Automatic Parallelization using Open MP Directives

Deepika Dash, Anala M R

Abstract-With the increase in the advent of parallel computing, it has become necessary to write OpenMP programs to achieve better speedup and to exploit parallel hardware efficiently. However, to achieve this, the programmers are required to understand OpenMP directives and clauses, the dependencies in their code, etc. A small mistake made by them, such as wrongly analysing a dependency or wrong data scoping of a variable, can result in an incorrect or inefficient program. In this paper, we propose a system which can automate the process of parallelization of a serial C code. The system accepts a serial program as input and generates the corresponding parallel code in OpenMP without altering the core logic of the program. The system has used different data scoping and work sharing constructs available in OpenMP platform.The system designed here aims at parallelizing "for" loops, "while" loops, nested "for" loops and recursive structures.The system has parallelized "for" loop by considering the induction variable. And converted "while" loop to "for" loop for parallelization. The system is tested by providing several programs such as matrix addition, quick sort, linear search etc. as input. The execution time of programs before and after parallelization is determined and a graph is plotted to help visualize the decrease in execution time.

IndexTerms–Automatic Parallelization Tool, collapse, OpenMP, OpenMP directives and clauses, pragma directives, parallel computing, recursive structures, task, taskwait

I. INTRODUCTION

Parallel computing has become increasingly popular by virtue of the magnitude of benefits that it offers. Its area of application encompasses several fields of science, medicine and engineering such as web search engines, medical imaging and diagnosis and also multiple areas of mathematics.The High-Performance Computing (HPC) market is estimated to grow from USD 28.08 Billion in 2015 and projected to be of USD 36.62 Billion by 2020, at a high Compound Annual Growth Rate (CAGR) of 5.45% during the forecast period [1].

Parallel computing involves breaking down large problems into smaller sub-problems which may be carried out in parallel.This process of problem solving exploits the parallel hardware architecture of modern day computers and hence enables fastercomputation. While this may not make a difference to simple programs, it helps to save a lot of time, power and cost when dealing with very large, complex problems involving big data or heavy computationsThere are several systems which are implemented as serial programs, converting them to parallel programs will give the product an edge as there are multiple benefits from parallelization. Developing the parallel code for a complex serial program involving numerous dependencies and complexities is not an easy task.

It requires a thorough understanding of OpenMP platform. Manually parallelizing a program is often prone to errors. Debugging the code also becomes tedious. The system developed aims at addressing such problems, by providing a tool which automates the process of parallelization. Automatic parallelization abstracts parallel computing platforms from developers thereby eliminating the knowledge requirement of developer about parallel computing platform [2]. There are several parallel computing platforms such as OpenMP, CUDA, MPI etc. The system developed uses OpenMP as it is easy to use and portable. The primary function of the system is to generate an accurate and efficient OpenMP program for a given sequential C code. To accomplish this, the system is provided with three main modules: Custom Parser, OpenMP Analyzer and Code Generator. The first step deals with parsing of the code. It checks whether the input program is syntactically correct. It returns error messages, if any, to the user. The parser also populates the data structures namely, variable table, statement and function details tables which are used by the OpenMP Analyzer. The OpenMP Analyzer detects blocks of code which have potential for parallelization such as recursions, for loops, while loops, etc. It also generates the dependency graph. The Code Generator adds the directives and clauses to generate parallel program as final output.The generated parallel codes are checked for correctness and speedup. The results obtained are plotted on a graph.

The rest of the paper is organized as follows: In section 2, we present a brief description about tasks, data scoping and collapse clause. In section 3 we describe the methodology by providing algorithms for implementing the various modules. In section 4 and 5 we discuss related work and experimental results respectively. Section 6 provides a conclusion to the paper. The last section, section 7 describes the future enhancements.

II. BACKGROUND

OpenMP specification version 3.0 introduced a new feature called tasking. Tasking facilitates the parallelization of applications where units of work are generated dynamically, as in recursive structures[3]. Data scoping attribute clauses namely, shared and firstprivate are also used. A brief description of these clauses is given below:

- Shared: The shared clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables [4].

Revised Manuscript Received on September 05, 2019.
Prof. Anala M R, Dept. of CSE, RVCE, Bengaluru, India
Prof. Deepika Dash, Dept. of CSE, RVCE,Bengaluru, India

- Firstprivate: The firstprivate clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered [4].

Collapse clause is used to parallelize nested "for" loops. This clause is used only when the loops satisfy prefect nesting and rectangular iteration space. OpenMP provides an atomic directive which specifies the next statement must be done by one thread at a time[5]. The omp atomic directive allows access of a specific memory location atomically. This ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location[6].

