The Journal of Systems and Software 117 (2016) 282–295

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Unveiling parallelization opportunities in sequential programs

Zhen Li[a,*], Rohit Atre[b], Zia Huda[a], Ali Jannesari[a,c], Felix Wolf[a]

[a] Technische Universität Darmstadt, Darmstadt 64289, Germany
[b] RWTH Aachen University, Aachen 52062, Germany
[c] University of California, Berkeley, Berkeley CA 94720, USA

A R T I C L E I N F O

Article history:
Received 11 July 2015
Revised 11 March 2016
Accepted 17 March 2016
Available online 25 March 2016

Keywords:
Parallelism discovery
Program analysis
Profiling
Data dependence
Parallelization

A B S T R A C T

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. In this paper, we present an approach to automatically identify potential parallelism in sequential programs of realistic size. In comparison to earlier approaches, our work combines a unique set of features that make it superior in terms of functionality: It not only (i) detects available parallelism with high accuracy but also (ii) identifies the parts of the code that can run in parallel—even if they are spread widely across the code, (iii) ranks parallelization opportunities according to the speedup expected for the entire program, while (iv) maintaining competitive overhead both in terms of time and memory.

1. Introduction

Although the component density of microprocessors is still rising according to Moore's 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 the exception (Johnson, 2010). 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), identify potential parallelism in sequential programs based on data dependences. It profiles dependences, but instead of only reporting their violation it also watches out for their absence. The use of signatures (Sanchez et al., 2007) to track memory accesses, a concept borrowed from transactional memory, keeps the overhead at bay without significant loss of information, reconciling the first with the last requirement. We use the dependence 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 dependences 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

* Corresponding author. Tel.: +49 15111198669.
E-mail addresses: li@cs.tu-darmstadt.de (Z. Li), atre@aices.rwth-aachen.de (R. Atre), huda@cs.tu-darmstadt.de (Z. Huda), jannesari@eecs.berkeley.edu (A. Jannesari), wolf@cs.tu-darmstadt.de (F. Wolf).

http://dx.doi.org/10.1016/j.jss.2016.03.045
0164-1212/© 2016 Elsevier Inc. All rights reserved.

bring our approach closer to what a user needs than alternative methods (Ketterlin and Clauss, 2012; Zhang et al., 2009; Garcia et al., 2011) do.

We expand on earlier work (Li et al., 2013; 2015a; Ul-Huda et al., 2015), which introduced the concept of computational units (CUs) in parallelism discovery for the first time (Li et al., 2013), and related algorithms for identifying parallelism based on the notion of computational units (Li et al., 2015a; Ul-Huda et al., 2015). At that time, computational units were defined at instruction level, a granularity that is too fine to identify thread-level parallelism. Furthermore, the internal read-compute-write pattern of a CU was not clearly defined, leading to unnecessary complication for both the parallelism discovery algorithms and the users. Finally, it had significant time and memory overhead due to the brute-force data-dependence profiling method. Compared to the earlier approach, this paper has the following contributions:

1. Revised definition of computational units. According to the new definition, the three phases (read phase, compute phase, and write phase) of a CU can be clearly distinguished. The new definition is compatible with the one presented in earlier work (Li et al., 2013), but allows CUs to be built on any granularity.
2. Improved parallelism discovery algorithms adapted to the new definition of CUs. The algorithms identify more parallelization opportunities, including the ones identified by the earlier approach (Li et al., 2015a; Ul-Huda et al., 2015).
3. A ranking method to draw attention to the most promising parallelization targets.

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 demonstrate the results of applying our method on benchmarks from four different benchmark suites with two case studies. We also 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. As a common characteristic, all of them capture dynamic dependences to assess the degree of potential parallelism. Since this procedure is input sensitive, the analysis should be repeated with a range of representative inputs and the final validation is left to the user. Such dynamic approaches can be broadly divided into two categories. Tools in the first merely count dependences, whereas tools in the second, including our own, exploit explicit dependence information to provide detailed feedback on parallelization opportunities or obstacles.

Kremlin (Garcia et al., 2011) belongs to the first category. Using dependence 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 (Zhang et al., 2009) follows a similar strategy. Built on top of Valgrind, it calculates the number of instructions and the number of violating read-after-write (RAW) dependences across all program constructs. If the number of instructions of a construct is high while the number of RAW dependences 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 (loops, `if`s, functions, etc.). 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 (Ketterlin and Clauss, 2012) belongs to the second category. It records data dependences and attaches them to the nodes of an execution tree it maintains (i.e., a generalized call tree that also includes basic blocks). In comparison to DiscoPoP, Parwiz lacks a ranking mechanism and does not explicitly identify tasks. They have to be manually derived from the dependence graph, which is demonstrated using small text-book examples.

Reducing the significant space overhead of tracing memory accesses was also successfully pursued in SD3 (Kim et al., 2010b). An essential idea that arose from 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. (2007) is a representative example of this approach. Sampling also falls into this category. Vanka and Tuck (2012) profiled data dependences based on signature and compared the accuracy under different sampling rates.

Prospector (Kim et al., 2010a) is a parallelism-discovery tool based on SD3. It tells whether a loop can be parallelized, and provides a detailed dependence 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.

3. Approach

Fig. 1 shows our parallelism-discovery workflow. It is divided into three phases: In the first phase, we instrument the target program and execute it. Control flow information and data dependences are obtained in this phase. In the second phase, we search for potential parallelism based on the information produced during the first phase. The output is a list of parallelization opportunities, consisting of several code sections that may run in parallel. Finally, we rank these opportunities and write the result to a file.

3.1. Phase 1: Control-flow analysis and data-dependence profiling

The first phase includes both static and dynamic analyses. The static part includes:

- Instrumentation. DiscoPoP instruments every memory access, control region, and function in the target program after it has been converted into intermediate representation (IR) using LLVM (Lattner and Adve, 2004).
- Static control-flow analysis, which determines the boundaries of control regions (loop, if-else, switch-case, etc.).

The instrumented code is then linked to libDiscoPoP, which implements the instrumentation functions, and executed. The dynamic part of this phase then includes:

- Dynamic control-flow analysis. Runtime control information such as entry and exit points of functions and number of iterations of loops are obtained dynamically.
- Data-dependence profiling. DiscoPoP profiles data dependences using a signature algorithm.
- Variable lifetime analysis. DiscoPoP monitors the lifetime of variables to improve the accuracy of data-dependence detection.
- Data dependence merging. An optimization to decrease the memory overhead.

Note that we instrument intermediate representation, which is obtained from the source code of the application. That means libraries used in the application can only be instrumented when

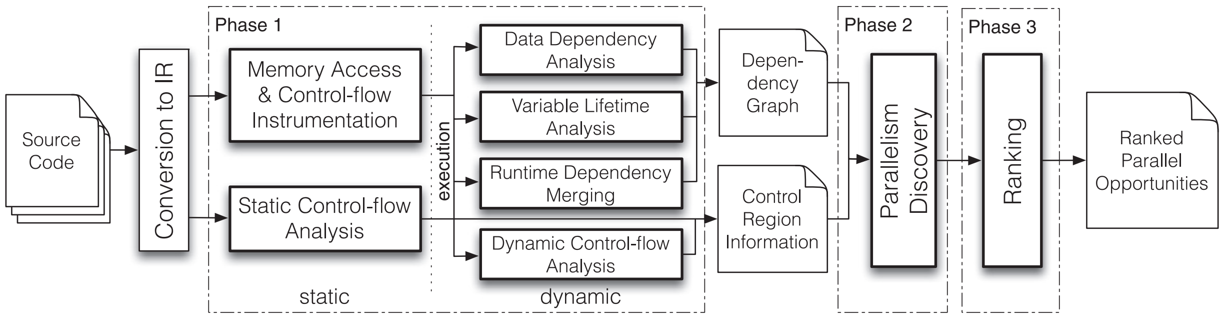

Phase 1
Source Code
Conversion to IR
Memory Access & Control-flow Instrumentation
Static Control-flow Analysis
execution
Data Dependency Analysis
Variable Lifetime Analysis
Runtime Dependency Merging
Dynamic Control-flow Analysis
static
dynamic
Dependency Graph
Control Region Information
Phase 2
Parallelism Discovery
Phase 3
Ranking
Ranked Parallel Opportunities

Fig. 1. Parallelism discovery workflow.

the source code of the libraries is available. We believe that this approach is sufficient for discovering parallelism since it is nearly impossible to parallelize binary code manually.

