OpenMP is Not as Easy as It Appears

Rogério Gonçalves*†, Marcos Amaris†, Thiago Okada†, Pedro Bruel† and Alfredo Goldman†
*Department of Computer Science – DACOM
Federal Technological University of Paraná – UTFPR
Campo Mourão/PR, Brazil
rogerioag@utfpr.edu.br
†Software Systems Laboratory – LSS
Institute of Mathematics and Statistics – IME
University of São Paulo – USP
São Paulo/SP, Brazil
{rag, amaris, thiagoko, phrb, gold}@ime.usp.br

***Abstract*—This paper aims to show that knowing the core concepts related to a given parallel architecture is necessary to write correct code, regardless of the parallel programming paradigm used. Programmers unaware of architecture concepts, such as beginners and students, often write parallel code that is slower than their sequential versions. It is also easy to write code that produces incorrect answers under specific conditions, which are hard to detect and correct.**

The increasing popularization of multi-core architectures motivates the implementation of parallel programming frameworks and tools, such as OpenMP, that aim to lower the difficulty of parallel programming. OpenMP uses compilation directives, or *pragmas*, to reduce the number of lines that the programmer needs to write. However, the programmer still has to know when and how to use each of these directives. The documentation and available tutorials for OpenMP give the idea that using compilation directives for parallel programming is easy. In this paper we show that this is not always the case by analysing a set of corrections of OpenMP programs made by students of a graduate course in Parallel and Distributed Computing, at University of São Paulo. Several incorrect examples of OpenMP *pragmas* were found in tutorials and official documents available in the Internet. The idea that OpenMP is easy to use can lead to superficial efforts in teaching fundamental parallel programming concepts. This can in its turn lead to code that does not develop the full potential of OpenMP, and could also crash inexplicably due to very specific and hard-to-detect conditions. Our main contribution is showing how important it is to teach core architecture and parallel programming concepts properly, even when you have powerful tools such as OpenMP available.

I. Introduction

There is an ongoing evolution of parallel processing platforms. The pursuit for high performance in general-purpose and scientific applications have turned to the exploration of the parallel processing capabilities provided by multi-core processors and arrangements of coprocessors like the Xeon Phi [1]. The machine currently at the top of the *TOP500* list [2] uses multi-core processors (12 cores) and Xeon Phi coprocessors [3]. Multi-core processors have become popular on personal computers and mobile devices. This popularization increases the need for specialized developers able to work in parallel applications and apply parallel computing models [4] [5].

Today there is a wide gap between legacy code applications that could benefit from parallel computing power and the new parallel computing platforms that could provide that power. However, legacy applications are generally sequential and therefore not easily scalable to multi-core architectures and parallel computing platforms.

Asanovic *et al.* [6] describes this scenario, and defends the need for Structural and Computational Patterns in Parallel Computing. These patterns allow the portability of parallel programming solutions, while also allowing specialization of applications with the same behavior. In some situations it is not a matter of achieving the highest possible performance, but instead of making the application achieve acceptable performance, benefiting from a new parallel computing platform [6]. A similar approach is proposed by Pllana *et al.* in a development environment that uses a combination of Model-Driven Development and Parallel Building Blocks [7]. The approach consists in creating the model and then generating or transforming the code using *model-to-source* tools.

In this context, the approach of using Compilation Directives to guide the compiler in the parallelization process has gained popularity. These directives are implemented using the compiler's pre-processing directives and are utilized as annotations that provide tips about the sequential code. Among the tools and extensions that use compilation directives are the `OpenMP` API [8] [9] used for writing programs for multi-core architectures, and the `OpenACC` programming standard [10], used for writing programs for heterogeneous CPU/GPU architectures. The `OpenMP 4.0` specification supports offloading capabilities [11] such as `OpenACC` and `AMD HSA` [12]. These tools aim to make easier the task of designing and implementing efficient and provably correct parallel algorithms. However, the abstraction layers built by the extensions also have the potential to conceal architectural details and to generate new parallel programming challenges.

When comparing `OpenMP` to other available tools for developing multi-threaded applications such as the `pthreads` interface, the API can be considered almost costless. However, when writing more complex applications this may not be true. The design and implementation of such complex applications require more knowledge about platforms and execution models than what can be shown by tutorials, that present high level and trivial examples. These tutorials try to convince the user that programming with directives is simple and easy to do, which is not always true. For instance, to achieve good performance

in some cases it is necessary to describe many restrictions on annotations [13] [14].

Krawezik and Cappello [15] evidenced this difficulty by comparing MPI and three OpenMP versions of benchmarks. The most popular use of `OpenMP` directives is at the loop level. The programmer discovers potentially parallel loops and annotates the code with directives starting and closing parallel regions. Good results were obtained only when SPMD was implemented on OpenMP, and achieving good performance required a significant programming effort [15].

In this work we have gathered data from assignments made by graduate students for the *Introduction to Parallel and Distributed Computing* course. Among other exercises, the students were asked to find examples of `OpenMP` code in tutorials and official manuals that failed under special conditions. These conditions could be created by, for example, placing barriers or forcing racing conditions by exposing inaccurate thread management.

We argue that it is necessary to consider fundamental concepts of architectures and parallel programming techniques when teaching students new to the subject, so that they can learn when and how to apply the programming models [16]. The abstractions provided by parallel programming tools over concepts such as threads, barriers and loop parallelization should be carefully studied, analysed and conceptually built in the classroom before they are used by the students in their assignments.

We found eight tutorials with different kinds of `OpenMP` errors in the experiment with the students' assignments. Many of these errors where similar to the most common `OpenMP` mistakes [17]. The most common type of error identified in our experiment was the management of shared or private variables. This problem is common across multiple programming paradigms.

The **main contribution of this work** is to show the importance of teaching core architecture and parallel programming concepts thoroughly, despite the availability of powerful tools such as `OpenMP`. We also aim to show that the `OpenMP` API is not as easy to use as it might seem.

