Debugging in Parallel or Sequential: An Empirical Study

Yulei Pang[1*], Xiaozhen Xue[2], Akbar Siami Namin[3]

[1] Department of Mathematics, Southern Connecticut State University, New Haven, USA.
[2] Department of Computer Science, Southern Connecticut State University, New Haven, USA.
[3] Department of Computer Science, Texas Tech University, Lubbock, USA.

* Corresponding author. Tel.: +1-2033927292; email: pangy1@southernct.edu
Manuscript submitted March 10, 2014; accepted October 5, 2014.
doi: 10.17706/jsw.10.5.566-576

Abstract: Faults need to be identified, localized, and removed from programs. Empirical studies show that coverage-based faults localizations effectively target bugs, even in the presence of multiple faults. Debugging is a time-consuming activity and thus it is beneficial to accelerate the process by employing appropriate techniques. The need for speeding up the debugging process is even more immense when the program under test contains multiple faults. A program with multiple faults can be debugged in parallel where each sub-process specifically targets localizing one of the bugs. The immediate research question that arises here is that how significant is the improvement and performance achieved when debugging is performed in parallel compared to the sequential fault localization strategy. This paper investigates and compares the performance of parallel and sequential debugging in effectively localizing faults where the performance is measured according to fault localization cost required by each strategy. Based on the experimental study of several open source Java programs conducted in this paper, we observe that debugging when is performed in parallel outperforms sequential strategy in terms of the total cost.

Key words: Coverage-based faults localization, debugging, empirical study.

1. Introduction

Software testing is the most time consuming and expensive activity in the software engineering life cycle [1]. Despite the effort spent on developing effective testing and debugging techniques, most of software products are released with severe bugs. A common practice of testing includes: 1) devise and execute test cases to expose potential failures in software, 2) localize the buggy feature in the program under test that causes exhibition of faulty behavior, and 3) fix or remove the bug. Once a faulty behavior is observed, the test engineer should employ proper testing and fault localization techniques to accelerate the debugging process. Hence, any test automation techniques that localize faults as early as possible would improve the testing and in particular the debugging process significantly.

Coverage based faults localization is an automated statistical debugging approach to direct the tester to the source of faults with minimal user intervention. It uses the information captured during program execution to rank a program's statements in decreasing order of suspiciousness [2]. The suspiciousness scores highly depend on two types of data: test cases execution profiles and test status. Fault localization techniques based on coverage have been widely studied and it is a common belief that these automated debugging techniques greatly improve debugging process in the presence of single or multiple faults [3], [4].

A software tester may employ several techniques and strategies to speed up the debugging process when

dealing with faulty programs containing multiple bugs. For instance, a simple strategy to accelerate debugging process is to conduct debugging multiple faults in parallel and thus saving the time required to identify and fix the bugs. The first important research question is whether debugging in parallel can decrease the cost associated with the fault localization technique employed when compared to sequential debugging. In other words, how effective is parallel debugging? Is there any statistically significant difference between sequential and parallel debugging? Research indicates the effectiveness of parallel debugging when applied to small programs developed in C language [5]. This motivates us to pose the second question. It is of interest to learn whether the results obtained for C programs carry over to debugging programs developed in other programming languages and in particular those with objected-oriented features. The evaluation metrics used in the previous study was designed for debugging programs with single fault. The performance of the existing ranking metrics initiates the third research question. In order to address these questions, this paper first proposes evaluation metrics, then investigates and compares the performance of debugging in sequential and parallel. This paper makes the following key contributions:

- Compare the performance of sequential and parallel debugging strategies with respect to the fault localization cost associated with each strategy;
- Verify the results when the number of faults is controlled for;
- Investigate the influence of ranking metrics on the performance of the faulty localization strategies.

