Parallelizing Audio Analysis Applications - A Case Study

Ali Jannesari
University of California, Berkeley, USA
jannesari@eecs.berkeley.edu

Zia Ul Huda, Rohit Atre, Zhen Li and Felix Wolf
Technical University of Darmstadt, Germany
{huda, atre, li, wolf}@cs.tu-darmstadt.de

***Abstract*—As multicore computers become widespread, the need for software programmers to decide on the most effective parallelization techniques becomes very prominent. In this case study, we examined a competition in which four teams of graduate students parallelized two sequential audio analysis applications. The students were introduced with PThreads, OpenMP and TBB parallel programming models. Use of different profiling and debugging tools was also taught during this course. Two of the teams parallelized libVorbis audio encoder and the other two parallelized the LAME encoding engine. The strategies used by the four teams to parallelize these applications included the use of taught programming models, focusing on both fine-grained and coarse-grained parallelism. These strategies are discussed in detail along with the tools utilized for the development and profiling. An analysis of the results obtained is also performed to discuss speedups and audio quality of the encoded output. A list of the lessons to be remembered while parallelizing an application has been provided as well. These lessons include best pedagogical methods, importance of understanding the program before choosing a programming model, concentrating on coarse-grained parallelism first, looking for dependency relaxation, parallelism beyond the predefined language constructs, the need of practice or prior experience in parallel programming and the need for assisting tools in parallelization.**

Keywords*—Teaching parallel programming, Project based Parallelization, Parallelizing existing sequential applications

I. Introduction

Multicore processors with several cores on a single chip have become widespread and standard. This trend will not only continue to accelerate in the coming years but will also require software developers to focus more on parallel programming. Focusing on the practical skills, tools and techniques required for developing software for multicore systems can lead to exploring some new software engineering questions and provide some very useful answers.

Programmers often face several questions while parallelizing a program, such as which programming model would work best, which parallelization techniques would prove to be most useful, which tools are necessary for detecting parallelization, and how the existing sequential applications can be modified to exploit parallelism.

We conducted a case study of parallelizing real world programs using existing tools and libraries to answer some of these questions. The study occurred during a multicore programming course. Fourteen graduate computer science students participated in the course. The first four weeks of the course were dedicated for the introductory lectures and exercises about parallel programming. In the last ten weeks, different performance optimization techniques for parallelized applications and use of different instrumenting/statistical profiling tools like Gprof[1] and Valgrind[2] were taught to the students. During these ten weeks, the students were given the task to parallelize real world applications. We selected audio analysis applications, i.e. sequential libVorbis [3] and LAME [4] audio encoding applications since they are widely used, popular, complex and well documented. The students were divided into four teams and a competition was held between them to efficiently parallelize the selected applications. Teams A and B consisting of four students each, competed with each other to parallelize the libVorbis encoder, while teams C and D consisting of three students each, competed to parallelize the LAME encoder. We analyzed their strategies and approaches to parallelization for each application in detail and also performed a quantitative analysis of the results obtained by the teams. The winner among the four teams was decided based on a multitude of criteria.

Each of the four teams employed different strategies to parallelize their respective application. Teams A and B focused on coarse-grained parallelism. Team A concentrated on extending the C-interface of libVorbis with OpenMP, while Team B used the flow graph construct available in TBB for the same purpose and as a result were able to achieve higher speedup. Each team working on LAME employed a different strategy. While one of them used OpenMP to encode each input chunk separately the other team chose the master/worker pattern using PThreads library. Due to the use of serial optimizations, better control over the thread functionality through PThreads, and additional techniques like dynamic scheduling, the latter was able to gain the best average speedup of 9.04 for 12 threads on an Intel Xeon X5670 CPU with 6 hyper-threaded cores, in all of the teams. Unless specified otherwise, all speedups reported from hereon will be for the aforementioned configuration.

Students were asked to submit a weekly report about their progress and problems faced during parallelization. Additionally, we conducted periodic interviews with our students during the project phase to assess the effectiveness of the material taught. The general feedback of the students indicated that the tools and optimization techniques introduced during the course, helped them overcome problems during the parallelization of their respective applications. Based on our analysis we have drawn a list of lessons learned while parallelizing an application, which includes best pedagogical methods, having a thorough understanding of the program before choosing a programming model, focusing on coarse-grained parallelism first, finding relaxable dependencies, parallelizing beyond pre-defined language constructs, understanding the importance of prior experience in programming and the need of assisting tools for parallelization.

The rest of the paper is organized as follows. Section II provides an overview of the material introduced during the course. Section III provides an overview of libVorbis and LAME respectively. Section IV discusses the various strategies the teams employed to parallelize their respective application. Section V provides a quantitative comparison between various aspects of the teams' results for each of the projects, followed by Section VI, which provides a summary of the lessons we can offer using this case study. Related work is discussed in Section VII. Section VIII concludes this case study.

II. Pedagogical Aspects

In this section we briefly discuss the material taught in lectures during the multicore programming course. The lectures were mainly divided into three sections: parallel programming; profiling tools; debugging tools and optimization techniques.

A. Parallel Programming

The first set of four lectures were dedicated to introduce the basics of parallel programming. Students were taught the basic concepts of multicore hardware and different memory architectures. The second, third and fourth lectures introduced parallel programming using PThreads [5], OpenMP [6] and Intel TBB [7] respectively. Each lecture was followed by a programming exercise. The exercise tasks were designed to give a hands-on experience to parallel programming for the above mentioned programming models. Solving the exercise tasks was mandatory for students. Their solutions were revised, graded and used as the basis for grouping the students together into teams for the final project phase.

B. Profiling Tools

At the end of first phase of the lectures, the students were introduced to the project tasks: parallelizing libVorbis [3] and LAME [4]. The students were given two weeks to analyze the serial code of these projects. During this phase, we introduced them profiling tools: Gprof[1], Valgrind[2] and Intel VTune[8]. They used these tools for analyzing the sequential code of their respective projects. Outputs of the profiling tools revealed call graphs, hotspots and many other important information about the sequential code. This helped them form effective parallelization strategies. The strategies will be discussed in detail in section IV.

