Teaching Parallel Computing and Dependence Analysis with Python

Neftali Watkinson, Aniket Shivam, Alexandru Nicolau and Alexander V. Veidenbaum
Department of Computer Science, University of California, Irvine
Irvine, California
watkinso@uci.edu, aniketsh@uci.edu, nicolau@ics.uci.edu, alexv@ics.uci.edu

***Abstract*—Languages with a high level of abstraction, such as Python, are becoming popular among programmers and are being adopted as the primary programming language in pedagogy. A potential drawback of using such languages is that the architectural aspects, such as data layout in memory, get completely hidden. Therefore, the students have difficulty in understanding advanced computer science topics such as Parallel Computing. Computer architectures have evolved to allow multiple levels of parallelism. From mobile devices to supercomputers, a lot of tasks are done in parallel. Parallel Programming models have become ubiquitous and computer science graduates should know how to take advantage of those models. Therefore, it becomes necessary to expose students to the concepts of parallel programming early in the curriculum. This work describes a lesson plan for teaching Parallel Computing, using Data Dependence analysis and Loop transformations, to Python Programming students. We analyze our teaching experience, evaluation of students' understanding and likelihood of using parallel programming in introductory courses in the future.**

I. Introduction

Modern architectures have evolved towards greater number of cores on a chip, as well as, improving the processing capabilities of individual cores. Each core in the current multi-core architectures includes the capability to process Single Instruction Multiple Data (SIMD) or Vector instructions. In addition to multi-core architectures, popular general purpose accelerators such as GPUs and recently deep learning accelerators such as Google's TPU use the concepts of parallel computing to accelerate applications. The concepts of parallel computing are critical to improving performance for various domains of applications such as scientific applications, game engines, big data applications, deep neural networks, etc. Thus, from computer science students' perspective, the understanding of the concepts of parallel computing become very important in order to perform tasks in the above mentioned industries.

In a lot of applications, for example, in either scientific or neural networks, most of the execution time is spent in small sections of code that are sometimes called *kernels*. These kernels perform repetitive operations on various data points and, in most programming languages, are written as loop nests. Therefore, improving performance of the loop nests lead to better performance for the entire application. For last several decades, a lot of effort from the research community in the field of compilers [10], [3], [6] have gone into improving performance of serial code as well as exposing parallelism in the loop nests. One critical part of exposing parallelism in the loop nests is the analysis of data dependence [14], [3]. Data dependence allow compilers to model the relation between the operations on data points and check validity of operations that can be performed concurrently. For a programmers' perspective, these data dependencies can be analyzed while writing code. Programmers can provide hints for the compilers to produce parallel code. Several interfaces (APIs) and programming models, such as OpenMP [9], OpenACC [8], Pthreads, C++ threads, MPI [5], etc., are available for programmers to direct compilers towards generating parallel code for various target architectures and type of systems. Most widely used programming languages for writing parallel, high-performance code are `Fortran`, `C` and `C++`.

On the other hand, Python has become a popular language for fast prototyping and it's widely regarded as an "easy-to-learn" language. It can be attributed to its scripting nature that allows for quick fixes on the run. The popularity of Python has percolated into universities and has become a go to language for Computer Science (CS) and non-CS majors. At the University of California, Irvine (UCI), students need to take 3 quarters of Python programming (ICS 31, 32 and 33) before being able to take `C` as a secondary language (ICS 45C). Usually, university students are taught `C` or `C++` before being introduced to the concept of parallel computing [1]. UCI does enforce ICS 45C as a prerequisite for elective Parallel Computing classes. As shown in a recent study [2], Python users struggled more than `C++` users when dealing with basic programming questions.

The main drawback of limiting access to parallel computing after the student has studied two programming language, is that for the first two years of studies the student has only been introduced to serial programming. There's a recognized need to introduce parallel programming early on [11], [12]. Efforts in using Python for teaching parallel computing will allow for the student to, first, being able to use libraries that can generate parallel code and, second, to understand the required code transformations and coding skills to manually improve the performance [4], [15].

This work describes a lesson plan to teach parallel computing in a Python programming setting. We use data dependence analysis and loop transformations for teaching the details of parallelism. Then we analyze an experience at UCI where students of ICS 33 gained hands on experience with Python

modules for parallel computing. We analyze the results and compare them to a previous experience with C and C++ students [13]. We also administer a survey that sets the baseline for students' understanding in parallel computing.

