DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities

Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf

Abstract The stagnation of single-core performance leaves application developers with software parallelism as the only option to further benefit from Moore's Law. However, in view of the complexity of writing parallel programs, the parallelization of myriads of sequential legacy programs presents a serious economic challenge. A key task in this process is the identification of suitable parallelization targets in the source code. We developed a tool called DiscoPoP showing how dependency profiling can be used to automatically identify potential parallelism in sequential programs. Our method is based on the notion of computational units, which are small sections of code following the read-compute-write pattern that can form the atoms of concurrent scheduling. DiscoPoP covers both loop and task parallelism. Experimental results show that reasonable speedups can be achieved by parallelizing sequential programs manually according to our findings. By comparing our findings to known parallel implementations of sequential programs, we demonstrate that we are able to detect the most important code locations to be parallelized.

1 Introduction

Although the component density of microprocessors is still rising according to Moores Law, single-core performance is stagnating for more than ten years now. As a consequence, extra transistors are invested into the replication of cores, resulting in the multi- and many-core architectures popular today. The only way for developers to take advantage of this trend if they want to speed up an individual application is to match the replicated hardware with thread-level parallelism. This, however, is often challenging especially if the sequential version was written by someone else. Unfortunately, in many organizations the latter is more the rule than

German Research School for Simulation Sciences, Aachen, Germany
RWTH Aachen University, Aachen, Germany
e-mail: {z.li,a.jannesari,f.wolf}@grs-sim.de

the exception [1]. To find an entry point for the parallelization of an organization's application portfolio and lower the barrier to sustainable performance improvement, tools are needed that identify the most promising parallelization targets in the source code. These would not only reduce the required manual effort but also provide a psychological incentive for developers to get started and a structure for managers along which they can orchestrate parallelization workflows.

In this paper, we present an approach for the discovery of potential parallelism in sequential programs that—to the best of our knowledge—is the first one to combine the following elements in a single tool:

1. Detection of available parallelism with high accuracy
2. Identification of code sections that can run in parallel, supporting the definition of parallel tasks—even if they are scattered across the code
3. Ranking of parallelization opportunities to draw attention to the most promising parallelization targets
4. Time and memory overhead that is low enough to deal with input programs of realistic size

Our tool, which we call DiscoPoP (= Discovery of Potential Parallelism), reverses the idea of data-race detectors. It profiles dependencies, but instead of only reporting their violation it also watches out for their absence. We use the dependency information to represent program execution as a graph, from which parallelization opportunities can be easily derived or based on which their absence can be explained. Since we track dependencies across the entire program execution, we can find parallel tasks even if they are widely distributed across the program or not properly embedded in language constructs, fulfilling the second requirement. To meet the third requirement, our ranking method considers a combination of execution-time coverage, critical-path length, and available concurrency. Together, these four properties bring our approach closer to what a user needs than alternative methods [2, 3, 4] do. We expand on earlier work [5], which introduced the algorithm for building the dependency graph based on the notion of computational units—at that time a purely dynamic approach with significant time and memory overhead. This is why this paper concentrates mainly on the overall workflow, including a preceding static analysis, the minimization of runtime overhead, the ranking algorithm, and an evaluation using realistic examples along with a demonstration of identifying non-obvious tasks.

The remainder of the paper is structured as follows: In the next section, we review related work and highlight the most important differences to our own. In Section 3, we explain our approach in more detail. In the evaluation in Section 4, we run the NAS parallel benchmarks [6], a collection of programs derived from real CFD codes, to analyze the accuracy at which we identify and rank parallelism in spite of the optimizations we apply. Also, we show how we find parallel tasks and pipeline patterns that are not trivial to spot. Finally, we quantify the overhead of our tool both in terms of time and memory. Section 5 summarizes our results and discusses further improvements.

2 Related Work

After purely static approaches including auto-parallelizing compilers had turned out to be too conservative for the parallelization of general-purpose programs, a range of predominantly dynamic approaches emerged. Such dynamic approaches can be broadly divided into two categories. Tools in the first merely count dependencies, whereas tools in the second, including our own, exploit explicit dependency information to provide detailed feedback on parallelization opportunities or obstacles.

Kremlin [4] belongs to the first category. Using dependency information, it determines the length of the critical path in a given code region. Based on this knowledge, it calculates a metric called self-parallelism, which quantifies the parallelism of a code region. Kremlin ranks code regions according to this metric. Alchemist [3] follows a similar strategy. Built on top of Valgrind, it calculates the number of instructions and the number of violating read-after-write (RAW) dependencies across all program constructs. If the number of instructions of a construct is high while the number of RAW dependencies is low, it is considered to be a good candidate for parallelization. In comparison to our own approach, both Kremlin and Alchemist have two major disadvantages: First, they discover parallelism only at the level of language constructs, that is, between two predefined points in the code, potentially ignoring parallel tasks not well aligned with the source-code structure. Second, they merely quantify parallelism but do neither identify the tasks to run in parallel unless it is trivial as in loops nor do they point out parallelization obstacles.

Like DiscoPoP, Parwiz [2] belongs to the second category. It records data dependencies and attaches them to the nodes of the execution tree (i.e., a generalized call tree that also includes basic blocks) it maintains. In comparison to DiscoPoP, Parwiz lacks a ranking mechanism and does not explicitly identify tasks. They have to be manually derived from the dependency graph, which is demonstrated using small text-book examples.

