The Basic Building Blocks of Parallel Tasks

Rohit Atre
Aachen Institute for Advanced Study in Computational Engineering Science, Germany
RWTH Aachen University, Germany
r.atre@grs-sim.de

Ali Jannesari
German Research School for Simulation Sciences, Germany
RWTH Aachen University, Germany
a.jannesari@grs-sim.de

Felix Wolf
German Research School for Simulation Sciences, Germany
RWTH Aachen University, Germany
f.wolf@grs-sim.de

ABSTRACT

Discovery of parallelization opportunities in sequential programs can greatly reduce the time and effort required to parallelize any application. Identification and analysis of code that contains little to no internal parallelism can also help expose potential parallelism. This paper provides a technique to identify a block of code called Computational Unit (CU) that performs a unit of work in a program. A CU can assist in discovering the potential parallelism in a sequential program by acting as a basic building block for tasks. CUs are used along with dynamic analysis information to identify the tasks that contain tightly coupled code within them. This process in turn reveals the tasks that are weakly dependent or independent. The independent tasks can be run in parallel and the dependent tasks can be analyzed to check if the dependences can be resolved. To evaluate our technique, different benchmark applications are parallelized using our identified tasks and the speedups are reported. In addition, existing parallel implementations of the applications are compared with the identified tasks for the respective applications.

1. INTRODUCTION

As a result of the stagnating single core performance, multicore processors with several cores on a single chip have become widely popular. This trend will continue to speed up in the coming years and will require software developers to focus more on parallel programming, especially at the thread level. Hence, discovering potential parallelization targets in sequential programs can be very helpful. It would be a very likely scenario in a major organization that the developer who is tasked with parallelizing an application had not developed the original sequential version. In such cases, the discovery of available parallelization opportunities can greatly reduce the time and effort required to parallelize the application. The use of these types of analyses can not only save time, but also help improve the performance of the application and ease the developer into the process of parallelizing the application.

As programmers try to solve more complex problems and use more sophisticated algorithms to do it, the program code written to implement these algorithms becomes complex as well. This type of code may be better parallelized by using techniques other than data-parallelism. Although this complexity does not necessarily prevent data-parallelism, the identification of tasks and use of task-parallel constructs could prove to be more efficient in certain cases [5]. Focusing on loops for discovering potential parallelism in sequential programs has already been covered in the previous works [12] [13] [14] . In this paper, our focus is on identifying tasks that can run in parallel.

There are two main categories of tasks that can be identified for parallelism. A task in data-parallelism is similar to OpenMP *task* construct which runs the same code on different threads. The tasks involved in task parallelism are similar to the OpenMP *sections* construct. They can be two completely different code sections that perform different operations with no clear relation between them and run in parallel.

Our analysis defines a concept called Computational Unit (CU) [13] [14]. A CU can be used as a building block for various purposes like forming parallel tasks, creating stages of a pipeline, or forming the nodes of a flow graph. Every CU follows a read-compute-write pattern. It means that a program state is first read from the memory, a new state is computed, and finally the new state is written back. This characteristic makes CU a logical unit for forming larger tasks. A CU is a code-granularity level which is independent of the language and can be used for both program analysis and expressing the parallelism in programs. It is the smallest unit of code that can be assigned to a thread while running it in parallel with other CUs or with tasks formed by merging these CUs. A CU itself has little to no further (internal) parallelism.

This paper proposes a static technique to identify CUs. The identification of CUs takes place at different levels of the program, taking the various program regions into account. The next step is to merge the CUs to form tasks that can be run in parallel with other tasks. The computation performed by these tasks can be scattered throughout the program. As a result, we can find the different types of tasks mentioned above.

Dynamic data dependences are used to check the available parallelism in sequential programs. Based on the absence

COSMIC '15, February 08 2015, San Francisco Bay Area, CA, USA
Copyright 2015 ACM 978-1-4503-3316-0/15/02 ...$15.00
http://dx.doi.org/10.1145/2723772.2723778

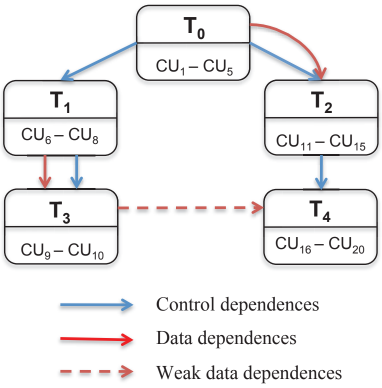


Figure 1: A task graph with control and data dependences.

of dependences between these tasks, it can be concluded whether the tasks can be run in parallel with other tasks. Dependences can also be used to detect various patterns for parallelism. For instance, the CUs can be merged to form the various stages of a pipeline. The number of dependences between the tasks also provides an indication of the amount of effort required to run the tasks in parallel. The analysis for CU identification and task formation takes place at compile time and keeps a low time and memory overhead in order to deal with the sequential programs of realistic size. To account for the input sensitivity of the dynamic analysis, a representative input is chosen to cover most of the execution pathways and the code is also run multiple times.

Identifying parts of the code that are tightly bound together and have very little internal parallelism in turn reveals the code sections that require little effort to be run in parallel with other similar code sections. Figure 1 shows tasks T_0 - T_4 which are code sections formed by putting CUs together and have high number of control and data dependences within them. They have very little internal parallelism. Grouping the sequential code in this way reveals the parts of the code that are not dependent on each other or are very weakly dependent. In Figure 1, it can be observed that the pair of tasks T_0, T_2 or the pair T_0, T_1 is tightly coupled because of the presence of control and data dependences that prevent parallelism. However, tasks T_1 and T_2 have no control or data dependence between them and hence they can be run in parallel. Similarly if the data dependences between tasks T_3 and T_4 are weak and can be resolved then the two tasks can be run in parallel with each other.

In essence, our work provides a technique to discover potential parallelism in sequential programs by identifying tasks that can run in parallel with other tasks and have little to no internal parallelism. This is accomplished by

1. Defining CUs that perform a unit of work in a program and act as a building block for parallel tasks.

2. Expressing the sequential program with a CU graph using CUs and data dependences.

3. Partitioning the CU graph to produce meaningful tasks or other parallel patterns.

4. Evaluating the tasks to determine whether they can run in parallel with other tasks.

To evaluate our results we have analyzed applications from the STARBENCH Parallel Benchmark Suite [2] and the PARSEC Benchmark Suite [3] and generated tasks for the applications. Based on the program regions analyzed, we narrowed our results to the most important tasks taking the total execution time of the code regions into account. For tasks that can run different code on different threads, we have parallelized some of the applications by assigning these tasks to different threads and reported the speedups based on suggested parallelization opportunities. For tasks that run the same code on multiple threads, we have compared the existing parallel version of the applications with the identified tasks to verify their validity.