The rest of the paper is organized as follows. In Section II, we discuss our pedagogical approach for teaching parallel computing to student with background in Python. Section III presents the performance of students on the assignments and quizzes. Sections IV discusses our observations for the pedagogical approach and students' performance. Section V describes the future work in this area. We conclude the paper with Section VI.

II. Methodology

Our lesson plan consists of three sessions: 2 lectures to introduce the concepts and describe the tools with examples. Plus one practical session where students get hands on experience with the provided examples. Afterwards they are given a one week home assignment where they will try to apply parallelism to 5 different loop nests that cover different aspects of parallelism and dependence analysis. The lecture notes are written using Jupyter Notebooks, which allows for real code demonstrations as we go through the material. The notes are handed out to the students as well.

A. Lecture 1: Introduction to the Concepts of Parallel Computing and Tools

We start the lecture by reviewing `Numpy`, a python package for scientific computing. Numpy is essential for working with the home assignment and this shouldn't be a new information for the students since it's reviewed during ICS 32. It's libraries are precompiled in Python C, therefore providing further opportunities for optimization. We also show them how to gain immediate improvement by using optimized versions of Python such as the Intel Python Distribution (IDP) which, among other thinks, takes advantage of SIMD processing present in the modern Intel processors.

During the same lecture, we introduce students to an overview of parallel computing, including concurrency and vector instructions (SIMD). This is followed by an explanation of Python's Global Interpreter Lock (GIL) which blocks automatic parallel execution and multi threading of python code. Students then learn about the threading and multiprocessing libraries that allows for manual parallelization of code. For multi-core architectures, we teach them that multiprocessing is the best available option for exposing local parallelism, since it allows the code to run on multiple cores. Because of the GIL, multithreading doesn't gain any performance improvement.

B. Lecture 2: Dependence Analysis, Loop Transformations and Writing Parallel Code in Python

In the second lecture, we begin by teaching students about data dependences [14], [3] between statements. We concentrate on three types of data dependence relations. First, data flow-dependence (also referred to as true dependence or RAW for read after write), where, as per serial execution, the data is used by a statement after the same data was computed by another statement or the statement in a different loop iteration. Second, data anti flow-dependence (or WAR for write after read), where the data is changed by a statement before being read by another statement. Third, output dependence (or WAW for write after write) where the data is changed by a statement after being computed by another statement. We also explain read after read (RAR) dependence, but note that it rarely hinders parallelism. We explained these data dependences using the theoretical explanation followed by simple examples.

Next, we teach students about some loop transformation techniques [10], [6] that are important for exposing parallelism in the loop nests. We focus on three important loop transformations, i.e., loop distribution or loop fission, loop interchange or loop permutation, and loop chunking or iteration grouping. In loop chunking, a parallelizable loop is divided into sub-parts so as divide workload equally among available threads. For example, for a dividing a loop with N iterations to be processed by 4 threads, the first chunk of iterations (0 to N/4) is giving to thread 1, second chunk of iterations (N/4 to N/2) is given to thread 2, third chunk of iterations (N/2 to 3N/4) is given to thread 3 and the last chunk of iterations (3N/4 to N) is given to thread 4. In loop distribution, a loop with two or more statements is separated into two or more loops each with at least a statement. Just like any other loop transformation, presence of data dependences decides the legality of distribution. This transformation is important because it allows for parts of loop (set of statements) to be parallelized, while the rest of the loop is restricted to be executed serially. In loop permutation, we change the order of loops in nested loops. This transformation allows for putting the loop that can be parallelized to be moved to be the outer-most loop. Moving the parallelizable loop as the outer-most loop allows for more workload for individual threads and also reduces the burden of repetitive thread creations/joining inside a loop. We demonstrate these transformations with examples and demonstrate an easily parallelizable function that computes the Sobel image filter for edge detection.

For teaching parallelism in Python, we used the `multiprocessing` package. Implementing it is not trivial. To use this package to parallelize loops, the loops need to be outlined to a function where function parameters are the loop bounds. Then for creating a thread, the `Process` function is called from the `multiprocessing` package. The arguments are the function containing the loop body followed by loop bounds. This resembles similarity to parallelizing loops in `C` using `pthreads`. For data structures, usually arrays, that are shared (read or written) by multiple threads, in Python users need to create new arrays using the `Array` function from the `multiprocessing` package. These arrays are visible to the threads as shared memory. Therefore, users are responsible for creating these new arrays and copying their existing Python lists or Numpy arrays into these shared memory arrays. In `C` for example, the arrays can be passed by reference and can be shared by multiple threads without any need for a new array allocation or declaration.