3.1.1. *Hybrid control-flow analysis*

We perform the control-flow analysis in a hybrid fashion. During instrumentation, the boundaries of control structures (loop, if-else, and switch-case, etc.) are logged while traversing the IR of the program. The boundaries are indexed by source line number, which allows us later to provide detailed information to the user. At the same time, we instrument control structures and functions.

During execution, inserted instrumentation functions log runtime control-flow information dynamically. Instrumentation functions for loops count the number of iterations, while functions for branches remember which branch is being executed so that data dependence information can be correctly mapped onto control-flow information. Instrumentation functions for function calls log the function boundaries. This is done dynamically because a function may have multiple return points and can return from different positions during execution.

3.1.2. *Signature-based data-dependence profiling*

Our earlier approach (Li et al., 2013) consumes more than 500 MB memory on very simple programs. For a relatively small benchmark (streamcluster) that performs computations iteratively, it consumes 3.3 GB to profile the dependences. Such high memory consumption limits our approach from being applied to real-world applications.

To lower the memory requirements of the data-dependence profiling, we record memory accesses based on signatures. This idea is originally introduced in (Vanka and Tuck, 2012). A signature is a data structure that supports the approximate representation of an unbounded set of elements with a bounded amount of state (Sanchez et al., 2007). It is widely used in transactional memory systems to uncover conflicts. A signature usually supports three operations:

- Insertion: A new element is inserted into the signature. The state of the signature is changed after the insertion.
- Membership check: Tests whether an element is already a member of the signature.
- Disambiguation: Intersection operation between two signatures. If an element was inserted in both of them, the resulting element must be represented in the resulting intersection.

A data dependence can be regarded as a conflict because a data dependence exists only when two or more memory operations access the same memory location in some order. Therefore, a signature is also suitable for detecting data dependences. In our approach, we adopt the idea of signatures to store memory accesses. A fixed-length array is combined with a hash function that

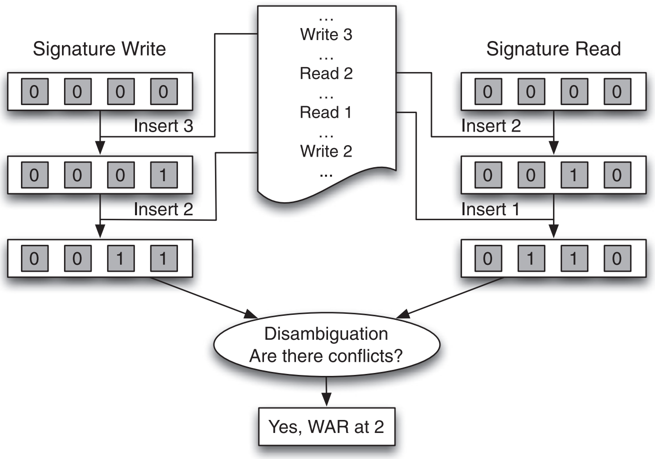


Fig. 2. Signature algorithm example.

maps memory addresses to array indices. In each slot of the array, we save the source line number where the memory access occurs. Because of the fixed length of the data structure, memory consumption can be adjusted as needed.

To detect data dependences, we use two signatures. One for recording read operations, one for write operations. When a memory access c at address x is captured, we first obtain the access type (read or write). Then, we run the membership check to see if x exist in the signature of the correspondent type. If x already exist, we update the source line number to where c occurs and build a data dependence between the current source line and the previous source line. Otherwise, we insert x into the signature. At the same time, we check whether x exist in the other signature. If yes, a data dependence has to be built as well. An alternative would be performing membership check and disambiguation whenever a write operation occurs, since read-after-read (RAR) dependences do not prevent parallelization.

Fig. 2 shows an example of how our algorithm works. The signature size in this example is four. Four memory accesses are recorded, including two write and two read accesses. A disambiguation of the two signatures indicates a conflict at address 2. In this case, a write-after-read (WAR) dependence must be built.

We insert a function at the beginning of the target program to initialize the data structures. Every read and write operation of the target program is instrumented. Since disambiguation usually incurs a bigger overhead than the membership check does, we build data dependences using membership check whenever possible.

3.1.3. *False positives and false negatives*

Representing an unbounded set of elements with a bounded amount of state means adding new elements can introduce errors.

The membership check of a signature can deliver false positives, which further lead to false positives and false negatives in dependences.

The false-positive rate of the membership check is a function of the signature size and the number of elements. Assume that we use one hash function, which selects each array slot with equal probability. Let m be the number of slots in the array. Then, the probability that the hash function does not use a slot during insertion is:

$$1 - \frac{1}{m}$$

After inserting n elements, the probability that a certain slot is still *unused* is:

$$\left(1 - \frac{1}{m}\right)^n$$

Now the estimated false-positive rate (EFPR), i.e., the probability that a certain slot is *used* is thus:

$$EFPR = 1 - \left(1 - \frac{1}{m}\right)^n$$

Obviously, to control the false-positive rate of the membership check, we need to adjust the size of the signature m to the number of variables n in the program. There are two solutions. The first solution is to set the size of signatures as big as possible so that the false positive rate is guaranteed to be low. This may sounds useless at the first time, but it is actually quite a simple, effective, and applicable solution. Experiments show that to profile NAS benchmarks with accuracy higher than 99.6%, the average memory consumption is 649 MB. Any ordinary PC that equipped with more than 2 GB memory has enough resource to secure a high accuracy. In this paper, we use this solution to profile data dependences, keeping the error rate lower than 0.4%.

The second solution is to calculate the size of the signatures in advance, if the amount of memory is really limited. Experiments show that when using a fixed number of slots (800,000) for profiling the NAS benchmarks, the EFPR vary between 0.01% and around 60%, meaning memory is wasted on small programs while not enough for the big ones. To avoid such a scenario, the user can specify an maximum EFPR, and DiscoPoP will choose the size of signature accordingly. In this way, memory can be used more efficiently and the quality of the final suggestions can be also assured.

To calculate the size of signature based on a given EFPR, we need to know the number of variables in the program. To avoid running the program more than once, we estimate the number of variables during instrumentation. The number is counted based on the intermediate representation (IR) of the program produced by the front-end of LLVM. Although the IR is in Static Single Assignment (SSA) form, it provides the possibility to distinguish constants, global variables, named and unnamed variables. Thus it is easy to define rules that filter out the variables that originated from the program. The rules are relaxed so that the counted number is always bigger than the real number of variables. This will result in a bigger size of the signature, leaving the actual false-positive rate usually below the specified EFPR threshold.

3.1.4. Parallel data-dependence profiling

We have our data-dependence profiler parallelized in order to lower the time overhead further. The basic idea behind the parallelization of our approach is to run the profiling algorithm in parallel on disjoint subsets of the memory accesses. In our implementation, we apply the producer-consumer pattern. The main thread executes the target program and plays the role of the producer, collecting and sorting memory accesses, whereas the worker threads play the role of consumers, consuming and analyzing memory accesses and reporting data dependences.

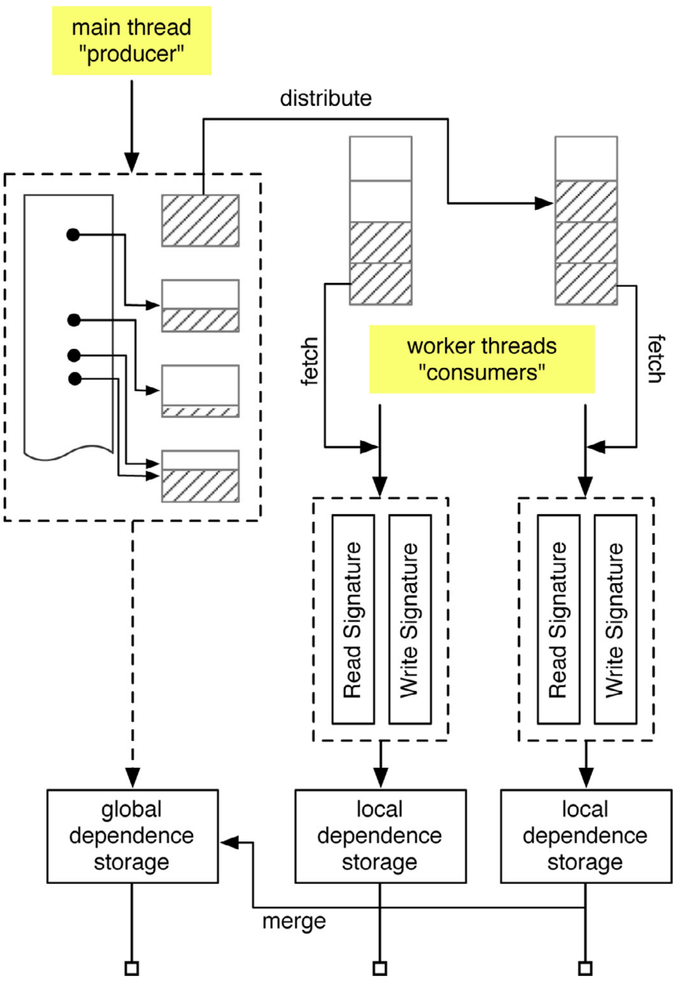


Fig. 3. Architecture of a parallel data-dependence profiler for sequential programs.