Reducing the significant space overhead of tracing memory accesses was also successfully pursued in SD3 [7]. An essential idea that arose form there is the dynamic compression of strided accesses using a finite state machine. Obviously, this approach trades time for space. In contrast to SD3, DiscoPoP leverages an acceptable approximation, sacrificing a negligible amount of accuracy instead of time. The work from Moseley et al [8] is a representative example of this approach. Sampling also falls into this category.

Prospector [9] is a parallelism-discovery tool based on SD3. It tells whether a loop can be parallelized, and provides a detailed dependency analysis of the loop body. It also tries to find pipeline parallelism in loops. However, no evaluation result or example is given for this feature.

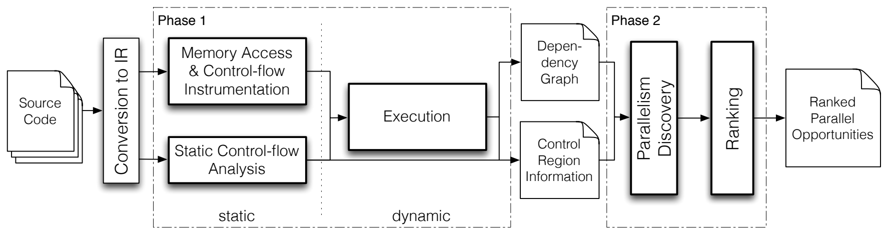
Phase 1
Source Code
Conversion to IR
Memory Access & Control-flow Instrumentation
Static Control-flow Analysis
Execution
static
dynamic
Dependency Graph
Control Region Information
Phase 2
Parallelism Discovery
Ranking
Ranked Parallel Opportunities

Fig. 1 The work flow of DiscoPoP.

3 Approach

Figure 1 shows the work flow of DiscoPoP. The work flow of DiscoPoP is divided into two phases: In the first phase, we instrument the target program and execute it. Control flow information and data dependencies are obtained in this phase. In the second phase, we build computational units (CUs) for the target program, and search for potential parallelism based on the CUs and dependence among them. The output is a list of parallelization opportunities, consisting of several code sections that may run in parallel. These opportunities are also ranked to allow the users focus on the most interesting opportunities.

3.1 Dependence profiling

Data dependences can be obtained in two major ways: static and dynamic analysis. Static approaches determine data dependences without executing the program. Although they are fast and even allow fully automatic parallelization in some restricted cases [10, 11], they lack the ability to track dynamically allocated memory, pointers, and dynamically calculated array indices, which usually makes their assessment too pessimistic for practical purposes. In contrast, dynamic dependence profiling captures only those dependences that actually occur at runtime. Although dependence profiling is inherently input sensitive, the results are still useful in many situations, which is why such profiling forms the basis of many program analysis tools [4, 9, 2]. Besides, input sensitivity can be addressed by running the target program with changing inputs and computing the union of all collected dependences.

Dependence profiling component severs as the foundation of our tool. The profiler produces the following information:

- pair-wise data dependencies
- source code locations of dependencies and the names of the variables involved
- runtime control-flow information

```
1:60 BGN loop
1:60 NOM {RAW 1:vim} {WAR 1:60|i} {INIT *}
1:63 NOM {RAW 1:59|temp1} {RAW 1:67|temp1}
1:64 NOM {RAW 1:60|i}
1:65 NOM {RAW 1:59|temp1} {RAW 1:67|temp1} {WAR 1:67|temp2}
    {INIT *}
1:66 NOM {RAW 1:59|temp1} {RAW 1:65|temp2} {RAW 1:67|temp1}
    {INIT *}
1:67 NOM {RAW 1:65|temp2} {WAR 1:66|temp1}
1:70 NOM {RAW 1:67|temp1} {INIT *}
1:74 NOM {RAW 1:41|block}
1:74 END loop 1200
```

Fig. 2 A fragment of profiled data dependencies in a sequential program.

We profile detailed pair-wise data dependencies because we do not want to lose the chance to report root causes that preventing parallelism. If detailed information is not required by a certain analysis, dependencies can be easily merged into coarser grain with the help of control-flow and variable name information, for example, dependencies between loops, between functions, or between objects. Control-flow information is necessary for building computational units.

A fragment of profiled data is shown in Figure 2. A data dependency is represented as a triple `<sink, type, source>`. `type` is the dependency type (RAW, WAR or WAW). Note that a special type `INIT` represents the first write operation to a memory address. In this case, `source` of the dependency is empty, which is represented as `*`.

`sink` and `source` are the source code locations of the latter and the former memory accesses, respectively. `sink` is further represented as a pair `<fileID:lineID>`, while `source` is represented as a triple `<fileID:lineID|variableName>`. As it is shown in Figure 2, data dependences with the same `sink` are aggregated together.

Identifier `NOM` (short for "NORMAL") means that the source line specified by aggregated `sink` has no control-flow information. Otherwise, `BGN` and `END` represent the entry and exit point of a control region, respectively. In Figure 2, a loop starts at source line 1:60 and ends at source line 1:74. The number following `END loop` shows the actual number of iterations executed, which is 1200 in this case.

