Examining the Expert Gap in Parallel Programming

Sebastian Nanz[1], Scott West[1], and Kaue Soares da Silveira[2]

[1] ETH Zurich, Switzerland
`firstname.lastname@inf.ethz.ch`
[2] Google Inc., Zurich, Switzerland
`kaue@google.com`

Abstract. Parallel programming is often regarded as one of the hardest programming disciplines. On the one hand, parallel programs are notoriously prone to concurrency errors; and, while trying to avoid such errors, achieving program performance becomes a significant challenge. As a result of the multicore revolution, parallel programming has however ceased to be a task for domain experts only. And for this reason, a large variety of languages and libraries have been proposed that promise to ease this task. This paper presents a study to investigate whether such approaches succeed in closing the gap between domain experts and mainstream developers. Four approaches are studied: Chapel, Cilk, Go, and Threading Building Blocks (TBB). Each approach is used to implement a suite of benchmark programs, which are then reviewed by notable experts in the language. By comparing original and revised versions with respect to source code size, coding time, execution time, and speedup, we gain insights into the importance of expert knowledge when using modern parallel programming approaches.

1 Introduction

The belief that "parallel programming is hard, and best left to experts" has long become a developers' mantra. Indeed, concurrency makes parallel programs prone to errors such as atomicity violations, data races, and deadlocks, which are hard to detect because of their nondeterministic nature. Furthermore, achieving performance is a significant challenge, as scheduling and communication overheads or lock contention may lead to adverse effects, such as parallel slow down.

In spite of these facts, the comfort of leaving parallel programming to domain experts is fading away: the industry-wide shift to multicore processors has made parallelism relevant for mainstream developers. To support their efforts, a variety of advanced programming languages and libraries have been designed, promising an improved development experience over traditional multithreaded programming. The effectiveness of these approaches in practice hinges on their ability to allow developers to easily achieve good results constructing parallel programs. However, how can one quantify "good results", and "easily"? This will clearly depend on what improvements to a parallel program are still left to be made, and how much effort they require to be implemented.

In this paper, we propose to study these effects by examining the expert gap in parallel programming. The expert gap is the distance in expertise between an expert and a competent (though inexperienced in the expert's domain) developer of a parallel program. The gap is quantified by the difference in lines of code used, absolute performance, scalability, and the correction cost (in coding time) to bring the novice code up to the standards of the expert. This is expressed by the following research questions:

Q1: *To what extent do expert comments reduce code size?*
Q2: *To what extent do expert comments reduce execution time?*
Q3: *To what extent do expert comments increase speedup?*
Q4: *What is the overhead of implementing the experts' corrections?*

To address the research questions, we performed a study with four popular parallel programming approaches: Chapel [1], Cilk [2], Go [3], and Threading Building Blocks (TBB) [4]. In the study, we asked notable experts in the respective approaches to review a suite of six parallel benchmark programs [5] implemented by an experienced developer, who had however no previous expertise in the approaches. After implementation of their comments, the experts performed a second review to check that their comments had been addressed appropriately.

We recruited high-profile experts, namely either leaders or prominent members of the respective compiler development teams:

- Brad Chamberlain, Principal Engineer at Cray Inc., technical lead on Chapel
- Jim Sukha, Software Engineer at Intel Corp., Cilk Plus development team
- Luuk van Dijk, Software Engineer at Google Inc., Go development team
- Arch D. Robison, Sr. Principal Engineer at Intel Corp., TBB chief architect

This process led to a solution pool of 48 programs, i.e. six problems in four approaches, each before and after expert review. The data also allows the approaches to be compared with each other, which is an extensive study in itself and *not* the goal of this paper; for reference, we report the complete results in a companion technical report [6].

The remainder of this paper is structured as follows. Section 2 provides background on the four approaches and the benchmark problems used in the study. Section 3 presents the results of the study for each of the four metrics. Section 4 discusses threats to validity. Section 5 presents related work and Section 6 concludes with an outlook on future work.

2 Background

This section provides background on the parallel programming approaches and the benchmark problems used in this study.

