STUDYING LOOP-BASED ANOMALIES DETECTION IN DEBUGGING LARGE-SCALE PARALLEL APPLICATIONS

Thanh-Dang Diep*, Anh-Tu Do-Mai, Nam Thoai
Faculty of Computer Science and Engineering, HCMC University of Technology, VNUHCM
*Corresponding Author: dang@hcmut.edu.vn

(Received: July 30th, 2016; Accepted: August 31th, 2016)

ABSTRACT

Debugging large-scale parallel applications is a problematic issue. Characteristics of scalability bring about an exponential increase in errors and many impacts on performance. With suffering unacceptable memory overhead and debugging time, traditional techniques, such as checkpointing or record and replay, have become obsolete when applying to large-scale parallel applications. The ex-scale trend is coming, which demands cutting-edge large-scale parallel applications debugging techniques. Instead of prior works based on locating exact errors, we propose a trace file based approach by detecting abnormal behaviors arising frequently in complicated message-passing channels. In this paper, anomalies are message leaks which probably lead to unexpected program outputs and make programmers unable to inspect manually errors. The technique utilizes one state-of-the-art detection algorithm which is related three ordered rules. The proposed algorithm is proved the precision and effectiveness by theoretical proofs and experimental results. With acceptable overhead, this technique shows the potential for applying to large-scale parallel applications in general, especially ones running on the computer clusters at Ho Chi Minh City University of Technology in particular.
Keywords: *Debugging, Message leak, Parallel, Large-scale, Unusual behavior*

TÓM TẮT

Truy lỗi cho các ứng dụng song song có tính mở rộng là một bài toán thách thức. Các đặc tính về khả năng mở rộng gây ra việc gia tăng theo hàm mũ các lỗi lập trình và gây ra nhiều ảnh hưởng nghiêm trọng đến vấn đề hiệu suất. Với chi phí hao tổn khó có thể chấp nhận được về mặt bộ nhớ và thời gian, các kỹ thuật truy lỗi cổ điển chẳng hạn như checkpointing hoặc record & replay đã trở nên lạc hậu khi áp dụng vào các ứng dụng song song có tính mở rộng. Xu hướng ex-scale đang đến gần đòi phải phải tìm ra các kỹ thuật truy lỗi hiện đại thích hợp cho các ứng dụng song song để thay thế cho các phương pháp cổ điển. Thay vì tiếp cận theo hướng tìm kiếm chính xác lỗi trước đây, chúng tôi thực hiện theo một hướng tiếp cận khác dựa trên trace file nhằm phát hiện các hành vi bất thường xảy ra thường xuyên trong các kênh truyền thông điệp phức tạp. Trong bài báo này, các hành vi bất thường là các rò rỉ thông điệp gây ra kết quả chương trình không mong muốn cho người lập trình và làm cho họ không thể phát hiện ra lỗi bằng mắt thường. Kỹ thuật truy lỗi này sử dụng một giải thuật phát hiện hiện đại dựa trên ba quy luật có thứ tự. Giải thuật được đề xuất đã được kiểm chứng về tính chính xác lẫn tính hiệu quả dựa trên các chứng minh lý thuyết và các kết quả thực nghiệm. Với phí tốn khả thi về bộ nhớ và thời gian, kỹ thuật truy lỗi này có thể áp dụng vào các ứng dụng song song có tính mở rộng nói chung cũng như là các ứng dụng song song đang chạy trên hệ thống máy tính cụm hiện có tại trường Đại Học Bách Khoa Thành Phố Hồ Chí Minh nói riêng.
Từ khóa: *Truy lỗi, Rò rỉ thông điệp, Song song, Tính mở rộng, Hành vi bất thường*

INTRODUCTION

In large-scale parallel applications with long runtime, messages exchanging happens frequently, locating exactly errors seems impossible once the execution result produced unexpectedly. The traditional debugging techniques (Thanh-Phuong and Nam, 2010, Thanh-Phuong, 2011, Thoai et al., 2002a, Thoai et al., 2002b, Thoai and Volkert, 2002) no longer support efficiently contemporary large-scale parallel applications because of the overhead of

computation and the scalability feature. Hence, our suggested debugging technique is approached by another method. The programmers focus on detecting behaviors which are called unusual behaviors, and then infer the origin of errors. Not all unusual behaviors cause errors, but these behaviors should be taken as warning points which are vulnerable to errors.

