Finding Parallel Regions with Temporal Planning

Claudio Scheer
Master's Degree in Computer Science
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. I used the STP planner to find a temporal plan in which some actions can be execute in parallel. The results show that temporal planning can be an alterantive 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. 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 reduce the problem of identifying a parallel region to a search problem. The compiler domain was formalized using PDDL. 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 idea of a PDDL domain is to describe 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 and 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, for example. The initial state must be a set of predicates. Hence, the source code provided as an 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 Plannign

According to (Haslum et al. 2019), actions in temporal planning have a duration. Therefore, the planner will try to find a schedule in which some actions can be executed in parallel.

There are different approaches that can be used to formalize temporal actions with PDDL. In this paper, I used `:durative-action`. This action is represented in four sections, as 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 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

I used the STP (Simultaneous Temporal Planner) planner, introduced by (Blanco et al. 2018), to find the best temporal plan. STP uses a modified version of the Fast Downward (Helmert 2006) planner that can generate a temporal plan.

The version used in this paper is provided by the Artificial Intelligence and Machine Learning Group - Universitat Pompeu Fabra[1].

The STP planner needs to 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 more time STP 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. However, this paper would not cover all 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 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 have a loop, the level of abstractions becomes a problem. 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 how to make a sum of all 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 Listing 2, where the initialization of a position in the array depends on 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};
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 }
```

[1]https://github.com/aig-upf

[2]The concurrency between variables is not taked into account here.

Listing 2: Initializing array - C++

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

Formalization

I used only two `:durative-action` to formalize the problem. 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 my objective is not to cover the execution time of the instructions in this paper, I set the duration time 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 a future work, it may be a good approach to define different duration time for each instruction.

By definition, each operation and assignment instruction must have an identifier. With this identification, we can create the dependency tree for the instructions. The dependency tree makes it easy to test whether all dependencies for a specific instruction have already been executed. Since each problem has a specific dependency tree, the dependency tree must be defined in the initial state of the problem. In the initial state, in addition to the dependency tree, the identifier for the operation and assignment instructions must also be defined.

As discussed in Section Temporal Plannign, the sections `:conditions` and `:effects` have three different categories. In my 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 a parameter. 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 parent of the current instruction and whether the parent instruction have already been executed.

The Table 1 shows the truth table that represents the condition within 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.

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

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

```
(:durative-action assignment
   :parameters (
      ?instruction_id - id
      ?id - assignment
   )
   :duration (= ?duration 1)
   :condition (and
      (at start (assignment_id ?id ?instruction_id))
      (at start (not (executed_assignment ?id)))
      (at start (forall (?parent - id)
         (or
            (not (dependency_tree ?parent ?instruction_id))
            (executed_instruction ?parent)
         )
      ))
   )
   :effect (and
      (at end (executed_instruction ?instruction_id))
      (at end (executed_assignment ?id))
   )
)
```

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.

Here, 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.

```
(:durative-action binary_operation
   :parameters (
      ?instruction_id - id
      ?idA - assignment
      ?idB - assignment
      ?operation_id - operation
      ?idC - assignment
   )
   :duration (= ?duration 1)
   :condition (and
      (at start (operation_id ?operation_id ?instruction_id))
      (at start (forall (?parent - id)
         (or
            (not (dependency_tree ?parent ?instruction_id))
            (executed_instruction ?parent)
         )
      ))
      (at start (not (executed_operation ?operation_id)))
      (at start (not (executed_binary_operation ?idA ?idB ?
          operation_id ?idC)))
      (at start (executed_assignment ?idA))
      (at start (executed_assignment ?idB))
   )
   :effect (and
      (at end (executed_instruction ?instruction_id))
      (at end (executed_operation ?operation_id))
      (at end (executed_binary_operation ?idA ?idB ?operation_id ?
          idC))
   )
)
```

Listing 4: Formalization of the binary operation action

Results

In this section, I will show a simple example of formalization. Regarding our objective, we hope that the temporal plan found by the planner shows the instructions that we can execute in parallel.

The example is represented by the source code shown in Listing 5. This simple source code sums two variables and set the result in the variable `c`. The variables `a` and `b` are not dependent on each other. Therefore, we must be able to execute the two assignment instructions in parallel, as shown in Figure 1.

```
int main()
{
   int a = 3;
   int b = 3;
   int c = a + b;
   return 0;
}
```

Listing 5: Source code

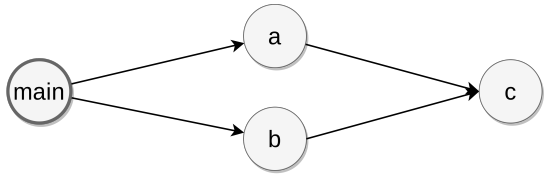

Figure 1: Dependency tree

The formalization of this problem is as shown in Listing 6.

```
(:init
   (executed_instruction id0)

   (assignment_id assignmentA id1)
   (assignment_id assignmentB id2)
   (operation_id sumAB id3)
   (assignment_id assignmentC id4)

   (dependency_tree id0 id1)
   (dependency_tree id0 id2)
   (dependency_tree id1 id3)
   (dependency_tree id2 id3)
   (dependency_tree id3 id4)
)

(:goal (and
   (executed_assignment assignmentA)
   (executed_assignment assignmentB)
   (executed_binary_operation assignmentA assignmentB sumAB
        assignmentC)
   (executed_assignment assignmentC)
))
```

Listing 6: Source code formalization

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

[3]The main instruction represents the starting point of the application.

0.000	1.000	2.000
assignmentA		
assignmentB		
	sumAB	
		assignmentC

Figure 2: STP parallel plan

The Figure 2 shows that, as soon as the program starts, we can execute the `assignmentA` and `assignmentB` instructions at the same time. To execute the `sumAB` instruction, we must 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, we would have a different dependency tree. Now, instead of line 10 in Listing 6, would 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.

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, we still have many questions:

- What is the best way to abstract a source code?
- How can we abstract nested loops?
- In a for loop, for example, when should we formalize the index (usually the variable `i`) assignment?
- Are the durative actions defined in this paper sufficient?
- Can we map the temporal plan back to the source code?
- How can we correctly choose the maximum number of parallel instructions in STP?
- Is STP the best temporal planner for this problem?
- Shouldn't we use different duration times for each assignment and binary operation?
- Do all regions need to run in parallel?
- How should `if` instructions be handled?

Despite all these question, we have a guarantee: this approach can identify parallel regions. Beyond that, it is important to stand out that the possibility to abstract the problem can make the formalization easier and the search time lower.

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.; Sánchez, L. M.; García, 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* 26:191–246.