The rest of this paper is organized as follows. Section 2 discusses the current state of the art and related work. Section 3 provides a detailed explanation of the aforementioned technique that identifies the CUs and uses them to form tasks. Section 4 provides analysis and evaluation of this technique using applications of the STARBENCH and the PARSEC benchmark. Finally, Section 5 provides a conclusion and a summary of our approach and discusses our future work.

2. RELATED WORK

Several attempts have been made to identify coarse-grained parallelism in sequential programs by partitioning the code. Sura, O'Brien, and Brunheroto [19] identify *fibers* as a sequence of instructions without any control flow or memory carried dependences. They partition the code into fibers and also build a code graph. However, the code sections generated by their approach target very fine grained parallelism using dedicated hardware queues for low latency transfer of values. Lauderdale and Khan [11] have attempted to identify fine-grained units of work called *Codelets* that can be described by the application for parallelism. They propose a codelet runtime that represents these units of work as small descriptor objects. These objects reference run/cancel fork functions and include a description to store the codelet's state information. Their work however did not describe how the identification of these codelets takes place. They also propose a C-based language SCALE because generating codelets manually and ensuring they interact correctly would be a daunting task.

MAPS [4] is an integrated framework that mainly concentrates on MPSoC application parallelization for embedded systems. It only focuses on parallelization for sequential C programs since a majority of the MPSoC software has been written in C. It identifies code sections called *Coupled Blocks*. These code blocks are identified with the constraints that they should be schedulable and should be tightly coupled by data dependences. The task suggestions from MAPS are contiguous blocks of code. Our approach identifies tasks based on the computation being carried out and the tasks can be scattered across the program or a code construct. Ottoni et al [16] propose a *Decoupled Software Pipelining*. They analyze the dependence graph and merge the strongly connected components to generate a directed acyclic graph (DAG) out of a loop. Their approach targets loops and exploits pipeline parallelism in sequential applications. Li et al [13] [14] use dynamic analysis to form CUs by monitoring the memory accesses and identifying read-after-write patterns. Since, a simple assignment operation could correspond to such a pattern, the CUs produced are very fine grained.

Other works that try to assess the extent of potential parallelism in sequential programs primarily perform dynamic

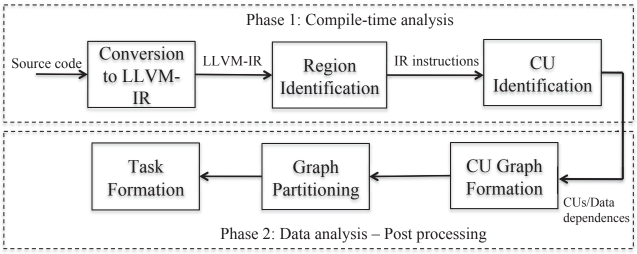


Figure 2: Overview of the technique for CU identification and task formation.

dependence analysis or use runtime scheduling frameworks or utilize a combination of both. Kremlin [6] computes a metric called self-parallelism which is calculated by determining the length of the critical path of a given program region using the dependence information. This metric lets Kremlin quantify parallelism for every region analyzed. Alchemist [21] tries to identify predefined constructs that can be treated as candidates for asynchronous execution. It estimates the effectiveness of parallelizing a certain construct by profiling the dependence distance using Valgrind [15]. Parwiz [9] records data dependencies and attaches them to the nodes of the execution tree it maintains. These tools try to measure the amount of parallelism available between two predefined points in the code sections. While these tools might identify tasks for data parallelism, they might not detect the tasks that do not confine to the constructs of the source code and are useful for task parallelism.

3. APPROACH

This section introduces the concept of regions and CUs with the help of examples. Then we describe our approach that identifies CUs and uses them to form tasks that can be parallelized at different levels of the program. An overview of our approach can be seen in Figure 2. There are two main phases of the analysis. The first phase comprises the steps performed during the compilation of the source code. The second phase processes the data gathered during the first phase and identifies the tasks for parallelism.

Our analysis is based on LLVM [10]. LLVM is a collection of modular and reusable compiler and toolchain technologies. In the first step of our analysis, we convert the source code of the program into the intermediate representation specified by LLVM (LLVM-IR) [1]. Based on this IR, we analyze the application to recognize the various program regions. The regions are helpful in defining the boundaries for the formation of CUs. The next step is to perform an analysis within every region to classify and group instructions together. The grouping of instructions is based on the computation they carry out. This step is followed by identifying the CUs using the information gathered from the instructions. The next step is to use the dynamic data dependences and CUs together to create a CU graph. The partitioning of the graph provides us with a list of tasks. The tasks that belong to functions or code regions with a large percentage of the total execution time of the program are analyzed as potential candidates for parallelism.

3.1 Phase 1 : Compile time analysis

The first phase of the analysis takes place during the compilation of the program. The source code is converted into LLVM-IR and the instructions of each region are analyzed to identify CUs. It is important to note that the instructions in the context of our analysis refer to the instructions available through LLVM-IR and not machine-level instructions.

3.1.1 Region identification

A *Region* is a connected sub-graph of a control-flow graph that has exactly two connections to the remaining graph [7]. Every region has a single entry and a single exit. A region *contains* another region if the nodes of the other region are a subset of the nodes of the first region. The analysis first identifies all the regions of the input program based on the information available at the compile time.

Every function in the program is considered a top-level region for the analysis. A region can be contained within another region and is considered as a sub-region of the enclosing region. Listing 1 shows a code region. The function *netlist::get_random_pair* is the top-level region. A region may contain further sub-regions in the form of loops or other control structures. The regions and sub-regions are important to decide the granularity or the program level at which the identification of parallel tasks would be beneficial to the programmer.

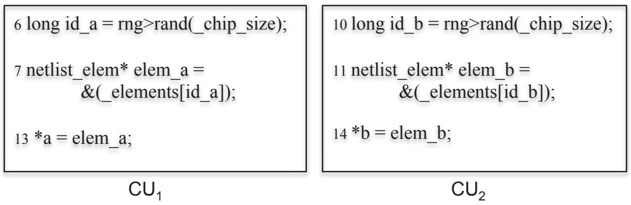
3.1.2 CU identification

The concept of CUs is central to our method of identifying the tasks for parallelism in sequential programs. *A CU is defined as a set of instructions that form a read-compute-write pattern.* A CU differs from the basic block such that a basic block contains operations that are consecutive and has only one entry and one exit point. A CU however, is a grouping of instructions that are not necessarily consecutive but perform a computation and is based on the use of a set of variables together. A single CU or a group of CUs merged together can provide the code sections that perform a task. The grouping takes place considering the amount of computation that CUs share and the data dependences between them. Hence a group of CUs have little to no internal parallelism. These code sections can be examined to see if they can be run concurrently with other code sections or themselves to exploit the available parallelism.