To determine the dependence type (RAW, WAR, WAW) correctly, we need to preserve the temporal order of memory accesses to the same address. For this reason, a memory address is assigned to exactly one worker thread, which becomes responsible for all accesses to this address.

Fig. 3 shows how our parallel design works. The main thread executes the program to be analyzed and collects memory accesses in chunks, whose size can be configured in the interest of scalability. One chunk contains only memory accesses assigned to one thread. The use of maps ensures that identical dependences are not stored more than once. At the end, we merge the data from all local maps into a global map. This step incurs only minor overhead since the local maps are free of duplicates. Detailed description of the parallel dependence profiler is in related work (Li et al., 2015b).

3.1.5. Variable lifetime analysis

Although false positives are a basic property of signatures and cannot be completely eliminated, we apply an optimization to lower the false-positive rate further. The main idea is to remove variables from the signature once it is clear that they will never be used again during the remainder of the execution. Thus, we need a way to monitor the lifetime of a variable. The lifetime of a variable is the time between its allocation and deallocation. The lifetime of variables has an impact on the correctness of the data dependence analysis because signature slots of dead variables might be reused for new variables. If this happens, a false dependence will be built between the last access of the dead variable and the first access of the new variable.

To resolve this problem, we perform variable lifetime analysis dynamically. This means we observe the allocation and deallocation of variables, including both explicit methods like `new`/`delete` and `malloc`/`free`, and implicit allocation and deallocation of local variables. To achieve this, we exploit dynamic control-flow information, which is helpful to determine the lifetime of local variables allocated inside a control region. Although there is no explicit deallocation of local variables, they die once the program leaves the control region where they have been allocated. In this way, signature slots for local variables can be reused without the danger of building false dependences. With variable lifetime analysis, our signature algorithm can support more variables with the same amount of memory.

3.1.6. Runtime data dependence merging

Recording every data dependence may consume an excessive amount of memory. DiscoPoP performs all the analyses on every instruction that is dynamically executed. Depending on the size of both the source code and the input data, the size of the file containing processed data dependences can quickly grow to several gigabytes for some programs. However, we found that many data dependences are redundant, especially for regions like loops and functions which will be executed many times. Therefore, we merge identical data dependences. This approach significantly reduces the number of data dependences written to disk.

A data dependence is expressed as a triple `<Dependent-Line, dependence-Type, Depends-On-Line>` with attributes like variable name, thread ID (only available for multi-threaded programs), and inter-iteration tag. Two data dependences are identical if and only if each element of the triple and all attributes are identical. When a data dependence is found, we check whether it already exists. If there is no match, a new entry for the dependence is created. Otherwise the new dependence is discarded. For a code region that is executed more than once, we maintain only one set of dependences, merging the dependences that occur across multiple instances. When the parallelism-discovery module reads the dependence file, it still treats these multiple execution instances as one. For example, a loop will always be considered as a whole and its iterations will never be expanded.

Merging data dependences may hide parallelism that is only temporarily available. For example, the first half of the iterations of a loop can be parallelized but the second half cannot. With data dependences merged, parallelism that exists in the first half of the iterations can be hidden. We recognize that temporarily available parallelism is definitely promising. However, discovering such parallelism requires a significant amount of time and memory since every iteration must have its own instance of profiled data, and parallelism must be checked between every two instances. We implemented a version without dependence merging, and it failed to profile most of the NAS benchmarks. For those programs it can profile, the size of the dependence file ranged from 330 MB to about 37 GB with input class W (6.1 GB on average).

The effect of merging data dependences is significant. After introducing runtime data dependence merging, all the NAS benchmarks can be profiled and the file size decreased to between 3 KB and 146 KB (53 KB on average), corresponding to an average reduction by a factor of $10^5\times$. Since the parallelism-discovery module redirects the read pointer in the file when encountering function calls rather than processing the file linearly, data dependence merging drastically reduces the time needed for parallelism discovery. The time overhead of data dependence merging is evaluated in Section 4. Since we still want to cover temporarily available parallelism, an efficient profiling and analysis method for loops is under development.

3.2. Phase 2: Parallelism discovery

During the second phase, we search for potential parallelism based on the output of the first phase, which is essentially a graph of dependences 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) dependences 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.

3.2.1. Computational units

The first definition of computational units (CUs) was presented in earlier work (Li et al., 2013). However, CUs were only defined in instruction level, which is too fine-grained to identify thread-level parallelism (CUs have to be grouped to form tasks). The internal computation process of a CU was not clearly defined, making the parallelism discovery algorithms less powerful and unnecessarily complicated. In this paper, we redefine computational units to overcome the disadvantages stated above.

A computational unit is a collection of instructions following the *read-compute-write* pattern: a set of variables is read by a collection of instructions and is used to perform a computation, then the result is written back to another set of variables. The two sets of variables are called *read set* and *write set*, respectively. These two sets do not necessarily have to be disjoint. Load instructions reading variables in the read set form the *read phase* of the CU, and store instructions writing variables in the write set form the *write phase* of the CU.

A CU is defined by read-compute-write pattern because in practice, tasks communicate with one another by reading and writing variables that are global to them, and computations are performed locally. Thus, we require the variables in a CU's read set and the write set to be global to the CU. The variables local to the CU are part of the compute phase of the CU as they will not be used to communicate with other tasks during parallelization. To distinguish variables that are global to a code section, we perform variable scope analysis, which is available in any ordinary compiler. Note that the global variables in the read set and the write set do not have to be global to the whole program. They can be local to an encapsulating code section, but global to the target code section.

The CUs defined above can be built for any granularity: instruction level, basic block level, or even function level. In this paper, CUs are built for every *region*. A region is a single entry single exit code block. The difference between a region and a basic block is that not every instruction inside a region is guaranteed to be executed, meaning a region could be a group of basic blocks with branches inside. A region can be a function, a loop, an if-else structure, or a basic block. In our earlier approach, we worked on fine-grained parallelism due to the less-powerful definition of CUs. In this paper, we focus on thread-level parallelism. Thus we are interested in regions like functions and loops, which contain important computations that can potentially run in parallel.

3.2.2. Cautious property

According to the definition of CUs in Section 3.2.1, CUs are *cautious*. Cautious property (Pingali et al., 2011) was previously defined for operators in unordered algorithms: an operator is said to be cautious if it reads all the elements of its neighborhood before it modifies any of them. By adapting it to the CU, we say a code section is cautious if every variable in its read set is read before it is written in the write phase.

Cautious property is an alternative representation of the read-compute-write pattern. Not only it gives a clear way of separating

```
for each region R in the program do
    globalVars = variables that are global to R
    isCautious = true
    for each variable v in globalVars do
        if v is read then
            readSet += v
            for each instruction Irv reading v do
                readPhase += Irv
            end
        end
        if v is written then
            writeSet += v
            for each instruction Iwv writing v do
                writePhase += Iwv
            end
        end
    end
    for each variable v in readSet do
        for each instruction Ir reading v do
            for each instruction Iw writing v do
                if Ir happens after Iw then
                    isCautious = false
                end
            end
        end
    end
    if isCautious then
        cu = new computational unit
        cu.scope = R
        cu.readSet = readSet
        cu.writeSet = writeSet
        cu.readPhase = readPhase
        cu.writePhase = writePhase
        cu.computationPhase =
        (instructions in R) − (readPhase + writePhase)
    end
    else
        for each read instruction Iv violating cautious property
        do
            build CU for instructions do not belong to any CU
            before Iv
        end
    end
end
```

Algorithm 1: Algorithm of building CUs.

read phase and write phase, but also allows multiple CUs to be executed speculatively without buffering updates or making backup copies of modified data because all conflicts are detected during the read phase. Consequently, tasks extracted based on CUs do not have any special requirement on runtime frameworks.