C. Practical Session

For our third session, the students gain hands on experience doing loop transformations in the computer lab. We provide them with examples for each transformation as well as an easy to follow template for implementing multiprocessing. We ask them to apply each transformation to a loop nest and try to run multiprocessing to see if they gain any performance improvement. This is the only opportunity for troubleshooting installation problems or misconceptions they may have about loop transformations. The goal of this session is to have students successfully transform and run our examples with and without multiprocessing. We use Python's time module to measure execution times.

D. Assignment on Parallelizing Loop Nests

Finally, the students have a take home assignment. We give them 5 loop nests that they have to analyze, transform, if necessary, and perform `chunking`. The students report their findings using an online quiz. Each loop nest has different data dependence patterns and students where challenged to identify them and expose parallelism using loop transformations. Out of the 5 loop nests, one is embarrassingly parallel (requires no transformation), one can't be parallelized due to data dependence and the rest of them can only be parallelized after performing transformations. The loop nests use a similar template as the one used for the lab session, so the students only need to write the parallel version of the loop nests and perform `chunking`.

1) Loop 1: The first loop contains a RAR and a RAW dependence between lines 4 and 5. However, since each line is working on a different row of the matrix, the dependence will not hinder parallelism using vectors up to size N. Loop distribution could further relax the dependence.

Listing 1. Loop 1

```
1 def loop1_serial():
2   for i in range(1,N-1):
3     for j in range(1,N-1):
4         A[i][j] = C[i][j] * D[i][j]
5         B[i][j] = A[i-1][j-1]
6                   * C[i][j]
7   return
```

2) Loop 2: The second loop contains a similar dependence as Loop 1, however in this case the dependence is tighter (A[i][j] is read right after is written). While parallelism shouldn't be an issue either, distribution can also help.

Listing 2. Loop 2

```
1 def loop2_serial():
2     for i in range(1,N-1):
3       for j in range(1,N-1):
4           A[i][j] = C[i][j] * D[i][j]
5           B[i][j] = A[i][j] * C[i][j]
6                     * D[i][j]
7
8     return
```

3) Loop 3: In this case there is WAR and RAW dependence in line 4, and RAW dependence between line 4 and 6. This loops requires interchange and distribution in order to be parallelized.

Listing 3. Loop 3

```
1 def loop3_serial():
2   for i in range(1,N-1):
3     for j in range(1,N-1):
4         D[i][j] = (D[i-1][j]
5                    + D[i][j]
6                    + D[i+1][j])/3
7         A[i][j] = D[i-1][j-1] * 5
8   return
```

4) Loop 4: The fourth loop cannot be manually parallelized with the transformations we taught because line 4 has all 4 kind of dependencies towards both dimensions of the matrix.

Listing 4. Loop 4

```
1 def loop4_serial():
2   for i in range(1,N-1):
3     for j in range(1,N-1):
4         D[i][j] = (D[i-1][j]
5                    + D[i][j-1]
6                    + D[i][j]
7                    + D[i][j+1]
8                    + D[i+1][j])/5
9   return
```

5) Loop 5: The last loop doesn't need any transformations. The statement at line 5 has a WAR dependence with itself but this loop can be applied towards multiprocessing right away.

Listing 5. Loop 5

```
1 def loop5_serial():
2   for i in range(N):
3     for j in range(N):
4       for k in range(N):
5         A[i][j] = A[i][j] + C[i][k]
6                   * D[k][j]
7   return
```

6) Expected Parallel Code for Loop 1: The metrics on which we evaluated the solutions from the students was in two parts. First, did they figure out the dependence between the computations and transform the loops to expose parallelism. Second, did they correct use `multiprocessing` package to create threads and distribute workload equally.

The expected solution for the transformation of Loop 1 is shown in listing 6. The correct way to parallelize the loop and create two threads is shown in listing 7.