Listing 1: Function *netlist::get_random_pair* to demonstrate two CUs

```
1
2   //Region 0; Depth 0
3   void netlist::get_random_pair(netlist_elem** ↩
        a, netlist_elem** b, Rng* rng)
4   {
5       //get a random element
6       long id_a = rng->rand(_chip_size);
7       netlist_elem* elem_a = &(_elements[id_a]);
8
9       //now do the same for b
10      long id_b = rng->rand(_chip_size);
11      netlist_elem* elem_b = &(_elements[id_b]);
12
13      *a = elem_a;
14      *b = elem_b;
15      return;
16  }
```

To better understand what a CU is, consider the example in Listing 1. The source lines 6 and 10 perform the initialization of variables *id_a* and *id_b* with a random value. Lines 7 and 11 perform the task of calculating *elem_a* and *elem_b* by using the variables *id_a* and *id_b* respectively. These two


```
6 long id_a = rng>rand(_chip_size);

7 netlist_elem* elem_a =
        &(_elements[id_a]);

13 *a = elem_a;
```

```
10 long id_b = rng>rand(_chip_size);

11 netlist_elem* elem_b =
        &(_elements[id_b]);

14 *b = elem_b;
```

Figure 3: A code section separated into two CUs.

operations are performed independently of one another. Finally, the lines 13 and 14 are responsible for writing the final computation back to **a* and **b*. In essence, the set of LLVM-IR instructions corresponding to the lines {6, 7, 13} perform one computation and the ones corresponding to the lines {10, 11, 14} perform another computation and these computations are independent of each other.

The two computations mentioned above follow a basic rule where a variable or a group of variables are read and then they are used to perform another calculation. This is followed by the final state being written to another variable as a store operation. Hence, these two computations can be said to follow a *read-compute-write* pattern. The two computations can be visualized as seen in Figure 3. The final *store* instruction that writes a value to **a* uses all the instructions that correspond to the lines {6, 7, 13} to perform that write operation. Similarly, the group of instructions that correspond to the lines {10, 11, 14} are used for the final *store* instruction that writes **b*. These two sets of instructions can individually be defined as CUs. These CUs form the building blocks of the tasks which can be created for exploiting parallelism in the sequential programs.

The highest level at which the analysis for the identification of CUs is performed is a function. Identifying parallel tasks at a function level or within a function would be a logical choice since a function in itself is normally designed to perform a specific task.

To identify CUs within a function, every region within the function is identified first. Every function is considered a top level region and the information necessary to identify the regions within it is gathered. Once every region inside a function is identified, the next step is to collect all the instructions that belong to every region and proceed with the analysis of the instructions.

3.1.3 CU initialization

In order to create a list of CUs for every region, every *store* instruction within this region is firstly identified. A *store* instruction that initializes a variable with a constant value and does not use any other variables is marked as an initialization and is not considered a part of any CU. For every other *store* instruction within the region, a CU is created. For every *store* instruction that redefines a variable, a new CU is created as well.

Listing 2 shows some of the instructions that belong to $CU_1 = \{6, 7, 13\}$ of Figure 3 and are responsible for computation of **a* in the form of LLVM-IR. The *store* instruction in line 15 is responsible for the final write operation. The next step is to identify the remaining instructions of the CU that are used to perform this final write operation. To identify and add the remaining instructions that belong to a CU, an analysis is performed on the instructions. LLVM provides a variety of inspection and traversal routines and defines classes that can be very useful for this purpose. This also includes a use-def chain where all the instances of *Value* class used by an instance of *User* class can be iterated over. Instances of class *Instruction* are common *User*s. This allows us to iterate over all the operands that any particular instruction uses. The operands for the instructions in LLVM-IR are either the variables defined in the program or they are other instructions of LLVM-IR. Consider the *store* instruction at line 7 of Listing 2. The operands for this instruction are *%call* and *%id_a*. The operand *id_a* is a variable and looking at the IR it can be observed that the operand *%call* is another instruction which is defined at line 6. *%call* is used by the *store* instruction at line 7 and eventually at line 15. Hence, the rest of the instructions that belong to a CU can be identified by iterating recursively over the operands which are instructions themselves. This also lets us keep track of all the variables that are used to perform the final write operation.

Listing 2: LLVM-IR instructions for calculation of **a* from Listing 1

```
1
2   store %class.netlist_elem** %a, ↩
        %class.netlist_elem*** %a.addr, align 8
3   %this1 = load %class.netlist** %this.addr
4   %_chip_size = getelementptr inbounds ↩
        %class.netlist* %this1, i32 0, i32 3
5   %1 = load i32* %_chip_size, align 4
6   %call = call i64 @_ZN3Rng4randEi(%class.Rng* ↩
        %0, i32 %1)
7   store i64 %call, i64* %id_a, align 8
8   %_elements = getelementptr inbounds ↩
        %class.netlist* %this1, i32 0, i32 4
9   %2 = load i64* %id_a, align 8
10  %call2 = call %class.netlist_elem* ↩
        @_ZNSt6vectorI12netlist_elemSaIS0_EEixEm
11       (%"class.std::vector"* %_elements, i64 %2)
12  store %class.netlist_elem* %call2, ↩
        %class.netlist_elem** %elem_a, align 8
13  %11 = load %class.netlist_elem** %elem_a, ↩
        align 8
14  %12 = load %class.netlist_elem*** %a.addr, ↩
        align 8
15  store %class.netlist_elem* %11, ↩
        %class.netlist_elem** %12, align 8
```

```
Data: Region, CUSet
Result: List of CUs within a region.
Get all the instructions for the Region;
for every instruction do
    if isa<StoreInst > then
        GetOperandFrom(instruction);
        Get variable name from Operand;
        Create a new CU for this variable;
        Add instruction to CU.ComputeInstructions;
        Insert CU in CUSet for this region;
        AnalyzeInstructionForCU(CU, instruction);
    end
end
Insert CUSet in map <Region >.
```

Algorithm 1: Finding CUs for a region.

The method for finding all the CUs is shown in Algorithms 1 and 2. Algorithm 1 shows how each region is identified first and then every region is associated with a list of CUs identified within the region. All the instructions belonging to a region are collected first. For every *store* instruction, the variable defined is identified. A new CU is created and the *store* instruction is added to the list of computation instructions for it. The next step is to identify other instructions that belong to this CU. This is done to complete the CU and analyze the remaining instructions of the region.

