were developed

3.3.5.1.3. Based on Graphical Assistance Tools:

This classification is based on if the tool provides a visualisation of the parallelization phase or not.This information could reduce burden on programmers in understanding the generated program.

3.3.5.2. Tools Described:

3.3.5.2.1. First Generation Tools:

- Automatic and Interactive parallelization
- Object Based parallel Programming assist
- Animation Choreographer
- DEEP Development Environment
- PTP-PLDT by IBM
- GRED
- VISO
- The SUIF compiler

3.3.5.2.2. Second Generation Tools:

- Alchemist
- DProf
- Prospector
- Coarse Grain Parallelization
- LoopSampler
- iPAT/OMP
- Capo
- Kremlin
- Kismet
- Cilk++
- Holistic approach for auto-parallelization
- Polaris
- SD3
- Prism
- AutoFuture
- Vector Fabrics

- Pluto
- Par4All
- Cetus
- S2P
- EasyPar

3.3.5.3. Conclusion:

The paper provides 6 a brief overview of the work done in the past in the area of auto-parallelization. By providing robust classifications, it renders future literature surveys simpler. The paper also presents EasyPar as a parallelization tool that can be used to not only generate parallel code but also to generate code to run on GPU's. The extensive literature survey done by the authors eases the work for future work in the area, until such a time there has been significant changes to the domain.

3.4. AST Generation and Querying

3.4.1. Rose

Rose is an open-source compiler-based tool that is compatible with multiple programming languages such as C, C++, OpenMP, etc.

Rose tool is used for generating data dependencies, along with intermediate representations such as Abstract Syntax Trees. It provides for compilation and decompilation processes between high-level code and intermediary code. This tool has been used in the paper mentioned in section 3.1.1.

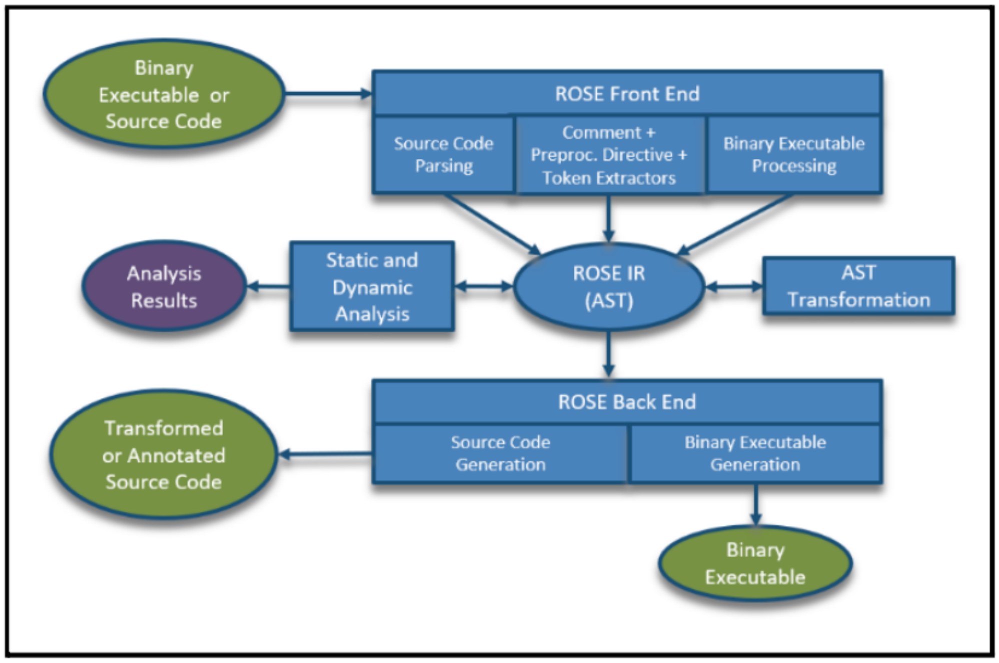

Binary Executable or Source Code
ROSE Front End
Source Code Parsing
Comment + Preproc. Directive + Token Extractors
Binary Executable Processing
Analysis Results
Static and Dynamic Analysis
ROSE IR (AST)
AST Transformation
ROSE Back End
Transformed or Annotated Source Code
Source Code Generation
Binary Executable Generation
Binary Executable

Figure 3: Working of ROSE Compiler

3.4.2. Clava/Lara - João Bispo, João M.P. Cardoso, Clava: C/C++ Source-to-Source compilation using LARA, 2020.

3.4.2.1. Introduction

Clava is a comprehensive compiler-based tool, which aims to extend the functionalities provided by Clang. It is based on a Javascript-based language called Lara. It works on an enriched Abstract Syntax Tree, with multiple attributes associated with each of the nodes. It provides useful information about data and control flow mechanisms involved in a program. It works for the programming languages of C and C++. In addition to supporting compilation and execution on Linux based systems, it also provides an online IDE.

3.4.2.2. Implementation

Following is the working of Clava:

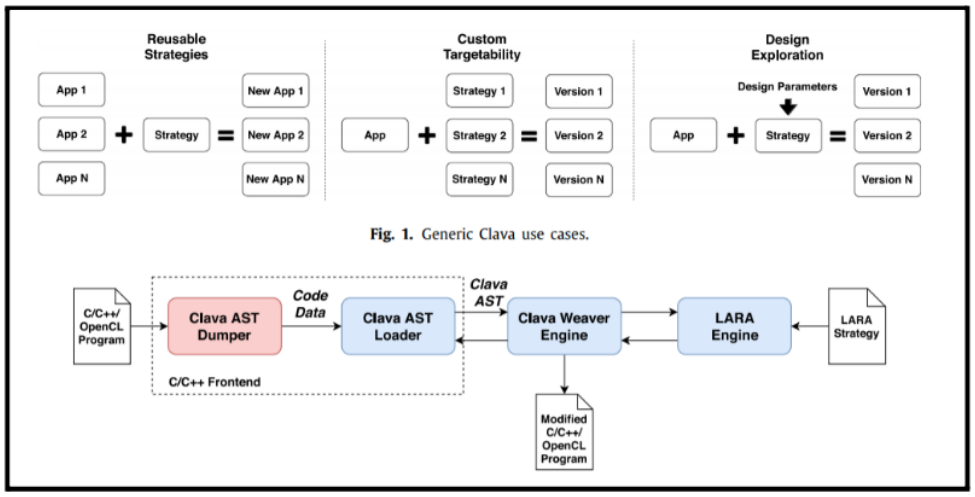


Figure 4: Working of CLAVA

3.4.2.3. Conclusion

The Clava tool is a very useful tool to identify data and control flow information from the input program. It has elaborate attribute information for each node in the Abstract Syntax Tree, which enables multiple functionalities. It also provides us with a means of modifying the program to enable functional parallelism (both intra and inter-functional).

3.5. Functional parallelism

3.5.1. Sean Rul, Hans Vandierendonck, Koen De Bosschere, Function Level Parallelism Driven by Data Dependencies, 2007.

3.5.1.1. Introduction

This paper proposes a method for acquiring and assimilating potential parallelism in programs. The proposed method is said to reveal the amount of parallelism present in the sequential programs and also suggest an appropriate parallel construct for the program such as a pipeline architecture, master-slave design and so on.

The method is not an exact analysis but rather a profile-based approach and hence, being dependent on the input, is not safe in terms of accuracy and correctness. It focuses more on measuring memory dependencies at a functional level and constructs two graph representations of the profile data: the interprocedural data flow graph - showing the data flow between functions and the data sharing graph - denoting the data structures used to share data. The visualisation of these graphs helps in finding the sections of data that could be modified and parallelised and revealing the data structures that may need synchronization for ensuring thread safety.

3.5.1.2. Implementation

- To construct a call graph, they record and form a caller/callee relations among functions, keeping a track of the number of times a function is executed, different callees of a function and the execution time it consumes

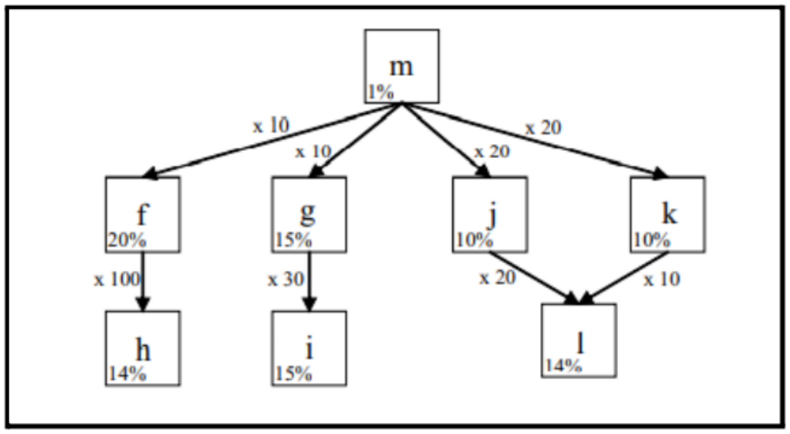
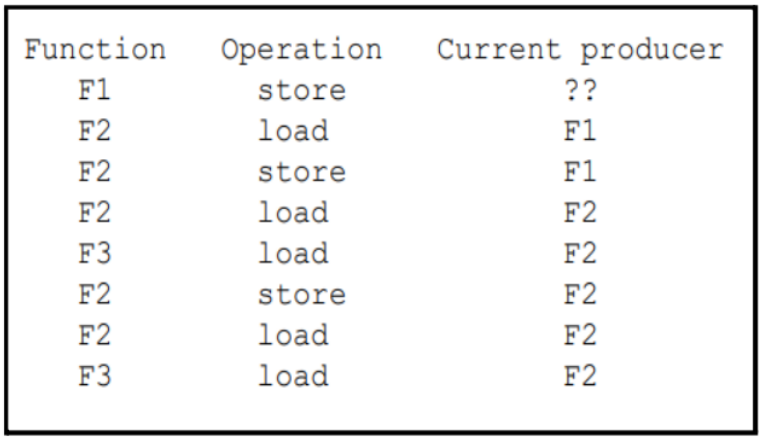


Figure 5: A Call Graph from data-dependency analysis

- Data Dependencies: Identifies and records in a matrix, inter function dependencies of data.



Function	Operation	Current producer
F1	store	??
F2	load	F1
F2	store	F1
F2	load	F2
F3	load	F2
F2	store	F2
F2	load	F2
F3	load	F2

Figure 6: Memory Dependencies

- The above figure shows a trace of memory operations for a specific variable.
- A matrix is built for each variable modified.
- Using these matrices, the functions are clustered based on the strength of connection among them and inter and intra cluster data streams are represented as a directed graph.

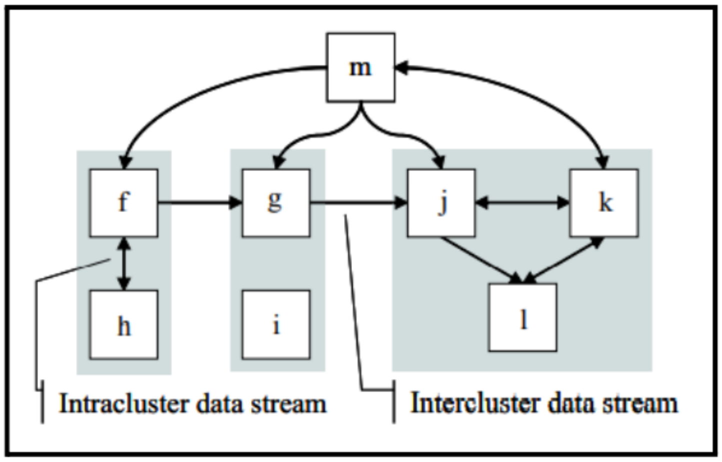
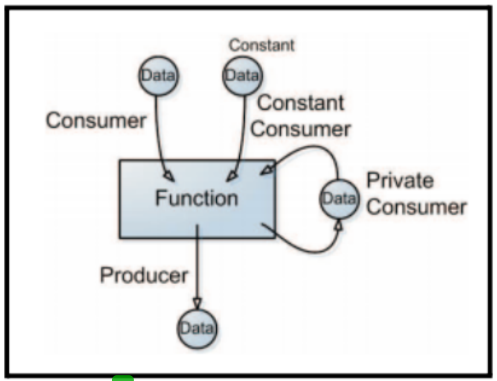


Figure 7: Interfunctional data flow graph

- To show how data sharing is done, a data-sharing graph is constructed consisting of function nodes and data nodes and the relation among them.
- A consumer is a function that reads data modified by a different function. The function writing such data will be the producer. Private consumption is the modification of local data. Constant consumption is the untraceable modification of data.



5

Figure 8: Classification of data dependencies

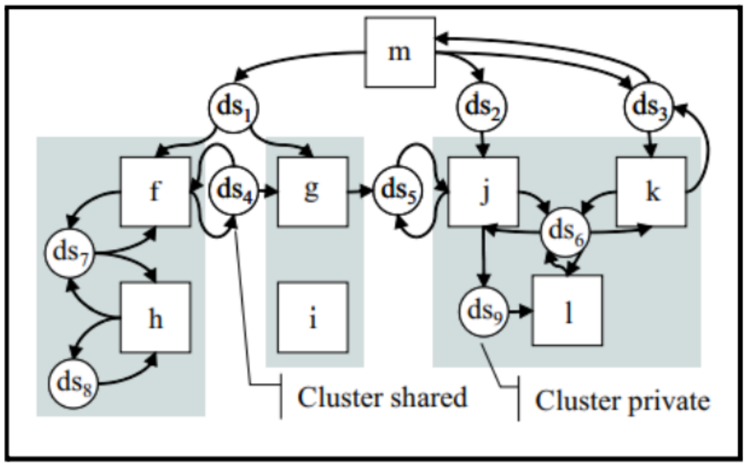


Figure 9: Data sharing graph

- The paradigms discussed:
    - **Master-Slave paradigm:** a Master thread creates several slave threads and assigns a part of the work to each. Synchronization is maintained by the use of barriers.
    - **Workpile paradigm:** Each thread requests a part of the work from a queue called the workpile. Threads can also push work to the pile.
    - **Pipeline paradigm:** Follows a simple producer-consumer relation, each stage in the pipeline produces data for the next stage of the pipeline

3.5.1.3. Conclusion

For generating results, an attempt was made to parallelise a compression procedure in bzip2. Based on the graphical analysis discussed, 4 functional clusters were discovered and the pipeline paradigm was implemented. A similar analysis was made for the decompression process. The results:

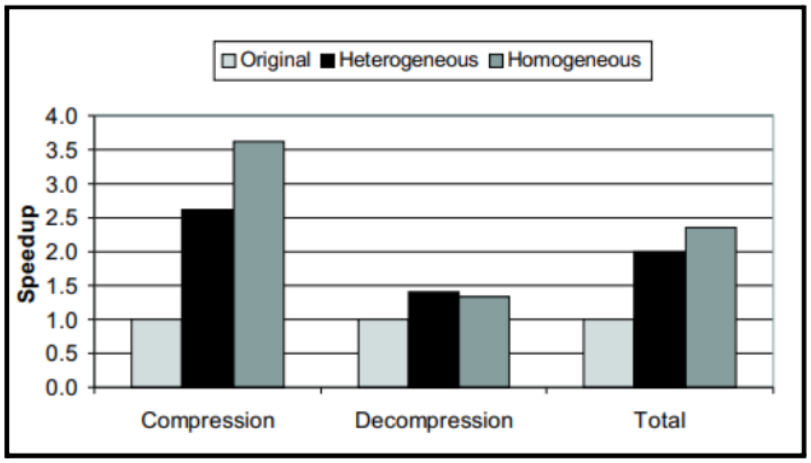


Figure 10: Speedup results of parallelized bzip2

A parallel programmer can use this as a tool to detect and automate, to some extent, the parallelisation of code, but he will still need to validate its correctness.

3.6. Parallel programming libraries

3.6.1. OpenMP

- OpenMP provides support for parallel programming in C, C++, Fortran in shared memory environments.
- The programmer has to explicitly mention the regions that can be parallelized. This is done using the OpenMP directives which are used to annotate the sequential code. We can define the number of threads to be used as well. OpenMP initiates those many threads and runs the parallel region across these threads.
- Usage: Works using pragma directives
  #pragma omp directive_name more_options
- Effectively used to parallelize code by hand (programmer needs to be aware of what can be parallelized, does not need to know how to write code for parallelizing it)
- For example, a loop can be fully parallelised, or partially parallelised, if you identify it, OpenMP can parallelize this part, you can customize by mentioning thread counts, etc.
- OpenMP provides for synchronization inherently.
- For task-level parallelism, OpenMP provides a "parallel" construct using which we can manually define the 15 regions of code that can be executed independently of the rest of the program.

3.6.2. Pthreads

- Library to spawn and manage POSIX threads
- The library is considered to be very effective for multi-processor and/or multi-core systems. They are useful when there is the scheduling of processes on specific processors/threads so as to gain improvements in both distributed and parallel computing.
- The overhead of using "fork" and spawning new processes is higher than the usage of threads. This is due to the system not needing to initialize the 13 new system virtual memory space and environment for every thread usage.