Focusing on loops is an approach which has already been studied in (Ahn et al., 2009, Laguna et al., 2012, Mitra et al., 2014, Bahmani and Mueller, 2014, Wu and Mueller, 2013). In this paper, message leak problem is main unusual behavior that is going to be discovered to debug large-scale parallel applications. A leaked message stands in loop context could lead to many other leaked ones, which makes change to the order of sending and receiving events and has serious effects on the result of execution. Thus, being able to detect the presence of leaked messages has a great help to programmers with locating errors.

The matter of trade-off between trace's size and algorithm's complexity must be advised prudently in proposing new approach. Generally, trace file must be optimized to minimize the overhead of computing and storing processes.

In this paper, a new debugging technique called loop-based unusual behaviors detecting technique is proposed to warn programmers about message leaks which manifest in loops in large-scale parallel applications. The rest of this paper is structured as follows: Section Message Leak Problem defines the message leak problem. Some important concepts that pave the way for debugging to large-scale parallel programs are also given in this section. The method to implement message leak problem and other related issues are described in section Message Leak Detection and section Implementation, whereas some evaluations and experimental results are listed in section Evaluation. Finally, Conclusions and Future Work are given in the last section.

MESSAGE LEAK PROBLEM

The parallel applications which are covered in this paper belong to message-passing model. There are two important kinds of considered events: sending events and receiving events. In the limited scope of this paper, we are not greedy to wrap up all aspects of this field; we instead just consider applications whose processes communicate by point-to-point method.

Match in pairs

A sending event is , where is denoted $RankSend$ index of process at which the sending event happens and $Dest$ is index of process which the sending event sends message to. A receiving event is denoted as $R(RankRecv, Src)$, where $RankRecv$ is index of process at which the receiving event happens and Src is index of process which the receiving event receives message from.

A sending event is called "match in pairs" in case the message sent by sending event S is received by receiving event R (See figure 1). We denoted as $S \approx R$

$$S \approx R \Rightarrow \begin{cases} RankSend = Src \\ RankRecv = Dest \end{cases}$$

Otherwise, does not match in pairs with , which denoted as $S \not\approx R$ when both S and R do not operate on the same message or the following condition satisfied:

$$\begin{bmatrix} RankSend \neq Src \\ RankRecv \neq Dest \end{bmatrix}$$

Unusual behaviors in loops

Each parallel program is often organized in a set of loop cycles which each of them is a set of iterations. Using loops brings some unpredictable troubles, especially in case of large-scale applications. Errors occurring within loop cycles are able to propagate, which explains why loops are easily vulnerable to errors.

Programmers normally intend to code a program that all sending and receiving events can match in pairs in same iteration. However, in some cases, there are few redundant sending or receiving events in different iterations, this context can be happened by three causes:

(1) A redundant sending event sent a message to a receiving event in different iteration, the same as the redundant receiving event received message from another sending event in another iteration.

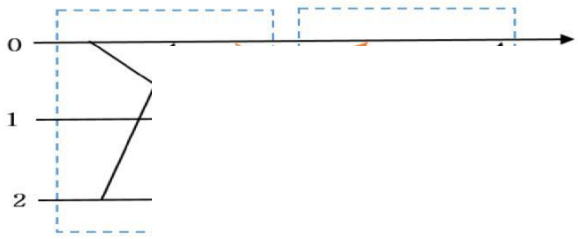

Figure 1. Leaked messages within iterations event graph

(2) No matching receiving event corresponds to a redundant sending event, even in either other iterations or other loop cycles. It means that message sent by this sending event will be received by no receiving event. The application might still run normally without any hang. The only consequence is the loss of this message, which leads to producing wrongful result.