III. METHODOLOGY

The system developed enables the user to parallelize a complex serial code involving difficult and time-consuming calculations. The user only needs to provide the serial code in C as input to the system. The system carries out the task of parallelizing the input code by identifying the data and control dependencies, blocks of code that can be parallelized such as recursions, loops etc. These blocks of code are then parallelized by inserting OpenMP clauses and directives as needed. The steps involved in the implementation of the various modules are discussed below.

3.1 Parallelization of nested loops using Collapse clause

"For" loop parallelization is achieved through the use of the directive "#pragma omp parallel for". It is possible to parallelize nested "for" loops too by adding collapse clause to the pragma directive of the outermost "for" loop, however, this is beneficial only for certain kinds of nested loops. In some situations, loops perform better without nested parallelization. Thus, there is a necessity to understand conditions to be satisfied in order to make nested loop parallelism beneficial. It has been found that perfect nesting and rectangular iteration space makes nested loops a suitable candidate for use of collapse clause. Loops are said to be perfectly nested if there are no lines of code between the "for" loop headers. An example of such a code is matrix addition. The C code for Matrix multiplication can also be modified to satisfy the criteria. Such programs show considerable improvement in performance on using the collapse clause. It is the responsibility of the system to identify such loops and add the clause accordingly.

Algorithm 1: Parallelization of nested loops

Input: Sequential C code with nested loops
Output: C code with parallelized nested loops

```
1:  procedure CollapseClauseGenerator
2:  if nested "for" loops present then
3:    if loops are perfectly nested then
4:      if rectangular iteration space flag(variable)==1 then
5:  Determine level of nesting
6:        Append collapse clause to "for" loop's pragma directive
7:      Endif
8:    Endif
9:  Endif
10: end procedureCollapseClauseGenerator
```

The first step is to detect nested "for" loops and check whether they satisfy perfect nesting and rectangular iteration space. The loop is checked for perfect nesting by setting a flag to '1' if the body of the outer "for" loop consists of any statement apart from the inner "for" loop. Each time the condition is satisfied, a variable named nest_level is incremented. Such nested "for" loops are parallelized by appending collapse clause to the outermost "for" loop's pragma clause.

3.2 Conversion of "while" loops into equivalent "for" loops

The WhileToForConverter module is responsible for converting certain "while" loops suitable for conversion into equivalent "for" loops, by identifying the iteration variable using which the initialization and increment/decrement expressions are generated. This step is carried out during parsing stage itself. The converted loops are then parallelized as regular "for" loops.

Algorithm 2: Conversion of "while" loops

Input: Sequential C code with "while" loops
Output: C code with Equivalent "for" loops

```
1: procedure WhileToForConverter
2: Determine iteration variable
3: Generate initialization expression
4: Generate increment/decrement expression
5: Replace "while" header with "for" header
6: Remove increment/decrement expression from body
   of converted loop.
7: end procedure CollapseClauseGenerator
```

The module first identifies "while" loops with single conditions and which are not nested within other loops. Iteration variable is determined for the loops which satisfy the above conditions. During the parsing stage, variable names and their values are stored in a variable table. Once the iteration variable is identified from the "while" loop header, its value is determined from the variable table. This data is used to generate the initialization expression. The increment/decrement expression is generated by identifying the relation operator in the condition expression of the "while" loop.

3.3 Parallelization of recursions using task directive

Two modules are provided for the implementing parallelization of recursions. These modules help identify recursive structures of code to parallelize using tasks and analyse them for shared and firstprivate variables by applying task scoping rules. The taskwait directive is added where require. The RecursionRecognizer module makes use of the statement table, where details such as statement type, nesting level, etc are stored.

The statement type indicates whether a particular statement is a function call, expression, declaration, etc. If the statement type is "function call", then the module checks whether the function is recursive and accordingly sets a flag in the function details table.

Algorithm 3: Detection of recursions

Input: Statement table
Output: Updated function details table

```
1:  procedure RecursionRecognizer
2:  Read statement table.
3:  Detect function calls.
4:  if function is recursive then
5:      Set RecursionBit(variable)to 1.
6:  Endif
7:  end procedure RecursionRecognizer.
```

The TaskDirectiveGenerator module adds the task construct above the lines of code as indicated by the RecursionRecognizer module. This module makes use of the statement and function details table which is updated by the RecursionRecognizer.