Algorithm 1 shows the algorithm of building CUs. For each region, we firstly get all variables that are global to the region, and classify them into read set (inputs) and the write set (outputs). Read phase and write phase of the region are built according to the read set and write set. Then we check if the read phase and write phase satisfy the cautious property. If so, the region is recognized as a computation unit.

It is possible that a variable is written but never read. Such case does not violate the cautious property. The cautious property is violated only when a global variable is firstly written and then read. In such a case, we call the read instruction that happens after write a *cautiousness violating point*.

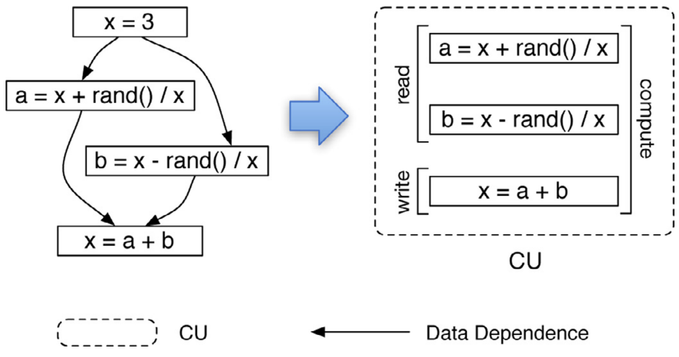


Fig. 4. Building a CU.

A region can be not cautious, meaning it contains more than one computational unit. Functions are usually cautious since most of them communicate to the outside program via predefined parameters and return values. However, loop bodies (especially C-style code) are usually not cautious. It is common that nearly all the variables accessed inside a loop body could be global to the loop. When a region is not cautious, we find the cautiousness violating points in the region, and break the region into multiple code sections according to the violating points. We then build CU for each of the code sections. In the end, a region that is not cautious contains multiple CUs. This process is shown in the last else branch in Algorithm 1.

3.2.3. Special variables in building CUs

Function parameters and return values deserve special treatment when determining read set and write set of a function. We treat them as follows:

- All function parameters are included in read set,
- Function parameters passed by value are *not* included in write set,
- Return value is stored in a virtual variable called `ret`, and `ret` is included in write set.

The first rule is obvious. We follow the second rule because parameters passed by value are copied into functions, thus modifications to them do not affect their original copies. The return value must be included in write set. However, it is common that the return value does not have a name. That is why we always call it `ret` when building CU statically.

Loop iteration variables also require special treatment. Concretely, following rules apply to them:

- By default, loop iteration variables are considered as local to loops,
- If a loop iteration variable is written inside the body of a loop, it is considered as global to the loop.

We treat loop iteration variables in a special way because inter-iteration dependences on them in loop headers do not prevent parallelism. However, if their values are updated inside loop body, the normal iteration process may be interrupted, and dependences on loop iterations variables must be checked in order to determine whether the loop can be parallelized.

3.2.4. Example of building CUs

To show that the new definition of CUs is compatible with the earlier one, we use the same example as the one used in earlier work (Li et al., 2013), and show that the new algorithm build the same CU. The example is shown in Fig. 4.

In this example, *readSet* and *writeSet* are both {x}. Each loop iteration calculates a new value of x by firstly reading the old value

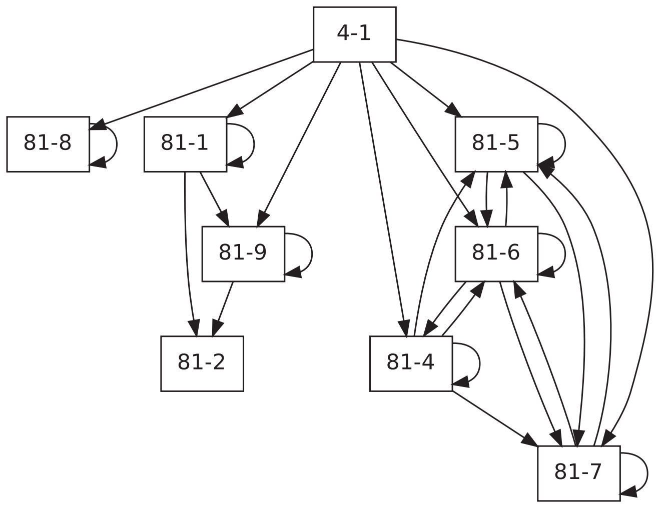


Fig. 5. Part of the CU graph of *sparselu*.

of x, then by computing a new value via local variables a and b. Finally, the new value is written back to x. For a single iteration, the loop region is cautious since all the reads of x happen before the write to x. Following the read-compute-write pattern, lines 3–5 are in one CU, as shown in Fig. 4. At source-line level, the compute phase (line 3–5) of the CU overlaps with its read phase (line 3–4) and write phase (line 5). At instruction level, the three phases are separate to one another. If a and b were declared outside the loop, then they would be considered global to the loop as well. This would mean the loop would be made up of two CUs with lines 3–4 being one CU and the line 5 being the second CU.

Note that CUs never cross region boundaries. Otherwise a CU could grow too large, possibly swallowing all the iterations of a loop and many other code sections, and hiding important parallelism that we actually want to expose.

3.2.5. CU graph

With CUs and data dependences, we represent a program using a *CU graph*, in which vertexes are CUs and edges are data dependences. Data dependences in a CU graph are always among read phases and write phases of CUs. Dependences that are local to a CU are hidden because they do not prevent parallelism among CUs.

CUs are built statically. A CU represents a computation. In execution, it is possible to have multiple "instances" of the same computation, like two calls to the same function with different parameters. Since data dependences are obtained during execution, it is effectively recording dependences between instances of CUs, and then merge the instances of the same CU together. Although this solution may miss some parallelism, producing complete "CU instance" graph is nearly impossible because of the high overhead in terms of both time and memory. To preserve as much information as we can, data dependence properties like "inter-iteration" or "intra-iteration" are kept so that parallelism discovery algorithms can work properly.

Fig. 5 shows a part of the CU graph of *sparselu* (Duran et al., 2009). CU IDs (strings in vertexes) are in the format of *File_ID - Local_CU_ID*. Edges (data dependences) are always from source to sink. When statement *B* depends on statement *A*, we call *A* the source of the dependence, and *B* the sink.

3.2.6. Detecting parallelism

Our earlier approach (Li et al., 2013; 2015a; Ul-Huda et al., 2015) identifies three kind of parallelism: DOALL loops, independent tasks, and pipelines. With the new definition of CUs, we improve the parallelism discovery algorithms to cover the following parallelism: DOALL loops, DOACROSS loops, SPMD tasks, and MPMD tasks. Both DOACROSS loops and MPMD tasks can lead to pipelines, but we distinguish them in the new approach since it is possible to apply different implementation techniques other than pipeline for SPMD tasks. Moreover, the new classification covers both independent and dependent tasks. In a word, the new algorithms cover more parallelism, and classify them in a more detailed way.

DOALL loops. A loop can be categorized as a DOALL loop if there is no inter-iteration dependence. For nested loops, whether an inner loop is DOALL or not does not affect outer loops. This is the easiest type of parallelism to be discovered since it only needs to check if there is an inter-iteration dependence among the CUs belong to the body of the target loop.

When checking inter-iteration dependences, we check read-after-write (RAW) dependences only. The condition is relaxed because usually inter-iteration write-after-read (WAR) and write-after-write (WAW) dependences do not prevent parallelism (suppose a variable is always assigned a new value at the beginning of each iteration). This may lead to false positives, but we expect that false positives are rare. Thus, our algorithm detecting DOALL loops is optimistic.

Note that data dependences on iterating variables are already taken care by the special treatment described in Section 3.2.3.

DOACROSS loops. When a loop has inter-iteration dependences, it is possible to further analyze the dependence distances of the inter-iteration dependences to discover DOACROSS (Kennedy and Allen, 2002) loops. A DOACROSS loop has inter-iteration dependences, but the dependence are not between the first line of an iteration and the last line of the previous iteration. This means in a DOACROSS loop, iterations are not independent but can partly overlap of one another, providing parallelism that can be utilized by implementing reduction or pipeline. Dependence distances can be easily checked since data dependences are indexed by source line numbers. For a non-DOALL loop, we classify it as a DOACROSS loop if there is no inter-iteration dependence that from the read phase of the first CU (in single-iteration execution order) to the write phase of the last CU of the loop body. Note that the first CU and the last CU can be the same.

SPMD task parallelism. As its name suggests, Single-Program-Multiple-Data (SPMD) tasks execute the same code but work on different data. It is similar to data decomposition. To identify SPMD task parallelism, it only needs to check if a CU depends on itself.