2.1 Overview of the approaches

Table 1 summarizes the characteristics of Chapel, Cilk, Go, and TBB, together with year of appearance, and the corporation currently supporting further development.

Chapel [1] describes parallelism in terms of independent computation implemented using threads, but specified through higher-level abstractions. It provides a variety of parallel constructs such as parallel-for (**forall**), **reduce**, and **scan**, leading to very concise parallel code. Its programming model targets both high-performance computers as well as clusters and desktop multicore systems and clusters.

Cilk [2] Cilk exposes parallelism through high-level primitives that are implemented by the runtime system, which takes care of load balancing using dynamic scheduling through work stealing. The keyword **cilk_spawn** marks the concurrent variant of the function call statement, which starts the (possibly) concurrent execution of a function. The synchronization statement **cilk_sync** waits for the end of the execution of all the functions spawned in the body of the current function; there is an implicit **cilk_sync** statement at the end of all procedures. Lastly, there is an additional **cilk_for** construct. This construct is a limited parallel variant of the normal **for** statement, handling only simple loops.

Go [3] is a general-purpose programming language targeted towards systems programming. Parallelism is expressed using an approach based on Communicating Sequential Processes (CSP) [7]. The statement **go** starts the execution of a function call as an independent concurrent thread of control, or *goroutine*, within the same address space. *Channels* (indicated by the **chan** type) provide a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type; channels can be synchronous or asynchronous. Few parallel constructs are readily available in Go, resulting in more verbose code. For example, to construct a parallel-for loop the work gets dispatched to a channel from one go routine, while a number of goroutines fetch work from this channel and process it.

Threading Building Blocks (TBB) [4] is a parallel programming template library for the C++ language. Parallelism is expressed using Algorithmic Skeletons [8], and the runtime system takes care of scheduling and load balancing using work stealing. Similar to Chapel, a variety of parallel constructs are available, such as `parallel_for`, `parallel_reduce`, and `parallel_scan`.

2.2 Benchmark problems

We chose the problems suggested in [5] as benchmarks, which comprehend a wide range of parallel programming patterns. Reusing a tried and tested set has the benefit that estimates for the implementation complexity exist and that problem

Name	Programming abstraction	Communication paradigm	Programming paradigm	Year	Corporation
Chapel	Partitioned Global Address Space (PGAS)	message passing / shared memory	object-oriented	2006	Cray Inc.
Cilk	Structured Fork-Join	shared memory	imperative / object-oriented	1994	Intel Corp.
Go	Communicating Sequential Processes (CSP)	message passing / shared memory	imperative	2009	Google Inc.
TBB	Algorithmic Skeletons	shared memory	C++ library	2006	Intel Corp.

Table 1. Main language characteristics

selection bias can be avoided by the experimenter. We chose these particular benchmark problems as it was important to keep the amount of time spent with each problem reasonably small (experts could devote only a limited amount of time to the review). The problems have been designed for this purpose [5]; in order to be more representative of large applications, they can also be chained together.

Again to keep the number of implementations manageable, we selected the following six from [5]:

– Random matrix generation (randmat)
– Histogram thresholding (thresh)
– Weighted point selection (winnow)
– Outer product (outer)
– Matrix-vector product (product)
– Chaining of problems (chain)

Note that the last problem, chain, corresponds to a chaining together of the inputs and outputs of the other five.

3 Results

This section presents and discusses the data collected in the study. Table 2 provides absolute numbers for all versions of the code, before and after expert review. The figures in this section display the relative differences between the expert and non-expert versions.

3.1 Source code size

The differences between the non-expert and expert versions with respect to lines of code are given in Figure 1. Addressing research question **Q1**, it is apparent from the figure that suggested changes decreased the source code size only moderately, and increased it in several cases. Indeed, on average the code size decreased, over all languages, by only 1.6% (standard deviation of 13.9%).

