# Beyond Data Parallelism: Identifying Parallel Tasks in Sequential Programs

Zhen Li[1(✉)], Bo Zhao[2], Ali Jannesari[1], and Felix Wolf[1]

[1] Technische Universität Darmstadt, 64289 Darmstadt, Germany
{li,jannesari,wolf}@cs.tu-darmstadt.de
[2] Xi'an Jiaotong University, Xi'an 710049, China
zhaobo36@stu.xjtu.edu.cn

**Abstract.** Today, millions of legacy programs are awaiting their parallelization. For this reason, the automatic discovery of parallelism in sequential programs is now receiving considerable attention. However, past efforts mainly concentrated on data parallelism hidden inside loops. As programming models begin to support more irregular types of parallelism, centered around the notion of tasks in various forms, methods are needed to identify code sections that could potentially represent parallel tasks. In this paper, we present a novel approach to automatically finding parallel tasks in sequential programs. We first created a dynamic dependence graph, then isolated tasks, and finally produced a task graph according to the dependences we find. With the help of a source-to-source code translator, parallel code is automatically generated. We conducted a range of experiments to cover both tasks executing the same code and tasks executing different code. Results showed that our method achieved reasonable speedups on the test cases.

**Keywords:** Parallelism discovery · Task parallelism · Computational unit · Data dependence · Parallel programming

## 1 Introduction

While writing parallel programs from scratch has always been considered a difficult task, parallelizing legacy programs written by someone else, today a common scenario in many organizations, is even harder [8]. For this reason, many methods have been proposed to assist programmers in parallelizing sequential programs. The most attractive idea is to build a compiler that automatically translates a sequential into a parallel program. Such compilers support a set of directives that the programmer has to insert into the source code to mark sections that can run in parallel. Although this approach requires only minor changes to the source code, it leaves an important but time-consuming job to the programmer: finding parallelism in the sequential program.

To support programmers also in this initial stage of the process, methods have been proposed to discover potential parallelism automatically. So far, their main target has been data parallelism in loops, which can be exploited by distributing


G. Wang et al. (Eds.): ICA3PP 2015, Part IV, LNCS 9531, pp. 569–582, 2015.
DOI: 10.1007/978-3-319-27140-8_39

iterations of a loop among multiple threads. However, as more programming models such as OpenMP and Intel TBB [18] aim at task-based parallelism, this original focus of parallelism discovery becomes too narrow. In contrast to loop-based data parallelism, task parallelism does not require that every thread to execute the same code. Tasking can exploit parallelism between arbitrary code sections, including parallelism within individual iterations of a loop or between different loops.

In this paper, we present an approach to the detection of parallel tasks in sequential programs. As the first step, we run our data-dependence profiler [14] to extract the dynamic data-dependence graph. This graph is then transformed into another graph, whose edges represent only true data dependences and whose nodes are small pieces of computation without any noteworthy internal parallelism. We call these nodes *computational units* (CUs) and we call the graph *CU graph*. Then we search the graph for *strongly connected components* (SCCs) and *chains*, we merge the CUs they contain, and label them as potential tasks. Finally, we feed the generated task graph to a code transformation component that can transform serial C/C++ code into Intel TBB [18] parallel code. The code transformation component then translates the sequential source code into equivalent parallel code using TBB flow graph template.

The remainder of the paper is structured as follows. In the next section, we review related work and highlight the differences to our own. In Sect. 3, we explain our approach in more detail. Evaluation results and case studies are presented in Sect. 4. Finally, Sect. 5 summarizes our paper and discusses possible improvements.

## 2 Related Work

Methods for assisting parallelization mainly fall into one of two not necessarily disjoint categories. Methods in the first category focus primarily on data dependence analysis to find parallelism, whereas methods in the second category put more emphasis on the runtime system as their primary vehicle of parallelization.

***Dynamic Dependence Analysis.*** 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 dependence to asses the degree of potential parallelism. Using dependence information, Kremlin [6] 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. Finally, Kremlin reports self-parallelism for each region, in the same way as an ordinary performance profiler such as gprof would report the time. Alchemist [20] is centered around the notion of futures, treating predefined constructs as candidates for asynchronous execution. It profiles the dependence distance (the number of instructions between the source and sink of a dependence) to estimate the effectiveness of parallelizing a certain construct. AutoFutures [15] adopts a similar idea, but goes one step

further in that it automatically transforms the code. However, it seems to be still at a preliminary stage with negative speedup results reported for some of the test programs. Previous work [5] identified task parallelism in C applications for multiprocessor System-on-Chip (MPSoC) platforms based on the notion of a coupled block, which is a group of statements tightly coupled by dependences. A coupled block is treated as a task.

