An Automatic Parallelizing Model for Sequential Code Using Python

Afif J. Almghawish[*], Ayman M. Abdalla, Ahmad B. Marzouq
Faculty of Science & I.T, Al-Zaytoonah University of Jordan,
Amman 11733, Jordan

Abstract— This paper presents a new model for auto parallelization of some given source code without the need of code modification or additional information from the programmer. The proposed model, call it PyParallelize, is based on dynamic AST manipulation and refinement. This AST implementation is efficient for automatic parallel conversion and we use it to present a new model that provides better accuracy results. The PyParallelize model is implemented using the Python programming language to utilize its rich libraries. In addition to parallelizing sequential code, it is include a method for detecting recursive functions. The performance and accuracy of the proposed model were tested using a parallel computer, and the results were better than benchmark models.

Keywords— AST, Pattern Matching, Automatic Parallelization, Loop Parallelization, Python

I. INTRODUCTION

Parallelization problems have been actual research-subjects for many researchers due to their benefit to many real-world applications, so a lot of efforts should be devoted to improve existing methods of parallelization and to develop new ones.

Many approaches based on automatic parallelization of sequential applications simplify writing parallel programs for sequential computers. An automatic parallelization compiler for multicore systems that handles the coarse-grain syntactical variations in the source code was introduced in [1, 2]. Pydron, which is a system to parallelize and execute sequential Python code on a cloud, cluster, or multi-core infrastructure, was presented by [3]. They showed that Pydron uses semi-automatic parallelization and can perform parallelization with an API of only two decorators.

The approaches that extract coarse-grained task-level parallelism combined with the extraction of *DoAll* loops from sequential applications written in FORTRAN were presented by [4]. A coarse-grained thread-level parallelization techniques for C and FORTRAN applications presented in [5] employed intraprocedural analysis to spawn threads spanning function boundaries. The presented framework can apply analysis and optimization techniques, such as scalar privatization, reduction recognition, array analyses, and cache optimizations. As a target platform, they chose an eight-core Digital Alpha Server 8400 that was considered high-performance architecture at the time of their research. A more current task-level parallelization approach was presented by [6]. Their approach was integrated in the MPSoC Application Programming Studio (MAPS) and performed a semi-automatic parallelization system in which the user can manually steer the granularity of the extracted parallelism. Analysis on Cetus and Par4all using sample programs was performed by [7], and they concluded that the tools do parallelization in an effective way for single loops but do not show effective results for nested loops.

The method in [8] focused on the runtime performance that showed the speedup of the resulting parallelized code based on virtual operations and data partitioning used to exploit parallelism structured around array primitives. A suite of compilation techniques to convert general C code into parallel code was presented in [9]. It was written in C and relied on stream filters to obtain performance gains. Experiments have shown that the compiler was able to generate efficient programs that competed with equivalent hand-coded applications. The compiler found parallelism by searching for map-reduce patterns in the source code. This method is a non-trivial step towards the main goal of allowing application developers to produce high-performance parallel code without having to worry about details such as race-conditions and thread synchronization.

In this paper, we present a new model based on analyzing the Abstract Syntax Trees (AST) to help automate the process of parallel conversion by providing information about the source code without the need of any code modification or additional information from the programmer. This model, called PyParallelize, successfully implements a novel automatic method for parallelizing source code using Python programming language. PyParallelize does not focus on data level parallelism but focuses on task level parallelism, and it parallelizes nested loops of two levels. Test results showed that PyParallelize is able to achieve parallel speed up and performance better than Pydron [3], PIPS [10] and Cetus [11].

II.   METHODOLOGY AND IMPLEMENTATION OF PYPARALLELIZE MODEL

The proposed model, PyParallelize, will be presented as a set of steps to be performed where we will implement this methodology by linking each step with its implementation concurrently. This research work discusses the methods and analysis of PyParallelize taken to achieve a parallelized source code. PyParallelize is focused on dynamic AST manipulation and refinement. The steps taken by PyParallelize include AST generation, pattern generation, pattern matching, pattern rules, pattern analysis, and parallel conversion. These steps, and the libraries used by PyParallelize, are described in the following subsections.

A. Python Libraries Used by PyParallelize

Python libraries aid the construction of PyParallelize by providing a simple backend without losing performance or accuracy. The used Python libraries are the *graphviz* library, the *lib2to3* library, and the *batchOpenMPI*.