3.1.4 CU completion

Algorithm 2 shows the process of identifying all the instructions of a CU. To identify the remaining instructions that belong to a CU, we create a list of all the variables used by the *defining* variable of a CU. This involves analyzing all the instructions that are used by the final *store* instruction and updating the list of used variables that belong to this CU. Every operand of the instruction is checked to see if it is an instruction in itself. As demonstrated earlier in Listing 2, the operand *%call* in line 7 actually corresponds to an instruction which is defined at line 6. All the variables that are involved in the computation are identified and they are added to the list of used variables for the CU.

Once all the CUs are updated according to the instructions of the region, the set of CUs is associated with the region along with the details like the depth of the region with respect to the top-level region, and the instruction count for the region. A list of CUs with a corresponding ID and the instruction count is created for the entire program to better identify and process the code sections as tasks. All the instructions that belong to the top level region and any region within a region are collected and this analysis is performed on them.

```
Data: CU, Instruction
Result: A complete CU based on the use-def chain of
        the Instruction.
for every operand in the Instruction do
    Get variable name from operand;
    if isa<Instruction >(operand) then
        Add instruction to CU.ComputeInstructions;
        AnalyzeInstructionForCU(CU, instruction);
    end
    Add variable to CU.usedVariables;
end
```

Algorithm 2: Identifying all the instructions of a CU.

3.2 Phase 2: Data analysis - Post processing

To form the tasks using CUs, we perform the following steps:

1. Collect information about the common instructions between every two CUs.
2. CU graph formation:
    - Use the dynamic analysis to identify the dependences and apply them between the CUs.
    - Create a CU graph using common instructions and dependences as the edges with CUs as the vertices.
3. Graph Partitioning: Partition the CU graph where CUs are weakly connected based on the weights of the edges.
4. Task formation: Create a list of all the CU groups that are formed because of the partitioning as identified tasks. Prioritize the tasks that correspond to the functions taking a large percentage of the total execution time of the program.

Listing 3: Code from function *RebuildGrid* of *parsec.fluidanimate* to demonstrate common instructions.

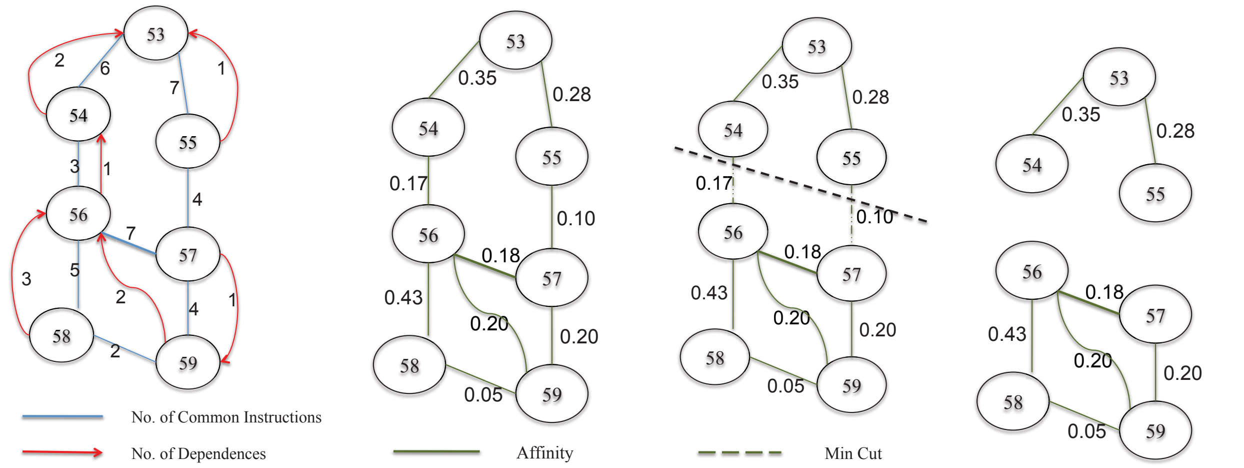
```
1     ...
2     Cell *cell2 = &cells2[i];
3     int np2 = cnumPars2[i];
4     //iterate through source particles
5     for(int j = 0; j < np2; ++j)
6     {
7       //get destination for source particle
8       int ci = (int)((cell2->p[j % ↩
      PARTICLES_PER_CELL].x - domainMin.x) / ↩
      delta.x);
9       int cj = (int)((cell2->p[j % ↩
      PARTICLES_PER_CELL].y - domainMin.y) / ↩
      delta.y);
10      int ck = (int)((cell2->p[j % ↩
      PARTICLES_PER_CELL].z - domainMin.z) / ↩
      delta.z);
11      ...
12    }
```

3.2.1 Use of common instructions

Since regions can exist within other regions, the CU analysis identifies the CUs in such a way that the code sections can overlap with each other. For instance, a CU could be a subset of another CU or two CUs could have instructions (and therefore, lines of code) that are common to each other. This is possible because the same set of instructions or same computation in the program code can be used to eventually define more than any one particular variable. Also, more than one task can share the same code. Understanding which code is shared among tasks can help introduce parallelism by narrowing parts of the code that would require replication, privatization etc. Consider the code snippet from Listing 3. It belongs to the function *RebuildGrid* of the program *Fluidanimate* from the PARSEC benchmark. The CU analysis identifies following CUs for this code region: $CU_1 = \{2,5,8\}$, $CU_2 = \{2,5,9\}$ and $CU_3 = \{2,5,10\}$. It can be observed that the definition of variables *ci*, *cj* and *ck* uses and shares the computation from the lines {2,5} which is the overlap between the three CUs. This overlap exists because all the three variable definitions use the variables *cell2* and *j*. If the overlap between any two CUs is large, it is not logical to place them in different tasks. Hence, for every region, if one CU is a subset of another CU, then only the larger of the two CUs is considered for the analysis. Also, if the two CUs only partly overlap with each other, then the information about the common and the unique instructions between the two CUs is recorded. To store the common and unique instructions between two CUs, the intersection and symmetric difference of the set of instructions of both the CUs is calculated. This information is one of the metrics used for deciding if the two CUs should be put together or separated when tasks are formed.

3.2.2 Using data dependences

The next step is to gather data dependences of the target program. The data dependences are collected by dynamic program analysis using the tool DiscoPoP [13]. The dynamic analysis information obtained from this tool provides us read-after-write (RAW) dependences between the source lines of the program. We only consider RAW dependences



(a) CU graph with CUs as vertices and RAW dependences and common instructions as edges.

(b) CU graph with affinities between the CUs

(c) CU graph with a minimum cut.