In order to get pair-wise data dependencies dynamically, every load and store instruction is instrumented. Entry and exit points of control regions are determined statically, but loops are still instrumented so that the number of executed iterations can be recorded. Source code location and variable names are obtained with the help of debug symbols, thus the compiler option `-g` must be specified to compile the program.

```
 1 void netlist:: (netlist_elem** a, netlist_elem** b, Rng* rng)
 2 {
 3    //get a random element
 4    long id_a = rng->rand(_chip_size);
 5    netlist_elem* elem_a = &(_elements[id_a]);
 6
 7    //now do the same for b
 8    long id_b = rng->rand(_chip_size);
 9    netlist_elem* elem_b = &(_elements[id_b]);
10
11    while (id_b == id_a)
12    {
13       id_b = rng->rand(_chip_size);
14       elem_b = &(_elements[id_b]);
15    }
16    *a = elem_a;
17    *b = elem_b;
18    return;
19 }
```

Fig. 3 Function `netlist::get_random_pair` of *parsec.canneal* that contains two CUs.

3.2 Computational unit

During the second phase, we search for potential parallelism based on the output of the first phase, which is essentially a graph of dependencies between source lines. This graph is then transformed into another graph, whose nodes are parts of the code without parallelism-preventing read-after-write (RAW) dependencies inside. We call these nodes *computational units* (CUs). Based on this CU graph, we can detect potential parallelism and already identify tasks that can run in parallel.

A Computational Unit (CU) is defined as a set of instructions that form a *read-compute-write* pattern. A CU differs from the basic block such that a basic block contains operations that are consecutive and has only one entry and one exit point. A CU however, is a group of instructions that are not necessarily consecutive but perform a computation and is based on the use of a set of variables. A single CU or a group of CUs merged together can provide the code sections that perform a task. These code sections can be examined to see if they can be run concurrently with other code sections or themselves to exploit the available parallelism.

To understand what a CU is, consider the example in the listing 3. The source lines 4 and 8 perform the initialization of variables *id_a* and *id_b* with a random value. Lines 5 and 9 perform the task of calculating *elem_a* and *elem_b* by using the value of *id_a* and *id_b* respectively. These two operations are performed independent of one another. The *while* loop in the function is responsible for checking if the two random values are equal and reassigning *id_b* if it is true. Finally, the lines 16 and 17 are responsible for writing the final computation back to **a* and **b*. In essence, the group of LLVM-IR instructions corresponding to the lines {4, 5, 16}

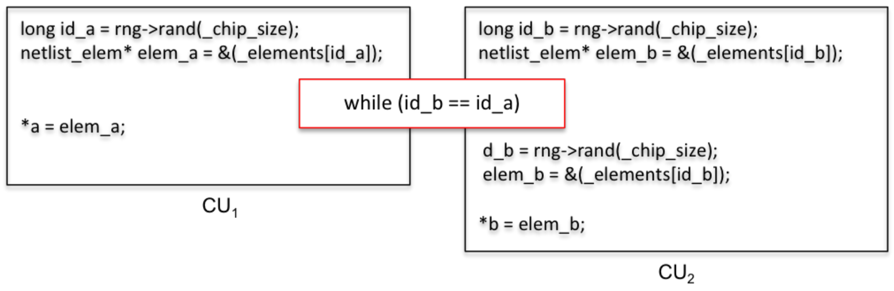
long id_a = rng->rand(_chip_size);
netlist_elem* elem_a = &(_elements[id_a]);
*a = elem_a;
while (id_b == id_a)
long id_b = rng->rand(_chip_size);
netlist_elem* elem_b = &(_elements[id_b]);
d_b = rng->rand(_chip_size);
elem_b = &(_elements[id_b]);
*b = elem_b;
CU1
CU2

Fig. 4 The two CUs contained in Function `netlist::get_random_pair` of *parsec.canneal*.

perform one computation and the ones corresponding to the lines {8, 9, 13, 14, 17} perform another computation and these tasks are independent of each other except for the equality check condition of the *while* loop. In case of the *while* loop, identifying code sections that that require synchronization or replication across concurrent threads will part of our future work.

The two computations mentioned above follow a basic rule where a variable or a group of variables are read and then they are used to perform another calculation. This is followed by the final state being written to another variable as a store operation. Hence, these two computations can be said to follow a *read-compute-write* pattern. The two computations can be visualized as seen in the figure 4. The final *store* instruction that writes a value to **a* uses all the instructions that correspond to the lines {4, 5, 16} to perform that write operation. Similarly, the group of instructions that correspond to the lines {8, 9, 13, 14, 17} are used for the final *store* instruction that writes **b*. These two sets of instructions can individually be defined as CUs. These CUs form the building blocks of the tasks which can be created for exploiting parallelism in the sequential programs.

Using the common instructions and the RAW dependencies between the CUs, a CU graph is constructed. The nodes of this graph are CU IDs. The CU graph has two types of edges. First type of edge between any two CU nodes signifies RAW dependence between them and is a directed edge. The weight of an RAW edge is the number of RAW dependencies between the two CUs. The second type of edge signifies that there are common instructions between the two CUs. This is an undirected edge and its weight is the number of common instructions between the two CUs. Figure 5(a) shows a CU graph with red edges as RAW dependencies between the CUs and blue edges as the CUs connected because of the common instructions between them. A CU graph is generally a disconnected graph with several connected components.