C. Debugging Tools & Optimization Techniques

Once the students developed their first working parallel versions, they needed to validate and optimize their applications in order to get better performance with correct functionality. During this phase we introduced gdb [9], Helgrind [10], [11], [12], TotalView [13] and ThreadSanitizer [14] as some of the debugging tools available for the multi-threaded applications. We encouraged the students to look for data races and other errors using these tools and report their output as a part of their final submission. These tools helped the students to get rid of some bugs in their parallelized applications.

The students learned some optimization techniques in the last phase of the project. We taught the basic techniques like load balancing, synchronization optimizations and cache-friendly programming. The students used the above mentioned tools to detect performance bottlenecks in their parallel programs and then applied the taught optimization techniques to overcome these bottlenecks and make their parallel applications more efficient.

III. LibVorbis and LAME

In this section we briefly introduce libVorbis [3] and LAME [4].

A. LibVorbis

Ogg Vorbis is a fully open, non-proprietary, patent-and-royalty-free, general-purpose compressed audio format for mid to high quality (8kHz-48.0kHz, 16+ bit, polyphonic) audio and music at fixed and variable bitrates from 16 to 128 kbps/channel. LibVorbis is the reference implementation of the Vorbis codec. It is the lowest-level interface to the Vorbis encoder and decoder, working with the packets directly.

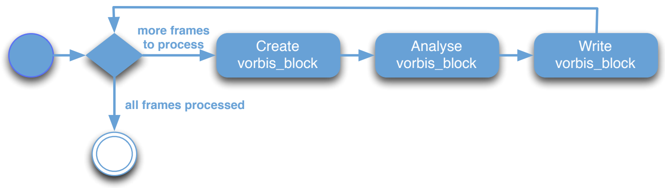

Fig. 1. Workflow for libVorbis Encoding.

Teams A and B were asked to parallelize the encoder of libVorbis. Figure 1 shows the encoding workflow of libVorbis. The encoder of libVorbis gets the audio data and initializes a `vorbis_block` for it. Then it analyses and encodes the `vorbis_block`. The encoded `vorbis_block` is written to the output stream. This process is repeated until the audio data is available for encoding.

B. LAME

LAME is a high quality MPEG Audio Layer III (MP3) encoder licensed under the LGPL. It encodes audio at mid-high and variable bit-rates into MP3 format.

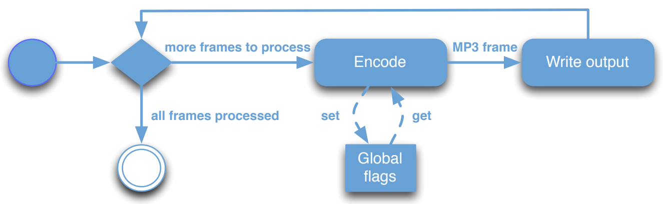

Fig. 2. Workflow of Lame MP3 encoder.

TABLE I. EXERCISE RESULTS.

Marks	# of Students
$\geq 90\%$	6
$\geq 80\%$	5
$\leq 80\%$	3

Teams C and D were asked to parallelize LAME. Figure 2 shows the encoding process of LAME. LAME encodes frames of the input audio one by one. During the encoding process, analysis states are saved and used to guide the encoding process of the next frame. This means each frame depends on the previous frame during the encoding process. Breaking these dependencies significantly reduces the quality of the encoded MP3 file.

IV. TEAM STRATEGIES

Table I shows the percentage of points achieved by students in the exercise sessions in the first phase of the course. We distributed the students into four teams based on their points. We tried to construct teams consisting of students drawn uniformly from every category.

Every team was free to choose a parallelization strategy of their own liking. The teams were asked to submit a weekly report of their progress during the project. This helped to keep track of the effort spent by the teams on different tasks and also the different strategies they followed. Any difficulties and changes in the strategy were also reported during the project .

The different strategies are described in the following sections.

A. Team A

Team A was comprised of four students. Their task was to parallelize libVorbis. They planned to use three different approaches: 1) workflow changes to allow using data decomposition patterns, 2) internal / fine-grained parallelization and 3) serial optimizations.

Team A's plan was to understand the code (one week), test parallelization ideas (one week), parallelize and optimize (two weeks), test and evaluate (two weeks) and document (two weeks). The team used Vampir[15], Gprof, Microsoft Visual Studio and Intel VTune to profile the code and detect the hot spots from the code for fine-grained parallelization. Most of these tools were introduced during the course.

They used OpenMP to incrementally parallelize the hot spots in the code. However, the speedups achieved were not promising. Several low level functions also had premature exit conditions, which are not allowed in OpenMP, so they abandoned this approach. Serial optimizations like the use of fast *abs* function calls reduced the execution times of serial programs by 5%.

The team largely focused on the coarse-grained parallelization of the code due to the workflow of the application, which included an interface feeding the data samples serially to the libVorbis library for encoding. They parallelized the interface to encode multiple samples in parallel.

For this approach, they wanted to extend the existing C-interface of libVorbis itself. Using Intel TBB, a C++ library, would break compatibility and was thus discarded for this implementation. They decided to use the task construct of OpenMP to manage heterogeneous tasks, while still being able to exploit the strengths of OpenMP, i.e., automatic creation, destruction and scheduling of threads.

The first parallelized version achieved a considerable speedup of about 3.4 times. Applying the profiling tools, taught during the course, on the parallel version revealed the following issues: A large sequential phase in the beginning, due to a sequential read of the whole input file, was detected; A synchronization overhead at the end of each encoding task was found due to sequential output of the data. Both of these issues reduced the overall speedup. Figure 3 shows the described issues with green showing the number of logical cores used by the encoding process. To overcome these issues, the team started to optimize their solution. They imported the data from the input file in chunks instead of all at once. This allowed the first parallel tasks to be created earlier, thus reducing the large sequential phase in the beginning, as seen in Figure 4 compared to that in figure 3. They discovered a pipeline pattern at this stage, but could not rewrite the entire codebase due to the time limitations.

The team tried to reduce the large amount of synchronizations at the end of the whole process using OpenMP *taskwait* and *usleep* for synchronizing the threads. They also decided to use hyper-threading for data output to reduce the sequential write at the end. In order to achieve this, they moved the writing of the encoded packets into tasks. Every 100th task must write out the next 100 finished workpackets. The restriction on the number of blocks writing at one time was chosen because otherwise other tasks might be stuck waiting to enter the critical section required for the write. So a uniform concurrency was achieved from start to end, as seen in sample Figure 4. This, along with further fine-tuning of the application and the use of thread affinity, enhanced the average speedup of the application to almost 6.0.