Note that iterations in a DOALL loop can also be considered as SPMD task parallelism. However, since DOALL loops can usually be parallelized using specialized mechanisms that are more efficient (like `#pragma parallel for` in OpenMP, and `tbb::parallel_for()` in TBB), we categorize DOALL loops separately. In this paper, SPMD task parallelism refer to independent calls to the same function with different parameters, possibly combined with recursive pattern.

MPMD task parallelism. In contrast to SPMD task parallelism, Multiple-Program-Multiple-Data (MPMD) tasks execute different code. Once tasks are allowed to execute different code, identifying only independent tasks is not sufficient. Multiple MPMD tasks that are dependent of one another may lead to a pipeline, or even task graph, unless the dependences form a circle. Thus, we report MPMD task parallelism if dependences among CUs that belong to target code sections do not form a circle. That is said, MPMD task parallelism is the most general type of parallelism we identify in this paper.

Note that the implementation of MPMD task parallelism can be different, and the resulting performance varies. When task graph is implemented, the performance is greatly determined by the power of the scheduler in use.

Our approach is semi-automatic at the moment, which means it does not generate the parallel code automatically. It is a program-understanding approach, and it leaves the implementation part to the users, including the correctness validation of the found parallelism. We do so because fully automatic sequential-to-parallel code transformation techniques for general purpose code are still in research. So far, the best working automatic approaches are for specific cases, such as polyhedral-based parallelizing methods for well formed, perfectly nested loops (Grosser et al., 2012) and decoupled software pipelining for DOACROSS parallelism (Ottoni et al., 2005).

It is very difficult to predict whether data races would be introduced during parallelization. In many cases, parallelization means (partially) redesigning the algorithm. It is possible that an object is accessed only once in the sequential program but must be accessed by all the threads in the parallel version. It is also possible that there is a data structure in the parallel version that does not even exist in the sequential program. Data race detection in parallel code (Jannesari et al., 2009; Cai and Cao, 2015; Sorrentino et al., 2010) is a well-known difficult research topic, and we believe that predicting possible data races in the resulting parallel code from analyzing sequential code is even harder.

3.3. Phase 3: Ranking

Ranking parallelization opportunities of the target program helps users to focus on the most promising ones. Three metrics are involved: *instruction coverage, local speedup*, and *CU imbalance*.

3.3.1. Instruction coverage

The instruction coverage (IC) provides an estimate of how much time will be spent in a code section. The estimation is based on the simplifying assumption that each kind of instruction costs about the same amount of time. Given a code section i and the whole program P,

$$IC(i) = \frac{N_{inst}(i)}{N_{inst}(P)}$$

where $N_{inst}(i)$ and $N_{inst}(P)$ are the number of instructions of code section i and the whole program P, respectively. Note that N_{inst} always represents the total number of instructions which are really executed at runtime. For a loop, N_{inst} is the sum of the number of instructions across all iterations.

3.3.2. Local speedup

The local speedup (LS) reflects the potential speedup that would be achieved if a code section was parallelized according to the suggestion. Since it refers only to a given code section and not necessarily to the whole program it is called local. The local speedup is based on the critical path (the longest series of operations that have to be performed sequentially due to data dependences) and Amdahl's Law, which is why super linear effects are not considered.

Given a code section i of the target program:

$$LS(i) = min\left(N_{threads}, \frac{N_{inst}(i)}{length(CP(i))}\right)$$

where $N_{inst}(i)$ is the total number of instructions of code section i, and length(CP) is the length of the critical path of i—again, based on the assumption that each kind of instruction costs the same amount of time. $N_{threads}$ is the number of threads. If the local speedup exceeds the number of threads, it will be just equal to the number of threads.

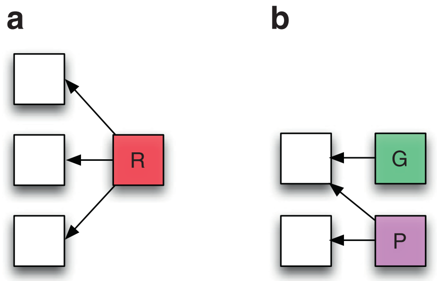


Fig. 6. Scenarios with different degrees of CU imbalance.

3.3.3. CU imbalance

The CU imbalance reflects how evenly CUs are distributed in each stage of the critical path, which means whether every thread has some work to do in each step of the computation. Otherwise, some of the threads have to wait because of data dependences, which means the suggested parallelization may have a bottleneck. We define the CU imbalance for a code section i as

$$CI(i) = \frac{\sigma(i)}{MP(i)}$$

where $\sigma(i)$ is the standard deviation of the number of CUs in each stage of the critical path, and $MP(i)$ is the number of CUs in the largest stage of the critical path of node i. The CU imbalance is a value in $[0, +\infty)$. The more balanced the CU ensemble is, the smaller the value becomes.

Fig. 6 provides an example. Under the assumption that each CU has the same number of instructions, both of situations have a local speedup of two and will complete all the tasks in two units of time. However, the arrangement in Fig. 6(a) requires three threads while 6(b) requires only two. The red CU (R) in 6(a) needs the results from three CUs, constituting a bottleneck of the execution. Although the purple CU (P) in 6(b) is in the same situation, the other thread still has some work to do (green CU, G) so that it does not need to wait. The CU imbalance values of the two situations (6(a): $\sqrt{2}/3 = 0.47$, 6(b): $0/2 = 0$) reflect such a difference. Note that a code section containing no parallelism (CUs are sequentially dependent) will also get a CU imbalance of zero, which is consistent with our definition.

Our ranking method now works as follows: Parallelization opportunities are ranked by their estimated global speedup (GS) in descending order, with

$$GS = \frac{1}{\frac{IC}{LS} + (1 - IC)}.$$

Should two or more opportunities exhibit the same amount of global speedup, they will be ranked by their CU imbalance in ascending order. Note that since LS is never bigger than the number of threads and IC is always smaller than 1, GS can never exceeds the number of threads, either.

4. Evaluation

We conducted a range of experiments to evaluate the effectiveness of our approach. We applied our method on benchmarks in Barcelona OpenMP Task Suite (BOTS) (Duran et al., 2009), PARSEC benchmark (Bienia, 2011), NAS Parallel Benchmarks (NPB) (Bailey et al., 1991), and Starbench benchmark (Andersch et al., 2011). All four benchmark suites contain sequential benchmark applications as well as their equivalent parallel versions. After applying our method on the sequential benchmark applications, we compare the

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

Benchmark	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

identified parallelization opportunities to the existing parallel versions in order to evaluate our approach. For the opportunities that do not have corresponding parallel version, we implemented our own parallel version for these applications.

Our approach is implemented in LLVM 3.6.1 (Lattner and Adve, 2004), and all benchmarks are compiled using Clang 3.6.1 (Lattner, 2011) with `-g -O0` for instrumentation, and `-O2` for execution. Experiments were run 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). The performance results reported are an average five independent executions. Whenever possible, we tried different inputs to compensate for the input sensitivity of data-dependence profiling approach, resulting more complete data dependences for each benchmark.

4.1. Identification of DOALL loops

The purpose of the first experiment was to detect DOALL loops and see how the approximation in data dependence profiling affects the accuracy of the suggestions on parallelism. We took our test cases from the NAS Parallel Benchmarks (NPB) 3.3.1, a suite of programs derived from real-world computational fluid dynamics applications. The suite includes both sequential and OpenMP-based parallel versions of each program, facilitating a quantitative assessment of our tool's ability to spot potential loop parallelism. 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 loops that we identified as parallelizable are listed under "# parallelizable". At this stage, prior to the ranking, DiscoPoP considers only data dependences, which is why still many loops carrying no dependence 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, proving the effect of the signature approximation to be negligible. 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 (Ketterlin and Clauss, 2012), the authors reported an average of 86.5% after applying their tool to SPEC OMP-2001. Kremlin (Garcia et al., 2011), 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).

Table 2
Detection of DOACROSS loops in benchmarks from Starbench and NAS.