The rest of the paper is organized as follows: Section 2 reviews the related work including coverage-based fault localization techniques along with the sequential and parallel debugging strategies. Section 3 proposes an evaluation metric that fits better for localizing multiple faults. The research questions are posed in Section 4. The experimental setup is described in Section 5. Section 6 reports the results. The threats to the validity of the experiments conducted in this paper are discussed in Section 7. We conclude our findings in Section 8.

2. Background

2.1. Coverage Based Fault localization

Debugging based on executing test cases and capturing relevant statistics about code coverage and test status is known as coverage-based fault localization. This statistical debugging technique measures the suspiciousness of program statements and ranks them according to a ranking metric. A variety of ranking metrics have been proposed and studied [6]-[8]. Tarantula was the first debugging tool using ranking metric as shown in Expression (1).

$$Tarantula(s) = \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf} + \frac{ep}{ep+np}} \tag{1}$$

Ochiai ranking metric has been generally reported more effective than the metric used in Tarantula, because Ochiai coefficient also takes the absence in failed runs into account. The Ochiai ranking metric is computable through Expression 2. The metric was extended to hypothesis testing based on contingency tables and in particular to chi-square test often utilized in crosstab analysis Expression (3).

$$ochiai(s) = \frac{ef}{\sqrt{(ef+ep)(ef+nf)}} \tag{2}$$

$$chisquare(s) = \frac{N*(ef*np\text{-}nf*ep)^2}{e*f*n*p} \tag{3}$$

A recent study introduced the application of odds ratio ranking metric and conditional odds ratios to the fault localization problem. It has been reported that the odds ratio metric outperforms the existing ranking metrics Expressions (4)-(5).

$$oddratio(s) = \frac{\frac{ef+0.1}{e+0.1} \Big/ \frac{ep+0.1}{e+0.1}}{\frac{nf+0.1}{n+0.1} \Big/ \frac{np+0.1}{n+0.1}} \tag{4}$$

$$conditional\_oddsration(s) = \frac{(ef)k \times (np)k}{(ep)k \times (nf)k} \tag{5}$$

where:

- *ef*: the number of test cases exercising the underlying statement and failed;
- *ep* the number of test cases exercising the underlying statement and not failed;
- *nf*: the number of test cases not exercising the underlying statement but failed;
- *np*: the number of test cases not exercising the underlying statement and not failed;
- *e*: the total number of test cases exercising the underlying statement;
- *n*: the total number of test cases not exercising the underlying statement;
- *f*: the total number of failing test cases;
- *p*: the total number of passing test cases;
- *N*: the total number of test cases;
- *(XX)k*: the number of test cases that exercising statement k, and also following the explanation mentioned above.

2.2. Parallel and Sequential Debugging

The traditional debugging strategy is based on locating and fixing one-fault-at-a-time, also called as sequential debugging [5]. Basically, the developers or testers have little information about the number of faults exist in the software under test. As a result, they iteratively localize and fix faults, one at a time, and then re-execute the entire test pool. This iterative process continues until there is no failing test case in the test pool.

To speed up the debugging process when there may be multiple faults in a program, the test engineers may perform localizing and fixing faults concurrently and in parallel by creating separate process and flow for debugging each fault. A typical practice is that each developer may focus on only fixing one single failing test case where the passing test cases are utilized. The idea of parallel is pretty popular is the area of big data and I/O [15][16]. Although some techniques, including test cases clustering, are possibly adoptable in parallel debugging, this paper employs a plain and simple parallel debugging where each processer, and not necessarily test engineer, debug one single failing test case where all the other passing test cases are also utilized.

Algorithms 1 and 2 describe both sequential and parallel debugging, respectively, where:

- *TC* is the set of all test cases devised for the program under test;
- *TCn* is a *subset* of *TC;*

- *TP* is the *set* of passing test cases;
- *TF* is the *set* of failing test cases;
- *TFn* is a *subset* of *TF,* and it contains one failing test case.

