Dependence-Based Code Transformation for Coarse-Grained Parallelism

Bo Zhao[*,†,‡], Zhen Li[†,‡], Ali Jannesari[†,‡], Felix Wolf[†,‡], and Weiguo Wu[*]

[*]Xi'an Jiaotong University, Xi'an, China

[†]German Research School for Simulation Sciences, Aachen, Germany

[‡]RWTH Aachen University, Aachen, Germany

bo.zhao@rwth-aachen.de, {*z.li, a.jannesari, f.wolf*}*@grs-sim.de*, *wgwu@mail.xjtu.edu.cn*

ABSTRACT

Multicore architectures are becoming more common today. Many software products implemented sequentially have failed to exploit the potential parallelism of multicore architectures. Significant re-engineering and refactoring of existing software is needed to support the use of new hardware features. Due to the high cost of manual transformation, an automated approach to transforming existing software and taking advantage of multicore architectures would be highly beneficial. We propose a novel auto-parallelization approach, which integrates data-dependence profiling, task parallelism extraction and source-to-source transformation. Coarse-grained task parallelism is detected based on a concept called Computational Unit(CU). We use dynamic profiling information to gather control- and data-dependences among tasks and generate a task graph. In addition, we develop a source-to-source transformation tool based on LLVM, which can perform high-level code restructuring. It transforms the generated task graph with loop parallelism and task parallelism of sequential code into parallel code using Intel Threading Building Blocks (TBB). We have evaluated NAS Parallel Benchmark applications, three applications from PARSEC benchmark suite, and real world applications. The obtained results confirm that our approach is able to achieve promising performance with minor user interference. The average speedups of loop parallelization and task parallelization are 3.12x and 9.92x respectively.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

COSMIC '15, February 08 2015, San Francisco Bay Area, CA, USA

http://dx.doi.org/10.1145/2723772.2723777

General Terms

Languages, Performance

Keywords

Code transformation, source-to-source, data dependence analysis, TBB, coarse-grained parallelism

1. INTRODUCTION

Multicore and manycore architectures have become popular as a result of the stagnating single core performance. In order to take advantage of this trend, applications must be well parallelized. However, many existing software products are mostly written sequentially, so that they fail to exploit the parallelism of multicore architectures. Manually rewriting these legacy programs is time consuming and is a big economic challenge. Hence, effective tools and methodologies to parallelize existing software with minimum manual programming effort and user intervention are in great demand.

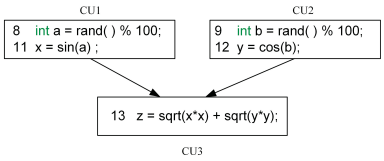
There are three main obstacles in the parallelization process of existing software: The first obstacle is to gain a thorough understanding of the code to identify control and data dependences. In order to guarantee correctness, the parallelized program must have proper synchronizations to preserve data dependences. The second obstacle is how to extract coarse-grained parallelism. Because multi-core processors are powerful in executing multiple code sections simultaneously, coarse-grained parallelism such as task parallelism is expected. The third obstacle is to generate parallel code which can express this coarse-grained parallelism effectively.

Previous works [7, 18] extract parallelism from DOALL and DOACROSS loops. Parallelizing research compilers such as SUIF [11], Polaris [21] and Open64 [1] have been put forward to generate parallel code. These approaches heavily rely on static analysis for identifying data dependences. Static approaches are conservative in finding parallelism due to the lack of runtime information. Thus they mainly focus on loop-level parallelism. Although these approaches do not target on coarse-grained parallelism such as task parallelism, the insight they provide is highly valuable to the parallelization effort.

In this paper, we propose a novel auto-parallelization approach that integrates data-dependence profiling, task paral-

```
1  #include <cmath>
2  #include <cstdlib>
3  using namespace std;
4
5  int main()
6  {
7      double x,y,z;
8      int a = rand() % 100;
9      int b = rand() % 100;
10
11     x = sin(a);
12     y = cos(b);
13     z = sqrt(x*x) + sqrt(y*y);
14
15     return 0;
16 }
```

(a) Source code



(b) CU graph

Figure 1: a sample CU example

lelism extraction and source-to-source transformation. Combining static analysis and dynamic analysis, we extract both DOALL loops and parallel tasks based on a graph, whose nodes are Computational Units (CU) [16] and edges are data dependences. After having the DOALL loops and parallel tasks, we developed a source-to-source transformation tool to generate parallel source code using Intel Threading Building Blocks(TBB) [2], which offers rich and complete libraries to express parallelism and does not need special compiler support. To evaluate our approach, we transformed NAS Parallel Benchmarks(NPB) [5], three applications from PARSEC benchmark suite [6] and real-world applications: Mandlebrot and FaceDetection. The average speedups of DOALL loop transformation and parallel task transformation are 3.12x and 9.92x respectively.

The rest of this paper is organized as follows. In the next section, we introduce necessary backgrounds, including DiscoPoP [16, 17], the program analysis tool we used to extract DOALL loops and parallel tasks, the concept of Computational Unit, and TBB. Our approach is explained in section 3, and evaluation results are presented in section 4. Section 5 summarizes related work and finally section 6 concludes this paper.

2. BACKGROUND

We use a LLVM [13]-based program analysis tool called DiscoPoP (=Discovery of Potential Parallelism) to help us find potential parallelism. It profiles programs and detects their control and data dependences. DiscoPoP is able to detect write-after-read (WAR), write-after-write (WAW) and read-after-write (RAW) dependences among variable accesses. Runtime control information such as entry and exit points of functions and number of iterations of loops are obtained dynamically. Based on the data dependence information, DOALL loops can be easily targeted.

2.1 Computational Unit

DiscoPoP also identifies code sections called Computational Unit (CU). Our transformation relies on these CUs. A CU is used as a building block for forming parallel tasks. It follows the read-compute-write pattern: a program state is first read from memory, the new state is computed and finally written back. We perform use-def analysis to determine CUs for every region [12]. A region is a subgraph of the control-flow graph that is connected to the remaining graph by only two edges, an entry edge and an exit edge. CUs are used for forming coarse-grained tasks. Dependences among CUs can be easily deducted from profiled dependences.

