Comparing Sequential and Parallel Code Review Techniques for Formative Feedback

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
andrew@cs.auckland.ac.nz

Arthur Lewis
University of Auckland
Auckland, New Zealand
alew525@aucklanduni.ac.nz

Beryl Plimmer
University of Auckland
Auckland, New Zealand
b.plimmer@auckland.ac.nz

ABSTRACT

The practice of Peer Review is widespread across a range of academic disciplines. We report on a study that compared two different approaches of peer reviewing program code — reviewing a sequence of solutions to the same problem (sequential code review), and reviewing a set of multiple solutions side-by-side (parallel code review). We found that the parallel approach was preferred by the majority of participants in the study and there were some indications that it might be more helpful for reviewers, but the sequential approach elicited more written comments in general, and more specific critical comments compared with the parallel approach. Although parallel reviews may be preferred by reviewers, using sequential reviews appears to result in increased levels of formative feedback for the recipient.

CCS CONCEPTS

• **Social and professional topics → Computing education**;

KEYWORDS

Peer review, peer assessment, peer feedback, code review, sequential code review, parallel code review, formative feedback, novice programmers, CS1

ACM Reference Format:
Andrew Luxton-Reilly, Arthur Lewis, and Beryl Plimmer. 2018. Comparing Sequential and Parallel Code Review Techniques for Formative Feedback. In *ACE 2018: 20th Australasian Computing Education Conference, January 30-February 2, 2018, Brisbane, QLD, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3160489.3160498

1 INTRODUCTION

Software development projects in industry frequently include code review in the development process. Such reviews aim to identify defects and improve the quality of the code [13, 15]. Ultimately, the main purpose of these industry code reviews is to produce a better software product. This contrasts with peer review in education, in which the purpose of the review is to help students learn about code, and to develop higher order cognitive skills [1, 4, 14]. These higher order cognitive skills may include holistic or "soft" skills such as communication [13], giving and receiving feedback [23], and critical thinking [4, 27].

Peer review can be formally be defined as "… an arrangement in which individuals consider the amount, level, value, worth, quality, or success of the products or outcomes of learning of peers of similar status" [24]. In simpler terms, peer review is the process by which a given piece of work (professional or academic) is evaluated by one or more people with the same qualifications or competency as the original creator(s) of that work.

Peer Review has been used in the context of academic education for several decades, and spans areas such as Academic Writing, Science, Engineering, Medicine, Business and Computer Science [5, 6]. Existing education literature has generally reported positive outcomes from peer review activities in classrooms [1, 21], but there have also been concerns over the difficulty of managing the distribution and workflow of reviews, the consistency and quality of reviews, and the perception of some students that tasks involving the evaluation of student work should be performed by instructors [14, 22, 28].

Many of these concerns have been addressed by recent developments in online tools, and by shifting pedagogical trends. A variety of software tools supporting peer review have largely mitigated the administrative cost [17]; an analysis of student peer marking compared with expert marking of introductory programming assignments has demonstrated a high correlation between the expert and peer marks [10]; and, pedagogies in which students learn by taking on roles traditionally held by instructors are becoming more common as software tools that handle the administrative burden are developed [8, 11].

The focus of this paper is the review of program code. Although some aspects of programming can be automatically evaluated using software tools, other aspects require manual (i.e., human) oversight. Determining the syntactic correctness of code is achievable using standard compilers, and feedback is typically provided through integrated development environments such as Visual Studio[1] or Eclipse[2]). Other objective metrics, such as the functional correctness of code, can be tested with test suites like JUnit[3] that may be integrated into the development environment. Compilers and other software tools are not yet capable of determining code quality by considering factors such as program design, efficiency or documentation. For this purpose, human intervention is necessary to give subjective (or qualitative) feedback.

ACE 2018, January 30-February 2, 2018, Brisbane, QLD, Australia

ACM ISBN 978-1-4503-6340-2/18/01…$15.00
https://doi.org/10.1145/3160489.3160498

[1] https://www.visualstudio.com/
[2] https://eclipse.org/
[3] http://junit.org/junit4/

Students engaging in peer review activities typically provide formative feedback (sometimes described as 'peer feedback') on the work authored by their peers [7]. Formative feedback helps educators to better gauge the student reviewer's understanding of programming aspects [25]. Formative feedback is also reported to be beneficial to the recipients of the review, by prompting them to act on the reviewer's feedback [16, 18].

