Finding Parallel Regions with Temporal Planning

Claudio Scheer

Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre - RS, Brazil
claudio.scheer@edu.pucrs.br

Abstract

Testing whether a region in a program can be run in parallel is not an easy task. As an approach to this problem, this paper proposes a formalization of a source code compiler with PDDL. With this approach, it is possible to reduce the identification of parallel regions in a source code to a conventional search problem. The STP planner was used to find a temporal plan in which some actions can be execute in parallel. The results show that temporal planning can be an alternative to find parallel regions, but more studies are still needed, mainly on how to translate a source code to a PDDL problem.

There are still many programs that do not exploit the parallel capacity of today's multiprocessors. One of the main reasons for this is because the cost to rewrite a program is high. Among the steps involved in the process of migrating a program to a parallel approach, the most expensive is to identify the regions of the program than can be executed in parallel. In addition to identifying the regions, it is necessary to validate whether the parallel execution of that region brings positive results.

According to (del Rio Astorga et al. 2018), loops detection, variable dependencies, identifying whether the arguments are read or written, among other analyzes, are the main patterns that identify a region parallel in a program. However, over time, these caracteristics may change and the static analysis will also need to be changed.

Therefore, instead of using a static analysis of the source code, the approach in this paper reduces the problem of identifying a parallel region to a search problem. The compiler domain uses PDDL as a planning representation. In a nutshell, the domain works as a source code compiler and the problem formalization as the source code. The Section Formalization discusses in more detail how the PDDL domain works.

The PDDL domain describes actions that can be performed in an initial state, to achieve a set of goals. In this paper, the initial state is the dependency tree of the variables and operations of the source code. The set of goals is the complete execution of all instructions in the source code, taking into account the dependency tree.

A problem formalized in PDDL cannot understand source code written in C++ as input. The initial state must be a set of predicates. Hence, the source code provided as the initial state for the planner must be mapped to a set of predicates. This process is the bottleneck of the proposed approach. Some ideas for this problem are discussed in the Section Source Code Translation.

The Section Bibliography discusses what temporal planning is and some details of the STP planner used. In the Section Results, I show the results obtained in a real world example. Finally, the Section Conclusion leaves some questions that still need to be answered in this new approach to find parallel regions.

Bibliography

Temporal Planning

According to (Haslum et al. 2019), actions in temporal planning have a duration. Therefore, the planner searches for a schedule in which independent actions can be executed in parallel.

There are different approaches that can be used to formalize temporal actions in PDDL. In this paper, `:durative-action` was used. Durative actions is represented in the four sections listed below.

- `:parameters`: parameters needed to execute the action;

- `:duration`: time the action takes to run;

- `:condition`: conditions that need to be respected to apply the effects;

- `:effect`: effects that will be applied to the state;

The sections `:condition` and `:effect` are separated in three categories: `at start`, `over all` and `at end`. As described by (Haslum et al. 2019), these categories represent the conditions and effects used at each stage of the durative action. The `at start` statements are used when starting the action. The `over all` statements are used during the time the action is being executed. The `at end` statements are used at the end of the action.

STP

The STP (Simultaneous Temporal Planner) planner, introduced by (Blanco et al. 2018), was used to find the best tem-

poral plan for a problem. STP is a modified version of the Fast Downward (Helmert 2006) planner that allows solving temporal planning problems. The version used in this paper is provided by the Artificial Intelligence and Machine Learning Group - Universitat Pompeu Fabra[1].

The STP planner receive as a parameter the maximum number of actions that can be executed at the same time. When finding parallel regions, this parameter is a problem, because we do not know how many instructions can be executed in parallel. Therefore, in some cases, it is necessary to test different values for this paramenter. It is important to note that the higher this number, the longer the STP planner takes to find a temporal plan.

Source Code Translation

The correct translation of the source code is the main point to find parallel regions using temporal planning. And, this paper do not cover all the possibilities in a source code. The focus is on source codes with assignments and operations between two variables and simple loops.

It is easy to formalize in a PDDL problem a source code without loops. The dependency tree is easy to create. More about these type of problems is discussed in Section Results.

However, when the source code has a loop, the level of abstractions becomes an issue. For example: is it necessary to formalize the assignments that control the index of a for loop? Two examples of a for loop in C++ are shown below.