Figure 1 shows a simple example of CUs. Source line 8 performs initialization of the variable `a` with a random value. Line 11 reads `a`, performs some computation and writes the result to variable `x`. These lines together form CU_1. CU_2 is created using the same rules, consisting of lines 9 and 12. Line 13 writes the final computation to variable `z` using `x` and `y`, thus it forms CU_3. CU_1 and CU_2 can be executed independently, but CU_3 depends on CU_1 and CU_2, as is shown in Figure 1(b).

2.2 Program Execution Tree

DiscoPoP generates a Program Execution Tree (PET) which represents the executed code among a program in tree style. The root of the tree is the whole program. Internal nodes represent control structures and leaves are basic blocks– sections of code with only one entry point and only one exit point. As shown in Figure 2, CUs and their dependences are mapped to PET and thus form a CU graph. CUs are attached to leaf nodes, and dependences can exist both inside and between leaf nodes. Coarse-grained tasks are extracted from the CU graph.

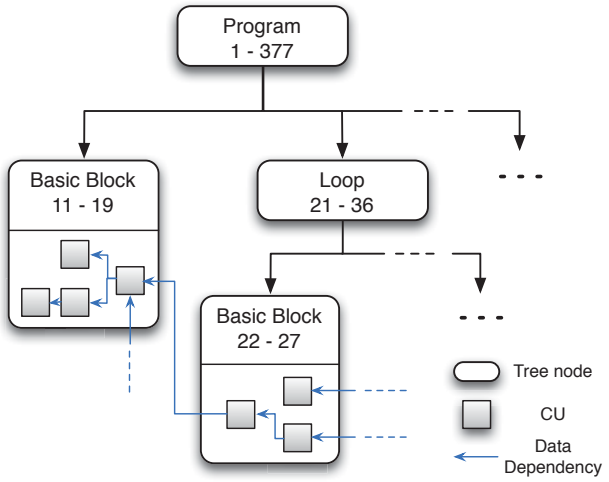


Figure 2: Example of CU graph

2.3 Threading Building Blocks

Intel Threading Building Blocks (TBB) is an open source C++ template library that targets shared memory parallel programming. It offers a range of high-level parallel primitives such as `parallel_for`, pipeline and *flow graph* to parallelize applications through the use of tasks rather than threads. Tasks are of short span and more lightweight than threads. The underlying runtime library is responsible for mapping tasks to threads in an efficient manner. TBB uses task stealing to balance the parallel workload across available processing cores in order to increase core utilization and scalability. In this paper, we mainly use `parallel_for` to

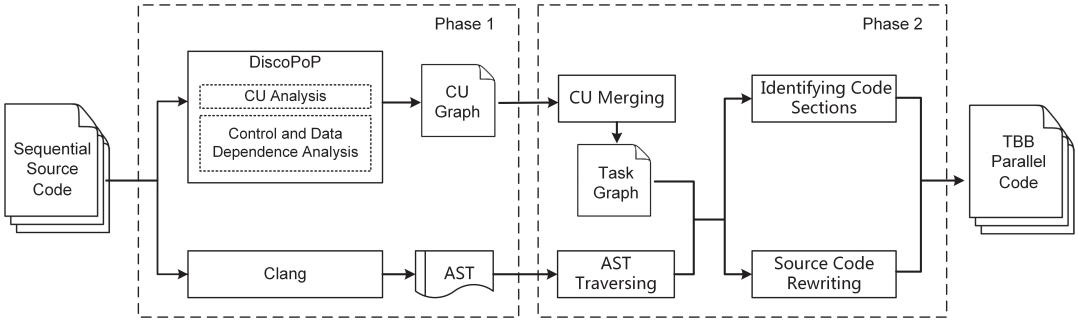


Figure 3: Workflow

generate parallel code for DOALL loops and flow graph to parallelize tasks.

The flow graph allows users to easily create both dependence graphs and reactive, messaging passing graphs that execute on top of Intel TBB tasks to expresses the control-flow in an application. It consists of three primary components: a graph object, nodes and edges. The graph object is the owner of the tasks created on behalf of the flow graph. Nodes are created to express the computations performed by the application. Edges can express the dependences between these computations. The Intel TBB library is able to exploit the parallelism that is implicit in the graph structure and the resources available on the target machine.

3. APPROACH

In this section, we present our transformation tool to transform serial C/C++ code into Intel TBB parallel code using `parallel_for` and flow graph. Figure 3 shows the workflow divided into two phases.

In the first phase, the program is analyzed by DiscoPoP. After the program is instrumented and executed, DOALL loops, CU graph, control flow information and all dependences through the whole program are obtained. At the same time, LLVM front end Clang [3] is used to parse the input source code into an abstract syntax tree (AST). This information is then sent to the code transformation module.

In the second phase, the transformation module merges CUs in the CU graph and generates a more coarse-grained task graph. Transformation is performed at AST level using Clang libraries. The transformation module traverses the Clang AST of the source code in order to locate DOALL loops and the code sections targeted by the task graph. Then the Source Code Rewriting module rewrites the targeted source code strings in the Clang AST context using TBB `parallel_for` and flow graph templates. The rest of this section explains these two phases in detail.

3.1 DOALL Loops

A DOALL loop can be parallelized because there are no loop-carried (inter-iteration) dependences. Figure 4(a) shows a simple DOALL loop. In this paper, we focus on for-loops, because the lower and upper bound of a for-loop can be easily extracted via AST. However, our approach can be easily extended to support while-loops and do-while-loops with the help of former techniques [14, 19, 9] to extract exit points of such loops.

3.1.1 Finding DOALL Loops

As the control flow within a loop iteration moves forward, a backward dependence is always loop-carried. After finishing profiling data dependence in phase 1, DiscoPoP performs a post-analysis to determine parallelizable loops. If there is no backward or self-dependence within a loop, the loop is definitely a DOALL loop. The simple example shown in Figure 4(a) is detected by DiscoPoP as a DOALL loop.