All of the approaches mentioned above discover parallelism from massive raw data dependences without respecting computation patterns. They overlook the truth that the computation of a task usually does not contain any noteworthy parallelism inside, and the set of variables used for communication among tasks usually does not overlap with the set of variables used for computations.

Other approaches primarily concentrate on the efficiency of profiling dependences. Parwiz [10] is an optimized data-dependence profiler that attaches the dependences it finds to the nodes of an execution tree (i.e., a generalized call tree that also includes basic blocks) that it maintains. Based on this execution tree, it can identify DOALL [9] loops in sequential programs. Prospector [11] is a parallelism-discovery tool based on the memory-efficient data dependence profiler SD3 [12]. It tells whether a loop can be parallelized and provides a detailed dependence analysis of the loop body.

***Scheduling.*** Runtime scheduling frameworks are another way of adding parallelism to sequential programs. DSWP [16] and DOMORE [2] target DOACROSS [9] loops, scheduling their iterations in a pipeline style according to previously identified (static) dependences. Anantpur and Govindarajan [7] profile cross-iteration dependences for DOACROSS loops and try to accelerate their execution using GPUs. Ye and Chen [19] profile data dependences on the superblock level. Using a meta-reorder buffer to measure and exploit the available parallelism, superblocks are dynamically analyzed, reordered, and dispatched, respecting data dependences.

Generally, scheduling techniques incur a non-negligible fixed overhead, which changes very little if the number of data dependences in the program varies. Moreover, most scheduling approaches focus solely on DOACROSS loops, missing the potential parallelism outside such loops. While scheduling approaches do not require any effort on the part of the programmer, the above limitations often render the speedup they can achieve inferior to manual parallelization.

## 3 Approach

Our approach consists of the following steps. First, we profile the target program to extract the dynamic data-dependence graph. This graph then undergoes two transformations aimed at isolating tasks and dependences between them. During both transformations, nodes are merged to simplify the graph structure. At the end, some nodes may emerge as independent or dependent tasks. Finally, we submit the generated task graph to a code transformation component that transforms serial C/C++ code into Intel TBB parallel code.

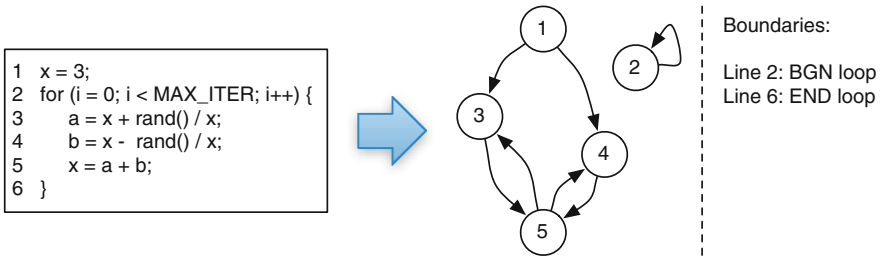



**Fig. 1.** Data-dependence graph and control-structure boundaries produced by DiscoPoP. Vertices are source lines and edges are data dependences.

### 3.1 Extracting Data Dependences

To generate the data-dependence graph, we use an efficient dynamic dependence profiler [14]. The nodes of the graph are source-code lines and the edges are data dependences between them. Figure 1 shows the profiler's output for a sample code section. The output also includes control-structure boundaries and the names of variables involved in dependences. Of course, relying on dynamic dependences makes our approach input-sensitive. However, the effects of the input sensitivity can be ameliorated by (i) running the target program with a range of inputs and merge the outputs (ii) letting the user specify a representative input that covers the typical execution flow.

### 3.2 Identifying Computational Units

The first transformation of the dependence graph isolates small pieces of code without any noteworthy internal parallelism, which we call *computational units* (CUs). A CU is built for a collection of instructions following the *read-compute-write* pattern: a set of variables is read by a collection of instructions and used to perform computation, then the result is written back to another set of variables. We call the two sets *read set* and *write set*, respectively. The two sets do not have to be disjoint. The load instructions reading variables in the read set form *read phase* of the CU, and the store instructions writing variables in the write set form *write phase* of the CU.

