Identifying Parallel Patterns in C++ Codes

David del Rio Astorga,[1] Manuel F. Dolz,[1] Luis Miguel Sanchez,[1]
J. Daniel García,[1] Marco Danelutto,[2] Massimo Torquati[2]

[1]Department of Computer Science, University Carlos III of Madrid, 28911–Leganés, Spain.
[2]Department of Computer Science, University of Pisa, 56127–Pisa, Italy.

ABSTRACT

Introduction. Since free performance lunch of processors is over, parallelism has become the new trend in hardware design and software development. Nevertheless, it remains a large portion of production software that is still developed in sequential. Taking into account that these software modules contain millions of code lines, the effort needed to identify parallel regions is extremely high. In this direction, we present Parallel Pattern Analyzer Tool (PPAT), a software component that aids discovering and annotating parallel patterns in source code. This tool eases the transformation from sequential source code into parallel.

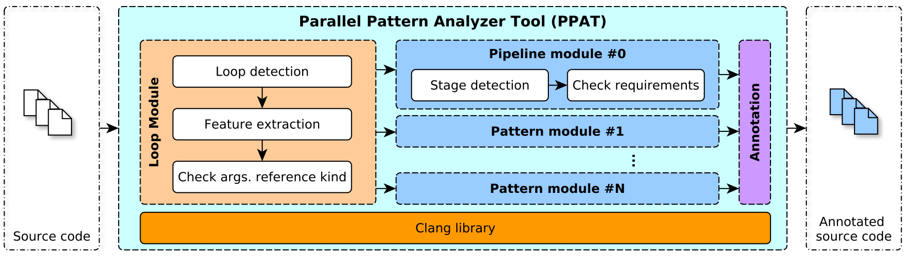

FIG. 1: Workflow diagram of PPAT.

Workflow. Figure 1 depicts the general workflow diagram of PPAT. This tool takes advantage of the Clang library to generate the Abstract Syntax Tree (AST) and walk through it in order to collect relevant information about the source code and identify parallel patterns. First, the tool receives the sequential source code files that should be analyzed. Next, the following steps are executed: *i) Loop detection*, *ii) Feature extraction* and *iii) Check arguments reference kind*. Finally, marked loops are passed to the different pattern analyzer modules, i.e., `Pipeline`, `Farm` and `Map` (three well-known parallel patterns). In the last stage, parallel pattern modules are annotated by using RePhrase attributes syntax. The produced code will be refactored into parallel code.

Evaluation. We perform an experimental evaluation of PPAT to analyze how many loops of the benchmarks `Rodinia`, `NAS` Parallel Benchmarks (NPB) can be refactored into parallel patterns. Our evaluation methodology is based on a comparison between a manual inspection and an automatic one, using PPAT, of the loops appearing in the benchmark codes. The experimental evaluation demonstrates that PPAT is able to obtain similar performance results as the "handmade" parallel versions of the benchmark suites tested. Therefore, reducing the human effort in transforming sequential codes into parallel code.

(a) `Rodinia`

		PPAT			Manual		
Test	Loops	P	F	M	P	F	M
`b+tree`	80	3	7	7	2	7	7
`particlefilter`	44	1	8	8	1	10	10
`BFS`	7	0	1	1	0	2	1
`nw`	12	0	6	6	0	6	6
`cfd`	78	16	12	12	15	13	13
`lavamd`	10	0	1	1	0	2	2
`heartwall`	54	1	4	2	0	4	3
`nn`	2	0	0	0	0	0	0
`Backprop`	28	0	2	2	0	5	5

(b) `NAS`

		PPAT			Manual		
Test	Loops	P	F	M	P	F	M
`IS`	16	1	8	8	0	9	9
`LU`	187	1	37	37	1	81	81
`FT`	41	0	7	7	3	20	20
`EP`	8	1	2	2	0	3	3
`MG`	80	1	26	26	1	44	44
`UA`	321	3	116	116	2	171	170
`DC`	30	2	5	5	1	7	7
`SP`	250	1	51	51	1	103	103
`BT`	181	1	46	46	1	78	78

TABLE I: Results for the `Rodinia` and `NAS` benchmark suites. P, F and M stand for the number of `Pipeline`, `Farm` and `Map` patterns detected, respectively.

Conclusions. We presented PPAT, a tool that: *i)* is completely independent of the refactoring tool used, since it identifies parallel patterns; *ii)* performs a static analysis and avoids the use of profiling techniques, and thus, it becomes much faster than other approaches; and *iii)* guarantees that parallel patterns detected comply with a series of requirements that ensure the correctness of the parallelization.