ISSN:0975-9646

Implementing Coarse Grained Task Parallelism Using OpenMP

Manju Mathews [#1], Jisha P Abraham [#2]

[#] *Department of Computer Science and Engineering,*
Mar Athanasius College of Engineering, Kerala, India

Abstract— **Parallelization is an important technique to increase the performance of software programs. Parallel programs are written to make efficient use of multiple cores. Most of the existing legacy applications are sequential without any multithreading or parallel programming. Manual efforts required to parallelize these applications are huge and complicated. So there is a need for automatic parallelization tools. The proposed system implements coarse grained task parallelism with the insertion of OpenMP directives in an input C program. The output is a multithreaded C program which can utilize multiple cores on multi-core shared memory systems. The system is placed between application and compiler and generated output needs to be compiled like a normal C program. This is a source to source conversion tool as the input and output are both C source code. To extend beyond the reach of loop level parallelism, task level parallelism is proposed and provides better scalability. Depending on the nature of the program, parallelization may have a performance improvement.**

Keywords— **Automatic Parallelization, Task Parallelization, Parallel Programming, Coarse grained Parallelism.**

I. INTRODUCTION

The industry trends suggest an increasing number of cores in future processors. To utilize available cores parallel programs should be written. Parallel architectures are well suited for running computationally intensive applications such as scientific simulations. The existing legacy applications do not have any parallelization and when run on multiple cores, utilize only single core. To make best use of multiple cores, there is a need to parallelize these applications.

Manual parallelization is difficult as it requires huge analysis efforts to identify parallelizable code sections, identifying dependency and manually inserting directives at proper places. Parallelization manually is complex as it requires an understanding of the application and its domain. Also requires expertise in parallel programming domain. There is a need for automatic code parallelization tools [1] that can convert sequential applications to parallel and address race conditions and collisions.

There are different parallelization tools, frameworks and language extensions each considering different hardware architecture, memory architecture etc. OpenMP and MPI are language extensions for shared memory parallel machines. Different tools are suitable for different classes of applications and best possible performance may not be obtained from a single tool. Many tools are semi-automatic as the dependency analysis is done manually and parallelization done using libraries or API's. The common approach is on loop level or data parallelism.

The limitation of SMP programming is its limited scalability due to fine grained loop level parallelism. The focus here is to provide more scalability using coarse grained task level parallelism. In this approach, rather than loop level parallelization, we go for task level parallelization. The tasks could be outer level loops (for, while, do-while), if-then statements, function call sites etc. The proposed system identifies the first level tasks in an input program, performs dependency analysis, and generates suitable OpenMP [2] constructs for parallelization. The system provides synchronization in case of dependencies. The parallel code generated runs utilizing multiple cores on SMP's.

First we consider only the outer-level loops and do not go for nesting. i.e., if there is a for- loop inside an if statement, we consider only the outer if statement block for parallelization. As the next level of this work, we consider nesting i.e., precisely two level nesting and we identify the performance benefits. It's always better to parallelize the outer most loop in a loop nest, as this reduces overhead due to parallelization and maximizes the work done for each thread.

The performance benefits of automatic parallelization [3] depends heavily on the nature of application at hand. Some applications with many independent heavy tasks provide a good speedup. Others with many dependencies and light weight tasks does not benefit from parallelization. Instead they may suffer from a time overhead due to parallelization. That is the benefits of multithreaded and multicore parallelism could be completely nullified. Also the performance depends on the number of cores available.

The rest of the paper is organized as follows: Section II gives a brief literature survey of the various tools for parallelization. Section III details the implementation of the proposed system. In section IV, the performance results of the output parallel code is analysed.

II. LITERATURE SURVEY

There are several tools available for parallelization. APIs like OpenMP, Pthreads are for shared memory architectures, MPI for distributed memory architectures and OpenCL and CUDA for GPGPUs. The challenges in parallelization are different writing styles, identifying parallelizable sections, code with I/O, identifying dependencies etc. Automatic parallelization is employed when quick results are required with some performance in case of huge applications.