- The effectiveness of threads is higher on multiprocessor systems, but still, gains can be found on uniprocessors as well. This is due to the exploitation of latency in I/O and other possible interrupts.
- C++ thread library is built on top of Pthreads

GNU Parallel Algorithms:

- C++ provides a few parallel algorithms for a few standard sequential algorithms
- These are provided under different header files
- These are provided in a separate namespace as GNU extensions
- The following table shows some of the sequential algorithms and their equivalent parallel algorithms:

Algorithm	**Header**	**Parallel algorithm**	**Parallel header**
std::accumulate	numeric	__gnu_parallel::accumulate	parallel/numeric
std::adjacent_difference	numeric	__gnu_parallel::adjacent_difference	parallel/numeric
std::inner_product	numeric	__gnu_parallel::inner_product	parallel/numeric
std::partial_sum	numeric	__gnu_parallel::partial_sum	parallel/numeric
std::adjacent_find	algorithm	__gnu_parallel::adjacent_find	parallel/algorithm
std::count	algorithm	__gnu_parallel::count	parallel/algorithm
std::count_if	algorithm	__gnu_parallel::count_if	parallel/algorithm
std::equal	algorithm	__gnu_parallel::equal	parallel/algorithm
std::find	algorithm	__gnu_parallel::find	parallel/algorithm
std::find_if	algorithm	__gnu_parallel::find_if	parallel/algorithm

std::find_first_of	algorithm	__gnu_parallel::find_first_of	parallel/algorithm
std::for_each	algorithm	__gnu_parallel::for_each	parallel/algorithm
std::generate	algorithm	__gnu_parallel::generate	parallel/algorithm
std::generate_n	algorithm	__gnu_parallel::generate_n	parallel/algorithm
std::lexicographical_compare	algorithm	__gnu_parallel::lexicographical_compare	parallel/algorithm
std::mismatch	algorithm	__gnu_parallel::mismatch	parallel/algorithm
std::search	algorithm	__gnu_parallel::search	parallel/algorithm
std::search_n	algorithm	__gnu_parallel::search_n	parallel/algorithm
std::transform	algorithm	__gnu_parallel::transform	parallel/algorithm
std::replace	algorithm	__gnu_parallel::replace	parallel/algorithm
std::replace_if	algorithm	__gnu_parallel::replace_if	parallel/algorithm
std::max_element	algorithm	__gnu_parallel::max_element	parallel/algorithm
std::merge	algorithm	__gnu_parallel::merge	parallel/algorithm
std::min_element	algorithm	__gnu_parallel::min_element	parallel/algorithm
std::nth_element	algorithm	__gnu_parallel::nth_element	parallel/algorithm

std::partial_sort	algorithm	__gnu_parallel::partial_sort	parallel/algorithm
std::partition	algorithm	__gnu_parallel::partition	parallel/algorithm
std::random_shuffle	algorithm	__gnu_parallel::random_shuffle	parallel/algorithm
std::set_union	algorithm	__gnu_parallel::set_union	parallel/algorithm
std::set_intersection	algorithm	__gnu_parallel::set_intersection	parallel/algorithm
std::set_symmetric_difference	algorithm	__gnu_parallel::set_symmetric_difference	parallel/algorithm
std::set_difference	algorithm	__gnu_parallel::set_difference	parallel/algorithm
std::sort	algorithm	__gnu_parallel::sort	parallel/algorithm
std::stable_sort	algorithm	__gnu_parallel::stable_sort	parallel/algorithm
std::unique_copy	algorithm	__gnu_parallel::unique_copy	parallel/algorithm

3.7. Vector Representation of Source Code

3.7.1. code2vec: Learning Distributed Representations of Code

3.7.1.1. Introduction

- This paper enlists a method which uses a neural model to represent sections of source code as vector embeddings, which are essentially low-dimensional numerical vectors .
- These vector embeddings act as semantic representations of source code, capturing the meaning, intent and structure in the given section of code.

- The paper explains a method to first convert the code to its abstract syntax tree, then analyze different paths in the generated tree to produce individual vectors, which is finally aggregated into the final vector representation.
- Using these numerical representations of source code snippets, it is possible to use this in multiple applications such as code labelling, code captioning, clone detection and so on.

3.7.1.2. Implementation

- To capture the varying inherent importance of different sections of code within the given source program, it is necessary to identify the relative importance of these sections in influencing the final vector embeddings. The paper recognises this requirement and uses a path-based neural attention model.
- The paper also enlists a method to produce different vector embeddings for similar programs (not identical) to capture the subtle differences between the two programs. The attention model enables the calculation of a weighted average value on the attentions produced for different sections of code, extracted by the structure of an Abstract Syntax Tree.
- To represent the code in the form of an AST, numerical values are attached in a randomised manner in a bottom up approach to capture sub-trees. These numerical representations of the AST are then fed into the attention model which generates the final numerical representation of the input source code.

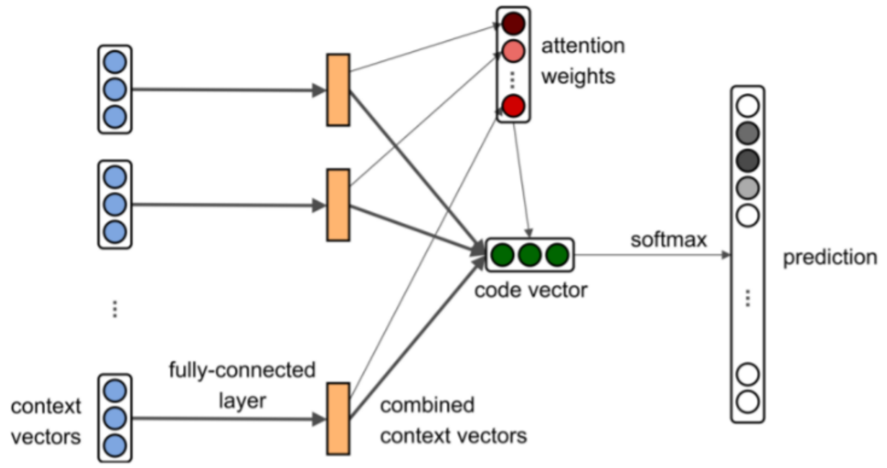


Figure 11: Architecture of Path-attention neural model used in Code2Vec

3.7.1.3. Conclusion

- The model requires an annotated training data set.
- The model is able to perform better than other popular vector representation techniques available due to the nature of the attention model concept.
- Vector embeddings capture subtle differences and can thus be used to perform algorithm recognition (program comprehension).

3

3.7.2. A Novel Neural Source Code Representation based on Abstract Syntax Tree

3.7.2.1. Introduction

34

- The paper proposes a method meant for representing source code, which performs better than other existing state-of-the-art models.
- This method uses an Abstract Syntax Tree based Neural Network model.
- Other existing techniques use the entire AST. However, there are issues related to the large size of the generated AST which could prove to be detrimental to the performance of the model. The proposed method chooses to break down the full AST into smaller statement trees.
- These smaller statement trees are then converted to vectors by capturing the lexical, as well as the syntactical information, present inherently in the source code.

3.7.2.2. Implementation

27

- The given source code is parsed into an Abstract Syntax Tree. It is split into smaller statement trees using the preorder traversal algorithm.
- Each of these smaller statement trees are encoded to vectors using "Statement Encoders".
- After this, Bidirectional Gated Recurrent Units are used to capture and model the inherent structure of statements. Multiple hidden states of the Recurrent units are combined into one single vector, which is the final representation of the given source code.

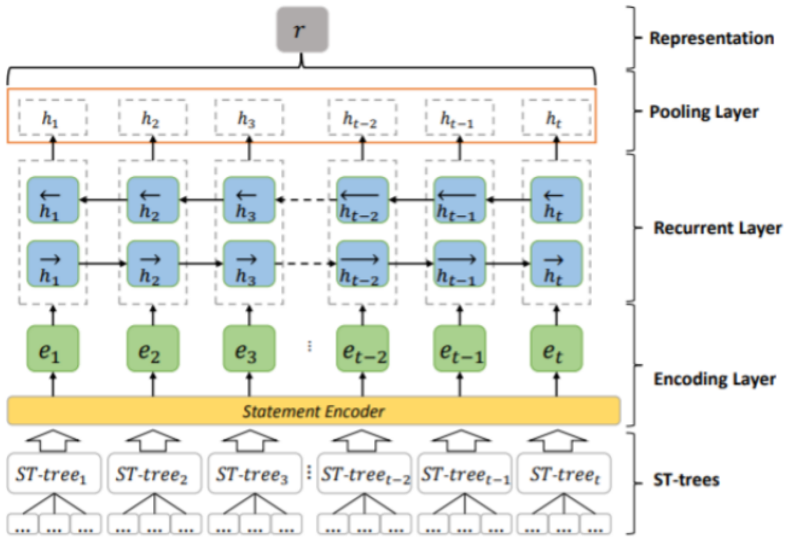

r
Representation
Pooling Layer
Recurrent Layer
Encoding Layer
Statement Encoder
ST-trees
h_1 h_2 h_3 h_{t-2} h_{t-1} h_t
e_1 e_2 e_3 e_{t-2} e_{t-1} e_t
$ST\text{-}tree_1$ $ST\text{-}tree_2$ $ST\text{-}tree_3$ $ST\text{-}tree_{t-2}$ $ST\text{-}tree_{t-1}$ $ST\text{-}tree_t$

Figure 12: Model architecture implemented in the paper

3.7.2.3. Conclusion

- The paper proposes an efficient approach to learn and represent vector embeddings of source code using "AST-based Neural Network (ASTNN)".
- The model is successful in capturing both the lexical and syntactical information in the given code, in addition to identifying the code structure.
- Evaluation of the model is done on two popular program comprehension applications - source code classification and code clone detection.
- The model requires large-scale datasets to perform effectively.
- The model is not language-agnostic. It doesn't perform well on multi-variety datasets.

4. Product Requirements Specification

4.1. Introduction

The purpose of this document is to elucidate the requirements of an Auto Parallelisation Software, which uses Program Comprehension, at both a functional and non-functional level. The document is intended to provide a brief description of the intricacies involved in building such a software, present an in-depth description of the functionalities included, and finally illuminate the non-functional requirements of the software.

4.2. Project Scope

The proposed project is an auto parallelisation tool that performs source to source compilation. It takes in the user code as input, brings it to an Intermediate Representation, analyses and segments the code into multiple parallel paradigms based on data and control dependencies and parallelizes the aforementioned segments efficiently.

Purpose:

Tool to automate the generation of parallel code covering a variety of problem types.

Benefits:

- Parallel programming helps in executing code efficiently.
- It saves time as it executes the applications in a shorter wall clock time.
- Larger, complex problems can be solved due to the ability to parallelize code.
- It reduces cost as sequential code leads to the under utilization of available hardware resources. Whereas parallel code tries to extract the best possible performance from the underlying hardware.
- Manual parallelization of code is a difficult and error-prone process. Automating the process of parallelization makes the entire process faster, easier and accurate, thereby saving valuable time and cost.
- The proposed tool should be able to cover a wide range of problem types. This generalization makes it indispensable to the development of efficient software.

Objectives:

- Comprehend the source code.
- Analyse data flow and data dependencies.
- Identify parallelizable segments.
- Identify hardware-dependent code optimizations.
- Generate and test equivalent parallel code.
- Check for correctness of the generated parallel code.

Goals:

- Automate the generation of parallel code.
- Cover a wide range of possibilities and variations with respect to the problem types.
- Generate equivalently correct parallel code without introducing additional vulnerabilities and issues.
- Parallel code generated should extract the best possible performance from the underlying hardware.
- Provide an efficient and accurate alternative to manually parallelizing code.

Coverage of the System:

- Ideally, the proposed software should be generalized enough to cover a wide range of possibilities with respect to the problem types that it can handle.
- The possible applications of the proposed software can be broadly divided into 2 categories:
    - **Parallelization of legacy software** - Softwares that have already been developed in a sequential manner can be parallelized by using our software, thereby resulting in considerable improvement in performance.
    - **As a tool to aid in the development of new software** - The number of developers skilled to write accurate parallel code is alarmingly low as most of them have been taught to develop code in a sequential manner. Thus, our proposed software can be used to develop new software that can be parallelized for best possible performance. This will end up saving valuable time and cost

for the software developers.

Limitations of the system:

- Generalizing the software to cover all possible types of problems could be a challenge.
- Effectively handling the issues that arise due to parallelization such as deadlocks, race conditions, starvation etc determines the success of the generated parallel code.

4.3. Product Perspective

Parallel computing has played a major role in a variety of areas such as computational simulations for scientific, engineering and commercial applications. The cost benefits that one gains along with the increase in performance provides compelling arguments in favour of parallel computing.

The need for parallelizing source code is ever increasing with the improvements in hardware and the advent of multicore and multithreaded processors. Without parallelizing source code, it is not possible to exploit the available hardware resources to extract the best possible performance.

However, it is difficult to find skilful developers capable of writing parallel code. It is also a difficult task to migrate and manually parallelise applications, the process being more error prone.

Automating the process of parallelizing source code would help overcome these problems, saving valuable time which would have otherwise been spent in studying, analysing, identifying and parallelizing the code manually.

We propose to design a novel pipeline to find sections of code that can be parallelised in the input program, through analyzing the algorithms used, integrating them and extracting common paradigms or cliches from them. The recognized regions of code are then modified into the corresponding parallelized equivalent, by using OpenMP directives and/or making calls to parallel implementations in C++ STL or other equivalent libraries.

4.4. Product Features

- Convert the input sequential source code to an intermediate representation (an Abstract Syntax Tree).
- Parse the tree to gather a basic understanding of the code to obtain predicates or rules.
- Identify and analyze the data and control dependencies.
- After aggregating all the information obtained until this point, the code is segmented into sections that can be parallelized.
- The parallelized equivalent of the source code is generated.
- Optimizations are done to fully utilize the underlying hardware resources.
- The generated code is tested and checked for correctness.
- Performance metrics are then used to evaluate the generated code.

4.5. Operating Environment

There are no specific requirements with respect to the environment per se.

But the system running the generated parallel code should have the necessary features to support multiprogramming. This is with respect to support for multiple threads, multiple processors and/or multiple cores.

4.6. General Constraints, Assumptions and Dependencies

- **Legal implications:** The source code being transformed by our pipeline should have the necessary permissions to access and update it. The tools, methods, approaches we are proposing to use should not infringe any existing copyrights, patents etc.

- **Usage limitations:**
  - Generalizing the parallelization process for any type of sequential code.
  - Effectively handling the issues that would arise due to parallelization such as deadlocks, race conditions, starvation etc.

- Handling the overheads involved in parallelizing the source code to gain sufficient speed up.

- It is assumed that the source code being processed is free from semantic and run-time errors that could manifest during or after the parallelization process.
- It is assumed that the end-user has a system that provides support for multiprogramming in terms of multiple threads, multiple processors, multiple cores.

4.7. Risks

- Failure of the dependency analysis of the code implies no guarantee to the correctness of the final transformation.
- Failure of a thread or starvation due to the creation of more threads or execution of other processes alongside would reduce the time efficiency or may even lead to a crash.
- If issues such as race conditions, deadlocks etc. are not handled properly, it could lead to serious issues.
- Hardware failures in terms of the failure of handling multiple threads, multiple processors, multiple cores could defeat the very purpose of parallelizing code.
- Version Compatibility problems: If hardware-specific optimizations are made, the generated code cannot be reused as is, on other systems. The code will have to be sent through our pipeline again and the corresponding parallel code should be generated specific to that hardware.

4.8. <u>Functional Requirements</u>

Validity Test on inputs:

- The inputs are the same as the ones being processed by the original sequential source code.
- So, as long as the input is valid for the original source code, it will be valid input for our generated parallel code as well.

The sequence of operations:

Intra-Function Parallelism:

- Convert source code to an Intermediate Representation (like an enriched Abstract Syntax Tree).
- Parse the tree with a top-down and requirement-driven approach to identify basic concepts. These concepts are represented either as rules or embeddings.
- Compose the basic concepts to form bigger known algorithms.
- If any algorithm is found, match the corresponding region of code and identify the data and control dependencies for the region.
- Choose from one of the parallel alternatives from the algorithm repository to modify and replace the corresponding region in the AST.
- Verify if the modified region fits in accurately by analysing the data and control dependencies around the region.
- After the AST is modified, we use our compiler to decompile the AST back to the original source code language.

Inter-Function Parallelism:

- We perform a similar analysis as before to retrieve the data and control flow and dependencies from the Abstract Syntax Tree of the source code.
- Based on the results of the analysis, an appropriate parallel paradigm is chosen
- This parallel paradigm is applied to the source code and an attempt is made to parallelise the same at a functional level
- A generator program generates a new source code that complies with the parallel paradigm suggested.
- Verification is then performed to ensure the correctness of the code.

Error handling and recovery:

Various errors and issues might arise due to parallelization. These include:

- Deadlocks
- Race conditions
- Starvation

It might also be difficult to effectively manage multiple threads. It may result in coherency issues and so on.

There is a requirement to handle these error conditions in the right manner.

Consequences of parameters:

- The information about the underlying hardware configuration is significant to generate the most optimized version of the parallel code possible, that extracts every bit of performance from the underlying hardware.
- Information needed to evaluate the performance metrics:
    - The wall clock execution time of the original sequential source code
    - The wall clock execution time of the generated parallel code
    - Number and type of hardware resources available
- The sequential source code being fed into our pipeline must be free from semantic and run-time errors as these will propagate to the generated parallel code and there is nothing that our software can do about it.

Relation of output to input:

Output is just the parallelized version of the input sequential source code.

The difference could be in terms of OpenMP pragma directives added or replacement of segments of code with the most efficient parallel version of it etc.

4.9. External Interface Requirements

4.9.1. User Interfaces

- The proposed pipeline, implemented as a tool, should have an easy to use interface - a GUI that lets the user upload the input sequential source code.
- The tool performs the different stages in the pipeline and outputs the generated parallel code if applicable.
- The performance metrics are also shown with the evaluated values.
- The relative timing of inputs and outputs: The generation of output should not take a considerable amount of time from the moment the input is fed into the pipeline.

4.9.2. Hardware Requirements

- Input is fed from standard input devices like keyboard, mouse etc.
- Output devices include standard output devices like display monitor, printer etc.
- The software product should be able to perform the required I/O through these devices.
- It should have the required permissions to spawn multiple threads and access all the available cores or processors.

4.9.3. Software Requirements

4.9.3.1. GCC/G++

Since the proposed software would be generating parallel code for the C++ language, we would need the gcc/g++ utility to compile and generate the required executables.

Version Requirements: GCC 5.0 onwards

4.9.3.2. CLAVA/LARA

The LARA language facilitates querying on the Abstract Syntax Tree of the input code. Clava is an S2S compiler for C/C++, built with LARA as its foundation. It provides better ways to modify and transform C/C++ code, statically and dynamically.

4.9.3.3. Python

Python utility is needed as some of the helper functions and code required to perform some functionalities will be written in python.

Version Requirements: Python 3.0 onwards

4.9.3.4. OpenMP

OpenMP provides support for parallel programming in C, C++, Fortran in shared memory environments. It was introduced in 1997. The programmer has to explicitly mention the regions that can be parallelized. This is done using the OpenMP directives which are used to annotate the sequential code. OpenMP initiates the required number of threads (can be set by the user) and runs the parallel region across these threads.

Version Requirements: OpenMP version 4.0 onwards

4.9.3.5. Pthreads

Pthreads library will be required to create, spawn and manage threads as the threads are the basic units of execution and the means by which parallelism can be achieved.

4.10. Non-Functional Requirements

4.10.1. Performance Requirement

- The performance at the very minimum should match that of the original sequential source code.
- The speedup gained from parallelizing should be substantial enough to overlook the overheads involved in generating the parallel code.
- Should ensure speedup or efficiency boost in a particular or all metrics as per user requirements
- The performance is evaluated against various metrics as follows. For the generated parallel code to be acceptable as per the user requirements, the calculated performance metrics should be greater than a certain threshold as specified by the user.