The review of programming code in an academic environment typically occurs when students are given a series of individual solutions to a programming assignment to review. Each of these solutions is typically reviewed in a sequential manner and independently of each other. We denote this more traditional approach "Sequential Code Review".

Because code is organized into discrete chunks with well-defined structure (such as classes, methods/functions/procedures), it is possible to lay out these discrete chunks side by side and compare them in parallel. This has some similarities with the process that developers go through when using the `diff` command to look at differences between versions of code. However, in this case the review process does not focus on different versions of the same code, but rather different solutions to the same problem. We call this alternative process for reviewing code "Parallel Code Review".

The purpose of our research is to determine if the peer review workflow (i.e., sequential or parallel) impacts student perception of the peer review task, or the nature of the feedback provided. The research questions for this study are:

- RQ1: Does the choice of sequential or parallel workflow have an impact on the perception of the code review task?
- RQ2: Does the choice of sequential or parallel workflow impact the nature of formative feedback?

2 METHODOLOGY

We conducted an experiment to compare code review using sequential workflow with code review using parallel workflow. We used a counter-balanced, between subjects, experimental design. Each participant was allocated to one of four different groups (A–D). Participants in all groups completed two peer review tasks, one using sequential workflow and one using parallel workflow. However, the ordering of the workflow, and the programs that participants reviewed were counter-balanced across the groups, as indicated in Table 1.

Table 1: Ordering of review tasks

	First task		Second task	
Grp	Code	Workflow	Code	Workflow
A	Set 1	Sequential	Set 2	Parallel
B	Set 1	Parallel	Set 2	Sequential
C	Set 2	Sequential	Set 1	Parallel
D	Set 2	Parallel	Set 1	Sequential

After completing each of the code review tasks, participants were asked to complete a questionnaire, and after completing both tasks, participants completed a third questionnaire comparing the sequential and parallel workflow. An optional interview was conducted with a few participants at the end of the session to ask further questions about their answers in the questionnaires. Quantitative data from the questionnaires was collated, and is reported in tabular form in this article. Qualitative data was analyzed and grouped into categories using Thematic Analysis [2, 3].

2.1 Participants

Participants were recruited from the population of undergraduates and postgraduates in the Computer Science Department at the University of Auckland.

The participant demographics are reported in Table 2. The participants consisted of twelve undergraduates with varying levels of expertise, and seven graduate students. Eight of the undergraduates were first year students, either at the end of a typical introductory programming course (i.e., a CS1 course), or those enrolled in a second programming course focused on data structures and algorithms (i.e., a CS2 course). Students are taught Python programming in the first year curriculum, but use a variety of other languages in subsequent years. The twelve graduate participants were tutors and/or markers of several undergraduate Computer Science courses. The participants represent a range of different ability levels, but all of the students were able to understand the code sufficiently to comment on the code style.

Table 2: Participant demographics

Student experience	N	Male	Female
First year	8	5	3
Second year	2	2	0
Third year	2	2	0
Masters	3	3	0
Doctorate	4	3	1
Total	19	14	5

2.2 Tasks

For each code review task, participants were given a problem description and four samples of code that were solutions to the problem. All the programs were syntactically and functionally correct, but varied in style and approach. Participants were asked to identify the positive aspects of each program, and to suggest possible improvements for each program.

Specific criteria listing desirable features were not provided, but rather participants were given general holistic grading guidelines. The use of holistic guidelines encourages creativity, rather than constraining comments to specific criteria, and are the preferred approach to elicit formative feedback [20].

2.2.1 Sequential Code Review. In the sequential code review condition, participations were given the first code sample and were asked to complete the code review before the subsequent code sample was made available. Participants were not permitted to return to a previous review and revise their comments.

2.2.2 Parallel Code Review. In the parallel code review condition, participants were given all four code samples at the same time

```
def free_space(cargo, capacity):
    if cargo % capacity == 0:
        return 0
    return capacity - cargo % capacity
```

Figure 1: A good solution from Set 1

```
def free_space(cargo, capacity):

    index=0
    cargo1=cargo
    while (cargo1 >= capacity):
        cargo1 = cargo1-capacity
        index=index+1

        if cargo1!=0:
            index=index+1

    return capacity*index-cargo
```

Figure 2: A poor solution from Set 1

and were permitted to position the code adjacent to each other on the table.