We define a CU by read-compute-write pattern because in practice, tasks communicate with one another by reading and writing values to variables that are global to them. Thus, we require the variables in a CU's read set and write set to be global to the CU, and the variables used in a CU's computation should be local. To distinguish variables that are global to a code section, we analyze variable scope information, which is available in any ordinary compiler. Note that the global variables in read set and write set do not have to be global to the whole program. They can be variables that are local to an encapsulating code section, but global to the target code section.

**Algorithm 1.** Algorithm of building CUs (pseudocode).

```
1  for each region R in the program do
2      globalVars = variables that are global to R
3      isCautious = true
4      for each variable v in globalVars do
5          if v is read then
6              readSet += v
7              for each instruction Irv reads v do
8                  readPhase += Irv
9              end
10         end
11         if v is written then
12             writeSet += v
13             for each instruction Iwv writes v do
14                 writePhase += Iwv
15             end
16         end
17     end
18     for each instruction Ir in readPhase do
19         for each instruction Iw in writePhase do
20             if Ir happens after Iw then
21                 isCautious = false
22                 break
23             end
24         end
25     end
26     if isCautious then
27         cu = new computational unit
28         cu.scope = R
29         cu.readSet = readSet
30         cu.writeSet = writeSet
31         cu.readPhase = readPhase
32         cu.writePhase = writePhase
33         cu.computationPhase =
34         (instructions in R) - (readPhase + writePhase)
35     end
36 end
```

We further require that the load and store instructions in read phase and write phase are *cautious* [17]. Cautious property is previously defined for operators in unordered algorithms. By adapting it to CU, we say a CU is cautious if it reads all the variables in its read set before it writes any variables in its write set. Cautious property guarantees the read-compute-write pattern. It does not only give a clear way of separating read phase and write phase, but also allows multiple CUs to be executed speculatively without buffering updates or

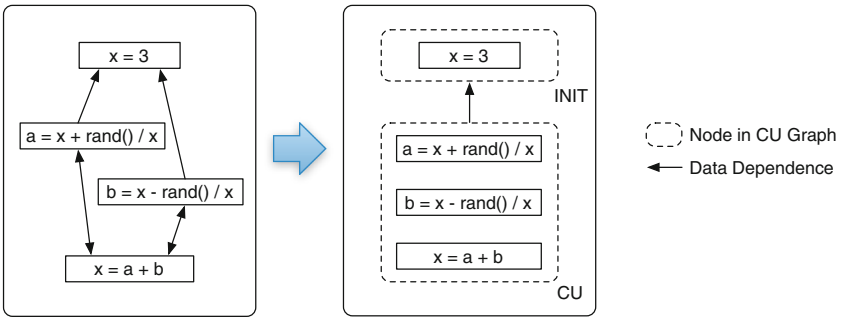



**Fig. 2.** Building a CU.

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.

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 practice, a basic block rarely contains noteworthy parallelism because it usually contains a small number of instructions. Code in different branches of an if-else structure are semantically exclusive, thus rarely run in parallel. Hence, we mainly focus on regions like functions and loops, which usually contain important computations that can potentially run in parallel. In our approach, regions of a program are traversed by implementing the algorithm of building CUs shown in Algorithm 1 using the *region pass* in LLVM [13].

Figure 2 shows a CU built from the code section shown in Fig. 1. Each loop iteration calculates a new value of `x` with the help of local variables `a` and `b`. For a single iteration, the loop region is cautious since all the read to `x` happen before the write to `x`. Following the read-compute-write pattern, lines 3–5 are in one CU, and the CU depends on the initialization of `x`, as shown in Fig. 2. Note that CUs never cross control-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 parallelisms that we actually want to expose.

Identifying CUs simplifies the dependence graph by not only merging vertices into CUs, but also hiding dependences that are local to the computations of CUs. After identifying CUs, edges in the dependence graph are all inter-CU dependences, which are always among instructions in read phases and write phases. Giving the truth that the number of global variables to a code section is usually far less than the number of local variables, identifying CUs delivers a significant simplification for the dependence graph. We call the simplified dependence graph *CU graph*.

