2016 IEEE International Parallel and Distributed Processing Symposium

Automatic Parallel Pattern Detection in the Algorithm Structure Design Space

Zia Ul Huda*, Rohit Atre†, Ali Jannesari*†, and Felix Wolf*
*Laboratory for Parallel Programming
Technische Universität Darmstadt
Darmstadt, Germany
{huda, jannesari, wolf}@cs.tu-darmstadt.de
†RWTH Aachen University
Aachen, Germany
atre@aices.rwth-aachen.de

***Abstract*—Parallel design patterns have been developed to help programmers efficiently design and implement parallel applications. However, identifying a suitable parallel pattern for a specific code region in a sequential application is a difficult task. Transforming an application according to support structures applicable to these parallel patterns is also very challenging. In this paper, we present a novel approach to automatically find parallel patterns in the algorithm structure design space of sequential applications. In our approach, we classify code blocks in a region according to the appropriate support structure of the detected pattern. This classification eases the transformation of a sequential application into its parallel version. We evaluated our approach on 17 applications from four different benchmark suites. Our method identified suitable algorithm structure patterns in the sequential applications. We confirmed our results by comparing them with the existing parallel versions of these applications. We also implemented the patterns we detected in cases in which parallel implementations were not available and achieved speedups of up to 14×.**

***Keywords*-parallelism; parallel patterns; task parallelism**

I. Introduction

In recent years, the compiler community has responded to the need for automatic parallelization; for example, state-of-the-art compilers such as the Intel compiler [1] can automatically detect parallel loops. However, compilers may miss coarse-grained parallelism due to their dependence on conservative static analysis.

To ease the burden of parallel programming, software engineers have introduced parallel design patterns [2]. These patterns provide solutions for recurring problems in parallel software development. Although parallel patterns are helpful for programmers, much effort is still needed to find appropriate places to apply them in the software architecture. This problem is exacerbated if one's job is to parallelize an existing sequential program. A programmer or software architect needs to have deep knowledge of not only parallel patterns but also the software under study to correctly parallelize it.

Researchers have developed tools that detect parallelism in sequential applications [3]. In this paper, we discuss an approach to automatically detect four patterns: multi-loop pipeline, task parallelism, geometric decomposition, and reduction. We detected the multi-loop pipeline pattern using linear regression analysis. To the best of our knowledge, there is no previous work detecting a multi-loop pipeline pattern. In contrast to previous approaches that detect task parallelism [4], [5], our technique classifies the detected tasks into categories such as *workers* and *barriers*. This classification helps implement task parallelism using supporting structures like *master/worker*. Our approach improved the detection rate of reduction and geometric decomposition, as compared to previous approaches [1], [6].

Our work is an extension of our parallelism discovery tool DiscoPoP (Discovery of Potential Parallelism) [7], [8]. Using this approach, we not only detected parallel patterns in sequential applications, we also classified related code sections according to the structures of the parallel patterns we detected. This simplifies the transformation of a sequential application into a parallel one.

We evaluated our approach on 17 sequential applications from four different benchmark suites: Starbench [9], BOTS [10], Polybench [11], and Parsec [12]. We successfully detected multi-loop pipelines, task parallelism, geometric decomposition, fusion, and reduction. We compared the detected patterns with the existing parallel versions of the benchmarks. For some benchmarks, a parallel version does not exist. We manually implemented the patterns we detected in these cases to validate our approach. We achieved 14× speedup running 32 threads in the best case of our hand-implemented parallel version.

The remainder of the paper is organized as follows. In Section II, we introduce DiscoPoP. Section III explains our pattern detection techniques in detail. In Section IV, we share our evaluation results. We discuss related works in Section V. Section VI concludes the paper and discusses future work.

II. Background

In this section we introduce our tool, DiscoPoP. It is built on top of LLVM [13] and conduct three types of analyses. The first analysis divides an input code into Computational

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.60

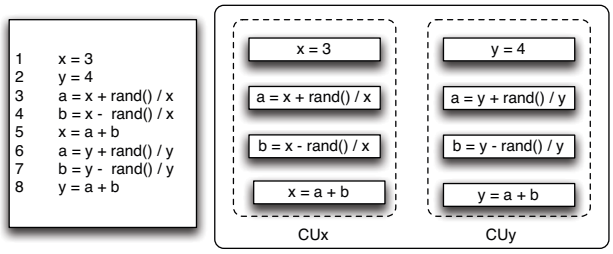

Figure 1. An example showing division of code into CUs.

Units (CUs). CUs follow the *read-compute-write* pattern [4]. It means that a program state is first read from memory, a new state is computed, and finally the new state is written back. This makes CUs logical units for the formation of larger tasks. CUs are detected at a code-granularity level that is independent of language and can be used for both program analysis and the expression of parallelism. These units of code can be assigned to a thread while running in parallel to other CUs or tasks formed by merging these CUs. CUs are building blocks for various patterns such as tasks in a task pool or stages in a pipeline.

Figure 1 shows an example of building CUs in a code section. DiscoPoP identifies two CUs. Line 1 of the code reads a value into variable x. Lines $3, 4$ and 5 compute some results from the value of x. Additionally, line 5 writes the final result to x. The local variables a and b are ignored; they are only used as temporary variables for computation. Therefore, lines 1 (read); $3, 4, 5$ (compute); and 5 (write) constitute one CU (called CU_x in the figure). Similarly, CU_y consists of lines 2 (read); $6, 7, 8$ (compute); and 8 (write). The construction of CU_y shows that a CU may consist of code lines that are not contiguous in the actual source code.