Performance Metrics :

- **Speedup:** ratio of sequential to parallel execution times
- **Efficiency:** ratio of performance to the computational resources used to gain that performance
- **Redundancy:** ratio of the number of instructions executed by the sequential to the parallel version of the code
- **Utilization:** the ratio between the computational resources used and the resources available
- **Restrictions:** limitations as defined by Amdahl's law and other related laws

Software Quality Attributes:

- **Correctness:** The generated parallel code should compare with the original sequential source code in terms of correctness. The newly generated parallel code should successfully pass all the test cases that the original sequential source code passes. The outputs generated must be similar in content and format. The only considerable difference must be with respect to the speed up.

- **Maintainability:** The pipeline developed should have clean code and should be well documented to ensure that maintenance is easy.
- **Reliability:** It should reliably generate parallelized code without adding any issues.
- **Robustness:** It should be able to handle a wide spectrum of problem types in terms of generating the correct parallel code for it.
- **Testability:** It should be testable for different scenarios.
- **Usability:** It should have an easy and convenient to use interface that anyone can use with minimum additional knowledge.

4.10.2. Safety Requirements

- There should not be any loss or damage to the original source code that is being parallelized.
- The generated code should not damage or corrupt the data that it works on.

4.10.3. Security Requirements

- The proposed software should not copy or store any temporary files that may be generated.
- The data generated during the process of parallelization should be completely discarded after the required parallel code is generated. If not, this will lead to issues such as violation of security and privacy rules of the original software whose source code is being parallelized. This is very significant as the software might include sensitive data, patented and copyrighted technology.
- We should ensure that no security breaches are introduced in the modified source code
- The security standards of the original sequential source code should be maintained as such. Security vulnerabilities should not be introduced while generating the parallel code.

5. System Design

5.1. Introduction

In the earlier days, there was very little computational power that could be provided by the processors on a computer system. Software was written sequentially, which meant every instruction was executed one by one. This meant large software took a very long time to execute.

But there was an exponential boom in the development of computer hardware, with faster and more powerful processors coming into the market. The clock speed drastically increased which enabled more instructions to be executed in very short durations of time. However, this continuous increase in power and computational capacity was cut short a few years ago. This meant that the ever-increasing need for more and more computational power could no longer rely on developing hardware. We had to shift focus to the software side, to make intelligent and efficient use of available hardware.

Parallel computing is a paradigm that enables us to make efficient use of available hardware to run multiple instructions on multiple hardware resources, simultaneously. This could mean running separate tasks on multiple cores/threads available on a single system or running separate tasks on multiple systems connected in a cluster. This enables us to execute our software faster and saves valuable time in several use cases. Thus, we propose a novel approach to solve the problem of Parallelisation of sequential code, by the use of multiple techniques integrated into one package to convert any general sequential source code to its parallel equivalent source code. We achieve this primarily by using the concepts of Program comprehension, Task and Functional level parallelism.

The purpose of this document is to elucidate the high-level design of an Auto Parallelisation Software, that takes in sequential source code as its input and generates the parallelized equivalent. The document is intended to provide details about the design of the software.

The high-level design of the proposed software includes the following:

- A component or module capable of generating an enriched Abstract Syntax Tree (AST).
- The generated AST should be such that it should enable us to query the nodes to retrieve useful information about the input sequential source code.
- A component or module that queries the AST and analyzes the input code to generate 18 data dependency graphs, control flow graphs, data flow graphs.
- A component or module that assimilates the information from the various graphs generated. The information extracted from this is used to determine the segments of code that are candidates for parallelization.
- A component or module that takes in these candidate segments of code, parallelizes them using different techniques proposed and developed by us to generate the equivalent parallel code.

5.2. Current System

Present-day tools designed to convert sequential to parallel code focus mainly on loop-level parallelization. Several tools implementing different techniques to achieve loop level parallelisation exist such as Pluto, Parawise, Compar, etc. These are also called Parallelizing Compilers. Some of the most popular ones and their approach is mentioned below in brief:

- **Pluto**

  This tool makes use of a concept called the Polyhedral model. This tool focuses on achieving loop-level parallelization and achieves a good degree of optimisation.

- **ParaWise**

  This is a commercially available paid tool designed to convert sequential code to parallel code pertaining to loops and some level of task parallelization. It provides ample customization options to users to control the level of parallelization.

- **ComPar**

  This is a semi-automatic tool designed to optimize code by fusing other source-to-source compilers' outputs that can be achieved from auto parallelizing compilers which need no manual intervention. It makes use of the outputs obtained from other S2S compilers such as AutoPar, Cetus and Par4All.

The approach we present is focused on achieving more generalised parallelization. We do so by trying to understand the intent and logic of code, termed as Program comprehension, which enables us to replace such code with its parallel equivalent, thus achieving task level parallelization. Our design also includes a Master-Worker scheduling algorithm which enables us to execute independent functions on multiple threads, thus achieving functional parallelization.

5.3. Design Details

The platforms, systems and processes that the proposed software depends on are:

- A process that generates an enriched AST: The Abstract Syntax Tree corresponding to the input sequential source code must be generated. Ubiquitous tools and compilers like gcc, clang generate AST. The generation of the AST should be modified to contain more information about the input sequential source code so that we can query the nodes of the enriched AST effectively.

33

- A process that generates the Data Dependency Graphs, Control Flow Graphs and Data Flow graphs: The generation of accurate Data Dependency Graphs, Control Flow Graphs and Data Flow Graphs is significant to ensure the success of the forthcoming stages in the proposed pipeline. Ubiquitous tools and compilers like gcc, clang generate some of these graphs. They will have to be modified to better suit our requirements in terms of the analysis that needs to be performed on it to extract meaningful insights about the input sequential source code.

- **GCC/g++**

  Since the proposed software would be generating parallel code for the C language, we would need the GCC utility to compile and generate the required executables.

  g++ will be required as the development of some of the functionality in our proposed software is done in C++.

Version Requirements: GCC 5.0 onwards

- **CLAVA / LARA**

  The LARA language facilitates querying on the AST of the input code

  Clava is a Source to Source compiler for C/C++, built with LARA as its foundation. It provides better ways to modify and transform, statically and dynamically, C/C++ code.

- **Python**

  Python utility is needed as some of the helper functions and code required to perform some functionalities are written in python.

  Version Requirements: Python 3.0 onwards

- **OpenMP**

  One of the approaches we are considering involves generating parallel code that consists of OpenMP directives inserted at the right place in the input sequential source code. This is the reason the proposed software requires the OpenMP utility.

  Version Requirements: OpenMP version 4.0 onwards

- **Pthreads**

  Pthreads library will be required to create, spawn and manage threads as the threads are the basic units of execution and the means by which parallelism can be achieved.

- **Thread Pool:**

  The proposed software involves the implementation of thread pools to enable us to gain fine grained control over the executing threads. We use an open source implementation of thread pool, which is a C++ header only library which provides a definition of a simple thread pool, capable of accepting and executing tasks and provides a means to capture the return value of functions using the C++ concept of future and promises.

5.3.1. Novelty

Our proposed design integrates multiple techniques to convert sequential code to its parallel equivalent. We make use of program comprehension to understand the nature and intent of code, and replace relevant sections of code with the parallelised equivalent using mappings. Such sections/functions of code are then executed in parallel by identifying the underlying data and control dependencies, enabling us to execute independent portions of code simultaneously. To achieve fine grain control over efficient execution of such multiple functions, we design a scheduling algorithm based on the dependencies, allowing for better control and improving performance in an adaptive manner. This proposed methodology is novel based on our literature survey.

5.3.2. Innovativeness

The proposed methodology of scheduling threads after checking for data dependencies, and using program comprehension for improving individual function performance by replacing them with optimised parallel program is a technique that has not been tried before. The method allows for a new approach that may shed new perspective and also open avenues for more work on the front, especially due to the fine control gained by the way we have implemented thread scheduling.

5.3.3. Interoperability

The input so far is C/C++ code, and the resultant program is in C++. This allows for the program to be run on any machine after compilation. However, the performance will depend on the machine, the number of threads and any related machine dependent optimizations.

5.3.4. Performance

The performance at the very minimum should match that of the original sequential source code.

- The speedup gained from parallelizing should be substantial enough to overlook the overheads involved in generating the parallel code.

● Should ensure speedup or efficiency boost in a particular or all metrics as per user requirements

The performance is evaluated against various metrics applicable to our proposed software as follows:

- **Speedup:** ratio of sequential to parallel execution times.
- **Efficiency:** ratio of the performance to the computational resources used to gain that performance.
- **Redundancy:** ratio of the number of instructions executed by the sequential to the parallel version of the code.
- **Utilization:** ratio between the computational resources used and the resources available.

5.3.5. Security

The proposed methodology should not copy or store any temporary files that may be generated. Also, the data generated during the process of parallelization should be completely discarded after the required parallel code is generated. If not, this will lead to issues such as violation of security and privacy rules of the original software whose source code is being parallelized. This is very significant as the software might include sensitive data, patented and copyrighted technology.

We will need to ensure that no security breaches are introduced in the modified source code. That is, the security standards of the original sequential source code should be maintained as such.

5.3.6. Reliability

As the resultant code is in C++, there is no issue with the code working on different machines. However, we have to ensure that the function runs as it used to previously and produces the same results as before parallelization. To ensure this, we make sure that all the data dependencies are satisfied before the usage of any data by any thread. This is done by a thorough analysis of the data dependency graph and using that information in scheduling in a manner ensuring the satisfaction of data-dependencies.

5.3.7. Maintainability

All our source code is well documented with comments. Additionally, our report carries a detailed explanation of our ideas, and implementation. This is useful in understanding the purpose of our code and the design thought behind it. We have ensured from the start to keep the project in an organised manner, allowing for incremental improvements, while keeping the structure easy to understand.

5.3.8. Legacy to modernization

By using our methodology, upon scaling, it would be possible to convert legacy code written in a sequential manner into one that runs on parallel architecture. This would help improve the performance of the legacy code, with the improvement depending on the program, it's domain and the design of the program.

5.3.8. Reusability

The progress made by our work could find potential reusability in GPU programming and possibility of automating conversion of programs into GPU specific architectures, like into CUDA or Vulkan programs. This could be an area of research where our methodology could have a possible impact, with some necessary modifications in the idea to suit those exact system requirements.

5.3.9. Application compatibility

Our methodology tries to be as system independent as possible, with regards to compatibility. We have tried to maximize the number of threads utilized, while not setting any minimum threshold. Since all our code is written in languages that have a wide range of support across systems and OS's, we suspect no specific compatibility issues to arise. However, we do require support for OpenMP, gcc (along with g++) and python, as dependencies. We have assumed the underlying architecture to have random access memory, and also to have multiple-cores and threads.

5.3.10. Resource utilization

The very core intent of our research has been to improve the performance of programs by the utilisation of existing resources that are otherwise not being used. Our design so far uses as many threads as possible in it's thread pool during scheduling of the client program, to run in parallel.

6. Implementation and Pseudo Code

In our research, we have set our goal to find the limit of parallelism we can achieve for the given sequential source code. The objective is to gain the maximum possible speedup that can be achieved by parallelizing a particular program. In this regard, we have experimented with different approaches that work on achieving parallelism at different levels of abstraction and hence providing different degrees of control over the execution of the program itself.

Some Assumptions made while trying out the different approaches during Capstone Phase 1:

- The input sequential source program consists of atomic functions that perform only one functionality.
- Our research on Program comprehension techniques to convert sequential to parallel code has provided us with valuable insights on analysing a program to obtain its intent(s). This enables us to predict and define a paradigm (or cliche or algorithm) associated with every function/region of code. The different paradigms are pre-defined and stored in a database, and the corresponding mappings to one or more optimised versions for the paradigm are defined and stored as well.
- In Phase 1, we have assumed we have the paradigm identifier and database set up, which enables us to modify and replace code within functions to achieve intra-function parallelisation. By assuming the availability of this functionality, we have been able to work on Intra and Inter level functional parallelism and show meaningful results at the end of Phase 1.

- We plan to work on the paradigm identifier and populate the corresponding database at the beginning of Phase 2.
- The cases we are trying to handle now predominantly involve programs with functions (procedural programming).
- The function calls are sequential, i.e there are no selection, looping statements encapsulating these function calls. However these can be present within the functions themselves.

6.1. Parallelization Phase

6.1.1 Method 1: Inter and Intra-Function Parallelism by AST Querying and replacement with OpenMP Directives

6.1.1.1 Details about the approach:

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is carried out using a tool called Clava.
- Next, querying of the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript and is compatible with the Clava tool, thereby allowing us to query the AST.
- The following two functionalities have been implemented:
    - **Functional parallelism**
        - Here, the primary objective is to find functions that are independent of each other.
        - Functions are said to be independent of each other if they are performing operations on different sets of data. If they are working on the same data, none of the functions involved should be modifying this data. However, they are allowed to read the data simultaneously.
        - To identify the candidate functions that can be run parallely (functional parallelism), data dependencies that exist are found.
        - This is done by performing read-write dependency analysis.

- Two functions are inherently said to be candidates for functional parallelism if they work on different data. Hence these functions are independent of each other and can be parallelized.
- If the functions being considered work on the same data, the following cases arise:
  - If the argument(s) used by the function(s) is/are passed by using the mechanism of pass by value, then any modification being made to the argument(s) is local to the function only. Hence the functions being considered are independent.
  - If the argument(s) used by the function(s) is/are passed as constant parameters, then it means that these functions cannot modify these argument(s), hence the functions being considered are independent.
  - If the argument(s) used by the function(s) is/are passed by using the mechanism of pass by reference, then any modifications being made to the argument(s) within a function will propagate across the entire program. Therefore read-write dependency analysis will have to be performed to find if there is indeed a write to such an argument happening within the function.
    - If a write is being performed to such an argument, then the functions cannot be parallelized, they have to be run in the same sequential order they are being called by the user, to maintain the intended semantics.
    - If there exists only read dependencies and there is no write being performed, then the functions are independent and hence can be run in parallel.
  - It is not enough to only consider arguments being passed to the functions under consideration to decide whether they are independent. There can also be cases where the functions use and modify global variables. Read-write dependencies are analyzed for such global variables as well.
    - If the functions under consideration only read these global variables, then the functions are independent and can be run in parallel
    - If the functions under consideration write into these global variables, then the functions using these global variables cannot be parallelized, they have to be run in the same sequential order they are being called by the user, to maintain the intended semantics

- Once the independent functions are established, they are parallelized by inserting OpenMP directives at accurate points in the code. This is done by creating OpenMP sections and associating each parallelizable function to one OpenMP section.

- **Intra Function Parallelism (Parallelism within the function)**
  - Here, considering that the functions are atomic having only one purpose and that we know what each function is intended to do (we obtain this information from the defined paradigm identifier and pre-defined database, which we have assumed is available in Phase 1, on which we intend to work at beginning of Phase 2), we try to map the functions to the parallelized equivalent of it.
  - For example, if the user has a function to perform sort where the user has written his own logic, we can map it to the gnu_parallel:sort algorithm available under the parallel algorithms library in C++.
  - There are obviously going to be cases where the function cannot be mapped to an algorithm in the parallel algorithms library. In such a scenario, we would like to map the user’s sequential code to the parallelized equivalent that we have written. Hence a database needs to be created to map different possible sequential algorithms to their parallelized equivalents. This has to be made comprehensive with respect to the number of different cases that can arise.
  - Once the mapping is identified, the user's code within the function is replaced by the mapped parallelized equivalent.

- The parallelized equivalent of the input sequential source code is generated by applying the above-mentioned functionalities.
- The speedup achieved was considerable as can be seen in the following graph:

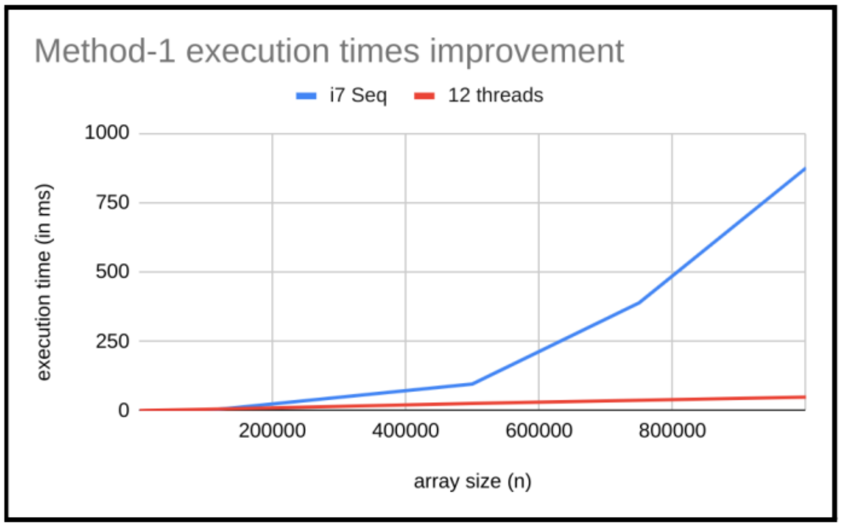


Figure 13 : Results of Method-1 on i7 9th gen

Input Code:

```
void sort(int* arr_,const int arr_n)
{
      for ( int x = 0;x < arr_n-1;++x)
      {
            int min_ = i;
            for(int y = x+1;y < arr_n;++y)
            {
                  if(arr_[y] < arr_[min_])
                  {
                        min_ = y;
                  }
            }
            int temp_ = arr_[x];
            arr_[x] = arr_[min_];
            arr_[min] = temp_;
      }
}
```

```
int my_max(int* a4, int n4)
{
        int max = a4[0];
        for ( int x = 1; i < n4; ++x)
        {
                if(a4[x]>max)
                {
                        max = a4[x];
                }
        }
        return max;
}

int my_min(int* const a4, const int n4)
{
        int min = a4[0];
        for ( int x = 1; x<n4; ++x)
        {
                if(a4[x]<min)
                {
                        min = a4[x];
                }
        }
        return min;
}



int main()
{
        int arr1[] = {7, 34, 5, 3, 1};
        int n = sizeof(arr1) / sizeof(arr1[0]);

        my_sort(arr1, n);
        int min1 = my_min(arr1, n);
        int max1 = my_max(arr1, n);
}
```

Output Code:

```
void my_sort(int *arr, int const arr_n) {
    __gnu_parallel::sort(arr, arr + arr_n );
}
```

```
int my_max(int *a4, int n4) {
   return *__gnu_parallel::max_element(a4, a4 + n4 );
}

int my_min(int * const a4, int const n4) {
   return *__gnu_parallel::min_element(a4, a4 + n4 );
}

int main() {
   int arr1[5] = {7, 34, 5, 3, 1};
   int n = sizeof(((arr1))) / sizeof(((arr1[0])));
   my_sort(arr1, n);
   #pragma omp parallel sections
   {
      #pragma omp section
      int min1 = my_min(arr1, n);
      #pragma omp section
      int max1 = my_max(arr1, n);
   }
}
```