	Problem Version[1]	randmat nv	randmat ex	thresh nv	thresh ex	winnow nv	winnow ex	outer nv	outer ex	product nv	product ex	chain nv	chain ex
Source code size	Chapel	33	32	58	61	72	74	55	58	34	36	145	159
	Cilk	48	40	119	95	146	139	83	72	65	58	320	251
	Go	52	71	141	118	144	191	103	98	89	86	345	330
	TBB	52	53	110	98	142	137	83	81	63	62	302	302
Coding time (min)	Chapel	76	100	121	156	134	155	55	64	43	45	76	137
	Cilk	101	154	251	294	112	121	26	39	12	15	77	118
	Go	45	76	132	163	92	163	24	31	18	21	56	91
	TBB	35	37	196	207	41	43	32	43	23	23	24	26
Execution time (sec)[2]	Chapel	18.7	3.1	7.8	13.1	21.4	21.3	1.6	1.6	1.4	1.4	36.0	36.0
	Cilk	0.5	0.4	0.9	0.8	0.8	0.7	0.3	0.2	0.3	0.2	2.4	1.7
	Go	2.9	0.5	2.1	1.6	2.0	1.3	1.5	2.4	1.1	0.3	177.7	38.4
	TBB	0.3	0.2	1.2	0.6	1.0	1.0	0.3	0.3	0.2	0.2	2.8	2.8
Speedup[2]	Chapel	1.2	2.8	2.8	2.8	2.3	2.1	3.4	3.5	1.7	1.7	2.0	2.1
	Cilk	13.6	16.8	13.4	14.9	19.1	20.2	8.1	8.1	4.2	5.8	17.3	20.2
	Go	4.1	21.2	8.9	8.1	8.0	11.5	10.4	4.7	1.9	7.5	0.6	1.9
	TBB	20.7	21.2	8.1	14.8	9.4	9.5	7.4	7.4	7.2	7.3	12.5	12.6

[1] nv: novice; ex: expert [2] average times and speedups are given

Table 2. Measurements for all metrics, before and after expert comments

There are differences between the individual languages though. The increases in size for Chapel, on average by about 4.3% (standard deviation 4.2%), can be explained mainly by a single requested change from the expert: ranges/domain definitions were consistently hoisted outside of parallel regions, which saves them being recomputed.

Cilk solutions decreased in size on average by 14.5% (standard deviation 6.2%). This change can be traced back to one of the expert comments to replace **cilk_spawn**/**cilk_sync** style code with **cilk_for**; according to the expert, **cilk_for** simplifies the code while doing the same recursive divide-and-conquer underneath, and should therefore be preferred.

Increase in code size on average by 6.7% (standard deviation 22.1%) are visible in Go. The outliers are randmat and winnow with increases of 36.5% and 32.6% on average. For randmat, this can be explained by a suggested change of data structure; since the randmat program is small to begin with, this relatively small change amounts to a seemingly large increase percentage-wise. For winnow, the increase in performance results from the suggestion of the expert to add parallel merge sort, which is not part of Go's standard library; the original sort didn't parallelize well, resulting in a performance hit.

Lastly, the TBB code size decreased on average by a very moderate 2.8% (standard deviation 4.4%).

In summary, while there is a moderate decrease in code size on average, program restructuring can also lead to moderately increased code sizes (Cilk), and performance considerations may give reason to increase code size by about one third (Go). Large code size increases tend to indicate an algorithmic change,

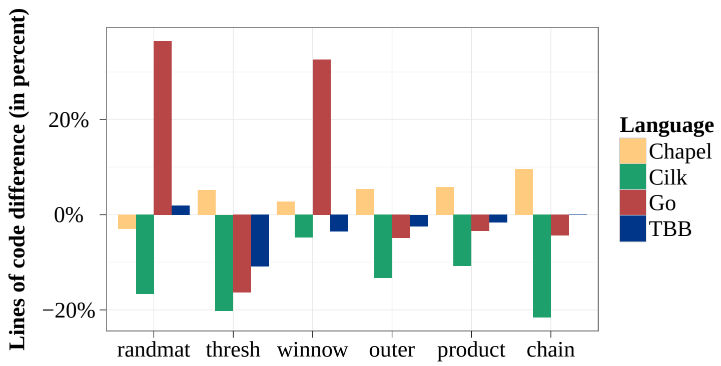

Fig. 1. Source code size (LoC) difference

as in the case of Go. Large code size decreases indicate that functionality was duplicated needlessly and can be removed, as in the case of Cilk. Small changes indicate tweaking, where the code was overall fine, but could use refinement.