Algorithm 1: `parallel_for` loop transformations

```
   Input: parallelizable loop list, AST
   Output: transformed loops using tbb::parallel_for
1  get AST node astNode;
2  while astNode is not at the end of the AST do
3      if astNode is a for statement then
4          if astNode is in the parallelizable loop list then
5              get loop variable's name loopVarName and type
               loopVarType;
6              get loop upper bound UB;
7              get loop upper bound vriable's name UBName and
               type UBType;
8              get loop lower bound LB;
9              get loop lower bound vriable's name LBName and
               type LBType;
10             if loopVarName = UBName = LBName and
               loopVarType,UBType,LBType are compatible then
11                 rewrite this for loop using tbb::parallel_for
                   template;
12             end
13         end
14     end
15     astNode = astNode's child;
16 end
```

3.1.2 Transforming DOALL Loops

In phase 2, a for-loop is transformed using the procedure described in Algorithm 1. The target loops must follow the form of `for(initialization; condition; increase) loop_body`. The `condition` subexpression must be a binary operator. The loop boundaries and increament must not be changed during iterations. Clang libraries provide methods to extract the `initialization` subexpression, `condition` subexpression, `increase` subexpression, and the `loop_body`. Because the iteration space of `parallel_for` is a half-open interval `[begin,end)`, we need to consider loop boundaries when performing transformation and map the for-loop iteration space to a half-open interval form. The source-to-source translator traverses the AST in a recursive descent fashion, using AST node iterators to traverse each node's children. If the targeted loop is traversed, the Source Code Rewriting module is invoked. The transformation is divided into four steps. We use the for-loop shown in Figure 4(a) as a walk-through example to illustrate these four steps in detail.

Step 1: Determining loop variable and loop upper bound. Clang library method *getCond*() is used to

get the `condition` subexpression: `i < size`. It is a binary operator `"<"`, therefore we get the right operand `size` and the left operand `i`; `i` is the loop variable. Because of the binary operator `"<"`, the element `size` is excluded from the iteration space, which means that the iteration space is a half-open interval. Therefore, `size` is set to be the loop upper bound. It is then interpreted as the value "200,000" in compile time. If the binary operator is `"<="` or `">="`(the increment step is negative), that means the right operand `size` is included in the iteration space. In order to transform it to a half-open interval, the loop upper bound is set to be `size+1` or `size-1`.

Step 2: Determining loop lower bound. *getInit* () method is used to get the `initialization` subexpression: `int i=0`. It is an initialization statement. When we traverse the variable `i` whose type and name is identical to the loop variable found in step 1, we can make sure that `i` is the loop variable. Hence, the initialized value `0` is set to be the loop lower bound. The `initialization` subexpression could also be an assignment statement such as `i = 0`. In that case, the value in the right side of assignment operator is the loop lower bound. Because initialization and assignment are different types of nodes in Clang AST, they need to be checked separately.

Step 3: Determine the step of `parallel_for`. *getInc*() method is used to get the `increase` subexpression. In our example, it is a unary operator `"++"`. The step is accordingly set to be 1. If it is a unary operator `"--"`, the step is set to be -1. If it is a compound assignment operator such as `"i +=` *step*`"` or `"i -=` *step*`"`, the step is set to be *step* or $-$*step*. The more complicated situation is the assignment statement such as `"i = i +` *step*`"` or `"i = i -` *step*`"`. The transformation module first checks whether `"i"` is the loop variable and then sets the step to *step* or $-$*step*.

Step 4: Generating `parallel_for` code. *getBody*() is used to get `loop_body` and it is saved as a string. Because of implicit type conversion in C++, we need to check whether the loop variable types in `initialization`, `condition` and `increase` subexpressions are compatible. Then the Source Code Rewriting module deletes the original for-loop and inserts transformed code in the original for-loop's location using the `parallel_for` template and lambda functions in C++11. The transformed code is shown in Figure 4(b).

3.2 Task Graph

In order to achieve coarse-grained task parallelism, the transformation module extracts coarse-grained tasks from CU graph generated in phase 1 by DiscoPoP and generates the corresponding task graph. In phase 2, task graph is mapped to TBB flow graph templates. We use the code shown in Figure 1(a) as a walk-through example to illustrate our task graph transformation approach.

3.2.1 Building Task Graph

The problem with directly mapping CUs to TBB flow graph nodes is that a CU could be a subset of another CU or

```
#include <cmath>
#include <cstdlib>
using namespace std;

int main()
{
    const int size = 200000;
    ...
    for(int i=0; i<size; i++)
        output[i] = sqrt(sin(a[i])*sin(a[i]) + cos(a[i])*
            cos(a[i]));
    ...
    return 0;
}
```

(a) Before transformation

```
#include "tbb/parallel_for.h"
#include <cmath>
#include <cstdlib>
using namespace std;

int main()
{
    const int size = 200000;
    ...
    tbb::parallel_for(0,200000,1,[&](int i) {
      output[i] = sqrt(sin(a[i])*sin(a[i]) + cos(a[i])*cos
          (a[i]));
        });
    ...
    return 0;
}
```

(b) After transformation

Figure 4: Example of for-loop transformation

two CUs could have instructions in common. The reason is that more than one variable can use the same computation code blocks or more than one task can be performed by using the same partial code in the program. The XML file shown in Figure 5 is the result of CU analysis of our example shown in Figure 1(a). Each CU has a set of attributes including ID, source lines, instruction numbers and so on. Five CUs are extracted after CU analysis in phase 1 of our approach. They are CU_0:{9}, CU_1:{9,12}, CU_2:{11,12,13}, CU_3:{8} and CU_4:{8,11}. CU_0 and CU_3 are respectively subsets of CU_1 and CU_4. CU_1 and CU_2 have a common intersection at line 12. CU_4 and CU_2 have a common intersection at line 11.

To solve this problem, we merge CUs to more coarse-grained tasks according to the following rules:

Rule 1: If one CU is the subset of another CU, only the larger CU is considered to be a task.

Rule 2: If two CUs overlap with each other, the information about unique and common instructions between them is examined. Both the number of shared instructions and the number of unique instructions of each CU are calculated. If and only if the ratios of the number of common instructions to the number of each CU's unique instructions are both greater than a given threshold θ , these two CUs are merged into a coarse-grained task and mapped to a flow graph node. Based on empirical experiments, we found that 0.5 is the best threshold for the evaluation programs.