The rest of the paper is organized as follows. The following section discusses related work. Section III discusses concurrency and other concepts in parallel computing. Section IV presents the idea of compilation directives. Section V discusses the methodology applied in the experiments, and Section VI presents our results. Section VII presents the perceptions of the graduate students regarding the exercise and learning `OpenMP`. Finally, Section VIII presents the conclusions and next steps.

II. Related Work

There has been a large body of works that have presented results of studies and experiments in teaching parallel computing courses. Süß and Leopold [17] discuss the teaching of parallel computing, in particular the use of `OpenMP`. The authors present the most common errors identified in assignments given in parallel computing classes. The errors were classified in two categories: *correctness mistakes* and *performance mistakes*. Errors that impact the correctness of a program are errors of the first category. Errors which make programs perform slower are classified in the second category. The work also provides a checklist containing tips to avoid the most common mistakes.

Falcao [18] defends that teaching a parallel programming model or interface such as `OpenMP` in the beginning of undergraduate courses is not much harder than teaching a language like C or C++. The benefits of teaching these skills early on in the course would be interesting, considering that the majority of computer architectures are now are multi-core. Parallel computing should be taught with the use of higher level tools. One of the approaches proposed by Ferner *et al.* [19] is to use compiler directives to describe how a program should be parallelized.

The need for Structural and Computational Patterns to Parallel Computing is defended by Asanovic *et al.* [6] as the way to allow portability of solutions and the specialization of applications with the same behavior. Using patterns to design applications make it easy to change the target platform. The applications will be bound to platforms in low levels of architecture and probably inside a specialized function which can be easily changed.

A similar approach is proposed by Pllana *et al.* [7] in a programming environment that uses a combination of Model-Driven Development and Parallel Building Blocks. The idea is to create the model and then generate or transform the code using model-to-source tools. The proposed environment uses intelligent agents to assist the programmer, this is done during the high level program composition and when the programmer loads a BLAS function for some dense linear algebra operation, the tool suggests the use of the appropriate parallel basic block. The parallelism is inside the parallel building blocks and the programmer is not exposed directly to the complexity of parallel programming problems.

Speyer *et al.* [20] defend that implementing applications with high performance is harder than ever, specially due to the emergence of multi-core architectures. This leads to the development of hybrid applications that use both the `OpenMP` and `MPI` (*Message Passing Interface*) APIs. This presents even greater challenges to the programmers, since they now need to know how to use both code annotations and message passing abstractions to achieve high performance. The paper then compares different parallel programming tools with regard to their ease of use. There is a substantial effort to start writing parallel and high performance code even with these tools.

Some problems can be solved using different directive combinations, and the chosen combination influences the code that is generated. The `Parallware` [21] source-to-source compiler showed how different strategies and uses of directive combinations can lead to different performance results. The use of the wrong `OpenMP` directives and strategies can significantly reduce the performance over a sequential version.

The `OmpVerify` [22] system implements a static analysis tool based on the polyhedral model to extract the parallel portions of a program automatically and to detect important classes of common data-race errors in `OpenMP parallel for` directives. Saillard *et al.* [23] statically validates barriers and work sharing constructs to avoid the improper use of directives that could cause deadlocks. Süß and Leopold [24], present an user's experience with parallel sorting and OpenMP.

They have used several versions of a simple parallel sorting algorithm to show some weaknesses of OpenMP.

Substantial efforts have been made to achieve the simplest possible way to parallelize code, and the task is not easy. In this paper we present an exploratory analysis of common `OpenMP` errors, confirming that programming with high-level APIs and abstractions can still be difficult.

III. Concurrency and Multithreading Programming

A program is composed of data declarations, assignments and control flow, following the syntax and abstractions of a programming language. A concurrent program is a set of sequential programs that can be executed in parallel [25].

In 2004, the first multi-core processor was launched [26] into the mainstream market. The increasing number of cores of current parallel machines has motivated concurrent execution of tasks in all scientific and commercial domains, although this paradigm may still face some resistance [27] [28] [29]. Two important concepts are *concurrency* and *parallelism*. Concurrency occurs when multiple tasks are logically active at the same time and parallelism occurs when multiple tasks are actually active at the same time. In this case parallelism is related to physically performing tasks in parallel. In a computer with a multi-core processor we can have physical parallelism.

In practically every language that supports multi-threading, the implementation of this concept follows the idea of having a function with the code to be executed by the thread when it is created. Programming multi-threaded applications is a complex task, independently of the language used. The non determinism that is introduced by the use of threads can be controlled using synchronization mechanisms such as locks [28], semaphores and barriers, but these mechanisms increase the complexity of the program even more.

In C/C++ languages the `pthreads` library can be used to control thread execution on GNU/Linux. This library implements the standard POSIX API (IEEE 1003.1c) and provides mechanisms to create and synchronize threads. The choice to use the `pthreads` library, and not `OpenMP`, can be motivated by the fact that `pthreads` is a common library, available on platforms with GNU/Linux, and operates in a lower level of abstraction in comparison with `OpenMP` threads. The Code 1 presents a sample of code that uses the `pthreads` library.

```
1  int main(){
2    pthread_t thread1;
3    DATA_TYPE *params = 10;
4    int iret1;
5
6    iret1 = pthread_create( &thread1, NULL,
         task_function, (void*) params);
7    printf("Thread 1 return: %d\n", iret1);
8    pthread_join(thread1, NULL);
9
10   return 0;
11 }
12
13 void *task_function(void *ptr) {
14   DATA_TYPE *value;
15   value = (DATA_TYPE *) ptr;
16   /* Computing */
17 }
```

Code 1. The Sample of pthreads code.