B. AST Manipulation and Refinement

PyParallelize operates by translating Python into an intermediate AST. The tree is then converted into a Pattern and evaluated by Pattern Matching. First, we take the source code and generate its AST using the *graphviz* library. As an example, consider the simple source code shown in Fig. 1.

```
for i in range(0,5):
        print (i)
```

Fig. 1. Simple source code

The generated AST for this example is shown in Fig. 2. By looking into Fig. 2, the structure of the source code can be determined and analyzed in the subsequent steps.
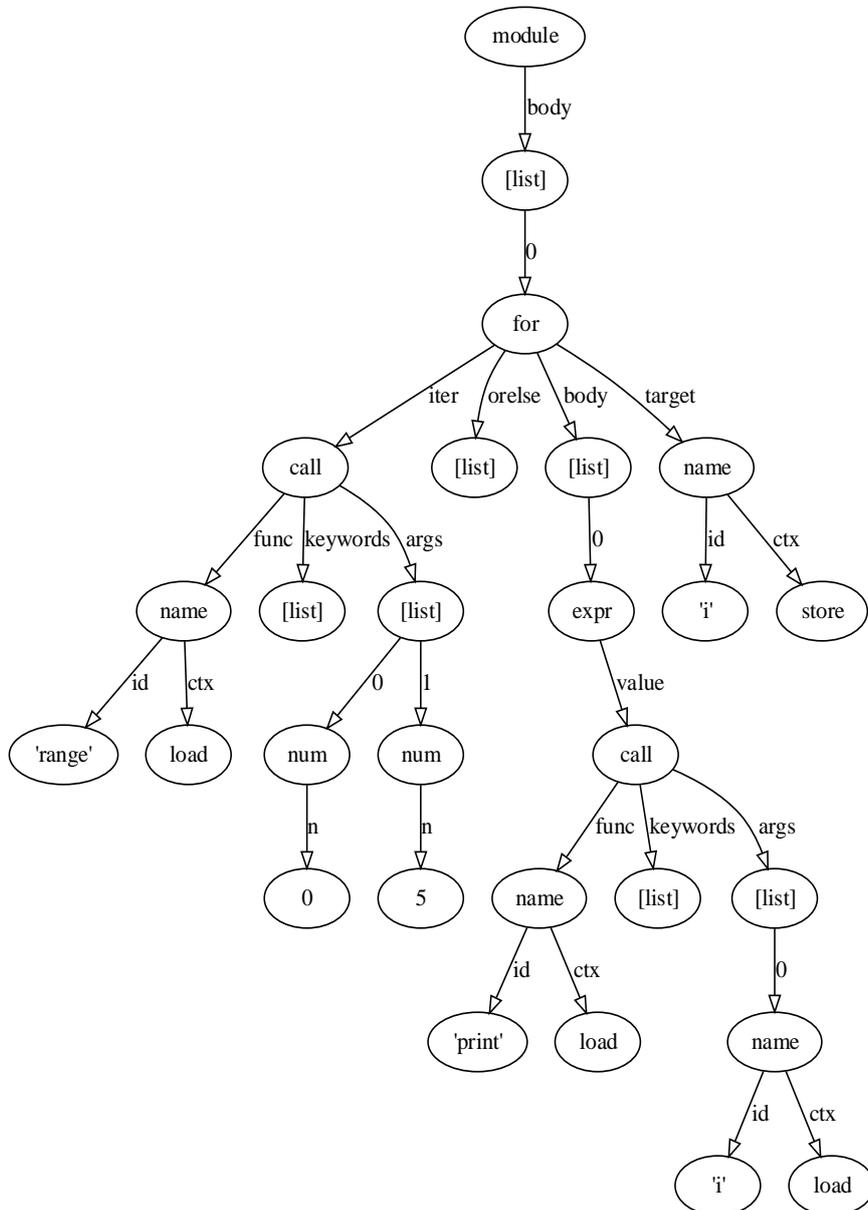
Fig. 2. AST example

C. Pattern Generation

The analysis of the AST and the generation of its pattern are done using the Python *lib2to3.pytree* library, so the obtained generated pattern for "for-loop" will be as shown in Fig. 3.

```
for_stmt< 'for' 'i' 'in' power< 'range' trailer< '(' arglist< '0' ',' '5' > ')'> > ':' simple_stmt<
print_stmt< 'print' atom< '(' 'i' ')' > > '\n' > >
```

Fig. 2. Generated for-loop pattern

The generated pattern requires further analysis to be applied to source code as a general rule. The generalization of the pattern will be described in the pattern analysis step.

