Parallelization Tools
Priti Ranadive$ and Vinay G. Vaidya#

$ Research Scholar, Symbiosis Institute of Research and Innovation
# VP and Head, Center for Research in Engineering Sciences and Technology (CREST), KPIT Cummins Infosystems Ltd., Pune, India

Abstract— With the increasing proliferation of multicore processors, parallelization of applications has become a priority task. One considers parallelism while writing new applications, to exploit the available computing power. Similarly, parallelization of legacy applications for performance benefits is also important. Many different tools, frameworks, language extensions are now being available in the market to help programmers to make their applications parallel. There is no common view as to which tools are best since each tool has different aims. Hence, it is very important that programmers take the right decision while choosing parallelization tools. This paper gives an overview of the available tools and classifies these tools for better understanding. We also introduce our parallelization tool, the S2P Tool, which is a fully automatic parallelization tool. It converts sequential C source code to parallelized C source code without any manual intervention. We also discuss some other representative tools briefly and provide insight so that programmers would identify the right kind of parallelization tools for their applications.

I. INTRODUCTION
Multicore processors have affected the software industry largely [2][3][4]. Easy availability of multicores has forced programmers to rethink the way they design and write their software applications. Parallelism is an important aspect to consider while creating new applications since it guarantees performance gains using multicores. Similarly, for existing legacy applications there is need to either re-write them or convert them to parallel using some tools.

To exploit parallelism new tools, frameworks, languages or extensions to current languages are required. Today a plethora of parallelization tools, frameworks, languages are available in the market. Each one has its importance and may be suitable for parallelizing different kinds of applications. No one tool is suitable for parallelizing all kinds of applications. It is also difficult to exploit the best parallel performance using any one tool.

Hence, in this paper we have classified different available tools, frameworks or language extensions based on the user assistance they provide, based on the target hardware/applications or based on the software design process followed. We also discuss in which situations a particular tool would be helpful and give useful hints to the readers so they can make good choice of parallelization tool for their application.

Henceforth, in this paper we refer to all parallel programming languages or extensions to existing languages, or frameworks or graphical tools as “parallelization tools”. In this paper we do not include tools that are used to check performance bottlenecks of parallelized applications.

The process to parallelize any give code is as shown in figure 1. The parallelization process starts with identifying code sections that the programmer feels have parallelism possibilities. Often this task is difficult since the programmer who wants to parallelize the code has not originally written the code under consideration. Another possibility is that the programmer is new to the application domain. Thus, though this is the first stage in the parallelization process and seems easy at first it may not be so.

The next stage is to shortlist code sections out of the identified ones that are actually parallelizable. This stage is again most important and difficult since it involves lot of analysis. Generally, for codes written in C/C++ languages are difficult to analyze since they use pointers heavily. Many special techniques such as pointer alias analysis, functions side effects analysis [7][8] are required to conclude whether a section of code is dependent on any other code. If dependencies in the identified code sections are more the possibilities of parallelization decreases.

Sometimes, modifying the code, without modifying its functionality, can remove dependencies and this is the next stage in parallelization. Code is transformed such that the functionality and hence the output is not changed but the dependency, if any, on other code section or other instruction is removed. The last stage in parallelization is generating the parallel code. This code is always functionally similar to the original sequential code but has additional constructs or code sections which when executed handle multiple threads or processes.

![Figure 1 – Parallelization stages](image-url)

Different parallelization tools help users/programmers at different stages in parallelization. Figure 2 shows few example tools that aid at different stages of parallelization. In the rest of the sections we will discuss different tools and classify them...
Based on user assistance required, target architecture suited and software design.

Figure 2 – Example of tools aiding at different parallelization stages

II. CLASSIFICATION – BASED ON USER ASSISTANCE

This section classifies parallelization tools based on the user assistance that is required for parallelization tools to generate parallelized code. As seen in figure 2, tools like SUIF or Polaris [1][5] aid users at all stages of parallelization and hence are fully automatic tools. Tools like OpenMP [10] or MPI [11] only aid the code generation part but the user assistance is required to identify parallelizable code sections. We classify such tools that require partial user assistance as semi-automatic tools. Similarly, tools like PryProf [6] are semi-automatic since it does not help user to analyze identified code for parallelization. The third type is manual parallelization and obviously, there is no tool involved since all the stages in the parallelization process involve the programmer or user. The effectiveness of the manual parallelization process depends on the user’s experience and intelligence. Figure 3 shows this categorization of parallelization tools.