In the end, the team reported that they could have used the pipeline construct available in TBB for their approach but firstly, it would break the compatibility of libVorbis, written entirely in C, and secondly, when they discovered the pipeline pattern, they had already made an effort to implement the parallelized version in OpenMP using OpenMP tasks.

B. Team B

Team B was also comprised of four students with the same task of parallelizing libVorbis. They studied the code and profiled its dependencies to extract its dependency graph using Callgrind and Kcachegrind[16], both based on Valgrind, in the first week of project. They found the computationally intensive regions of the code and discovered that the task encoding the data block by block had the potential to be parallelized.

The approach of Team B was to use the pipeline pattern for achieving data parallelization as well as fine-grained parallelism. However, parallelizing the loops using OpenMP did not give appreciable speedup and in fact, the runtime was increased due to synchronization overhead. Also, the team was not able to parallelize all the loops found because they could not break the dependencies. This approach was abandoned.

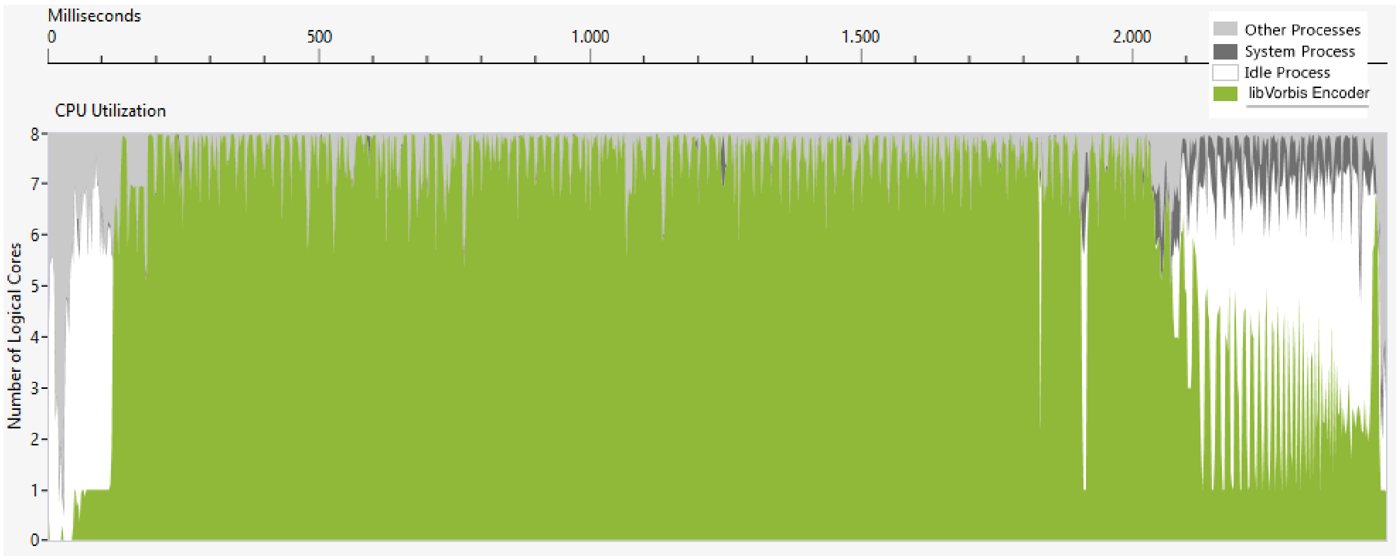

Fig. 3. Team A: First 3 seconds of sample concurrency plot for first parallelized version.

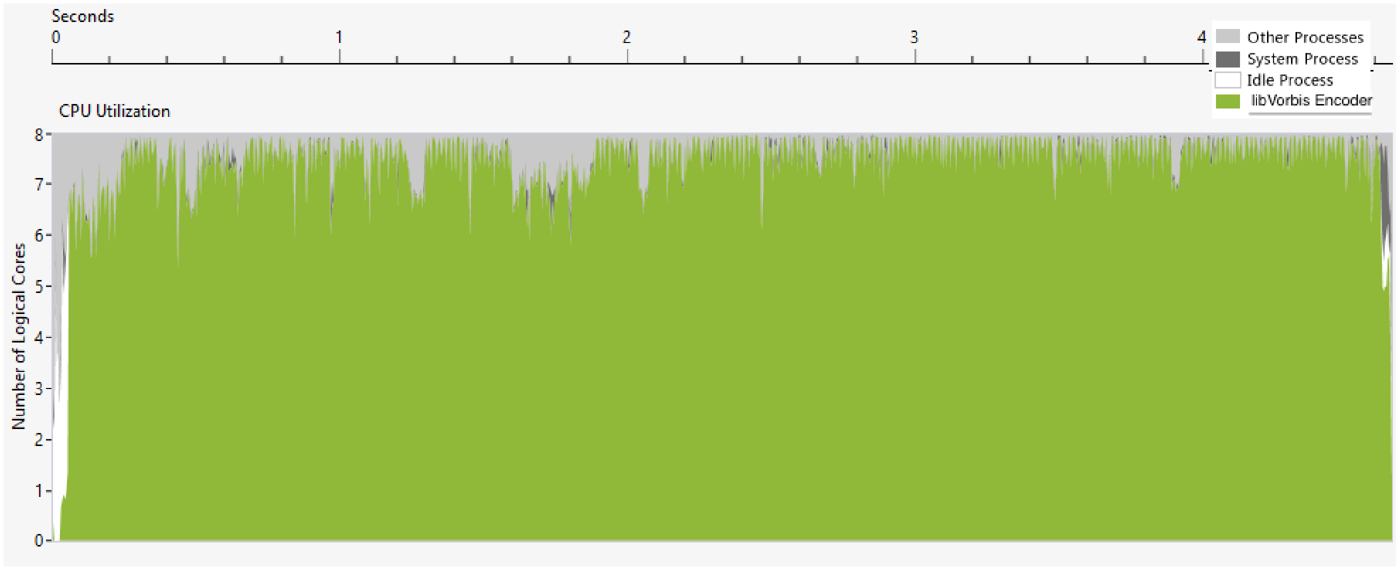

Fig. 4. Team A: Sample concurrency plot of final parallelized version for entire encoding process.