Table 1. Sequential Debugging

Algorithm 1 Sequential debugging
1: Run all test cases in *TC*
2: **while** TF is not empty do
3: Perform fault localization for each test case in *TF*
4: Fix the bug
5: Re-run all test cases in *TC*
6: **end while**

Table 2. Parallel debugging

Algorithm 2: Parallel debugging
1: Run all test cases in *TC*
2: **for** *n* = 1 **to** *N* **do**
3: *TCn = TFn* $\cup$ *TP*
4: Perform fault localization for each test case in *TCn*
5: **end for**

3. Faults Localization Cost

According to the fault localization literature [9], the fault localization cost is defined as the percentage of the program that needs to be examined before reaching the first faulty statement when ranking schemes are used to order executable statements. As Expression 6 indicates the range of possible values for fault localization cost varies between *0* and *1* where the effectiveness of the employed fault localization technique decreases with the increase of the cost value.

$$\cos t = \frac{rank\_of\_the\_first\_fault}{size\_of\_the\_program} \tag{6}$$

$$total\_\cos t = \frac{codes\_checked}{size\_of\_the\_program} \tag{7}$$

The cost metric given in Expression 6 in fact reflects the effectiveness of the underlying fault localization technique. This metric is developed for measuring cost for programs containing a single fault. In practice, there may be more than one fault in a program. Therefore, a more appropriate metric is needed for accurately evaluating fault localization techniques. This paper introduces a metric called *total cost*, and it is defined as the proportion of statements needs to inspect before reaching all faulty statements. The total cost metric is measurable using Expression 7. It is important to note that the major difference between our metric and existing metrics used in calculating fault localization cost is that the total cost metric embraces inspecting and discovering all possible faults instead of localizing the first one, even though the metrics look alike. It may be possible that some statements rank equally, i.e. tie ranking scores. In calculating the overall fault localization cost when some non-faulty statements are ranked tie with a faulty statement, we took into computation exercising all those statements that were ranked equal with the faulty statement, i.e. the worst-case scenario where the debugger inspects all non-faulty and faulty statements with equal ranks.

Given the main purpose of this paper, i.e. comparing sequential and parallel debugging, the introduced metric to measure debugging cost seems a better choice compared to existing single-fault based cost metrics.

4. Research Questions

In order to evaluate and compare the performance of two faults localization strategies, i.e. sequential and parallel, we pose and study the following research questions:

RQ1. Which fault localization strategy is more effective, sequential or parallel? A recent study, based on experimentations on some C programs, reports that parallel debugging is better when multiple faults are present. In practice, there are some other programming languages, which may have different features, e.g., Java, which is an objected, oriented programming language. It is interesting to learn whether the results obtained for the C programs still carry over to other programming languages and in particular programs with object-oriented features.

RQ2. Is the result consistent for various numbers of faults? In practice, the debuggers are unaware of the number of faults in a program under test. Since our empirical study aims at providing a practical implication for debuggers, it is important to take into account the number of faults to investigate whether the captured results still hold for various numbers of faults.

RQ3. Does the selection of ranking metric influence the effectiveness of debugging strategies? The Ochiai and odds-ratio ranking metrics are reportedly more effective than the others. There are some other more effective ranking metrics, e.g., conditional odds ratio, which is based on contingency table analysis. This research question compares the performance of the two debugging strategies when different ranking metrics are employed.

5. Experimental Study

5.1. Subject Programs

Table 3. Subject Programs. LOC: Lines of Codes, NC: Number of Classes, NT: Number of Test Cases, NF: Number of Faults Studied

Program	Description	LOC	NC	NT	NF
NanoXML(v1)	XML parser	7,646	24	214	10
XMLsec(v1)	XML encryption	21,613	143	92	10
Jmeter(v3)	Load tester	43,400	389	78	10
Jtopas(v1)	Text parser	5400	50	54	10

Because Table 3 lists the subject programs used for the experiment. We obtained these extensively studied modest size Java programs from the Software Infrastructure Repository (SIR)[10]. The first two programs (NanoXML and Jtopas) contain TSL (test specification language) test suits, and the last two (Jmeter and XML-security) are Junit test cases. NanoXML is an XML parser for Java. Jtopas is a small Java library for tokenizing and parsing text. Jmeter is a Java desktop designed to load test functional behavior and measure performance. Xml-security library includes a well-developed digital signature and an encryption scheme. We studied one release of each program. Table 3 lists the chosen programs versions, the number of classes, the number of lines of statements, the number of available test cases, and number of faults studied for each program version.