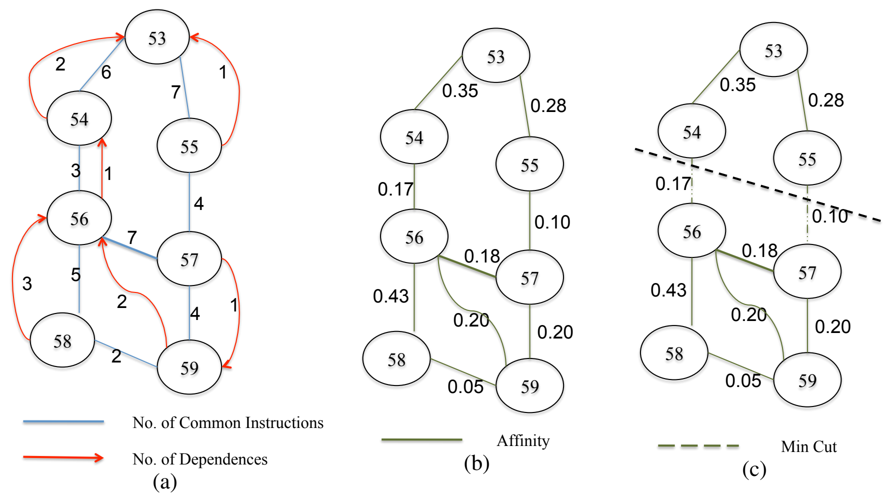

Fig. 5 Example of a CU Graph and Task Formation.

3.3 *Detecting parallelism*

DiscoPoP finds potential parallelism based on the CU graph. It is well known that among the four kinds of data dependencies, read-after-read (RAR) does not affect parallelization. Write-after-read (WAR) and write-after-write (WAW) are easy to resolve by privatizing the affected variables. Only read-after-write (RAW) seriously prevents parallelization.

3.3.1 DOALL loops

A loop can be parallelized according to the do-all pattern if there are no loop-carried or inter-iteration dependences. A forward or self-dependence is always loop-carried, as the control flow within a loop iteration moves in a forward direction, which is why dependences within the same iteration must point backward. Note that inner loops in loop nests, which may reverse the control flow direction whenever a new inner iteration starts, are treated separately. The absence of forward or self-dependences is easy to verify based on the graph matrix, whose upper triangle shows all forward and self-edges.

Of course, there may also exist loop-carried dependences in backward edges of the graph matrix. However, to reliably distinguish them from intra-iteration dependences, our dependence profiler would have to record the iteration number along with each memory access, substantially increasing its memory overhead. On the other hand, loop-carried dependences in a backward direction that are not accompanied by dependences in a forward direction in the same loop are very rare. Basically,

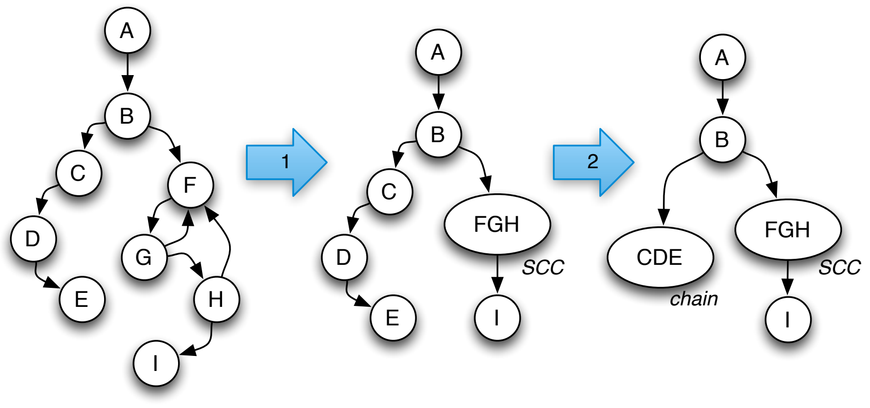

Fig. 6 Forming tasks in a code section of fluidanimate.

the absence of forward and self-dependences in a loop is a good indicator of the absence of loop-carried dependences. For this reason, we decided to refrain from the costly classification of backward dependences into loop-carried or not loop-carried.

3.3.2 Tasking

To identify potential parallel tasks, we firstly merge CUs contained in *strongly connected components* (SCCs) or in *chains*. In graph theory, an SCC is a subgraph in which every vertex is reachable from every other vertex. Thus, every CU in an SCC of the CU graph depends on every other CU either directly or indirectly, forming a complex knot of dependences that is likely to defy internal parallelization. Identifying SCCs is important for two reasons:

1. Algorithm design. Complex dependences are usually the result of highly optimized sequential algorithm design oblivious of potential parallelization. In this case, breaking such dependence requires a parallel algorithm, which is beyond the scope of our method.
2. Coding effort. Even if such complex dependences are not created by design, breaking them is usually time-consuming, error-prone, and may cause significant synchronization overhead that may outweigh the benefit of parallelization.

Hence, we hide complex dependences inside SSCs, exposing parallelization opportunities outside, where only a few dependences need to be considered. Figure 6 shows the task-forming process for a code section (starting from serial.c: 341) in function `RebuildGrid()` of fluidanimate, a program from the PARSEC Benchmark Suite [12]. In step 1, CU F, G and H are grouped into SCC_{FGH}. After contracting each SCC to a single vertex, the graph becomes a directed acyclic graph.

Moreover, we group CUs that are connected in a row without a branching or joining point in between into a *chain* of CU. Although they do not form an SCC, we still group them together since each CU contains only a few instructions, and there is no benefit in considering each CU as a separate task. In step 2 of Figure 6, CU C, D and E are grouped into the $chain_{CDE}$. Finally, we declare each SCC or chain a potential task and derive a parallelization plan from the dependences that exist between them.