DiscoPoP's second analysis uses a dynamic approach to obtain data dependence and control region (loops and functions) information of an input source code [14]. The dynamic nature of this analysis makes its output sensitive to the input of the profiled application. To overcome this limitation, we run the profiled application with different representative inputs whenever possible and merge the outputs of the profiled runs.

The outputs of the two phases of DiscoPoP can be used for different kinds of analyses such as race detection [15], threads communication patterns [16] or parallelism discovery. Parallelism discovery varies from simple task and loop parallelism [4], [17] to complex parallel patterns - the subject of this paper.

III. Approach

Researchers have described patterns that exploit the concurrencies available in the structures of algorithms [2], [18]. They have also provided supporting structure patterns for the implementation of algorithm structure patterns. In Table I, we show patterns in the algorithm structure space supported by our approach and the best supporting structures according to related works. Our approach classifies CUs in a region according to the design of the related supporting structures. Thus the parallel version of the input application can be easily implemented.

Table I
Mapping of algorithm structure patterns to supporting structures

Type	Task	Data	Flow of data
Algorithm structure	Task parallelism	Geometric decomp., Reduction	Multi-loop pipeline
Supporting structure	Master/worker	SPMD	SPMD

Our approach uses *Program Execution Trees* (PETs) [8] generated using the outputs of the CU and dependence analyses of DiscoPoP. An example PET is shown in Figure 2. The nodes of the tree are control regions of the program. In our analysis, we use functions and loops as control regions (as reported by DiscoPoP). When a new loop starts or a function is called, a new child node is created in the tree. Iterations of a loop are merged together into a single tree node and the total number of iterations of the loop is recorded. Recursive calls of a function are merged together into a single node. This node is explicitly marked as recursive in the PET. All nodes in the PET contain the number of instructions in the LLVM's intermediate representation (IR) of corresponding regions. Loops and functions with a high percentage of instruction counts are identified as *hotspots*. The PET preserves the sequential execution order of the child nodes.

Each node in a PET contains CUs that are lexically located in these regions and represented by these nodes. Data dependences are mapped onto a pair of CUs. This mapping creates a *CU graph* with CUs as vertices and data dependences between them as edges.

A. Multi-loop Pipeline

Multi-loop pipelines are special cases of pipelines which are hidden because they stretch across more than one loop. In this scenario, one iteration of a loop depends on one or more iterations of a preceding loop in a sequential application. This can be easily transformed into a pipeline by mapping iterations of different loops onto different stages of the pipeline. In Listing 1, we show an example code where a multi-loop pipeline exists. Every i_{th} iteration of the later loop depends on the i_{th} iteration of the first loop. An important problem in the detection of multi-loop pipelines is the detection of iteration numbers of different loops that depend on each other.

We developed a mechanism that automatically detects multi-loop pipelines. All pairs of hotspot loops (in which one loop is data dependent on the other) are gathered from the

Table II
EFFECTS OF THE VALUES OF COEFFICIENTS a AND b IN EQUATION 1 ON MULTI-LOOP PIPELINES

Coefficient	Value	Description
a	1	one iteration of the loop y depends exactly on one iteration of the loop x.
a	< 1	1 iteration of loop y depends on $1/a$ iterations of loop x.
a	> 1	a iterations of loop y's depend on 1 iteration of x. So a iterations of loop y can be executed after 1 iteration of loop x.
b	0	All iterations of loop y depend on all iterations of loop x.
b	< 0	no iteration of loop y depend on first b iterations of loop x.
b	> 0	First b iterations of loop y do not depend on any iteration of loop x.

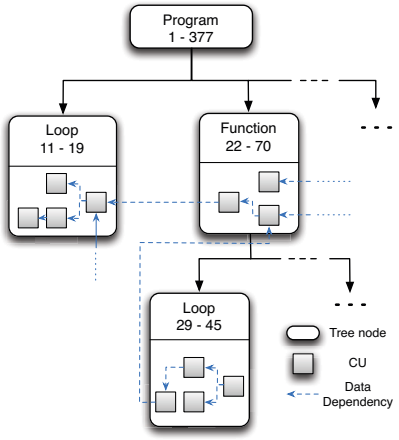

Figure 2. An example execution tree with control regions and a CU graph mapped onto them.

PET of a serial program. An LLVM-pass instruments only the load and store instructions that create these dependences. The pass records the iteration numbers of the loops for these instructions.

```
1    for (. . .)// Loop x
2       a[i] = foo(i);
3
4    for (. . .)// Loop y
5       b[i] = bar(a[i]);
```

Listing 1. Example of multi-loop pipeline.

The instrumented program is then executed with representative inputs; all the recorded information about the iteration numbers and memory addresses are dumped into an output file. Suppose, there is a dependence between loop x and loop y (as in Listing 1) where loop x writes to and loop y reads from the same memory locations. A post-analysis filters out the last iteration number i_x of loop x that wrote to a memory location m and the first iteration number i_y of loop y that reads from the same memory location m. These filtered iteration numbers are stored in pairs (i_x, i_y) for each loop pair detected in the previous step. Each pair denotes that iteration i_y of loop y is dependent on iteration i_x of loop x.

To estimate the relationship between pairs of iterations, we used linear regression analysis [19]. Linear regression estimates the relationship between a dependent variable Y and an independent variable X. Linear regression helps in finding coefficients a and b of the linear equation:

$$Y = aX + b \tag{1}$$

Once we have these coefficients, we can estimate the potential existence of a multi-loop pipeline between two loops. It is a perfect multi-loop pipeline when each i_{th} iteration of loop y depends exactly on the i_{th} iteration of loop x. The values of coefficients a and b for this line are 1 and 0, respectively.