Using the above two rules for our example, CU_0 is merged to CU_1. CU_3 is merged to CU_4. The number of instructions in the intersection of CU_1 and CU_2 is one, thus the ratios of number of instructions at line 12 to the number of instructions of CU_1 and CU_2 are both less than 0.5. CU_1 and CU_2 are not merged. Because the ratio of number of instructions at line 12 to the number of instructions of CU_1 is greater than the ratio of number of instructions at line 12 to the number of instructions of CU_2, line 12 is excluded from CU_2. The same applies for CU_4 and CU_2. Finally,

```
<CU id="1:3" instr_count = "4">
  <lines count="1">1:8</lines>
  <variableDefined count_uses_itself = "1">a</
      variableDefined>
  <computeInstructions>0x31d2db8,0x31d3f48,0x31d4050,0
      x31d4100,</computeInstructions>
  <usedVariables>rand,call,rem,a,</usedVariables>
</CU>
<CU id="1:0" instr_count = "4">
  <lines count="1">1:9</lines>
  ...
</CU>
<CU id="1:4" instr_count = "5">
  <lines count="2">1:8,1:11</lines>
  ...
</CU>
<CU id="1:1" instr_count = "5">
  <lines count="2">1:9,1:12</lines>
  ...
</CU>
<CU id="1:2" instr_count = "12">
  <lines count="3">1:11,1:12,1:13</lines>
  ...
</CU>
```

Figure 5: The sample CU analysis result

we get three merged CUs : CU'_1 {8,11}, CU'_2 {9,12} and CU'_3 {13}, as shown in Figure 1(b). Merged CUs' IDs are reordered according to the line numbers. Each merged CU is a task.

When detecting dependences between tasks, we reorganize the dependence results from DiscoPoP as tuples of source line numbers and variable names in form of `<line1,line2, var1,var2,...,varn>` where $\texttt{line}_1$ depends on $\texttt{line}_2$ on variables $\texttt{var}_1$,$\texttt{var}_2$,..., and $\texttt{var}_n$. For our example of Figure 1(a), we get four dependence tuples:

```
Tuple1:<line 11, line 8, a>
Tuple2:<line 12, line 9, b>
Tuple3:<line 13, line 11, x>
Tuple4:<line 13, line 12, y>
```

For `Tuple1`, line 11 and line 8 are in the same task: CU'_1. This dependence within one task is not an edge in the task graph. The same applies for `Tuple2`. For `Tuple3`, line 13 is contained in CU'_3 and line 11 is contained in CU'_1. This means $task_3$ depends on $task_1$ on variable `x`. `Tuple3` is set to be the incoming edge of $task_3$ and the outgoing edge of $task_1$. According to the same rule, `Tuple4` is added to the task graph as an edge from $task_2$ to $task_3$.

After all the tasks and edges of the task graph have been identified, the transformation module inserts a special virtual task: $task_0$. $Task_0$ is the start point and initiates the flow graph.

3.2.2 Transforming Flow Graph

The flow graph transformation is processed using the procedure described in Algorithm 2. It is divided into three steps:

Step 1: Identifying code sections corresponding to each task. For each task (including $task_0$) 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 coarse-grained CU.

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 single or none incoming edge and multiple outgoing edges. If all the successors receive the same data, the Source Code Rewriting module inserts a TBB `broadcast_node`. If the successors receive different data, a TBB `split_node` is inserted. For example of Figure 1(a), the code section shown in Figure 6(a) is inserted in the source code. Because there is no input data in $task_0$ in our example, a `broadcast_node` template is inserted using the type `continue_msg` in TBB. If input data exists, the Identifying Code Sections module gets the corresponding type via AST and the `broadcast_node` or `split_node` template uses that type.

- The current task has single incoming edge and single or none outgoing edge. The Source Code Rewriting module directly transforms it to flow graph `function_node` using a lambda function. For example of Figure 1(a), $task_1$ is in this case. The output variable name `x` can be gathered from the outgoing edge. Then the Identifying Code Sections module gets the corresponding type `double`. The incoming edge is from the `broadcast_node`, therefore, its corresponding type is `continue_msg`. Finally, the `function_node` code shown in Figure 6(b) is inserted in original source code.

- 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 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

Algorithm 2: Flow graph transformations

Input: Task Graph, AST
Output: transformed code using TBB flow graph

```
1  foreach task_i in task graph do
2      foreach astNode in AST do
3          line = line number of astNode;
4          if line is in task_i then
5              add this line as a string to codeBody of task_i;
6              if line is not the last line of task_i then
7                  remove this line via AST context;
8              else
9                  location = location of this line;
10                 if incoming edge<= 1 and outgoing edges> 1
                   then
11                     if successors receive same data then
12                         insert a broadcast_node;
13                     else
14                         insert a split_node
15                     end
16                 else if incoming edge= 1 and outgoing
                   edge<= 1 then
17                     insert a function_node;
18                 else if incoming edge> 1 and outgoing
                   edge= 1 then
19                     add a join_node in task graph;
20                     choose a buffering policy;
21                     insert this join_node in location;
22                     insert a function_node after join_node;
23                 else
24                     insert a join_node;
25                     insert a function_node;
26                     insert a split_node;
27                 end
28             end
29         end
30     end
31 end
32 foreach edge in task graph do
33     insert make_edge
34 end
```

received at each port.

```
tbb::flow::broadcast_node<tbb::flow::continue_msg> Node0;
```

(a) Code of `broadcast_node`

```
tbb::flow::function_node<tbb::flow::continue_msg,double>
    Node1(g,unlimited,[&](tbb::flow::continue_msg _msg) {
        int a = rand() % 100;
        x = sin(a);
        return x;
});
```

(b) Code of `function_node` for task 1

```
tbb::flow::function_node<tbb::flow::tuple< double,double
    >,double> Node3(g,tbb::flow::serial,[&](tbb::flow::
    tuple<double, double> nodeTuple) {
        z = sqrt(tbb::flow::get<0>(nodeTuple) * tbb::flow
            ::get<0>(nodeTuple)) + sqrt(tbb::flow::get
            <1>(nodeTuple) * tbb::flow::get<1>(nodeTuple)
            );
        return z;
});
```