After the process of forming chains and SCCs, we can suggest some task parallelism between independent chains and SCCs, that is, without RAW dependencies between them. Note that a chain of CUs may start and end anywhere in the program, without the limitation of predefined constructs, and the code in a chain of CUs does not need to be continuous.

However, some task parallelism can also be utilized with a small amount of refactoring effort, that is, dependences between potential tasks exist but are weak. We cover these parallelism by applying *minimum cut* on CU graph. In CU graph, a high value of weight on the edges of any two vertices indicates that those two CUs either share large amount of computation or they are strongly dependent on one another. Using these two metrics, we calculate a value called *affinity* for every pair of CU nodes in the graph. The affinity between any two CU nodes hence indicates how tightly coupled the two CUs are. A low value of affinity between two CUs signifies that it's logical to separate the two CUs while forming tasks. The two types of edges in the graph are replaced by a single undirected edge. The weight of this edge is the affinity between the two CUs. Figure 5(b) demonstrates the graph with the two types of edges between the vertices replaced by single edge with affinity as the weight.

The next step is to calculate the minimum cut of a connected component using Stoer-Wagner's algorithm [13]. In graph theory, a *cut* of a graph is a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge. A *minimum cut* is a set of edges that has the smallest number of edges (for an unweighted graph) or smallest sum of weights possible (for a weighted graph). A minimum cut creates a disconnected graph with two connected components, each of which is further analyzed for finding relevant tasks. Figure 5(c) shows the CU graph with a minimum cut.

Identifying the minimum cut of a graph divides the graph into two components that were weakly linked. This indicates that we are separating our code with minimum number of dependences and common instructions affected. For each component, the minimum cut is calculated further to divide it into two more components. The process is repeated recursively over all the components of the CU graph until the components available are CUs themselves.

3.3.3 Pipeline

To detect pipeline pattern, we use *template-matching* [14] technique. There, both template and target program are represented by vectors. Cross-correlation between two vectors is used to determine how similar they are. We adapt this concept for the detection of parallel patterns in CU graphs.

```
$tree = getExecutionTree(serialProgram)$
$Hotspots = findHotspots(tree)$
**for** ***each*** $h$ ***in*** $Hotspots$ **do**
    $CUGraph = getCUGraph(h)$
    $n = getNumberOfCUs(CUGraph)$
    **for** ***each*** $p$ ***in*** $ParallelPatterns$ **do**
        $\mathbf{p} = getPatternVector(p, n)$
        $\mathbf{g} = getGraphVector(p, CUGraph)$
        $CorrCoef[h, p] = CorrCoef(\mathbf{p}, \mathbf{g})$
    **end**
**end**
$return\ CorrCoef$
```

Algorithm 1: Parallel pattern detection.

Algorithm 1 shows the overall work flow of our approach. We first look for hotspots in the input program—sections such as loops or functions that have to shoulder most of the workload. For each hotspot and pattern, we then create a pattern vector **p**, whose length is equal to the number *n* of CUs in the hotspot. The pattern vector plays the role of the template to be matched to the program. After that, we create the pattern-specific graph vector **g** of the hotspot's CU subgraph, which represents the part of the program to which the pattern vector is matched. Vectors **p** and **g** are derived from adjacency matrices reflecting dependences in the pattern and in the CU graph, respectively. As a next step, we compute the correlation coefficient of the two vectors using the following formula:

$$CorrCoef = \frac{\mathbf{p} \cdot \mathbf{g}}{||\mathbf{p}||\ ||\mathbf{g}||}$$

The correlation coefficient of the pattern vector and the graph vector of the selected section tells us whether the pattern exists in the selected section or not. The value of the coefficient is always in the range of $[0, 1]$. A 1 indicates that the pattern exists fully, whereas a 0 indicates that it does not exist at all. A value in between shows the pattern can exist but with some limitations which we need to work around. Our tool points out to the dependences, which cause the value of correlation coefficient of a pattern to be less than 1. This helps the programmer to resolve these specific dependences, if he wants to implement that pattern.

Implementing a pipeline only makes sense if its stages are executed many times. For this reason, we restrict our search for pipelines to loops, functions with multiple loops, and recursions. In order to find a pipeline, we first let DiscoPoP deliver the CU graphs of all hotspots. Because DiscoPoP counts the number of read and write instructions executed in each loop or function, we currently use this readily available metric as an approximation of the workload when searching for hotspots.

Table 1 Detection of parallelizable loops in NAS Parallel Benchmark programs.

Program	Executed		OpenMP-annotated loops			
	# loops	# parallelizable	# OMP	# identified	# in top 30%	# in top 10
BT	184	176	30	30	22	9
SP	252	231	34	34	26	9
LU	173	164	33	33	23	7
IS	25	20	11	8	2	2
EP	10	8	1	1	1	1
CG	32	21	16	9	5	5
MG	74	66	14	14	11	7
FT	37	34	8	7	6	5
Overall	787	720	147	136	96	45

A more comprehensive criterion, including execution times and workload, will be implemented in the future. We then compute an adjacency matrix for each hotspot graph, which we call the *graph matrix*. For each graph matrix, we create a corresponding pipeline pattern matrix of the same size, which we call the *pipeline matrix*. Pipeline matrices encode a very specific arrangement of dependences expected between CUs. For example, there must be a dependence chain running through all CUs in the graph because a pipeline consists of a chain of dependent stages. This specific property helps us to derive the pipeline pattern vector from the matrix.