OpenMP helps to generate parallel code which when executed creates multiple threads. OpenMP version 3.0 supports task constructs. Some of the automatic parallelization tools are:

A. *PLUTO*

Automatic parallelizing tool based on polyhedral model [4]. Transforms C programs considering coarse grained parallelism and data locality simultaneously. Generates OpenMP parallel code from sequential C program sections.

B. *The Intel Compiler*

The Intel compilers [5] automatically generate multithreaded code for C, C++ and FORTRAN. The focus is on computationally intensive loop structures. Loops are parallelized based on data flow analysis.

C. *The Par4All Tool*

Par4All is an automatic parallelization tool for C and FORTRAN based on PIPS (Parallelization Infrastructure for Parallel Systems) [6] compiler framework. Par4All generates OpenMP or CUDA output for shared memory processor or GPGPU and also performs some optimizations.

D. *S2P Tool*

Generates OpenMP parallel C code for shared memory machines. The tool considers loop level as well as task level parallelism. Task parallelism implemented using Pthreads API. The tool considers only outer level loops in the main program without nesting.

Majority of the tools target loop or data parallelism by parallelizing independent loop iterations in the programs. However, parallelizable loops constitute only a small portion of the application in hand. There is an overhead due to parallelization. To provide scalability and support larger class of applications, the proposed system focus on identification of coarser tasks and implementing task level parallelism using OpenMP constructs.

III. Proposed System

The proposed system is an automatic parallelization tool which takes an input C program and provides a multithreaded parallel code. The output C program contains OpenMP task constructs inserted at relevant places to attain coarse-grained task parallelism. This approach decomposes a program into tasks, coordinates tasks to obtain parallelism while maintaining the actual functionality. The output program need to be compiled like a normal program and runs making efficient use of multiple cores. Fig. 1 shows the proposed system in software execution model.

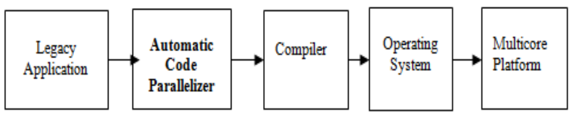

Fig. 1 Automatic Code Parallelizer in Software Execution Model

The proposed system consists of a front end that can scan and parse application code and an intelligent backend that performs static dependency analysis [8] to identify parallelizable sections of code. The backend has a code generator that generates parallel code automatically without human intervention. The parallel code maintains functionality of the sequential code but can execute faster and can optimally utilize all the available cores on the hardware. Fig.2 shows high level block diagram of the proposed system.

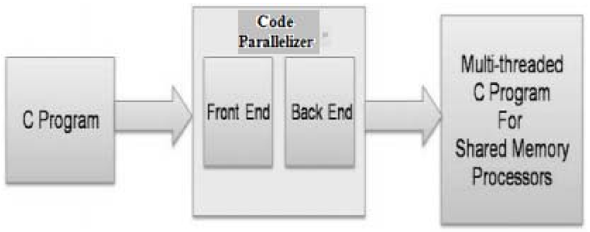

Fig. 2 Block diagram of Automatic Code Parallelizer

A. *The Front End*

The front end does a static analysis to gather code information in an intermediate file format. This intermediate form serves the basis of various analysis done by backend modules. The different modules are variables identifier, blocks identifier, block-rwlist identifier etc. The sequential code is scanned line by line and generates tokens. These tokens are parsed as per ANSI C grammar and metadata about various constructs stored in intermediate file format.

The variables identifier module identifies all local and global variables and stores information like variable name, type, scope, line number etc. The blocks identifier identifies various code blocks that is considered for parallelism. The blocks are based on program structures and can be if statement block, while loop block, do-while loop block, for loop block etc. The block-rwlist identifier module identifies the list of variables used inside each block. All the information in the intermediate file format is provided to the back end for further analysis.

B. *The Back End*

The main modules in the back end are dependency analyser, TDM creator and code generator. Back end does a static dependency analysis and records dependency in TDM .Code Generator generates parallel code by inserting OpenMP directives and inserts synchronization constructs consulting TDM where ever appropriate.