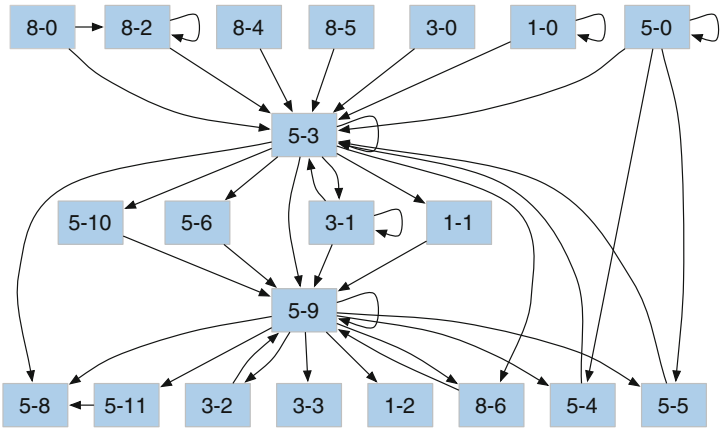



**Fig. 3.** Part of the CU graph of *rot-cc*.

Figure 3 shows a part of the CU graph of *rot-cc*, a benchmark from Starbench parallel benchmark suite [1]. According to the figure, although it is quite clear that some CUs can run in parallel (e.g. 8–4 and 8–5), it still requires effort to tell whether other CUs can run in parallel. Thus, we simplify CU graph further into a directed acyclic graph (DAG), which we call *task graph*.

### 3.3 Forming Tasks

The second transformation of the dependence graph helps identify either independent or dependent tasks, the latter in a potential pipeline arrangement. Whenever possible, we merge CUs contained in *strongly connected components* (SCCs) or in *chains*. The idea of merging CUs in SCC comes from previous work [16]. In graph theory, an SCC is a subgraph in which every vertex is reachable from every other vertex. Thus, every CU in an SCC of the CU graph depends on every other CU either directly or indirectly, forming a complex knot of dependences that is likely to defy internal parallelization. Identifying SCCs is important for two reasons:

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

Hence, we hide complex dependences inside SSCs, exposing parallelization opportunities outside, where only a few dependences need to be considered. Figure 4 shows the graph simplification process by substituting SCCs and chains

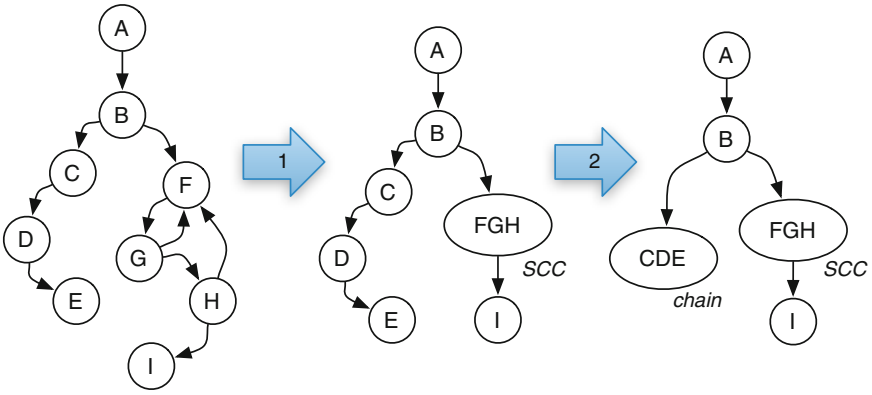



**Fig. 4.** Simplifying CU graph by substituting SCCs and chains of CUs with vertices.

of CUs with vertices. In step 1, CU $F$, $G$ and $H$ are grouped into $SCC_{FGH}$. After contracting each SCC to a single vertex, the graph becomes a directed acyclic graph. Moreover, we group CUs that are connected in a row without a branching or joining point in between into a *chain* of CU since a chain of CU does not contain significant parallelism inside, and merging them can lower the communication overhead among tasks. In step 2, CU $C$, $D$ and $E$ are grouped into $chain_{CDE}$. We call the simplified graph *task graph*.

Finally, we declare each vertex in the task graph a potential task. If the task graph has more than one entries, a virtual task ($task_0$) is added to be the predecessor of all the entry nodes. The virtual task ensures that a task graph has only one entry node, which simplifies the code transformation algorithm mentioned below.