(3) No matching sending event corresponds to a redundant receiving event, even in either other iterations or other loop cycles. This one is definitely an error that could lead to a hang. Whether the programmer intentionally or inadvertently let that happen, those cases are all considered as unusual behaviors, those sending and receiving events are all considered as unusual ones.

Definition of Leaked Message

Message leak is an unusual behavior that belongs to case (1), (2) mentioned in the previous section. Message leaks problem could leads to some unanticipated serious errors if programmers do not control the dataflow in communication well. This problem is worth considering alleviating the high rate of those errors happening.

Parallel application is often organized as a set of loop cycles $L_1, L_2, ..., L_m$, each loop cycle L_t is a set of n iterations. Each iteration $I_{t,k}$ contains two subsets $S_{t,k}$ and $R_{t,k}$, which is denoted as $I_{t,k} = (S_{t,k}, R_{t,k})$, where $S_{t,k}$ is a set of sending events and $R_{t,k}$ is a set of receiving events

$$S_{t,k} = \{ S_{t,k}[1], S_{t,k}[2], ..., S_{t,k}[p] \}$$
$$R_{t,k} = \{ R_{t,k}[1], R_{t,k}[2], ..., R_{t,k}[q] \}$$

Clearly, the number of iterations of loop cycle L_t is $|L_t|$. The number of sending events operating in iteration $I_{t,k}$ is the number of elements of subset $S_{t,k}$, which is denoted as $|S_{t,k}|$. Similarly, $|R_{t,k}|$ denotes the number of receiving events in iteration $I_{t,k}$. With these notations and formulas, the message leak problem happens within iterations when:

The case where message leak happens within loop cycles could be re-explained as:

$$\exists\, p \neq q, h, k, i, j : S_{p,h}[i] \approx R_{q,k}[j]$$

MESSAGE LEAK DETECTION

To detect automatically message leak problem, this paper suggests three following rules

Rule 1: *Message leak is detected if*

$$\exists t, k: |S_{t,k}| \neq |R_{t,k}|$$

From the definition of "match in pairs", a couple of sending event and receiving event matching in pairs must satisfy two following conditions:

$$\begin{cases} RankSend = Src \\ RankRecv = Dest \end{cases} \quad (*)$$

Because the parallel programs considered in this paper are the large-scale parallel programs which comprise of loop cycles, if $(*)$ is applied to consecutively compare each pairs of sending and receiving event in implementation, the overhead will become very high. Thus, we proposed a slightly looser system of equations but the overhead reduces significantly.

Noticing that the parallel application is assumed had passed through Rule 1, which could be:

$$|S_{t,k}| = |R_{t,k}|$$

Rule 2: *Message leak is detected if*

$$\begin{cases} \sum_{i}^{n} RankSend_{t,k}[i] \neq \sum_{j}^{n} Src_{t,k}[j] \\ \sum_{j}^{n} RankRecv_{t,k}[j] \neq \sum_{i}^{n} Dest_{t,k}[i] \end{cases} \quad (**)$$

$$With \ |S_{t,k}| = |R_{t,k}| = n$$

With this rule are used as an extra one to the first rule, it is able to detect message leaks more accurately in case the programmer makes mistake of coding by using wrong source parameter at receiving events or wrong destination parameter at sending events.

Applying two previous rules to detect leaked messages in executing large-scale parallel

applications, it is able to detect message leaks in most common cases. Nonetheless, it still contains some exceptional cases causing message leaks after the aforementioned two rules are applied. Therefore, we proposed the third rule to improve this detection accuracy.