(c) Code of `function_node` for task 3

```
tbb::flow::join_node<tbb::flow::tuple<double,double>, tbb
    ::flow::queueing> joinNode1(g);
```

(d) Code of `join_node` with queueing policy

```
tbb::flow::join_node<tbb::flow::tuple<type_1,type_2,...,
    type_n>, tbb::flow::reserving> joinNodeID(g);
```

(e) Code of `join_node` with reserving policy

```
tbb::flow::join_node<tbb::flow::tuple<type_1,type_2,...,
    type_n>, tbb::flow::tag_matching> joinNodeID
(g,
 [](const pair<type_1,tag_type> &p) -> size_t
 {return (size_t)p.first; },
 [](const pair<type_2,tag_type> &p) -> size_t
 {return (size_t)p.first; },
 ...
 [](const pair<type_3,tag_type> &p) -> size_t
 {return (size_t)p.first; }
 );
```

(f) Code of `join_node` with tag_matching policy

Figure 6: Source code of TBB flow graph nodes for different cases

The class `join_node` supports three buffering policies:

1. `queueing`: The incoming message is added to an unbounded first-in first-out queue in each input port. The `join_node` greedily consumes all messages as they arrive and generates an output whenever it has at least one item at each input queue.
2. `reserving`: The `join_node` only attempts to generate a tuple when it can successfully reserve an item at each input port. If it cannot successfully reserve all inputs, it releases all of its reservations and will only try again when it receives a message from the port or ports it was previously unable to reserve.
3. `tag_matching`: The `join_node` uses hash tables to buffer messages in its input ports. When it has received messages at each port that have matching keys, it creates an output tuple with these messages.

The buffering policy should be determined by users, because it is a semantics problem that can not be solved by our tool. The transformation tool will report which step needs the `join_node` and its corresponding dependences. The user must specify the buffering policy as parameters for the tool. For our example for $task_3$, we choose the `queueing` buffering policy. For each input variable of $task_3$ `<x,y>`, the Source Code Rewriting module gets the corresponding type information in the AST context: `<double,double>`. Then the code block shown in Figure 6(d) is inserted in the source code. Code blocks related to other two buffering policies are shown in Figure 6(e) and Figure 6(f). At the same time, the `join_node` is inserted in the task graph, it becomes the predecessor of the current task($task_3$). Next, the `function_node` is inserted. In order to get the output tuple from the `join_node`, the `function_node` must replace input variables `x, y` with `get<0>, get<1>` in $CU.codeBody$. The `function_node` shown in Figure6(c) is inserted in the source code.

For the task that has multiple incoming and outgoing edges, we first add a `join_node`, then a `function_node` and finally a `split_node`.

```
#include "tbb/flow_graph.h"
#include <cmath>
#include <cstdlib>
using namespace std;

int main()
{
    tbb::flow::graph g;
    double x,y,z;
    tbb::flow::broadcast_node<tbb::flow::continue_msg>
        Node0(g);

    tbb::flow::function_node<tbb::flow::continue_msg,
        double> Node1(g,tbb::flow::unlimited,[&](tbb::
        flow::continue_msg _msg) {
        int a = rand() % 100;
        x = sin(a);
        return x;
        });

    tbb::flow::function_node<tbb::flow::continue_msg,
        double> Node2(g,tbb::flow::unlimited,[&](tbb::
        flow::continue_msg _msg) {
        int b = rand() % 100;
        y = cos(b);
        return y;
        });

    tbb::flow::join_node<tbb::flow::tuple<double,double>,
        tbb::flow::queueing> joinNode1(g);

    tbb::flow::function_node<tbb::flow::tuple< double,
        double >,double> Node3(g,tbb::flow::serial,[&](
        tbb::flow::tuple<double, double> nodeTuple) {
        z = sqrt(tbb::flow::get<0>(nodeTuple) * tbb::flow
            ::get<0>(nodeTuple)) + sqrt(tbb::flow::get
            <1>(nodeTuple) * tbb::flow::get<1>(nodeTuple)
            );
        return z;
        });

    tbb::flow::make_edge(Node0,Node1);
    tbb::flow::make_edge(Node0,Node2);
    tbb::flow::make_edge(Node1,get<0>(joinNode1.
        input_ports()));
    tbb::flow::make_edge(Node2,get<1>(joinNode1.
        input_ports()));
    tbb::flow::make_edge(joinNode1,Node3);

    Node0.try_put(tbb::flow::continue_msg());
    g.wait_for_all();

    return 0;
}
```

Figure 7: The transformed flow graph source code

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. In our example, there is one edge from $task_0$ to $task_1$ and

one edge from $task_0$ to $task_2$. The following code is inserted:

```
tbb::flow::make_edge (Node0,Node1);
tbb::flow::make_edge (Node0,Node2);
```

The edges from `function_node` to `join_node` are defined in different forms. There are two separate egdes from $task_1$ and $task_2$ to the `join_node` of $task_3$. For each input variable `<x,y>` in `join_node`, we add an edge:

```
tbb::flow::make_edge (Node1,tbb::flow::get
    <0>(joinNode1.input_ports()));}
tbb::flow::make_edge (Node2,tbb::flow::get
    <1>(joinNode1.input_ports()));}
```

When all the nodes and edges have been defined, the transformation terminates and the parallel code is completed. The transformed flow graph code of the example in Figure 1(a) is shown in Figure 7.

4. EVALUATION

To evaluate our approach, we conduct a range of experiments transforming programs from NAS Parallel Benchmarks (NPB) [5] 3.3, three applications from PARSEC [6] 3.0 and two applications from Intel Concurrent Collections (CnC) [4]: Mandelbrot and FaceDetection. For all of the test cases, both a sequential version and an open source equivalent parallel version are available. This allows us to compare the performance of the parallel code generated by our tool against open source parallel implementations from expert programmers.

All the results are obtained on a server with 2x8-core Intel Xeon E5-2650 2 GHz processors and 32GB memory, running Ubuntu 12.04 (64-bit server edition). Both sequential and transformed parallel code were compiled using GCC 4.8.1 at optimization level `-O3`. TBB 4.3 was used during the transformation. Performance results are average values of ten individual runs.