Listing 6. Solution for Loop 1 (Loop Transformation)

```
def loop1(start,end):
    for i in range(start,end):
        for j in range(1,N-1):
            A[i][j] = C[i][j]
                    * D[i][j]
  return
def loop2(start,end):
    for i in range(start,end):
        for j in range(1,N-1):
            B[i][j] = A[i-1][j-1]
                    * C[i][j]
  return
```

Listing 7. Solution for Loop 1 (Multi-Process Creation)

```
p1 = multiprocessing.Process(
      target=loop1,
      args=(1,int(N/2)))
p2 = multiprocessing.Process(
      target=loop1,
      args=(int(N/2),N))
p1.start(); p2.start()
p1.join(); p2.join()
p1 = multiprocessing.Process(
      target=loop2,
      args=(1,int(N/2)))
p2 = multiprocessing.Process(
      target=loop2,
      args=(int(N/2),N))
p1.start(); p2.start()
p1.join(); p2.join()
```

III. Results

We tested our approach with 42 students from ICS 33. They were promised full credit on their lowest lab score (they have weekly lab assignments). We held theory sessions on Tuesday and Thursday, gave them the assignment and held a practice session in the computer lab on Monday of the following week. The assignment along with the evaluation quiz was due the following Friday.

During the lectures, students engaged in questions about parallelism. They were able to follow along as we demonstrated the coding examples. In the lab, all of the students were able to run IDP and to setup the environment for multiprocessing with the sample code we gave them for testing. However, when they started working on their own assignments we started receiving feedback on how they were having trouble running multiprocessing on their own code. We tried to troubleshoot as much as we could before the session ended and made sure that they could run the examples before they left. We also gave them further reference materials in case they needed more examples for applying multiprocessing. Out of the 40 students, only 23 submitted their code and quiz responses.

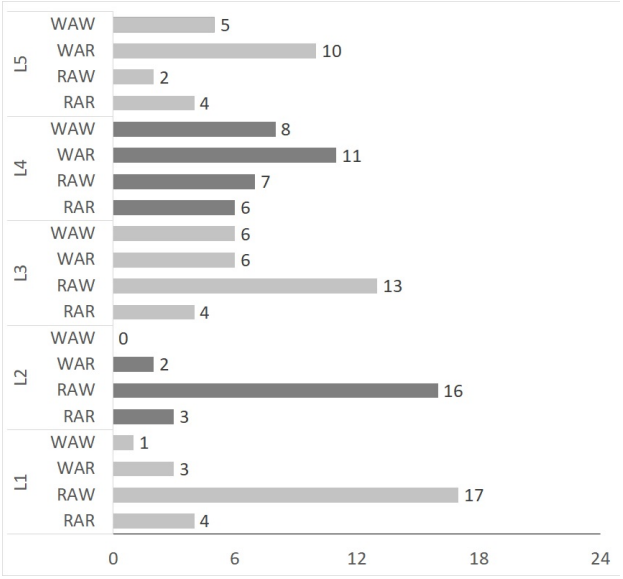

Fig. 1. Dependences Identified by Students per Loop

A. Dependence Quiz

For the quiz, students could choose multiple options for every loop. For loop 1, 77.3% of the students identified the RAW dependence and 82.6% used distribution as their main transformation. One of the students didn't do any transformation noticing that the dependence is only true for threads bigger than N-1 and instead parallelized it directly managing the size of the data each process would handle.

Loop 2 had similar results with 84.2% realizing the presence of the RAW dependence and 90.9% using distribution. In this case they noted a big performance difference between serial and parallel, with some exceptions by students who noted they were running on single core or dual core environments including one student who was running on a virtual machine.

For the third loop, while 68.4% of the students found the RAW dependence, only half of them (31.6% overall) noted the WAR and WAW dependence. This last group opted for loop permutation which is the right transformation for this case. The rest of them opted for distribution. Most of the students reported serial or serial with IDP as being the fastest which makes sense if they weren't able to deal with the dependencies.

For the fourth loop, 40% of the students found all four dependencies. 76.6% tried loop permutation and 28% tried distribution. While all of them noted serial to be faster, only 20% mentioned that the loop could not be run in parallel due to dependence.

For the last loop, 55.6% found the WAR dependence and 60% did loop permutation. 34% of the students reported a successful run in parallel using multiprocessing. Figure 1 shows the dependences reported per loop.