Benchmark		Exec. time [%]	DOACROSS	Implemented	# CUs
Starbench	rgbyuv	99.9	✓	pipeline (DOALL)	5
	tinyjpeg	99.9	✓	pipeline	2
	kmeans	99.5	✓	reduction	4
BOTS	nqueens	~100	✓	reduction	1
NAS	CG	96.9	✓	reduction	4
	BT	99.1	✗	–	–
	SP	99.1	✗	–	–
	FT	49.3	✗	–	–
	MG	49.8	✗	–	–

4.2. Identification of DOACROSS loops

For loops that are not DOALL, we further analyze the dependence distance of inter-iteration dependences in them. It is obvious that parallelizing small loops (in terms of workload) with inter-iteration dependences is not beneficial. Thus we focus on non-DOALL loops that are hotspots in terms of execution time. Table 2 summarizes the biggest non-DOALL loops in benchmarks from Starbench (Andersch et al., 2011), BOTS (Duran et al., 2009), and NAS. Recall that a non-DOALL loop is classified as a DOACROSS loop if there is no inter-iteration dependence that from the read phase of the first CU (in single-iteration execution order) to the write phase of the last CU of the loop body. As shown in Table 2, target loops in *BT, SP, FT*, and *MG* are not DOACROSS loops. Column "Implemented" shows the implementation mechanism in official parallel versions.

Among the loops that are identified as DOACROSS, two (*rgbyuv, tinyjpeg*) are suitable for pipeline implementation while the other three (*kmeans, nqueens*, and *CG*) can be parallelized with reduction. As we mentioned before, the implementation choice has to be made by the users. However, distinguishing which implementation is the best for a DOACROSS loop is relatively easy since the inter-iteration dependences are reported. We verified that the DOACROSS loops identified in *tinyjpeg* is implemented as pipelines in official parallel implementation. However, the target loop in *rgbyuv* is an interesting case.

Case study—rgbyuv. The target loop in *rgbyuv* is in bmark.c, line 151. The source code of the loop is shown in Fig. 7. The target loop has five CUs: CU_1 (line 2), CU_2 (line 3), CU_3 (line 4), CU_4 (line 6–8), and CU_5 (line 10–12). The CU graph of the loop body is shown on the left side in Fig. 8. Obviously, CU_1, CU_2, and CU_3 are too small, so we consider them as a single computation without hiding parallelism, leading to the simplified CU graph shown on the right side in Fig. 8.

At the beginning we know nothing about the code of *rgbyuv*, just as every programmer that parallelizes sequential code written by someone else. Simply following the simplified

```
for(int j = 0; j < args->pixels; j++) {
    R = *in++;
    G = *in++;
    B = *in++;

    Y = round(0.256788*R + 0.504129*G +
  0.097906*B) + 16;
    U = round(-0.148223*R - 0.290993*G
  + 0.439216*B) + 128;
    V = round(0.439216*R - 0.367788*G -
  0.071427*B) + 128;

    *pY++ = Y;
    *pU++ = U;
    *pV++ = V;
}
```

Fig. 7. The target loop in *rgbyuv* (bmark.c, line 151).

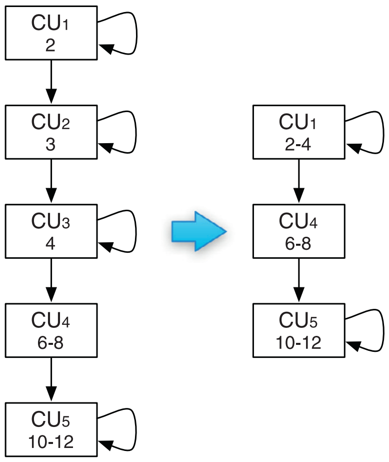


Fig. 8. CU graphs of the loop body of the loop in *rgbyuv* (bmark.c, line 151).

CU graph in Fig. 8, we know the loop can be parallelized as a three-stage pipeline. Since CU_1 and CU_5 have self-dependences, the first stage and the third stage has to be sequential stage, while the second stage can be a parallel stage. We implement the pipeline using Intel TBB (Reinders, 2007). Each stage is implemented as a filter class, and stages are connected using `tbb::parallel_pipeline`. Moreover, `filter::serial_in_order` attribute is specified for stage 1 and 3. In a word, everything was done following the output of our tool, and we did not bother understanding the code. Note that the loop body is not considered as a whole due to the source code control structure. The loop body is divided into three tasks, and the task boundaries are not aligned with the loop boundaries.

The best performance of our implementation appears when using 4 threads, with a speedup of 2.29. Using more threads than the number of stages of a pipeline usually do not give better performance, especially when most of the stages are sequential. When examining the official parallel implementation of *rgbyuv*, we found that the target loop is parallelized as DOALL, not DOACROSS. This means the inter-iteration dependences on CU_1 and CU_5 do not prevent parallelism. This is true because the inter-iteration dependences are on pointers (`in`, `pY`, `pU`, and `pV`), not the data to which they point. Thus, to utilize the DOALL parallelism it just need to make the pointers local.

This example shows that simply following the output of our tool yields good speedup, and understanding the code is still important. Nevertheless, our tool reveals interesting parallelization opportunities and data dependences that potentially prevent parallelism, helping the users to achieve a better implementation much faster.

```
*solutions = 0;
for (i = 0; i < n; i++) {
    a[i] = (char) i;
    if (ok(i + 1, a)) {
        nqueens(n, i + 1, a,&res);
        // reduction
        *solutions += res;
    }
}
```

Fig. 9. Function `nqueens()` from *nqueens* of BOTS.

Table 3
Detection of SPMD task parallelism in BOTS benchmarks.

Benchmark	Function	Exec. time [%]	SPMD task	Implemented
sort	sort	74.9	✗	✗
	cilksort	74.8	✓	✓
	seqmerge	52.0	✗	✗
	cilkmerge	34.4	✓	✓
	seqquick	22.6	✗	✗
fib	fib	~100	✓	✓
fft	fft	~100	✗	✗
	fft_aux	97.2	✓	✓
	fft_twiddle_16	83.0	✓	✓
	fft_unshuffle_16	12.7	✓	✓
floorplan	add_cell	~100	✓	✓
health	sim_village	~100	✓	✓
sparselu	bmod	89.6	✗	✗
	sparselu	34.4	✓	✓
strassen	OptimizedStrassenMultiply	95.2	✓	✓
	MultiplyByDivideAndConquer	82.0	✓	✓
	FastAdditiveNaiveMatrixMultiply	61.9	✗	✗
	FastNaiveMatrixMultiply	21.4	✗	✗
uts	serTreeSearch	99.6	✓	✓

DOACROSS loops identified in *kmeans, nqueens* and *CG* are implemented using reduction in the official parallel implementations. As an example, the target loop in *nqueens* is shown in Fig. 9. The inter-iteration dependence is due to the variable `solutions`, which is a classic scenario of reduction. The DOACROSS loops in kmeans and *CG* are similar to the example shown above, but the code is more complicated.

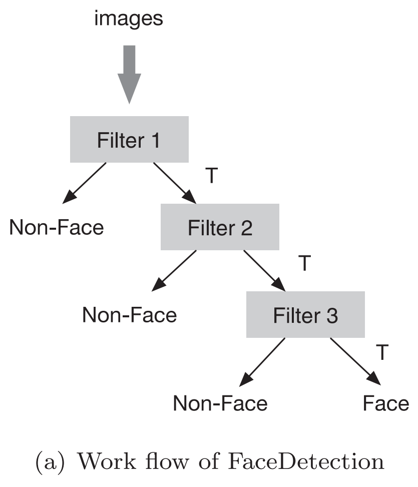
4.3. Identification of SPMD tasks

To evaluate the detection of SPMD tasks, we apply our method on benchmarks from Barcelona OpenMP Task Suite (BOTS) (Duran et al., 2009). We choose BOTS benchmarks because they are parallelized using OpenMP tasks, providing many opportunities of finding SPMD tasks. Recall that to identify SPMD tasks, we check if a CU depends on itself. Due to the large number of CUs in a program, we focus on CUs that correspond to functions only, and functions that are hotspots in terms of execution time. Table 3 shows the results of detecting SPMD tasks in BOTS benchmarks.

SPMD tasks are found in eight BOTS benchmarks containing 19 functions that are hotspots. 12 functions are identified as SPMD tasks, which are all parallelized in the official parallel implementations. Note that none of the SPMD tasks shown in Table 3 come from loops. The common pattern of these tasks is that there are multiple calls to the same function with different arguments. In many benchmarks like *sort, fib*, and *fft*, computations are performed recursively. At each recursion level, multiple SPMD tasks are created.

Table 4
Detection of MPMD tasks in PARSEC benchmarks, libVorbis, and FaceDetection.