4 Evaluation and Results

We conducted a range of experiments to evaluate the ability of DiscoPoP to detect DOALL loops, potential parallel tasks and pipeline patterns. The performance of DiscoPoP is also analyzed. Test cases are the NAS Parallel Benchmarks 3.3.1 [6] (NAS), a suite of programs derived from real-world computational fluid-dynamics applications, the Starbench parallel benchmark suite [15] (Starbench), which covers programs from diverse domains, including image processing, information security, machine learning and so on, benchmarks from PARSEC Benchmark Suite 3.0 [12], and a few real-world applications. Whenever possible, we tried different inputs to compensate for the input sensitivity of dynamic dependence profiling.

4.1 DOALL loops

To evaluate the ability of DiscoPoP to detect DOALL loops, we searched for parallelizable loops in sequential NPB programs and compared the results with the parallel versions provided by NPB. Table 1 shows the results of the experiment. The data listed in the column set ”Executed” are obtained dynamically. Column ”# loops” gives the total number of loops which were actually executed. The number of

Table 2 Parallel tasks identified in Starbench compared to existing parallel implementations.

Program	Task suggestion	Execution time (%)	Matched in parallel implementations	# CUs used
c-ray	`render_scanlines()`	100.0	yes	4
k-means	`cluster()`	99.6	yes	3
md5	`MD5_Update()`	93.5	yes	7
rotate	`RotateEngine::run()`	90.3	yes	6
rgbyuv	`processImage()`	100.0	yes	7
ray-rot	`render_scanlines()`	97.2	yes	10
rot-cc	`RotateEngine::run()`	54.7	yes	13

loops that we identified as parallelizable are listed under "# parallelizable". At this stage, prior to the ranking, DiscoPoP considers only data dependencies, which is why still many loops carrying no dependency but bearing only a negligible amount of work are reported. The second set of columns shows the number of annotated loops in OpenMP versions of the programs (# OMP). Under "# identified" we list how many annotated loops were identified as parallelizable by DiscoPoP.

As shown in Table 1, DiscoPoP identified 92.5% (136/147) of the annotated loops. A comparison with other tools is challenging because none of them is available for download. A comparison based exclusively on the literature has to account for differences in evaluation benchmarks and methods. For Parwiz [2], the authors reported an average of 86.5% after applying their tool to SPEC OMP-2001. Kremlin [4], which was also evaluated with NPB, selects only loops whose expected speedup is high. While Kremlin reported 55.0% of the loops annotated in NPB, the top 30% of DiscoPoP's ranked result list cover 65.3% (96/147).

4.2 Tasking

We have applied two separate strategies to evaluate the ability of DiscoPoP to detect parallel tasks. Firstly, we compare the parallel implementations of the applications from the Starbench benchmark suite with the tasks identified by our analysis. In this case, our goal is to verify if the approach identifies valuable and logical homogeneous tasks. Secondly, we use some of the programs from the PARSEC benchmark and parallelize these applications based on the heterogeneous tasks which are identified as potential candidates for parallelism.

4.2.1 Comparison with existing parallel implementations.

Our first evaluation strategy involves providing a comparison of the identified tasks with the existing parallel versions of the applications for Starbench parallel benchmark suite. The table 2 shows the overview of the evaluation performed. Column "Task suggestion" shows the location where parallel tasks are identified using our approach and column ”Matched in parallel implementations” shows whether the identified tasks exist in the official parallel implementations from Starbench. The tasks were identified by prioritizing the main algorithm functions and the functions that consumed the majority of the total execution time of the program as shown in column ”Execution time (%)”.

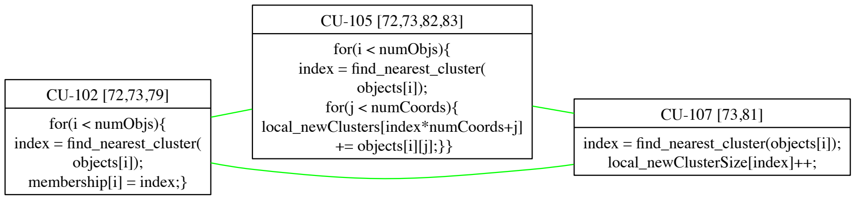

Fig. 7 Connected Component of the CU Graph of *k-means* corresponding to function `cluster()`.

As an example, we show the tasks identified in *k-means*, a clustering algorithm widely used in the domains of data-mining and artificial intelligence. The application consists of two iteratively repeated phases. One is a clustering phase and the other is a reduction phase that computes new clusters. In the sequential version, the function `kmeans()` calls the function `cluster()` which performs the clustering phase. The remaining body of the function `kmeans()` performs the reduction phase. The function `cluster()` takes 99.6% of the total execution time of the program. This makes it a good candidate for analysis of the tasks from the list of the tasks identified for the program.

The analysis identifies both of the aforementioned phases individually as tasks. The `cluster()` function is identified as a task by grouping 3 CUs from the CU graph. Figure 7 shows the connected component of the graph for the function `cluster()`. As for the reduction phase, only the part of the function `kmeans()` that performs this phase is identified as task by the analysis. In the pthreads version of the program, every thread executes the function `work()`, which contains the same code as the sequential version of `cluster()`. The reduction phase is run by the main thread thereafter.