3.2 Execution time

The performance tests were run on a 4 $\times$ Intel Xeon Processor E7-4830 (2.13 GHz, 8 cores; total 32 physical cores) server with 256 GB of RAM, running Red Hat Enterprise Linux Server release 6.3. Language and compiler versions used were: chapel-1.6.0 with gcc-4.4.6, for Chapel; Intel C++ Compiler XE 13.0 for Linux, for both Cilk and TBB; go-1.0.3, for Go.

Each performance test was repeated 30 times, and the mean of the results was taken. All tests use the same inputs, the size-dominant of which is a $4 \cdot 10^4 \times 4 \cdot 10^4$ matrix (about 12 GB of RAM). This size, which is the largest input size all languages could handle, was chosen to test scalability. The language Go provided the tightest constraint, while the other languages would have been able to scale to even larger sizes. An important factor in the measurement is that for all problems the I/O time is significant, since they involve reading/writing matrices to/from the disk. In order for the measurements to not be dominated by I/O, all performance tests were run with input and output code removed (input matrices were generated on-the-fly instead).

In Figure 2 the differences in execution time are displayed. Addressing research question **Q2**, on average expert comments reduced execution time by 18.1% (standard deviation 38.3%).

Again, results for the individual approaches show a number of differences. On average, execution time was reduced by 2.5% (standard deviation 48.0%) for Chapel. There is one outlier: in the problem thresh, the expert execution time increases by about 67.8%. The expert gave comments on a version that was compiled with version 1.5 of Chapel. After changing to version 1.6 (as suggested

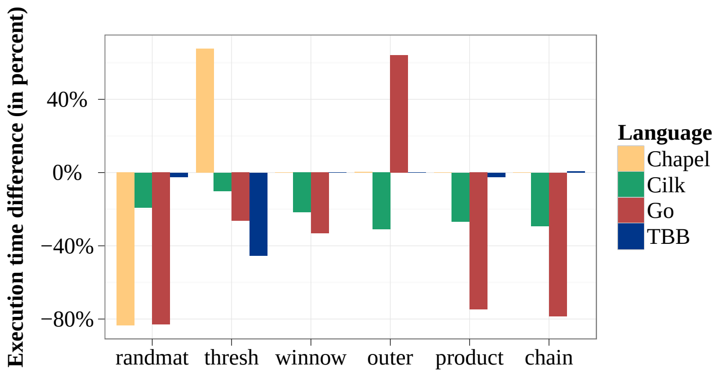

Fig. 2. Execution time difference

by the expert) for the final measurements, the *non-expert* version experienced a significant reduction in execution time, while the expert version remained the same; this illustrates the often fragile nature of optimization.

Cilk's execution times were consistently reduced, on average by 23.0% (standard deviation 7.8%).

Go's execution times were reduced by 38.6% on average (standard deviation 55.8%). These improvements can be attributed to one important change in the way parallelism was achieved. In the non-expert versions, a divide-and-conquer pattern was frequently used. Instead, the expert recommended a distribute-work-synchronize pattern. While the divide-and-conquer approach creates one goroutine per task, the distribute-work-synchronize creates one for each processor core; for fine-grained task sizes, the overhead of the excessive creation of goroutines then causes a performance hit. Again, there was an outlier. In the problem outer, the Go expert had suggested to change the data structure from a one-dimensional to a two-dimensional array for clarity, without apparent performance differences on smaller problem sizes on a desktop machine. In the final measurement, it is however the cause of a 64% increase in execution time in the expert version; this highlights the fact that program optimizations have to take both the target machine and the target problem size into account.

TBB's execution times were consistently reduced by 8.3% on average (standard deviation 18.2%).

In summary, expert comments reduced execution time by a moderate amount. Also, there were outliers which increased the execution times, highlighting the fact that performance profiling is important in addition to expert knowledge.