The second approach was to implement the pipeline using the TBB flow graph construct for data parallelization. Using TBB, Team B was able to get the first complete parallel version of the application in two weeks, the quickest of all 4 teams. This version was able to encode the data blocks in parallel.

Team B reported that a lot of time and effort was spent to find out all global variables which were shared among different threads. Then for each global variable, they detected all the source code lines at which those global variable might be accessed or modified by different threads. They needed to add synchronizations at these locations in order to avoid any data races.

The pipeline was implemented using both the pipeline and flowgraph constructs of TBB, but the version with the pipeline construct was found to be slower than the one with flowgraph construct. The reason for this was the usage of token-based scheduling of the pipeline construct, since the pipeline needs a specific token count as a maximum limit for the number of parallel processes. Finding an optimal number of tokens is very difficult. Unlimited node capacity for the flowgraph construct got the best speedup for the application, as the node capacity was determined by TBB automatically. The team was able to achieve an average speed up of 8.5.

Team B reported that their decision to use TBB for parallelization at the higher level of the application saved them from unnecessary complications or problems which they would have encountered if a pipeline had been constructed manually using OpenMP or Pthreads. Also, all the communication and synchronization done internally by TBB was optimal, thus improving the overall speedup of the program. TBB enabled them to quickly develop the parallel version.

TABLE II. TEAM COMPARISONS.

Team	Project	Profiling Tools	Library	Parallelism Proposed	Parallelism Implemented
A	libVorbis	Vampir, Gprof, MS Visual Studio, Intel vTune	OpenMP	Coarse and Fine-grained	Coarse-grained
B	libVorbis	Callgrind, Kcachegrind	TBB	Coarse and Fine-grained	Coarse-grained
C	LAME	Gprof, Intel vTune	OpenMP	Coarse and Fine-grained	Coarse-grained
D	LAME	Gprof, Intel vTune, intel Inspector, Scalasca	PThreads	Coarse and Fine-grained	Coarse and Fine-grained

C. Team C

Team C consisted of 3 students tasked with parallelizing the LAME MP3 encoder. The first week was spent analyzing the code and creating the dependency graph. They used Intel Vtune and Gprof for dependency analysis. Due to the nature of MP3 encoding, Team C discovered that the dependencies between two consecutive frames were very hard to break.

Two sets of flags named *global* flags and *internal* flags are used for each frame in the serial LAME encoder. Each frame has access to global flags and changes them according to its own state. The global flags are then used by the next frame.

The team opted for data parallelism at a higher level and had a working implementation in the next two weeks of the project. They considered three strategies: a three-stage pipeline with input, encode and output, a seven-stage pipeline and lastly, data parallelism.

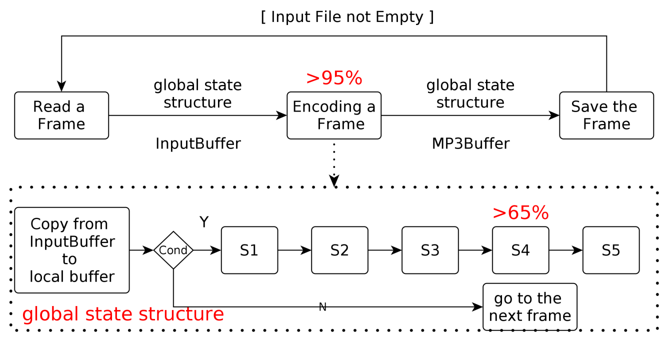

Fig. 5. Pipeline implementation approach of Team C.

The three-stage pipeline strategy shown in the upper part of figure 5 was found to be ineffective in practice, as the encoding stage took 95% of the overall processing time. In the same way, one of the stages of the seven-stage pipeline shown in lower part of figure 5, took 65% of the total processing time. This imbalance between stages would have worsened the pipeline's runtime and so the pipeline strategy was abandoned.

For data parallelism, the team found that the front end of the LAME encoder reads a frame of 1,152 samples from input and encodes them one by one until it is done. So three possible strategies could be derived for parallelization: encode samples in parallel in a frame containing 1,152 samples, encode frames in parallel or lastly, partition the input file into several sections to encode in parallel.

The team chose the third strategy for parallelization as the first two were not feasible due to global and internal flag dependencies. First, the whole file is read into main memory. The global flags are initialized at the start and copied to each section to avoid any race conditions. Each section is then encoded separately using OpenMP parallel constructs. The team had a working parallel implementation of a LAME encoder in the 8th week of the project. The only drawback of this strategy was the loss of information at the start of each section from the previous section, hence degrading the sound quality at that point. To reduce these effects on the output file, Team C created chunks of the input file with biggest size possible. The smallest chunk size allowed was 100 frames. The team was able to achieve an average speedup of 8.3.

The last two weeks of the project were spent on testing the parallelized version, documentation and evaluation. The team used Helgrind and Intel Vtune for the testing purposes. They reported a lot of data races reported by Helgrind but all of them were false positives either due to OpenMP thread creation or due to *memcpy* and *malloc* calls. In the end, Team C reported they achieved better speedup by parallelizing the front end of Lame. They had no time left to implement their pipeline ideas and explore if they could improve the runtime with that method.

D. Team D

Team D, consisting of three students, also parallelized the LAME MP3 encoder. Their parallelization approach was the global domain decomposition of the audio file. The input file is split into chunks and encoded in parallel. To break the dependencies between the last frame and first frame of successive chunks, ghost frames (copies of the last frames of the previous chunk) were introduced before each chunk.

The master/worker design pattern was implemented using the PThreads library. The master thread reads the input file and stores the decoded PCM data in the input ring buffer. If enough data for one chunk is read, the master creates a task struct and places it in the task queue, where it will be consumed by one of the worker threads. Worker threads encode their assigned chunk (including the previously mentioned ghost frames) and write their result into the output ring buffer. The master thread collects the finished task structs and writes the encoded result to the output file in the correct order. The team also planned to vectorize further code sections to improve the speedup. Some serial optimizations were also proposed.

The team used Intel VTune, Intel Inspector[17], Gprof and Scalasca[18] to profile the code. Manual scripts were written to automate the testing of the parallelized version as well as analysis of output files.