C. *Dependency Analysis*

Data dependence relations determine when two statements can be executed in parallel. The four basic types of data dependence are:

1) Flow Dependence: Also called True Dependence. Occurs when data written by one statement is read by other.

S1: z = x + y
S2: c = z * 2
Here S1 writes z and S2 reads z. There is a flow dependence.

2) Anti-Dependence: Occurs when data read by one statement is written by the other.

S1: z = x + y
S2: x = c * 2

Here there is anti-dependence between S1 and S2 caused by variable x

3) Output Dependence: Occurs when data written in one statement is later written by other.

S1: z = x + y
S2: c = x * 2

Here there is an output dependence between S1 and S2 caused by variable z.

4) Input Dependence: Occurs when data read by one statement is later read by other statement.

S1: z = x + y
S2: c = x * 2

Here there is an input dependence between S1 and S2, caused by variable x.

For Task parallelization, we need to analyse flow dependence and output dependence. The dependency analysis is done using variable update analysis. Dependence information is stored in an n X n matrix called Task Dependency Matrix (TDM), where 'n' is the number of blocks identified. A value '1' indicates dependency between corresponding blocks.

Algorithm for Dependency Analysis

1) Create and initialize an n X n matrix (TDM) where n is the number of tasks or blocks identified

2) For each task, compare with all other successive tasks.

3) If a variable written in one task is read or written in another task, mark dependency in TDM as '1' in the row and column corresponding to the tasks.

4) Repeat for all the identified tasks or blocks

D. Generating Parallel Code

Code Generator generates the parallel multithreaded code by inserting OpenMP directives. The number of threads can be specified using OMP_SET_NUM_THREADS () routine. All the data needed for generation of parallel code is obtained in a completely automatic manner without human intervention. The tasks which are independent are executed in parallel. In case of dependencies identified from TDM, synchronization constructs are inserted. Data scoping can be specified explicitly. If there is a write dependency on a variable, specify as shared explicitly.

The steps for Single-level Task parallelization can be summarized as follows.

1) Selection of appropriate hardware

2) Identification of parallelizable code blocks, like loops, condition statements, expressions etc.

3) Performing dependency analysis for code blocks and storing dependency information in TDM

4) Generating parallel code using OpenMP constructs

5) In case of dependencies, provide appropriate synchronization constructs

6) Execute parallel code and compare performance [9] with sequential version

In case of I/O statements, proposed system considers printf as a read and scanf as a write. The generated parallel code needs to be compiled using compiler option –fopenmp to enable recognition of OpenMP directives. The parallel code runs utilizing all the cores in multi-core shared memory systems. The performance depends on the nature of the program, degree of parallelism it has, overhead due to parallelization, number of available cores, processing speed, memory architecture etc.

IV. Results and Performance Analysis

For testing purpose, dual core Intel Core i5 with Hyper Threading (HT) is used (TABLE I). Operating System is Ubuntu 14.04. The compiler is gcc 4.8.2 supporting OpenMP version 3.1. A dual core with HT has 2 physical cores but scheduler considers as 4 logical cores.

TABLE II
Testing Environment

Processor Type	Intel i5 processor-dual core (HT)
Operating System	64-bit Ubuntu Linux 14.04
Clock Speed	2.2 GHz
RAM Size	4 GB

Two test applications are considered for parallelization with a number of nested loops. The proposed Automatic Code Parallelizer is run by varying the number of threads and performance results are analysed.

Fig.3 shows the execution time on 2 cores with varying thread number for test program1. With 2 threads there is an improvement. However there is no improvement after 2 threads because the test program contains only 2 parallel code blocks. Rest of the blocks are executed sequentially because of dependency.

Fig.4 shows execution time on 2 cores for test program2 with varying number of threads. There is an overhead due to parallelization. Also performance is at its peak when run with 4 threads. Test program2 consists of 4 independent blocks that could be run in parallel. Hence best performance when thread number is 4.