B. Code Results

When looking at the code that each student wrote, we identified some common mistakes. For example, many students

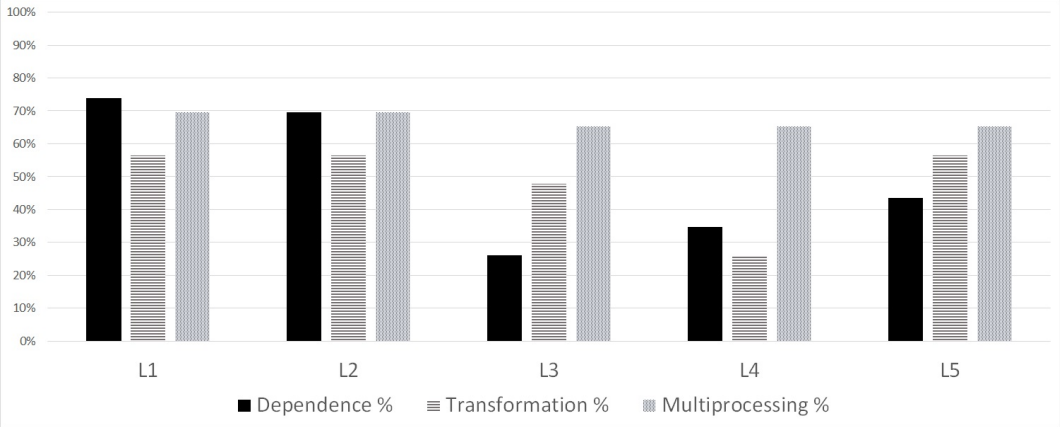

Fig. 2. Distribution of Correct Answers and Implementation of Transformations

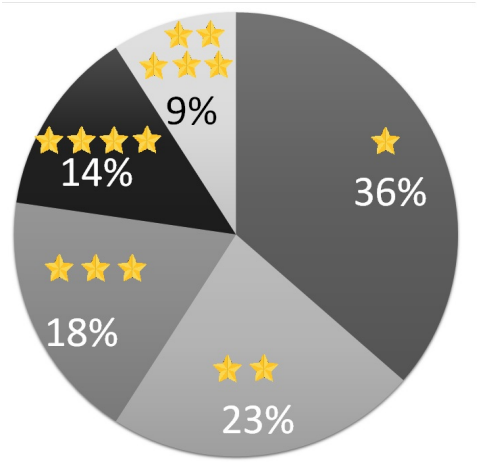

Fig. 3. Ranking of how likely the students will try to do manual parallelism on their own

would try loop distribution first. This would work for loop 1 and loop 2 but loop 3 requires loop interchange first. The other mistake is not transforming loop 4 but still try to run it using multiprocessing. Figure 2 shows the percentages per loop of how many students identified the right dependences, applied the right transformations, and where able to successfully do multiprocessing.

IV. Analysis

We asked the students to tell us, using a 5 star rating, how likely they will parallelize their code in the future in order to improve code performance. One star means not likely at all and 5 stars means very likely. Figure 3 shows their response. Most of the students responded with 1 and 2 stars. We believe this is because of how cumbersome the `multiprocessing` module is. From the very beginning of our planning process we knew that using Python for parallel computing would represent a bigger challenge. There are third party libraries available for Python that allow for auto-parallelism of code, but they are not easier to use. According to [7], Python is not mature enough for teaching parallel computing and shouldn't be adopted as the primary programming language in computer science majors until its drawbacks are justified or resolved. Our experience compared to [13] does show that compared to students with similar background, where the main difference is in the programming language (C and C++), Python students struggled considerably more in implementing manual parallelism. Python requires learning the `multiprocessing` module which has a steep learning curve. Meanwhile, C and C++ compilers have the option to auto-vectorize or auto-parallelize code or use programming models such as OpenMP in a black-box manner. This allows users to only concentrate on exposing parallelism to the compilers. C and C++ students don't need any new programming skills to see the impact of parallel computing. Whereas, in Python using packages such as `multiprocessing` was our only feasible option to expose students to the concept of parallelism.

Listing 8. Loops used in student survey

```
1  Loop A:
2  for i in range(0,100):
3      A[i] = A[i] + A[i]
4  Loop B:
5  for i in range(0,100):
6      A[i-1] = A[i] + A[i]
7  Loop C:
8  for i in range(0,100):
9      A[i] = A[i-1] + A[i+1]
10 Loop D:
11 for i in range(0,100):
12     A[i] = A[i] + A[i+1]
```

We surveyed a separate group of 330 students with similar background. All but 8 of them are from Computer Science related majors, 92% have taken ICS 33 and 58% have experience with C or C++. We introduced a brief explanation of parallelism and defined data dependence as "when a computation has to wait for another computation to finish". Then we asked them to analyze 4 loops as shown in Listing 8, each one with one computation but with different types of dependences.