In the parallel computing domain, automatic parallelization of code is the most difficult task. The main difficulties involved are as follows:

1. Different styles of writing code
2. Identifying parallelization sections in the code
3. Dependency analysis techniques
4. Handling communication and I/Os
5. Generating fully functional multithreaded or parallel code

Let us discuss each of these issues one by one. The issue of handling different styles of writing code mainly relates to how much inherent parallelism a code has. Two different programmers can implement an algorithm in different ways. Nevertheless, how they write the code would affect the extent of parallelization achieved. For example, consider the following C code that operates on an array

```c
for(i=0;i<A[i]!="\0";i++)
{
}
```

One can write the same code as follows

```c
char *p=&A[0];
while(*p!='\0')
{
    *p = *p + 32;
}
```

Both the example codes perform similar operations and give same results. However, the code using pointers when considered for parallelization, would need to consider alias analysis [7] to take parallelization decision. Alias analysis is a time consuming and difficult task in dependency analysis and most of the times automatic parallelization tools would end up taking safe approximations or safe decisions. Safe approximations guarantee functionality but will not parallelize otherwise parallelizable code sections thus affecting the percentage parallelization achieved and even the performance of the application. The execution time speedup achieved after parallelizing such code would be less.

Identifying parallelization code is also another difficult task. Many tools target loops for parallelization and do not consider other sections of code. Some tools take parallelization decisions based on the profiling information obtained for blocks of code. These decisions are heuristics based or require offline execution of code. Parallelization of code sections that are not loops and that contain I/Os is very difficult. Similarly, the granularity of blocks of codes matters. Too small code block sizes would lead to performance bottlenecks and too large code sizes would lead to lesser extraction of parallelization.

Automated dependency analysis is difficult as discussed in the earlier section. Dependency analysis includes side-effect analysis [8], alias analysis, loop carried dependency analysis [13] and task dependency analysis. All these techniques have been favorite research topics for years but again the safe approximations taken by these techniques would hamper the total amount of parallelization achieved.
Code generation module in fully automatic parallelization tools generally target particular architecture and are language or environment specific. Similarly, I/O is not parallelizable and is a major bottleneck in parallelization. Even though these issues make automatic parallelization difficult, many researchers have attempted to implement automatic parallelization tools. S2P tool, SUIF compiler, Polaris compiler, Rice FORTRAN D compiler [14], Paradigm compiler [15] and Par4All [16] are few such automatic parallelization tools. Discussing all the available tools is out of scope of this paper. Hence, we discuss our tool the S2P tool.

III. THE S2P TOOL

The S2P Tool is a fully automated parallelizing compiler that converts sequential C source code to a parallel-multithreaded source code for shared-memory architecture. It includes the conventional parallelization techniques that form the basis for most commercial parallelizers today. It performs data dependence analysis, which determines whether loop iterations operate on different array elements, to determine if the loop is parallelizable. It is a more contemporary tool as compared to the SUIF and Polaris but uses similar compiler optimization and parallelization techniques as the SUIF compiler.

As shown in figure 4, the S2P Tool placed between the application and the compiler. It converts a given C source code that is sequential into a parallel C code by adding threading constructs (OpenMP and pthreads). The S2P tool is a parallelizing compiler having a front end or a scanner parser module and a backend, though it does not check for syntactical errors. Instead, it stores parsed information in an intermediate format similar to the AST and this information is used by the backend to analyze the code for dependencies.

![Figure 4 - S2P tool in typical software execution model](image)

Some of the major features of S2P tool include task parallelism and loop parallelism, static analysis and profiling, scanning and parsing of the code, control flow graph, variable updates, alias analysis, side effect analysis, if-else analysis, task formulation, multithreading approach and graceful termination.

The S2P tool has a Graphical User Interface (GUI) to visualize the results of parallelization obtained by using the tool. The GUI provides functionalities to input the sequential source files, include the libraries required by the program, compile the parallelized code and benchmark it against the original compiled code. Figure 5 shows marking of different tasks in the source code. The GUI also graphically represents the control flow graph, marked parallelizable blocks, TDM and caller graphs etc. Figure 6 shows the caller graph for a function.

![Figure 5 - Marking of different tasks](image)

![Figure 6 - Caller graph of a function](image)

S2P PERFORMANCE RESULTS

The testing of current S2P tool version is done on a Quad-core machine ((Intel Quad-Core Q6000 Series, Cache: 8 MB Cache, Clock speed: 2.4 GHz). Table 1 below shows the execution time of serial code and parallel code on 1, 2, 3, and 4 cores for 2 test codes. The first test code is algorithm for travelling salesman problem (TSP) and second is implementation of MP3 decoder. The performance depends on the dependencies in the code as well as the availability of the cores at the execution time. We have achieved improvements from 7% to 29% on various test codes. Figure 7 shows a screenshot of the S2P performance benchmarking result.