4.2.2 Parallelization based on the suggested heterogeneous tasks.

In this section we investigate some applications of PARSEC benchmark suite. We parallelized these applications based on the tasks formed by using CUs or by directly considering CUs as tasks. We assigned these tasks to separate threads and calculated the speedup obtained. We parallelized these cases mainly using OpenMP `section` and `task` directives. Table 3 shows the results of the applications paral-

Table 3 Speedups obtained by parallelizing identified tasks in PARSEC benchmarks.

Program	Function	Code refactoring	# Threads	Local speedup
Fluidanimate	`RebuildGrid()`	Yes	2	1.60
Fluidanimate	`ProcessCollisions()`	No	4	1.81
Canneal	`routing_cost_given_loc()`	Yes	2	1.32
Blackscholes	`CNDF()`	NA	NA	NA

Table 4 Pipeline patterns identified in PARSEC benchmarks and libVorbis.

Program	# of pipeline in parallel version	Corr. coef.	# Detected	Speedup
bodytrack	1	0.96	1	N.A.
dedup	1	1.00	1	N.A.
ferret	1	1.00	1	N.A.
blackscholes	0	0.00	0	N.A.
fluidanimate	0	0.94	1	1.52 (3T)
libVorbis	N.A.	1.00	1	3.62 (4T)

lelized. The local speedups represent an average of five independent executions of the programs. Column ”# Threads” shows the number of threads used to parallelize the suggestions. Column ”Code refactoring” indicates if the refactoring the code like adding necessary synchronization or replicating some part of the code across multiple threads was necessary to parallelize the program based on the suggestion. For our future work, we would like to predict the various kinds of synchronizations or code refactoring necessary to parallelize a suggestion based on the available dependences and CUs identified.

4.3 Pipeline

We applied DiscoPoP on five benchmarks from PARSEC and libVorbis to detect pipeline pattern. Among the test cases, three benchmarks (*bodytrack*, *dedup*, and *ferret*) have pipeline patterns according to their existing parallel implementations, while two of the test cases (*blackscholes* and *fluidanimate*) do not. libVorbis has a natural pipeline work flow, but we cannot confirm it before applying DiscoPoP since there is no existing parallel implementation for it.

Table 4 shows the results of detecting pipeline patterns on the test cases. For all the three cases that contain pipeline patterns in parallel implementations, the utilized pipeline are successfully identified. Thus we did not parallelize them again. No pipeline pattern is detected in *blackscholes*, which is also as expected. However, pipeline pattern is detected in *fluidanimate*, which do not have a pipeline in its parallel implementations. After examining the code, we believe the parallelism does exist, and we parallelized the code section following the suggestion. Our parallelization yields a speedup of 1.52 using three threads. Note that the correlation coefficient for this pipeline pattern is 0.94, which implies that code refactoring may be needed. Actually, parallelizing this place requires quite a lot of effort.

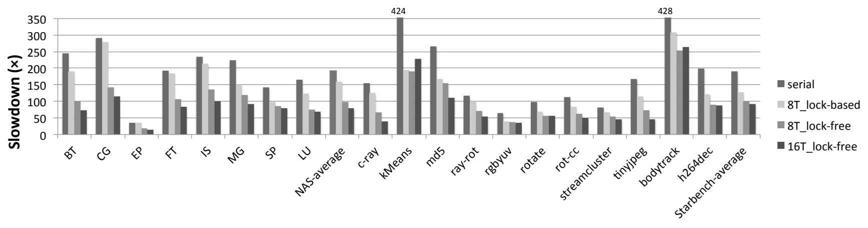

Fig. 8 Slowdowns of data dependence profiler for sequential NAS and Starbench benchmarks.

LibVorbis is a reference implementation of the Ogg Vorbis codec. It provides both a standard encoder and decoder for the Ogg Vorbis audio format. In this study, we analyzed the encoder part. The suggested pipeline resides in the body of the loop that starts at file encoder_example.c, line 212, which is inside the main function of the encoder. The pipeline contains only two stages: `vorbis_analysis()`, which applies some transformation to audio blocks according to the selected encoding mode (this process is called analysis), and the remaining part that actually encodes the audio block. After investigating the loop of the encoding part further, we found it to have two sub-stages: encoding and output.

We constructed a four-stage pipeline with one stage each for analysis, encoding, serialization, and output, respectively. We added a serialization stage, in which we reorder the audio blocks because we do not force audio blocks to be processed in order in the analysis and the encoding phase. We ran the test using a set of uncompressed wave files with different sizes, ranging from 4 MB to 47 MB. As a result, the parallel version achieved an average speedup of 3.62 with four threads.

4.4 Overhead

We conducted our performance experiments on a server with 2 x 8-core Intel Xeon E5-2650 2 GHz processors with 32 GB memory, running Ubuntu 12.04 (64-bit server edition). All the test programs were compiled with option `-g -O2` using Clang 3.3. For NAS, we used the input set W; for Starbench, we used the reference input set.

4.4.1 Time overhead.

First, we examine the time overhead of our profiler. The number of threads for profiling is set to 8 and 16. The slowdown figures are average values of three executions compared with the execution time of uninstrumented runs. The negligible time spent in the instrumentation is not included in the overhead. For NAS and Starbench, instrumentation was always done in two seconds.