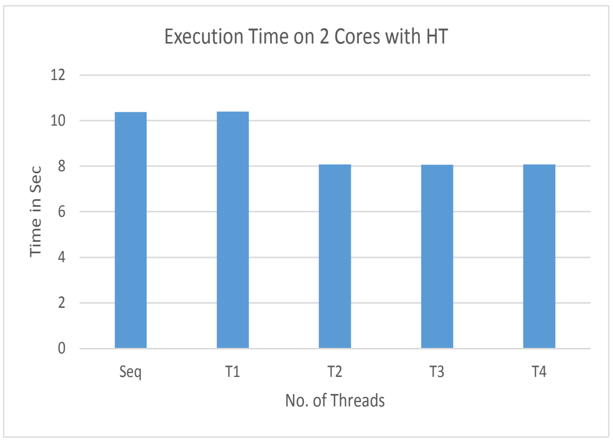

Fig. 3 Execution Time of Test Program 1 by Varying Number of Threads

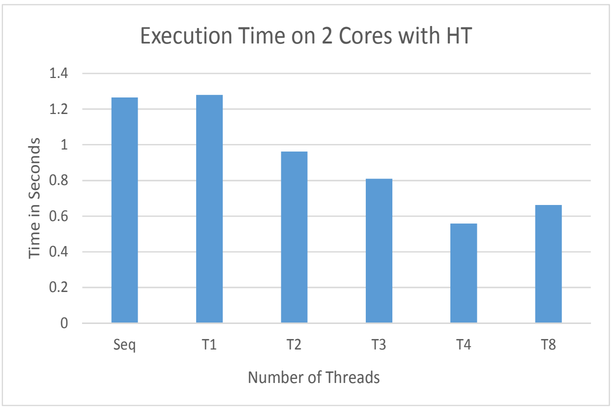

Fig. 4 Execution Time of Test Program2 by Varying Number of Threads

V. Conclusion and Future Work

Parallelization is an important technique to increase the performance of software programs. There is a need to convert existing legacy applications to parallel for getting performance gains on multiple cores. Manual parallelization is tedious and error prone and requires huge efforts. So automatic parallelization tools are preferred. The proposed system focus on implementing a task level parallelism using OpenMP constructs. The tool takes a sequential C program as input and provides a functionally equivalent parallel version. Different code blocks considered for parallelism are loop blocks, condition statements, expressions etc. Parallelism results from execution of independent tasks concurrently. In case of dependencies synchronization constructs are inserted to ensure sequential semantics. The performance gained by code parallelization depends on the inherent degree of parallelism present in the original sequential program. Performance is also proportional to the available number of cores.

As a future work, plans to implement nested parallelism, specifically 2 level nesting. Also dependency analysis need to be extended to pointers and function call side effects. There is a need to identify various optimizations to reduce the overhead due to parallelization.

References

[1] P. Randive and V. G. Vaidya, “Parallelization tools,” in Second International Conference on MetaComputing, 2011.ev and V. P. Veiko, *Laser Assisted Microtechnology*, 2nd ed., R. M. Osgood, Jr., Ed. Berlin, Germany: Springer-Verlag, 1998.

[2] OpenMP Official site: http://www.openmp.org/

[3] http://en.wikipedia.org/wiki/Automatic_parallelization_tool

[4] Pluto tool, http://pluto-compiler.sourceforge.net

[5] http://software.intel.com/en-us/articles/automatic-parallelization-with-intel compilers/

[6] Par4All tool, http://www.par4all.org/

[7] Aditi Athavale, Priti Ranadive, M. N. Babu, Prasad Pawar, Sudhakar Sah, Vinay Vaidya, and Chaitanya Rajguru, “Automatic Sequential to Parallel Code Conversion The S2P Tool and Performance Analysis”, Global Science & Technology Forum (GSTF) Journal on Computing, 2012, vol. 1, no. 4, pp. 128-137

[8] Anish Sane, Priti Ranadive, and Sudhakar Sah, "Data dependency analysis using data-write detection techniques", In International Conference on Software Technology and Engineering (ICSTE) 2010, vol. 1, pp. 1-9.

[9] Multicore Processor, Parallelism and Their Performance Analysis I Rakhee Chhibber, IIDr. R.B.Garg I Research Scholar, MEWAR University, Chittorgarh.