![Table 1. Execution Time for Serial and Parallel Code](image)
The test codes used for the performance testing of the S2P tool i.e. the TSP code and the MP3 code when parallelized manually took 10 days and 30 days respectively. Person averagely skilled at writing parallel code did the manual analysis and parallelization of the mentioned codes. As opposed to the manual parallelization, the TSP and MP3 codes were parallelized completely automatically by using the S2P tool in 10 minutes and 30 minutes respectively. The generated output code is an error-free parallelized source code. If the user is not satisfied with the performance of the S2P parallelized code he/she can further optimize it manually. This process would be much faster as opposed to the complete manual approach.

The S2P tool is offered for evaluation to interested readers/researchers by mailing to the authors of this paper.

**When to use automatic parallelization tools**

1. When the code is huge – example 200+ files and 250000 lines of code
2. When the application and its domain is not known
3. When the application to be parallelized is not a real time application
4. When applications is data intensive, examples computational fluid dynamics (CFD) code or crash simulation or chemical simulation code that have huge execution time and abundant loops for parallelization.
5. When getting last bit of the performance is not important but getting some performance and quick parallelization is a fair enough achievement.

Semi-automatic parallelization is useful and used very frequently used. For example, OpenMP is very widely used for parallelizing existing sequential codes. OpenMP does not help in parallel code identification or code transformation but it helps to generate parallelized code when which executed creates multiple threads. OpenMP is particularly useful for loop level parallelism but the current version of OpenMP 3.0 includes constructs for task level parallelism also. It is an extension to languages like C or C++ and provides constructs or APIs that can be inserted into the original sequential code. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. Simple example of an OpenMP code is as follows:

```c
1: int main()
2: {
3:    const int N = 20000;
4:    int i, a[N];
5:    #pragma omp parallel for
6:    for (i = 0; i < N; i++)
7:       a[i] = 2 * i;
8:    return 0;
9: }
```

In the code example, the code at line number 5 is the OpenMP construct added to the original sequential code. This code when executed will spawn number of threads to execute the for loop section in parallel. The execution of OpenMP constructs results in a fork-join of threads, as shown in figure 8. MPI uses a similar approach but it spawns new processes instead of threads and is used in a cluster or distributed memory execution environment, whereas, OpenMP is most suited for shared memory target architecture.

**When to use semi-automatic parallelization**

1. When the code is medium sized and manual code analysis is possible. Example 6 to 10 files and 2500 lines of code
2. When the application and its domain is known or code has data level parallelism

![Table showing performance of TSP and MP3 codes](image)

<table>
<thead>
<tr>
<th>Num of Cores</th>
<th>TSP Code (1 file, LOC: 602)</th>
<th>MP3 Decoder Code (7 files, LOC: 4933)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Execution Time (sec)</td>
<td>Execution Time (sec) for Parallelized Code</td>
<td>Serial Execution Time (sec) for Parallelized Code</td>
</tr>
<tr>
<td>1</td>
<td>21.28</td>
<td>48.00</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>

![Figure 7 - Screenshot of S2P tool showing performance of 24%](image)

The test codes used for the performance testing of the S2P tool i.e. the TSP code and the MP3 code when parallelized manually took 10 days and 30 days respectively. Person averagely skilled at writing parallel code did the manual analysis and parallelization of the mentioned codes. As opposed to the manual parallelization, the TSP and MP3 codes were parallelized completely automatically by using the S2P tool in 10 minutes and 30 minutes respectively. The generated output code is an error-free parallelized source code. If the user is not satisfied with the performance of the S2P parallelized code he/she can further optimize it manually. This process would be much faster as opposed to the complete manual approach.

The S2P tool is offered for evaluation to interested readers/researchers by mailing to the authors of this paper.

![Figure 8 – Fork-Join execution mode in OpenMP](image)
3. When the application to be parallelized is real time application and manual intelligence or tweaking of code may increase parallelization chances or increase the parallel program performance.

4. When getting last bit of the performance is important and the programmer who is parallelizing has good expertise in parallel programming and knows its issues in-and-out.

With our personal experience, our opinion is that one should not go for manual parallelization since it is error-prone, time consuming. It also requires high level of expertise in the application domain along with thorough knowledge of parallelization.

IV. CLASSIFICATION – BASED ON TARGET HARDWARE/ENVIRONMENT

Many parallelization tools or frameworks specifically target a particular hardware or environment for executing the parallelized code. For example, a tool may generate parallel code based on OpenMP, thus the target machine on which this parallelized code would execute the best would be shared-memory architecture. In this section, we present different tools or frameworks that generate architecture or target specific code. Figure 10 shows this classification.

Embedded applications target Asynchronous Multi-Processors (AMPs) architectures generally consisting of a RISC and DSPs working together. Tools that generate parallel code for such target architectures are generally semi-automated. They involve some kind of a graphical user interface, which allows users to specify the application in a graphical form. The graphical information is converted into Data Flow Graphs (DFGs) [17] and can be easily parallelized.