The Listing 1 shows an algorithm that adds up all the items in a vector. In this case, the variable `i` accesses different loctions of the vector. In theory, the sum of `s` plus `x[i]` can be executed in parallel. Since the order of a sum does not matter and `s` is shared between threads[2], in the end, `s` will be equal to 6.

```
1  int main()
2  {
3      int s = 0;
4      std::vector<int> x = {1, 2, 3};
5      for (int i = 0; i < x.size(); i++)
6      {
7          s += x[i];
8      }
9      return 0;
10 }
```

Listing 1: Sum of vector - C++

In the formalization of the Listing 1, the assignment of the variable `i` can be ignore. However, in the Listing 2, where the initialization of a position in the array depends on the value of the previous position, the variable `i` cannot be ignored. If the loop is executed in parallel, the previous position may not have been initialized yet, throwing an exception.

```
1  int main()
2  {
3      int a[3] = {0};
```

---

[1]https://github.com/aig-upf
[2]The concurrency between variables is not taked into account here.

```
4      a[0] = rand();
5      for (int i = 1; i < 3; ++i)
6      {
7          a[i] = a[i - 1] + rand();
8      }
9      return 0;
10 }
```

Listing 2: Initializing array - C++

There are many variants that can happen in a source code an the level of abstraction depends on how the program was built.

Formalization

Two `:durative-action` were used to formalize the PDDL domain. One to handle assignment instructions and another to handle binary operations. Binary operations must be understood as any operations (sum, multiplication, XOR, etc) that involve two variables.

All `:durative-action` must have a duration time. However, since the goal is not to cover the execution time for specific instructions, the duration was set to 1 for all actions. This approach ensures that when operations cannot be executed in parallel, the total execution time will be increased by one. In future work, it may be a good approach to define a different time for each instruction.

By definition, each operation and assignment instruction must have an identifier. With the identification, the instructions dependency tree can be created. The dependency tree makes it easy to test whether all dependencies for a specific instruction have already been executed or not. As each problem has a specific dependency tree, the dependency tree must be defined in the initial state of the problem. In addition to the dependency tree, the identifier of the binary operations and assignments must also be defined in the initial state.

As discussed in Section Temporal Plannign, the sections `:conditions` and `:effects` have three different categories. In the proposed formalization, all preconditions must be respected at the beginning of the action and the effects is applied only at the end of the action.

Assignment Action

The assignment action will receive the assignment instruction and the instruction identifier as parameters. The first two conditions will ensure that the identifier belongs to the the instruction and that the assignment has not yet been executed.

The third condition is common the all durative actions. The `forall` operator loops through all identifiers in the dependency tree, testing whether the identifier is the parent of the current instruction and whether the parent instruction has already been executed.

The Table 1 shows the truth table that represents the condition in the `forall` operator. The variable A represents that the current identifier is a child of the parent identifier. B represents that the parent identifier has already been executed. As shown in the truth table, the only case where the condition returns false is when the current identifier is a

child of the parent identifier and the instruction represented by the parent identifier has not yet been executed.

A	B	$\neg A \vee B$
1	1	1
1	0	0
0	1	1
0	0	1

Table 1: Dependency tree condition truth table

The `:effect` of the assignment action is to mark the instructions as executed. The Listing 3 shows the formalization of the assignment action.

```
1  (:durative-action assignment
2     :parameters (
3        ?instruction_id - id
4        ?id - assignment
5     )
6     :duration (= ?duration 1)
7     :condition (and
8        (at start (assignment_id ?id ?
              instruction_id))
9        (at start (not (executed_assignment ?
              id)))
10       (at start (forall (?parent - id)
11          (or
12             (not (dependency_tree ?parent ?
                   instruction_id))
13             (executed_instruction ?parent)
14          )
15       ))
16    )
17    :effect (and
18       (at end (executed_instruction ?
              instruction_id))
19       (at end (executed_assignment ?id))
20    )
21 )
```

Listing 3: Formalization of the assignment action

Binary Operation Action

This action is responsible for executing a binary operation between two variables. The action takes as parameters the two assignment variables that will be used in the operation and the assignment instruction that will receive the result of the binary operation.

The conditions to execute the action are that the binary operation has not yet been executed and the assignment instructions of the two input variables have already been executed.

As shown in Listing 4, the effects are similar to the previous action: the binary operation is marked as executed, releasing the next instructions that holds in this operation.

```
1  (:durative-action binary_operation
2     :parameters (
3        ?instruction_id - id
4        ?idA - assignment
5        ?idB - assignment
```

```
6        ?operation_id - operation
7        ?idC - assignment
8     )
9     :duration (= ?duration 1)
10    :condition (and
11       (at start (operation_id ?operation_id
              ?instruction_id))
12       (at start (forall (?parent - id)
13          (or
14             (not (dependency_tree ?parent ?
                   instruction_id))
15             (executed_instruction ?parent)
16          )
17       ))
18       (at start (not (executed_operation ?
              operation_id)))
19       (at start (not (
              executed_binary_operation ?idA ?
              idB ?operation_id ?idC)))
20       (at start (executed_assignment ?idA))
21       (at start (executed_assignment ?idB))
22    )
23    :effect (and
24       (at end (executed_instruction ?
              instruction_id))
25       (at end (executed_operation ?
              operation_id))
26       (at end (executed_binary_operation ?
              idA ?idB ?operation_id ?idC))
27    )
28 )
```

Listing 4: Formalization of the binary operation action

Results

In this section, the results achieved are discussed. Loops are not shown in the results because more research is still needed to better understand how to abstract the set of predicates from a source code. Therefore, with regard to the objective of the paper, it is expected that the temporal plan found by the planner shows the instructions that can be execute in parallel. In Listing 5, the source code used in the examples discussed is presented. The source code is simple, but it will help to show the capacity of the proposed approach.