Three programs from NAS Benchmarks, FT, MU and LU are excluded from our experiments because their transformed parallel code can not be compiled using compiling flag `g++ -std=c++11`. A brief overview of programs is given in Table 1. Each program has been executed using two different input data sets. However, in order to compensate for the input sensitivity of dynamic analysis, we tried more inputs whenever possible.

DOALL loop transformation and flow graph transformation are performed in different level, therefore, they are able to coexist and nest with each other, which means a node of a flow graph may contain multiple `parallel_for` loops. Users can configure different transformations through command line. The default configuration is to apply both transformations. In our evaluation, we separate the two transformations to better clarify their own speedups.

4.1 DOALL Loop Transformation

DiscoPoP gives a ranked parallelizable loop list. We targeted the top 30% of the ranked loop list. Table 2 shows the number of loops our approach transformed and their coverage of the original sequential execution time.

Figure 8 shows the speedups of the evaluated programs generated based on our for-loop automatic transformation. They are evaluated with 16 threads.

Table 1: Applications and data sets

Program	Suite	Data Sets
BT	NPB3.3-OMP-C	A,B
CG	NPB3.3-OMP-C	A,B
EP	NPB3.3-OMP-C	A,B
IS	NPB3.3-OMP-C	A,B
SP	NPB3.3-OMP-C	A,B
Mandelbrot	Intel CnC samples	(2,000, 2000, 10,000)
FaceDetection	Intel CnC samples	20,000 images
Blackscholes	PARSEC 3.0	65,536 options
Fluidanimate	PARSEC 3.0	5 frames 300,000 particles
Canneal	PARSEC 3.0	400,000 elements 128 temperature steps

Table 2: Number of transformed loops and their respective coverage of sequential execution time

	Auto	Manual
Program	#loops(%cov)	#loops(%cov)
BT	34 (83.3%)	30 (99.9%)
CG	9 (79.9%)	16 (93.1%)
EP	8 (99.9%)	1 (99.9%)
IS	8 (33.5%)	3 (61.8%)
SP	69 (72.3%)	34 (61.8%)
Blackscholes	3 (99.9%)	1 (99.9%)
Mandelbrot	2 (99.9%)	2 (99.9%)

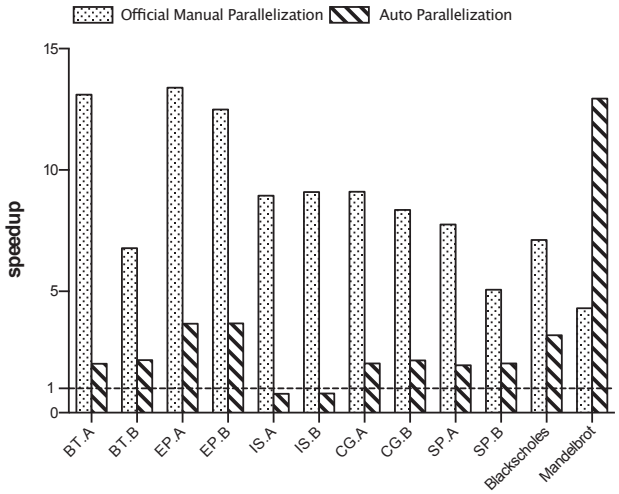


Figure 8: Speedups of official parallelized code and automatically parallelized code

For NAS benchmarks, our automatic parallelizatioin approach achieved an average speedup of 2.2 across the applications and data sizes. The average speedup of open source manually parallelized OpenMP programs is 9.41. The reason that the performance of our generated parallel code is lower than the official manually parallelized code can be attributed to three important factors: Firstly, the manually parallelized versions include not only parallelized loops but also other kinds of parallelization such as task parallelism

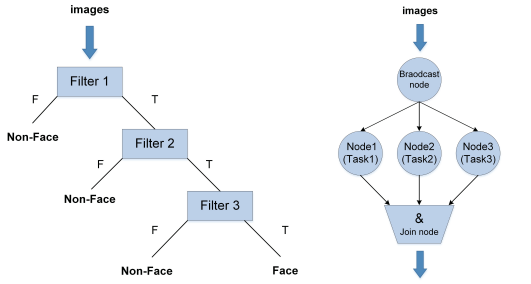
(`#pragma omp parallel sections`). In contrast, our approach only focused on parallelizing loops in NAS benchmarks. Secondly, our approach may parallelize each level of nested loops, which may bring additional overhead. Some NPB applications do not parallelize nested loops. Finally, our approach covers code sections which take less than 80% of the original sequential execution time while the official parallel versions usually cover code sections which take more than 90% of the original sequential execution time.

For Blackscholes, the speedups of auto-transformed code and the official parallel version are 3.19 and 7.12, respectively. The official parallel version manually divides the data sets according to the number of threads while the auto-transformed code does not. However, dividing data sets is heavily semantic related and doing it automatically is far beyond the state of the art.

For Mandelbrot, the speedup of auto-transformed version is 12.95. It outperforms the manually parallelized version, whose speedup is 4.30. The auto-transformed code performs better because the 2D perfectly nested loop which takes 99.9% of the sequential execution time is transformed into a recursively divisible two-dimensional iteration space template `tbb::blocked_range2d`, which has very low overhead comparing to the CnC parallelization method implemented in the manual version.

4.2 Flow Grah Transformation

To evaluate the flow graph transformation, we transformed one Intel CnC sample application: `FaceDetection` and two applications in PARSEC benchmark suite: `Fluidanimate` and `Canneal`. We transformed subgraphs of the task graphs which have been mapped to Program Execution Trees (PET). Flow graph is equivalent to task graph with additional `join_nodes`.



(a) Logic of FaceDetection (b) Flow graph

Figure 9: FaceDetection logic and flow graph

FaceDetection is an abstraction of a cascade face detector used in the computer vision community. The face detector consists of different filters. As shown in Figure 9(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. Our approach detected each filter as a task. When images stream into the face detector, each filter can be executed in parallel and returns a bool value of the classified result. Then a join node is inserted to buffer all the bool values. In order to decide whether an image is a face, every bool value corresponding to that specific image is needed. Thus we configure the transformation tool to use `tag_matching` buffering policy in the join node. After performing logical conjunction of these bool values, weather the image is a face can be decided. The flow graph of FaceDetection is shown in Figure 9(b).