### 3.4 Automatic Code Transformation

In the end, we submit the generated task graph to a code transformation component that transforms serial C/C++ code into Intel TBB [18] parallel code. Transformation is performed at AST level using Clang libraries. The transformation module traverses the Clang AST of the source code in order to locate the code sections targeted by the task graph. Afterwards, a source code rewriting module rewrites the targeted source code strings in the Clang AST context using TBB flow graph templates. The transformation component also supports DOALL loops. A DOALL loop is transformed into a TBB `parallel_for` template with its loop body filled as a lambda expression.

The flow graph transformation algorithm is divided into three steps:

Step 1: **Identifying Code Sections Corresponding to Each Task**. For each task in the task graph, the transformation module gets all its source code lines via the AST context and save them to *CU.codeBody* in the corresponding task. Note that the virtual task does not correspond to any code section.

Step 2: **Generating Source Code of the Flow Graph Node**. The source code rewriting module generates the flow graph node based on the following three cases:

- The current task has a single or none incoming edge and multiple outgoing edges. If all its successors receive the same data, we insert a TBB `broadcast_node`. Otherwise, a TBB `split_node` is inserted. When there is no input data for `broadcast_node` or `split_node`, the node template uses type `continue_msg` defined in TBB. Otherwise the corresponding data types must be obtained via AST and be passed to the template.
- The current task has a single incoming edge and single or none outgoing edge. In this case, the source code rewriting module directly transforms it to a flow graph `function_node` using a lambda function.
- The current task has multiple incoming edges and single out going edge, which means at least two variables need synchronization before they are passed to the current task. Hence, we must first add a `join_node` to synchronize the operations on these variables and then insert the `function_node`. A `join_node` has multiple input ports and generates a single output tuple that contains a value received from each port.

Step 3: **Generating Source Code of Flow Graph Edges**. After all of the flow graph nodes have been defined in the source code, the corresponding code for edges must be added according to the task graph.

It's worth mentioning that `join_node` supports three different buffering policies: `queueing`, `reserving`, and `tag matching`. Currently the buffering policy need to be determined by users, because it is usually semantic related and can not be solved by our tool. When all the nodes and edges have been defined, the transformation terminates and the parallel code is complete.

## 4 Evaluation

As we mentioned in Sect. 1, task parallelism does not require that every thread to execute the same code. Like loop-based data parallelism, tasking can certainly exploits parallelism among iterations of a loop (each task executes the same code). However, tasking also exploits parallelism between different loops and functions (each task executes different code). In this section, we show that our method can handle both kinds of task parallelism.

We have mentioned that the buffering policy of TBB `join_node` needs to be determined by users. Another thing that has to be determined by users is the data chunk size when data decomposition is needed. They are the only two things our method requires. Determining data chunk size automatically requires an auto-tuning technique, which is beyond the scope of this paper.

We conducted a range of experiments to evaluate our method. All experiments ran on a server with $2 \times 8$-core Intel Xeon E5-2650 2 GHz processors with 32 GB memory, running Ubuntu 12.04 (64-bit server edition). Time and speedup numbers represent an average of five independent executions.

### 4.1 Tasks Executing the Same Code

When tasks execute the same code, it means the parallelism comes from data decomposition. In sequential code, such parallelism usually resides in loops, where each iteration perform computation on a piece of input data. To determine whether a loop in sequential code can be parallelized, we only need to check if the partial CU graph of the loop has no circle, including edges that come from a CU and point to itself. Otherwise, an iteration of the loop reads data produced in the previous iteration, and the loop cannot be parallelized.

Programs containing loops where each iteration can be emitted as a task are easy to find. In our experiments, we chose three benchmarks (*BT*, *SP*, and *CG*) from NAS parallel benchmark suite [3], *blackscholes* from PARSEC benchmark suite [4], and two applications (*mandelbrot*, *ann training*) that are commonly used in parallel programming courses.

Instead of transforming each iteration into a `tbb::task`, we use `tbb::parallel_for` for loops because it utilize the thread pool in TBB for better efficiency. On the other hand, `tbb::task` always creates a new thread for a task.