We denote the area under the line of a perfect pipeline as $\int_{perfect}$. Similarly, linear regression gives us a regression line for the relationship between the iterations of loops x and y. The area below the generated regression line is denoted by $\int_{current}$. To estimate the efficiency of multi-loop pipeline between the iterations of loops x and y, we calculate the multi-loop efficiency factor e with formula:

$$e = \frac{\int_{current}}{\int_{perfect}} \tag{2}$$

The value of e represents how efficient the multi-loop pipeline will be. If the value of e is equal to 1.0, then we have a perfect multi-loop pipeline. If the value of e is close to or equal to 0, the multi-loop pipeline will be inefficient; the second loop y will have to wait for almost all iterations of the first loop x to finish before any iteration of loop y can start. If e is much higher than 1, there is a possibility of running both loops almost in parallel to each other with minimal synchronization needed between their iterations. In addition to the efficiency factor e, the coefficients a and b of Equation 1 give some insights into the implementation of a multi-loop pipeline. The effects of the values of a and b on a multi-loop pipeline are shown in Table II. With the help of all three values (e, a, and b), programmers can easily predict the practicality of multi-loop pipelines.

If there is a chain of more than two loops depending on each other (e.g. loop y depends on loop x and loop z depends on loop y), our tool separately outputs the relationships between loops x and y as well as loops y and z. If there is a chain dependence of n loops, it gives n pairs of relationships. A pipeline of n stages can be easily implemented by merging the information provided by

the tool. Moreover the loops in each stage of a multi-loop pipeline may be parallelized using other parallel patterns e.g. do-all, reduction or pipeline etc.

Loop Fusion: In some cases, a detected multi-loop pipeline can be optimized using fusion. Loop fusion is a loop optimization technique used by compilers to merge two loops into a single one if they iterate over the same range [20]. This is done to reduce loop overhead and increase granularity. However, fusion done by compilers is limited by static analysis and only effective if the loops are next to each other. Our approach suggests the fusion of loops based on dynamic analysis and the suggested loops may be lexically apart from each other in the actual source code.

We use the data gathered during our analysis of multi-loop pipelines to detect a fusion. A fusion of loops x and y can occur if:

- both loops x and y are do-all loops.
- the values of coefficients a and b of Equation 1 are exactly 1 and 0, respectively. As a result, the efficiency factor e from multi-loop analysis will be 1.

Both conditions ensure that the fused loop will not contain any loop-carried dependences and can be parallelized using do-all. If these conditions are met, we suggest fusing them together as a single loop and parallelizing it with do-all. This reduces the synchronization overhead for each loop and makes the parallelization more coarse-grained.

Another advantage of employing fusion is that it improves locality of reference [20]. DiscoPoP currently does not report the amount of data being handled by a single loop iteration. Therefore, our approach does not consider locality of reference when suggesting loop fusion. We will address this feature in our future work.

B. Task Parallelism

Task parallelism pattern is a collection of concurrent independent tasks. The most straightforward method of finding task parallelism is to look for totally independent CUs in the CU graph of a region and consider them independent tasks. However, most cases include dependences between CUs. Previous task parallelism detection techniques like [4], [5] were limited; they did not report the appropriate synchronization between the detected parallel tasks. We use *Breadth First Search* (BFS) to detect task parallelism in the CU graph of a region and classify them in a way that helps programmers to easily synchronize them in parallel execution.

Algorithm 1 illustrates our task parallelism detection approach. We start with the first unmarked CU in the CU graph of a hotspot in serial order of execution and mark it as a *fork* CU. All unmarked CUs dependent on the current CU are marked as *worker* CUs. A dependent CU is marked as a *barrier* CU if it was already marked. A barrier CU depends on more than one CU. A worker or barrier CU may behave as a fork CU for all of its dependent CUs. Once all

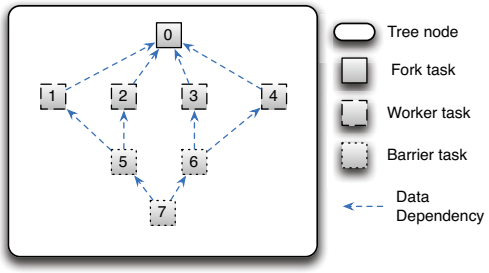

Figure 3. CU graph of function `cilksort()` from *sort* benchmark.

Algorithm 1: Detection of task parallelism.

```
Function detectTP(CUGraph)
    while !AllCUsMarked(CUGraph) do
        S = getFirstUnmarkedCU(CUGraph)
        S.mark = "Fork"
        Queue.push(S)
        while !Queue.empty() do
            N = Queue.pop()
            foreach D in N.dependents do
                if D.mark is NULL then
                    D.mark = "Worker"
                end
                else
                    D.mark = "Barrier"
                end
                Queue.push(D)
            end
        end
    end
    checkParallelBarriers(CUGraph)
```

CUs traversable by a current fork CU have been successfully marked, the algorithm searches for any unmarked CUs in the CU graph. If one is found, this CU is marked as a fork point and the entire process described above is repeated for its dependent nodes.

The output of our task parallelism detection algorithm is a classification of CUs into fork, worker, and barrier CUs. It contains precise details about which CUs fork which worker CUs and which CUs are barrier CUs for these workers. In Figure 3, we show the CU graph of function `cilksort()` of the *sort* application and the classification of CUs by our algorithm. The output of our algorithm for this CU graph shows that CU_0 forks four workers: CU_1, CU_2, CU_3 and CU_4. CU_5 is a barrier for workers CU_1 and CU_2. CU_6 is a barrier for workers CU_3 and CU_4. Finally, CU_7 is a barrier for CU_5 and CU_6. To see if two barriers can run parallel to each other, we check for a directed path from one barrier to the other in the CU graph (or vice versa). The absence of a

path denotes that two barriers can run in parallel; otherwise, they cannot. For example, there is no directed path between CU_5 and CU_6. Hence, they can run in parallel. On the other hand, there are paths from CU_7 to CU_5 and CU_6. So, CU_7 cannot run in parallel with either of these two barrier CUs.