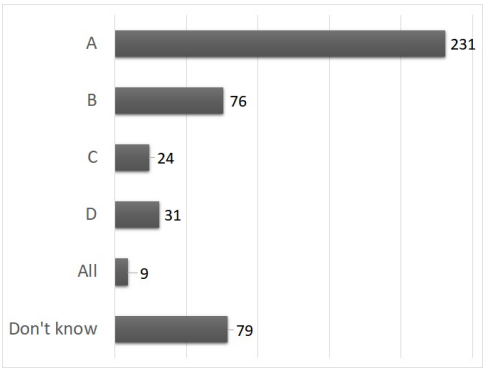

Fig. 4. Survey results for which loop is free of dependence

Only loop A is embarrassingly parallel and is dependence free according to the given description. 70% of the students identified Loop A as being dependence free, while up to 25% chose any of the other loops or that they didn't know the answer(Figure 4). When asked which loop they think would run the fastest in a parallel setting, 73% chose loop A, while 20% couldn't answer the question.

Finally we asked them to explain why they think that loop would run the fastest, only 45% pointed to the lack of dependence while the remaining 55% chose unrelated reasons (arithmetic operations being faster or because the loops has less iterations). In comparison, students that went through our plan had a better and more detailed understanding of data dependence and parallelism than students who didn't.

This work, along with the experience in [13], shows that students can handle selected topics in parallel computing in lower level courses. They are even able to apply these concepts in practice. In this experience, students also learned how to do workload distribution which added another level of complexity. With a better tool that allows students to concentrate on just the code analysis, we believe the results for taking benefits of parallel computing could be improved greatly.

V. Future Work

Analyzing the adequacy of Python as a main programming language for education is beyond the scope of this paper. However, we plan to circumvent the drawbacks of using Python for parallel computing by implementing libraries that the students can import into their projects. These will allow for automatic dependence analysis that they can use to verify their transformations, and for a more streamlined implementation of multiprocessing. This will allow instructors to focus on the high level aspects of parallel computing without having to delve too much on technical details, as we believe these are more appropriate for advanced courses.

VI. Conclusion

We described a lesson plan that involved theoretical and practical learning about parallel computing. Our pedagogical approach was applied to students with one year of Python programming experience. We taught students the basics of data dependence analysis and loop transformation techniques. Students analyzed, transformed and parallelized loop nests and reported on their findings. Our results show that students were able to grasp the theoretical knowledge and apply the concepts of parallel computing.

Our overall experience suggests that it is possible to teach concepts of parallel computing to intermediate Python students, even though the support in Python is limited and complex when compared to `C` or `C++`. Brief introductions to the Parallel Computing paradigm in early courses have the potential to ease and motivate the student towards taking advanced parallel computing classes.

References

[1] J. Adams, C. Nevison, and N. C. Schaller. Parallel computing to start the millennium. In *ACM SIGCSE Bulletin*, volume 32, pages 65–69. ACM, 2000.

[2] N. Alzahrani, F. Vahid, A. Edgcomb, K. Nguyen, and R. Lysecky. Python versus c++: An analysis of student struggle on small coding exercises in introductory programming courses. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 86–91, New York, NY, USA, 2018. ACM.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb 1993.

[4] V. Dolgopolovas, V. Dagienė, S. Minkevičius, and L. Sakalauskas. Python for scientific computing education: Modeling of queueing systems. *Scientific Programming*, 22(1):37–51, 2014.

[5] W. D. Gropp, W. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[6] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[7] A. Marowka. On parallel software engineering education using python. *Education and Information Technologies*, 23(1):357–372, 2018.

[8] The OpenACC application programming interface, version 2.6. https://www.openacc.org/, Nov. 2017.

[9] The OpenMP application programming interface, version 4.5. https://www.openmp.org/, Nov. 2015.

[10] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[11] S. K. Prasad, A. Y. Chtchelkanova, S. K. Das, F. Dehne, M. G. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, et al. Nsf/ieee-tcpp curriculum initiative on parallel and distributed computing: core topics for undergraduates. In *SIGCSE*, volume 11, pages 617–618, 2011.

[12] B. Rague. Teaching parallel thinking to the next generation of programmers. *Journal of Education, Informatics and Cybernetics*, 1(1):43–48, 2009.

[13] N. Watkinson, A. Shivam, Z. Chen, A. Veidenbaum, and A. Nicolau. Using data dependence analysis and loop transformations to teach vectorization. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1143–1148. IEEE, 2017.

[14] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, Apr 1987.

[15] G. Zaccone. *Python Parallel Programming Cookbook*. Packt Publishing Ltd, 2015.