Rule 3: *Message leak is detected if*

$$XOR\{P(S_{t,k}[1], P(S_{t,k}[2]), \dots, P(S_{t,k}[n])\} \neq XOR\{P(R_{t,k}[1], P(R_{t,k}[2]), \dots, P(R_{t,k}[n])\}$$

With

$$P(S_{t,k}[i]) = RankSend_{t,k}[i] * Dest_{t,k}[i]$$

$$P(R_{t,k}[j]) = RankRecv_{t,k}[j] * Src_{t,k}[j]$$

In this rule, exclusive-OR operator is going to be used as a simple hash to compare two sets: sending events set and receiving event set. The collision probability of hashed values using exclusive-or is $\frac{1}{2^n}$, where n is the number of bits of hashed keys. The multiplications $P(S)$, $P(R)$ expand the range of key values belonging to $[0, 2^{2n} - 1]$, this leads the collision probability to remain $\frac{1}{2^{2n}}$. Moreover, the more processes participate in, the smaller this probability will be.

The following example is a case that satisfies Rule 1 and Rule 2, but still remains leaked messages. Rule 3 is proposed to solve this case:

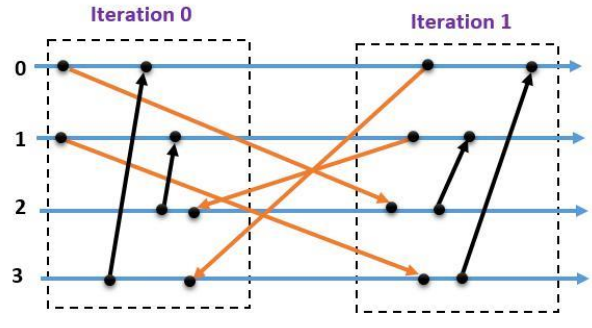

Figure 2. Message leak problem example

Figure 2 shows that, all messages sent in one iteration are received by receiving events in another iteration, which means there is existence of leaked messages within iterations. The number of sending events and the number of receiving events are equal and equal 4 in each iteration, this satisfies Rule 1 but no leaked message is detected. We also have:

$$\sum RankSend = \sum RankRecv = \sum Dest = \sum Src = 6$$

Rule 2 is also unable to apply in this case.

With Rule 3, at iteration 0:

$$XOR\{P(S_0)\} = 1, XOR\{P(R_0)\} = 0$$

Iteration 1 has:

$$XOR\{P(S_1)\} = 0, XOR\{P(R_1)\} = 1$$

Because of $XOR\{P(S_0)\} \neq XOR\{P(R_0)\}$, clearly, with Rule 3, iteration 0 has leaked messages. In addition, we also have:

$$XOR\{P(S_0)\} \wedge XOR\{P(S_1)\} = XOR\{P(R_0)\} \wedge XOR\{P(R_1)\}$$

The equation above proves that at the end of iteration 1, leaked messages are no longer leaked; all leaked ones have matched their corresponding sending or receiving events.

IMPLEMENTATION

Trace generating

With three rules in Section Message Leak Detection, leaked message detection requires only a little information of sending and receiving events. Hence, the structure of a trace file contains a set of behavior patterns corresponding to iterations in each loop:

Table 1. Pattern of iteration behaviors

IterIds
NumSend
NumRecv
SumDest
SumSrc
SumRankSend
SumRankRecv
XorSend
XorRecv

These above values will finish computation at the end of an iteration, which is called corresponding behavior pattern as Table 1. Each iteration has one and only one such pattern. Moreover, behaviors within iterations may be similar in some loops, so to prevent tracing the same pattern many times, the pattern is going to be compared with the previous one. Those values are just updated as a new pattern of iteration behaviors in case of not match comparison.

Therefore, on each process, the set of behavior patterns getting after execution is done determines information of sending and receiving events that process knew in message passing environment. Considering those sets on entire processes, gathered data is sufficient to locate leaked messages within iterations, loop cycles.