We calculate *estimated speedup* metric for the task parallelism we detected. This is calculated by dividing the total number of LLVM-IR instructions in the hotspot by the total number of LLVM-IR instructions on the critical path of that hotspot. This metric gives an indication of the effectiveness of the task parallelism detected in that hotspot.

Our output format for task parallelism detection algorithm helps programmers to easily transform code using supporting structures like master/worker or fork/join. The implementation of these supporting structures requires correct synchronization between parallel tasks. Our approach identifies these synchronization points in the form of fork, worker, and barrier CUs.

C. Geometric Decomposition

We often need to run the same program code over a huge amount of data, motivating the term *Single Program Multiple Data*. Such a paradigm can be easily parallelized if the data can be divided into chunks and processed independently by separate threads. This kind of data parallelism is called geometric decomposition pattern [2].

Do-all is a case of geometric decomposition in which the iterations of a loop run independently of each other on separate data. However, do-all pattern is restricted to loops. Programmers may overlook geometric-decomposition opportunities on the function level. In our approach to geometric decomposition detection, we gather all immediate child nodes of a hotspot function node from the PET. We analyze all the loops in the current function and all the loops in the functions called directly from the current function. If all the analyzed loop nodes are either do-all or reduction loops, then we suggest this function as a candidate for geometric decomposition. Algorithm 2 shows the pseudo-code of geometric decomposition pattern detection.

DiscoPoP cannot determine the amounts and types of data being supplied to a function. Currently, the programmer must decide how the data entered into the function can be split into separate chunks so that the same function can be called separately for each chunk of data in separate threads. This often facilitates speedup; geometric decomposition coarsens the granularity of parallelization and reduces synchronization overhead as compared to the implementation of a separate do-all or reduction pattern for each individual loop in the function.

D. Reduction

Reduction pattern enables programmers to parallelize loops with a specific type of inter-iteration dependence. It can only be used when a loop uses an associative binary operator to reduce all elements of a container to a single scalar value, e.g., a loop summing all the elements of an array. State-of-the-art compilers generally recognize reduction, though pointer aliasing and array referencing may make them miss some reduction opportunities [1]. Our dynamic approach overcomes these limitations.

Algorithm 2: Detection of geometric decomposition.

```
Function detectGD(PET, FuncNode)
    Nodes = getImmediateChildren(PET, FuncNode)
    for each child in Nodes do
        if child is Loop then
            if !doallLoop(child) OR
            !reductionLoop(child) then
                return false
            end
        end
        else if child is Function then
            if !AllLoopsDoallOrReduction(child)
            then
                return false
            end
        end
    end
    return true
```

Algorithm 3: Reduction detection.

```
Input: loopID
Vars = getAllWrittenVars()
for each Var in Vars do
    writeLines = getWriteLines(Var)
    if |writeLines| != 1 then
        next
    end
    readLines = getReadLines(Var) if
    |readLines| != 1 || readLines != writeLines
    then
        next
    end
    Result[loopID]+ = Reduction candidate at
    writeLines.
end
return Result
```

We detect reduction during the same LLVM-pass we use for multi-loop pipelines. The pass instruments all LLVM-IR instructions creating inter-iteration dependences in a loop. However, there are two differences between reduction and multi-loop pipeline analysis: 1) dependence analysis between the iterations is done for iterations of the same loop

instead of two separate loops; and 2) source line numbers are also recorded for each write and read operation for each variable involved.

Currently, we can detect only the simplest case of reduction. In Algorithm 3, we show the steps for the detecting a reduction pattern. All variables accessed by instructions, instrumented during the instrumentation phase, are checked for reduction. If a memory address is written only on a single source line of a loop and read only at the same source line, the loop is reported as a possible candidate for a reduction pattern. The detection results also contain source line numbers where reduction may occur.

Our approach does not automatically identify the operator used at the source line number reported for a possible reduction. Currently, this burden is still left to the programmer, who must decide whether the operation in the reported source line number can be parallelized using reduction.

IV. Evaluation

We evaluated our approach with four different benchmark suites. Starbench consists of C/C++ benchmarks from diverse fields such as image processing, hashing, compression and so on [9]. The Barcelona OpenMP Task Suite (BOTS) [10] has been designed to study task parallelism in OpenMP. The Polyhedral Benchmark suite (Polybench) [11] consists of benchmarks with kernels from areas such as data mining and linear algebra etc. We also used another benchmark named *fluidanimate* from the Parsec benchmark suite [12] to test multi-loop pipeline detection. Starbench, BOTS, and Parsec include sequential as well as parallel versions of these benchmarks, allowing us to compare our detection results with the available parallel versions in these suites. We only implemented those detected patterns ourselves that were not found in the parallel versions of the respective benchmarks. Polybench does not have any parallel version available, so we implemented all the detected patterns in polybench by manually transforming the sequential versions. We explicitly mention whenever we implemented a detected pattern on our own.

We conducted our evaluation on a machine with 2×8-core Intel Xeon E5-2650 2 GHz processors with hyper-threading and 32 GB memory, running Ubuntu 12.04 (64-bit server edition). We compiled all the benchmarks with Clang 3.3 using `-g -O2` flags. We report the overall detection results of all these benchmarks in Table III. We tested all these benchmarks with a maximum of 32 threads and we report the number of threads where the benchmarks achieved their highest speedup. Whenever possible, we used the multiple inputs provided in benchmark suites to mitigate the limitations of dynamic analysis. In the following subsections, we will discuss the results of each detection approach in detail.

