Abstract: Multicore processors are very commonly used today in the form of dual core and quad core processors. The industry trend suggests that the number of cores is still going to rise in the coming years. This multicore paradigm shift has forced the software industry to change the way applications are written. To utilize the available cores to their highest potential parallel programs are needed. Similarly, legacy application codes need to be re-written or parallelized so that the new multicore hardware is exploited fully. Writing parallel programs manually is difficult, cost and time consuming and hence there is a need for tools that can aid to convert legacy sequential codes to parallel codes. To address this need we present the S2P tool (Serial to Parallel) an automatic code conversion tool which parallelizes sequential C programs for multicore shared memory architectures. The tool is a source-to-source conversion tool. The S2P tool performs static analysis of the sequential code and supports both task level and loop level parallelization. The output source code is a multithreaded code. In this paper, we also present our benchmarking results and compare the S2P tool with other auto-parallelization tools. Additionally, we also present few basic criteria for benchmarking.

Keywords: Automatic Parallelization, Parallel programming, Performance, Benchmarking

1 Introduction

Parallel architectures have been evolving for past few decades. They are mainly developed for running computationally intensive applications such as scientific simulation. However, today’s processors are faster, the scale at which such computationally intensive problems are being solved has also increased. To better utilize this computing power we need architectural changes at the software level as well. One approach would be to design the software in such a way that it uses the underlying hardware to its fullest. However, what if the software is already built and deployed widely? In such case, either we can rebuild it considering the hardware platform or use parallelization tools to convert the existing code.

Parallelization tools or frameworks take into consideration the hardware architecture, memory architecture, data and control dependencies in the software. Most of the existing parallelization tools or frameworks are semi-automatic in nature implying that the dependency analysis be performed manually and once the dependencies are identified, parallelization of the code can be done using libraries or APIs provided by the framework. Examples of such parallelization libraries or frameworks are MPI and OpenMP. However, these tools aid to parallelize codes, the major and most challenging problem of code analysis remains. Parallelizing simulation applications requires humungous efforts. These efforts increase the time, cost, and the parallelized codes needs testing for its functional equivalence with the sequential counterpart.

To reduce the manual analysis burden, time and effort, we present a completely automated parallelization tool (Serial-to-Parallel i.e. S2P tool), which converts a program written in C language into its parallelized, functionally equivalent version which is tailored for multicore architecture.

Automatic conversion of serial code into its functionally equivalent parallel version remains an open challenge for researchers for the past decade. Many researchers from academia and industry have been working towards such automatic parallelization tool. These tools can transform legacy serial code into a parallel code to execute on parallel architectures. Though there are few tools that perform this task, the industry is still in search of a tool that will give optimum performance improvement for variety of programs. The major challenges involved in design and implementation of such a tool include finding alias variables[1], dependency between statements[2], side-effects of function calls[3] etc. Additionally the tool has to deal with the variety in coding styles, length, and number of files. It is also important to take into consideration the amount of inherent parallelism the application provides.

Some such tools and research efforts are compiled in paper [4]. It discusses tools like CAPTools: an interactive computer aided parallelization toolkit, Portland Group’s HPF compiler and the MIPSPro FORTRAN compiler and compares their performance in terms of required user interaction, limitations, portability and performance. Though [4] has compiled this list it is old and new tools have appeared. Hence, in this paper we compare two new tools with S2P tool and present few basic performance criteria to analyze and evaluate automatic parallelization tools. The three automatic parallelization tools are evaluated, based on the stated criteria and performance results presented in the further sections.

The outline of the paper is as follows: Introduction to different automatic parallelization tools, followed by information about selected test application codes. We then present the benchmark criteria and performance results of all the mentioned tools. The paper concludes with future opportunities in the field of automatic parallelization tools and their benchmarking.

2 Literature Survey of Automatic Parallelization Tools
There are number of automatic parallelization tools available which support different programming languages like FORTRAN and C. Following is the list of such tools or compilers.

1. Vienna Fortran compiler [5]
2. SUIF compiler [6]
3. Polaris compiler [7]
4. iPat/OMP [8]
5. PLUTO [9]
6. Par4All [10]

Due to limited scope of this paper we will not discuss all the mentioned tools. The tool like Vienna Fortran, SUIF and Polaris are great tools but less maintained today. We do not discuss the PLUTO tool since it exploits loop parallelization and also performs loop transformations. Whereas the S2P tool exploits both task and loop parallelization and does not perform any code transformations. Hence making an apple-to-apple comparison of these tools would not be possible. We choose Par4All and CETUS for our comparison and following sections describe them in brief.