2.2.3 Code problems and solutions. Python was used as the language for the code review task. Each code sample was relatively simple, consisting of approximately 5–10 lines of code. The solutions to be reviewed were selected from solutions submitted by students in a previous first year course to ensure the code being reviewed was authentic. Both problem sets used code that involved basic flow of control, variables, expressions, conditions, loops, and functions with parameters and return values.

The problem description for the solutions used in Set 1 was:

> Write a function called `free_space` that calculates the amount of free space remaining in the last box, if a given amount of cargo is packed into boxes of a given capacity. The function header has been provided for you. You can assume `cargo` and `capacity` are both integer values.
>
> `def free_space(cargo, capacity):`

An example of a solution with generally good style, and a solution with generally poor style from the code review task are shown in Figure 1 and Figure 2 respectively.

The problem description for the solutions used in Set 2 was:

> Write a function called `legal_drinker` that determines if a person is legally able to drink alcohol, and returns a Boolean value of `True` if they are legally permitted to drink and `False` otherwise. If the person is over the age of 18, or the alcohol is supplied by their guardian, then they are legally permitted to drink. The function header has been provided for you. You can assume the parameters `under_18` and `supplied_legally` are both Boolean values.

```
def legal_drinker(under_18,supplied_legally):
    return not under_18 or supplied_legally
```

Figure 3: A good solution from Set 2

```
def legal_drinker(under_18,supplied_legally):
    if under_18 == True:
        if supplied_legally == True:
            return True
        else:
            return False
    else:
        return True
```

Figure 4: A poor solution from Set 2

> `def legal_drinker(under_18, supplied_legally):`

An example of a solution with generally good style, and a solution with generally poor style from the code review task are shown in Figure 3 and Figure 4 respectively.

3 RESULTS

3.1 Perception of Review Workflow

After experiencing code review in both sequential and parallel workflow, participants were asked which approach made it easier to make judgments about the overall quality of a piece of code, and why it was easier. The vast majority of participants preferred the parallel approach. Table 3 summarizes the responses.

Table 3: Participant preferences

Group	Sequential	Parallel
Undergraduates	0	12
Graduates	2	5
Total	2	17

The reasons that participants gave for preferring the parallel workflow were grouped into three themes. The number of responses falling into each theme is summarised in Table 4.

Explict vs implicit standards. This theme comprises comments in which a participant treated parallel workflow as providing a standard and/or benchmark for comparing code, thereby making the holistic grading guidelines easier to interpret. For example:

> *I had a baseline to compare it (i.e. code) to. Guidelines were helpful but having another code to compare against is easier.*

Some responses in this category indicated that it was difficult for participants to make holistic judgments about the quality of code when they saw the code in isolation.

> *The grading schedule was unclear for reviewing an individual piece of work because there were no reference points.*

Differentiation. This theme captures responses that suggest it is easier to identify differences between solutions when they are spatially located next to one another. This category also consisted of responses where participants explicitly and even implicitly reported having a better sense of attention to detail while reviewing code using parallel workflow. For example:

> *Easier to look at each piece of code and compare them. Looking at it sequentially makes it slightly forgettable and you can lose attention to detail.*

and

> *It's always easy and beneficial to compare what you are reviewing to something else. It makes it easier to pick up your mistakes and also to improve what you already know.*

Useful for novices. This theme covered responses that involved code comparison being a good way to help people with little programming experience (i.e., novices) learn effectively. For example:

> *Would be especially useful for someone with little programming experience.*

The number of responses falling into each theme is summarised in Table 4. The majority of participants reported that the explicit comparison performed in the parallel workflow made it easier to determine which code was higher quality, and helped them to pay attention to the details.

Table 4: Participant reasons for preferences

Theme	Responses
Explict vs implicit standards	15
Differentiation	10
Useful for novices	4

3.2 Impact on Nature of Formative Feedback

To determine if the workflow had any impact on the formative feedback provided by participants, we analysed both the quantity and the quality of the review comments. In this section we use the term "formative feedback" to refer to the comments provided by participants on the positive and negative aspects of the code.

The code review form had separate areas for participants to identify both positive and negative (i.e., areas with room for improvement) aspects of the code. We treat the feedback on each code sample as a separate review, resulting in 76 parallel code reviews and 76 sequential code reviews (i.e., 19 participants each completed four reviews for each condition).

To compare the quantity of words written by participants during the sequential code review with the quantity of words written during the parallel code review, we calculated the average number of words written in each condition (parallel and sequential) for each aspect (positive and negative). Table 5 summarizes the responses.