5.2. Experimental Setup

The faults in the subject programs originated from two sources. A number of defects have already been hand-seeded by other researchers, and they were ready to use when we downloaded the subject package. In addition to the hand-seeded faults, we also used MuJava tool [11] to generate a good number of mutants

to simulate realistic faults generated by the mutation operators implemented by MuJava. In order to conduct the experiments in a reason time, we randomly selected some mutant faults from the pool of mutants. We performed the mutant generation and selection for the purpose of increasing the number of faults to 10 for each program, so that we can conduct sufficient analysis and evaluation.

Table 4. The Number of Faulty Versions Created for Each Quantity of Faults

#Faults	NanoXML(v1)	XMLsec(v1)	Jmeter(v3)	Jtopas(v1)	total
2	45	45	45	45	180
3	120	120	120	120	480
4	210	210	210	210	840
5	252	252	252	252	1008
6	210	210	210	210	840
7	120	120	120	120	480
8	45	45	45	45	180
9	10	10	10	10	40
10	1	1	1	1	4
Total	1,013	1,013	1,013	1,013	4,052
#Exe	216,782	93,196	79,014	54,702	443,095

Since the study focuses on the effect of multiple faults, we repeatedly activated a desired number (range from 2 to 10) of faults simultaneously in each program. For example, for versions with two faults of NanoXML, we listed all the possible combinations of 2 out of the 10 available hand-seeded faults and mutants, and thus we activated each combination of two each time. Table 4 shows the number of versions with multiple faults for each program.

5.3. Data Collection

We ran each test case for each faulty version and captured the code coverage as well as test status (i.e., passing/failing). We compared the output of each faulty version against the output of the original version of the program that contained no faults. When the output of the faulty program was different than the output of the original program, the test case was labeled as failing. Similarly, when the outputs were exactly the same, the test case was tagged as passing. In order to get the profile, i.e. coverage information for each execution, we instrumented each program manually by inserting *print* statement in each block, where block is defined as a straight-line piece of code without any jumps. We then developed a number of Java utility programs to count the number of failing/passing test cases and capture the statistics required for each block including *ef*, *nf*, *ep* and *np*. We also implemented a Java utility program to compute the suspiciousness score based on the under study ranking metrics.

6. Results

6.1. Parallel vs. Sequential Debugging

We choose the Ochiai ranking metric to evaluate the two debugging strategies, since it is widely used in previous studies [3]. As Table 5 shows, the data are organized in one table where each row represents the average cost for each program without taking into account the number of faults. The mean value measured for fault localization cost when sequential debugging is employed 31.22%, whereas, the mean value for cost needed when parallel debugging is utilized is 30.47%. The result indicates that overall parallel debugging strategy slightly outperforms the sequential debugging, i.e., the difference is negligible and it is only around 1 percent.