Benchmark	Function	Implemented	Solution	# threads	Speedup
blackscholes	CNDF	✗	omp sections	–	–
canneal	routing_cost_given_loc	✗	omp sections	–	–
fluidanimate	RebuildGrid	✗	omp sections	–	–
fluidanimate	ProcessCollisions	✗	omp sections	–	–
fluidanimate	ComputeForces	Data decomposition	Pipeline	3	1.52
libVorbis	main (encoder)	✗	Pipeline	4	3.62
FaceDetection	facedetector	Pipeline	Pipeline	32	9.92



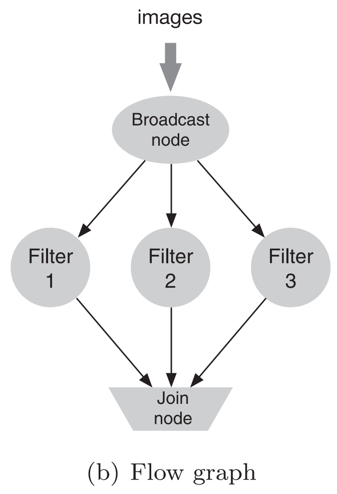


Fig. 10. Work flow of FaceDetection and the corresponding flow graph.

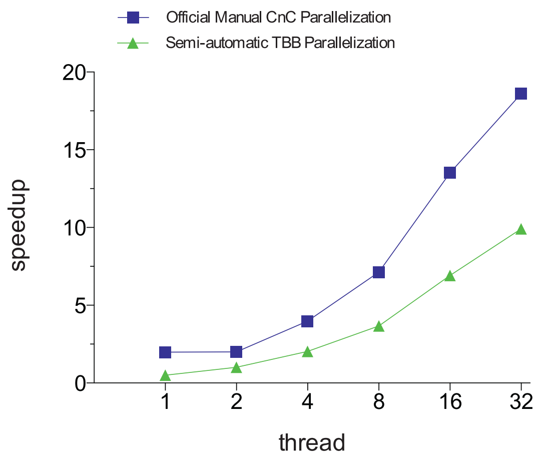


Fig. 11. FaceDetection speedups with different threads.

4.4. Identification of MPMD tasks

To evaluate the detection of MPMD tasks, we applied our method on PARSEC benchmarks (Bienia, 2011) and two other applications: the open-source Ogg codec *libVorbis* and an Intel Concurrent Collections (CnC) sample program *FaceDetection*. In contrast to SPMD tasks that widely exist in BOTS benchmarks, MPMD tasks execute different code. Generally speaking, programs containing MPMD tasks perform multiple kinds of computations rather than a single computation on big input data. This implies that to discover MPMD tasks, it is better to focus on programs that have more lines of code (LOC). Moreover, it is well known that pipeline and flow graph patterns are common in multimedia processing applications. That is why the programs we use in this section are generally bigger in terms of code size, and two programs that process audio (*libVorbis*) and image (*FaceDetection*) are included.

Table 4 summarizes the results of evaluating the detection of MPMD task. As the results show, MPMD tasks are not the main type of parallelism in simulating applications (*blackscholes, canneal*, and *fluidanimate*). All the MPMD tasks found in these programs are from non-hotspot computations. They are not parallelized in official parallel implementations, and parallelizing them using `omp section` does not give any speedup. The only interesting place in these programs is the `ComputeForces` function in *fluidanimate*. The parallelization story, however, is similar to the case study shown in Section 4.2. We parallelized the function body following the output CU graph using TBB and achieved a speedup of 1.52 using three threads. On the contrary, the official parallel version of *fluidanimate* shows this function is parallelized using data decomposition, yielding almost linear speedup.

Case study—FaceDetection. Face Detection is an abstraction of a cascade face detector used in the computer vision community. The face detector consists of three different filters. As shown in Fig. 10(a), each filter rejects non-face images and lets face images pass to the next layer of cascade. An image will be considered a face if and only if all layers of the cascade classify it as a face. The corresponding TBB flow graph is shown in Fig. 10(b). A join node is inserted to buffer all the boolean values. In order to decide whether an image is a face, every boolean value corresponding to that specific image is needed. Thus we configure the transformation tool to use `tag_matching` buffering policy in the join node. `tag_matching` policy creates an output tuple only when it has received messages at all the ports that have matching keys.

The three filters take 99.9% of sequential execution time. We use 20,000 images as input. The speedup of our transformed flow graph parallel version is 9.92 $\times$ using 32 threads. To evaluate the scalability of the automatically transformed code, we compare the speedups achieved by official Intel CnC parallel version and our

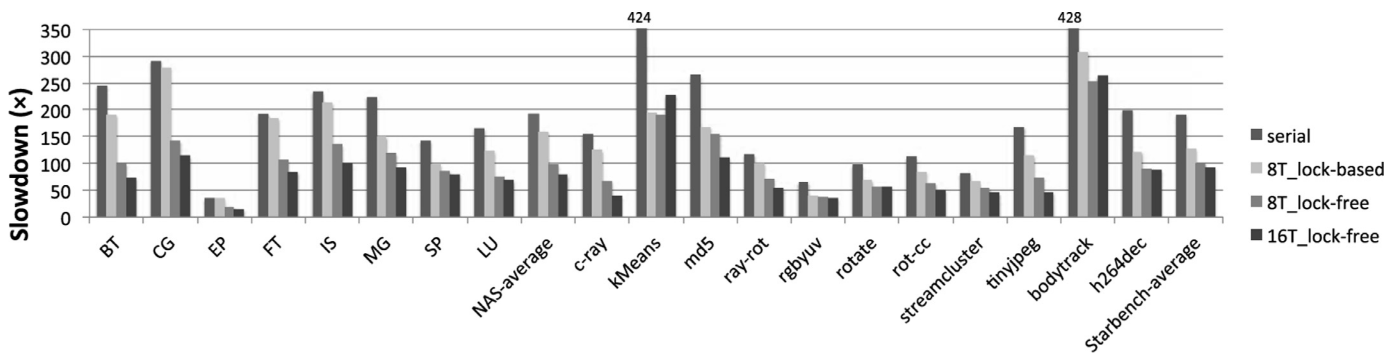


Fig. 12. Slowdowns of data dependence profiler for sequential NAS and Starbench benchmarks.

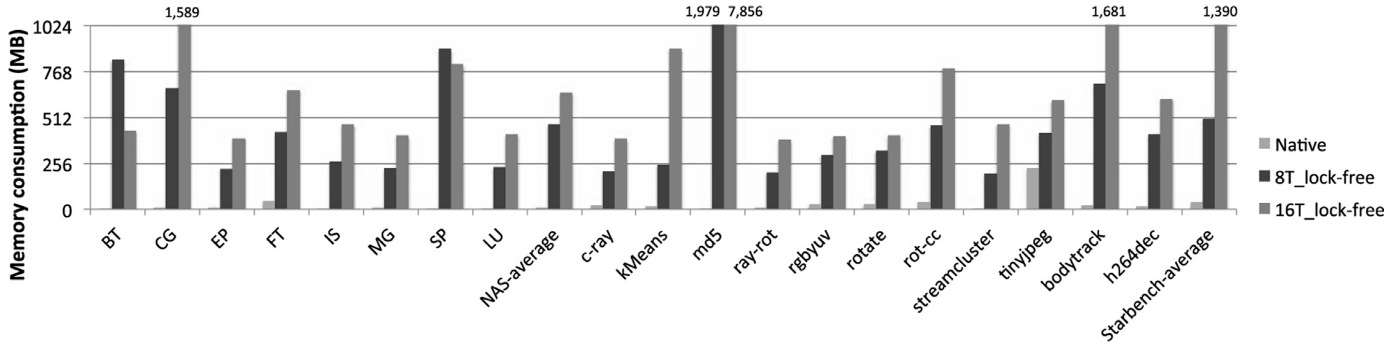


Fig. 13. Memory consumption of the profiler for sequential NAS and Starbench benchmarks.

transformed TBB flow graph version using different number of threads. The result is shown in Fig. 11. The performance is comparable using two and four threads. When more than eight threads are used, the official CnC parallel version outperforms ours. The reason is that the official CnC parallel code is heavily optimized and restructured. For example, some data structures are altered from `vector` to CnC `item_collection`. As shown in Fig. 11, when using just one thread, the speedup of official CnC parallel version is already 2 × because of the optimization.

The remaining application, 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.

4.5. Ranking method

We also evaluated the precision of our ranking method. The results are shown in Table 1. Column "# in top 30%" lists the number of suggestions matched by actual parallelization in the OpenMP version (# identified) that end up in the top thirty percent after ranking. We believe that only few programmers would examine all the suggestions one by one and that for most the first 30% would be the upper limit. As one can see, 70.6% (96/136) of the matched suggestions can be found in the top 30%. This means by examining only 30% of the suggestions, 70% of the actually implemented parallelism can be explored.