D. Pattern Matching

This step is done to match the generated pattern with Pattern Rules based on the Python *lib2to3.fixer* library, as shown in Fig. 4.

```
def transform(self, node, results):
        name = results["name"]
        name.replace(Name("i_changed", prefix=name.get_prefix()))
```

Fig. 3. Transform function definition

Using the transform function definition shown in Fig. 4, the source code is transformed as shown in Fig. 5.

```
for i_changed in range(0,5):
        print (i_changed)
```

Fig. 4. Transformed source code

E. Pattern Rules

Pattern rules give PyParallelize a method of detecting occurrences of loops and function. The accuracy of PyParallelize is affected by the quality of the pattern rules taken after pattern analysis. In this step, the pattern rules for loops, functions and recursion are as discussed below, where their definitions are shown in Fig. 6, 7 and 8, respectively.

i. *Loops.* Check to ensure that there is no assignment or arithmetic operation on variables (e.g., $a=1+1$). If there is such a condition, exclude the loop form PyParallelize.

```
for_stmt< 'for' old_stmt=(any) any* > any*
|
not(arith_expr< any* >| expr_stmt)
```

Fig. 5. Pattern rule for for loop

ii. *Functions.* If there is a call of another function or passing of modified variables, exclude the function.

```
funcdef< 'def' old=' parameters< '(' 'str' ')' any* > any* > any*
|
not (expr_stmt<any*>)
```

Fig. 6. Pattern rule for function

iii. *Recursion.* Check for recursion by checking the return value of the function. If the return value is a call for the same function, mark the function as a variable for parallelization.

These present a semi-generalized form of patterns rules. The details of the rules are described further in the Pattern analysis step.

```
funcdef< 'def' old_stmt=(any) parameters<  any* >  > any*
|
not (expr_stmt<any*>)
|
return_stmt func=(any) >
|
old_stmt==func
```

Fig. 7. Pattern rule for function with recursion

F. Pattern Analysis

To generalize PyParallelize for working with different source codes, follow the *lib2to3* library Pattern grammar generation rules. Python grammar is based on a modified version of Extended Backus–Naur Form (EBNF) grammar. Full Grammar Specification of Python is available online [12].

```
# A grammar to describe tree matching patterns.
#
# - 'TOKEN' stands for any token (leaf node)
# - 'any' stands for any node (leaf or interior)
# With 'any' we can still specify the sub-structure.
# The start symbol is 'Matcher'.
Matcher: Alternatives ENDMARKER
Alternatives: Alternative ('|' Alternative)*
Alternative: (Unit | NegatedUnit)+
Unit: [NAME '='] ( STRING [Repeater]
            | NAME [Details] [Repeater]
            | '(' Alternatives ')' [Repeater]
            | '[' Alternatives ']' )
NegatedUnit: 'not' (STRING | NAME [Details] | '(' Alternatives ')')
Repeater: '*' | '+' | '{' NUMBER [',' NUMBER] '}'
Details: '<' Alternatives '>'
```

Fig. 8. *lib2to3* Pattern Grammar

Fig. 9 shows the general rule for patterns. The main terms in these rules are described below.

Matcher: Alternatives ENDMARKER

Matcher is the Python grammar symbol (e.g., for_stmt< , while_stmt<, funcdef<).

Alternatives are enclosed by parentheses (e.g, '(')').

Details are the descriptions of the arguments inside the grammar symbol.

ENDMARKER is the end marker of pattern ('>').

By using the information described in the *lib2to3* grammar pattern, the general rules for the generation of pattern rules are established.

1) Synchronization points: A synchronization point is where the source code needs to be executed sequentially [3]. PyParallelize cannot skip around synchronization points. Every operation that comes before the synchronization point must finish while all operations that come after must wait. A single synchronization point inside a loop would force the iterations to run sequentially; one after the other. This makes parallelization impossible. The main operations that cause a synchronization point are calls of system built-in functions and operations that modify an object.

2) Generalization analysis: The patterns taken from the pattern generation procedure are a specific form of a pattern for the specific source code. These cannot be used as a general form of patterns. Thus, the patterns need further analysis, which is described as follows.

i. *Variables.* The AST used by PyParallelize is directional, acyclic, and bipartite. There are two types of nodes: *Value-nodes*, which represent immutable data, and *Task-nodes*, which represent operations on data. In general, variables become value-nodes. Expressions and statements become tasks. Intermediate values in expressions are also represented by value-nodes. In PyParallelize, dependencies between tasks can only result from data dependencies. Data dependency can be handled by marking the assignment of the variable with the last read flag as a rule that can ensure synchronization of the variable.

