A Course-Based Usability Analysis of Cilk Plus and OpenMP

Michael Coblenz[1], Robert Seacord[2], Brad Myers[1], Joshua Sunshine[1], and Jonathan Aldrich[1]

[1] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{mcoblenz, bam, sunshine, jonathan.aldrich}@cs.cmu.edu

[2] Secure Coding Institute
600 Tivoli Drive
Gibsonia, PA, USA
rcs@securecodinginstitute.com

***Abstract*—Cilk Plus and OpenMP are parallel language extensions for the C and C++ programming languages. The CPLEX Study Group of the ISO/IEC C Standards Committee is developing a proposal for a parallel programming extension to C that combines ideas from Cilk Plus and OpenMP. We conducted a preliminary comparison of Cilk Plus and OpenMP in a master's level course on security to evaluate the design tradeoffs in the usability and security of these two approaches. The eventual goal is to inform decision-making within the committee. We found several usability problems worthy of further investigation based on student performance, including declaring and using reductions, multi-line compiler directives, and the understandability of task assignment to threads.**

Keywords—API usability, empirical studies of programmers, parallel programming, OpenMP, Cilk Plus

I. Introduction

Now that multicore machines are widely available, it is particularly important to provide programming models that enable users to take advantage of multiple cores without introducing programming errors. By 1989, there was significant work on programming languages to make parallel programming easier [1], and many other researchers have investigated the usability tradeoffs in the design of parallel programming languages in general (see §V). Although the versions of the ISO/IEC standards for C and C++ that were published in 2011 provided support for multithreaded programs, it was clear that additional extensions were necessary for programmers to effectively make use of modern processors. As a result, ISO/IEC JTC1/SC22/WG14, the international standardization working group for the programming language C, formed the C parallel language extensions (CPLEX) study group.

Cilk Plus [2][3] and OpenMP [4][5] are language extensions for parallel programming in C and C++. Cilk Plus provides a set of keywords and OpenMP provides a set of pragmas that programmers can use to instruct the compiler to parallelize their programs. For example, Cilk Plus provides `cilk_for`, which instructs the compiler to perform the iterations of a `for` loop in parallel. CPLEX is currently evaluating and harmonizing these language extensions. CPLEX's goal is to produce a standard or technical specification that explains how the proposed extension will work for both C and C++. (Note that although OpenMP addresses multiple languages, such as C++ and FORTRAN, our study only examined the C interfaces.)

Many questions must be resolved in the process of developing a standard approach to parallel programming for C. Cilk Plus provides a small set of language keywords: `cilk_for`, `cilk_spawn`, and `cilk_sync`. In contrast, OpenMP makes use of `#pragma` statements to support parallelization capabilities such as critical sections and fine-grained control for assigning work to threads. OpenMP seems to be the more popular of the two interfaces, possibly because the use of pragmas allows the code to be compiled and serially executed by implementations that do not support OpenMP. We have been working to provide data to the CPLEX Study Group on the *usability and correctness* tradeoffs of these approaches. In particular, we were asked which language extension was most effective at helping programmers avoid errors that could potentially lead to safety or security concerns and we have received feedback from the study group that the information we have provided to them (and discussed in this paper) is useful.

Although the two language extensions provide significantly different features and expose them in different ways, they both support shared-memory fork-join parallel computation. Shared-memory parallelism involves multiple threads of execution sharing one memory space (in contrast to message-passing parallelism, in which data are shared by passing messages explicitly). Fork-join parallelism involves specifying portions of the program that run in parallel. When encountering a parallel portion of the program, control is forked into multiple threads of execution; when the parallel portion is complete, the threads join together and the program continues with a single thread of execution.

Our study focuses on the APIs for defining and using *reducers* because they are an interesting and important part of both language extensions. Reducers combine partial results in fork-join parallel programs. For example, consider a `for` loop that is parallelized with `cilk_for`, where the loop iterates over a set of inputs and stores results in an accumulator. The runtime computes how many iterations will be performed in the loop (Cilk Plus and OpenMP do not allow the loop counter to be modified inside the loop body of a parallel loop) and divides the iterations among threads. Each thread accumulates results in its own private copy of the accumulator. Then, the results from threads are *reduced* in pairs until all the results are combined in a single accumulator.