By week four of the project, Team D had a working parallel version of the LAME encoder. Vectorization of some code segments was included in this implementation. The team was able to achieve a significant speedup, but they had delayed sound in the start of the output file.

The team continued to optimize their implementation by vectorizing more code segments and implementing a scheduling policy. In the initial implementation, a static chunk size of 512 frames was used. The worker threads had to sit idle in the beginning of the program as the master thread created tasks for them serially. This problem could be solved by using a small chunk size but this led to a large number of ghost frames between chunks, hence increasing runtime and decreasing output quality. Figure 6 shows the runtimes achieved for a single input file for different versions of the parallelized project.

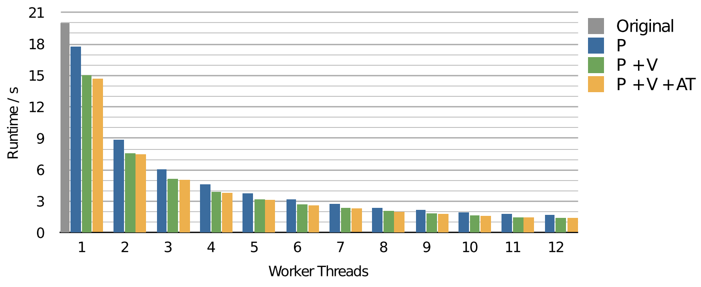

Fig. 6. Team D: Runtime plot for a sample input file: parallelized (P), parallelized & vectorized (P + V) and parallelized and vectorized using auto-tuned parameters (P+V+AT).

In order to achieve the best performance they decided to incorporate a scheduling strategy which dispatched tasks of different sizes comparable to the guided scheduling strategy used in OpenMP for the parallel execution of *for* loops. In the beginning, using a minimal chunk size, small tasks are created such that threads could start encoding with a relatively short waiting time. Given *n* worker threads, the chunk size was gradually increased after every *n* tasks until a maximum chunk size was reached. Thus, part of the input file was encoded without much overhead caused by ghost frames. Towards the end, the chunk size was again reduced until the last few created tasks had the minimal chunk size. These small tasks can be used to achieve a workload distribution at the end such that all threads end at approximately the same time, thus acheiving very good load balancing.

The last two weeks were used for bug fixing and fixing memory leaks discovered using Valgrind. The evaluation of the parallelized version and documentation was also done in those last two weeks. Team D developed a script to find the optimal parameters for their parallelized application. The results of their auto-tuning are shown in the Figure 6 as *P+V+AT*. At the end they were able to achieve an average speedup of 9.04 for the different types and sizes of input files. Their speedup surpassed the speedups achieved by all other three teams in the competition.

Team D summarized their work by reporting that the use of vectorization and load balancing in their project helped improve their speedup considerably. Still, they reported that the use of more resources by the master thread, serial overhead at the start and end of the program, interthread communication and computation of redundant ghost frames limited their speedup. Moreover, their results showed that the LAME encoder can benefit from serial optimizations.

V. Quantitative Evaluation

Quantitative comparisons between the teams working on libVorbis and LAME revealed some interesting points. Table III shows the total LOC without blank lines and comments for each of the teams, which are compared to the sequential versions of their respective projects. Compared to sequential LAME and libVorbis, the parallel versions vary by about 0.4% and relatively few lines express parallelism. The table also provides details about the number of lines modified, added and removed by the teams. For instance, it is clear that Team B had to modify and add fewer LOC to sequential libVorbis in comparison to Team A since they used the existing language constructs in TBB. Also, Team D modified and added the maximum number of LOC as they made effective use of the flexibility and better control over thread functionality provided by PThreads, and thus attained the best speedup.

Team D achieved an impressive speedup of 9.04 for 12 threads on an Intel Xeon X5670 CPU with 6 hyper-threaded cores. They also claimed that the benefits from their applied vectorizations could have been much more apparent if their implementation had been run on current Intel Haswell CPUs (Intel Xeon Phi co-processors with a vector width of 256bit / 512bit and direct SIMD table lookup support). These CPUs support the *Advanced Vector Extensions 2* (AVX2) instruction set which is an enhancement over Intels previous vector extension called *Streaming SIMD Extensions* (SSE). For libVorbis, the speedup of Team B was better (8.45) than Team A (5.98) for 12 threads on the same CPU.

The audio files for measuring the performance of the teams were taken from *Sound Quality Assessment Material recordings for subjective tests* (SQAM) benchmark [19] and were converted into the WAVE format. Each chosen file was encoded three times for all the specified number of threads and average speedup for all sample files was calculated. The comparison for both LAME and libVorbis can be seen in Figure 7 which clearly shows that Team D achieved the maximum speedup for 12 threads, followed by Team B, which was the better of the two libVorbis teams. All the teams attained their respective maximum speedup at 12 threads since the underlying architecture had 6 physical cores and supported *hyper-threading*. It is noticeable from Figure 7 that the speedup beyond 12 threads quickly drops off for libVorbis, but declines gradually for LAME.

Since the parallel encoders would not produce output files that are bitwise equal, all the teams were asked to encode the input wave file with a parallel and serial encoder using identical settings and report the comparison of the two output waveforms. A direct comparison of these two encoding results showed a maximum deviation of only 0.04% in all of the teams. This demonstrated that the encoding performed by the parallel encoders developed by the teams did not deteriorate the audio quality of the input files. In addition, any differences between the files encoded by the original encoder and the parallel encoder were inaudible.

TABLE III. LOC COMPARISONS FOR SEQUENTIAL LIBVORBIS AND LAME WITH THE PARALLELIZED VERSIONS OF THE FOUR TEAMS

Program	Total LOC[a]	LOC Modified	LOC Added	LOC Removed	Total effort in Person-Hours[b]
Sequential libVorbis	**54725**	-	-	-	-
Team A	56120	890	1398	3	∼4*50
Team B	55138	6	519	106	∼4*50
Sequential LAME	**41197**	-	-	-	-
Team C	41885	31	695	7	∼3*60
Team D	42894	126	1843	146	∼3*60

[a] Without blank lines or comments.
[b] ∼n*x represents the effort of n people for x hours each.