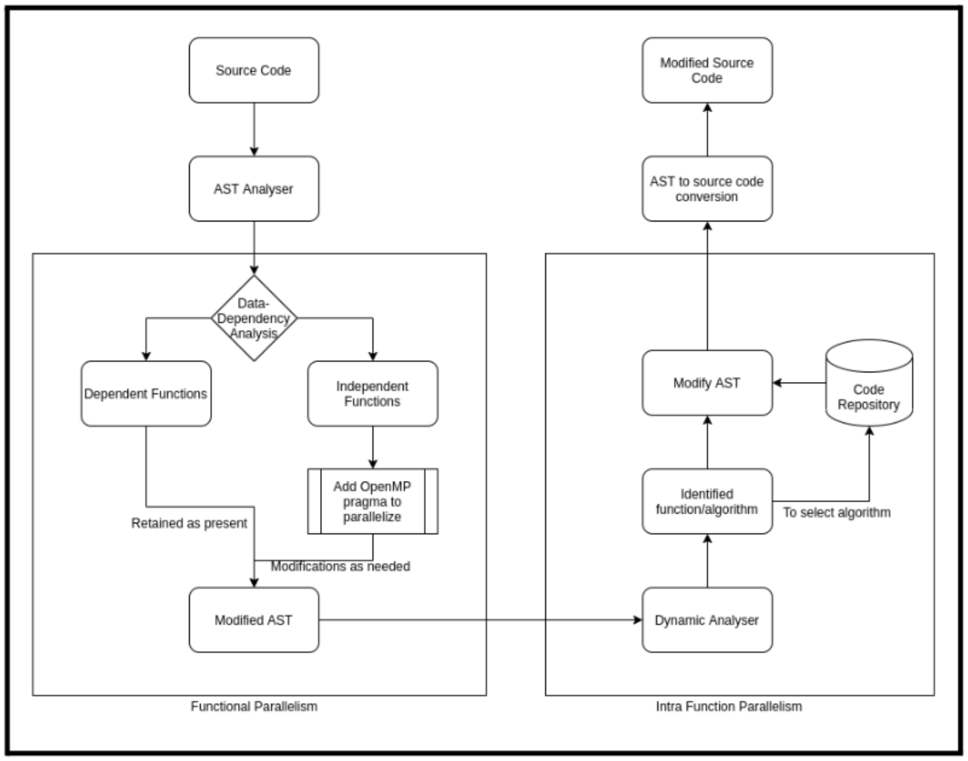


Figure 14: Method-1 implementation

6.1.1.2. Inferences from Method 1

- The degree of parallelism that could be achieved was still limited. This is because we add OpenMP directives and the rest of the details are abstracted. Parallelizing of the code based on the directives inserted, creating, spawning threads, assigning functions to the threads, scheduling and managing these threads are all abstracted as this is managed by the OpenMP library based on the directives we have inserted.
- Hence we don't have the much-needed flexibility with respect to the above parameters, thereby limiting the parallelism that can be achieved.
- This method could also not support the grouping of independent functions together due to limitations with respect to the OpenMP directives.

- We realized that we would need to gain more fine-grained control over the actual execution of threads than what we could achieve using OpenMP.
- The above-mentioned reasons were the motive to consider a different approach - namely Method 2.

6.1.2. Method 2: Naive Thread Scheduling using C++ concepts of Promises and Futures

6.1.2.1. Details about the approach

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is done using a tool called Clava.
- Next, querying the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript that is compatible with the Clava tool, thereby allowing us to query the AST.
- We then perform a read-write dependency analysis to get significant information about the input sequential source code.
- The information assimilated from the read-write dependency analysis performed above combined with the ability to query the enriched AST is used to populate text files that will be used by the next phase in our pipeline.
- The information about the functions that are modifying the arguments and the arguments being modified by the respective functions are written into one text file.
- The order of function calls along with information about the function name, the arguments being passed to these functions are written into another text file.
- The next phase in this approach includes the naive thread scheduling algorithm written using C++ promises and futures.
- The text files generated are used by the thread scheduling algorithm.
- The information gathered from these text files is used to establish the order in which the functions can be called to exploit the possible parallelism.

- Functions that have no dependencies are executed in parallel.
- Functions that have dependencies are reordered in such a way that these functions are executed only after the dependencies are satisfied as mentioned in the original sequential code, thereby maintaining the semantics of the program.
- This is achieved by using the concept of C++ promises and futures.
- The use of C++ promises and futures ensures that no function that has a dependency is executed until all its dependencies are fulfilled.
- C++ thread libraries are used to create, spawn and schedule the threads.
- One function is executed as part of a thread.
- The promises and futures are associated with the threads based on the dependencies

Following are some test cases and obtained results:

Example 1: (Works with this method)

Input program (only the function calls inside main):

```
fn_A(arr1,n)
fn_B(arr1,n)
fn_C(arr1,n)
fn_D(arr2,n)
fn_E(arr2,n)
```

Constraints:

fn_A function modifies arr1, which implies fn_B and fn_C dependent on arr1, is in turn dependent on the execution of fn_A

Output:

```
std::promise<void> p_arr1_0
thread ti(fn_A, params);
thread ti(fn_D, params);
thread ti(fn_E, params);
std::future<void> f_arr1_1= p_arr1_0.get_future();
thread ti(fn_B, params);
thread ti(fn_C, params);
```

Analysis:

Function calls fn_B and fn_C with arguments arr1 need to be executed only after fn_A on arr1 finishes its execution.

However, fn_D and fn_E are two function calls on arr2, which can be independently executed.

Hence the reordering of function calls to schedule fn_A, fn_D, fn_E happen as expected. Future is called on a promise set on fn_A(arr1, n). This implies, there is a wait on fn_A(arr1, n) to finish its execution, only then the consecutive statements are executed. Thus fn_B and fn_C on arr1 are correctly executed on a possibly modified arr1 variable.

This is a test case that behaves as expected and can be scheduled using this method.

Example 2: (Not optimised with this method)

Input program (only the function calls inside main):

```
fn_A(arr1,n)
fn_B(arr1,n)
fn_C(arr1,n)
fn_A(arr2,n)
fn_B(arr2,n)
fn_C(arr2,n)
```

Constraints:

fn_A function modifies arr1

fn_A function modifies arr2

fn_B and fn_C is dependent of execution of fn_A

Output:

```
std::promise<void> p_arr1_0
thread ti(fn_A, params);
std::promise<void> p_arr2_0
thread ti(fn_A, params);
std::future<void> f_arr1_1= p_arr1_0.get_future();
thread ti(fn_B, params);
thread ti(fn_C, params);
std::future<void> f_arr2_1= p_arr2_0.get_future();
thread ti(fn_B, params);
thread ti(fn_C, params);
```

Analysis:

There are two function calls - fn_A(arr1, n) and fn_A(arr2, n), which are modifying their respective arguments (in this case, arr1 and arr2 respectively). This means that fn_A and fn_C function calls on arr1 and arr2 need to be executed only after the execution of fn_A on arr1 and arr2 respectively.

The fn_B and fn_C calls can independently execute as long as their dependency on corresponding fn_A calls is satisfied. However, in the output generated by this method, there is an unnecessary wait on completion of fn_A(arr1, n), which is affecting the execution of fn_B and fn_C on arr2, something which is completely independent of this particular fn_A call. Hence such cases cannot be handled using this method.

The future approach is to obtain fine grained control through our own thread scheduling algorithm.

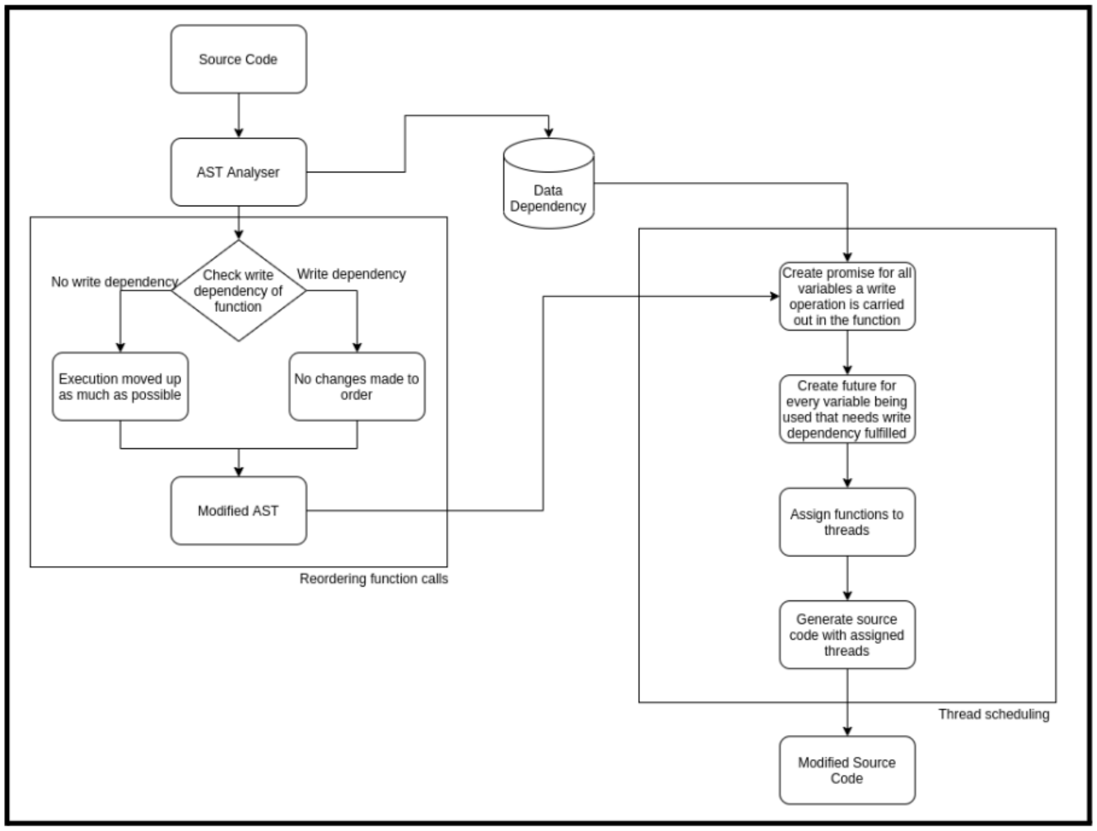

Source Code
AST Analyser
Data Dependency
Check write dependency of function
No write dependency
Write dependency
Execution moved up as much as possible
No changes made to order
Modified AST
Reordering function calls
Create promise for all variables a write operation is carried out in the function
Create future for every variable being used that needs write dependency fulfilled
Assign functions to threads
Generate source code with assigned threads
Thread scheduling
Modified Source Code

Figure 15: Method-2 implementation

6.1.2.2. Inferences from Method 2:

- This method provided a more fine grained control over thread creation, management and scheduling giving more flexibility and opportunity to maximize the parallelism that can be achieved.
- This method allowed us to go a level down in terms of the abstraction as we are dealing with threads directly in this approach.
- One of the major drawbacks of this approach was that grouping of function calls could not be performed properly.
- The reordering of functions had limitations to the cases where it was accurate and provided sufficient speed up.

- As grouping of functions could not be done, there were cases where certain functions ended up waiting for the completion of execution of certain other functions they didn't even depend on, even when the required dependencies for the execution of this particular function had already been satisfied.
- This is not desirable and thereby we needed a more fine grained control over thread scheduling to be able to achieve the maximum possible speedup by parallelism.
- This was the intent behind our next approach - namely Method 3

6.1.3. Method 3: Optimised Thread Scheduling for Functions using Master-Worker based approach to achieve Functional Parallelism

6.1.3.1. Details about the approach:

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is done using a tool called Clava.
- Next, querying the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript that is compatible with the Clava tool, thereby allowing us to query the AST.
- We then perform a read-write dependency analysis to get significant information about the input sequential source code.
- The information assimilated from the read-write dependency analysis performed above combined with the ability to query the enriched AST is used to populate text files that will be used by the next phase in our pipeline.
- The information about the functions that are modifying the arguments along with the information about the arguments being modified by these functions is written into one text file.
- The order of function calls along with information about the function name, the return type of the function, the arguments being passed to these functions, the parameters of the

functions, the data types of both the arguments and the parameters are written into another text file.

- The information about the functions, their return types, the line number where the function is being called and the variable to which the returned value is being assigned (in case of a non - void return type).
- The next phase in this approach is the thread scheduling algorithm.
- The text files populated by the previous phase are used by the thread scheduling algorithm.
- The information from these text files is assimilated and stored in appropriate data structures.
- The main idea behind this approach is to follow a master worker based method to ensure the execution of processes that are independent of each other simultaneously and ensure the right order of execution of dependent functions, to maintain the correctness of execution.
- Here, there are two master threads that are always running.
    - One master thread that schedules the functions to the worker threads
    - One master thread that tracks the worker threads to know their status
- We make use of 2 separate task queues - ready and wait queues.
- The wait queue is intended for those functions which are dependent on a previously called function(s). This implies that such functions need to wait until its dependency functions finish their execution. Such functions get appended into the wait queue.
- Once the dependency functions finish their execution, the functions waiting in the wait queue are moved into the ready queue, implying they are ready to be executed. Functions inside the ready queue are assigned separate threads for execution. Once assigned, such functions are dequeued from the ready queue.
- To ensure the synchronization is maintained in common data structures used to carry out the scheduling process, such as the ready and wait queues, other lists used to maintain dependent functions and corresponding arguments, mutex locks have been used to prevent any race conditions.
- To make our scheduling algorithm generalized for any sorts of client programs, we make use of a program generating technique, which produces the parallelised version of the client code, with appropriate thread allocation and mutex locks in place.
- To avoid excessive overheads generated due to the allocation of new threads and the deallocation of existing threads when the processes finish their execution, we make use of a thread pool. 29 A thread pool uses a fixed number of threads, which remain allocated until all

the processes finish their execution. It makes use of the C++ concepts of 'future' and 'promise' to obtain the return value of functions that are assigned as processes to threads. This avoids repeated cycles of allocation and deallocation of threads and improves efficiency.

- Processes are assigned to the thread pool, which maintains and coordinates the assignment of the process to one of the available threads. To exploit the full hardware potential available on the system, we assign all available hardware threads to the thread pool, thereby increasing performance.

Input program (only the function calls inside main):

```
fn_A(arr1,n)
fn_B(arr1,n)
fn_C(arr1,n)
fn_A(arr2,n)
fn_B(arr2,n)
fn_C(arr2,n)
```

Constraints:

fn_A function modifies arr1

fn_A function modifies arr2

Output:

In the generated parallel program, following is the sequence of events:

```
// declare "special" array which holds all arguments of function calls
presently being executed
// define two master thread functions to update special array and move
functions from wait queue to ready queue when there are no conflicting
dependencies
// Push function call to either ready or wait queue depending on
whether any of its arguments is present in the special array presently
```

```
push_to_ready_queue(fn_A, arr1, n)

push_to_wait_queue(fn_B, arr1, n);
push_to_wait_queue(fn_C, arr1, n);
push_to_ready_queue(fn_A, arr2, n);
push_to_wait_queue(fn_A, arr2, n);
push_to_wait_queue(fn_A, arr2, n);
```

Analysis:

The input program was the same as the one used in the previous method.

Previously in Method 2, the fn_A(arr2, n) was waiting unnecessarily until fn_B and fn_C finished their respective executions (which were indirectly waiting for fn_A to finish its execution). However, with the use of dynamic wait and ready queues, functions are pushed to execution queues purely based on their arguments dependency on currently executing functions. In this case, fn_A(arr2, n) wouldn't need to wait on anything since none of its arguments (both arr2 and n) are presently not being modified by any previous methods. Thus, there is no unnecessary waiting and execution flow is continuous.

Thus, in this method we were able to ensure independence among the two dependent function clusters through fine grained control over the scheduling of threads. Execution of the first cluster is independent of the execution of the second cluster. Intra-cluster dependency is still maintained implying that execution of fn_B and fn_C on arr1 would still wait on the execution of fn_A on arr1. By having fine grained control over the thread scheduling algorithm, we were able to fix the inter-cluster dependency issue that was present in the previous method.