Algorithm 4: Generation of task directive

Input: Statement table and function details table
Output: C code with parallelized recursions

```
1:  procedure TaskDirectiveGenerator
2:  Read statement and function details table.
3:  if RecursionBit(variable)==1 then
4:      Scope variables.
5:      Insert task directive.
6:  Endif
7:  if value returned by recursive function call is used
    in following lines then
8:  Insert taskwait directive.
9:  Endif
10: end procedure TaskDirectiveGenerator.
```

The module checks whether the given function call has recursion flag set to 1. If yes, it adds the task directive above its recursive calls. Then data scope of the variables is determined and appended to the task directive. If the function's return type is not void then it adds a taskwait construct after the last recursive call.

IV. RELATED WORK

Par4All and Cetus do not support parallelization of recursive structures. They may have to be replaced by "for" loops to enable parallelization. The system developed here is capable of parallelizing recursive functions through the use of task and taskwait directives. This is checked by providing C programs such as Quick-sort, Merge-sort, etc. as input. In the output programs generated, the recursive functions are parallelized by insertion of task directive at the appropriate lines. Pluto, Cetus and Par4All cannot handle parallelization of "while" loops. Hence "while" loops need to be converted to equivalent "for" loops. The system developed here is capable of automatically converting suitable "while" loops into equivalent "for" loops, which are then parallelized. Intel C++ and Fortran Compilers have the ability to analyse the dataflow in loops to determine which loops can be safely and efficiently executed in parallel [7]. It also performs a check on the code to determine the need for parallelization. Synchronisation and loop scheduling can both be significant sources of overhead in shared memory parallel programs [8].

V. RESULTS AND DISCUSSIONS

The system developed converts input serial C code to corresponding parallel code by adding OpenMP clauses and directives. However, the correctness of the output program generated by the system needs to be verified. This can be achieved by comparing the output generated by both the codes (serial and parallel). A set of sample programs are chosen for each module developed. The first set involves programs for collapse directive, this includes matrix addition and matrix multiplication. The second set consists of programs for while-to-for conversion such as string palindrome check and linear search. The third set deals with recursive programs such as quick sort, merge sort and parallel sum which is used for verifying the module for task directive.

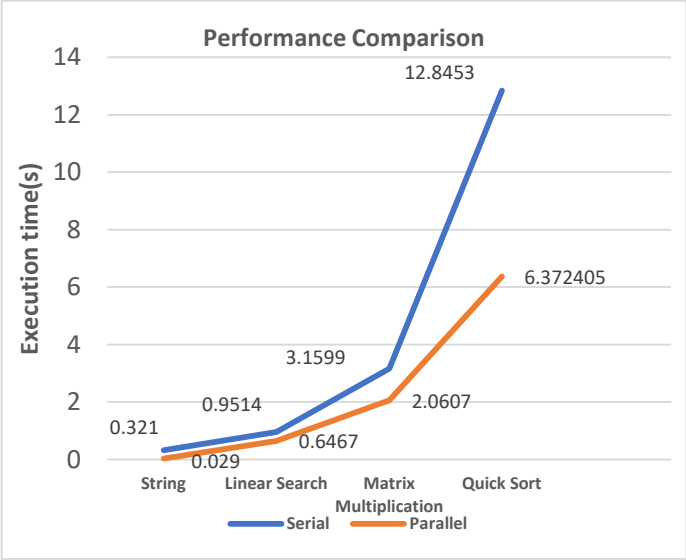

Fig. 1Performance comparison

The execution time of the input serial programs and output parallel programs were determined. A graph is plotted to analyse the results. The graph in Fig. 1 shows that execution time of parallel programs generated by the system is much less than that of their serial counterparts.

VI. CONCLUSION

The aim of the system, to automate the process of parallelization of C code, was achieved. The system was tested by running several programs for each module, e.g.: Matrix Multiplication and Matrix Addition for testing nested loop parallelism, Quick-sort, Merge-sort, etc. for testing parallelization of recursive structures etc.