This sample code shows some important concepts. Until `line 6` only one thread is executing the main flow of instructions. When the `pthread_create()` function is called, a new thread is created and then the application have two threads executing in parallel. For synchronization between threads barriers are necessary. The library implements barriers and they can be used in the code by calling `pthread_join()`. In `line 8` we can see an example of this, where the thread that reaches that point will wait for the completion of other threads to continue the execution of the main flow.

To explain the use of `pthreads` we use the sample code in Code 2 to create a parallel version. The code have a reduction operation, in the multiplication of elements of A and B to the *dot* variable. In this case, we want to parallelize the loop instances by dividing the amount of instances among a number of threads.

```
1  for(i = 0; i < N; i++){
2    dot += A[i] * B[i];
3  }
```

Code 2. Sample of Basic Code.

The Code 3 shows a vector product operation that was implemented using `pthreads`. This example shows that we need to write several lines of code to perform a simple operation like sharing work between threads.

```
1  int A[SIZE], B[SIZE];
2  int dot = 0;
3
4  pthread_mutex_t lock;
5
6  int main( int argc, char *argv[] ){
7    int num_threads, i, part_size;
8
9    pthread_t tid[MAX_THREADS];
10
11   pthread_attr_t attr;
12   pthread_attr_init(&attr);
13
14   /* Initialization vectors was supressed. */
15
16   part_size = (SIZE + num_threads - 1) /
         num_threads;
17
18   for (i=0; i < num_threads; i++) {
19     struct arg_struct *args = malloc(sizeof(
           struct arg_struct));
20
21     args->id = i;
22     args->i_part = (i * part_size);
23     args->e_part = MIN(((i + 1)* (part_size)),
           SIZE);
24
25     pthread_create(&(tid[i]), &attr, vecDot, (
           void *) args);
26   }
27
28   for (i=0; i<num_threads; i++) {
29     pthread_join(tid[i], NULL);
30   }
31
32   printf("Final result: %d.\n", dot);
33
34   return 0;
35 }
36
37 void *vecDot(void *arguments){
38   struct arg_struct *args = arguments;
39   int i, partial_dot = 0;
```

```
40
41    for(i = args->i_part; i < args->e_part; i++){
42      partial_dot += A[i] * B[i];
43    }
44
45    /* Critical region. */
46    pthread_mutex_lock(&lock);
47    dot += partial_dot;
48    pthread_mutex_unlock(&lock);
49
50    pthread_exit(0);
51  }
```

Code 3. The vetdot sample using pthreads.

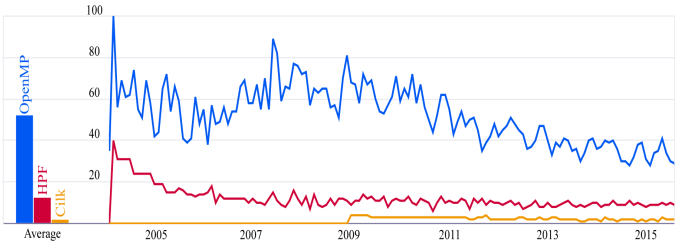

Fig. 1. Comparison of the Google Trends results for "OpenMP", "HPF" and "Cilk".

In the main function, the arrays are initialized and divided among the number of created threads. Each thread executes the `vetDot` function, saving the calculated *partial_dot* value in the global *dot* variable. This is done inside a delimited critical region, defined by the use of using *pthread mutexes*.

IV. Compilation Directives

Since the introduction of heterogeneous parallel computing platforms tools have been developed to provide interfaces between compilers, runtimes and execution environments. The compilation directives is one of the approaches that has excelled in the context of parallel computing, because annotating code is usually easier than rewriting it. These directives are commonly implemented using the pre-processing directives `#pragma`, in `C/C++`, and sentinels `!$`, in the Fortran language.

Directives are used as annotations that provide tips about the original code and guide the compiler in the parallelization process of candidate regions. During the pre-processing, the *tokens* that compose the directive can be replaced using macros, for example. The parallel version of the code will use features provided by the `OpenMP` runtime.

The use of directives was not introduced by `OpenMP`. Directives have been used as a macro specification language used in the pre-processing of languages such as C/C++ and Fortran. The macros transform the source code before the compilation. Languages such as `HPF` [30] [31] had compiler directives for parallelism before the rise of `OpenMP`. However, with the popularity of multi-core processors and `OpenMP` supporting C/C++ and Fortran applications, `OpenMP` has become a robust runtime for multi-core processors and an interesting application of compilation directives. Figure 1 presents a comparison between the search results for the terms "`OpenMP`", "`HPF`" (High Performance Fortran) and "`Cilk`". The constantly higher number of searches for "`OpenMP`" can be taken as confirmation of its popularity.

Using compilation directives is interesting because legacy code applications can be annotated and then parallelized by the runtime, to achieve performance improvements on multi-core processors. If the annotated code is analyzed by a compiler that supports `OpenMP` [8] [32] [11] [9] [33], a parallel version will be generated. Otherwise, the directives will be ignored and the code will remain sequential. Using the `OpenMP` runtime, parallel C/C++ or Fortran programming requires only the learning the directives' usage.

`OpenMP` is a set of compilation directives, functions libraries and environment variables that are based on the C/C++ and Fortran languages. The high level directives are able to expose the low level parallelism resources available on native implementations like the `pthreads` interface on GNU/Linux. Using the `OpenMP` directives much of the work is done automatically and implicitly, parallel regions have implicit barriers and the declaration of shared or private variables is simple.

`OpenMP` implements the `fork-join` model, in which multiple threads of execution perform tasks defined implicitly or explicitly by `OpenMP` directives [32] [11]. Threads are created from annotated code blocks. The *master thread* executes sequentially until it finds the first parallel region. The similar `fork` operation creates a team of threads to execute a parallel region, the *master thread* is part of the team. Nested parallel regions are allowed. When all threads in a team reach the barrier that marks the end of the parallel region, the master thread continues until it finds a new parallel region. From the annotated blocks, the runtime can generate implicit threads to execute the parallel regions or assign to some other thread the execution of explicit tasks defined by the programmer using the `task` directive.