3.3 Speedup

Figure 3 shows the changes in speedup on 32 cores; the speedup is measured relative to an execution on a single thread. Addressing research question **Q3**, across all languages and problems a speedup of 1.5 is achieved on average (standard deviation 1.1).

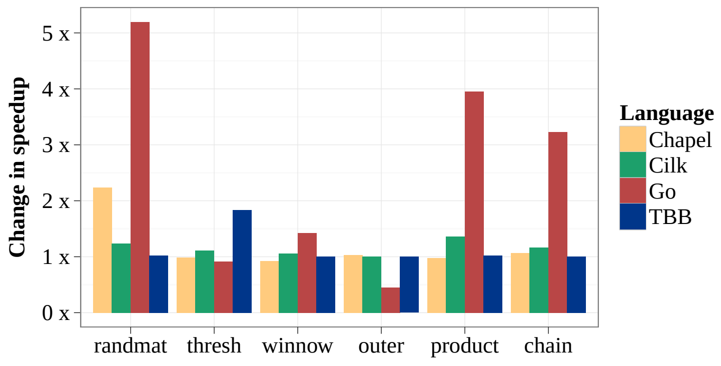

Fig. 3. Change in speedup

Except for Go, speedup seems to have been influenced little by the expert comments; most of the time no further speedup (i.e. 1 × speedup) is visible. Chapel shows on average a speedup of 1.2 (standard deviation 0.5), Cilk 1.2 (standard deviation 0.1), and TBB 1.1 (standard deviation 0.3).

In Go a more substantial average speedup of 2.5 (standard deviation 1.9) is visible, which is due to strong improvements in the case of randmat, product, and chain. This is most likely caused by the change in concurrency pattern used, as discussed in Section 3.2. It emphasizes the fact that it is critical in Go to know about idiomatic patterns to make full use of the performance offered by the language. A slowdown is visible for the outer problem in Go, which corresponds to the discussed issue in outer for the execution time difference.

In summary, speedup improvements due to expert comments are moderate in general. Only in the case of Go, the knowledge of an idiomatic pattern brought about significant improvements. Go highlights the fact that expertise is more valuable in approaches where there are fewer prepackaged solutions (such as parallel-for constructs).

3.4 Correction time

Figure 4 displays which fraction of the original coding time was spent on implementing the corrections suggested by the experts. Addressing research question **Q4**, to implement the expert comments took about 29.9% of the original time spent on average (standard deviation 24.6%). Over all languages and problems, a maximum of 79.4% of the original coding time was spent.

This moderate overhead is reflected consistently by Chapel (29.3%, standard deviation 26.4%), Cilk (34.8%, standard deviation 20.2%), and Go (46.1%, stan-

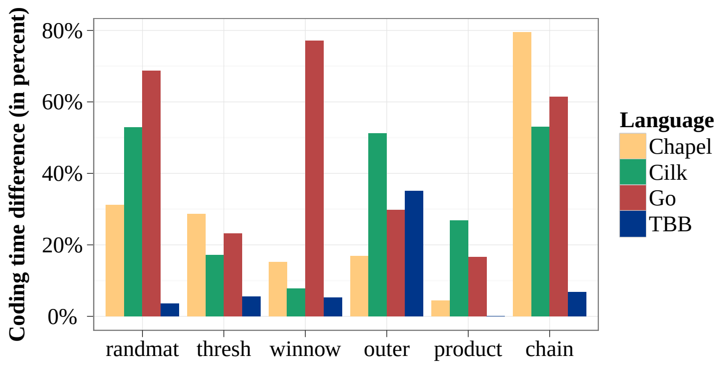

Fig. 4. Correction time

dard deviation 26.0%). Only TBB has a significantly lower coding time overhead of 9.4% (12.8% standard deviation).

In summary, none of the original problems needed to be rewritten completely; the changes were incremental. This is in accordance with the noted changes in source code size. Also, in combination with the observations about speedup, it is clear that the time spent correcting the Go code translates into a notable speedup, making the availability of an expert an effective way to increase scalability and performance of the code.