A. Multi-loop Pipeline

Our tool detected multi-loop pipelines in six benchmarks. *ludcmp*, *reg_detect*, and *fluidanimate* had multi-loop pipelines. We further classified the other three i.e. *rot-cc*, *correlation*, and *2mm* as fusion.

Table IV shows the values for the coefficients a and b as well as the efficiency factor e for the detected multi-loop pipelines. In *ludcmp*, we found a multi-loop pipeline between the two loops in function `kernel_ludcmp()`. The first loop was a do-all loop; the second loop had inter-iteration dependences. We found a one-to-one dependence between iterations of the two loops so it was reported as a perfect multi-loop pipeline. We implemented the detected multi-loop pipeline and achieved a maximum speedup of 14.06 with 32 threads. The first stage of the pipeline was implemented additionally as a parallel do-all loop.

In *reg_detect*, we found a multi-loop pipeline between the two loops in the function `kernel_reg_detect()`. Again, the first loop was a do-all loop and the second loop had inter-iteration dependences. Interestingly, the value of the coefficient b was -1; no iteration of the second loop had any dependence on the first iteration of the first loop. We manually analyzed the code and confirmed this was indeed the case. Listing 2 shows the skeleton of the two loops from *reg_detect*. The value of e was slightly affected by the value of coefficient b. We implemented a multi-loop pipeline for *reg_detect* by peeling the first iteration of the first loop. The remaining iterations of both loops had a one-to-one dependence. Our parallel version achieved a speedup of 2.26 with 16 threads.

```
1   void kernel_reg_detect(){
2      ...
3      for (i=0; i<PB_MAXGRID-1; i++) {
4         ...
5         mean[i][j] = ....
6      }
7      for (i=1; i<PB_MAXGRID-1; i++) {
8         ...
9         path[i][j] = path[i-1][j-1] +
              mean[i][j]
10     }
11  }
```

Listing 2. Two dependent hotspot loops of *reg_detect*.

In `ComputeForces()` of *fluidanimate*, we found two hotspot loops with multi-loop pipeline between them. The pseudo code of these two loops is shown in Listing 3. Neither of these two loops were do-all loops. The values for e, a, and b were recorded as 0.97, 0.05, and -3.50, respectively. Based on our description of coefficients in Table II, one iteration of the second loop depends on 20 iterations of the first loop (i.e. $1/a = 1/0.05 = 20$). We manually analyzed the loops and found that each iteration of the first loop updated the densities of the current cell and neighboring cells. The second loop reads and (again)

Table III
OVERALL PATTERN DETECTION RESULTS FOR DIFFERENT APPLICATIONS FROM STARBENCH, BOTS, POLYBENCH AND PARSEC.

Application	Benchmark Suite	LOC	Exec Inst % in Hotspot	Speedup	Threads	Detected Pattern
ludcmp	Polybench	135	88.64%	14.06	32	Multi-loop pipeline
reg_detect	Polybench	137	99.50%	2.26	16	Multi-loop pipeline
fluidanimate	Parsec	3987	99.54%	1.5	3	Multi-loop pipeline
rot-cc	Starbench	578	94.53%	16.18	32	Fusion
Correlation	Polybench	137	99.27%	10.74	32	Fusion
2mm	Polybench	153	99.19%	13.50	32	Fusion
fib	BOTS	32	100.00%	13.25	32	Task parallelism
sort	BOTS	305	94.89%	3.67	32	Task parallelism
strassen	BOTS	399	90.27%	8.93	32	Task parallelism
3mm	Polybench	166	99.44%	12.93	16	Task parallelism + Do-all
mvt	Polybench	114	91.24%	11.39	32	Task parallelism + Do-all
fdtd-2d	Polybench	142	76.51%	5.19	8	Task parallelism
kmeans	Starbench	347	2.04%	3.97	8	Geometric decomposition + Reduction
streamcluster	Starbench	551	49.99%	6.38	32	Geometric decomposition
nqueens	BOTS	118	100.00%	8.38	32	Reduction
bicg	Polybench	191	74.58%	5.64	8	Reduction
gesummv	Polybench	188	65.33%	5.06	8	Reduction

Table IV
SUMMARY OF MULTI-LOOP PIPELINE DETECTION.

Application	a	b	e
ludcmp	1	0	1
reg_detect	1	−1	0.99
fluidanimate	0.05	−3.50	0.97

updates the densities of these cells, confirming the results of our technique.

We implemented a multi-loop pipeline for fluidanimate according to our detection results. Due to the complex dependences between the loops, we achieved a maximum speedup of only 1.5 with 3 threads. The parallel version of *fluidanimate* in Parsec was implemented using grid of parallel tasks with locks (stencil) to handle the complex dependences between loops.

```
1   for(i=o;i<num; ++i)
2      neigs[i] = get_neigs(cells[i]);
3   for{i=0; i<num; ++i}
4      ComputeDensities(cells[i],
           neigs[i]);
5   for{i=0; i<num; ++i}
6      ComputeForces(cells[i], neigs[i]);
```

Listing 3. Two dependent hotspot loops of *fluidanimate* benchmark.

We found three cases of fusion in the benchmarks *rot-cc, Correlation,* and *2mm*. All of these benchmarks had dependent do-all hotspot loops that could be fused. Since *Correlation* and *2mm* are from Polybench, they are not shipped with parallel versions. We implemented the parallel versions of these two benchmarks via fusion and the speedups achieved are shown in Table III. *Rot-cc* from Starbench has a parallel version available, which was implemented by fusing the same two hotspot loops that our tool had detected.