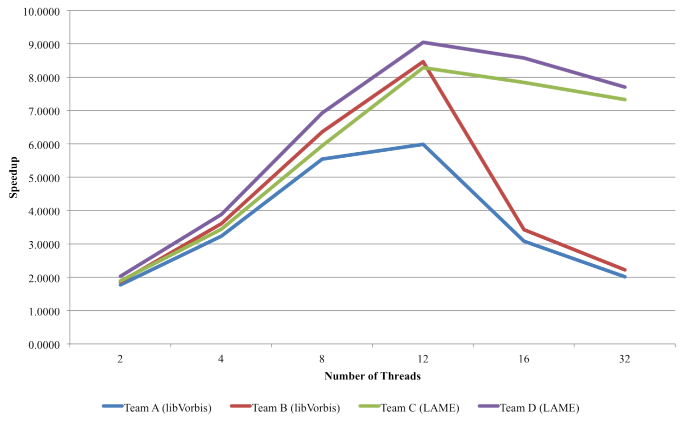

Fig. 7. Comparison of average speedup for libVorbis and LAME.

The difference between the size of the output files generated by the sequential encoder and the parallelized encoder of each team was also negligible. The maximum value for this difference among all the teams was found to be 0.112%, for one of the files from Team B.

In addition to the speedup achieved and the quality of their audio output, the teams were also given points based on effective use of profiling techniques, cache friendliness of the code and the number of deadlocks and data races. All the teams were asked to provide reports of their profiling tools. They were also rewarded points for certain code metrics that included LOC/(Lines of comments), coding style, cyclomatic complexity and so forth. Team D won the competition because of the best overall score.

VI. Lessons Learnt

We offer seven lessons learned from our experience.

A. *Best Pedagogical Methods*

This course provided a good opportunity for us to understand and experience the difficulties and challenges in teaching parallel programming to graduate computer science students. Most of our students had less experience of parallel programming. We learned that our students could develop reasonable parallel applications in a short time span. The students needed a hands-on experience of the parallel programming and the purposeful exercises provided after each programming model lecture helped them learn parallel programming quite quickly in a practical way. So, the intensive training provided to them in first four weeks of the course was sufficient for them to adequately parallelize the projects.

We set up four phases of the project and set some goals and milestones for the teams. Each phase was allocated a specific time and the teams were asked to submit their progress reports and provisional results at the end of each phase. These phases were: analysis of the existing sequential code, drawing a parallelization strategy and algorithm, development of a parallel version of the assigned application and validation and optimization of the developed parallel application. If a team completed one phase earlier than the proposed milestone, they could start working on the next phase. None of the teams missed any deadlines during the project. We interviewed our students and majority of them admitted that setting up goals with specific time constraints encouraged them to be more focused and motivated to achieve those specific goals. For us as tutors of the course, it was helpful to know the progress of each team after every milestone. We could verify the results of each team in a timely manner and provide feedback to them, instead of getting all of them together at the end of the project.

B. *Choose the appropriate Programming Model*

Before parallelizing the sequential program, the first question to ask oneself would be which parallel programming model should be chosen. The answer to this question depends on many aspects including the language used to write the sequential program, the execution environment of the target platform and the performance constraints. During the course, we noticed that one factor is very important for answering this question - understanding the program.

There is a wide range of legacy sequential applications. Most programmers, while parallelizing such a legacy application, do not tend to spend much time understanding the code and its characteristics. They usually choose a parallel programming model based on direct and objective constraints. However, we found that the time spent on understanding the code is very important. Of the two teams that worked on libVorbis, Team A decided to use OpenMP, since they thought it would be the easiest way to parallelize the code incrementally. The decision was made solely based on the features of programming models; the characteristics of the sequential program were not considered. In contrast, Team B, after spending some time reading the code, found that libVorbis works in a pipeline style. After discovering this, they believed that choosing TBB would be more convenient. Team A learned this lesson the hard way. After a few weeks, Team A noticed the same pipeline in libVorbis. However, they had already written the majority of their code and they did not have enough time to start over with a new programming model. Therefore, their application resulted in a worse speedup compared to Team B in the final competition. The programming skills of the members of Team A were quite impressive. However, they wasted a lot of time and paid the price of not understanding the code first.

C. Coarse-Grained Parallelism First

Fine-grained parallelism is an important part of parallelizing sequential code. However, the reports of the teams showed that replacing a small sequential computation routine with its parallel version did not significantly improve the performance, because the fraction of the runtime of parallelized code was too small compared to that of the remaining sequential code. In contrast, the most significant speedup came from coarse-grained parallelism. Team A, Team B and Team C concentrated on coarse-grained parallelism only, and achieved a speedup of 5.98, 8.45 and 8.28 with 12 threads respectively. Team D also concentrated on coarse-grained parallelization first. Once they had a parallel version, they then started adding fine-grained parallelization which led to a speedup of 9.04 with 12 threads. The improvement in speedup of Team D over Team C is only 9%. This clearly shows that to achieve a good speedup quickly, one should explore coarse-grained parallelism first.

Meanwhile, Team A also mentioned that the portion of fine-grained parallelism heavily depends on algorithm design and implementation. An algorithm designed and implemented by following the traditional sequential programming's golden rules in order to save time and memory, could result in massive dependencies between operations. Thus in some cases, fully utilizing fine-grained parallelism means redesigning the algorithm. Obviously, this requires more time to analyze the code and understand the program. As a result, we highly recommend focusing on coarse-grained parallelism first.

D. Look for Dependency Relaxations

Sometimes sequential applications cannot be parallelized due to data dependencies. Based on the specifications of the application, one can relax some of these dependencies, opening up opportunities for parallelization to achieve speedup.

During parallelization of the LAME encoder, teams C and D realized that although each MP3 frame depends on the flags set by the previous frame, the dependency can be relaxed. They divided the input file into chunks and used the idea of processing "ghost points" from fluid simulation problems on the border of the chunks. This relaxation of the dependencies along the border of chunks resulted in average speedups of 8.45 and 8.3 for Team C and D respectively. On the other hand, the effects of this dependency relaxation on the output were not very significant. There was no audible noise or loss of quality in the outputted MP3 file. Therefore, it is advisable to concentrate on the specifications of an application before starting the parallelization and to look for relaxable dependencies not only along the critical path but also beyond it in order to extract more parallelism.