In order to further investigate the significance of the difference between the two debugging strategies, a hypothesis testing was applied. Hypotheses test is a popular method in statistical inference. The study

begins with some theory, which is believed to be true or is used as a basis for argument without being proved. The question of interest is simplified into two competing hypotheses including the null hypothesis, usually denoted by H0, against the alternative hypothesis, denoted by H1. The null hypothesis H0 usually indicates that there are no differences in the mean of two groups under study, whereas, H1 indicates that there is a significant difference between the mean values of the two groups. The probability value (p-value) of a statistical hypothesis test is the probability of falsely rejecting the null hypothesis if it is in fact true. A small p-value may suggest that the null hypothesis is unlikely to be true. The smaller a p-value is, the more likely the null hypothesis is rejected. Similar to the previous study [12][13], we formulate the null and alternative hypotheses and used Wilcoxon to perform the test. Among several hypotheses tests, the Wilcoxon rank-sum test, sometimes called Mann--Whitney--Wilcoxon, is a non-parametric alternative to the two-sample t-test, which does not rely on the assumption of data belonging to any probabilistic distribution [14]. Therefore, non-parametric statistical methods are more robust than parametric methods in this scenario. Table 5 lists the result of the Wilcoxon rank-sum test for each program. The most common significant level of 0.05 was used to decide about the test. For all programs the p values are greater than 0.05, which indicate that there is no significant difference between the two debugging strategies.

Table 5. Results of the Wilcoxon Test

Program	Sequential	Parallel	p-value	Significant?
NanoXML(v1)	47.01	46.22	0.74	no
XMLsec(v1)	13.90	13.73	0.89	no
Jmeter(v3)	38.58	37.13	0.81	no
Jtopas(v1)	25.40	24.79	0.65	no

6.2. Number of Faults

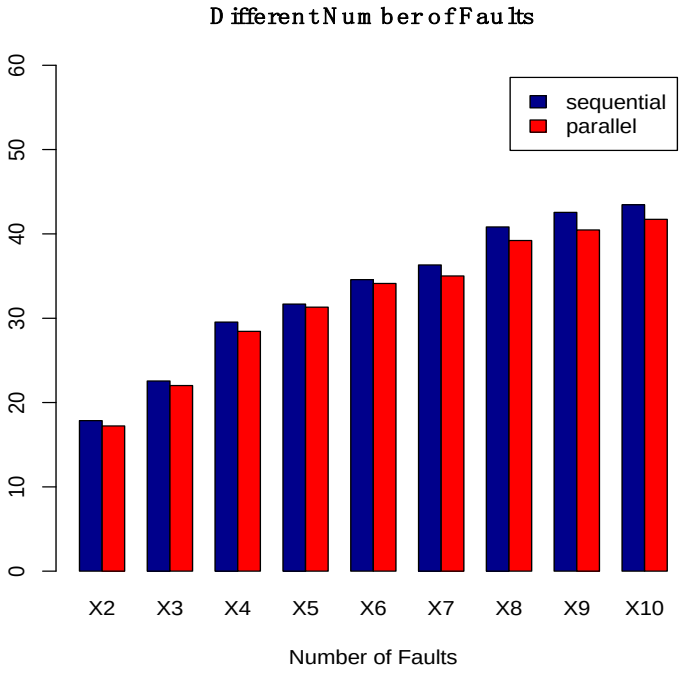

Fig. 1. Sequential vs. parallel debugging for different number of faults.

We also investigated the influence of the number of faults on the performance of two debugging strategies. Fig. 1 visualizes the results where the x-axis represent the number of faults studied and y-axis

represent the fault localization cost, i.e. the percentage of programs needed to be inspected before reaching the faulty statements. The blue and red bars visualize data for sequential and parallel debugging, respectively. It is apparent that the red bars are always shorter than the blue bars, which indicates that parallel debugging outperforms sequential debugging when faults number is in the range of 2 to 10. However, the differences do not look like very significant.

A further consideration is looking into each subject program separately. Fig. 2 depict the average fault localization cost for each program for various numbers of faults. Each curve represents a debugging strategy. In all four figures, the two curves are almost overlapped, implying that the differences between the sequential and parallel debugging in terms of the total cost is almost negligible, however, parallel debugging always slightly outperforms better. The similarity of the curves demonstrates the consistence of the result across the programs studied.