B. Task Parallelism

We detected task parallelism in 6 benchmarks in total. A summary of detected task parallelism is given in Table V. Three of these were from BOTS and three from Polybench. Although BOTS is developed for studying OpenMP task parallelism, we discovered that most of BOTS applications have do-all parallelism; this is implemented in the parallel version of BOTS using OpenMP tasks.

```
1   long long fib (int n){
2      long long x, y;        //sync
3      if (n < 2) return n; //sync
4      x = fib(n - 1);        //worker
5      y = fib(n - 2);        //worker
6      return x + y;}         //sync
```

Listing 4. Hotspot function of *fib* benchmark.

We detected task parallelism in the benchmarks *fib, sort,* and *strassen* from BOTS. All of these applications have multiple independent recursive calls that were identified as separate tasks. *Fib*, as we show in Listing 4, calls itself twice. DiscoPoP detected both of these function calls as independent tasks. The parallel version of *fib* has been implemented using task parallelism at the same locations. Our estimated speedup was 3.25; however, the parallel implementation of *fib* included in BOTS achieved a speedup of 13.25. There is such a big difference between these two because DiscoPoP does not record a function's number of recursive invocations. Our calculation is based on only one recursive step; the actual application may achieve a better speedup with the parallelism available in several recursive calls.

In Section III, we discussed the CU graph of *sort* in detail. Figure 3 shows the CU graph of function `cilksort()` in *sort*. Here CU_1, CU_2, CU_3, and CU_4 represent recursive calls to the function itself. CU_5, CU_6, and CU_7 are the calls

Table V
SUMMARY OF TASK PARALLELISM PATTERN DETECTION.

Application	Total Instructions	Instructions on Critical Path	Estimated Speedup
fib	52	16	3.25
sort	2478	1172	2.11
strassen	11722739	3349354	3.5
3mm	3293952	2195968	1.5
mvt	9600	4896	1.96
fdtd-2d	137560	63309	2.17

to function `cilkmerge()`.

Another task parallelism opportunity is detected in *sort* in function `cilkmerge()` with a similar CU graph as in *fib*. Task parallelism patterns detected in both `cilksort()` and `cilkmerge()` have been implemented in the parallel version of *sort* in BOTS. By exploiting both task-parallelism opportunities, the parallel implementation of *sort* from BOTS achieves a maximum speedup of 3.67 on 32 threads.

In *strassen*, there are seven independent recursive calls in function `OptimizedStrassenMultiply()`. We classified all of these as worker tasks. A loop after these seven recursive calls uses their computed results, thus becoming a barrier task. Exactly the same task parallelism pattern as the one we detected has been implemented in the parallel version of *strassen* for the same seven recursive function calls in BOTS.

```
1   void kernel_3mm(){
2      for (i = 0; i < _PB_NI; i++)//worker
3         E[i] = A[i]+B[i];
4      for (i = 0; i < _PB_NI; i++)//worker
5         F[i] = C[i]+D[i];
6      for (i = 0; i < _PB_NI; i++)//sync
7         G[i] = E[i]+F[i];}
```

Listing 5. Hotspot function of *3mm* benchmark.

In *3mm*, we detected three loops in function `kernel_3mm()` (as shown in Listing 5) as tasks. The first two loops were classified as independent worker tasks and the third loop as their barrier task, because it depends on the first two loops. Similarly, in *mvt* two independent loops in function `kernel_mvt()` were categorized as worker tasks. All the loops that were detected as tasks in both *3mm* and *mvt* were also classified as do-all loops. We implemented combined task and do-all parallelism for both of these benchmarks and achieved speedups of 12.93 and 11.39 for *3mm* and *mvt*, respectively.

We detected task parallelism in the only hotspot loop in the function `kernel_fdtd_2d()` from *fdtd-2d*. This loop consists of four CUs. The first three CUs are independent and the last CU depends on the other three. DiscoPoP classified the first three as worker CUs and the last as their barrier. After implementing the task parallelism for *fdtd-2d*, we achieved a speedup of 5.19.

C. Geometric Decomposition

We detected geometric decomposition in *streamcluster* and *kmeans* from Starbench. The main loop in function `streamCluster()` of *streamcluster* is shown in Listing 6. It creates new clusters in each iteration and can not be parallelized with do-all or pipeline because it uses the new clusters formed at the end of the last iteration as an input to the next iteration. Therefore, we detected no parallel pattern in `streamCluster()`. The next hotspot was function `localSearch()` being called within the biggest hotspot loop.

```
1   void streamCluster(){
2      while(1){
3         ...
4         localSearch(points);
5         ...
6         if (eof)
7            break;
8   }}
```

Listing 6. Hotspot loop of *streamcluster* benchmark.

All the loops in `localSearch()` and the loops in the functions called by `localSearch()` were detected as do-all or reduction loops. We did not report the loops we detected as reduction in Table III because they are not hotspots. Our tool recommended `localSearch()` as a possible candidate for geometric decomposition. We analyzed the parallel version of *streamcluster* from Starbench and found that it was parallelized via the geometric decomposition of function `localSearch()`. The pseudo code of the parallel *streamcluster* implementation is shown in Listing 7. Similarly, we reported the function `cluster()` of *kmeans* as a possible geometric decomposition candidate; this finding was confirmed by the existing parallel version of *kmeans*.

```
1   void streamCluster(){
2      while(1){
3         ...
4         for(i=0; i<num_chuncks; i++)
5            new_thread(localSearch(
                points[i*chunk_size],
                chunk_size));
6         ...
7         if (eof)
8            break;
9   }}
```

Listing 7. Parallel implementation of *streamcluster* benchmark.