Table 5: Mean quantity of feedback (i.e., word count) for each aspect and review workflow.

		Sequential		Parallel	
Code aspect	N	μ	σ	μ	σ
Positive	76	8.7	7.4	4.2	3.6
Negative	76	18.9	13.3	8.3	6.6

The results suggest that more feedback was written during sequential code review compared with parallel code review. A 2-tailed t-test indicated that this difference is statistically significant for both positive and negative feedback ($p < 0.001$ for both aspects).

The review comments were analyzed using two different approaches. Each approach analyzed a different characteristic of the feedback. Firstly, we explored the specificity of the feedback (Section 3.2.1), and secondly, we examined which aspects of code style were the subject of feedback in each condition (Section 3.2.2).

3.2.1 Specificity of Feedback. The formative feedback obtained from participants was first analyzed by classifying it into six categories. These categories were adapted from a study conducted by Hamer et al. [9] which investigated the differences between student and tutor peer feedback. In this section the categories are described, and exemplar comments from each category are provided.

Specific Positive (S+). This comprises participant comments that are positive about specific aspects of the code. Examples of comments in this category include:

- *Meaningful variable names*
- *Intuitive algorithm*
- *Efficient — made use of modulus for calculation*

Specific Negative (S-). This comprises participant comments that are negative about specific aspects of the code. Examples of comments in this category include:

- *Unnecessary variable declaration of variable* `legal_drinker`
- *No return after if-else*
- *Use of for loop slows down the running time*

Specific Advice (SA). This comprises participant comments that provide specific advice to improving aspects of the code. Examples of comments in this category include:

- *Could replace elif with else*
- *Variable use not really necessary. It does improve readability but not required. Second elif condition can be improved. Remove age check*

General Positive (G+). Feedback in this category consists of general comments that are of a positive nature. Examples of comments in this category include:

- *It works! :)*
- *Nice Style*

General Negative (G-). Feedback in this category consists of general comments that are of a negative nature. Examples of comments in this category include:

- *Not efficient*
- *Seems incomplete*

General Advice (GA). This category consists of participant responses that provided general advice of a constructive nature, but did not refer to anything specific within the code.

- *No issues but can be improved*

A single sentence from a reviewer may fall into more than one of the categories. For example, one participant comment that contains feedback belonging to both G+ and S- is shown below:

- *The code works (G+), although there are redundant conditions (S-)*

The categories of PV (personal value), and OT (off-topic) used by Hamer et al. [9] were not applied in this study. Feedback from the PV category was directed to the author of a given solution (e.g., *'I don't get what you are trying to do here'*), whereas feedback from the OT category was irrelevant to the code review (e.g., *'Any comments on my marking or comments my mail is xxxxx@hotmail.com'*). The PV and OT categories were excluded from this analysis since none of the participants in the study gave feedback belonging to these categories. This is most likely because participants in this study were writing feedback under experimental conditions and knew that the recipients of the feedback were the researchers, rather than other students.

Table 6: Number of code reviews that included specific and general feedback of various polarities for sequential (Seq) and parallel (Par) workflow

Feedback Type	Seq	Par	χ^2	p
Specific Positive (S+)	52	52	0.00	1.00
Specific Negative (S-)	29	16	5.33	0.02
Specific Advice (SA)	54	42	4.07	0.04
General Positive (G+)	39	31	1.69	0.19
General Negative (G-)	14	6	3.68	0.05
General Advice (GA)	3	1	-	-

Table 6 reports the number of code reviews that include formative peer feedback in each of the six feedback categories. Sequential code review resulted in an equal or higher amount of feedback than parallel code review in all 6 categories.

To determine which types of feedback varied significantly between the sequential and parallel review conditions, we performed a chi-squared test (df=2, N=76) for each category. We found a significant difference ($p < 0.05$) between parallel and sequential code reviews for S- and SA, with the category of G- being borderline significant ($p = 0.055$). There was insufficient data in the GA category to warrant significance testing. The observed differences in the categories of S-, SA, and G- suggest that the differences in workflow affect the amount and nature of the feedback provided. Possible reasons for these variations are discussed in Section 4.

3.2.2 Feedback on Code and Algorithm Style. The previous analysis focuses on aspects of reviews that are domain independent, such as the specificity and polarity (positive and negative facet) of the comments. Our domain is more specific, reviewing Python code, so we were interested to discover which aspects of the code were the focus of feedback from reviewers.