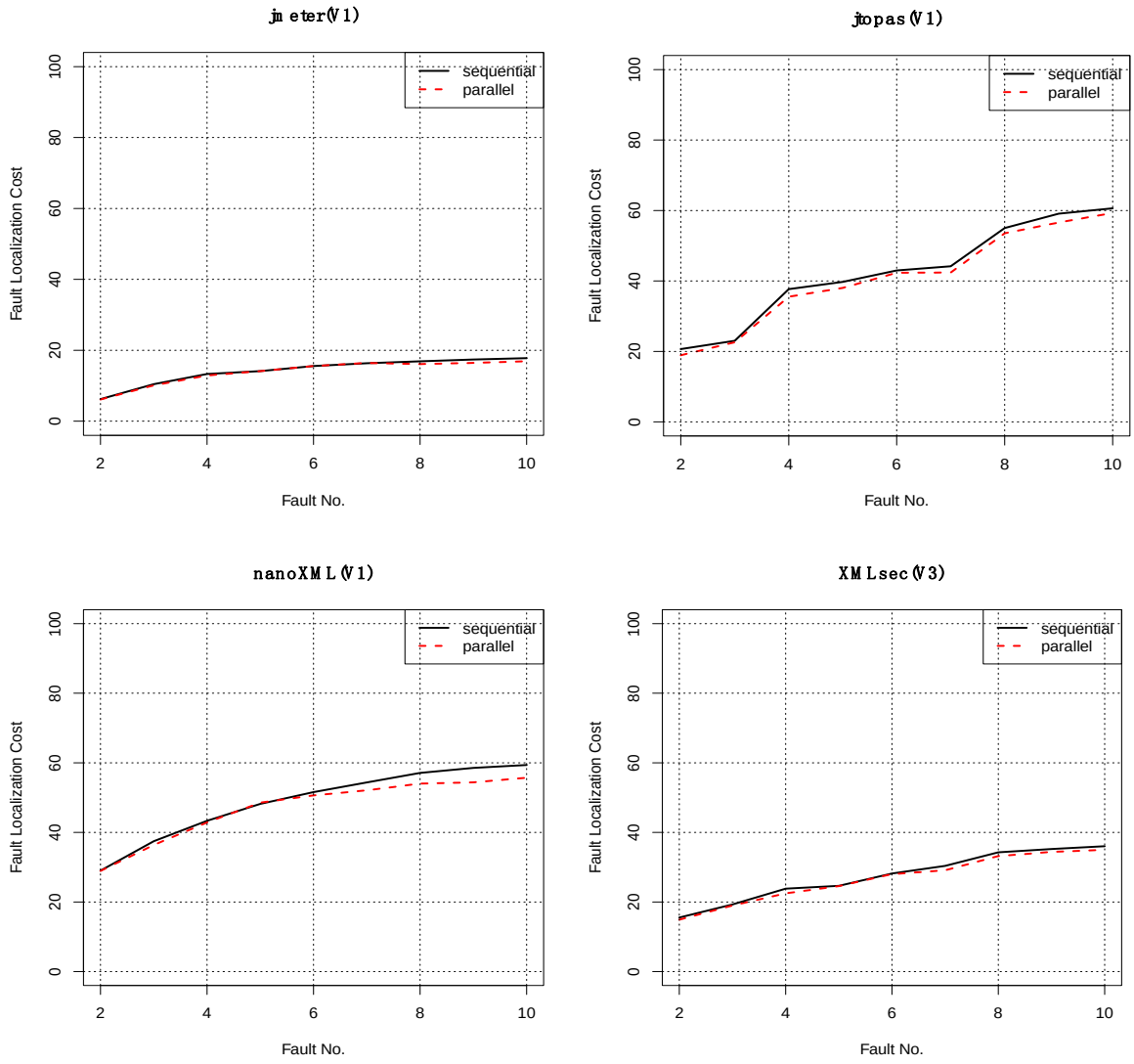

Fig. 2. The average cost of localizing each fault quantity per subject.