The source code in Listing 5 adds two variables and store the result in the variable `c`. The variables `a` and `b` are not dependent on each other. Therefore, the two assignment instructions can be executed in parallel, as shown in Figure 1.

```
1  int main()
2  {
3     int a = 3;
4     int b = 3;
5     int c = a + b;
6     return 0;
7  }
```

Listing 5: Source code

The formalization of the Listing 5 is shown in Listing 6.

```
1  (:init
2     (executed_instruction id0)
3
```
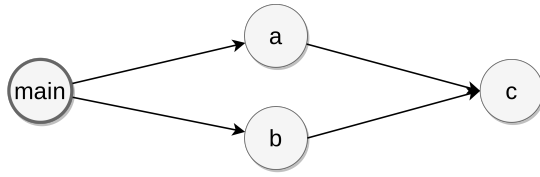
Figure 1: Dependency tree

```
4      (assignment_id assignmentA id1)
5      (assignment_id assignmentB id2)
6      (operation_id sumAB id3)
7      (assignment_id assignmentC id4)
8
9      (dependency_tree id0 id1)
10     (dependency_tree id0 id2)
11     (dependency_tree id1 id3)
12     (dependency_tree id2 id3)
13     (dependency_tree id3 id4)
14 )
15
16 (:goal (and
17     (executed_assignment assignmentA)
18     (executed_assignment assignmentB)
19     (executed_binary_operation assignmentA
          assignmentB sumAB assignmentC)
20     (executed_assignment assignmentC)
21 ))
```

Listing 6: Source code formalization

The main point in Listing 6 is on lines 9 and 10. In these lines, the instructions a and b are defined as dependent only on the main instruction[3]. The STP output of this formalization is represented visually in Figure 2.

0.000	1.000	2.000
assignmentA		
assignmentB		
	sumAB	
		assignmentC

Figure 2: STP parallel plan

Figure 2 shows that, as soon as the program starts, the instructions assignmentA and assignmentB can be executed at the same time. However, to execute the sumAB instruction, it is necessary to wait until the previous instructions are executed. Finally, to execute the assignmentC instruction, all previous instructions must be executed.

Imagining a source code similar to Listing 5, where the variable b is a result of the variable a plus 1, the dependency tree would be different. Now, line 10 in Listing 6 will be replaced by (dependency_tree id1 id2), ensuring that variable b is assigned only after the variable a. The output of this second example is shown in Figure 3.

---

[3]The main instruction represents the starting point of the application. In this example, id0.

0.000	1.000	2.000	3.000
assignmentA			
	assignmentB		
		sumAB	
			assignmentC

Figure 3: STP non-parallel plan

Conclusion

The idea of reducing the search for parallel regions in a source code to a search problem is unconventional. Therefore, there are still many questions to be answered:

- What is the best way to abstract a source code?

- How can nested loops be abstracted?

- In a for loop, for example, when should formalization take into account the index assignment (usually the variable i)?

- Are the two durative actions defined in this paper sufficient?

- Can the temporal plan be mapped back to the source code?

- How to choose the maximum number of parallel instructions in STP correctly?

- Is STP the best temporal planner for this problem?

- Shouldn't different duration times be used for each assignment and binary operation?

- Do all regions need to run in parallel?

- How should if instructions be handled?

Despite all these question, there is a guarantee: this approach can identify parallel regions. In addition, the possibility of abstracting the source code to a set of predicates allows to reduce the problem to the same proprierties, but with lower overheads.

References

[Blanco et al. 2018] Blanco, D. F.; Jonsson, A.; Verdes, H. L. P.; and Jimenez, S. 2018. Forward-search temporal planning with simultaneous events.

[del Rio Astorga et al. 2018] del Rio Astorga, D.; Dolz, M. F.; Snchez, L. M.; Garca, J. D.; Danelutto, M.; and Torquati, M. 2018. Finding parallel patterns through static analysis in c++ applications. *The International Journal of High Performance Computing Applications* 32(6):779–788.

[Haslum et al. 2019] Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

[Helmert 2006] Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191246.