We therefore categorized the feedback according to the aspect of the solution that the reviewer focused on. These categories were established following a thematic analysis of the feedback. In this analysis, we distinguish between feedback focused on aspects typically described as code conventions (e.g., code layout, naming of identifiers), which we label *code readability*, and feedback related to the semantics of the code (e.g., flow of control, syntax used for a given instruction, and the algorithm used), which we label *code structure*.

The four categories of feedback identified are described below, with exemplar comments.

Code Readability Positive (CR+). This category comprises participant comments that are positive in nature, and identify aspects related to typical code conventions, such as good choices of variable names, and general code layout. Examples of comments in this category include:

- *Good variable names chosen*
- *Meaningful variable names*

Code Readability Negative (CR-). This category comprises participant comments that are negative, or that suggest changing the code, and identify aspects of the code that relate to readability. These aspects include lack of white space and parenthesis, unwanted parentheses, the absence of comments or documentation, poor choice of variable names, or failure to adhere to variable naming conventions. Examples of comments in this category include:

- *Parentheses for if-else conditions are not really required in Python*
- *Missing spaces in expression*
- *Lack of parentheses in Boolean condition confuses readability*
- *Variable 'index' is misleading since it is not really an index*

Code Structure Positive (CS+). This category comprises participant comments that are positive in nature and relate to the algorithm's overall quality or style. These aspects include the algorithm's intuitive and/or innovative nature, the efficiency with respect to computations and/or space and time complexity, or feedback on the elegance/succinctness of the code.

- *Efficient — made use of modulus for calculation*
- *Well thought out — an innovative solution*
- *Good space and time complexity*
- *Nice branching style*
- *Concise*

Code Structure Negative (CS-). This category comprises participant comments that are negative, or that suggest changes to the code, and which relate to the algorithm's overall quality or style. These aspects include redundant variables, redundant conditions, redundant loops, or inefficient ways of writing code. Examples of comments in this category include:

- *Could replace* `elif` *with* `else`
- *Use of* `for` *loop slows down the running time*
- *Redundant variable declaration —* `cargo+1`
- *Bad practice to use* `i = i + 1`. *Better to use* `i += 1`
- *Nested condition makes the algorithm a bit tricky to understand. This branching structure could be improved*

- *It is better to use* `not legally_supplied` *and* `if legally_supplied` *instead of* `if legally_supplied == false` *and* `legally_supplied == true`

Note that many of the comments belonging to the general feedback categories (i.e., GA ,G+, G-) described in the previous subsection are not about code, and were therefore not categorized in our domain-specific analysis. Examples of comments that would not fall under any of the code-specific categories are:

- *Seems incomplete (G-)*
- *The code works (G+)*
- *No issues but can be improved (GA)*

Table 7: Number of code reviews that included feedback on code readability and code structure for sequential (Seq) and parallel (Par) workflow

Feedback Type	Seq	Par	χ^2	p
Code Readability Positive (CR+)	51	42	2.24	0.13
Code Readability Negative (CR-)	37	35	0.11	0.75
Code Structure Positive (CS+)	56	43	4.90	0.03
Code Structure Negative (CS-)	75	50	28.15	< 0.01

A summary of the number of reviews containing feedback belonging to each category is shown in Table 7. Sequential workflow has resulted in a greater number of reviews containing comments from the corresponding category compared with parallel workflow. To determine which categories of feedback were significantly different, we performed a chi-squared test, (df=2, N=76), for each category. We found a significant difference ($p < 0.05$) between parallel and sequential code reviews for both of the code structure categories (CS+ and CS-), but not for code readability (CR+ and CR-).

4 DISCUSSION

Most participants perceived that parallel code review provided an explicit standard or benchmark that made the review process easier. This is consistent with the findings of a study by Williams [26], in which students reported that they value the notion of comparing different standards of work. Similarly, Hanrahan & Isaacs [12] and Luxton-Reilly et al. [19] both reported that students liked to compare their own work with others and such comparisons helped them to establish expected standards.

Although participants claimed that parallel code review allowed them to more easily focus on details, and helped them to identify mistakes in the code, participants performing the parallel code review task provided less feedback overall. They wrote fewer words, gave less specific and general feedback in all categories (except for specific positive feedback which was equal), made fewer comments on code readability, and fewer comments on code structure.