6.3. Different Ranking Metrics

In order to address the research question *RQ3*, we replicated the experiments on different ranking metrics. The results are illustrated in Fig. 3. The results are consistent when different metrics are employed. For Tarantula, sequential debugging outperforms the parallel debugging where the difference is around 1

percent. For the other three ranking metrics, parallel debugging is also the better choice, and the difference range between 1 to 4 percent. We also observed the conditional odds ratio is the best metric to localize the multiple faults.

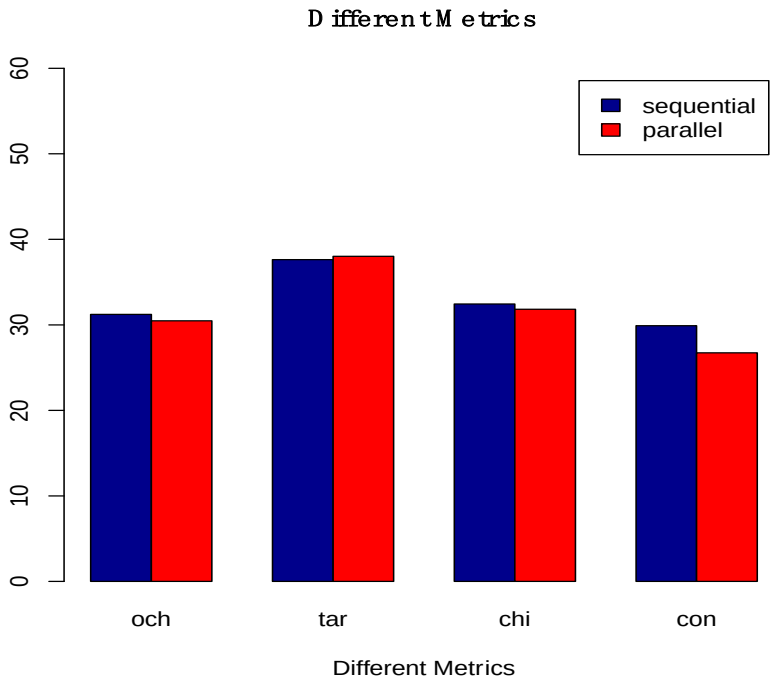

Fig. 3. The compare for nanoXML(V1).

7. Threats to Validity

External Threats. We conduct the experiments on some middle to large sized Java programs. The faults are partly hand-seeded by other researchers and partly simulated by mutation generators. Both the two kinds of faults may introduce external threats to the experiments. In order to validate the result, more study should be performed with real faults in large-scaled software programs.

Internal Threats. When computing the cost value of each ranking metric, instead of using existing tools, we developed a number of Java scripts to implement the ranking metrics. To make sure there are no internal threats introduced, we developed some other Java utility programs to track the intermediate values of the variables including *ef*, *ep*, *nf*, and *np*. Furthermore, we instrumented the original program by inserting the *print* statement manually in the block level instead of the statement level. Although this significantly reduced the cost to conduct the experiments, but different results may be obtained when statement level fault localization is performed.

Construct Threats. We proposed a new evaluation metric to measure the performance of two faults localization strategies. There exist some other ranking metrics to compute the effectiveness of faults localization techniques. The research results may vary based on different ranking metrics. In this study, we employed the plain parallel debugging technique where each processer contains one single failing test case, whereas, some techniques including test cases clustering could be adopted in the parallel debugging, and the results may be different from our study.

8. Conclusion

We return to address the research questions posed in section 4. *RQ1: Which strategy is more effective, sequential or parallel fault localization?* We conducted a number of experiments to measure the

effectiveness of the two strategies, the result in Section 6.1 indicates that overall parallel debugging outperforms sequential debugging, whereas, the different is statistically insignificant. *RQ2: Does the result vary for different numbers of faults?* Based on our empirical study and discussion in Section 6.2, we did not observe any evidence about the influence of the number of faults on the result of RQ1. *RQ3: Is the result of RQ1 consistent for various ranking metrics*? We observed that the effectiveness varies for different metrics. For Ochiai, chi-square, and conditional odds ratio ranking metrics, parallel debugging performs better. However, the sequential is the better strategy when Tarantula is adopted.