**Table 1.** Summary of parallelization results for tasks executing the same code.

| Program | # parallel loops | | # of threads | Speedup(auto) | Speedup(manual) |
|---|---|---|---|---|---|
| | auto | manual | | | |
| BT | 22 | 30 | 16 | 2.17 | 6.78 |
| SP | 26 | 34 | 16 | 2.03 | 5.07 |
| CG | 5 | 16 | 16 | 2.15 | 8.36 |
| blackscholes | 3 | 1 | 16 | 3.19 | 7.12 |
| mandelbrot | 2 | 2 | 4 | 2.02 | 3.96 |
| ann training | 4 | 2 | 4 | 1.91 | 3.07 |

Table 1 shows the results on parallelizing tasks residing in loops. In general, our method exploits fewer parallelism than experienced programmers and results in lower speedups, which is common to all of the automatic parallelization approaches. In *blackscholes* and *ann training*, our method parallelized more loops than the manually parallelized versions. However, all the additional loops that are automatically parallelized are small loops doing initialization. Parallelizing such loops does not bring any speedup, but rather incurs additional overhead in creating and destroying threads.

An interesting case is *mandelbrot*, where our method parallelized exactly the same places as the programmer did. However, the automatic parallelized version still has a lower speedup due to imbalanced workload. In *mandelbrot*, whether the matrix is divided row-wise or column-wise gives a different workload to each worker thread. Unfortunately, there is no way to get such information before

running a parallel version of the program. This case shows that although automatic parallelization method can bring some speedup for free, user's knowledge is still critical for a higher speedup.

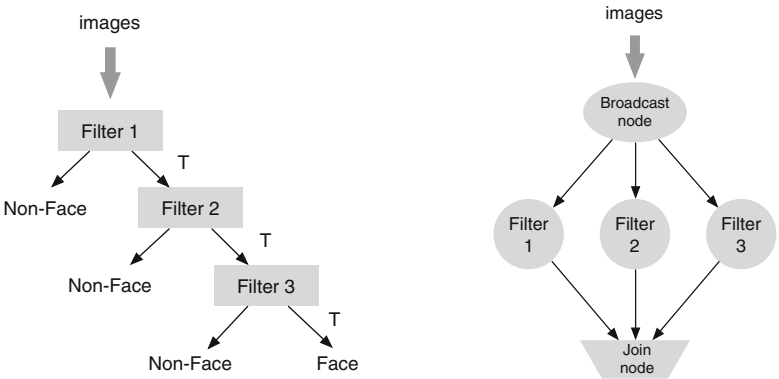
### 4.2 Tasks Executing Different Code

In our experiments, we found that applications containing task parallelism that different tasks run different code are mainly from multimedia processing area. Thus, we chose Intel CnC sample program FaceDetection, and Ogg Vorbis codec libVorbis as representative cases.

**FaceDetection.** FaceDetection 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. 5(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. 5(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× using 32 threads. To evaluate the scalability of the automatically transformed code, we compare the speedups achieved by official Intel CnC (short for "Concurrent Collections") parallel version and our transformed TBB flow graph version using different number of threads. The result is shown in Fig. 6. 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. 6, when using just one thread, the speedup of official CnC parallel version is already 2× because of the optimization (Fig. 6).

**LibVorbis.** We also tested the encoder of LibVorbis, a reference implementation of the Ogg Vorbis codec. In contrast to previous test cases, it contains a pipeline pattern, which is a special case of task flow graph. Four `function_nodes` are constructed for the four-stage pipeline, and our automatic version achieved a speedup of 2.41 with four threads. We got a lower speedup mainly because the code transformation tool uses task flow graph to mimic pipeline, which is less efficient than the specialized `tbb::pipeline` class. This test case enlightened us to improve our code transformation component to support pipeline pattern. We consider this task as a future work (Table 2).




(a) Work flow of FaceDetection (b) Flow graph

**Fig. 5.** Work flow of FaceDetection and the corresponding flow graph

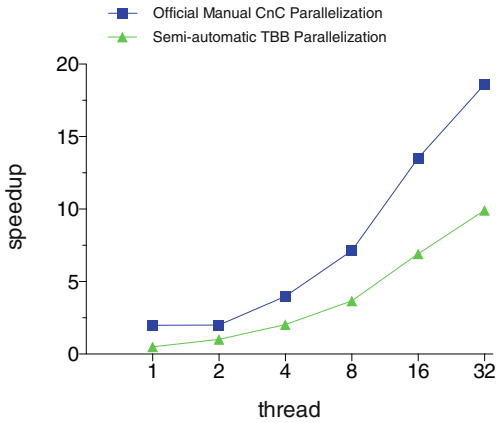



**Fig. 6.** FaceDetection speedups with different threads

**Table 2.** Summary of parallelization results for tasks executing different code.

| Program | Function | % of time | # of threads | Speedup(auto) | Speedup(manual) |
|---|---|---|---|---|---|
| FaceDetection | facedetector | 99.9 | 32 | 9.92 | 18.60 |
| LibVorbis | main (encoder) | 100.0 | 4 | 2.41 | 3.62 |