Comparison of execution times between sequential program and the generated parallel program:

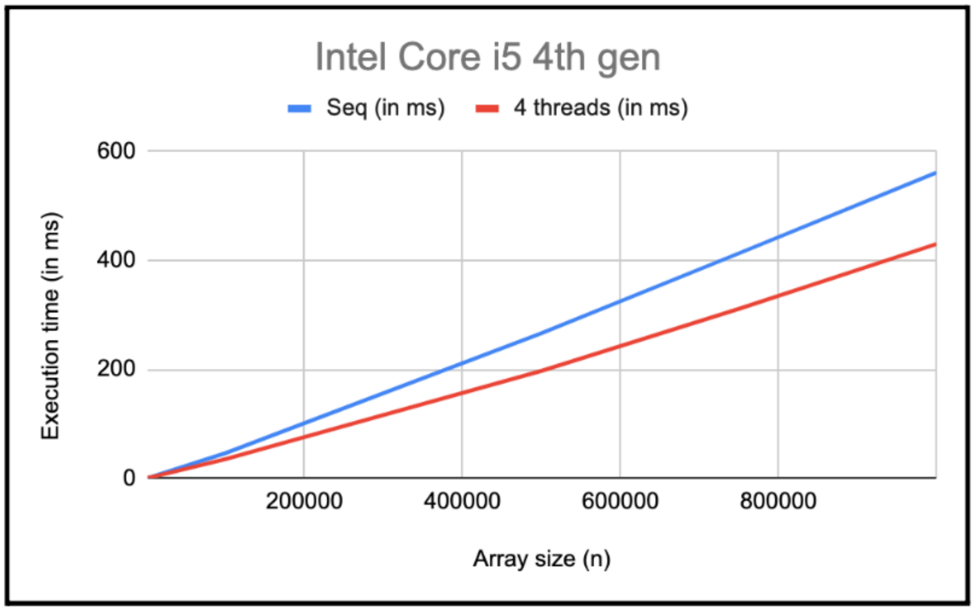


Figure 16: Results of Method-3 on i5 4th gen

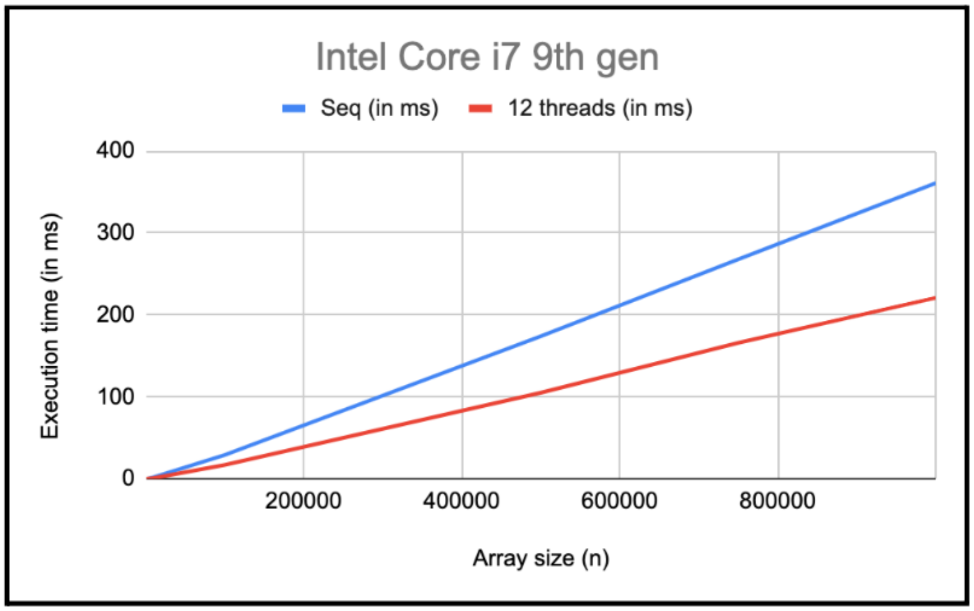


Figure 17: Results of Method-3 on i7 9th gen

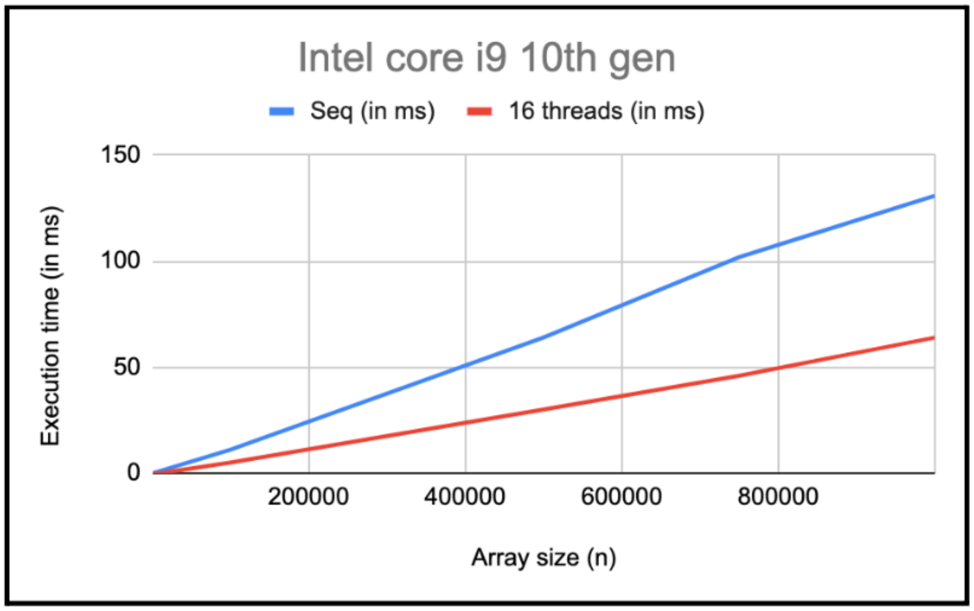


Figure 18 : Results of Method-3 on i9 10th gen

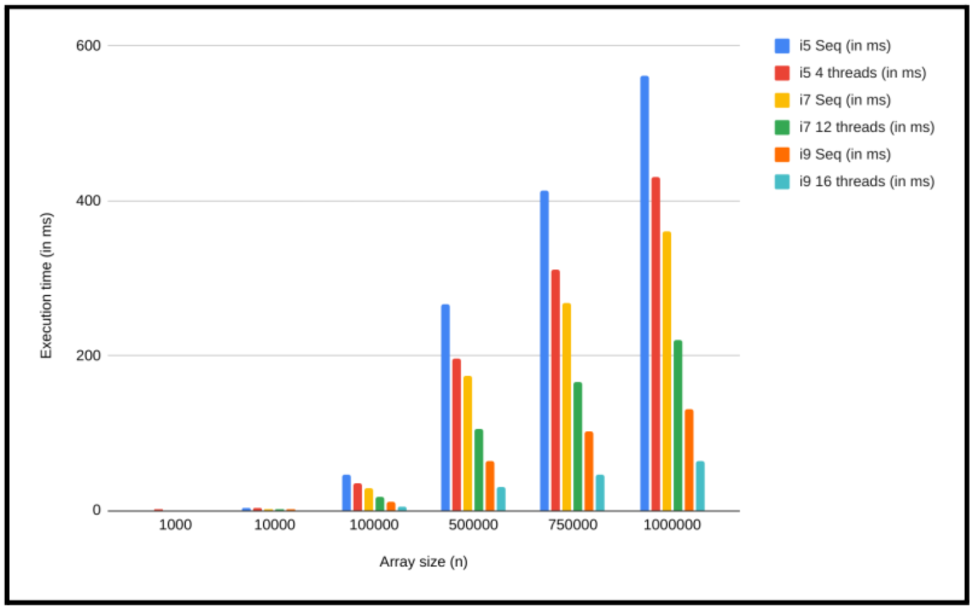


Figure 19 : Comparison of execution times of Sequential vs Parallel program across multiple CPU Architectures

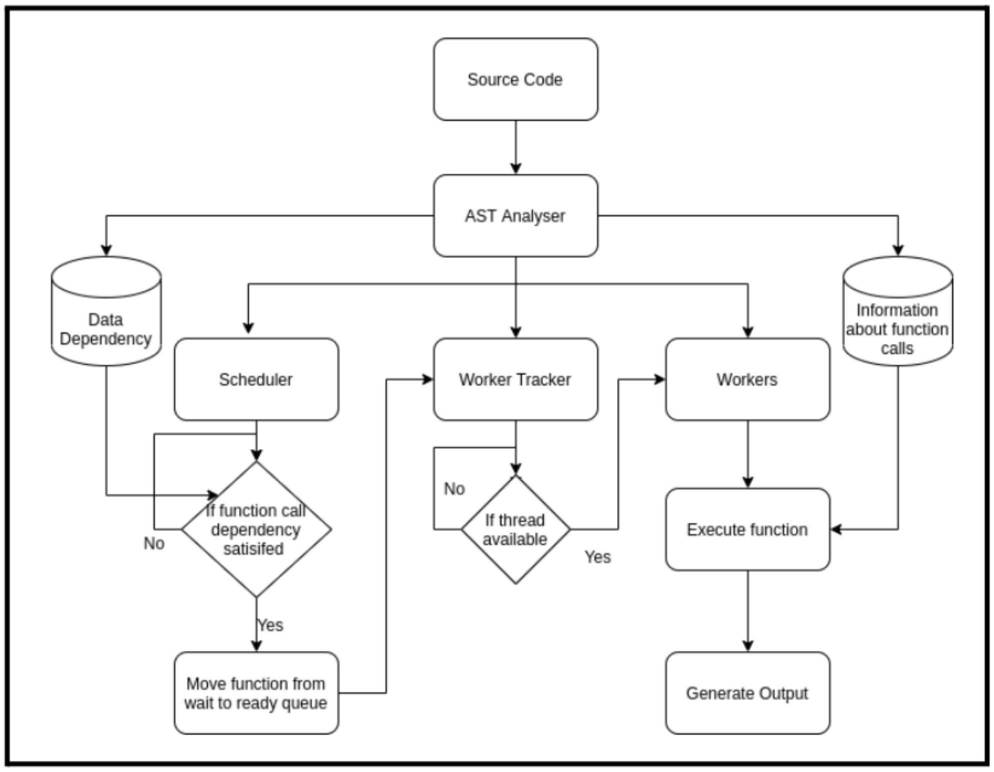


Figure 20: Method 3 implementation

6.1.3.2. Inferences from Method 3:

- This method provided a more fine grained control over thread scheduling allowing more flexibility and opportunity to maximize the parallelism that can be achieved.
- This method allowed us to go a level down in terms of the abstraction as we are dealing with threads directly in this approach.

- By controlling the scheduling of threads, the reordering of function calls could be done in a more accurate manner so as to gain the best possible speedup through parallelism.
- Through the use of semaphores and locks, we ensure there is synchronization among shared data

6.1.4. Method-4: Optimised Thread Scheduling for Functions using Non Master-Worker based approach to achieve Functional Parallelism

6.1.4.1 Implementation Details

- We begin with the generation of an Enriched Abstract Syntax Tree. This is different from a basic Abstract Syntax Tree as it provides ways to query the AST to obtain useful information and meaningful insights about the input source code. The generation of the Enriched Abstract Syntax Tree is done using a tool called Clava.
- Next, querying the generated AST is done. This is done by writing code in a language called LARA which is a language built on top of Javascript that is compatible with the Clava tool, thereby allowing us to query the AST.
- We then perform a read-write dependency analysis to get significant information about the input sequential source code.
- The information assimilated from the read-write dependency analysis performed above combined with the ability to query the enriched AST is used to populate text files that will be used by the next phase in our pipeline.
- The information about the functions that are modifying the arguments along with the information about the arguments being modified by these functions is written into one text file.
- The order of function calls along with information about the function name, the return type of the function, the arguments being passed to these functions, the parameters of the functions, the data types of both the arguments and the parameters are written into another text file.

- The information about the functions, their return types, the line number where the function is being called and the variable to which the returned value is being assigned (in case of a non - void return type) is written into another text file.
- The information about the occurrences of function calls and the line numbers before which the respective functions should finish execution is provided and written into a separate input file.
- The next phase in this approach is the thread scheduling algorithm.
- The text files populated by the previous phase are used by the thread scheduling algorithm.
- The information from these text files is assimilated and stored in appropriate data structures.
- Variables passed as arguments to functions and return value variables contribute as variable types whose dependency information needs to be collected and analyzed.
- Arguments to a function call: Variables passed as arguments to function calls from inside main are taken into consideration. The read/write dependency analysis on these arguments inside the function body is analyzed.
- If variables are passed by value, then the original variables in the main scope remain unchanged.
- If variables are passed by reference, then further checks are made to find out if variables passed as arguments are changed in any way inside the function body. If the variables are merely accessed, then it is known that they remain unchanged.
- Functions can return values of any type as well. These return values can be stored in a local variable inside the main scope.
- The next step is to find out the next point of usage of the above two types of variables. Since we perform inter-functional parallelism, care should be taken to ensure the variables have fully updated values, i.e., the functions should have completely finished execution. Thus, the function calls modifying any variables or returning any variables should be fully executed before the next point of usage of the aforementioned impacted variables.
- A point of usage of variables is referenced by the relative line number with respect to the beginning of the main body.
- Execution of functions in Method 4 is similar to how it works in Method-3. Functions are pushed into a thread pool, where it is executed on individual threads. In this method, we have eliminated the use of additional master threads to keep track of changing variables (referred to as special variables in Method 3). Instead, we use the additional information

gathered from extended data-dependency performed, giving us adequate information to execute functions before a specified point inside our program.

- In the previous method, we ensured the completion of all functions pushed into the thread pool by calling the "wait" or "get" method on all thread pool functions at the end of the program. However, in this present method, we preemptively call the "wait" or "get" method on the relevant function just before its designated point of execution (as determined and decided by the data dependency analysis). This way, all variables meant to be updated before their next point of usage are fully updated by a preemptive call to the corresponding function.
- To demonstrate, consider the following example:

Input Program:

```
transform_A(int *a, int n);
transform_B(int *b, int m);
transform_C(int *a, int n);
```

Output:

In the generated parallel program, following is the sequence of events:

```
// no special array required to keep track of modified arrays
// no ready and wait queues required to keep track of executing
functions
// definition of find_future function which ensures execution of the
mentioned function at that point of program
futures.emplace_back(transform_A, a, n)
futures.emplace_back(transform_B, b, m)
// transform_A needs to be complete before executing transform_C (which
uses array a)
find_future(transform_A, a, n)
futures.emplace_back(transform, a, n)
```

Analysis:

We see that pointer "a" is used again in transform_C after transform_A makes a modification on it. So when processing the data-dependencies, we keep a map, where we store that transform_A needs to complete before the execution of transform_C. This can be seen by the inclusion of the "find_future" function just before pushing the transform_C function into the thread pool.

By using this technique, we allow the threads to be able to decide the deadline for execution, hence eliminating the need for a master thread to monitor the threads and their execution.

- We use a map structure to store consecutive functions and their ordering local to the main body. This way, we have a reference to all the functions. This enables us to pick the right functions and preemptively finish its execution whenever needed.
- We handle code in the main function that are statements other than function calls. We use the data-dependency analysis to see what sections of these statements are dependent on any functions. If they are not dependent on any previous functions, then we align them for execution. If they do have a dependency, we add a barrier as deemed appropriate before this segment of code, to await completion of its dependencies.
- If a variable is being assigned the return value of a function, it is then dependent on that function. During our data-dependency analysis, we check for such instances, and in case of them we also deduce when the next usage of this variable happens at. We use this information, so that there is a barrier introduced before the next usage. This ensures that return values can be used in programming, while we continue to improve parallelization for function calls that return any type and not only void. The implementation of return type and return value handling is done using async programming concepts of promise and future.

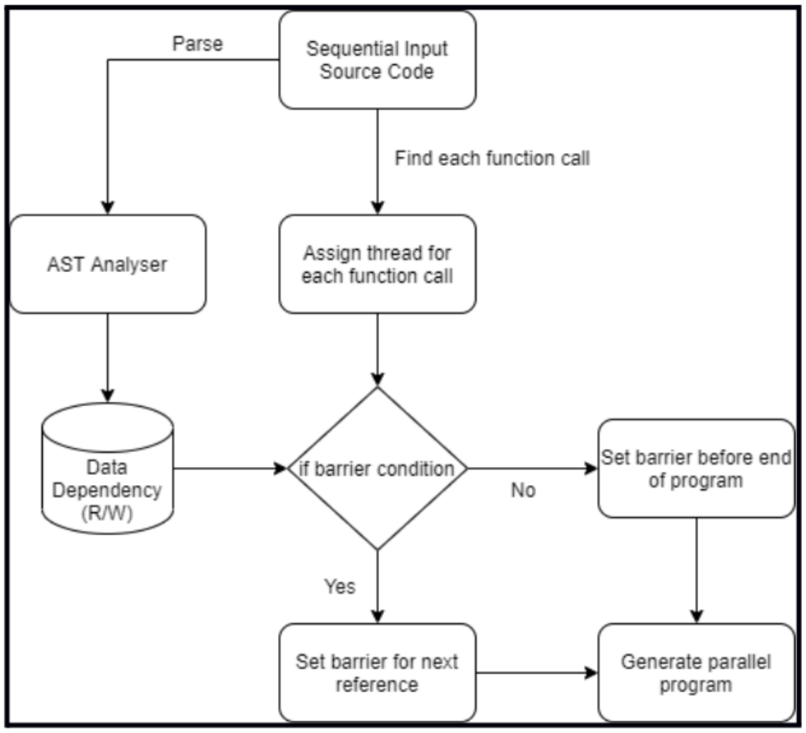

Sequential Input Source Code
Parse
Find each function call
AST Analyser
Assign thread for each function call
Data Dependency (R/W)
if barrier condition
No
Set barrier before end of program
Yes
Set barrier for next reference
Generate parallel program

Figure 21: Method 4 Implementation

- Consider the following example:

Input program:

```
int transform_A(int a)
{
    return a + 2;
}
void transform_B(int *a, int n)
{
    a[0] = 4;
    return;
}
int main(int argc, const char** argv)
{
    int res1;
    int *arr = {1,2,3,4,5,6};
    int b1 = 4;
    res1 = transform_A(b1);
    int res2 = 0;
    transform_B(arr, b1);
    for(int i = 0; i < res1; i++)
    {
     res2 += arr[i] + 2;
    }
    res1 = res2 + 2;
    return 0;
}
```

Output:

In the generated parallel program, following is the sequence of events:

```
// no special array required to keep track of modified arrays
// no ready and wait queues required to keep track of executing
functions
// definition of find_future function which ensures execution of the
mentioned function at that point of program
int main(int argc, const char** argv)
{
    int res1;
    int *arr = {1,2,3,4,5,6};
    int b1 = 4;
    futures.emplace_back(res1, transform_A, b1)
    int res2 = 0;
    futures.emplace_back(transform_B, arr, b1)
    for(int i = 0; i < res1; i++)
    {
     find_future(transform_B, arr, b1)
     res2 += arr[i] + 2;
    }
    res1 = res2 + 2;
    return 0;
}
```

Analysis:

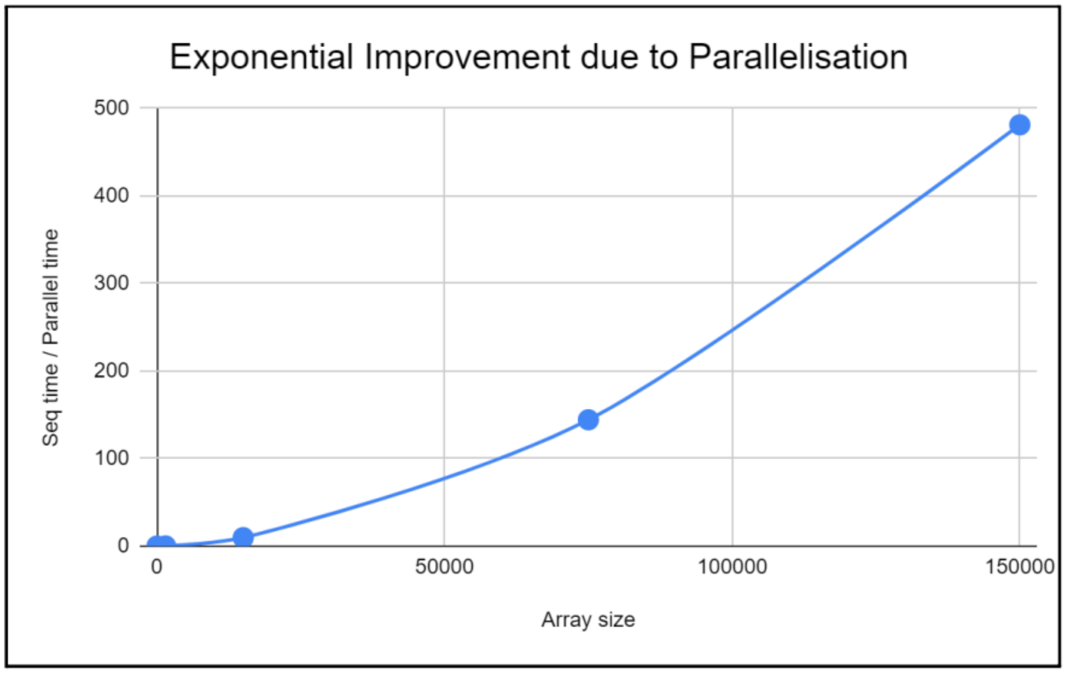
- The function transform_A makes a write operation on the variable passed, but it is a pass by value, so this has no implications on the argument passed. transform_B has passed by reference for the array, and it makes a write on this array. So for every call of tranform_B, we will need to set a barrier for the next usage of array passed. So in the main function, we set a barrier before the next usage of arr inside the loop, which is shown by the call to the find_future function. Note that, find_future called from within the loop will execute only once at the beginning of the loop if it identifies any redundant operations. This is handled in the internal implementation for find_future. For handling loops we check if all barriers set for it are cleared, if they are, we assign thread and execute the loops. In the above example, we notice that res1 used res2, so we set a barrier for that assignment, to wait for the loop to complete execution.
- We handle selection statements, like if-else and loops, in a similar fashion. For selection statements that have function call(s) within them, we apply the same function parallelization technique described before. This takes care of the selection statements, and even ensures that there are no race conditions as it is handled when the functions are being parallelized. Every other statement inside selection statements are taken care of as stated previously.