Multiple compilers implement `OpenMP`, such as `GCC` [34] and LLVM `clang` [35] [36] [37], for example. `OpenMP` is certainly easier to use than `pthreads` [38]. The directives allow a first contact with parallel programming that uses basic examples, letting the programmer experience an effortless application of parallel computing. However, complex parallel programming require more than simple `OpenMP` directives. Code 4 presents an example of the `OpenMP` compilation directives used to compute the dot product, which can be compared to a `pthreads` version presented in Code 3.

```
1  #pragma omp parallel firstprivate(aux_dot)
2  {
3    #pragma omp single
4    printf("Begin of the parallel region, number
         of threads: %d\n", omp_get_num_threads());
5
6    #pragma omp for schedule(runtime)
7    for(i = 0; i < SIZE; i++){
8      aux_dot += A[i] * B[i];
9    }
10
11   #pragma omp critical
12   dot += aux_dot;
13
14   #pragma omp master
15   printf("Final result: %d.\n", dot);
16 }
```

Code 4. Sample of OpenMP code.

Expanding the code with directives allows to see how OpenMP reduces the amount of code that the programmer needs to write. The code with the expansions is shown in Code 5.

```
1  long long A[SIZE], B[SIZE], dot = 0;
2
3  void subfunction0 (void *data) {
4    long long aux_dot = 0;
5
6    if (GOMP_single_start ())
7      printf("Begin of the parallel region, number
           of threads: %d\n", omp_get_num_threads
          ());
8    GOMP_barrier ();
9
10   {
11     long i, _s0, _e0;
12     if (GOMP_loop_runtime_start (0, SIZE, 1, &
           _s0, &_e0))
13       do {
14         long _e1 = _e0;
15         for (i = _s0; i < _e0; i++){
16           aux_dot += A[i] * B[i];
17         }
18       } while (GOMP_loop_runtime_next (&_s0, &
             _e0));
19     GOMP_loop_end ();
20   }
21
22   GOMP_critical_start();
23   dot += aux_dot;
24   GOMP_critical_end();
25
26   if (omp_get_thread_num () == 0)
27     printf("Final result: %d.\n", dot);
28 }
29
30 int main() {
31   /* Initializations were supressed. */
32
33   GOMP_parallel_start (subfunction0, NULL, 2);
34   subfunction0 (NULL);
35   GOMP_parallel_end ();
36
37   return 0;
38 }
```

Code 5. OpenMP expanded code

The Code 5 shows calls to GOMP_parallel_start() (line 33) and GOMP_parallel_end() (line 35). These functions are responsible to start and finish the team of threads that run subfunction0(...). In the code it is also possible to see the functions that are mapped to barriers and synchronization. The calls to GOMP_barrier() in line 8, GOMP_critical_start() and GOMP_critical_end() in lines 22 and 24, are mapped to semaphore and barrier primitives that are implemented in specific OpenMP libraries.

The OpenMP runtime uses the standard thread library of the system, in this case, pthreads. The library is used together with other native system resources to control scheduling and synchronization. This shows that OpenMP is simpler comparing to code using pthreads, but the programmer still needs to master fundamental concepts about parallelism.

Tutorials present content in a structured way, showing the syntax of directives and exemplifying the use of annotations. The examples are simple in most cases, as can be seen in Code 4. This kind of example brings to student the idea that OpenMP is always very simple. They simply annotate the code and make the program parallel. In simple applications, we write one or two annotations in the code and generate a parallel version without difficulty. However, when we have more complex applications, it is necessary to know more concepts of parallel computing to apply the correct modifications in the source code.

Another common situation when using compilation directives is obtaining a better performance for a parallel version of the code. It is necessary to perform large efforts to include the annotations and their proper restrictions. That turns the process into something like "annotation or directive oriented programming". The time gained by not rewriting the application code is spent declaring directives and their restrictions.

Krawezik and Cappello [15] evidenced this difficulty by comparing MPI and three OpenMP versions of benchmarks. The most popular use of OpenMP directives is at the loop level. In this approach the programmer discovers potentially parallel loops by profiling and checking the dependencies. The programmer then annotates the code with directives starting and closing parallel regions at each discovered parallelizable loop. However, good results were obtained only when SPMD was implemented on OpenMP, and achieving good performance required a significant programming effort [15].

V. Methodology

The motivation to write this paper comes from experiences on teaching. During the classes of Distributed Systems and Parallel Programming, a graduate course at University of São Paulo, a problem was detected in an OpenMP code sample.

To explain the idea we use as a basis the code sample showed in Code 2 of section III. If we need to parallelize the loop instances, we can use the OpenMP *reduction* clause because the code has a *reduction* operation, but the idea of the tutorial was to show the use of the *critical* clause.

The code written on the tutorial sample to solve using *critical* clause is shown in Code 6. However, this code has a problem.

```
1 #pragma omp parallel
2 {
3   #pragma omp for
4   for(i = 0; i < N; i++){
5    aux_dot += A[i] * B[i];
6   }
7   #pragma omp critical
8   dot += aux_dot;
9 }
```

Code 6. Code of original OpenMP tutorial with problem.

The idea that was used to parallelize the computing on line 5 was to put the for loop inside a parallel region and parallelize the instances of the loop using the directive #pragma omp for. An auxiliary variable aux_dot was used to keep the partial results in each thread, but aux_dot was not declared as private to each thread. This generated a race condition, in which threads can read the same value and can be interrupted before storing the updated value.