As a starting point to investigating the *usability and correctness* tradeoffs of parallel programming, we focused on *learnability* of the approaches in a classroom. Although some prior research has evaluated *performance gains* of parallel programs, our study focuses on the ability of programmers to learn how to write correct code that still achieves a significant speedup from parallelism. Although this initial study only included eight students, we were able to gather interesting data that may inform standards development and guide future studies. The main contributions of this paper are the identification of potential usability problems in Cilk Plus and OpenMP that may be addressed by the design of a unified standard for parallel programming in C, along with an exploration of issues that students face when learning parallel programming.

II. Method

We recruited participants from a master's-level course on Secure Coding taught by Robert Seacord, the second author. The study was part of the students' final homework, at which point there were nine students in the class. The homework required completing the same programming task twice: once using OpenMP and once using Cilk Plus. We randomly chose five of the nine students in the class to do the task in OpenMP first and the rest to do the task in Cilk Plus first. Eight students submitted the assignment for grading and all of them agreed to participate in the study. Before assigning the programming task to the students, the instructor gave lectures on parallel programming and both Cilk Plus and OpenMP, providing code examples and API explanations. Similar class time was spent on each extension. The teaching and study materials are available at http://www.cs.cmu.edu/~mcoblenz/vlhcc2015/.

To focus the students on the problem of writing parallel code and not on algorithm design, we chose the simple task of parallelizing an existing (serial) implementation of a program that finds anagrams. The students were instructed to use the C APIs for both language extensions. The provided serial implementation included a recursive function that permuted a string in a buffer and, for each permutation, did a lookup to determine whether that string was in a dictionary. The students were instructed to not change the algorithm, as there are much more efficient ways of generating anagram lists, and we did not want changes in the algorithm to affect performance.

We provided the students with a virtual machine image with Ubuntu 14.04 and Eclipse 4.4.2 pre-installed. Eclipse was configured with two projects: one for OpenMP and one for Cilk Plus. The OpenMP project used gcc 4.9.2, which supported OpenMP 4.0. Because gcc 4.9.2 did not include support for `cilk_for`, we configured the Cilk Plus project to build with a branch of clang 3.4.1 that included support for Cilk Plus. Each project included a serial implementation and driver code that called the serial implementation, a parallel implementation that just called the serial implementation, and timing code to report the speedup, that is, the ratio of serial time to parallel time. In addition, we preinstalled Fluorite [6] so students who participated in research could send us editing logs for analysis. We received editing logs from all students.

We graded the assignment on the basis of correctness and performance; we required students to achieve a speedup of at least 1.5 on a dual-core system and awarded additional points for the top three speedups in each programming task and for any speedups above 1.5. In addition, because we wanted students to learn how to use reducers and because we wanted to study how successful students were at using reducers, we required students to use reducers with both language extensions. We deducted points for incorrect or unsafe results; for failure to use reducers; and for failure to achieve speedup goals. After the assignment was submitted and graded, we asked the students to answer some retrospective questions so we could learn more about their experiences.

III. Results

Eight students each submitted two programs. The results are summarized in Table 1.

Table 1: Summary of Results

n = 8	*OpenMP*	*Cilk Plus*
Number of correct programs	3	5
Average speedup	1.2	1.5
Speedup standard deviation	0.9	0.6
Number of programs with speedup at least 1.5	2	4

A. Correctness

Although all of the solutions used a correct algorithm, many of the solutions included race conditions, which affected correctness. Of the eight OpenMP solutions, only four attempted to use a reducer even though the assignment explicitly required it. Of those four, one did not actually use the reducer: although the student declared the reducer, a different accumulator was used to accumulate the results concurrently in multiple threads. The remaining three were thread-safe. Of the four submissions that did not use reducers, one used `#omp critical`, which specifies that the annotated code is a critical section, but failed to include the critical section in a block, so that only the first statement in the critical section was protected by the directive. As a result, this program contained a race condition. Logs for two of the other non-reducer submissions indicated that those students attempted to use reducers, but could not figure out how to use them and deleted their reducers before submission.