6.1.4.2. Results

Our main aim has been to improve resource utilization of underlying hardware to the maximum potential when executing a program, and thereby reduce the execution times. We have measured this by comparing the execution time of the sequential program and its parallel equivalents, generated by both the aforementioned Method 3 and Method 4. Although execution time of a program can be used to indicate speed-up, as our goal is also to improve the utilisation of available resources, we can not directly use only comparison of execution time between the original sequential source code and the generated parallel source code. In order to better understand the impact of resource utilisation, we have chosen to run the generation of parallel code and its execution on different hardware with varying numbers of cores and threads. We have specifically chosen an Intel i5 processor with 2 cores and an Intel i7 processor with 12 cores. These architectures are commonly available today to the average consumer and hence can be used as a reflection of the impact in the real world. This allows us to showcase that the parallel version of the program has increased the utilisation of the underlying available computational resource.

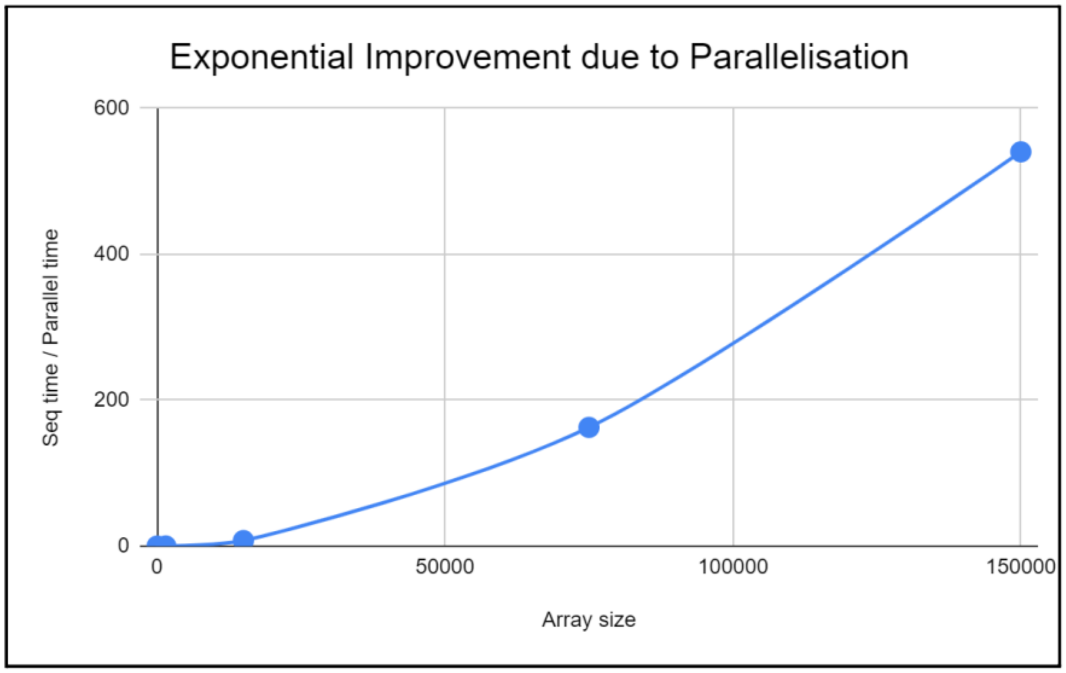
The Fig. 22 showcases the comparison of the sequential program with that of the parallel program generated by Method 4, Master Worker based approach, on two different architectures. The Y-axis is the ratio of execution time of sequential to parallel as we increase the number of computations carried out in a program. We have used ratios to represent the increase in this figure, as absolute comparisons are nearly impossible. The sequential program execution time increases non-linearly, while the increase in parallel time is seemingly more linear, Fig 24 shows for both methods. Fig 22 and Fig 23 can be compared to each other. They are the result of running on two different architectures. We notice that while in Fig 22 the speed up is 500 times almost, in Fig 23 it is 550 times. To generate Fig 23, we used an Intel i7 architecture with 12 cores as compared to the i5 architectures with 2 cores used to generate Fig 22. This can be used to infer that, with increased availability of hardware, we can improve the execution. This is possible as the underlying parallel program tries to maximise the resource utilisation, and on a machine with more cores and threads, it can achieve much higher parallelism.

The implementation of Method 4 is an evolution of some of the ideas established in Method 3 and additional design features. These changes streamline the resultant parallel code by reducing the number of mutex locks, increasing available threads as workers and removing all scheduling overheads. These changes along with the other described in previous sections justify the improvement in execution time. Fig 24 indicates the impact of these changes. Method 4 makes significant gains on Method 3 parallel code execution. As the number of computations are increased, Method 4 continues to perform much better as a result of more available threads and having no busy waiting for thread scheduling.



12

Figure 22: Ratio of Sequential execution times to Parallel execution times (Hardware setup : Core i5 - 2nd gen - 2 core machine)



12

Figure 23: Ratio of Sequential execution times to Parallel execution times (Hardware setup : Core i7 - 9th gen - 6 core machine)

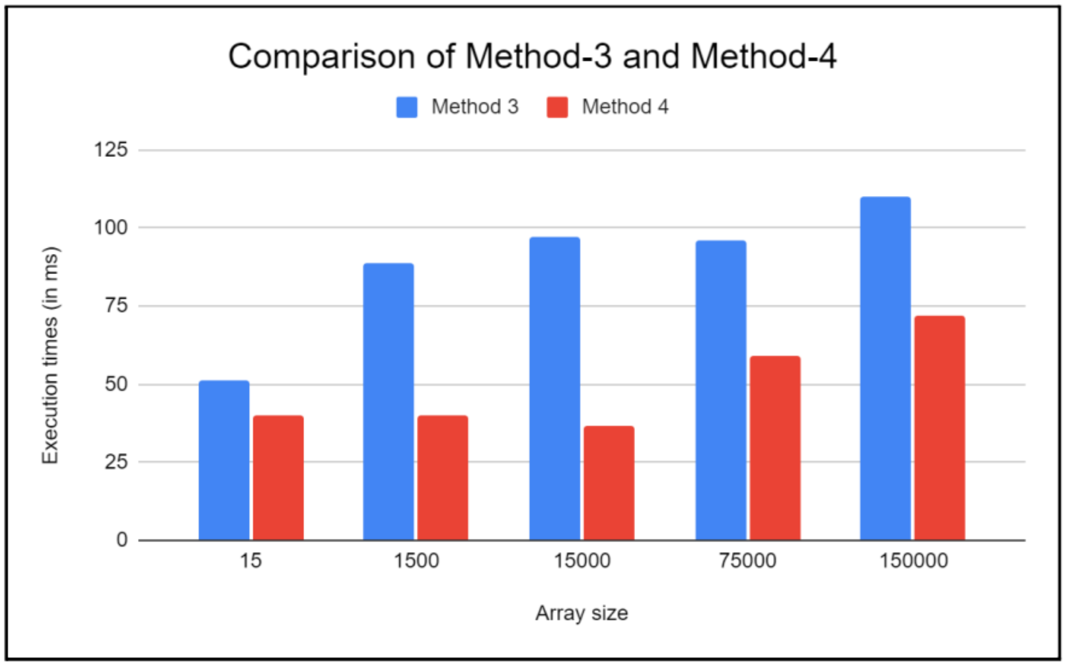


Figure 24: Comparison of execution times of Method-3 vs Method-4 (Hardware setup : Core i5 - 2nd gen - 2 core machine)

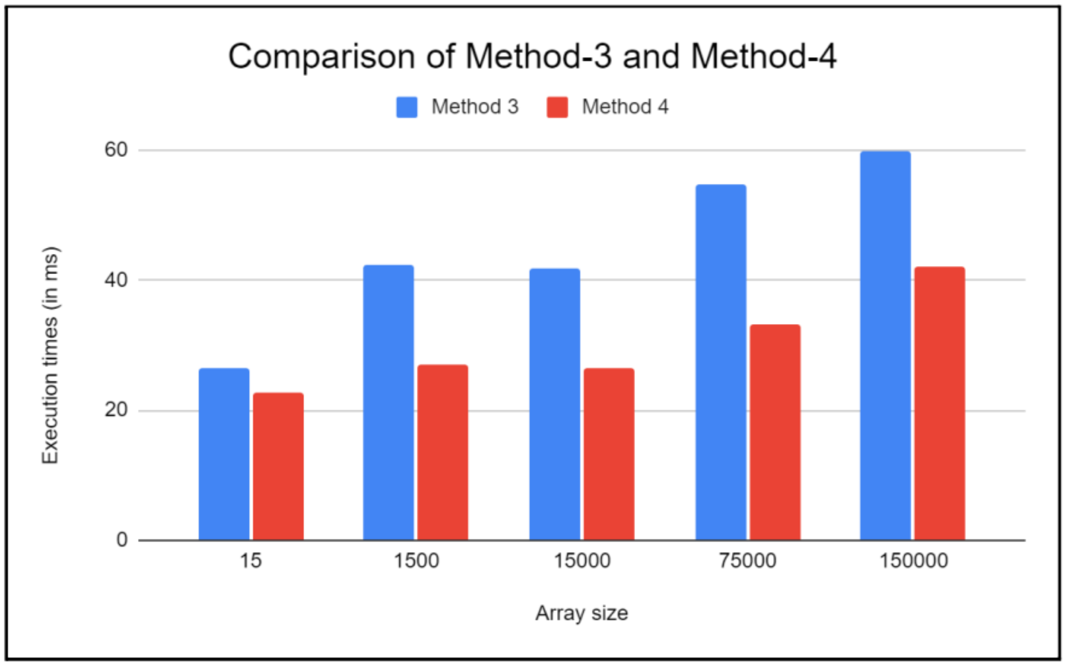


Figure 25: Comparison of execution times of Method-3 vs Method-4 (Hardware setup : Core i7 - 9th gen - 6 core machine)

6.1.4.3 Inferences of Method 4:

- Elimination of two additional master threads frees up available threads in the system, which can in turn be used for other tasks. This ensures more efficient usage of underlying hardware without any overheads.
- Simplified generated parallel code, with the elimination of if-else constructs, ready and wait queues, and a number of mutex locks required to keep certain data structures safe from race conditions.
- Due to the above two improvements mentioned, the generated program is now much shorter than the previous Method-3 approach.
- Thus, we have a significant improvement in the execution times of the newly generated parallel program when compared to that of the previous Method-3 approach.

6.2. Program Comprehension Phase

- The technique of program comprehension we employ enables us to identify the algorithm implemented in a specific function. If the program consists of multiple functions, then the requirement is to identify the algorithm tag for each of these functions.

  For example consider the following program:

```
void function_A(int* arr,const int arr_n)
{
    for(int i = 0; i < arr_n-1; ++i)
    {
        int min = i;
        for(int j = i+1;j < arr_n;++j)
        {
            if(arr[j] < arr[min])
            {
                min = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
}


int function_B(int* a, int n)
{
    int max = a[0];
    for (int i = 1; i < n; ++i)
    {
        if(a[i]>max)
        {
            max = a[i];
        }
    }

    return max;
}
```

```
int function_C(int* a, int n)
{
    int min = a[0];
    for (int i = 1; i < n; ++i)
    {
        if(a[i]<min)
        {
            min = a[i];
        }
    }
    return min;

}
```

- The program comprehension stage of our pipeline will map each of the above functions to the corresponding algorithm as identified. In this case, function_A will be mapped to "sort", function_B will be mapped to "max" and function_C will be mapped to "min".
- The entire program comprehension pipeline is represented in the below flow chart

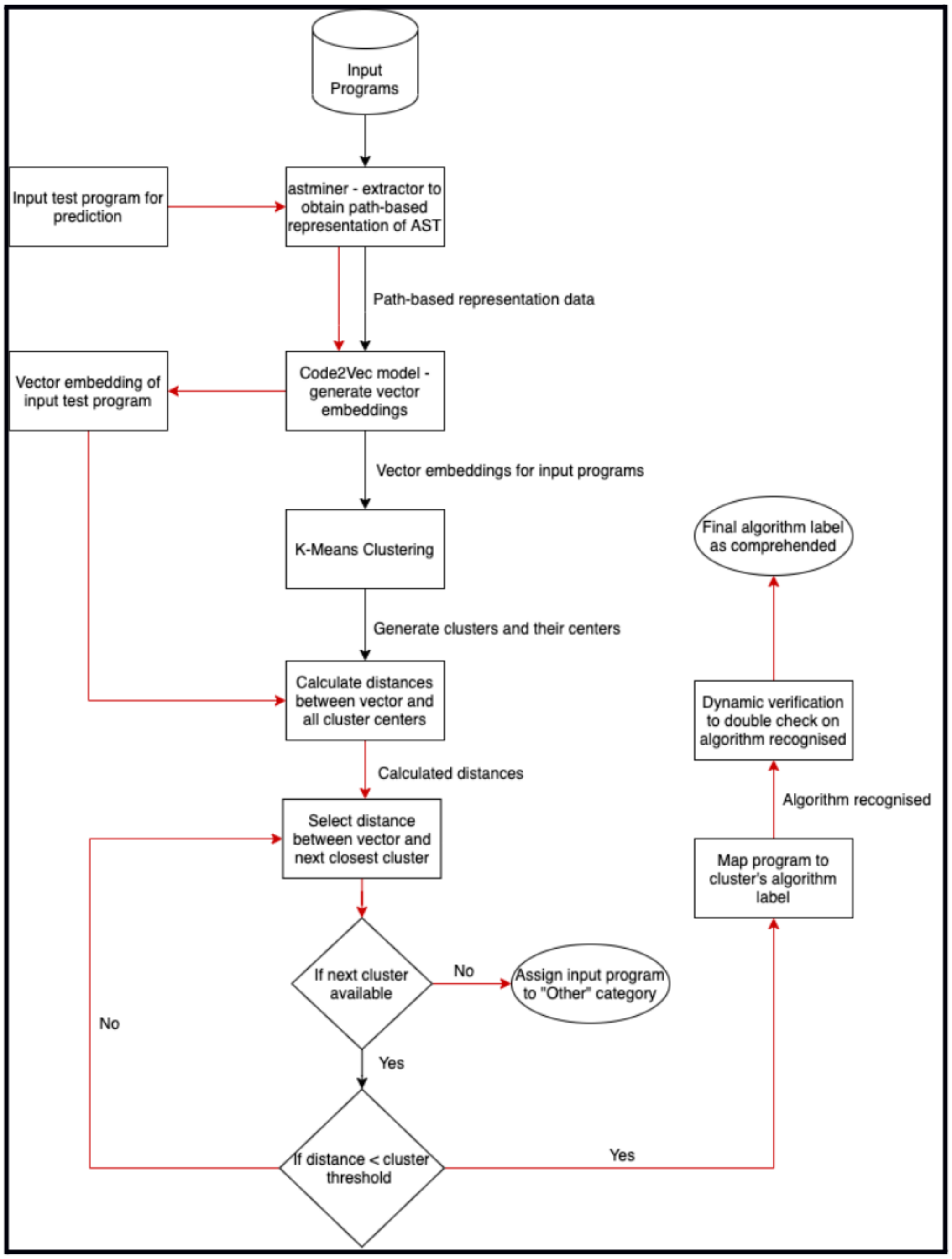

Input Programs
Input test program for prediction
astminer - extractor to obtain path-based representation of AST
Path-based representation data
Vector embedding of input test program
Code2Vec model - generate vector embeddings
Vector embeddings for input programs
K-Means Clustering
Final algorithm label as comprehended
Generate clusters and their centers
Calculate distances between vector and all cluster centers
Dynamic verification to double check on algorithm recognised
Calculated distances
Algorithm recognised
Select distance between vector and next closest cluster
Map program to cluster's algorithm label
No
If next cluster available
No
Assign input program to "Other" category
Yes
If distance < cluster threshold
Yes

Figure 26: Program Comprehension Implementation

- Our technique of program comprehension is to represent the input source code as vector embeddings, find similarities between these embeddings to group them into similar clusters, and additionally verify if the predicted label is accurate using a dynamic verification process.
- To represent the programs in the form of vectors, we employ a technique of embedding. It is a three-step process:
    - Use an extractor specifically designed for C/C++ programs to extract an Abstract Syntax Tree of the program and break it down into smaller sections based on nodes of the tree.
    - The nodes are then combined together to represent bigger sections, like a function definition or a selection construct body, to create so-called path-based context vectors.
    - These path based context vectors are then used to train a neural attention model to better represent individual vectors for each of the smaller sections, and then we concatenate these vectors based on weighted attention as computed by the model to finally form the single vector representation for the function or the program.
- The vectors thus obtained give a very good representation of each of the functions in the input program. Similar functions designed to implement the same algorithm would have very similar vector embeddings as well. To calculate the similarity between the vector embeddings, a simple technique such as cosine similarity can be employed.
- Based on these calculated similarity scores, it is now possible to group together similar functions. This technique is similar to the initial stages of the K-Means clustering algorithm. However, we do not use the concept of nearest cluster centroid to allot the cluster label to our functions.
- We have developed a means to calculate a threshold value for each of the clusters, which helps us in determining if a function belongs to the cluster or not. If the computed distance between the cluster centroid and the function is greater than the pre-computed threshold, then we skip the present cluster and perform the same comparison on the next closest cluster, since each cluster could essentially have different pre-computed threshold values. If a function doesn’t belong to any of the clusters based on the said comparison and allotment technique, then the function is classified into an “Others” category.