E. Look Beyond Predefined Language Constructs

Programmers tend to find parallelism in predefined language constructs such as functions and loops. It is natural to focus first on dependencies between the existing language constructs and thus to parallelize them. But there may be potential hidden parallelism available beyond these predefined language constructs. Concentrating on code blocks outside of these constructs and resolving dependencies between them may reveal more parallelization opportunities. This can result in the extraction of new tasks or discovery of new parallel design patterns.

Team A tried to parallelize libVorbis incrementally using OpenMP directives on the existing constructs, i.e. loops. They could only achieve a minor speedup with their approach. On the other hand, team B used a comprehensive approach and looked beyond the predefined language constructs. They discovered a pipeline pattern in the same application and transformed the code accordingly to achieve a better speedup. The additional speedup achieved by team B over team A shows the benefits of their approach. Similarly, teams C and D both analyzed the code of LAME extensively and discovered tasks beyond the predefined language constructs. They transformed the code to implement a pipeline pattern and a master/-worker pattern respectively. Both of the teams achieved similar speedups. It is recommended to try to identify parallelism between code sections of arbitrary granularity and not to rely on predefined language constructs for parallelization.

F. Black Magic or Not, Depends on Experience

In [20], the authors argued that parallel programming is not a black magic. The graduate students who had only one semester's experience with parallel programming did not find it overwhelming. However, it does not mean that the art of parallel programming can be managed quickly. One may be able to write parallel programs after a short training period, but writing high quality programs requires a lot of practice.

While evaluating the results of the teams, we found that the work of Team D was superior to that of the other teams. They chose PThreads since it provides the flexibility to control every aspect of the program. The sequential code was refactored precisely in accordance with their parallelization ideas. Fine-grained parallelism, where a single operation is too small to be assigned as a task, were accomplished by performing a batch of vectorized operations. On a high-level, audio frames were processed in a master/worker pattern constructed by the team, and both load balancing and pipeline efficiency were considered. In addition, to ensure that the encoded audio file did not lose quality at the connecting points of two audio frames, they added "ghost frames" at the borders. Finally, they even provided an auto-tuning script to decide the best parameters for different platforms in order to ensure that the best speedup could be achieved. Their design and implementation led to linear speed up in some individual test cases, and super linear speedup in the others. After a short interview, we found out that one of the team members had some parallel programming experience before joining the lab. Parallel programming is not a black magic, but it is still much more complex than sequential programming and more difficult to learn. Thus, practice in parallel programming is even more important than in sequential programming.

G. The Need for Assisting Tools is Still Urgent

It is commonly understood today that tools for assisting parallelization are very important. A lot of tools have been developed, some of which are commercial. During the course, we introduced many tools aimed towards assisting the programmer to write efficient parallel programs, such as Valgrind [2], Intel Parallel Studio, TotalView[13] and ThreadSanitizer [14]. Some of the teams also tried the program-analyzing tools provided by Microsoft Visual Studio. However, based on the results of interviews, only a few students said that they received any help from the tools and no one thought that the help these tools provided was sufficient. In fact many times, students complained about the meaningless output of the tools. For example, tools for checking the race conditions produced hundreds, even thousands, of warnings for a small program, and most of the warnings were false positives. One of the most frequently asked questions was "How can I get the data dependencies of the program/function?", and none of the tools provided ready-to-use results. The tools that were considered helpful by the students were still the same as those for finding hot spots (gprof-like) and memory leaks (Valgrind-memcheck, etc.) and are more mature. Tools for assisting programmers with writing efficient parallel programs are still urgently needed and the existing tools such as DiscoPoP [21], [22], [23] are mainly in the initial stages of development.

VII. Related Work

Parallel programming courses have been taught all over the world for quite some time now. Lin [24] reported the results from a parallel and distributed computing course. They taught a variety of topics including parallelization in Java, locks, cache coherence, memory architectures and many more. Their students reported that the topics covered in the course were too much for an introductory class. Also programming in multiple languages made the contents complex. Multiple labs were conducted to assess the knowledge of the students and the best passing rate was 67% in one of these labs. Our course was specific to C/C++ language only and we introduced only the most important programming models and topics to our students that helped them focus on parallelizing the projects more efficiently.

Many universities offer parallel programming course for undergraduate students. Yazici et al. [25] discussed the importance of using analogies for explaining parallel programming concepts to the students. They reported that the use of simple real-life examples as analogies helped the students to successfully design parallel algorithms. Their assessment demonstrated that a student's grades achieved in their course were highly related to the programming background and skills of that student. This verifies our claim that writing high quality parallel applications need a lot of practice.

Majority of the fields in science are evolving rapidly by the introduction of complementary fields. The use of parallel computers in almost every field of science have made parallel programming a recommended course for all scientists. Courses have emerged to cater the needs of such interdisciplinary fields [26]. Students in these courses come from different backgrounds (physics, chemistry) and with experiences in different languages (Python, Java, etc). At the very first step, the tutors need to teach them a basic common programming language such as C/C++, so that everybody has the same knowledge before continuing to learn the actual parallel programming lessons. For our case study, all of our students were from computer science field and were familiar with C/C++ programming. This helped us to concentrate more on the parallel programming concepts.

Pankratius et al.[20] conducted a case study of parallelizing Bzip2 application by graduate students. They reported that the students with the background of computer science found it difficult to parallelize Bzip2. On the contrary, our students did not have much difficulty in parallelizing LAME and libVorbis. We observed that the profiling and supportive programming tools used by the students have become more sophisticated and available than the they were at the time of their case study. This helped the students to find appropriate strategies to parallelize the applications and locate the important hotspots. However, progress is still needed in the tools to help writing efficient parallel programs.

Teaching parallel programming by giving the students a hands-on experience with real world applications is becoming popular. Breuer and Bader [27] used shallow-water equations to train their students. They have developed a code package to teach parallel programming in multitude of courses. Similarly, many other courses use the same strategy [28], [29], [30]. All of them emphasize that the students like to get experience of parallelizing a real world application instead of some toy and sample applications. Such courses help them understand the difficulties and challenges of parallel programming that can help them in their career and professional life.