4 Threats to Validity

As a threat to external validity, the study results are not necessarily generalizable to other languages and libraries. We have simply chosen four popular approaches, and it is interesting to see that the study results tend towards the same direction for all of them. But the results do not readily transfer to other approaches with entirely different programming abstractions and potentially different implementation quality.

Furthermore, it is arguable whether the study results transfer to large applications, due to the size of the programs used. The modest problem size is intrinsic to the study: the use of top experts is crucial to reliably answer the research questions and, unfortunately, this also means that the program size has to remain reasonable to fit within the review time the experts were able to donate. However, a recent study [9] confirms that the amount code dedicated to parallel constructs for 10K and 100K LOC programs is between 12 and 60 lines of code on average; this makes our study programs representative of the parallel portions of larger programs.

We use one developer (with six years of development experience; working at Google Inc.), and one expert per language (as listed in Section 1). Each expert was high-profile, i.e. using another expert or a group of experts would most likely lead to worse suggestions for improvement. Similarly to the point made above, using a group of developers, while preferable, would make the expert review step impossible (too many programs to review). To instead compare many novice programs with an ideal program is possible, but a markedly different study: an expert is required to filter out harmless differences between programs. Making all changes to turn one program into another is not representative of the effort required to bring the program up to an acceptable level.

Problem selection bias, a threat to internal validity, is avoided in part by using an existing problem set, instead of creating a new one. The threat that specific problems could be better suited to some languages than others remains, as it could already be present in the existing problem set. As a positive indication, none of the experts criticized the choice of problems.

5 Related work

Although the claim that parallel programming requires domain experts is often repeated, few studies investigate the validity of this claim in the context of modern parallel programming approaches. However, a number of studies on *comparing* approaches to parallel programming influenced our work, and we discuss them here.

Szafron and Schaeffer [10] assess the usability of two parallel programming systems (a message passing library and a high-level parallel programming system) using a population of 15 students, and a single problem (transitive closure). Six metrics were evaluated: number of work hours, lines of code, number of sessions, number of compilations, number of runs, and execution time. They conclude that the high-level system is more usable overall, although the library is superior in some of the metrics; this highlights the difficulty in reconciling the results of different metrics.

Hochstein et al. [11] provide a case study of the parallel programmer productivity of novice parallel programmers. The authors consider two problems (game of life and grid of resistors) and two programming models (MPI and OpenMP). They investigate speedup, code expansion factor, time to completion, and cost per line of code, concluding that MPI requires more effort than OpenMP overall in terms of time and lines of code. Hochstein et al. [12] compare programming effort for two parallel programming models (message-passing and PRAM-like), using one problem (sparse-matrix dense-vector multiplication), with two groups of students. The used metrics are: development time and program correctness. The results show a 46% lower PRAM-like development time compared to message-passing, and no statistically significant difference in correctness rates.

Rossbach et al. [13] conducted a study with 237 undergraduate students implementing the same program with locks, monitors, and transactions. While the students felt on average that programming with locks was easier than programming with transactions, the transactional memory implementations had the fewest errors. Ebcioglu et al. [14] measure the productivity of three parallel programming languages (MPI, UPC, and X10), using 27 students, and a single problem (Smith-Waterman local sequence matching). For each of the languages, about a third of the students could not achieve any speedup.

Nanz et al. [15] present an empirical study with 67 students to compare the ease of use (program understanding, debugging, and writing) of two concurrency programming approaches (SCOOP and multi-threaded Java). They use self-study to avoid teaching bias and standard evaluation techniques to avoid subjectivity in the evaluation of the answers. They conclude that SCOOP is easier to use than multi-threaded Java regarding program understanding and debugging, and equivalent regarding program writing. Pankratius et al. [16] compare the languages Scala and Java using 13 students and one software engineer working on three different projects. The resulting programs are compared with respect to programmer effort, code compactness, language usage, program performance, and programmer satisfaction. They conclude that Scala's functional style does lead to more compact code and comparable performance.