References

[1] Pressman, R. (2002). *Software Engineering: A Practitioner Approach*. McGraw-Hill, New York.

[2] Jones, J. A. & Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique, *ASE*, 273–282.

[3] Xue, X., & Namin, A. S. (2013). How significant is the effect of faulty interactions on coverage-based fault localization. *Proceedings of the IEEE International Symposium on Empirical Software Engineering and Measurement*.

[4] Xue, X., Pang, Y., & Namin, A. S., Trimming test suites with coincidentally correct test cases for enhancing fault localizations. *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference* (pp. 239–244).

[5] Jones, J. A., Bowring, J. F., & Harrold, M. J. (2007). Debugging in parallel. *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (pp. 16–26).

[6] Abreu, R., Zoeteweij, P., & Gemund, J. V. (2007). On the accuracy of spectrum-based fault localization. *Proceedings of the Academic and Industrial Conference Practice and Research Techniques* (pp. 89–98).

[7] Wong, W. E., Wei, T., Qi, Y., & Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. *Proceedings of the 2008 1st International Conference on Software Testing, Verification, and Validation* (pp. 42–51).

[8] Xue, X. & Namin, A. S. (2013). Measuring the odds of state-ments being faulty. *Reliable Software Technologies*, 109–126.

[9] Santelices, R. A., Jones, J. A., Yu, Y., & Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types.

[10] Do, H., Elbaum, S. G., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, *10(14)*, 405–435.

[11] Ma, Y., Offutt, J., & Y. R. Kwon. (2006). MuJava: a mutation system for Java. *Proceedings of the ICSE* (pp. 827–830).

[12] X. Xue, Y. Pang, & Namin, A. S. (2014). Feature selections for effectively localizing faulty events in gui applications. *Proceedings of the 2014 13th International Conference on Machine Learning and Applications*.

[13] Pang, Y., Xue, X., & Namin, A. S. (2013). Identifying effective test cases through k-means clustering for enhancing re-gression testing. *Proceedings of the 2013 12th International Conference on Machine Learning and Applications* (pp. 78–83).

[14] Siegel, S. (1956). Nonparametric statistics for the behavioral sciences.

[15] Liu, J., Zhuang, Y., & Chen, Y. (2015). Hierarchical collective i/o scheduling for high-performance computing, big data research.

[16] Liu, J., Chen, Y., & Zhuang, Y. (2013). Hierarchical i/o scheduling for collective i/o, in cluster, cloud and grid computing. *Proceedings of the 2013 13th IEEE ACM International Symposium on Cluster, Cloud and*

Grid Computing (pp. 211–218).

Yulei Pang received her M.S. degree in statistics from Texas Tech University, USA in 2012. She received the Ph.D. degree in mathematics from Texas Tech University in 2014. She is currently working as an assistant professor in Department of Mathematics, Southern Connecticut State University, USA. Her current research interests include applied mathematics and applied statistics.

Xiaozhen Xue received his B.S. and M.E. degree in software engineering from Beijing Jiaotong University, China, in 2007 and 2010 respectively. He got Ph.D. degree in computer science at Texas Tech University in 2014. He is currently an adjunct faculty member in Department of Computer Science, Southern Connecticut State University. His current research interests include software engineering and cyber security.

Akbar S. Namin received the M.S. degree in computer science from Lakehead University, Canada. He worked from 1993 to 2001 in industry and the IT field. He received the Ph.D. degree in the Department of Computer Science at the University of Western Ontario, London, Canada in 2009. Currently He is an assistant professor in Department of Computer Science at Texas Tech University. His research interests are software engineering, testing, and program analysis, cyber security and secure programming