(d) CU graph partitioned to identify tasks.

Figure 4: Demonstration of a CU graph and graph partitioning to form tasks.

since write-after-write (WAW) and write-after-read (WAR) dependences can usually be resolved by techniques like privatization. The RAW dependences can be used to identify dependences between the CUs. For every RAW dependence, identifying the source CU and the destination CU provides us the dependences between all the CUs of the program. The number of dependences between any two CUs can be more than one since a CU contains more than one source line. This number is another criterion to determine if the two CUs should be put together to form tasks or if they can be separated. If the number of dependences between any two CUs is high, then the two CUs are strongly dependent and resolving these dependences to separate them would require more effort.

3.2.3 *CU graph formation*

Using the common instructions and the RAW dependences between the CUs, a CU graph is constructed. The nodes of this graph are the CUs identified. The CU graph has two types of edges. The first type of edge between any two CU nodes signifies RAW dependence between them and is a directed edge. The weight of a RAW edge is the number of RAW dependences between the two CUs. The second type of edge signifies that there are common instructions between the two CUs and hence they share computation. This is an undirected edge and its weight is the number of common instructions between the two CUs. Figure 4a shows a CU graph with red edges as RAW dependences between the CUs and blue edges as the CUs connected because of the common instructions between them. A CU graph is generally a disconnected graph with several connected components.

3.2.4 *Graph partitioning and task formation*

In graph theory, a connected component of an undirected graph is a subgraph in which any two vertices are connected to each other, and which is not connected to any additional vertices in the remaining graph. Every connected component of each CU graph is analyzed to identify tasks. As established earlier, a high value of weight on the edges between any two vertices indicates that those two CUs either share large amount of computation or they are strongly dependent on one another. Using these two criteria, we calculate a value called *affinity* for every pair of CU nodes in the graph. The affinity between any two CU nodes hence indicates how tightly coupled the two CUs are. A low value of affinity between two CUs signifies that it is logical to separate the two CUs for forming tasks. The two types of edges in the CU graph are replaced by a single undirected edge. The weight of this edge is the affinity between the two CUs. Figure 4b demonstrates the graph with the two types of edges between the vertices replaced by single edge with affinity as the weight.

For the purpose of the analysis, every connected component of a CU graph is considered a graph in itself. The next step is to calculate the minimum cut in a connected component using Stoer-Wagner's algorithm [20]. In graph theory, a *cut* of a graph is a partition of the vertices of a graph into two disjoint subsets that are connected by at least one edge. A *minimum cut* is a set of edges that has the smallest number of edges (for an unweighted graph) or smallest sum of weights possible (for a weighted graph).

Identifying the minimum cut of a CU graph divides the graph into two components that were weakly linked. This indicates that the code is being separated with the minimum number of dependences and common instructions affected. For each component, the minimum cut is calculated further to divide it into two more components. The process is repeated recursively over all the components of the CU graph until the components available are CUs themselves. Figure 4c shows the CU graph with a minimum cut. Removing edges 54-56 and 55-57 together partitions the graph into two disjoint connected components with the smallest sum of weights between them removed. Thus these two edges are the minimum cut for the graph.

The output of this process are several groups of CUs where CUs are strongly linked within the group and can be considered tasks. Since this process is repeated recursively on all the connected components, the tasks are produced at various levels of granularity and can later be analyzed to identify the optimal level at which they can be parallelized. The end result is a list of tasks that contains tasks ranging from coarse-grained to fine-grained. The next step is to rank these tasks by putting the most promising tasks first. This is done by the identifying the functions that take the largest percentage of the program execution time and prioritizing tasks formed from these functions in the decreasing level of their granularity. The program execution tree generated by our dynamic analysis provides any control dependences existing between the tasks. In the absence of control or data dependences, any two tasks can be considered a valid parallelization opportunity.

4. EVALUATION

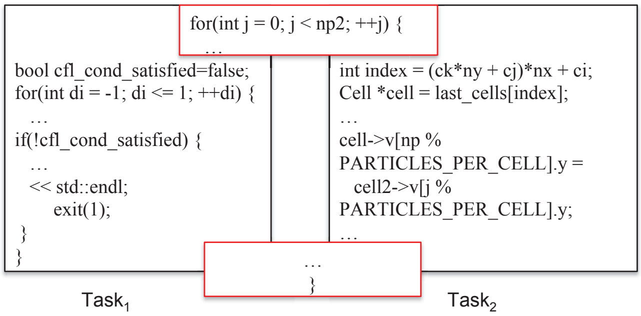
To measure the effectiveness of our approach, we analyzed applications from the STARBENCH parallel benchmark suite [2] and the PARSEC benchmark suite [3]. We chose these benchmark suites because both cover a broad range of application domains, various parallel patterns, and various sizes of applications. The test cases were compiled using a modified version of Clang 3.3 which integrates our analysis programs with the existing inspection and traversal routines of LLVM. All 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). All the benchmark programs were compiled with option *-g -O0*.

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

4.1 Parallelization based on the suggested tasks

In this section, we investigate some of the applications of the PARSEC benchmark suite. We parallelized these applications based on the tasks formed by using CUs or by directly considering CUs as tasks. We assigned these tasks to separate threads and calculated the speedup obtained. We parallelized these cases mainly using OpenMP *section* and *task* directives. Speedup reported refers to the ratio of execution time of the tasks run in parallel to the execution time of the sequential version of the same code region (function). As a result, it is called *local speedup*. The speedups represent an average of five independent executions of the programs. Table 1 shows the results of the applications parallelized. ”# of Threads” shows the number of threads used to parallelize the suggestions. Local speedups for each case can also be seen for the given number of threads in the table. Column ”Code Refactoring” indicates whether refactoring the code was necessary to parallelize the program. Refactoring the code mainly involved privatization, adding necessary synchronization or replicating some part of the code across multiple threads.

In addition to revealing the existing parallelism, the process of grouping tightly coupled code to identify tasks also reduces the number of dependences that need to be analyzed and resolved to parallelize any given suggestion. This reduces the amount of effort required to parallelize the application. In Table 1, ”# of RAW dep.” shows the total number of RAW dependences identified by our dynamic analysis within the given function. ”# of RAW dep. resolved” shows the actual number of RAW dependences that needed to be resolved to run the identified tasks in parallel.



```
for(int j = 0; j < np2; ++j) {
  …
```

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

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

```
…
}
```

Figure 5: Tasks identified using the CU graph for the function *RebuildGrid()* of *Fluidanimate*.