D. Reduction

We detected a reduction pattern in four benchmarks: *nqueens, kmeans, bicg,* and *gesummv*. In *nqueens*, we detected a reduction pattern in the main loop in function `nqueens()`. The existing parallel version of this benchmark in BOTS is implemented with reduction. The reduction loop detected in *kmeans* was inside a function suitable for geometric decomposition that has already been discussed.

Table VI
COMPARISON OF REDUCTION DETECTION RESULTS.

Tool	*nqueens*	*kmeans*	*bicg*	*gesummv*	*sum_local*	*sum_module*
Sambamba	NA	NA	✓	✓	✓	×
icc	×	×	×	×	✓	×
DiscoPoP	✓	✓	✓	✓	✓	✓

The reduction loop of *gesummv* had two reduction variables and our tool reported both of them. Both *bicg* and *gesummv* are from Polybench; therefore, we implemented their parallel versions manually via reduction. We achieved peak speedups of 5.64 for *bicg* and 5.06 for *gesummv* using 8 threads.

As we pointed out earlier, state of the art compilers use static analysis to detect reduction pattern in loops. To demonstrate the advantage of our approach, we implemented two synthetic benchmarks we call *sum_local* and *sum_module*. We show the code of these synthetic benchmarks in Listings 8 and 9, respectively. The reduction operation is done in the lexical extent of the loop in *sum_local*; in *sum_module*, the reduction operation is done in another function called from within the loop.

```
1   int sum_local(int* arr, int size){
2      int sum=0;
3      for (i = 0; i < size; i++)
4         sum += arr[i];
5   }
```

Listing 8. Code of *sum_local* reduction.

```
1   int sum_module(int &sum, int val){
2      int x = ...//do some heavy work on
           val
3      d += x;
4      return x;
5   }
6   int sum_module(int* arr, int size){
7      int sum=0;
8      for (i = 0; i < size; i++){
9         x = sum_module(sum, arr[i]);
10        foo(x);
11     }
12     return sum;
13  }
```

Listing 9. Code of *sum_module* reduction.

We compared our results for these two synthetic benchmarks with Intel's *icc* compiler [1] and *Sambamba* [6], another tool that statically detects reductions. The results are shown in Table VI. The limitations of static analysis prevented both *icc* and *Sambamba* from detecting the reduction in *sum_module*. In contrast, our approach detected the reduction patterns in both benchmarks.

V. RELATED WORK

The detection of sequential design patterns is an active research topic in software engineering, especially in object-oriented programming. Previous studies [21], [22], [23] look for sequential design patterns in UML diagrams of an existing application. In the field of parallel computing, Poovey et al. [24] detect parallel design patterns in an already parallelized application. They use different metrics such as thread count, memory sharing, and unique instructions etc. to recognize a pattern. Their propose appropriate parallel patterns to optimize the parallelized application. The MINIME [25] tool leverages the same technique to search for parallel patterns in parallel applications. However, its ultimate objective is to produce a synthetic benchmark with the same parallel patterns and parallelization behavior. These benchmarks can then be used for testing architectures under development. Both of these approaches detect patterns in already parallelized programs; conversely, we detect parallel patterns in sequential programs.

Molitorisz [3] propose a tool that detects parallel design patterns and applies automatic transformations to create a parallel version of a serial input application; it is able to detect master/worker and pipeline patterns. Parceive [26] is another automatic parallelization tool based on the dynamic analysis of a sequential application. It dumps trace data into a database and uses queries to detect parallelism. The corresponding evaluation reports the successful detection of a pipeline. All of these tools are limited to pipeline and master/worker patterns; our tool detects a broader range of patterns, including multi-loop pipeline, task parallelism, geometric decomposition, and reduction.

Cordes et al. [27] detect task parallelism in hierarchical task graphs for heterogeneous architectures. These tasks can be at the instruction level, the loop level, or the function level depending on the available architecture. Li et al. [5] search for task parallelism, data parallelism, and pipelines in program dependence graphs by identifying strongly connected components and coarsening their granularity with typed fusion. However, their approach supports only scalar dependences. They miss parallelization opportunities when dependences involve pointers or arrays. The *Tareador* tool detects task parallelism [28]. Its unit of parallelization is a function or a loop; we use CUs, which are more fine grained.

Wang et al. [29] automatically derive pipeline and do-all parallelism from a program dependence graph of a sequential application. Decoupled software pipelining converts a repeating region's dependence graph into a pipeline using strongly connected components [30]. This conversion also creates more opportunities for parallelism in different stages of the pipeline. To the best of our knowledge, there is no previous work on the detection of a pipeline covering multiple loops. Fusion is a well studied and reported optimization in the compilers community. Modern compilers such as Intel *icc* [1] use static analysis to fuse loops; it is usually conservative. Our approach uses dynamic analysis to detect possible fusion candidates that may reside in different parts of an application.

Streit et al. [6] developed a tool called Sambamba for the automatic detection of parallel patterns and subsequent parallelization. It covers a wide range of patterns: do-all, do-across, reduction, task parallelism, privatization, and speculation. These patterns are detected using static analysis. We use hybrid analysis to detect patterns more accurately, as shown in the case of reduction detection in Section IV.

VI. Conclusion & Future Works

We have extended DiscoPoP by adding functionality to detect four parallel patterns in the algorithm structure space. Our tool detects patterns in hotspot regions of applications and maps individual tasks found in these regions onto the pattern's support structure. This helps programmers implement the suggested pattern during parallelization.