- There is always a degree of uncertainty in the probabilistic design of the models we employ. The "Others" category is an important aspect of our Program Comprehension pipeline. It is important not to misclassify any of the functions, to ensure correctness and retain semantics of the original input source code. For instance, if we classify a function sorting an array, as a function reversing an array, this wrong label will affect the consecutive steps in our complete pipeline of parallelisation, and generate a semantically wrong parallel program in the end. However, it is still acceptable, if the same function sorting an array is classified into the "Others" category. In such a case, we will not be performing intra-function parallelisation on this particular function, however, it is safe from the wrong substitution of a parallel version with a different algorithm.
- Thus it is important to set strict threshold values of every cluster to ensure that we retain the semantics of the input sequential program.
- To further verify the allotted label through the clustering technique as mentioned, we make use of a dynamic verification process. Based on the identified algorithm label, we perform dynamic analysis on that function, verifying the output of that function for a predefined set of inputs.
- For example, if the clustering technique returns that the label for a function is sort, we can send in an unsorted array and verify if that function returns back a sorted array finally. This way, we double check on the identified label and further improve the accuracy of our complete program comprehension pipeline.
- Thus, through the use of multiple verification steps, we can be certain that this implementation returns the correct label for each of the functions in the input program, either the correctly identified specific label meant for an algorithm, or the "Others" label if there was any discrepancy in the prediction.

6.3. Additional Steps

- To ensure better parallelisation decisions and ensure correctness, we can make additional tweaks to our pipeline. Some of them are as follow:
    - If the execution times of the generated parallel program have increased, compared to the execution times of the input sequential program, due to context switching overheads, then we can skip parallelisation.

- If the outputs of the parallel and sequential programs are not identical, then this implies correctness of the original program has not been maintained. In such scenarios, we can skip parallelisation.

7. Conclusion

- Proposed Pipeline has been implemented in its entirety
- Assumptions made in Phase 1 have been eliminated

7.1. Parallelization:

- In the parallelizing phase, we have tried out 4 different approaches, namely:
  - Inter and Intra-Function Parallelism by AST Querying and replacement with OpenMP Directives
  - Naive Thread Scheduling using C++ concepts of Promises and Futures.
  - Optimised Thread Scheduling for Functions using Master-Worker based approach to achieve Functional Parallelism
  - Masterless Thread scheduling for functional parallelism
- With each approach we have tried to improve the parallel code being generated with respect to the different cases it can handle and with respect to the performance improvement that can be achieved.
- We have made improvements to Method 3 and arrived at a new approach, Method 4.
- Method 4 removes the Ready queue and Wait queue used in Method 3, thereby nullifying the requirement for the track and schedule Master threads.
- Method 4 handles a wider variety of cases , functions which return a value, selection and iterative statements.
- Method 4 simplifies the generated parallel code as compared to previous methods.
- With each approach, we have been moving down a level with respect to the level of abstraction that we are dealing with, thereby gaining a fine grained control over the different aspects concerning parallelization.

7.2. Program Comprehension:

- We have implemented the Program Comprehension Phase of our pipeline and integrated it with the Parallelization Phase.
- The input source code is represented as Vector Embeddings.

- The similarities between the vector embeddings of different programs is found to group them into clusters.
- Each of the clusters correspond to a previously defined category that has a parallel version in the backend database.
- It is tried to associate the new test program into one such cluster by using the appropriate thresholds.
- Dynamic Verification is used to additionally verify if the predicted label is accurate.
- “Others” category is introduced to ensure correctness of the program by avoiding any misclassification.

8. Future Work

- Refine and refactor the implemented code to gain possible improvements.
- Make the current implementations more efficient to reduce pre-processing time.
- Current implementation is a generalized technique, a possible area of future work would be to allow for domain specific optimisations as required.
- Explore possible improvements to increase the speedup in performance of the generated parallel code.
- Extend support to more cases wrt Program Comprehension by training on larger datasets and defining more categories with a parallel mapping.
- Experiment with any new improvements in Program Comprehension techniques.

Report

ORIGINALITY REPORT

4%	4%	2%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	anorien.warwick.ac.uk Internet Source	1%
2	congdongitchiase.blogspot.com Internet Source	<1%
3	researchr.org Internet Source	<1%
4	Submitted to Higher Education Commission Pakistan Student Paper	<1%
5	Sean Rul, Hans Vandierendonck, Koen De Bosschere. "Function level parallelism driven by data dependencies", ACM SIGARCH Computer Architecture News, 2007 Publication	<1%
6	link.springer.com Internet Source	<1%
7	www.cse.iitk.ac.in Internet Source	<1%
8	Submitted to The Robert Gordon University Student Paper	<1%

9	www.science.smith.edu Internet Source	<1 %
10	citeseerx.ist.psu.edu Internet Source	<1 %
11	hal.inria.fr Internet Source	<1 %
12	Yanyan Dai, Xiangli Zhang. "A Synthesized Heuristic Task Scheduling Algorithm", The Scientific World Journal, 2014 Publication	<1 %
13	Qichang Chen, Liqiang Wang, Ping Guo, He Huang. "chapter 16 Analyzing Concurrent Programs Title for Potential Programming Errors", IGI Global, 2011 Publication	<1 %
14	ftp.math.utah.edu Internet Source	<1 %
15	Lecture Notes in Computer Science, 2012. Publication	<1 %
16	www.cs.qub.ac.uk Internet Source	<1 %
17	B. Di Martino. "Two program comprehension tools for automatic parallelization", IEEE Concurrency, 2000 Publication	<1 %

18	L.C. Briand, S. Morasca, V.R. Basili. "Property-based software engineering measurement", IEEE Transactions on Software Engineering, 1996 Publication	<1%
19	Submitted to The University of Manchester Student Paper	<1%
20	B. Di Martino, C.W. Kessler. "Two program comprehension tools for automatic parallelization", IEEE Concurrency, 2000 Publication	<1%
21	Beniamino Di Martino, Antonio Esposito. "Automatic Dynamic Data Structures Recognition to Support the Migration of Applications to the Cloud", International Journal of Grid and High Performance Computing, 2015 Publication	<1%
22	Serguei Diaz Baskakov, Juan Gutierrez Cardenas. "Source to source compiler for the automatic parallelization of JavaScript code", 2021 IEEE XXVIII International Conference on Electronics, Electrical Engineering and Computing (INTERCON), 2021 Publication	<1%
23	Hamid Arabnejad, João Bispo, Jorge G. Barbosa, João M.P. Cardoso. "AutoPar-Clava",	<1%

	Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM '18, 2018 Publication	
24	diendan.congdongcviet.com Internet Source	<1 %
25	ithutech.com Internet Source	<1 %
26	tutorialspoint.dev Internet Source	<1 %
27	www.ijcaonline.org Internet Source	<1 %
28	"Applied Parallel Computing. New Paradigms for HPC in Industry and Academia", Springer Science and Business Media LLC, 2001 Publication	<1 %
29	Lekshmi S Nair. "An Analytical study of Performance towards Task-level Parallelism on Many-core systems using Java API", 2021 6th International Conference on Communication and Electronics Systems (ICCES), 2021 Publication	<1 %

30	export.arxiv.org Internet Source	<1 %
31	www.coursehero.com Internet Source	<1 %
32	www.grid.unina.it Internet Source	<1 %
33	"Information Security and Cryptology", Springer Science and Business Media LLC, 2011 Publication	<1 %
34	"Using and Improving OpenMP for Devices, Tasks, and More", Springer Science and Business Media LLC, 2014 Publication	<1 %

Exclude quotes	On	Exclude matches	< 5 words
Exclude bibliography	On		

Automatic Parallelization of Source Code using Thread Scheduling

Darshan D
Computer Science
PES University
Bangalore, India
darshand2000@gmail.com

Karan Kumar G
Computer Science
PES University
Bangalore, India
karan292000@gmail.com

Manu M Bhat
Computer Science
PES University
Bangalore, India
manumbhat09@gmail.com

Mayur P L
Computer Science
PES University
Bangalore, India
mayurpeshve2@gmail.com

N S Kumar
Computer Science
PES University
Bangalore, India
kumaradhara@gmail.com

***Abstract*—Modern computers have improved in hardware by leaps and bounds in recent times. The improvements have been in the availability of multiple cores, with better thread handling. Despite these advances, most programs are written as sequential programs and hence use only a single thread. These architectures can be exploited better by running these programs on multiple threads. This would require parallelization of the program, to the maximum extent possible. The transformation to parallel program poses many challenging tasks both for newer parallel program development and usage of already existing legacy program modules. The parallel program must handle race conditions, deadlocks, infinite wait problem resolution, and data dependencies. These require skilled programmers and is cumbersome for development. Another time and resource-consuming part of the development of these programs would be the debugging and testing needed for ensuring the correctness of the parallel program. These costs negate the benefits for an enterprise to manually convert software into parallel programs. Therefore auto-parallelization is a feasible and affordable solution, that will help improve the utilization of the available hardware.**

Currently available auto-parallelization tools focus on specific techniques such as loop parallelization or are developed as domain-specific solutions. We propose two techniques in an attempt to generalize auto-parallelization and handle a variety of cases. Our design focuses on task parallelism. We generate parallel source code that executes functions in parallel on different threads, i.e., inter-functional parallelism. The generated code is made available to the user to execute or if needed modify to their requirements.

***Index Terms*—auto-parallelization, parallel programming, thread scheduling, data-dependency analysis**

I. Introduction

Since the introduction of computers, hardware and the associated software has improved in many ways. Currently, multi-core multi-threaded architectures are commonly available to the average consumer at a reasonable cost. Unfortunately, the exploitation of these computational resources is limited since most programs run only on a single thread. There are considerable efforts being spent on programs to be converted into parallel programs. However, the cost involved in developing software that runs as parallel program are typically high. Legacy software was written for much older machines with constraints on resources, and its complete conversion to a parallelized version leads to a completely new development effort in itself. Note here that it requires highly skilled software developers who possess knowledge of parallel programming environments. They will need to design and program the software for the required parallelization. This design is invariably more complex compared to the design of a sequential program. The cost and time required for this development are not justified for enterprises. Alternatively, both full or partial automation of the process of parallelization to improve the performance of their software would be feasible for the enterprises hence justifying the cost and time required for it.

The above-stated reason has made automatic parallelization of source code an important area of research. There have been various different approaches for carrying out the transformation of programs to their parallel equivalent. Techniques used for this vary from loop parallelization, program comprehension, and even machine learning in recent times. Our approach focuses on functional parallelism, where we run functions in parallel on different threads. Data-dependency analysis on the original source code allows us to schedule these functions such that the correctness of the program is maintained. This effectively is an implementation of the philosophy of task parallelism.

The main contributions of this paper are:

- Auto-parallelization using the approach of executing function in parallel
- Code generation of resultant parallelized code, for user to execute and modify as required.
- Thread scheduling and handling technique optimised for our design of functional parallelism. We have opted to write our own thread handling instead of using parallel programming library directives. The details of this are

provided in Section III-B
- Issues generally associated with parallel programs such as race conditions, infinite wait problem, load balancing and busy waiting taken care in the generated program.

II. Related Work

The most popular technique for auto-parallelization is the usage of loop parallelization. In this technique, tools apply loop optimizations such as loop tiling, unrolling, and running loop iterations in parallel. Tools like Pluto [1] use this technique. The implementation of this uses a mathematical modeling concept called polyhedral analysis. Similarly, ParaWise [2] too is an auto-parallelization tool. ParaWise allows users to tune the parallelization with hyperparameters to cater to any domain-specific needs of the user. It generates a parallelized code based on the given hyper-parameter values.

Another interesting approach towards parallelization is the use of program comprehension. One of the earliest works on this technique was done by Martino et al [3] in 1994. In the original paper, the authors theorize a tool known as PAP(Parallelizable Algorithmic Patterns). The identified algorithms are replaced from a database containing parallel equivalent algorithms This is was later implemented as a proof of concept in the Vienna FORTRAN Compiler by the authors [4]. Another closely related work, was the development of a tool known as PARAMAT [5]. PARAMAT too uses program comprehension techniques, where it assigns ”concepts” to sub-tree of the program AST (Abstract Syntax Tree). It replaces these sub-trees with equivalent parallel code.

In recent times with the advances in machine learning, there have been attempts to use machine learning techniques for parallelizing source code as well. Peter Kraft et al [6] use neural networks and decision trees to predict the possible future state of a program given the current state of the registers and other necessary data. There are multiple predictions made about a specific future point in the program. These predictions are assigned to workers, which continue execution from the predicted point using the predicted state. When the execution of the original program reaches the point where the prediction was made, they check which of the worker worked on the correct predicted state, and jump into the last executed state of the worker. This allows for parallelization of execution.

There have been works in the area of functional parallelism as well, such as Sean Rul et al [8]. In this work, the authors explore functional parallelism where they cluster functions and execute clusters in parallel. Our exploration differs from their work on the basis of a different approach, we use a technique with master-worker architecture and another technique with barriers for execution, as discussed in detail in section III. However, this work lays the foundation for the idea of using data-dependency analysis to extract execution details about the function calls.

III. Method

In this section we present the two different techniques for auto-parallelization. They are namely, master-worker architecture based parallelisation and a worker-only architecture based parallelization technique. The preprocessing stage for both these techniques is the data-dependency analysis. The data-dependency analysis stage extracts crucial information about the given source code. This information is used by the later stages for carrying out the generation of parallelized program.

A. Data Dependency Analysis

For applying any non-cosmetic transformations to a source code, we must ensure the transformation does not affect the output of the program in any case. We have used data-dependency analysis for this purpose. The information obtained through data-dependency analysis is used in later stages to ensure correctness while transforming the program to a parallel program. To ensure correctness of program is maintained in the generated code, there are two important aspects that need to be considered. Before applying an operation on any data in the program, we must ensure that the data in question is up to date and all previous operations that needed to be carried out on it have been completed. Also, the result of any operation on the data present must have the intended effect on the data, such that operation downstream in the execution receive the output/modified data correctly.

To carry out data-dependency analysis we need to abstract out certain aspects of the programmer’s influence, like their style, etc. Also, it would be easier to extract information if the program were present as a data structure that can be queried easily. An Enriched Abstract Syntax Tree (AST) suits our needs for this purpose. We use the CLAVA/LARA tool [9] to generate the AST. LARA is a JavaScript-based language that allows us to query on the generated AST in an efficient manner.

The AST is used to carry out the following analysis:

- We query the AST for every function call in the ”main”(driver) function. The query returns the function being called and the parameters passed. We store this in an ordered data-structure. The order of storing is maintained to be the same as the order of function calls in the original source code. The data-structure is used later on, to run data-dependency analysis on these functions. This allows us to determine the implications the function can have on the data passed as arguments. Any built-in functions are not stored in this data-structure, as there is no need for carrying out data-dependency analysis on them. Example 1

```
int main(){
    int a = 10;
    int b = 20;
    transform_A();
    transform_B(&a);
    printf("%d %d", a, b);
    transform_C(a, b);
}
```

In the above example upon querying we obtain the function calls made. The data structure to store function

calls is populated with transform_A(), transform_B(&a), and transform_C(a,b), such that their relative order is maintained as in the given source code. Additionally, we avoid function printf() because data-dependency implications of built-in functions are already known.
When populating the data structure, we also store information on how the parameter is passed, i.e passed by reference or passed by value. This is relevant for the further data-dependency analysis we carry out on the function body.

- For running the data-dependency analysis on the function calls, the data-structure to store these functions are used extensively. Using this data-structure, we keep an account of all parameters passed by value. For these parameters, all write operations that are a precursor before the value is passed to the function call must be completed. Similarly for parameters passed by reference, all write operations prior to being passed must be completed. Additionally, for parameters passed be reference, we track any write operations made within the function. These write operation have implications outside the function body as well. For this reason, they must be tracked and these then form the write operations that must be satisfied before any other operation is carried out on the same data. All global variables being used within the function body are also kept track of, for both reads and writes. Effectively, global variables can be considered to be equivalent to passed by reference parameters.
  Consider the following example, Example 2

```
int c = 30;  // global variable
void transform(int *a, int b){
    // write operation on a
    // read on b and c
    a[b - 2] = c;
}
int main(){
    int a[] = {1,2,3,4,5};
    int b = 3;
    transform(int *a, int b);
    return 0;
}
```

  In the above example, the function "transform", there is a read operation on variable 'b' and a write operation on variable 'a'. As 'a' is passed by reference(pointer in the case of C language) the write operation has implications outside the function as well. Therefore, this operation is noted down and used as a barrier later on, to ensure correctness. We define barrier with more detail in subsection III-D.
- Handling return values of functions is a crucial part of generalising the parallelization. If a function has a return value, we query the AST to find the next read or write operation on the variable where the return value is stored. We store this information in another appropriate data-structure indicating the function call, the corresponding return variable and its next usage.
  Example 3:

```
int transform(int a){
    return a + 1;
}
int main(){
    int b = transform(10);
    // statements to be executed
    // that do not use b

    // first usage of b
    // after writing return val
    printf("%d\n", b % 2);
    // statements to be executed
    return 0;
}
```

  In the code segment of Example 3, printf() uses the return variable 'b'. Therefore, the function call - transform(10), the return variable 'b' and the next usage in printf are all stored in a data-structure for later usage in the parallelization phase.

B. Thread Handling

Our implementation used parallel programming directives in its infancy. But the usage of these directives restricted certain optimisations that were introduced. Also by designing our own scheduler and handling thread operations natively allowed us to optimise the implementation for the same. This also allowed us the freedom to only have the necessary scheduling and thread handling overloads, which reduced any other redundant overloads. The freedom offered through native thread handling was the finer control over execution. This control has been exploited in both our techniques, and also allowed us to ensure correctness in a robust manner.

C. Master-worker based auto parallelization Technique

The master-worker based parallelisation technique, as the name suggests uses master threads and workers. The masters are in charge of scheduling functions when they are ready to be executed and to maintain any relevant information about the execution, such as thread pool management. The worker threads execute the function assigned to them by the master. An overview of the program execution is presented in Fig. 1. The data-dependency analysis results are stored in the database "Data Dependency" and the details about function calls and other implementations of functions and helpers made in the original source code are stored in the database marked as "Information about function calls".