We also verified whether the top 10 suggestions for each program are really parallelized in the official OpenMP version. The results are listed in the column "# in top 10". For most of the programs, more than a half (for some of them even 90%) of the top 10 suggestions are parallelized, proving the effectiveness of our ranking method.

4.6. Overhead

In the last experiment, we measured the time and memory consumption of DiscoPoP, which are mainly incurred by the data-dependence profiler. The results are obtained by profiling NPB 3.3.1 with input size W and Starbench with the reference input.

4.6.1. 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.

The slowdown of our profiler when profiling sequential programs is shown in Fig. 12. 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.

4.6.2. Memory consumption

We measure memory consumption using the "maximum resident set size" value provided by `/usr/bin/time` with the verbose (`-v`) option. Fig. 13 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 latest definition of computational units (CUs) and embedding it in a framework of combined static and dynamic analysis, we can reconcile the identification of DOALL loops, DOACROSS loops, SPMD tasks, and MPMD tasks with efficiency both in terms of time and memory. Our approach found 92.5% of the parallel loops in NAS Parallel Benchmark (NPB) programs and successfully identified different types of tasks at different level of language constructs. Furthermore, we provide an effective ranking method, selecting the most appropriate parallel opportunities for the user. Our results show that 70% of the implemented parallelism in NPB can be explored by examining only the top 30% of our suggestions. Last but not the least, our method incurs only 78 × slowdown and 649 MB memory when profiling NAS benchmarks because of the highly efficient data-dependence profiler.

In the future, we want to extend our static analyses to reduce the number of instrumented functions—with the aim of further lowering the time and space overhead. On parallelism discovery side, we tend to explore more potentials of CUs to support different parallel programming models.

References

Andersch, M., Juurlink, B., Chi, C.C., 2011. A benchmark suite for evaluating parallel programming models. In: Proceedings 24th Workshop on Parallel Systems and Algorithms, pp. 7–17.

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., 1991. The NAS parallel benchmarks. International Journal of High Performance Computing Applications 5 (3), 63–73. doi:10.1177/109434209100500306.

Bienia, C., 2011. Benchmarking Modern Multiprocessors. Princeton University Ph.D. thesis.

Cai, Y., Cao, L., 2015. Effective and precise dynamic detection of hidden races for java programs. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE). ACM, New York, NY, USA, pp. 450–461.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E., 2009. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: International Conference on Parallel Processing, ICPP'. IEEE, pp. 124–131.

Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B., 2011. Kremlin: Rethinking and rebooting GPROF for the multicore age. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 458–469. doi:10.1145/1993498.1993553.

Grosser, T., Groesslinger, A., Lengauer, C., 2012. Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Process. Lett. 22 (04), 1250010. doi:10.1142/S0129626412500107.

Jannesari, A., Bao, K., Pankratius, V., Tichy, W.F., 2009. Helgrind+: An efficient dynamic race detector. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. IEEE Computer Society, pp. 1–13. doi:10.1109/IPDPS.2009.5160998.

Johnson, R.E., 2010. Software development is program transformation. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. ACM, pp. 177–180. doi:10.1145/1882362.1882400.

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

Ketterlin, A., Clauss, P., 2012. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, pp. 437–448. doi:10.1109/MICRO.2012.47.

Kim, M., Kim, H., Luk, C.-K., 2010a. Prospector: Discovering parallelism via dynamic data-dependence profiling. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism.

Kim, M., Kim, H., Luk, C.-K., 2010b. SD3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, pp. 535–546. doi:10.1109/MICRO.2010.49.

Lattner, C., 2011. LLVM and Clang: Advancing compiler technology. In: Proceedings of the 2011 Free and Open Source Developers' European Meeting (FOSDEM), pp. 1–2.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. IEEE Computer Society, Washington, DC, USA, p. 75.

Li, Z., Atre, R., Ul-Huda, Z., Jannesari, A., Wolf, F., 2015a. DiscoPoP: A profiling tool to identify parallelization opportunities. In: Tools for High Performance Computing 2014. Springer International Publishing. chapter 3, pp. 37–54. doi:10.1007/978-3-319-16012-2.

Li, Z., Jannesari, A., Wolf, F., 2013. Discovery of potential parallelism in sequential programs. In: Proceedings of the 42nd International Conference on Parallel Processing. IEEE Computer Society, pp. 1004–1013.

Li, Z., Jannesari, A., Wolf, F., 2015b. An efficient data-dependence profiler for sequential and parallel programs. In: Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium, pp. 484–493.

Moseley, T., Shye, A., Reddi, V.J., Grunwald, D., Peri, R., 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In: Proceedings of the 5th International Symposium on Code Generation and Optimization. IEEE Computer Society, Washington, DC, USA, pp. 198–208. doi:10.1109/CGO.2007.35.

Ottoni, G., Rangan, R., Stoler, A., August, D.I., 2005. Automatic thread extraction with decoupled software pipelining. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, pp. 105–118. doi:10.1109/MICRO.2005.13.

Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., Sui, X., 2011. The tao of parallelism in algorithms. SIGPLAN Not. 46 (6), 12–25. doi:10.1145/1993316.1993501.

Reinders, J., 2007. Intel Threading Building Blocks. O'Reilly Media.

Sanchez, D., Yen, L., Hill, M.D., Sankaralingam, K., 2007. Implementing signatures for transactional memory. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, pp. 123–133. doi:10.1109/MICRO.2007.24.

Sorrentino, F., Farzan, A., Madhusudan, P., 2010. PENELOPE: Weaving threads to expose atomicity violations. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). ACM, New York, NY, USA, pp. 37–46.

Ul-Huda, Z., Jannesari, A., Wolf, F., 2015. Using template matching to infer parallel design patterns. ACM Trans. Archit. Code Optim. (TACO) 11 (4), 64:1–64:21. doi:10.1145/2688905.

Vanka, R., Tuck, J., 2012. Efficient and accurate data dependence profiling using software signatures. In: Proceedings of the 10th International Symposium on Code Generation and Optimization. ACM, New York, NY, USA, pp. 186–195. doi:10.1145/2259016.2259041.

Zhang, X., Navabi, A., Jagannathan, S., 2009. Alchemist: A transparent dependence distance profiling infrastructure. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society, pp. 47–58. doi:10.1109/CGO.2009.15.

Zhen Li is currently a PhD student in the Laboratory for Parallel Programming at the Department of Computer Science of Technische Universität Darmstadt, Germany. He received his Bachelor and Master degrees in Computer Science from Harbin Institute of Technology, China. From 2011 to 2015, he studied at RWTH Aachen University, Germany. He moved to Darmstadt on February, 2015. His research interests include program analysis, efficient profiling techniques, data-dependence analysis, and parallelism discovery in sequential programs.

Rohit Atre is a PhD candidate at RWTH Aachen University in Germany. He is researching the use of static and dynamic analysis to identify parallelization in sequential programs. His research interests include task and loop parallelism, automatic parallelization, and static/dynamic analysis of sequential programs on multicore architectures. He completed his Bachelor of Technology in Computer Science & Engineering from India, and M.Sc. in Business Information Systems from France.

Zia Ul Huda received BS in Computer Science from University of the Punjab Lahore in 2006 and MS in Computer Science from University of Bonn in 2012. He is currently a PhD student at the Laboratory for Parallel Programming, Technische Universität Darmstadt, Germany. His research interests include parallel pattern detection, identification of parallelization opportunities and semi-automatic parallelization. He is a member of the IEEE.

Ali Jannesari is a research scientist at University of California, Berkeley. He is also associated as head of the Multicore Programming group at TU Darmstadt. His research focuses on program analysis, parallelism and software engineering. He is particularly interested in multicore systems and taking advantage of parallel computation for different applications. He was a junior research group leader at RWTH Aachen University and also a PostDoc fellow at Karlsruhe Institute of Technology and Bosch Research Center. Ali has a PhD in computer science from Karlsruhe Institute of Technology. He is a member of the IEEE Computer Society and ACM.

Felix Wolf is a full professor at the Department of Computer Science of Technische Universität Darmstadt in Germany, where he is head of the Laboratory for Parallel Programming. He specializes in methods, tools, and algorithms to exploit massive parallelism on modern computer architectures. His most recent contributions are in the areas of automatic performance modeling, adaptive scheduling, and parallelism discovery. He has published more than a hundred refereed articles on parallel computing, several of which have received awards.