Variables can be reassigned. This conflicts with the property that value-nodes represent immutable data. Hence, a one-to-one relationship between variables and value-nodes is not possible. We need to account for that by converting it to a static single assignment form [13] as shown in Fig. 10.

```
x=1                     x1=1
x=2      ——————>        x2=2
y=x      Convert        y=x2
```

Fig. 9. Static single assignment form

ii. *Loops.* The loop body in Python can be used as a modifier or a reader of variables combined with a conditional statement or without. In addition, function calls may cause a synchronization point. To address these problems, the complete loop construct is initially translated into a single task and marked. Then, at execution time, the loop is evaluated to meet the limitations for parallelization.

iii. *Functions.* Functions are first class objects in Python. When a function is defined, it is represented in the AST by a value-node. This value node is an input to a call-task, together with the arguments of the function.

In general, at conversion time, it is not known what the function can do. It is assumed that the function may have side effects or modify arguments passed to it in-place. This leads to additional edges connected to the task.

iv. *Conditional statements.* A conditional statement inside a loop or a function body adds ambiguity to variables if it is assigned in only one section. For such situations, a special task is added to the AST, which produces an undefined value as a result. The model is aware of such value nodes with an undefined content. It will produce the same exceptions in an attempt to use the value node as Python would when using an undefined variable.

v. *return, break, and continue statements.* These statements interrupt the regular control flow. This can be handled by converting the statement into a conditional statement and a flag variable in the AST. Once such a flag has been set, all code afterward is put into a condition checking the flag. Since the interrupting statement might be inside nested if statements, multiple conditions may be introduced. The task of the loop is aware of the flags and uses them to decide whether to replace the task by the body sub-graph or to end the loop, with or without a final replacement with the else sub-graph. In the case of the return statement, the return value is stored together with the flag. A conversion of for loop with break statement is shown in Fig. 11.

```
for i in items :
        #some code
        if i == end:
                break
        #some code
                Convert

⇩

for i in items until break_flag:
        #some code
        if i == end:
                break_flag=True
        if not break_flag:
                #some code
```

Fig. 10. for loop with break statement conversion

Consequently, after factoring all of the above in the analysis, the final pattern to be used with the AST is defined.

G. Synchronization Points Removal

The nature of Python prevents making strong guarantees from the source code alone. This forces the AST to have many synchronization points. The most common cause is call expressions that are not known at conversion time. Once the source code is converted, PyParallelize can check for flags and determine where the source code can or cannot be parallelized.

H. Parallel Conversion

After pattern matching, the conversion can be started by wrapping the code for parallelization in a function call, as seen in Fig. 12.

```
@parallel
def func():
        #do something
        return #some value
```

Fig. 11. Wrapped parallelized function definition

Python language does not support passing loops as objects, so loops are wrapped with a private function definition, as seen in Fig. 13.

```
@parallel
def loop()
    for i in range(10):
        #somecode
    return
```

Fig. 12. A loop wrapped in a parallelized function

After that, the call for methods from the *batchOpenMPI* library for loops and other types is done, as seen in Fig. 14, and the fully converted parallel source code is achieved.

```
f = batchOpenMPI.batchFunction(loop) #creating function wrapper
batchOpenMPI.begin_MPI_loop()
f.addtoBatch(4)
batchOpenMPI.processBatch()
batchOpenMPI.end_MPI_loop(print_stats=True)
```

Fig. 13. Call methods used to parallelize the source code

III. TESTING AND RESULTS

The testing process of the proposed methodology for the PyParallelize model was done using an Intel Xeon E5-2666 v3 processor with 56 cores running Red Hat Enterprise RHEL 7 64-bit version. Measurement of accuracy was done

for sequential source code and for converted parallel source code. The converted parallel source code was executed by varying numbers of cores such as 2, 4, and 8.

To measure the accuracy of PyParallelize, the parallel code produced by PyParallelize is analyzed and compared to hand-converted parallel code. Alternatively, the source code may be considered successfully converted if the output of the converted parallel code matches the output of the original sequential code. Table 1 shows various source codes with different criteria that could challenge PyParallelize. As shown in Table I, PyParallelize was able to successfully convert most of the given source codes but failed to convert nested loops with three or more levels due to the extra complexity of such loops.