This situation was identified as a possible problem. In the second tutorial the authors thought the problem was in the initialization of the variable *dot*. The code was "fixed" using the `firstprivate(dot)` clause. The semantic of this clause is: "all variables in the list are initialized with the original value before entering the parallel region". Code 7 presents the first attempt to fix the code in the class.

```
#pragma omp parallel firstprivate(dot)
{
  #pragma omp single
  printf("Begin of the parallel region, number
      of threads: %d\n", omp_get_num_threads());
  #pragma omp for
  for(i = 0; i < SIZE; i++){
    aux_dot += A[i] * B[i];
    printf("Thread %d executes the loop
        iteration %02d: %lld * %lld = %lld -> %
        lld \n", omp_get_thread_num(), i, A[i],
        B[i], (A[i] * B[i]), aux_dot);
  }
  #pragma omp critical
  dot += aux_dot;

  #pragma omp master
  printf("Final result: %lld.\n", dot);
}
```

Code 7. The first version of "fixed" code.

The author added lines on the code to show the final result, but with these modifications a new error was inserted. Only by the master thread prints the final result message, and this can occur before the completion of other threads, because the *critical* clause does not have barriers, it only prevents concurrent accesses. The corrected code still had problems. One team of students found another problem in the "fixed" version of sample code. Code 8 presents the fixed and final code version.

```
#pragma omp parallel firstprivate(aux_dot)
{
  #pragma omp single
  printf("Begin of the parallel region, number
      of threads: %d\n", omp_get_num_threads());
  #pragma omp for
  for(i = 0; i < SIZE; i++){
    aux_dot += A[i] * B[i];
    printf("Thread %d executes the loop
        iteration %02d: %lld * %lld = %lld -> %
        lld \n", omp_get_thread_num(), i, A[i],
        B[i], (A[i] * B[i]), aux_dot);
  }
  #pragma omp critical
  dot += aux_dot;
}
printf("Final result: %lld.\n", dot);
```

Code 8. The final version of code sample.

The last correction puts the final message out of the parallel region to avoid printing incomplete results. This occurs because at the end of all parallel regions in OpenMP there is one implicit barrier that joins the work flow and only the `master thread` continues the execution in this case.

This simple example illustrates how mistakes can be made. The example motivated the proposition of an exercise for the students. It also reinforced the importance of fundamental parallelism concepts even in simpler approaches. A challenge was then proposed for the students, as one of the activities of the course. After the introductory lecture on `OpenMP`, the groups should search available `OpenMP` tutorials on the Internet for errors, and fix them. This exercise had 27 participants, the group of students was divided in 6 teams of two students and other students preferred to work alone. Most students used the `GCC` compiler and the `GNU libgomp` library in their assignments.

We applied a questionnaire to obtain more information about the students. Most students already knew threads, but few students had experience with `OpenMP`. Figure 2 presents how many students had contact with each parallel programming paradigms before the course. Since this is a graduate level course, we had students of various ages. This is reflected by the large differences in experience time between students, showed in Figure 4.

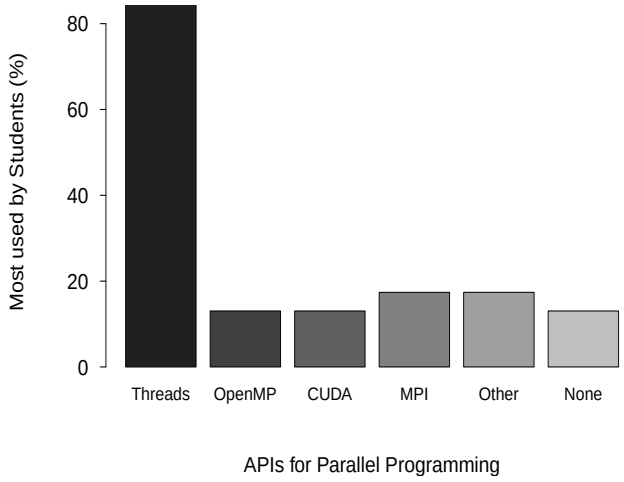

Fig. 2. APIs for Parallel Programming previously used by the students

The other question was about the programming languages known by the students. The objective of this information was to measure how easy it is to learn `OpenMP` when you know the basic concepts and what is the influence that the knowledge of other languages can have on the process. Figure 3 presents the distribution of programming languages known for the students involved in the experiment.

VI. Results

After the students submitted the assignment for evaluation, we analyzed the reports and tutorials where errors were found. One of the examples shown in class presented an error, and multiple errors were found in several tutorials available on the Internet. Table I summarizes the errors found in tutorials from the Internet and their corrections by the students.

The *first tutorial* in Table I has a problem caused by a confusion about which variables are shared and which should be private. This problem is similar to the ones identified by Süß and Leopold [17] as common correctness mistakes ("Forgetting to mark private variables as such"). It was the fourth most frequent mistake in that study.

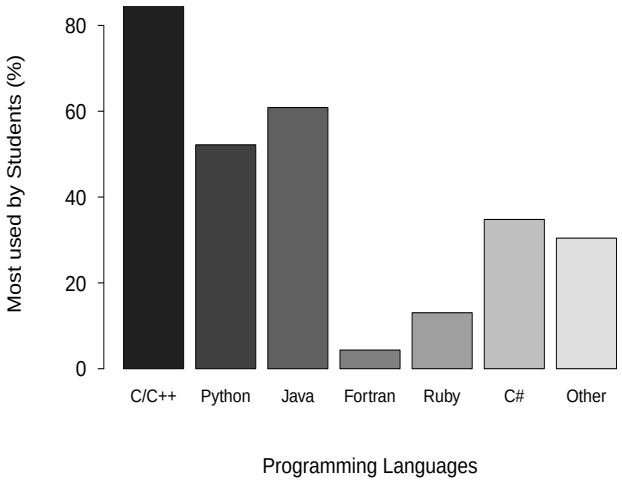

Fig. 3. Programming Languages more used by the students

Tutorial 2 presents a problem with the interaction between OpenMP and the system's OS and compiler. Specifically, the GCC `fork()` implementation. When we use the fork primitive, the Operating System creates other processes (child processes) duplicating all resources used by processes that execute the `fork()` call. Using OpenMP and creating child processes makes the thread pool created by OpenMP to be passed to child processes, but `fork()` does not duplicate the threads, which causes deadlock in the child processes.