The generated parallel codes were checked for correctness. A sample program consisting of a set of serial codes such as Matrix Multiplication, Monte Carlo pi calculation, Linear Search and Recursive Sum was run to generate parallel code. The serial code took an execution time of 13.16s while the parallel code took an execution time of 7.73s for the same input size. This shows that parallel codes that were generated by the system perform computations faster. The system helps to eliminate errors that may result on manual parallelization. The correctness of the parallel program was verified by comparing the outputs generated before and after parallelization. The speedup was also demonstrated by comparing the time taken during execution for large input sizes.The system, however has certain shortcomings. OpenMP provides a multitude of clauses and directives, but, only a subset of them have been implemented here."while loops" are parallelized by converting them to equivalent "for loops", this method is not suitable in some situations where "while loops" are preferred over "for loops", such "while loops" are required to be parallelized using different techniques(e.g. task directive).The user is required to have a basic understanding of OpenMP so that he/she may know as to when it is beneficial to parallelize a serial code. For eg: a serial code which is likely to work on smaller input sizes will run faster than the parallel counterpart due to the overhead incurred by parallelization (forks and joins).Fork/join of OpenMP threads consumes extra resource and overhead due to OpenMP threads increases with number of OpenMP threads used [9]. The system accepts only C code as input and converts it into a parallel code using OpenMP directives. However, there are other alternatives to OpenMP such as CUDA and MPI. CUDA is a parallel computing platform and programming model invented by NVIDIA[10]. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)[10].MPI is preferred when running a program on clusters. While this system provides a suitable platform for generating OpenMP programs for shared memory multiprocessors, it may not be the preferred choice for a user whose application demands CUDA or MPI program.

VII. FUTURE ENHANCEMENTS

The project can be further enhanced by adding some more features such as:

- **Schedule clause:**OpenMP offers the *schedule* clause, with a set of predefined iteration scheduling strategies, to specify how (and when) this assignment of iterations to threads is done [11]. The system must be able to identify the best suited scheduling technique for a given input program.

- **Synchronization constructs:**Two statements that are not determined to be concurrent cannot execute in parallel in any execution of the program. If two statements are determined to be concurrent, they may execute concurrently[12]. Some of these directives such as critical, taskwait, etc. have already been implemented but there is still scope to incorporate the other synchronization directives such as master, barrier, flush, etc.

- **Task construct:** The task directive has been implemented for handling recursions, however it is also possible to parallelize while loops using tasks [3]. Adding this feature will help to parallelize the "while" loops which cannot or should not be converted to "for" loops.

ACKNOWLEDGEMENT

We express our gratitude towards Amit G Bhat and Meghana N Babu, Dept. of CSE, R.V. College of Engineering,for providing insight and expertise that greatly assisted the project.We also thank our parents for instilling the confidence to complete this project. Lastly, we thank all the faculty members of R.V College of Engineering for the constant support and encouragement.

REFERENCES

1. MarketsandMarkets, "High Performance Computing Market by Components Type (Servers, Storage, Networking Devices, & Software), Services, Deployment Type, Server Price Band, Vertical, & Region - Global Forecast to 2020," Rep. TC 2204, Feb. 2016.
2. A. G. Bhat, Meghana N Babu, Anala MR "Towards Automatic Parallelization of "for" loops," in Advance Computing Conference (IACC), Bangalore, IEEE, 2015, pp. 136-142.
3. "Sun Studio 12 Update 1: OpenMP API User's Guide," Oracle, [Online]. Available: https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html.
4. IBM Knowledge Center, "Shared and private variables in a parallel environment," [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbcpx01/cuppvars.htm.
5. "OpenMP Synchronization," 26 July 2016. [Online]. Available: http://cs.umw.edu/~finlayson/class/fall16/cpsc425/notes/13-openmp-sync.html.
6. IBM Knowledge Center, "#pragma omp atomic - purpose," [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSGH2K_13.1.0/com.ibm.xlc131.aix.doc/compiler_ref/prag_omp_atomic.html.
7. Intel Developer Zone, "Automatic Parallelization with Intel Compilers," 2 Nov. 2011. [Online]. Available: https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers.
8. J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in Proceedings of First European Workshop on OpenMP, 1999, pp. 99-105.
9. W. Zhang et al., High Performance Computing and Applications: Second International Conference, HPCA 2009, Shanghai, China, August 10-12, 2009, Revised Selected Papers, Berlin Heidelberg: Springer, 2010.
10. nvidia, "CUDA Parallel Computing Platform," nvidia, [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html#sthash.YeEESwLd.dpu.
11. O. Hernandez et al., "Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications," in OpenMP Shared Memory Parallel Programming: International Workshops, IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1-4, 2005, Reims, France, June 12-15, 2006. Proceedings", Berlin Heidelberg, Springer, 2008, pp. 267-278.

12. Y. Zhang et al., "Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers," in Languages and Compilers for Parallel Computing: 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers, Berlin, Heidelberg, Springer, 2008, pp. 95-109.
13. [13] Anala M R, Deepika Dash " Framework for Automatic Parallelization" in 25th International Conference on High Performance Computing Workshops(HiPCW), Bangalore, IEEE, 2018, 978-1-7281-0114-9.