All of the eight Cilk Plus submissions attempted to use reducers, but one student did not actually use the reducer because, although the reducer was declared, the results were accumulated in an unprotected, shared accumulator. This bug

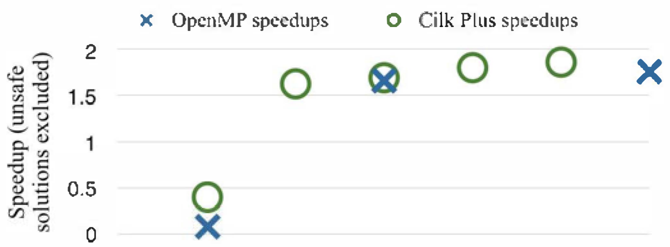

Fig. 1. Speedups by parallel language extension. Each column represents one student.

was analogous to a similar mistake in OpenMP, although different students made the mistake. Additionally, another student called `REDUCER_VIEW` on the reducer outside the parallel region, resulting in all parallel threads accessing the same reducer, which is unsafe. The other six students made safe use of the reducer, although one incorrectly shared the input across multiple threads (because the threads concurrently permuted the input in place).

The most common mistake in Cilk Plus was neglecting to free memory allocated in the reducer; although the Cilk Plus API for declaring reducers requires specifying a function to destroy the reducer, five of the eight submitted solutions used a function `that` specifies that nothing need be done to destroy a reducer. Although students presumably found this approach in sample code, it results in a memory leak in this case. A similar bug existed in OpenMP solutions as well, but we did not deduct points for it in that case because the documentation does not explain how to manage memory correctly and we did not discuss it in class. By default, OpenMP initializes reducers by copying from an initial reducer; students got lucky because this happened to work due to the implementation details of the particular accumulator included by the starter code.

Six of the eight Cilk Plus submissions declared reducers in global variables even though there was no need for the reducers to be in the global scope. This may be because available sample code used this approach and because the students may have been unaware of the problems with global variables, such as the obstacles they present to modularity [7].

B. *Performance*

Performance results are summarized in Fig. 1. Four of the five correct Cilk Plus solutions and two of the three correct OpenMP solutions achieved a speedup of at least 1.5. The correct solutions that did not achieve a speedup were slowed down by recursively allocating reducers and recursively using parallel for loops. Although with two cores an optimal solution would have generated approximately two tasks, recursively using `cilk_for` or `#pragma omp parallel for` resulted in dividing the problem into an exponential number of tasks with an exponential number of reducers. One student made this error with both language extensions, achieving a performance of 0.08x the serial time on OpenMP and 0.4 on Cilk Plus. Excluding this student and the unsafe solutions, the average speedup on OpenMP was 1.72, and the average speedup on Cilk Plus was 1.75.

C. *Time spent on task*

We analyzed the log files that Fluorite generated to estimate time spent working on the tasks (see Table 2). To avoid counting time spent taking breaks, we only included times between pairs of Eclipse commands that were less than two standard deviations above the mean time between commands.

TABLE 2: ESTIMATED TIMES SPENT ON TASKS (MINUTES)

	OpenMP average task time	*Cilk Plus average task time*
OpenMP-first students	675	210
Cilk Plus-first students	158	557

Based on these estimates, we observed a large learning effect of the first task on the second task. OpenMP took longer if it was first, but shorter if it was second. This could mean that Cilk Plus has a lower barrier to entry and teaches students more so that they can do OpenMP more easily. However, with such a small sample, we cannot eliminate the possibility that the Cilk Plus-first students were just better programmers.

D. *Student Opinions*

After the assignment was complete, we asked students to reflect on their experience. Only three students replied, and one would use Cilk Plus in a future project if given a choice; one would use OpenMP; and one would use standard thread libraries to do parallel programming manually. However, all three students said they liked Cilk Plus better for the homework, with some saying it was easier and the others saying it was simpler. Two of the students suggested providing more material on Cilk Plus and OpenMP, including more examples and practice problems.