One such interesting tool is the National Instrument’s LabVIEW [18] compiler. LabVIEW is a graphical programming environment used to develop measurement, test, and control systems using intuitive graphical icons and wires that resemble a flowchart. It is useful tool to perform advanced analysis and data visualization. Since it is a simulation tool, it is scalable across multiple targets and operating systems. Programming in LabVIEW involves creating graphical code (G) that resembles a flowchart, which is significantly different than programming in traditional sequential languages. Instead of writing a sequence of commands that execute one-by-one, LabVIEW programs contain variables and operations that connect one variable to the next – the LabVIEW compiler automatically determines the order of commands to execute to produce correct results. This also means that, with LabVIEW, when two parallel sections of code are independent of each other, they can run at the same time on different cores of a multicore processor.

To visualize this, consider the simple arithmetic program shown in Figure 9. LabVIEW can recognize that both the multiply and add functions and the subtract function can execute at the same time; they do not depend on each other to execute.

The LabVIEW compiler identifies many different pieces of parallel code on the block diagram and maps those pieces onto a fixed number of threads during run time. This prevents manually handling of thread details and avoiding creating too many threads, which can affect performance. LabVIEW has many different features discussing which are out of scope of this paper.

When to use target architecture specific tools –

1. When we have a hardware platform specifically designed for a class of applications

2. When we want to extract the parallelization performance to the fullest – by considering all special hardware capabilities

3. When application can be re-written or is smaller few 100 lines of code

4. When application is data intensive and can exploit SIMD benefits.

V. CLASSIFICATION – BASED ON SOFTWARE DESIGN/IMPLEMENTATION

Parallelization tools are classified according to the approach taken while designing or implementing parallel applications. This approach is generally applicable when sequential application code is to be completely re-written or new parallel applications are to be written. Parallel applications can be written using any of the following models –

- Message passing
- Shared memory
- Data parallelism
- Tasks and channels
- Functional programming
In this section, we provide a brief comparison of various tools based on certain criteria that should be considered while taking a parallel programming approach or while choosing the right kind of tool for parallel programming. Table 2 shows this comparison.

VI. TOOLS COMPARISON SUMMARY

In this section, we provide a brief comparison of various tools based on certain criteria that should be considered while taking a parallel programming approach or while choosing the right kind of tool for parallel programming. Table 2 shows this comparison.

VII. CONCLUSION

Many tools, frameworks, libraries and language extensions are available today for parallelization. The programmers have to identify and select the right kind of parallelization strategy and tool to get performance benefits from the parallel hardware being available in the form of multicore or GPGPUs. In this paper, we briefly discussed few tools and presented a classification and comparison of different approaches. We hope some of our tips will be useful to the readers/programmers to identify the right kind of tools for parallelization of the applications under consideration.

ACKNOWLEDGMENT

We would like to thank all S2P team members for their inputs and efforts to implement this tool.

REFERENCES


multiprocessor performance with the SUIF compiler”, IEEE Computer, August 1996


Table 2 – Comparison of Parallelization Tools

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Manual</th>
<th>OpenMP</th>
<th>CUDA (NVidia)</th>
<th>LabVIEW</th>
<th>SUIF</th>
<th>Haskell</th>
<th>S2P Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special Hardware needed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Need to re-write code</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>YES</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Manual Code Analysis Required</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>NA</td>
<td>No</td>
<td>NA</td>
</tr>
<tr>
<td>Manual Data Dependency checks</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Sometimes</td>
<td>No</td>
<td>NA</td>
</tr>
<tr>
<td>Need of User to handle data</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Sometimes</td>
<td>No</td>
<td>NA</td>
</tr>
<tr>
<td>Learning new programming constructs</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time and efforts</td>
<td>HIGH</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>HIGH</td>
</tr>
<tr>
<td>Suitable for parallelism</td>
<td>All</td>
<td>Data level</td>
<td>Block level</td>
<td>Data level</td>
<td>Data level</td>
<td>Block level</td>
<td>Data and Block level</td>
</tr>
<tr>
<td>Suitable for Architecture</td>
<td>All</td>
<td>Shared Memory</td>
<td>Distributed Memory</td>
<td>Shared Memory</td>
<td>Embedded or AMPs</td>
<td>Shared memory</td>
<td>Share memory</td>
</tr>
<tr>
<td>Type of tool</td>
<td>NA</td>
<td>Semi-automated</td>
<td>Semi-automated</td>
<td>No</td>
<td>Semi-automated</td>
<td>Fully automated</td>
<td>NA</td>
</tr>
</tbody>
</table>