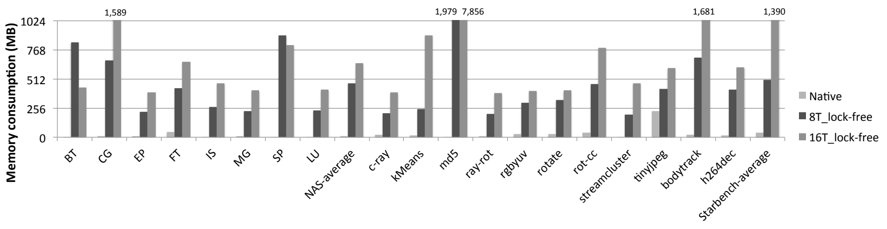

Fig. 9 Memory consumption of the profiler for sequential NAS and Starbench benchmarks.

The slowdown of our profiler when profiling sequential programs is shown in Figure 8. The average slowdowns for the two benchmark suites (”NAS-average” and ”Starbench-average”) are also included. As the figure shows, our serial profiler has a 190× slowdown on average for NAS benchmarks and a 191× slowdown on average for Starbench programs. The overhead is not surprising since we perform an exhaustive profiling for the whole program.

When using 8 threads, our parallel profiler gives a 97× slowdown (best case 19×, worst case 142×) on average for NAS benchmarks and a 101× slowdown (best case 36×, worst case 253×) on average for Starbench programs. After increasing the number of threads to 16, the average slowdown is only 78× (best case 14×, worst case 114×) for NAS benchmarks, and 93× (best case 34×, worst case 263×) for Starbench programs. Compared to the serial profiler, our parallel profiler achieves a 2.4× and a 2.1× speedup using 16 threads on NAS and Starbench benchmark suites, respectively.

4.4.2 Memory consumption.

We measure memory consumption using the ”maximum resident set size” value provided by `/usr/bin/time` with the verbose (`-v`) option. Figure 9 shows the results. When using 8 threads, our profiler consumes 473 MB of memory on average for NAS benchmarks and 505 MB of memory on average for Starbench programs. The average memory consumption is increased to 649 MB and 1390 MB for NAS and Starbench programs, respectively. The worst case happens when using 16 threads to profile *md5*, which consumes about 7.6 GB memory. Although this may exceed the memory capacity configured in a three-year-old PC, it is till adequate for up-to-date machines, not to mention servers that are usually configured with 16 GB memory or more.

5 Conclusion and Outlook

We introduced a novel dynamic tool for the discovery of potential parallelism in sequential programs. Building the concept of computational units (CUs) and embedding it in a framework of combined static and dynamic analysis, we can reconcile the identification of parallel tasks in the form of CU chains with efficiency both in terms of time and memory. CU chains are not confined to predefined language constructs but can spread across the whole program. Our approach found 92.5% of the parallel loops in NAS Parallel Benchmark (NPB) programs and successfully identified tasks spanning several language constructs as well as pipeline patterns. It also helped parallelizing a loop in spite of initial dependences. Implementing the generated plan achieved reasonable speedups in most of our test cases: up to 2.67 for independent tasks and up to 3.62 for pipelines using a maximum of four threads.

In the future, we want to support further types of task parallelism including, for example, TBB flow graph. Furthermore, we want to develop heuristics to validate the suggestions before submitting them to the programmer, providing more accurate and reliable results.

References

1. Johnson, R.E.: Software development is program transformation. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. FoSER '10, ACM (2010) 177–180
2. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 45, IEEE Computer Society (2012) 437–448
3. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO '09, IEEE Computer Society (2009) 47–58
4. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: Rethinking and rebooting gprof for the multicore age. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, ACM (2011) 458–469
5. Li, Z., Jannesari, A., Wolf, F.: Discovery of potential parallelism in sequential programs. In: Proceedings of the 42nd International Conference on Parallel Processing. PSTI '13, IEEE Computer Society (2013) 1004–1013
6. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. The International Journal of Supercomputer Applications (1991)
7. Kim, M., Kim, H., Luk, C.K.: SD3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 43, IEEE Computer Society (2010) 535–546
8. Moseley, T., Shye, A., Reddi, V.J., Grunwald, D., Peri, R.: Shadow profiling: Hiding instrumentation costs with parallelism. In: Proceedings of the 5th International Symposium on Code Generation and Optimization. CGO '07, Washington, DC, USA, IEEE Computer Society (2007) 198–208
9. Kim, M., Kim, H., Luk, C.K.: Prospector: Discovering parallelism via dynamic data-dependence profiling. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism. HOTPAR '10 (2010)

10. Amini, M., Goubier, O., Guelton, S., Mcmahon, J.O., xavier Pasquier, F., Pan, G., Villalon, P.: Par4All: From convex array regions to heterogeneous computing. In: Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques. IMPACT 2012 (2012)
11. Grosser, T., Groesslinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters **22**(04) (2012) 1250010
12. Bienia, C.: Benchmarking Modern Multiprocessors. PhD thesis, Princeton University (January 2011)
13. Von Luxburg, U.: A tutorial on spectral clustering. Statistics and computing **17**(4) (2007) 395–416
14. Dong, J., Sun, Y., Zhao, Y.: Design pattern detection by template matching. In: Proceedings of the 2008 ACM Symposium on Applied Computing. SAC '08, ACM (2008) 765–769
15. Andersch, M., Juurlink, B., Chi, C.C.: A benchmark suite for evaluating parallel programming models. In: Proceedings 24th Workshop on Parallel Systems and Algorithms. PARS '11 (2011) 7–17