IV. DISCUSSION

A. *Implications*

Although correct solutions in both systems were of similar performance, the number of correct solutions varied, with five correct Cilk Plus solutions but only three correct OpenMP solutions. The difference pertained primarily to the successful use of reducers. We believe that Cilk Plus's mechanism for defining reducers is more usable than OpenMP's. One possible cause is that although the Cilk Plus syntax for defining a reducer is similar to that of a function call, OpenMP uses a different syntax. For example, an OpenMP reducer might be defined as follows:

```
#pragma omp declare reduction
    (results_reduction :
        results_t :
        results_reduce(&omp_out, &omp_in))
  initializer(results_init(&omp_priv))
```

whereas a Cilk Plus example might be:

```
CILK_C_DECLARE_REDUCER(results_t)
    my_results_reducer =
          CILK_C_INIT_REDUCER(
                results_t, reduce,
                identity, destroy);
```

The Cilk Plus approach specifies all the relevant arguments in one construct that looks like a function call. In contrast, the OpenMP approach uses colons to delimit arguments, which is not standard in C. In addition, how to correctly use OpenMP's initializer is not described in the examples that could be easily found online. Only one of the students included an initializer clause, and that student found out about it by asking the course staff. Finally, one must read documentation to discover the predefined OpenMP variables, `omp_in` and `omp_out`, and their types. In contrast, Cilk Plus's mechanism for initializing a reducer, `CILK_C_INIT_REDUCER`, takes a function name, not a function call, so users do not need to be aware of the names of the inputs. The CPLEX proposed reduction syntax is closer to Cilk Plus than to OpenMP.

In both Cilk Plus and OpenMP, there were solutions that declared a reducer but failed to use it properly. One way of minimizing the chances that users will use a reducer incorrectly is to pass whole reducers rather than accumulators, which would be of distinct type from the reducers. In Cilk Plus, for example, a `typedef` for `CILK_C_DECLARE_REDUCER(T)` produces a type that is distinct from `T`. If the programmer inadvertently passes an accumulator instead of a reducer, the compiler can give an error.

In Cilk Plus, `REDUCER_VIEW(reducer)` returns an object of the accumulator type, and if this is called outside a parallel region, the correct thread-specific reducer is not used. A better language design is to allow the reducer to be passed as an argument and have the compiler infer (due to the implicit type conversion) when the thread-specific reducer is appropriate. Then, the language and runtime might be able to infer which instances of a reducer should implicitly get the thread-specific accumulator and which are actually passing the whole reducer. It would help if the compiler warned when it could not be sure that a given call to `REDUCER_VIEW` was always inside a parallel context; however, this analysis would need to be conservative and might give spurious warnings.

Overall, we find the difficulty our master's students had in writing correct parallel code using these tools very discouraging. Despite the simplicity of the task, the instruction that was designed to prepare them to complete it, and their prior academic background, many students submitted solutions with race conditions. This suggests that if we intend to have computer science graduates write correct parallel software, we will need new instructional techniques, new tools, or both.

B. *Limitations*

This preliminary study was not intended to represent the full range of use cases for either language extension; instead, it was intended to generate hypotheses and inform future studies. External validity may be limited by participants being novices at both Cilk Plus and OpenMP; by the short duration of the study and the simplicity of the programming task; and by the specific instructions that the students received before beginning their work. Construct validity may be limited by the instruction the students received; by the example and starter code the students were given; by the small number of participants; by the students' apparent lack of understanding of race conditions (as evidenced by the large number of solutions that included unprotected access to shared variables); and by an erroneous inclusion in the OpenMP starter code, though we believe this did not affect the results. Finally, the grading process in large part involved manual inspection of student code, so it is possible that some errors were not found, but this risk is mitigated by the small task size and low code complexity.

C. *Summary of Hypotheses*

We hypothesize that inconsistent and complex reducer syntax in OpenMP impedes programmers; giving reducers distinct types from their values would reduce error rates; and reducers and parallel language extensions in general do not obviate the need for programmer understanding of concurrency, but education can increase the chances of success.