The problem presented in *tutorial 3* was a syntax error, there was a missing "{" after the directive.

Tutorial 4, presented in Table I, was found by students because of an error using the `ordered` clause. When using this clause it is necessary to indicate that it will be used inside a loop. A similar mistake involving the clause was identified as common by Süß and Leopold [17]. The opposite situation happened when an `ordered` clause was put into a `for` construct without being specified by a separate `ordered` clause inside the loop. This is a restriction on the nesting of regions, an `ordered` region must be closely nested inside a loop or parallel loop region with an `ordered` clause [32].

Other problems in private variable definition were found in *tutorials 5* and *7*. In *tutorial 5* the `tid` variable is not defined as private, causing some values in the array not being calculated. In *tutorial 7* the variables `j` and `k` in the matrix multiplication are shared by all threads, generating wrong results.

In *tutorial 6* the `cout` function is called inside a parallel region without a protecting `critical` clause. The call becomes a critical section, because the threads are interrupted during the printing and messages are concatenated generating incorrect results. The code was modified by students using the `critical` clause to ensure that the call is executed by one thread at a time.

Tutorial 8 presents a synchronization problem that causes an error in the assignment of values for the `startv` and `endv` variables that keep the start and end positions of the shortest path. Because of this problem the algorithm does not reach all graph vertices. The critical section is created to update the minimum value and the vertex (`md` and `mv`), but the code is not avoiding the occurrence of updates before another thread updates the global array. If all vertices are not visited some initialized values are not updated during the program execution, which prints the incorrect values. The code was modified using the original algorithm idea and a barrier was added to ensure synchronization using the `#pragma omp barrier` directive after the `#pragma omp critical`.

TABLE I. TUTORIALS THAT PRESENTED SOME PROBLEM

Id	Tutorial	Identified Problem	Correction
1	`Tutorial de OpenMP C/C++ [39]`	Incorrect result. Race condition in results variables was detected.	The clause firstprivate was used to transform aux_dot in private variable in threads. `#pragma omp parallel firstprivate(aux_dot)`
2	`Guide into OpenMP [40]`	`OpenMP` interaction with Linux fork implementation using the GCC. The created thread pool are passed to child processes, but the fork() do not duplicate the threads causing deadlock in the child processes.	Make fork() before application of pragmas in the parent process. Organize the code for child processes create their own thread pool.
3	`Code Project [41]`	Sintax error. Missing the "{" after directive for.	The students tested take off "}" at the final causing segmentation fault error. So they put the "{" after the directive for.
4	`Parallel Processing with OpenMP [42]`	Error on use of ordered clause. It is necessary to indicate that the ordered clause will be used inside of loop.	The ordered clause was put.
5	`Task Parallelism in OpenMP [43]`	The tid is not defined as private. It is shared by all threads the vector A values are not calculated.	Declare tid as private using the `private(tid)` clause.
6	`Introduction to Parallel Processing. OpenMP [44]`	Use of `cout` function inside parallel region without critical clause. The threads are interrupted during the printing, messages are concatenated.	The code was modified using the critical clause.
7	`Introduction to Parallel and Distribution Systems. OpenMP Overview [45]`	In the matrix multiplication code the variables j and k are shared by all threads generating wrong result.	Addition of private clause on `#pragma omp parallel for private(j, k)`
8	`Introduction to OpenMP. Dijkstra sample [46]`	When the code is executed with more of 3 threads produces wrong results. There is an error in assigning values for $startv$ and $endv$ variables that keep the start and end positions of the shortest path. With this problem the algorithm does not reach all graph vertices. On line 88 a critical section is created for updating the minimum value and the vertex (md and mv) but the code is not avoiding that updates occurs before another thread update the global vector, it is printing the values set in the initialization vector.	The definition of variables $startv$ and $endv$ was modified using the original idea of algorithm. The `#pragma omp barrier` directive was added after the `#pragma omp critical`.

Table II summarizes the type of errors found by the students in Internet tutorials.

Comparing the problems found in the experiment to the most common OpenMP mistakes [17] we can see how some errors are recurrent, for instance, the decision about isolating data using declarations of shared or private variables.

TABLE II. SUMMARY OF ERRORS

Kind	Quantity	Tutorial
private-shared	3	1, 5 and 7
syntax error	1	3
wrong use of directive	1	4
synchronization problems	2	6 and 8

VII. GRADUATE STUDENTS PERCEPTION

The OpenMP exercise was proposed just after basic concepts of parallelism with shared memory were taught to the students. The wording of the exercise was: *"Search for a sample code in OpenMP tutorials and manuals on the Internet that uses compilation directives and has some kind of error. Write a report introducing the code, pointing the problems and describing which are the corrections made to eliminate error."*.

After that we provided an anonymous questionnaire using *Google Forms*. The question was: *"What is your opinion about your OpenMP experience and the OpenMP exercise¿ '*. We got 24 answers, all of them with positive feedback, and selected some of them below.

Several students commented on how easy it was to use OpenMP: "... I appreciate the idea of annotating an already existing program..."; "... with OpenMP it is possible to make an application parallel with low effort."; "Why, when I learned concurrent programming, I did not learn it (OpenMP)?". The students appreciated learning and using OpenMP in a simple example.

Several students also commented on the lack of a more explicit vision: "...When we use threads explicitly, I have the impression that it is easier to understand possible problems."; "When I use OpenMP, I am always afraid about the correctness, and if I used the correct optimization flags". There were also some complaints about how barriers work in OpenMP: "I had little experience with OpenMP and pthreads, but I believe that the understanding of pthreads is easier for those who already have experience in C. The (OpenMP) implicit barriers make me confused, since I never know when there is or not a barrier.".