We plan to support more parallel patterns and loop optimizations such peeling and fission in the future. Moreover, we want to improve our reduction detection so we can automatically infer the type of reduction operator and cover more complex reduction scenarios. We aim to define metrics that help choose the best pattern among multiple detected parallel patterns. Such metrics may also quantify the human effort needed for code transformation. Our future work also includes semi-automatic code transformation of a sequential application into a parallel one.

Acknowledgment

We appreciate our graduate student Sergei Krestianskov for his programming support and also the support provided by the Klaus Tschira foundation.

References

[1] Intel. (2015) Intel C++ compiler XE 13.1 user and reference guide. [Online]. Available: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/

[2] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Addison-Wesley Professional, 2004.

[3] K. Molitorisz, "Pattern-based refactoring process of sequential source code," in *Proc. of the 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 357–360.

[4] R. Atre, A. Jannesari, and F. Wolf, "The basic building blocks of parallel tasks," in *Proc. of the International Workshop on Code Optimisation for Multi and Many Cores*, ser. COSMIC '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:11.

[5] F. Li, A. Pop, and A. Cohen, "Automatic extraction of coarse-grained data-flow threads from imperative programs," *IEEE Micro*, vol. 32, no. 4, pp. 19–31, Jul. 2012.

[6] K. Streit, J. Doerfert, C. Hammacher, A. Zeller, and S. Hack, "Generalized task parallelism," *ACM TACO*, vol. 12, no. 1, pp. 8:1–8:25, Apr. 2015.

[7] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf, "DiscoPoP: A profiling tool to identify parallelization opportunities," in *Tools for High Performance Computing 2014*. Springer International Publishing, Aug. 2015, ch. 3, pp. 37–54.

[8] Z. U. Huda, A. Jannesari, and F. Wolf, "Using template matching to infer parallel design patterns," *ACM TACO*, vol. 11, no. 4, pp. 64:1–64:21, Jan. 2015.

[9] M. Andersch, B. Juurlink, and C. C. Chi, "A benchmark suite for evaluating parallel programming models," in *Proc. of 24th Workshop on Parallel Systems and Algorithms*, 2011.

[10] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Proc. of International Conference on Parallel Processing*, ser. ICPP'09. IEEE, 2009, pp. 124–131.

[11] L.-N. Pouchet. PolyBench/C the polyhedral benchmark suite. [Online]. Available: http://www.cs.ucla.edu/ pouchet/software/polybench/

[12] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[13] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. of the International Symposium on Code Generation and Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[14] Z. Li, A. Jannesari, and F. Wolf, "An efficient data-dependence profiler for sequential and parallel programs," in *Proc. of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India*. IEEE Computer Society, May 2015, pp. 484–493.

[15] A. Jannesari, M. Westphal-Furuya, and W. F. Tichy, "Dynamic data race detection for correlated variables," in *Proc. of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ser. ICA3PP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 14–26.

[16] A. Mazaheri, A. Jannesari, A. Mirzaei, and F. Wolf, "Characterizing loop-level communication patterns in shared memory applications," in *Proc. of the 44th International Conference on Parallel Processing, Beijing, China*, ser. ICPP'15, Sep. 2015.

[17] Z. Li, A. Jannesari, and F. Wolf, "Discovery of potential parallelism in sequential programs," in *Proc. of the Workshop on Parallel Software Tools and Tool Infrastructures, Lyon, France*, ser. PSTI'13, Oct. 2013, pp. 1004–1013.

[18] K. Keutzer and T. Mattson, "Our pattern language (opl): A design pattern language for engineering (parallel) software," in *ParaPLoP Workshop on Parallel Programming Patterns*, vol. 14, 2009.

[19] D. Freedman, *Statistical Models: Theory and Practice*. Cambridge University Press, Aug. 2005.

[20] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[21] M. Qiu, Q. Jiang, A. Gao, E. Chen, D. Qiu, and S. Chai, "Detecting design pattern using subgraph discovery," in *Proc. of the 2nd International Conference on Intelligent Information and Database Systems: Part I*, ser. ACIIDS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 350–359.

[22] M. Gupta, A. Pande, and A. K. Tripathi, "Design patterns detection using SOP expressions for graphs," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–5, Jan. 2011.

[23] J. Dong, Y. Sun, and Y. Zhao, "Design pattern detection by template matching," in *Proc. of the ACM Symposium on Applied Computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 765–769.

[24] J. A. Poovey, B. P. Railing, and T. M. Conte, "Parallel pattern detection for architectural improvements," in *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, ser. HotPar'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 12–12.

[25] E. Deniz, A. Sen, B. Kahne, and J. Holt, "MINIME: pattern-aware multicore benchmark synthesizer," *IEEE Transactions on Computers*, vol. 64, pp. 2239 – 2252, 2014.

[26] A. Wilhelm, B. Sharma, R. Malakar, T. Schle, and M. Gerndt, "Parceive: Interactive parallelization based on dynamic analysis," in *Proc. of PCODA*. IEEE, 2015, pp. 1–6.

[27] D. Cordes, O. Neugebauer, M. Engel, and P. Marwedel, "Automatic extraction of task-level parallelism for heterogeneous MPSoCs," in *Proc. of the 42Nd International Conference on Parallel Processing*, ser. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 950–959.

[28] V. Subotic, E. Ayguad, J. Labarta, and M. Valero, "Automatic exploration of potential parallelism in sequential applications," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer International Publishing, 2014, vol. 8488, pp. 156–171.

[29] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM TACO*, vol. 11, no. 1, pp. 2:1–2:26, Feb. 2014.

[30] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, "Decoupled software pipelining creates parallelization opportunities," in *Proc. of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 121–130.