V. Related work

The problem of designing programming languages for parallel programming is a long-studied area. In 1989, Bal, Steiner, and Tanenbaum [1] published a survey of over 300 programming languages for distributed systems. Although their paper predated both Cilk and OpenMP, it catalogued issues pertaining to the design of parallel programming systems, such as the question of message-passing versus shared-memory systems, and explained how numerous languages addressed the issues. Wilson proposed a set of programming problems for use in evaluating the usability of parallel programming systems [8]. However, in our study, for the students to focus on the parallelization and not on the overall problem, we chose an algorithmically much simpler problem.

Sadowski and Shewmaker argued for more research into usability of parallel programming systems in a position paper [9], finding that most of the research to date had been inconclusive but that parallel programming has been shown to be hard. They identified five challenges in measuring programmer productivity in parallel programming, such as the fact that one must target usability for a particular kind of programmers.

Some researchers have conducted comparisons between parallel programming languages. Nanz, West, da Silveira, and Meyer [10] used one experienced programmer to implement solutions to problems in Chapel, Cilk, Go, and Threading Building Blocks, and then had experts in each language evaluate the programmer's work. In contrast, our study focused on novice programmers; though this exposes different kinds of issues, our paper has the advantage of including more than one user. Hochstein, Basili, Vishkin, and Gilbert compared message-passing and PRAM-like models (PRAM is similar to a shared-memory system, but the processors execute iterations of parallel loops *synchronously*) with students in a course [11], finding that PRAM-like programs required less effort to write but were not significantly more correct.

VI. Conclusions

We conducted a course-based experiment with eight master's students, who each parallelized a simple program using reducers in both Cilk Plus and OpenMP. We observed that although only three of eight students wrote thread-safe OpenMP solutions, five did so in Cilk Plus, leading to a hypothesis that students are more likely to develop thread-safe reductions in Cilk Plus, but this hypothesis should be tested further. We identified several avenues for future research and for consideration in the design of a C parallel language extension, including issues pertaining to syntax and types. We also identified ways of improving our course content pertaining to parallel programming, including improving available examples and ensuring students have a better general understanding of race conditions and memory management.

Acknowledgments

This material is based upon work funded and supported in part by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center, in part by the NSA lablet contract #H98230-14-C-0140, and in part by NSF grant CNS-1423054. This material has been approved for public release and unlimited distribution. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

REFERENCES

1. Bal, H. E., Steiner, J. G., & Tanenbaum, A. S. (1989). Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), 261–322. http://doi.org/10.1145/72551.72552
2. Intel (2015, May 11). Intel® Cilk™ Plus Language Extension Specification Version 1.2 (2013-09-06) Available: https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm
3. Intel (no date). A Brief History of Cilk. Available: https://www.cilkplus.org/cilk-history
4. OpenMP Application Program Interface (2015, May 11). OpenMP: The OpenMP API Specification for Parallel Programming. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
5. Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 46–55. http://doi.org/10.1109/99.660313
6. Yoon, Y., & Myers, B. A. (2011). Capturing and analyzing low-level events from the code editor. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools—PLATEAU '11* (p. 25). New York, USA: ACM Press. http://doi.org/10.1145/2089155.2089163
7. Wulf, W., Shaw, M. (1973) Global variable considered harmful. ACM Sigplan notices 8(2), 28-34
8. Gregory V. Wilson, R. B. I. (1995). Assessing and Comparing the Usability of Parallel Programming Systems. Technical Report. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4614
9. Sadowski, C., & Shewmaker, A. (2010). The Last Mile. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research—FoSER '10* (p. 309). New York, USA: ACM Press. http://doi.org/10.1145/1882362.1882426
10. Nanz, S., West, S., Silveira, K. S. da, & Meyer, B. (2013). Benchmarking Usability and Performance of Multicore Languages. In 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 183–192). IEEE. http://doi.org/10.1109/ESEM.2013.10
11. Hochstein, L., Basili, V. R., Vishkin, U., & Gilbert, J. (2008). A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, *81*(11), 1920–1930. http://doi.org/10.1016/j.jss.2007.12.798