It is important to note that when a loop is considered as an identified task, it is assigned to a separate thread. It is not examined whether the individual iterations (a group of iterations) of the loop are independent of each other. Several loop analysis techniques (e.g. our previous works [12] [13]) have already identified and explored independent iterations for loop parallelism. In this paper, our focus is mainly on identifying code blocks that can be considered as tasks and our evaluation does not consider concurrent execution of iterations for loop parallelism.

4.1.1 *Fluidanimate*

Fluidanimate uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. We identified tasks that are useful for parallelism in three different places in *Fluidanimate*.

RebuildGrid() : Analyzing the tasks found in the function *RebuildGrid*, we realized that a code section performing Courant-Friedrichs-Lewy (CFL) condition check can be executed in parallel with the remaining part of the function. Three CUs correspond to the first task which performs the CFL condition check.The other 9 CUs belonging to the second task represent the remaining part of the function.

Figure 5 shows the tasks identified by merging the two sets of CUs separately. The task on the left represents the CFL condition check while the task on the right represents the remaining part of the function. As suggested, we parallelized *RebuildGrid* using two threads, with each thread executing a task. The *for* loop enclosing the two tasks in the sequential version of the program was replicated over both the tasks for parallelization and is highlighted in red. Local speedup for this case was found to be 1.6 with the two tasks running in parallel. The total number of RAW dependences within this function were found to be 300. The RAW dependences that actually prevented parallelism were resolved by replicating the loop over both the tasks. The remaining dependences were within the tasks identified.

ProcessCollisions() : This function contains six loop nests checking if a particle hits any of the six surfaces of the 3-D cube space. The order in which the checking of surfaces takes place does not affect the correctness of the process.

Table 1: Summary of parallelization results.

Program	Function	Code Refactoring	# of Threads	Local Speedup	# of RAW dep.	# of RAW dep. resolved
Fluidanimate	*RebuildGrid()*	Yes	2	1.60	300	16
Fluidanimate	*ProcessCollisions()*	No	4	1.81	121	0
Canneal	*netlist_elem::routing_cost_given_loc()*	Yes	2	1.32	19	2
Blackscholes	*CNDF()*	No	2	0.98	38	0
Fluidanimate	*ComputeForces()*	Yes	3	1.52	32	6

The analysis identified a CU for every *for* loop which was considered as an independent task. We parallelized the function using four threads and assigned the tasks to different threads. The local speedup achieved was 1.81. Since the loops were independent of each other, none of the 121 RAW dependences in the function had to be resolved.

ComputeForces() : While analyzing this function, we came across a case where a pipeline is discovered while examining our tasks with data dependences between them. The tasks formed by merging the CUs and dependences between the tasks are shown in Figure 6. The edges signify the RAW dependences between the tasks with the number of dependences as weights. The four tasks represent the four stages of the pipeline, where each of them is a loop nest within the function. These stages perform the following tasks in the given order: read input data, compute density (two steps), and compute force.

$Task_1$ takes only a small amount of time to read the input data. Since it is the smallest task (and a single CU) out of the four tasks, implementing it as an individual stage in the pipeline is not efficient. As a result, *ComputeForces()* is transformed into a pipeline with three stages. Stage 1 performs step one of the density computation, stage 2 performs step two of the density computation, and stage 3 of the pipeline performs force computation, respectively. We compared the performance of our parallel implementation with the sequential version using the *simlarge* input from the PARSEC benchmark. The sequential version took 12.66 seconds on an average, while the parallel version took only 8.35 seconds, leading to a speedup of 1.52 with three threads.

4.1.2 Canneal

Canneal is a kernel application that uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design. In *Canneal*, task parallelism is found in the function *netlist_elem::routing_cost_given_loc* . The function first calculates *in* and *out* cost separately. Then it adds them together as the routing cost. Figure 7 shows the CUs identified corresponding to this calculation. In this case, the CUs themselves were considered as tasks. We parallelized this function using two threads by adding the necessary synchronization and obtained a local speed up of 1.32. Only 2 out of 19 RAW dependences had to resolved which were addressed by the synchronization.

4.1.3 Blackscholes

In *Blackscholes*, the body of the function *CNDF()* is identified as a task and there are two independent calls to this function. Both the calls are assigned to two different threads but the overhead of creating/destroying threads is found to be more than the benefit of parallelization. In these cases, our parallel version runs slower than the sequential version.

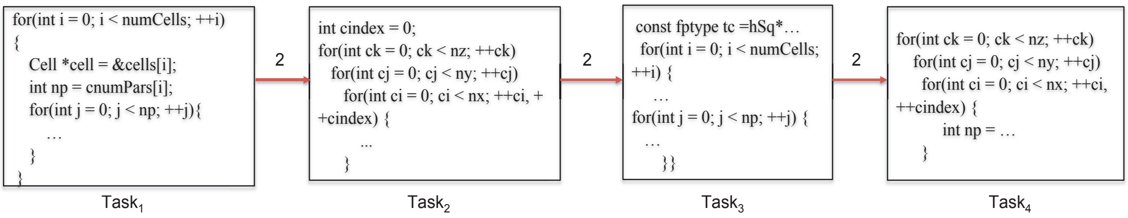
The pipeline discussed for *ComputeForces()* in *Fluidanimate* is only one of the many parallel patterns that can be found by combining CUs with the dynamic analysis information. With more investigation, CU graph can be also mapped to a TBB Flow Graph [17], which is a more general parallel programming construct. It can be observed that in general, identification and parallelization of tasks could be useful in cases where the tasks identified are computation intensive and are largely independent of each other. However, it also needs to be observed that the identification of tasks that can run in parallel cannot keep up with the increasing number of cores, and hence this type of parallelism does not scale well. Balancing the workload of these tasks across multiple threads is also a challenge.

4.2 Comparison with the existing parallel implementations

Our next evaluation strategy involves providing a comparison of the identified tasks with the existing parallel versions of the applications for the STARBENCH parallel benchmark suite. Table 2 shows the overview of the evaluation performed. Column 2 of the table shows the location in the sequential version of the program that was parallelized in the parallel version. Table 2 also shows the matching task identified using our approach in the sequential version. The tasks were identified by prioritizing the main algorithm functions and the functions that consumed the majority of the total execution time of the program as shown in Table 2.