We speculate that the observed difference in quantity of feedback may be due to the timing of the feedback. In the sequential code review, the feedback is written immediately after looking at the individual piece of code. There are no other distractions and the entirety of the feedback relates to the code that is the focus of attention. This requires participants to have internalised appropriate criteria before engaging in the peer review process, but during the process itself, reviewers can focus on one piece of code at a time. The workflow of parallel review entails looking at all four examples of code simultaneously. Reviewers have to keep a particular issue in mind while looking across all four code samples to identify if the issue is present. It is likely that this process imposes a higher cognitive load than looking at a single code sample, which may make it difficult to remember all the observed differences.

It is possible that participants focused more on elements that differed between the four solutions, and paid less attention to the elements that were commonly shared. In explanations of why participants found it easier to use parallel code review, they focused on differences:

- *I could quickly differentiate between good and bad code. Would be especially useful for someone with little programming experience.*
- *Allows benchmarking between code, can distinguish differences better.*
- *I evaluated the solution which follows python conventions the most. So I did not have to state things that were common*

Focusing on differences may result in fewer comments on elements that are shared between all code examples. In exit interviews, some participants were specifically asked why they wrote less for the parallel code reviews. Their responses suggest that parallel workflow may require more thinking about the problem, but not all judgements were explicitly documented. Participant comments that illustrate this interpretation are:

- *I compared pros and cons for all solutions before writing any feedback*
- *The parallel review gave me more context which enabled me to focus on the unique elements of style and judge the appropriateness of those elements.*

It is possible that asking reviewers to explicitly compare different code samples encourages norm-based assessment, where a given piece of code is judged against the quality of the other code samples, rather then an internalized set of standards that might better reflect criterion-based assessment. This may explain why common flaws (e.g., aspects categorised as specific negative feedback) were less frequently mentioned during parallel code review tasks. One possible way to address this concern might be to ensure that an ideal 'model' solution is included in the set of code samples to review. This may encourage reviewers to mention a greater number of possible improvements, since any negative aspect of the code could be contrasted with the model answer.

The review tasks were conducted on paper, and the overhead of writing the same comment on several pieces of paper in parallel may have discouraged reviewers from commenting. A software tool that supported parallel review might make it easier to make comments that apply to multiple code samples (e.g., through copy and paste functionality). However, there are significant challenges to developing a user interface that supports parallel code review, especially when reviews involve longer programs, or programs that consist of multiple files.

5 THREATS TO VALIDITY

In this section we discuss several possible threats to the validity of our findings.

Firstly, in typical peer review tasks, students create a piece of work and subsequently review other student solutions that have used the same specifications. In this study, the participants did not author solutions to the problems that they were reviewing, and furthermore, participants had a variety of prior experience which means that most would not be considered as 'peers' of the students who had created the code. However, prior work on peer review of novice programmer code has demonstrated that students and expert markers correlate highly in marks [9, 10], and that there are no significant differences in feedback between tutors and novices when subjective issues of style are concerned [9]. Although the tasks in this study were not peer review tasks, it seems likely that the characteristic differences between the workflow will transfer to tasks in which students are reviewing the work of their peers.

Secondly, this study was conducted in an artificial environment in which participants were observed by one of the researchers. The differences between workflow practices of students conducting code reviews in real educational environments may differ, particularly when marks are at stake. In future, if a software tool supporting parallel code review practice was deployed in a classroom setting, there may be an opportunity to determine if the findings from the artificial study environment transfer to a more natural classroom setting.

Thirdly, the code review tasks conducted in this study were designed to be completed in a timely fashion without imposing an undue burden on the voluntary participants. The code being reviewed was therefore relatively simple. It is unknown if the findings observed in this study would scale to more complex and lengthy code samples.

Finally, the wording used in the instructions for participants engaged in the sequential review tasks were slightly different from those given in the parallel task. In the sequential review condition, participants were asked "What do you think are the good aspects of this program?", while in the parallel review condition, participants were asked "What do you think were the positive aspects of each of these four solutions?" The difference between *good* and *positive* is minor, but may have introduced an unintentional bias.

6 CONCLUSIONS

In this study, we have compared two different techniques to help students review programming code. The first technique, *sequential code review* involved reviewing multiple pieces of code in a traditional, sequential manner. The second approach, *parallel code review* involved reviewing multiple pieces of code simultaneously. Both these techniques helped elicit formative feedback, which has been reported by education literature to be beneficial for both students and educators.