2.1 Par4All

Par4All is an automatic parallelizing and optimizing compiler that supports programs written in C and FORTRAN. It is based on PIPS (Parallelization Infrastructure for Parallel Systems) [12] source-to-source compiler framework. The “p4a” is the basic script interface to produce parallel code from user sources. It takes C or FORTRAN source files and generates OpenMP or CUDA [13] output to run on shared memory multicore processor or GPU respectively [14]. The global architecture of Par4All is given in Fig. 1.

![Fig. 1 Par4All architecture](image1)

2.2 CETUS

Cetus is a tool that performs source-to-source transformation of software programs. It also provides basic infrastructure to write automatic parallelization tools or compilers. Cetus architecture is similar to Polaris, the main difference is Polaris translates FORTRAN codes and Cetus is designed for programs written in C language. The basic parallelizing techniques Cetus currently implements are privatization, reduction variables recognition and induction variable substitution. Cetus enables automatic parallelization by using data dependence analysis with the Banerjee-Wolfe inequalities, array and scalar privatization. It uses GNU CPP for preprocessing, Yacc [15] & Bison [16] for scanning, and Antlr [17] which is bundled with its own scanner generator for parsing. Fig. 2 shows the all components and interfaces of Cetus.

![Fig. 2 Cetus components and interfaces](image2)

3 The S2P Tool

S2P is a Sequential to Parallel automatic code conversion tool developed by KPIT Cummins Infosystems Ltd. India. It takes C source code as input which may have multiple source and header files. The output code is a multi-threaded parallel code using pthreads functions and OpenMP constructs. The S2P tool targets task parallelization as well as loop parallelization. Fig. 3 shows the where S2P tool fits in the typical software execution model and the block diagram of the S2P tool respectively.

![Fig. 3 S2P tool in typical software execution model and Block diagram](image3)

The S2P tool contains total ten modules which are grouped into two major modules: Front end and Back End. The Front End generates an intermediate file as output, which acts as input to the Back End. The intermediate file contains information of the given source code. Front end has Pre-scanner, Scanner and Parser as sub modules.

Back End does the source code analysis and finally transforms the code into its parallel version using pthreads library and OpenMP constructs. It has modules like profiler, blocks identifier, alias analyser, side effect analyser, TDM creator, code transformer and parallelizer. The S2P tool identifies blocks based on the different programming constructs. For example a loop is one block, a function call site is one block. Thus a block can be one or several lines of code. These blocks are profiled to get information about execution time and identify blocks that are most time consuming and target them for parallelization. The Alias analyser and side effect analyser helps to identify dependencies among the blocks. The alias analyser finds which variables are pointing to same memory address. Side effect analyser gives information about which argument variables are changing due to
function call. TDM creator forms the task dependency matrix which gives information about dependent tasks. Code Transformer uses TDM and intermediate file and tries to restructure certain blocks of code like if-else [18], so that it can run in parallel. Parallelizer puts necessary pthreads and OpenMP constructs at proper places to generate multithreaded code as final output.

3.1 Features of S2P Tool

Task Parallelism and Loop Parallelism

The S2P Tool supports task level parallelism as well as loop level parallelism. With the help of dependency analysis, it identifies independent tasks in the code. Additionally, it has a special module for dependency analysis for loops, which identifies those loops in the program that have no dependency across their iterations and parallelizes these loops as well.

Static analysis and Profiling

The S2P Tool relies on the static information provided by the code and does not consider the dynamic or runtime aspects for extracting parallelization. In a sense it takes safe approximations and may miss the parallelization opportunity but at the same time guarantees functional correctness. Though the S2P tool does not perform dynamic analysis, it optimizes runtime performance of the resulting parallelized code by automatically profiling the sequential code. It then decides the optimized number of threads to be created to keep multithreading overhead to a minimum.

Scanning and Parsing of the Code

Lexical analysis of the entire code is performed and information is stored. The output of this analysis goes to a handwritten parser. When parsing a C code the S2P tool considers various programming styles. Current version of the tool supports ANSI-C grammar. The parser follows a recursive-descent strategy. Similar to a syntax analyzer in a compiler it generates a symbol table. Unlike a compiler that generates AST storing code information the S2P parser stores information in an intermediate file format that is much more detailed than AST. For example, S2P tool stores information about variable line numbers along with the data type, scope etc, similarly for information related to function calls. For e.g., unlike the AST, the S2P tool intermediate file stores return type information.