4.2.1 *c-ray*

c-ray is a simple brute-force ray tracer. It takes an input file with simple scene description and renders an image in PPM binary format. The function *render_scanlines()* performs this operation as seen from Table 2. This function was prioritized in particular to select tasks because it takes approximately 100% of the total execution time as observed from the call graph of the program. The analysis identified a task corresponding to the body of this function from the CU graph and it was made up of 4 CUs. Figure 8 shows a connected component from the CU graph of *c-ray* which was identified as a task. This connected component was one of the tasks identified for the function *render_scanlines()*. It can be observed in this figure that the CUs contain code sections that use a group of variables together to perform a computation and the connected component represents these code sections together as a task. Listing 4 shows the output produced corresponding to this task by our analysis. The task identified contains the 4 CUs demonstrated in Figure 8 and the line numbers that correspond to the task.



```
for(int i = 0; i < numCells; ++i)
{
   Cell *cell = &cells[i];
   int np = cnumPars[i];
   for(int j = 0; j < np; ++j){
      …
   }
}
```

```
int cindex = 0;
for(int ck = 0; ck < nz; ++ck)
  for(int cj = 0; cj < ny; ++cj)
    for(int ci = 0; ci < nx; ++ci, +
+cindex) {
      ...
    }
```

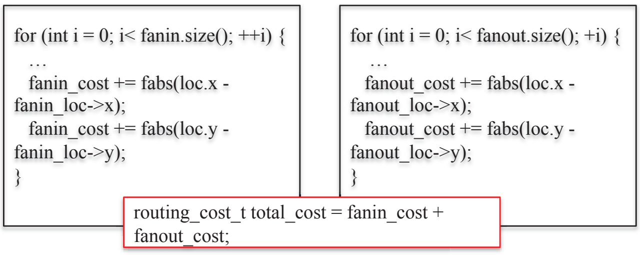
```
const fptype tc =hSq*…
 for(int i = 0; i < numCells;
++i) {
  …
for(int j = 0; j < np; ++j) {
 …
   }}
```

```
for(int ck = 0; ck < nz; ++ck)
   for(int cj = 0; cj < ny; ++cj)
     for(int ci = 0; ci < nx; ++ci,
++cindex) {
       int np = …
     }
```

Figure 6: Tasks as stages of a pipeline for the function *ComputeForces* of *Fluidanimate*.

Table 2: Summary of comparison results.

Program	Location Parallelized in the Parallel Implementation	% of Execution Time	Matching Task Suggestion	# of CUs used
c-ray	*c-ray-mt.c : render_scanlines()* : 273-296	∼100	*c-ray-mt.c : render_scanlines()* : 277-292	4
k-means	*k-means.c : cluster()* :71-85	99.6	*k-means.c : cluster()* :72-83	3
md5	*md5_bmark.c : process()* : 113-122	93.5	*md5.c : MD5_Update()* : 215-238	7
rotate	*rotation_engine.cpp : RotateEngine::run()* : 97-183	90.3	*rotation_engine.cpp : RotateEngine::run()* : 111-158	6
rgbyuv	*bmark.c : processImage()* : 138-171	∼100	*bmark.c : processImage()* : 145-162	7
ray-rot	*c-ray-mt.c : render_scanlines()* : 273-296 *rotation_engine.cpp : RotateEngine::run()* : 97-183	97.2 1.3	*c-ray-mt.c : render_scanlines()* : 277-292 *rotation_engine.cpp : RotateEngine::run()* : 111-158	4 6
rot-cc	*rotation_engine.cpp : RotateEngine::run()* : 97-183 *bmark.c : processImage()* : 138-171	54.7 25.5	*rotation_engine.cpp : RotateEngine::run()* : 111-158 *bmark.c : processImage()* : 145-162	6 7
bodytrack	*Observation.cpp : observe_tfunc()* : 117-124 *ParticleFilter.cpp : pfworker_tfunc()* : 55-95 *Projection.cpp : pj_tfunc()* : 154-151	NA	NA	NA

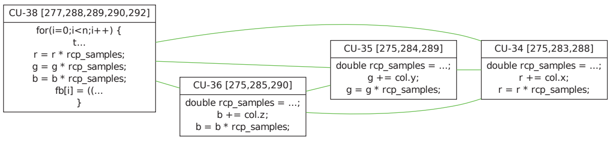


```
for (int i = 0; i< fanin.size(); ++i) {
  …
  fanin_cost += fabs(loc.x -
fanin_loc->x);
  fanin_cost += fabs(loc.y -
fanin_loc->y);
}
```

```
for (int i = 0; i< fanout.size(); +i) {
  …
  fanout_cost += fabs(loc.x -
fanout_loc->x);
  fanout_cost += fabs(loc.y -
fanout_loc->y);
}
```

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

Figure 7: CUs for the function *netlist_elem::routing_cost_given_loc* of *Canneal*.



```
CU-38 [277,288,289,290,292]
for(i=0;i<n;i++) {
t...
r = r * rcp_samples;
g = g * rcp_samples;
b = b * rcp_samples;
fb[i] = ((...
}
```

```
CU-36 [275,285,290]
double rcp_samples = ...;
b += col.z;
b = b * rcp_samples;
```

```
CU-35 [275,284,289]
double rcp_samples = ...;
g += col.y;
g = g * rcp_samples;
```

```
CU-34 [275,283,288]
double rcp_samples = ...;
r += col.x;
r = r * rcp_samples;
```

Figure 8: Connected Component of the CU Graph of *c-ray* corresponding to function *render_scanlines()*.

Listing 4: Task output corresponding to Figure 8 for *c-ray* produced after graph partitioning.

```
<TaskList>
    <Task id="0:1">
    <CUs count="4">34,35,36,38,</CUs>
    <lines count="12">1:275,1:277,1:279,1:281,
            1:282,1:283,1:284,1:285,
            1:288,1:289,1:290,1:292
    </lines>
    </Task>
    ...
<TaskList>
```

In the parallel implementation of the program, the target image is divided into three work units. The threading overhead is reduced by coarsening the task granularity. This is achieved by grouping the scanlines together into blocks. In the PThreads version, the function *render_scanlines()* is executed by the different threads. Table 2 shows the location where parallelization is performed in *c-ray*. Comparing the parallelization performed in *c-ray* with the identified task confirms that the the proposed task is valid and logical.

4.2.2 *k-means*

The *kmeans* kernel executes the k-means clustering algorithm [8]. It is used in the domains of data-mining and artificial intelligence. The application consists of two iteratively repeated phases. One is a clustering phase and the other is a reduction phase that computes new clusters. In the sequential version, the function *kmeans()* calls the function *cluster()* which performs the clustering phase. The remaining body of the function *kmeans()* performs the reduction phase. The function *cluster()* takes 99.6% of the total execution time of the program. This makes it a good candidate for analysis of the tasks from the list of the tasks identified for the program.

The analysis identifies both of the aforementioned phases individually as tasks. The *cluster()* function is identified as a task by grouping 3 CUs from the CU graph. For the reduction phase, the part of the function *kmeans()* that performs this phase is identified as task. The task corresponding to second phase is dependent on the first.