Table I Accuracy Rates of PyParallelize

Type of source code	Successful conversion	Successful output
Simple source code	100%	100%
With simple loops	99%	99%
With 2-level nested loops	99%	99%
With 3-level nested loops	60%	65%
With simple functions	100%	100%
With recursive function	98%	98%

The execution time for calculation of matrix multiplication, $c[n][n] += (a[n][n] * b[n][n])$, was computed for a large matrix size of array size of 70000 elements, and then the comparison of PyParallelize with Pydron, PIPS, and Cetus was done. The results of this comparison are shown in Fig. 15, which shows that PyParallelize was faster than these other methods.
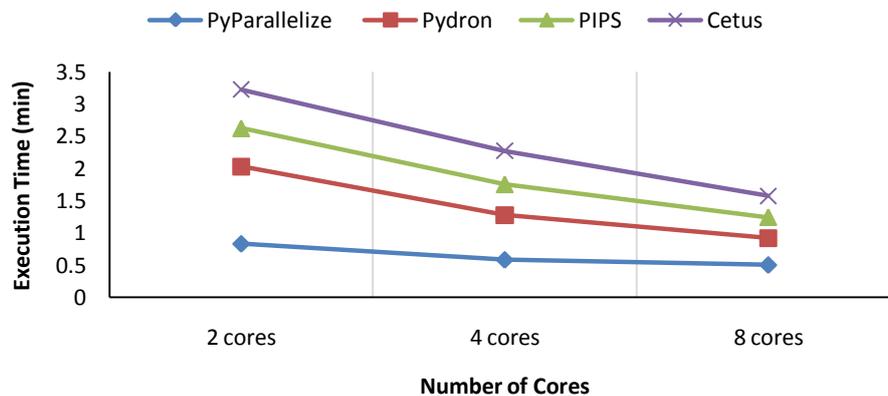
Fig. 15 Execution times (minutes) of matrix manipulations of array size of 70000 using PyParallelize, Pydron, PIPS, and Cetus

IV. CONCLUSIONS

A new parallelization model, named PyParallelize, was introduced, and it included a method for detecting recursive functions. It automates the process of parallel conversion by providing information about the source code without the need of any code modification or additional information from the programmer. The results obtained by implementing and testing PyParallelize showed that it was sufficiently efficient for automatic parallel conversion and it gave results with a relatively high rate of accuracy. However, this proposed model does not work efficiently when the number of nested loops has more than two levels, so this might be a good research topic for future work.

REFERENCES

[1]   J. M. Andión, M. Arenaz, G. Rodríguez and J. Touriño, "A Novel Compiler Support for Automatic Parallelization on Multicore Systems," *Parallel Computing*, vol. 39, no. 9, pp. 442–460, 2013.

[2]   J. M. Andión, M. Arenaz, G. Rodríguez and J. Touriño, "A Parallelizing Compiler for Multicore Systems," in *Proc. 17th Int'l Workshop Software & Compilers Embedded Systems*, pp. 138–141, 2014.

[3]   S. C. Muller, G. Alonso and A. Csillaghy, "Scaling Astroinformatics: Python + Automatic Parallelization," *IEEE Computer*, vol. 47, no. 9, pp. 41-47, 2014.

[4]   V. Sarkar, "Automatic Partitioning of a Program Dependence Graphinto Parallel Tasks," *IBM J. Research & Development*, vol. 53, no. 5-6, pp. 779–804, 1991.

[5]   M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Trans. Computers*, vol. 29, no. 12, pp. 84–89, 1996.

[6]   J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid and H. Kunieda, "MAPS: An Integrated Framework for MPSoC Application Parallelization," in *Proc. Design Automation Conf.*, pp. 754–759, 2008.

[7]   S. Prema and R Jehadeesan, "Analysis of Parallelization Techniques and Tools," *Int'l J. Information Computation Tech.*, vol. 3, no. 5, pp. 471-478, 2013.

[8]   W.-M. Ching and D. Zheng, "Automatic Parallelization of Array-oriented Programs for a Multi-core Machine," *Int'l J. Parallel Prog.*, 40, pp. 514–531, 2012.

[9]   L. L. Mata, F. M. Pereira and R. Ferreira, "Automatic Parallelization of Canonical Loops," *Science Computer Programming*, vol. 78, no. 8, pp. 1193–1206, 2013.

[10]   F. Irigoin, P. Jouvelot and R. Triolet, "Semantical Interprocedural Parallelization: An Overview of the PIPS Project," in *Proc. Int'l Conf. Supercomputing*, pp. 244–251, 1991.

[11]   S.-I. Lee, T. A. Johnson and R. Eigenmann. "Cetus–An Extensible Compiler Infrastructure for Source-to-Source Transformation," *Languages & Compilers Parallel Computing*, pp. 539–553, 2004.

[12]   Python Software Foundation, "Full Grammar Specification of Python," The Python Language Reference. [Online]. Available: https://docs.Python.org/3/reference/grammar.html.

[13]   K. Cooper and L. Torczon, *Engineering a compiler*. San Francisco: Morgan Kaufmann, 2012.