Participants reported that parallel code review was easier. They felt that it helped them focus on the details, it provided an explicit benchmark, and it would be easier for novices. However, the formative feedback obtained from sequential code review was found to be comprised of more written words. The feedback for categories related to the general positive, general negative and specific negative aspects of code was found to be greater for sequential code review. However, the number of specific positive comments was found to consistent across both code review techniques.

The review comments were also classified according to whether they focused on code readability or code structure, and whether they were positive or negative (critical). In all categories, the sequential code review process elicited a greater number of comments compared with parallel code review.

Although the parallel approach was preferred by the majority of participants, it resulted in less feedback overall. While there are some indications that parallel code reviews may be better for reviewers, the more traditional sequential code reviews appear to be better for the recipients of the reviews.

7 FUTURE WORK

Further investigation of the differences between parallel and sequential code review is needed to validate and generalize the findings reported here. Future work could replicate this study in different contexts, such as with longer code samples, or code that includes more complex algorithms.

Although differences were observed in the feedback provided by reviewers, we did not investigate how the feedback would be perceived by the recipients of the review, and in particular, how it might be used to improve their code. Future work could look at the differences in the effectiveness of parallel and sequential workflow for improving code quality, both for the recipients of the review, and for the reviewers themselves.

8 ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

This project was approved by the University of Auckland Ethics committee, approval number: UAHPEC 016357.

REFERENCES

[1] Karen Anewalt. 2005. Using Peer Review As a Vehicle for Communication Skill Development and Active Learning. *Journal of Computing Sciences in Colleges* 21, 2 (Dec. 2005), 148–155. http://dl.acm.org/citation.cfm?id=1089053.1089074

[2] R. E. Boyatzis. 1998. *Thematic Analysis and Code Development: Transforming Qualitative Information.* Sage Publications I, London.

[3] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[4] Robert Davies and Teresa Berrow. 1998. An Evaluation of the Use of Computer Supported Peer Review for Developing Higher-level Skills. *Computers & Education* 30, 1 (1998), 111–115. https://doi.org/10.1016/S0360-1315(97)00086-9

[5] Nancy Falchikov. 1995. Peer Feedback Marking: Developing Peer Assessment. *Innovations in Education and Teaching International* 32, 2 (1995), 175–187.

[6] Mark Freeman and Jo McKenzie. 2001. Aligning Peer Assessment with Peer Learning for Large Classes: The Case for an Online Self and Peer Assessment System. In *Peer Learning in Higher Education*, David Boud, Ruth Cohen, and Jane Sampson (Eds.). Kogan Page, 156–169.

[7] Sarah Gielen, Elien Peeters, Filip Dochy, Patrick Onghena, and Katrien Struyven. 2010. Improving the Effectiveness of Peer Feedback for Learning. *Learning and Instruction* 20 (Aug 2010), 304–315.

[8] John Hamer, Quintin Cutts, Jana Jackova, Andrew Luxton-Reilly, Robert McCartney, Helen Purchase, Charles Riedesel, Mara Saeli, Kate Sanders, and Judithe Sheard. 2008. Contributing Student Pedagogy. *SIGCSE Bulletin* 40, 4 (2008), 194–212. https://doi.org/10.1145/1473195.1473242

[9] John Hamer, Helen Purchase, Andrew Luxton-Reilly, and Paul Denny. 2015. A Comparison of Peer and Tutor Feedback. *Assessment & Evaluation in Higher Education* 40, 1 (2015), 151–164.

[10] John Hamer, Helen C. Purchase, Paul Denny, and Andrew Luxton-Reilly. 2009. Quality of Peer Assessment in CS1. In *Proceedings of the 5th International Workshop on Computing Education Research Workshop (ICER '09)*. ACM, New York, NY, USA, 27–36. https://doi.org/10.1145/1584322.1584327

[11] John Hamer, Helen C. Purchase, Andrew Luxton-Reilly, and Judithe Sheard. 2010. Tools for "Contributing Student Learning". In *Proceedings of the 2010 ITiCSE working group reports (ITiCSE-WGR '10)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1971681.1971683

[12] Stephanie J. Hanrahan and Geoff Isaacs. 2001. Assessing Self- and Peer-assessment: The Students' Views. *Higher Education Research & Development* 20 (2001), 53–69.