There are three filters in the application, which take 99.9% of sequential execution time. We use 20,000 images as input. As shown in Figure 10, the speedup of our transformed flow graph parallel version is 9.92 using 32 threads. There is a large data set streaming into the flow graph and the object of graph is created once but executed 20,000 times. The overhead of creating and initializing flow graph objects is relatively low. This transformation of flow graph is semi-automatic and users only need to specify buffering policy in `join_node`.

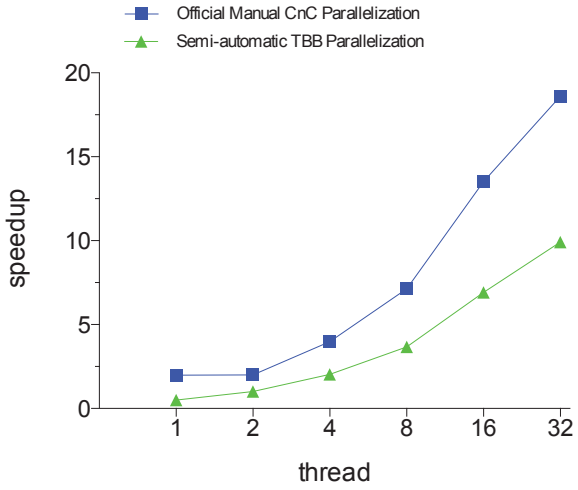


Figure 10: FaceDetection speedups with different threads

In order to illustrate the scalability of the transformed code, we compare the speedups achieved by official CnC parallel version and our transformed TBB flow graph version using different number of threads. The result is shown in Figure 10. 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 has been heavily optimized and restructed. For example, some data structures have been altered from *vector* to CnC *item_collection*. Besides, it uses not only task parallelism but also data parallelism. These changes are obviously beyond the capability of an auto-parallelization approach. In fact, as shown in Figure 10, when using just one thread, the speedup of official CnC parallel version is already 2.00, because of the optimizations.

Note that a parallel code with flow graph can result in high speedup only when the program actually has the flow graph pattern. If the flow graph transformation is applied for the program without a heavily executed flow graph, the speedup is relatively low. To demonstrate this problem, we evaluated two applications in the PARSEC benchmark suite.

Fluidanimate uses an extension of the Smoothed Particle Hydrodynamic (SPH) method to simulate an incompressible fluid for interactive animation purposes. Task parallelism with flow graph have been detected in two functions of fluidanimate: `RebuildGride` and `ProcessCollisions`.

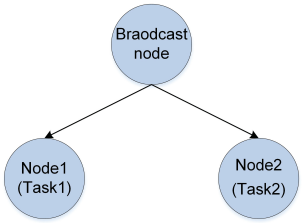
In `RebuildGrid`, a code section performing the Courant-

```
bool cfl_cond_satisfied=
     false;
for(int di = -1; di <= 1;
     ++di) {
    ...
    ...
    if(!cfl_cond_satisfied)
         {
        ...
        exit(1);
    }
}
```

(a) Task 1

```
int index = (ck*ny + cj)*nx
     + ci;
Cell *cell = last_cells[
     index];
...
...
cell->v[np %
     PARTICLES_PER_CELL].z
     = cell2->v[j %
     PARTICLES_PER_CELL].z;
```

(b) Task 2



(c) flow graph

Figure 11: Tasks and their code sections in `RebuildGrid` of fluidanimate

Friedrichs-Lewy (CFL) condition check can be executed in parallel with the remaining part of the function. Figure 11 shows the tasks detected by our approach and their corresponding code sections. Task 1 performs the CFL condition check. The flow graph is shown in Figure 11(c), which is defined within a deeply nested loop. At each iteration, the object of the flow graph is firstly created and then executed, therefore, the overhead is very high. The local speedup of the auto-transformed code is 1.63.

Another flow graph is identified in the function `ProcessCollisions`. It contains six loop nests checking whether a particle hits any of the six surfaces of the 3D cube space. DiscoPoP finds that these six loop nests do not have any data or control dependence with each other, so that they can run in parallel. Each loop nests is identified as a task in our approach. Figure 12 shows the six tasks and their corresponding code sections. Figure 12(g) shows the flow graph. Whenever `ProcessCollisions` is called, the objects of the flow graph are created and initialized, but they are executed only once. Such overhead is also very high. The local speedup of the auto-transformed code is 1.81.

Canneal is a kernel that uses cache-aware simulated annealing (SA) to minimize the routing cost of chip design. A flow graph is detected in function `netlist_elem::routing_coast_given_loc`. As shown in Figure 13, there are three tasks: $task_1$ and $task_2$ calculate *fanin_cost* and *fanout_cost* respectively. $Task_3$ adds those two values and writes the final result to *total_cost*. Figure 13(d) shows the corresponding flow graph. The situation is the same with `ProcessCollisions` in **Fluidanimate**. After creating and initializing the object, the flow graph is executed only once, which brings overhead. The speedup is 1.32.

5. RELATED WORK

Static automatic parallelization methods have been successfully applied to achieve loop parallelism [11, 21]. Unfortunately, many parallelization opportunities are missed due to the lack of runtime information. Speculative parallelization techniques [25, 8] exploit parallelism in a way that instructions can be scheduled at a time when it has not been determined that the instructions will need to be executed. However, to perform these approaches, hardware support is

```
x=0; // along the domainMin
     .x wall
for(y=0; y<ny; ++y)
{...}
```

(a) Task 1

```
x=nx-1; // along the
     domainMax.x wall
for(y=0; y<ny; ++y)
{...}
```

(b) Task 2

```
y=0; // along the domainMin
     .y wall
for(x=0; x<nx; ++x)
{...}
```

(c) Task 3

```
y=ny-1; // along the
     domainMax.y wall
for(x=0; x<nx; ++x)
{...}
```