There were also some more interesting ideas: "This experience encouraged me to write programs easier to parallelize with OpenMP.". The following answer summarizes the opinions on the ease of use of the API, and also on the possible difficulty of working with OpenMP: "... OpenMP appears to be a good framework for productivity, that is, it allows the creation of concurrent programs in less time and with less code rather than with pthreads. But it does not make parallel programming a trivial or error-free task.".

In another question we asked the students about their experience with programming and about the difficulties they had learning OpenMP directives. Figure 4 shows the results.

Many students reported that they had difficulties to learn OpenMP. The majority had no previous experience, as shown in Figure 2. In Figure 4 it can be seen that most students also had less than ten years of programming experience. There was only 31 days between the OpenMP introductory classes and the due date of the assignment, and this may have contributed to their difficulties.

These answers provide some evidence that, despite being considered an easy approach, OpenMP also requires knowledge and experience.

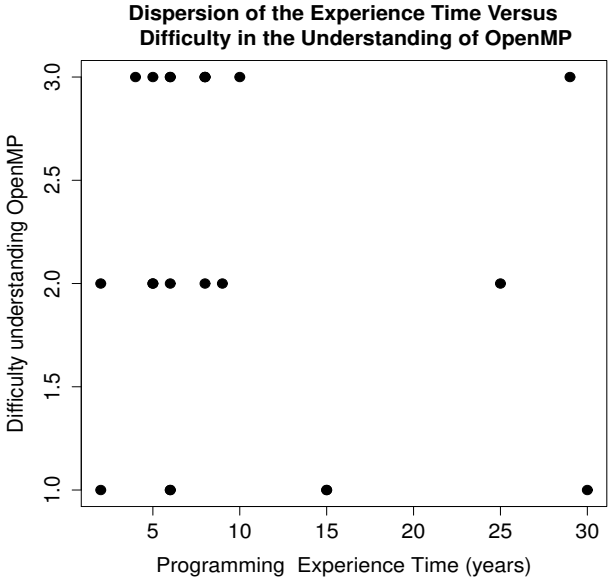

Fig. 4. Graphic of dispersion of the experience time versus difficulty in the understanding of OpenMP

VIII. CONCLUSIONS AND NEXT STEPS

This paper's results reiterate that, even though OpenMP can be easy to use, programmers still need to know the details about parallel computing architectures and platforms. Teaching and learning the fundamental concepts about parallel programming is, and will continue to be, necessary to write accurate and high-performance code.

The experimental results show that although Internet tutorials transmit the idea of ease-of-use of the compilation directives approach, the core concepts of parallelism are required to write good parallel programs. Knowing the underlying platform and architectural details, and what effects are caused by directives and their interactions is also required.

Using OpenMP for trivial programs is easy, but for more advanced applications it is necessary to know how to correctly use locks and barriers, avoid deadlocks, and distribute the work evenly between threads. The programmer also has to be cautious when scheduling and accessing memory. To do all this correctly it is necessary to learn the fundamental concepts underlying parallel programming.

The main goal of parallel programming is writing programs that are efficient, portable and correct, leveraging contemporary hardware platforms [6]. These goals are still hard to reach even when using high-level approaches such as OpenMP.

It is important to teach Computer Science students how to program parallel applications from the beginning. Instead of teaching sequential programming first and parallel programming much later in the graduation course, Computer Science courses should start with OpenMP or other high-level tools, but not before providing solid fundamentals in parallel computing.

Considering the students' answers, a positive outcome of this experiment is that the students have learned that they should not rely on copying code from the Internet, because they can contain errors. Students also learned that they should test their code thoroughly, as errors in parallel programs are not easily detectable.

Future work will repeat the experiment in other instances of the graduate course, improving the experiment data set. Using a more systematic approach, future work will look for possible errors in more tutorials available on the Internet, and explore the difficulties faced by beginner and expert developers of parallel applications.

Finally, considering the results presented in this paper and the fact that even those who are teaching make mistakes in their tutorials and classes, more attention has to be devoted to teaching parallel and concurrent programming, its techniques, concepts and tools.

ACKNOWLEDGMENTS

The authors would like to thank Students who participated in this exercise, CNPq, CAPES (BEX 3401/15-4, CAPES-COFECUB Project No. 828/15), São Paulo Research Foundation (FAPESP) (procs. #2012/23300-7) for the fellowships and the Araucária Foundation, Department of Science, Technology and High Education of State Government of Paraná (SETI-PR) and State Government of Paraná for the financial support.

REFERENCES

[1] C. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire, "Offload compiler runtime for the intel(r) xeon phi coprocessor," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1213–1225.

[2] J. J. Dongarra, H. W. Meuer, E. Strohmaier *et al.*, "Top500 supercomputer sites," *Supercomputer*, vol. 13, pp. 89–111, 1997.

[3] Top500. (2014) Top500 list of supercomputers site. [Online]. Available: http://www.top500.org/

[4] T. Chen and Y.-K. Chen, "Challenges and opportunities of obtaining performance from multi-core cpus and many-core gpus," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, April 2009, pp. 613–616.

[5] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 54.

[6] K. Asanovic, J. Wawrzynek, D. Wessel, K. Yelick, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, and K. Sen, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, p. 56, Oct. 2009. [Online]. Available: http://dl.acm.org/ft_gateway.cfm?id=1562783\&type=html

[7] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa, "Towards an intelligent environment for programming multi-core computing systems," in *Euro-Par 2008 Workshops - Parallel Processing*, ser. Lecture Notes in Computer Science, E. César, M. Alexander, A. Streit, J. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds. Springer Berlin Heidelberg, 2009, vol. 5415, pp. 141–151. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00955-6_19

[8] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=615255.615542

[9] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] OpenACC. (2013) OpenACC Application programming interface. version 2.0. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf

[11] OpenMP-ARB, "OpenMP Application Program Interface Version 4.0," OpenMP Architecture Review Board (ARB), Tech. Rep., 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[12] AMD. (2015) Heterogeneous computing. [Online]. Available: http://developer.amd.com/resources/heterogeneous-computing/

[13] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 404–418, March 2009.