This design has two master threads, the *Scheduler* and the *Worker Tracker*. The Scheduler is in-charge of scheduling functions that are ready for execution. The Worker Tracker is in charge of managing the thread pool. When a function is scheduled for execution, the Worker Tracker finds a free thread and sets the function for execution on this thread. If no threads are free, then the Worker Tracker waits for a thread to be freed

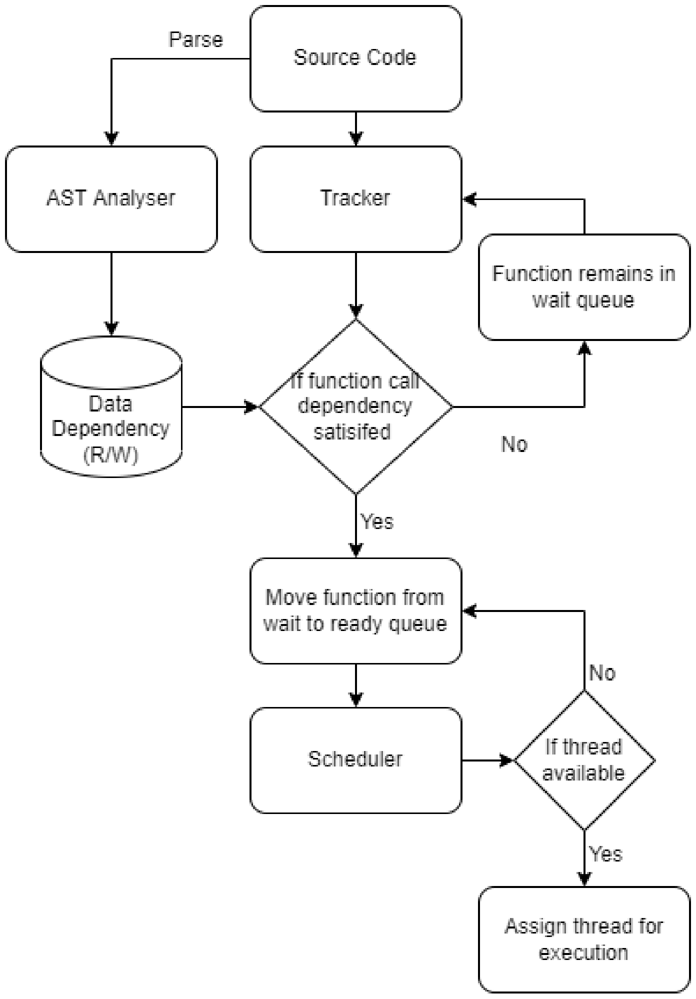

Parse
Source Code
AST Analyser
Tracker
Function remains in wait queue
Data Dependency (R/W)
If function call dependency satisifed
No
Yes
Move function from wait to ready queue
No
Scheduler
If thread available
Yes
Assign thread for execution

Fig. 1. Technique 1 Flowchart.

and assigns the function to thread at the earliest possible. There are two queue data-structures used by the master threads to keep track of functions' status. The wait queue holds functions that have dependencies yet to be satisfied and hence are not to be executed yet. The ready queue has functions for which all necessary dependencies have been satisfied, and hence can be executed. The Scheduler thread is responsible for moving the functions to wait queue and later to the ready queue, once the dependencies are satisfied. The wait queue is responsible for assigning functions in ready queue onto a free thread.

By executing functions only when dependencies are satisfied, we ensure that no function is passed any data that has not undergone the operations that need to be completed. This is key in ensuring the correctness of the program in its execution.

Consider the following example: Example 4:

```
int main(){
    //some code
    //transform_A makes a write
    //on parameter a
    transform_A(int *a, int n);
    transform_B(int *b, int m);
    transform_C(int *a);
    return 0;
}
```

In Example 4, assume the code before transform_A is called has no effect on the subsequent code. The function transform_A is making a write operation on pointer a. As there is no dependency that transform_A needs to wait for, it can be added to the ready queue. Since the function transform_B carries out its operation on pointer b, and changes made by function transform_A have no implications on it, we can add transform_B to the ready queue as well. However, the function transform_C uses the pointer a. So it needs to wait for the write operation of function transform_A to be completed. So we add transform_C to the wait queue, as it needs to wait for its dependency to be satisfied. The wait queue is intended for functions that are dependent on a previously called function(s). The functions in this queue need to wait until their dependency functions finish their execution. Once the dependency functions finish their execution, the functions waiting in the wait queue are moved into the ready queue, implying they are ready to be executed. Functions inside the ready queue are assigned separate threads for execution. Once assigned, such functions are dequeued from the ready queue.

Common data structures such as ready queue, wait queue, other lists used to maintain dependent functions, and corresponding arguments are used by the Scheduler. In order to maintain synchronization among these data structures, we use mutex locks. This prevents possible race conditions. For the purpose of generalization of our scheduling algorithm to handle all kinds of client code, we make use of a program generating technique, which produces the parallelized version of the client code, with appropriate thread allocation and mutex locks in place.

1) Implementation Level Optimisations: Since we continue to use threads for execution of different functions, it is useful to not have to allocate and deallocate them repeatedly.The process of allocation and deallocation of threads is a costly operation. To reduce this overhead of allocation and deallocation, we initialise a thread pool at the beginning of the execution of the program. These threads then remain allocated until the end of execution, after which the requirement is no longer present and the threads are then deallocated. The concept of promise and future from asynchronous programming is used, to obtain the return value of functions that are assigned to threads for execution. This allows for return value assignment to appropriate variables and correct usage of the same.

D. Non-Master-worker based auto parallelisation Technique

It can be observed in the previous technique that if a function is operating on any data. The latest it can complete operation before it blocks the parallel execution of the source code is when another operation or function needs to wait for its completion. This occurs when the later function/operation uses the data being read/written by the function being executed. Therefore we can assume this later function/operation is a deadline for the execution of the current function. In the data-dependency analysis, we have found the next usage for every parameter and return value. We use the aforementioned concept and the information from data-dependency analysis to allow for threads to self schedule and also self assign function to themselves for execution. This makes the two master threads redundant. Hence freeing two more threads to be made available in the thread pool. Also, this reduces the scheduling and thread management overheads. Example 5: For example, consider the same code as used in Technique 1

```
int main(){
    //some code
    //transform_A makes a write
    //on parameter a
    transform_A(int *a, int n);
    transform_B(int *b, int m);
    transform_C(int *a);
    return 0;
}
```

We see that pointer ”a” is used again in transform_C after transform_A makes a modification on it. So when processing the data dependencies, we keep a map, where we store that transform_A needs to complete before the execution of transform_C. By using this map, we allow the threads to be able to decide the deadline for execution, hence eliminating the need for a master thread to monitor the threads and their execution.

We describe *Barrier Condition* as a check of whether the return value of the function call is used at a later point in the program or if any of the arguments passed to the function call are modified inside the function and used at a later point. The outline of the Non-Master-worker-based auto parallelization Technique is shown in Fig. 2. Every function is assigned to a thread from the thread pool for execution. The thread first checks the barrier condition of this function.

- If the condition returns true, for all parameters that are modified and the return value, we go to the next usage of these variables and set a barrier just before the usage.
- If the condition returns false, we set the barrier for completion of execution to be just before the end of main. This case occurs if and only if no other piece of code after this function execution is dependent on the parameters and return value.

The function then continues its execution on the assigned thread. A barrier is eliminated only if the dependency that sets the barrier is satisfied. So for executing any function, all barriers set before it need to be eliminated. Therefore barrier condition check allows us to ensure the correctness of program because no function executes before its dependencies are satisfied.

By using the barrier condition, we not only make the master threads redundant and remove the overheads associated with scheduling and thread management. There is also no longer the requirement for ”ready” and ”wait” queues. By removing these data-structures, we also reduce busy waiting. As there

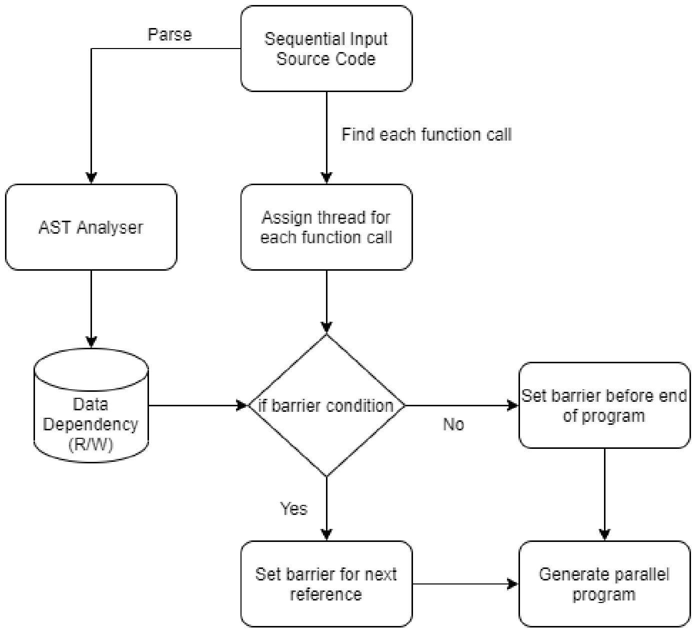

Sequential Input Source Code
Parse
Find each function call
AST Analyser
Assign thread for each function call
Data Dependency (R/W)
if barrier condition
No
Set barrier before end of program
Yes
Set barrier for next reference
Generate parallel program

Fig. 2. Technique 2 Flowchart.

are less mutex locks needed to modify these queues when executing. To describe the reduction in waiting quantitatively, there was a reduction from 8 mutex locks to 1 mutex lock in our implementation. All these changes also make the generated code much simpler and shorter than the previous technique. The changes are not just cosmetic, there is also an improvement in the execution speed as shown in section IV.

E. Handling non-function call in main

The ”main”(driver function will contain statements other than function calls as well. These statements can be called as *non-function call statements*. such statements need to be taken care of, ensuring they are executed correctly and all data-dependencies of these statements are satisfied. If possible, they must even be parallelized. To undertake this extension to our technique, we first group the statements between function calls into blocks. On these blocks, the data-dependency analysis technique used on functions are run. Similar as explained in previous sub sections, any statement with a barrier set before it by any previous function call or statement can not be executed until the barrier has been eliminated. Also, if a statement makes any write operations, then a barrier is set on the next usage of the data undergoing the write operation. This is similar to how the barrier is set with functions as well. Handling selection and iterative statements are done with a similar philosophy. For selection statements that have function call(s) within them, we apply the same function parallelization technique described earlier. This takes care of the selection statements and even ensures that there are no race conditions as it is handled when the functions are parallelized. Every other statement inside selection statements is taken care of as stated previously.

Example 6:

```
int transform_A(int a){
    return a + 2;
}
void transform_B(int *a, int n){
    a[0] = 4;
    return;
}
int main(){
    int res1;
    int *arr = {1,2,3,4,5,6};
    int b1 = 4;
    res1 = transform_A(b1);
    int res2 = 0;
    transform_B(arr, b1);
    for(int i = 0; i < res1; i++){
        res2 += arr[i] + 2;
    }
    res1 = res2 + 2;
    return 0;
}
```

The function transform_A makes a write operation on the variable passed, but it is a pass-by-value, so this has no implications on the argument passed. transform_B has passed by reference for the array, and it makes a write on this array. So for every call of transform_B, we will need to set a barrier for the next usage of the array passed. So in the main function, we set a barrier before the net usage of arr inside the loop. For handling loops we check if all barriers set for it are cleared, if they are, we assign a thread and execute the loop. In the above example, we notice that res1 used res2, so we set a barrier for that assignment, to wait for the loop to complete execution.

F. Comparing the Techniques

- The design of Technique 2 allows for the elimination for scheduling and thread management. This frees both the master threads and removes the requirement for both ready and wait queues. As a result, there is an efficient usage of underlying hardware without any overheads.
- Technique 2 eliminates many if-else constraints and reduces the number of mutex locks being used compared to Technique 1.
- The reduced overheads, scheduling and other changes simplifies the generated parallel code and even reduces the size of the generated code compared to Technique 1. We can, therefore, qualitatively conclude that Technique 2 is a significant improvement over technique 1, in terms of generating parallel code.

IV. Results

Our main aim has been to improve resource utilization of underlying hardware to the maximum potential when executing a program, and thereby reduce the execution times. We have measured this by comparing the execution time of the sequential program and its parallel equivalents, generated by both the aforementioned techniques in Section III.Although execution time of a program can be used to indicate speed-up. As our goal is also to improve the utilisation of available resources, we can not directly use only comparison of execution time between the original sequential source code and the generated parallel source code. In order to better understand the impact of resource utilisation, we have chosen to run the generation of parallel code and its execution on different hardware with varying number of cores and threads. We have specifically chosen an Intel i5 processor with 2 cores and an Intel i7 processor with 12 cores. These architectures are commonly available today to the average consumer and hence can be used as a reflection of the impact in the real world. This allows us to showcase that the parallel version of the program has increased the utilisation of the underlying available computational resource.

The Fig. 3 showcases the comparison of the sequential program with that of parallel program generated by Technique 2, Master Worker based approach, on two different architectures. The Y-axis is the ratio of execution time of sequential to parallel as we increase the number of computations carried out in a program. We have used ratios to represent the increase

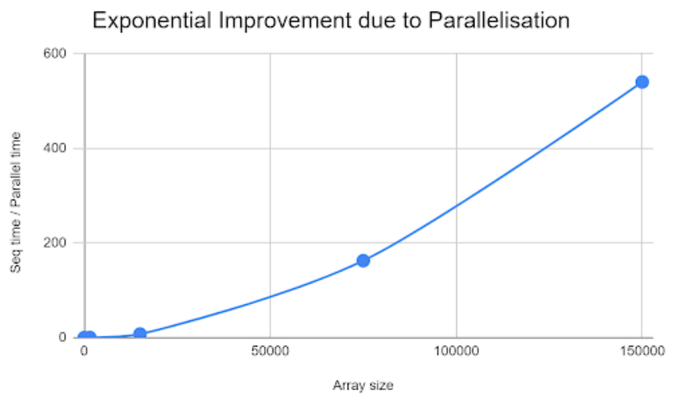


Fig. 3. Sequential vs Parallel

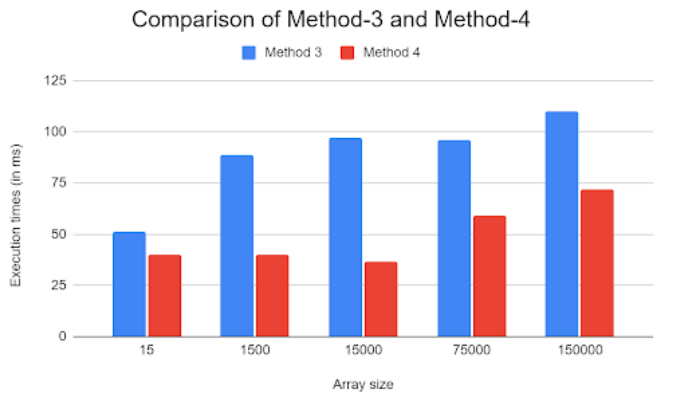


Fig. 4. Technique 1 vs Technique 2, on an i5 2 core machine

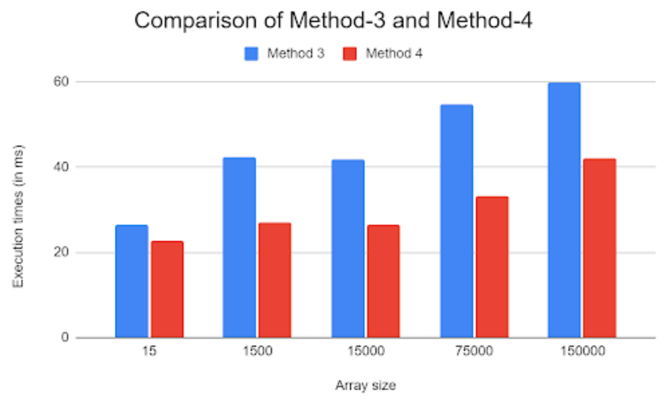


Fig. 5. Technique 1 vs Technique 2, on an i7 12 core machine

in this figure, as absolute comparisons are nearly impossible. The sequential program execution time increases non-linearly, while the increase in parallel time is seemingly more linear, Fig 4 and Fig. 5 shows for both methods. Fig 4 and Fig 5 can be compared to each other. They are the result of running on two different architectures. We notice that while in Fig ¡insert ref¿a the speed up is 500 times almost, in b. it is 550 times. To generate Fig 5. we used an Intel i7 architecture with 12 cores as compared to the i5 architectures with 2 cores. This can be used to infer that, with increased availability of hardware, we can improve the execution. This is possible as the underlying parallel program tries to maximise the resource utilisation, and on a machine with more cores and threads, it can achieve much higher parallelism.

The implementation of Technique 2 is an evolution of some of the ideas established in Technique 1 and additional design features. These changes streamline the resultant parallel code by reducing the number of mutex locks, increasing available threads as workers and removing all scheduling overheads. These changes along with the other described in previous sections justify the improvement in execution time. The Fig 4 and Fig. 5 indicates the impact of these changes. Technique 2 make significant gains on Technique 1 parallel code execution. As the number of computations are increased, Technique 2 continues to perform much better as a result of more available threads and having no busy waiting for thread scheduling.

V. Conclusion

In this work, we have outlined our approaches to auto-parallelization of source code. Our two approaches were that of a master-worker architecture and another which was a modification on the previous, which allowed us to remove the master threads and allow for achieving more parallelism. Both these methods involved an extensive data-dependency analysis. The information from the data-dependency analysis was used to ensure the correctness of the execution.

References

[1] Uday Bondhugula, J. Ramanujam, P. Sadayappan, PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System, 2007.
[2] ParaWise – Widening Accessibility to Efficient and Scalable Parallel Code - White paper
[3] Martino B. D. & Iannello G, Towards automated code parallelization through program comprehension, 1994.
[4] Cantiello P & Di Martino B, Automatic Source Code Transformation for GPUs Based on Program Comprehension, 2012.
[5] Di Martino B & Kessler C.W, Two program comprehension tools for automatic parallelization, 2000.
[6] Martino B. D. & Iannello G, Towards automated code parallelization through program comprehension, 1994.
[7] Peter Kraft, Amos Waterland, Daniel Y Fu Anitha Gollamudi, Shai Szulanski, Margo Seltzer. (2018). Automatic Parallelisation of Sequential programs
[8] Sean Rul, Hans Vandierendonck, Koen De Bosschere. (2007). Function Level Parallelism Driven by Data Dependencies
[9] Bispo, Joao & Cardoso, João. (2020). Clava: C/C++ source-to-source compilation using LARA. SoftwareX. 12. 100565. 10.1016/j.softx.2020.100565.