## 5 Conclusion and Outlook

Many efforts have been made to find potential parallelism in sequential programs. However, most of them focused on loop-based data parallelism. In this paper, we propose a novel method that can identify parallel tasks—another promising source of parallelism but harder to find because more irregular. We extract the

dependence graph dynamically from the program and subject it to several (simplifying) transformations at the end of which the tasks emerge. The main idea of this paper is the identification of computational units (CUs) and the localization of strongly connected components (SCC) and chains as representations of potential tasks. While CUs hide dependences that are local to computations, SCCs encapsulate complex dependences inside a task. The generated task graph is further submitted to a code transformation component that translate the sequential code into parallel TBB code. Experiment results showed that for both tasks executing the same code and different code, reasonable speedup is secured. The bottom line is that, we believe this work closes a gap in parallelism discovery technology, which is especially important for the systematic parallelization of larger general-purpose application portfolios, a challenge many organizations are facing today.

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

## References

1. Andersch, M., Juurlink, B., Chi, C.C.: A benchmark suite for evaluating parallel programming models. In: Proceedings 24th Workshop on Parallel Systems and Algorithms, PARS 2011, pp. 7–17 (2011)
2. August, D.I., Huang, J., Beard, S.R., Johnson, N.P., Jablin, T.B.: Automatically exploiting cross-invocation parallelism using runtime information. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, pp. 1–11. IEEE Computer Society (2013)
3. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
4. Bienia, C.: Benchmarking Modern Multiprocessors. Ph.D. thesis, Princeton University, January 2011
5. Ceng, J., Castrillon, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T., Kunieda, H.: Maps: an integrated framework for mpsoc application parallelization. In: Proceedings of the 45th Annual Design Automation Conference, DAC 2008, pp. 754–759. ACM (2008)
6. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: Rethinking and rebooting gprof for the multicore age. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 458–469. ACM (2011)
7. Govindarajan, R., Anantpur, J.: Runtime dependence computation and execution of loops on heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, pp. 1–10. IEEE Computer Society (2013)
8. Johnson, R.E.: Software development is program transformation. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 177–180. ACM (2010)

9. Kennedy, K., Allen, J.R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers Inc., San Francisco (2002)
10. Ketterlin, A., Clauss, P.: Profiling data-dependence to assist parallelization: framework, scope, and optimization. In: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 45, pp. 437–448. IEEE Computer Society (2012)
11. Kim, M., Kim, H., Luk, C.K.: Prospector: discovering parallelism via dynamic data-dependence profiling. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism, HOTPAR 2010 (2010)
12. Kim, M., Kim, H., Luk, C.K.: SD3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 43, pp. 535–546. IEEE Computer Society (2010)
13. Lattner, C., Adve, V.: 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, CGO 2004, pp. 75–86. IEEE Computer Society, Washington(2004)
14. Li, Z., Jannesari, A., Wolf, F.: An efficient data-dependence profiler for sequential and parallel programs. In: Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium, IPDPS 2015, pp. 484–493 (2015)
15. Molitorisz, K., Schimmel, J., Otto, F.: Automatic parallelization using autofutures. In: Pankratius, V., Philippsen, M. (eds.) MSEPT 2012. LNCS, vol. 7303, pp. 78–81. Springer, Heidelberg (2012)
16. Ottoni, G., Rangan, R., Stoler, A., August, D.I.: Automatic thread extraction with decoupled software pipelining. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38, pp. 105–118. IEEE Computer Society (2005)
17. 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.: The tao of parallelism in algorithms. SIGPLAN Not. **46**(6), 12–25 (2011)
18. Reinders, J.: Intel Threading Building Blocks. O'Reilly Media, Sebastopol (2007)
19. Ye, J.M., Chen, T.: Exploring potential parallelism of sequential programs with superblock reordering. In: Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCC 2012, pp. 9–16. IEEE Computer Society (2012)
20. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2009, pp. 47–58. IEEE Computer Society (2009)