[14] T. G. Mattson, "How good is openmp," *Scientific Programming*, vol. 11, no. 2, pp. 81–93, 2003.

[15] G. Krawezik and F. Cappello, "Performance comparison of mpi and openmp on shared memory multiprocessors," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 1, pp. 29–61, 2006. [Online]. Available: http://dx.doi.org/10.1002/cpe.905

[16] M. Ben-Ari and Y. B.-D. Kolikant, "Thinking parallel: the process of learning concurrency," in *ACM SIGCSE Bulletin*, vol. 31, no. 3. ACM, 1999, pp. 13–16.

[17] M. Süß and C. Leopold, "Common mistakes in OpenMP and how to avoid them: A collection of best practices," in *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, ser. IWOMP'05/IWOMP'06. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 312–323. [Online]. Available: http://dl.acm.org/citation.cfm?id=1892830.1892863

[18] G. Falcao, "How fast can parallel programming be taught to undergraduate students?" *Potentials, IEEE*, vol. 32, no. 4, pp. 28–29, July 2013.

[19] C. Ferner, B. Wilkinson, and B. Heath, "Toward using higher-level abstractions to teach parallel computing," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1291–1296.

[20] G. Speyer, N. Freed, R. Akis, D. Stanzione, and E. Mack, "Paradigms for parallel computation," in *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, July 2008, pp. 486–494.

[21] Parallware. (2014, May) Parallware: The openmp-enabling source-to-source compiler. [Online]. Available: http://www.appentra.com/parallel-computation-pi

[22] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott, "ompverify: Polyhedral analysis for the openmp programmer," in *OpenMP in the Petascale Era*, ser. Lecture Notes in Computer Science, B. Chapman, W. Gropp, K. Kumaran, and M. Müller, Eds. Springer Berlin Heidelberg, 2011, vol. 6665, pp. 37–53. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21487-5_4

[23] E. Saillard, P. Carribault, and D. Barthou, "Static validation of barriers and worksharing constructs in openmp applications," in *Using and Improving OpenMP for Devices, Tasks, and More*, ser. Lecture Notes in Computer Science, L. DeRose, B. de Supinski, S. Olivier, B. Chapman, and M. Müller, Eds. Springer International Publishing, 2014, vol. 8766, pp. 73–86. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11454-5_6

[24] M. Süß and C. Leopold, "A user's experience with parallel sorting and OpenMP," in *In Proceedings of the Sixth Workshop on OpenMP (EWOMP'04*, 2010.

[25] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[26] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095421

[27] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[28] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[29] S. Qadeer and D. Wu, "Kiss: Keep it simple and sequential," *SIGPLAN Not.*, vol. 39, no. 6, pp. 14–24, Jun. 2004. [Online]. Available: http://doi.acm.org/10.1145/996893.996845

[30] H. Richardson, "High performance fortran: history, overview and current developments," 1.4 TMC-261, Thinking Machines Corporation, Tech. Rep., 1996.

[31] H. Forum, "High performance fortran language specication," High Performance Fortran Forum, Tech. Rep., 1997. [Online]. Available: http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf

[32] OpenMP-ARB, "OpenMP Application Program Interface Version 3.1," OpenMP Architecture Review Board (ARB), Tech. Rep., 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[33] OpenMP API Site. (2012, Jan.) The OpenMP® API specification for parallel programming. [Online]. Available: http://openmp.org

[34] GNU Libgomp, "GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation," GNU libgomp, Tech. Rep., 2015. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-5.2.0/libgomp.pdf

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

[36] LLVM Clang. (2015) clang: a C language family frontend for llvm. [Online]. Available: http://clang.llvm.org/

[37] LLVM OpenMP. (2015) OpenMP®: Support for the OpenMP language. [Online]. Available: http://openmp.llvm.org/

[38] B. Kuhn, P. Petersen, and E. O'Toole, "OpenMP versus threading in C/C++," *Concurrency: Practice and Experience*, vol. 12, no. 12, p. 1165–1176, 2000. [Online]. Available: http://dx.doi.org/10.1002/1096-9128(200010)12:12⟨1165::AID-CPE529⟩3.0.CO;2-L

[39] M. C. R. Sena, J. A. C. Costa, C. A. Thomaz, E. S. S. Silveira, R. Filho, H. S., and A. A. Russo, "Tutorial de OpenMP C/C++," LCCV-UFAL, Tech. Rep., March 2008, sun Microsystems do Brasil, Programa Campus Ambassador HPC.

[40] OpenmpAndFork. (2008) Guide into OpenMP: Easy multithreading programming for C++. [Online]. Available: http://bisqwit.iki.fi/story/howto/openmp/#OpenmpAndFork

[41] C. Project. (2010) Code project tutorial. [Online]. Available: http://www.codeproject.com/Articles/60176/A-Beginner-s-Primer-to-OpenMP

[42] D. Sondak. (2011) Parallel processing with openmp.

[43] M. Hall. (2011) Lecture 9: Task parallelism in openmp. [Online]. Available: www.bu.edu/tech/files/2011/09/openmp.ppt

[44] S. Y. Cheung. (2013) Introduction to parallel processing. openmp. [Online]. Available: http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/91-pthreads/openMP.html

[45] C.-Z. Xu. (2004) Ece561:introduction to parallel and distribution systems. openmp overview. [Online]. Available: http://www.ece.eng.wayne.edu/~czxu/ece561/lnotes/OpenMP.pdf

[46] N. Matloff. (2006) Introduction to openmp. dijkstra sample. [Online]. Available: http://heather.cs.ucdavis.edu/~matloff/openmp.html#tutorials