VIII. Conclusion

In this paper we presented a case study that focused on parallelizing two real world applications, libVorbis and LAME, in order to explore and better understand the effective way of teaching parallel programming and the strategic choices that programmers need to make while parallelizing an application. We introduced Pthreads, OpenMP and TBB to the students in the beginning together with mandatory exercises for hands-on experience of these programming models. In the project phase, the students were devided into four teams. Team A and B competed to parallelize libVorbis. Team A extended the C-interface of libVorbis using the OpenMP *task* construct and were able to get an average speedup of 6.0 with the help of hyper-threading. Team B decided to implement the pipeline pattern using the flowgraph construct of TBB and achieved an average speedup of 8.5. Team C and D competed to parallelize the LAME encoder. Team C decided to use OpenMP to encode each section of the input file separately and managed to get an average speedup of 8.3. Team D chose the master/worker design pattern using the PThreads library to implement their strategy to encode each section of the input file. They performed local optimizations on their code like vectorization and also implemented a dynamic scheduling strategy. Team D won the competition with an average speedup of 9.04 and were the only team to successfully exploit both fine and coarse-grained parallelism.

We summarize our experience of the course in six lessons: Firstly, we recommend using purposeful and targeted exercises for the initial training of students together with setting time-based goals during the project phase. While parallelizing, we highly suggest reading and understanding the program before choosing the parallel programming model. We also found that coarse-grained parallelism is more useful as well as easier to implement mainly because of the use of high-level parallel programming constructs. Understanding the specifications of an application and relaxing some dependencies based on these specifications may help find more parallelism. This means that one has to look beyond the predefined language constructs and try to resolve dependencies between arbitrary code sections to exploit various parallel patterns. Although parallel programming is not a black magic, we still observed a significant difference between the work of experienced programmers and of the graduate students with only one semester training. Finally, the students expected to get ready-to-use results from the tools for paralleli zing a program. Therefore, the need for such assisting tools is still urgent and the existing tools need to be improved.

Acknowledgments.: We would like to thank our course students for their support. They worked hard and provided us with all the necessary information for this case study.

References

[1] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806987

[2] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[3] [Online]. Available: http://xiph.org/vorbis/doc/libvorbis/overview.html

[4] [Online]. Available: http://lame.cvs.sourceforge.net/

[5] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[6] [Online]. Available: http://openmp.org/wp/openmp-specifications/

[7] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[8] [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe

[9] R. M. Stallman and C. Support, *Debugging with GDB : The GNU source-level debugger, GDB version 4.16*. Boston, MA: Free software foundation, 1996. [Online]. Available: http://opac.inria.fr/record=b1104446

[10] A. Jannesari and W. F. Tichy, "Identifying ad-hoc synchronization for enhanced race detection," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –10.

[11] ——, "Library-independent data race detection," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 10, pp. 2606–2616, Oct. 2014.

[12] A. Jannesari, M. Westphal-Furuya, and W. F. Tichy, "Dynamic data race detection for correlated variables," in *Proc. of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ser. ICA3PP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 14–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075416.2075421

[13] C. Gottbrath and P. Thompson, "Totalview - totalview tips and tricks." in *SC*. ACM Press, 2006, p. 9.

[14] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71.

[15] M. S. Mller, A. Knpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace," in *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jlich and RWTH Aachen University, Germany, 4-7 September 2007*, ser. Advances in Parallel Computing, C. H. Bischof, H. M. Bcker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, Eds., vol. 15. IOS Press, 2007, pp. 637–644.

[16] J. Weidendorfer, "Sequential performance analysis with callgrind and kcachegrind." in *Parallel Tools Workshop*, M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer, 2008, pp. 93–113.

[17] [Online]. Available: http://software.intel.com/en-us/intel-inspector-xe

[18] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "A scalable tool architecture for diagnosing wait states in massively parallel applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, Jul. 2009.

[19] "Sound quality assessment material recordings for subjective tests (sqam)." [Online]. Available: http://tech.ebu.ch/publications/sqamcd

[20] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *IEEE Softw.*, vol. 26, no. 6, pp. 70–77, Nov. 2009.

[21] Z. Li, A. Jannesari, and F. Wolf, "An efficient data-dependence profiler for sequential and parallel programs," in *Proc. of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India*. IEEE Computer Society, May 2015, pp. 484–493. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.41

[22] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf, "Discopop: A profiling tool to identify parallelization opportunities," in *Tools for High Performance Computing 2014*. Springer International Publishing, Aug. 2015, ch. 3, pp. 37–54. [Online]. Available: http://www.springer.com/us/book/9783319160115

[23] Z. U. Huda, A. Jannesari, and F. Wolf, "Using template matching to infer parallel design patterns," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, pp. 64:1–64:21, Jan. 2015.

[24] H. Lin, "Teaching parallel and distributed computing using a cluster computing portal," *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, vol. 00, no. undefined, pp. 1312–1317, 2013.

[25] A. Yazici, A. Mishra, and Z. Karakaya, "Teaching parallel computing concepts using real-life applications," *INTERNATIONAL JOURNAL OF ENGINEERING EDUCATION*, vol. 32, no. 2, pp. 772–781, 2016.

[26] E. Cesar, A. Cortés, A. Espinosa, T. Margalef, J. C. Moure, A. Sikora, and R. Suppi, *Teaching Parallel Programming in Interdisciplinary Studies*. Cham: Springer International Publishing, 2015, pp. 66–77. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27308-2_6

[27] A. Breuer and M. Bader, "Teaching parallel programming models on a shallow-water code." in *ISPDC*, M. Bader, H.-J. Bungartz, D. Grigoras, M. Mehl, R.-P. Mundani, and R. Potolea, Eds. IEEE Computer Society, 2012, pp. 301–308. [Online]. Available: http://dblp.uni-trier.de/db/conf/ispdc/ispdc2012.html#BreuerB12

[28] G. Aloisio, M. Cafaro, I. Epicoco, and G. Quarta, "Teaching high performance computing parallelizing a real computational science application," in *International Conference on Computational Science*. Springer, 2005, pp. 10–17.

[29] K. Claypool and M. Claypool, "Teaching software engineering through game design," in *ACM SIGCSE Bulletin*, vol. 37, no. 3. ACM, 2005, pp. 123–127.

[30] R. Keller, "Teaching parallel programming to undergrads with hands-on experience," in *Workshop on Parallel, Distributed, and High-Performance Computing in Undergraduate Curricula (EduPDHPC) in conjunction with Sc-13: The International Conference for High Performance Computing, Networking, Storage, and Analysis*, Denver, CO, USA, 2013, pp. 1–8.