[13] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating Pedagogical Code Reviews into a CS 1 Course: An Empirical Study. In *Proceedings of the 40th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '09)*. ACM, New York, NY, USA, 291–295. https://doi.org/10.1145/1508865.1508972

[14] Christopher D. Hundhausen, Pawan Agarwal, and Michael Trevisan. 2011. Online vs. Face-to-face Pedagogical Code Reviews: An Empirical Comparison. In *Proceedings of the 42nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, NY, USA, 117–122. https://doi.org/10.1145/1953163.1953201

[15] Karen Keefe, Judithe Sheard, and Martin Dick. 2006. Adopting XP Practices for Teaching Object Oriented Programming. In *Proceedings of the 8th Australasian Computing Education Conference (ACE '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, 91–100. http://dl.acm.org/citation.cfm?id=1151869.1151882

[16] Sunny Lin, Eric Liu, and Shyan-Ming Yuan. 2001. Web-based peer assessment: Feedback for students with various thinking styles. *Journal of Computer Assisted Learning* 17 (Dec 2001), 420–432.

[17] Andrew Luxton-Reilly. 2009. A systematic review of tools that support peer assessment. *Computer Science Education* 19, 4 (Dec 2009), 209–232. https://doi.org/10.1080/08993400903384844

[18] Andrew Luxton-Reilly, Paul Denny, Beryl Plimmer, and Daniel Bertinshaw. 2011. Supporting Student-generated Free-response Questions. In *Proceedings of the 16th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. ACM, New York, NY, USA, 153–157. https://doi.org/10.1145/1999747.1999792

[19] Andrew Luxton-Reilly, Paul Denny, Beryl Plimmer, and Robert Sheehan. 2012. Activities, Affordances and Attitude: How Student-generated Questions Assist Learning. In *Proceedings of the 17th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. ACM, New York, NY, USA, 4–9. https://doi.org/10.1145/2325296.2325302

[20] Russell Mark, Haritos George, and Combes Alan. 2006. Individualising Students' Scores Using Blind and Holistic Peer Assessment. *Engineering Education* 1, 1 (2006), 50–60. https://doi.org/10.11120/ened.2006.01010050

[21] Yongwu Miao and Rob Koper. 2007. An Efficient and Flexible Technical Approach to Develop and Deliver Online Peer Assessment. In *Proceedings of the 8th International Conference on Computer Supported Collaborative Learning (CSCL'07)*. International Society of the Learning Sciences, 506–515. http://dl.acm.org/citation.cfm?id=1599600.1599693

[22] Ken Reily, Pam Ludford Finnerty, and Loren Terveen. 2009. Two Peers Are Better Than One: Aggregating Peer Reviews for Computing Assignments is Surprisingly Accurate. In *Proceedings of the ACM 2009 International Conference on Supporting Group Work (GROUP '09)*. ACM, New York, NY, USA, 115–124. https://doi.org/10.1145/1531674.1531692

[23] Elaine Silva and Dilvan Moreira. 2003. WebCoM: A Tool to Use Peer Review to Improve Student Interaction. *Journal on Educational Resources in Computing (JERIC)* 3, 1, Article 3 (Mar 2003). https://doi.org/10.1145/958795.958798

[24] Keith Topping. 1998. Peer Assessment Between Students in Colleges and Universities. *Review of Educational Research* 68, 3 (1998), 249–276.

[25] Scott A. Turner, Ricardo Quintana-Castillo, Manuel A. Pérez-Quiñones, and Stephen H. Edwards. 2008. Misunderstandings About Object-oriented Design: Experiences Using Code Reviews. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 97–101. https://doi.org/10.1145/1352135.1352169

[26] Eira Williams. 1992. Student Attitudes Towards Approaches to Learning and Assessment. *Assessment & Evaluation in Higher Education* 17, 1 (1992), 45–58. https://doi.org/10.1080/0260293920170105

[27] William J. Wolfe. 2004. Online Student Peer Reviews. In *Proceedings of the 5th Conference on Information Technology Education (CITC5 '04)*. ACM, New York, NY, USA, 33–37. https://doi.org/10.1145/1029533.1029543

[28] Andreas Zeller. 2000. Making Students Read and Review Code. In *Proceedings of the 5th Conference on Innovation and Technology in Computer Science Education (ITiCSE '00)*. ACM, New York, NY, USA, 89–92. https://doi.org/10.1145/343048.343090