Dependency Analysis

To find the dependencies between various program entities like variables, pointers, and functions, techniques such as control flow graph, pointer alias analysis, and side effect analysis are used. The pointer alias analysis is context insensitive and flow insensitive. The dependency analysis module considers the various points or line numbers where all variables are updated and based on the information code sections are declared as parallelizable.

Run Ahead Mechanism

To increase the performance of parallel programs, a technique of Induced Parallelism is included in the S2P Tool [18]. This work is currently in progress. It uses the run-ahead concept, wherein the ‘then’ and ‘else’ statements related to an ‘if’ statement are executed simultaneously on idle cores, considering the dependencies between these statements and the conditional expression in the ‘if’ statement.

Multithreading Approach.

Tasks are identified in the program with the help of dependency analysis techniques mentioned above. A task dependency matrix (TDM) is created using the dependency analysis data. For example, two tasks that are dependent on each other would be marked as 1 and independent tasks in the matrix will be marked as 0. This matrix is used to insert threading constructs including thread creation, thread synchronization etc. Thus even though tasks are dependent some ahead of time execution is achieved by executing the remaining part of the code in parallel once the dependency is resolved.

Graceful Termination

There can be exit statements in the program based on some conditions. The S2P Tool handles these statements in a manner such that even with task-level parallelism, the exit state of the program is retained as that of original program. Thus, the S2P tool guarantees functional correctness.

Loop parallelization.

Loop parallelization is the most common approach for parallelizing applications having data level parallelism. The S2P tool performs different dependency tests to check for dependencies across multiple iterations of the loop. Once a loop is identified as parallelizable, the S2P tool inserts appropriate OpenMP constructs in the code. The data for these constructs are populated using the information generated by the above-mentioned features.

4 Test Applications

The test applications selected for the experiment are as follows:

4.1 EP (Embarrassingly Parallel) NAS Parallel Benchmark (NPB 2.3) Code

To test parallelization tools and compilers NAS parallel benchmarks are considered as a starting point. These are derived from CFD (Computation Fluid Dynamics) codes. These benchmarks were designed to compare the performance of highly parallel computers and are widely recognized as a standard indicator of computer performance [19].

For testing purpose EP kernel is used. The current implementation of EP can run on any number of processors. The calculation also contains a significant number of logarithmic and square. Other than the above NAS benchmark code, we have used a standard matrix multiplication code for benchmarking.
4.2 Matrix Multiplication

In parallel computing, matrix multiplication code is considered important due to use of arrays and loops which operate on those arrays [20]. Multiplication of large matrices requires a lot of computation time as its complexity is $O(n^3)$, where $n$ is the dimension of the matrix.

5 Performance Criteria

We evaluate the S2P tool, CETUS and Par4All tools based on the following parameter. In the following subsection, we describe each performance criteria very briefly and we discuss its results in details in the next section.

5.1 Criteria 1 - Performance & Scalability of Output Program:

The most important criteria for automatic parallelization tools, is that the output parallel program should give better performance in terms of execution time compared to serial time. The tool should generate parallel codes with good performance and with scalability property. Execution time of parallel code $T(n+1)$ on $n+1$ number of processors must be less than or equal to execution time $T(n)$ on ‘n’ number of processors. We checked the performance of output parallel codes given by each tool with varying the numbers of cores.

5.2 Criteria 2 - Memory and Time Complexity:

One aspect of evaluating these types of tools is their efficiency in terms of run time and memory usage when dealing with realistic applications. Efficient usage of memory is important for source-to-source transformation tools or compilers because the entire input program must be kept in memory for different operations like alias analysis, interprocedural analysis, dependency graph formation, and transformation. For comparison, we measured memory usage and processing time of each tool for all selected test codes. To check the execution time of the tool, we used “time” command with verbose option.

5.3 Criteria 3 - Parallelization Overhead

How many extra instructions that the tool inserts into input code to make it parallel is also an important issue. In short, the time required to execute parallel code on single processor should be nearly same as that of serial code. In this criterion, we tried to find, which tool adds fewer instructions compared to other in the given serial code while doing parallelization process. For this, the serial and parallel time difference is measured for selected codes and then compared the results to see which tool puts minimum parallelization overhead.