Cantonnet et al. [17] analyze the productivity of two languages (UPC and MPI), using the metrics of lines of code and conceptual complexity (number of function calls, parameters, etc.), obtaining results in favor of UPC. Bal [18] is a practical study based on actual programming experience with five languages (SR, Emerald, Parlog, Linda and Orca) and two problems (traveling salesman, all pairs shortest paths). It reports the authors' experience while implementing the solutions.

6 Conclusion

In order to make developers embrace parallel programming, the reputation of parallelism as an arcane art has to be dispelled. Designers of parallel programming approaches work towards this goal, but results supporting their claims of improved performance and usability are scarce. While it is easy to check that one approach can offer performance improvements over another, it is entirely unclear whether a non-expert would ever achieve this performance in practice.

In this paper we presented, to the best of our knowledge, the first study that explores the alleged gap between expert and non-expert parallel programmers. The results positively confirm the effectiveness of the design of Chapel, Cilk, Go, and TBB: all the approaches "work" in the sense that, on average, a top expert can only to a moderate degree improve programs written by a non-expert; the study confirms this across four program metrics, namely code size, execution time, speedup, and coding time.

More studies are needed to investigate the difference between expert and non-expert usage, and we hope that our study incites more work in this direction. The most expensive resources in our study were the experts: the time they were able to spend on reviewing programs strongly limited the number of programs in the study. In future work, it would be interesting to replace expert review by comparisons of expert-written, "ideal" programs with non-expert ones. While such an approach would not be able to replicate the detailed insights gained by the review, it would make it easier to obtain more data for the analysis.

References

1. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications **21**(3) (2007) 291–312
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the 5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'95), ACM (1995) 207–216
3. Go programming language. `http://golang.org/`
4. Reinders, J.: Intel threading building blocks – outfitting C++ for multi-core processor parallelism. O'Reilly (2007)
5. Wilson, G.V., Irvin, R.B.: Assessing and comparing the usability of parallel programming systems. Technical Report CSRI-321, University of Toronto (1995)
6. Nanz, S., West, S., Soares da Silveira, K.: Benchmarking usability and performance of multicore languages. `http://arxiv.org/abs/1302.2837` (2013)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
8. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press (1991)
9. Okur, S., Dig, D.: How do developers use parallel libraries? In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12), ACM (2012) 54:1–54:11
10. Szafron, D., Schaeffer, J.: An experiment to measure the usability of parallel programming systems. Concurrency: Practice and Experience **8**(2) (1996) 147–166
11. Hochstein, L., Carver, J., Shull, F., Asgari, S., Basili, V.: Parallel programmer productivity: A case study of novice parallel programmers. In: Proceedings of the 2005 Conference on Supercomputing (SC'05), IEEE Computer Society (2005)
12. Hochstein, L., Basili, V.R., Vishkin, U., Gilbert, J.: A pilot study to compare programming effort for two parallel programming models. Journal of Systems and Software **81** (2008) 1920–1930
13. Rossbach, C.J., Hofmann, O.S., Witchel, E.: Is transactional programming actually easier? In: Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming (PPoPP'10), ACM (2010) 47–56
14. Ebcioglu, K., Sarkar, V., El-Ghazawi, T., Urbanic, J.: An experiment in measuring the productivity of three parallel programming languages. In: Proceedings of the Third Workshop on Productivity and Performance in High-End Computing. (2006) 30–37
15. Nanz, S., Torshizi, F., Pedroni, M., Meyer, B.: Design of an empirical study for comparing the usability of concurrent programming languages. In: Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM'11), IEEE Computer Society (2011) 325–334
16. Pankratius, V., Schmidt, F., Garretón, G.: Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. In: Proceedings of the 2012 International Conference on Software Engineering (ICSE'12), IEEE Press (2012) 123–133
17. Cantonnet, F., Yao, Y., Zahran, M.M., El-Ghazawi, T.A.: Productivity analysis of the UPC language. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), IEEE Computer Society (2004)
18. Bal, H.E.: A comparative study of five parallel programming languages. Future Generation Computer Systems **8**(1-3) (1992) 121–135