Comparison between the PThreads implementation of the clustering phase to the sequential task can be seen in Table 2. In the PThreads version of the program, every thread executes the function *work()*. The body of this function contains the same code as the sequential version of *cluster()*. The reduction phase is run by the main thread after this.

4.2.3 md5

md5 uses a standard implementation of the MD5 hash algorithm [18] and produces hash values. In the application, the function *process()*, which takes 93.5% of the total execution time, calls *MD5_Init()*, *MD5_Update()* and *MD5_Final()*. These three functions take approximately 0.0%, 46.7% and 46.7% of the total execution time respectively. These three functions need to run in the mentioned order. Our analysis does not directly identify the function *process()* as a task since it does not contain any computation but only contains function calls. However, the body of the function *MD5_Update()* is identified as a potential task using 7 CUs from the CU graph. The functions *MD5_Init()* and *MD5_Final()* perform initialization and assignment operations that are largely independent of each other. Hence, their contents were not identified as a task.

Table 2 shows the parallelization in *md5*. In the PThreads version of the program, the function *process()* is run on every thread. While the analysis does not provide an exact match with respect to this function, the task identified for *MD5_Update()* from the sequential version serves as a partial representation of the parallelized task from the PThreads version.

4.2.4 rotate

rotate takes the binary representation of an image and rotates it by 0, 90, 180 or 270 degrees. The parallelization approach in *rotate* is similar to *c-ray*. The function that performs the main algorithm is *RotateEngine::run()*. It takes 90.3% of the total execution time of the program. In the first step, this function determines the target image size. Then it operates on each pixel as the second step. Our analysis identifies two tasks within the body of the function *RotateEngine::run()*. Both the steps are individually identified as tasks with RAW dependences between them. The first step calls a function *RotateEngine::rotatePoint()*. The second step comprises of the rest of the body of the function *RotateEngine::run()*.

The PThreads version of the program is implemented in the same way as *c-ray*. A new function *RotateEngine::computeRow()* is defined and holds the contents of one of the tasks identified. It performs the second step from the *RotateEngine::run()* of the sequential version. Table 2 shows the comparison. Use of a new function in this way to perform the operations identified as tasks also confirms that our analysis can identify tasks that are logical.

4.2.5 rgbyuv

rgbbyuv is an RGB to YUV colour converter which processes PPM format pictures. The function *processImage()* is the main algorithm function performing the actual conversion and controlling the iterations in the sequential version of the program. This function takes approximately 100% of the total execution time of the program. Table 2 shows the task identified for this function. The task corresponds to the body of *processImage()* and is formed using 7 CUs from the CU graph.

In the parallel version, *processImage()* controls the iterations but a new function *convertThread()*, is created. This function contains the rest of the code from the sequential version of *processImage()*. Table 2 also shows the parallelization in *rgbyuv*. This is similar to the application *rotate* as it confirms that the part of the function which was identified as a task is the same part that is parallelized in the PThreads version.

4.2.6 rot-cc and ray-rot

rot-cc and *ray-rot* contain a combination of *rotate*, *rgbyuv* and *c-ray*, *rotate* kernels respectively. The task identification and corresponding matches in the parallel versions for these kernels is discussed above.

4.2.7 bodytrack

bodytrack is a computer vision application which tracks a human body with multiple cameras through an image sequence. The parallel implementation of *bodytrack* features three thread functions that run the functions *observe_tfunc()*, *pfworker_tfunc()* and *pj_tfunc()* in parallel. All the functions have little to no computation within them which can be seen from their size in Table 2. These functions call other library functions which were not analyzed for the identification of tasks. Hence, our analysis could not report any tasks corresponding to the parallelized version of bodytrack.

5. CONCLUSION AND FUTURE WORK

This paper discusses a novel approach to identifying computational units (CU) that form the basic building block of a parallel task. These CUs are detected across the boundaries of the program regions and are used to form tasks that can run in parallel. The identified tasks contain code that is tightly coupled because of the dependences within them. This in turn reveals the code sections that can run in parallel or the code sections that are weakly dependent but can still run in parallel after applying techniques like replication of code, privatization or other synchronization methods. Use of dynamic analysis information is also demonstrated to enhance the task formation and to check whether the tasks can be run in parallel. We evaluated our approach by analyzing applications from two benchmarks. We parallelized the applications based on the task suggestions obtained. We also analyzed the parallel implementations of the sequential programs to verify that the code sections parallelized have a matching task suggestion based on our approach.

In the future, we would like to extend our work to support various parallel patterns which will require better utilization of the dynamic analysis information. Support will be added for mapping a CU graph to parallel constructs like TBB Flow Graph to exploit more general parallelism. Automatic identification of the code sections that would be involved in code refactoring like replication, privatization etc., and scheduling the identified tasks for parallelization will be explored in detail. Load balancing the tasks that can run in parallel across multiple threads will be investigated. Use of machine learning techniques to estimate the proper size of tasks for different applications will also be investigated in the future.

6. REFERENCES

[1] Llvm language reference manual.

[2] M. Andersch, B. Juurlink, and C. C. Chi. A benchmark suite for evaluating parallel programming models. In *Proceedings 24th Workshop on Parallel Systems and Algorithms*, 2011.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: An integrated framework for mpsoc application parallelization. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 754–759, 2008.

[5] I. Foster. Task parallelism and high-performance languages. *IEEE Parallel and Distributed Technology*, 2:27–36, 1994.

[6] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 458–469, New York, NY, USA, 2011. ACM.

[7] T. GROSSER, A. GROESSLINGER, and C. LENGAUER. Polly âĂŤ performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[8] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*, volume 2. Springer, 2009.

[9] A. Ketterlin and P. Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 45, pages 437–448, Washington, DC, USA, 2012. IEEE Computer Society.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[11] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[12] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf. Discopop: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*, pages 1–10. Springer International Publishing, 2015 (to appear).

[13] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *Proceedings of the 42nd International Conference on Parallel Processing*, PSTI '13, pages 1004–1013, Washington, DC, USA, 2013. IEEE Computer Society.

[14] Z. Li, A. Jannesari, and F. Wolf. Discovering parallelization opportunities in sequential programs âĂŤ a closer-to-complete solution. In *Proceedings of International Workshop on Software Engineering for Parallel Systems*, SEPS '14, pages 1–10, Portland, OR, USA, 2014.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[16] 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, 2005.

[17] C. Pheatt. Intel&reg; threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[18] R. Rivest. The md5 message-digest algorithm. 1992.

[19] Z. Sura, K. O'Brien, and J. Brunheroto. Using multiple threads to accelerate single thread performance. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 985–994, Washington, DC, USA, 2014. IEEE Computer Society.

[20] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[21] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.