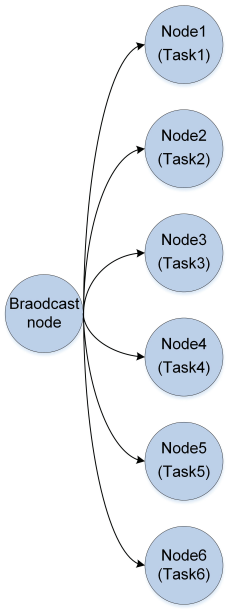
(d) Task 4

```
z=0; // along the domainMin
     .z wall
for(x=0; x<nx; ++x)
{...}
```

(e) Task 5

```
z=nz-1; // along the
     domainMax.z wall
for(x=0; x<nx; ++x)
{...}
```

(f) Task 6



(g) flow graph

Figure 12: Tasks and their code sections in `ProcessCollisions` of fluidanimate

```
routing_cost_t fanin_cost =
     0;
for (int i = 0; i< fanin.
     size(); ++i){
    location_t* fanin_loc =
         fanin[i]->
         present_loc.Get();
    fanin_cost += fabs(loc.
         x - fanin_loc->x);
    fanin_cost += fabs(loc.
         y - fanin_loc->y);
}
```

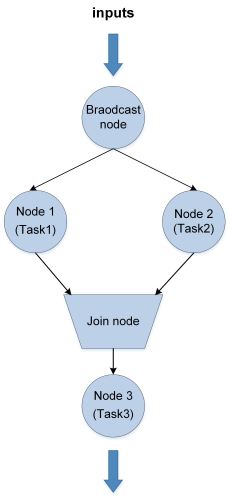
(a) Task 1

```
routing_cost_t fanout_cost
     = 0;
for (int i = 0; i< fanout.
     size(); ++i){
    location_t* fanout_loc
         = fanout[i]->
         present_loc.Get();
    fanout_cost += fabs(loc
         .x - fanout_loc->x
         );
    fanout_cost += fabs(loc
         .y - fanout_loc->y
         );
}
```

(b) Task 2

```
routing_cost_t total_cost =
     fanin_cost +
     fanout_cost;
```

(c) Task 3



(d) flow graph

Figure 13: Tasks and their code sections in `netlist_elem::routing_cost_given_loc` of canneal

usually needed.

Ottoni et al. [20] propose an automatic approach called Decoupled Software Pipelining (DSWP). DSWP exploits the fine-grained pipeline parallelism and generates the parallel code. It requires fine-grain communication primitives to communicate register values. Li et al. [15] automatically parallelize imperative C programs at thread level, but their

approach is limited to producer-consumer relations from array and pointer code.

The approach presented by Tournavitis et al. [24] uses both static analysis and dynamic profiling to detect potential parallelism. A machine-learning based prediction mechanism is used to map the parallelism to different architectures. They generate parallel code using OpenMP annotations and target on parallel for-loops. However, the code transformation is relatively simple. They do not perform high-level code restructuring, which can exploit coarse-grained task parallelism. In their recent work [23], they exploit pipeline parallelism.

OpenRefactory/C [10] is a tool providing many refactoring methods in C programs, but it does not automatically transform sequential code to parallel code. The approach presented in [22] transforms serial C++ code to parallel code using OpenMP directives. However, it requires users to define semantics of high-level abstractions in advance.

6. CONCLUSION AND FUTURE WORK

We proposed a novel auto-parallelization method integrating data-dependence profiling, task parallelism extraction and source-to-source transformation. We have made the following contributions:

- Based on the program analysis tool DiscoPoP, our parallelization approach automatically generates parallel code without requiring users to annotate parallel code sections in advance.
- We extract coarse-grained task parallelism from CU graph mapped to Program Execution Tree (PET).
- To the best of our knowledge, our approach is the first tool to transform sequential C/C++ source code to TBB parallel code, which exploits both loop parallelism and task parallelism without special compiler support.

Future works will focus on finding more parallel patterns in the task graph, since flow graph is general enough to be adapted to support other patterns such as pipeline. We also intend to use other techniques like automatic generation of unit testing to validate the correctness of the transformed code.

7. ACKNOWLEDGMENT

This work was partially supported by National Natural Science Foundation of China under the grant No.91330117 and National High-Tech Research and Development Plan of China under the grant No.2012AA01A306.

8. REFERENCES

[1] `http://www.open64.net`.

[2] `https://www.threadingbuildingblocks.org`.

[3] `http://clang.llvm.org`.

[4] `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc`.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81. ACM, 2008.

[7] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175. ACM, 1986.

[8] J. Dou and M. Cintra. Compiler estimation of load imbalance overhead in speculative parallelization. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 203–214. IEEE Computer Society, 2004.

[9] W. Ghardallou. Using invariant relations in the termination analysis of while loops. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1519–1522. IEEE Press, 2012.

[10] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. Openrefactory/c: An infrastructure for building correct and complex c transformations. In *Proceedings of the 2013 ACM Workshop on Refactoring Tools*, WRT '13, pages 1–4. ACM, 2013.

[11] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, Dec. 1996.

[12] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[13] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–. IEEE Computer Society, 2004.

[14] C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In *Proceedings of the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, PAS '95, pages 10–. IEEE Computer Society, 1995.

[15] F. Li, A. Pop, and A. Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro*, 32(4):19–31, July 2012.

[16] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *Proc. of the 42nd International Conference on Parallel Processing Workshops (ICPPW),Lyon, France*, pages 1004–1013, Oct. 2013.

[17] Z. Li, A. Jannesari, and F. Wolf. An efficient data-dependence profiler for sequential and parallel programs. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS '15, Hyderabad, India, 2015.

[18] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Comput.*, 24(3-4):445–475, May 1998.

[19] A. Mili, S. Aharon, and C. Nadkarni. Mathematics for reasoning about loop functions. *Sci. Comput. Program.*, 74(11-12):989–1020, Nov. 2009.

[20] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118. IEEE Computer Society, 2005.

[21] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for mpps. Technical report, In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign, 1993.

[22] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. A c++ infrastructure for automatic introduction and translation of openmp directives. In *OpenMP Shared Memory Parallel Programming*, pages 13–25. Springer, 2003.

[23] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388. ACM, 2010.

[24] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187. ACM, 2009.

[25] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 238–249. IEEE Computer Society, 1998.