5.4 Criteria 4 - User Interaction:

The process of transforming serial code to parallel code should have minimal user interaction and less user efforts. Ideally, an automated tool should take only essential inputs from user. The essential input to these kind of tools is source code and header files.

6 Experimental Setup

Two types of machines have been used for evaluation purpose. A Quad Core CPU with 2.66 GHz and 3 GB RAM with Ubuntu 10.04, 64-bit version operating system for checking scalability and speedup of output code. In addition, another is Intel Pentium D (2 cores) with 2.80 GHz Frequency and 1 GB RAM and having Ubuntu 10.04 32-bit version operating system, for testing the other criteria.

7 RESULTS

7.1 Performance & Scalability of Output Program

Fig. 4 & 5 show the execution time for parallel codes of EP and Matrix Multiplication given by each tool, with varying number of cores. For EP, the CETUS and Par4ALL give near about same performance as that of serial code. The S2P shows significant parallelization overhead, as it creates total 26 tasks. The tasks created are very small and the time spent in synchronizing 26 threads puts extra overhead.

The CETUS and Par4All parallelizes the “for” loops which has very small iteration size and only one or two instructions are present in the body. Due to this, parallel loop takes more time than that of serial one. In addition, these tools do not identify portion of the EP code, which takes considerable processing time. This time taking portion can be parallelized using OpenMP constructs like ‘copyin’, ‘master’ and ‘critical’. For the Matrix Multiplication code, all the three tools perform better with good scalability property.

Table 2 shows the serial vs. parallel code execution time. These two test codes viz., Travelling salesman problem (TSP- 1 file, Lines of code: 602) and MP3 decoder codes (7
files, Lines of code: 4933) could not be executed with the Par4All or CETUS tool since these tools were unable to parse these source code and errors were generated.

<table>
<thead>
<tr>
<th># Cores</th>
<th>TSP Code</th>
<th>MP3 Decoder Coder</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time (Sec)</td>
<td>Execution Time (Sec)</td>
</tr>
<tr>
<td>Sequential</td>
<td>Parallel</td>
<td>Sequential</td>
</tr>
<tr>
<td>1</td>
<td>21.56</td>
<td>44.47</td>
</tr>
<tr>
<td>2</td>
<td>15.99</td>
<td>34.80</td>
</tr>
<tr>
<td>3</td>
<td>16.70</td>
<td>35.02</td>
</tr>
<tr>
<td>4</td>
<td>16.82</td>
<td>35.24</td>
</tr>
</tbody>
</table>

7.2 Memory and Time Complexity

Fig. 6 & 7 show the memory usage and processing time required for each tool respectively. Among the three tools, S2P has less memory usages. However, its processing time is more for EP, as it does both loop and task level parallelization. CETUS internally calls other tools like PIPS and Antlr, due to this it shows higher memory usage.

Table 2. Execution Time for Serial and Parallel Code

7.3 Parallelization Overhead

Fig. 8 & 9 give comparative information about the amount of overhead added by each tool to transform input serial code into parallel code, respectively for EP and matrix multiplication code. For the Matrix Multiplication, all are having near about same overhead. However, for EP code S2P shows more overhead due to thread synchronization at task level as compared to other parallelization tools.

7.4 User Interaction

All the tools that we have selected provide command line user interaction. For Par4All and CETUS, user has to provide more options than S2P.

Table 1. Commands to execute the discussed automatic parallelization tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2P</td>
<td>$ S2P tool file1.c [files.c ...]</td>
</tr>
<tr>
<td>Par4All</td>
<td>$ p4a [options] &lt;source files&gt;</td>
</tr>
<tr>
<td>CETUS</td>
<td>$ cetus -parallelize-loops source files</td>
</tr>
</tbody>
</table>

8 CONCLUSION

In this paper, we have presented the S2P tool that is a fully automatic sequential to parallel code conversion tool. We also present four benchmarking criteria for evaluation of automatic parallelization tools and compilers. Through the experiments carried out for benchmarking, we come to the following conclusion - Though the CETUS and Par4All tools mentioned in this paper are able to generate parallel codes, more efforts are required to make these codes optimum in terms of performance. These tools should try to skip the loops that have smaller execution time. The performance of parallel code will increase when all the threads are mapped to physical cores. For task level parallelization, the task should have optimal size and less dependencies.
Acknowledgement - We would like to thank the CREST - S2P team members for the support extended to implement the S2P tool.

I. References