Signal functions

To apply the rules, we must instrument necessary data in loops. Where *Begin_Loop* and *End_Loop* are the beginning and the end of a loop cycle respectively, *Begin_Iteration* and *End_Iteration* are also alternately the beginning and the end of loop iterations which belong to the loop cycle. The aforementioned functions are all called signal functions of loops. Moreover, these functions also implement some tasks such as collecting, computing, storing data, etc.

```
Begin_Loop;
for {
        Begin_Iteration;
        // Code in loop
        End_Iteration;
}
End_Loop;
```

To insert the signal functions into loops, two techniques can be possible. One technique inserts the signal functions into compiler's source code while another technique transforms programmer's source code into new one included the signal functions. If applying the former, the waiting time is smaller in comparison with the latter, but implementation is very complex and thus, we use the latter to instrument essential data in loops.

EVALUATION

Our implementation is named as MessLeak. In the scope of this paper, key feature of parallel applications is scalability, so to evaluate how our approach is working; implemented experiments are going to focus on the effectiveness in lowering the overhead regarding to three aspects: (1) leaked message detection's precision, (2) traces' size and (3) trace generating time.

For (1), we used the scalable version (running this with more processes) of example in Table 2 to emphasize the role of Rule 3 in making leaked message detection much tighter. Moreover, through this experiment, also evaluate the accuracy of three detection rules. With aspects (2) and (3), we used HPL benchmark (version 2.1) with various tuning options.

All experiments were conducted on 48 core cluster with 8 compute nodes, 16 GB per node. Each measurement has been repeated three times to get average value. Our experiments just run on sufficient processes to reflect the trends of trace size and generation time growth with respect to execution scale.

Trace file size

The first experiment is going to evaluate the overhead for storing of MessLeak's trace file in comparison with several other well-known tracing tools TAU, VampirTrace and ScalaTrace. The difference of tracing purpose and amount of storing information is the main reason why we choose these tracing tools. Testing application used in this experiment is HPL. We configured MessLeak to be able to apply all three detection rules, which requires MessLeak has to store entire necessary parameters of iteration pattern as Table 1. This configuration will provide fully input data to solve message leak problem. Running this benchmark with 100 processes, we got the following result:

Table 2. HPL's trace files cross tracing tools

TAU	VampirTrace	ScalaTrace	MessLeak
6.5GB	1GB	238MB	348KB

With MessLeak, selective storing has positive effect on overhead. Although solving message leak problem has just focused on a subset of parallel applications, this experiment has emphasized the feasibility of leaked message detection debugging technique. Moreover, this satisfactory result also shows the potentiality of debugging approach by considering abnormal behaviors. The second experiment evaluates the efficiency of MessLeak when the number of iterations increases.

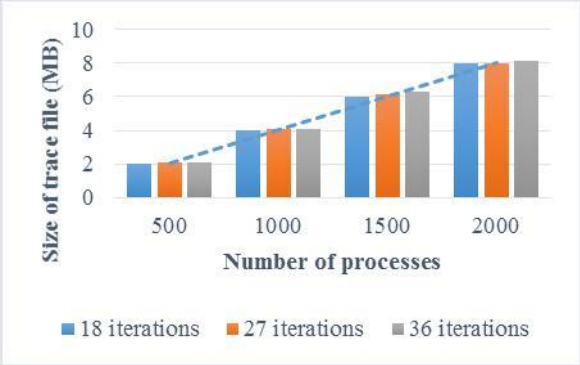

Figure 3. HPL's trace files within various iterations

Figure 3 compares the size of generated trace files when running with 18, 27, 36 iterations. Because of the similarity of iteration

behaviors, with MessLeak, the trace files' size may just have a little change when iterations have great growth. Regarding to large-scale parallel applications, consistency in iteration behaviors is a frequent existing feature, especially in case of SPMD programs. Moreover, with the increment of the number of processes, trace files' size increases in linear. Trace file size increment seems obvious when the application scales up, but MessLeak keeps that growth in linear, not exponent.

Trace generating time

In this section, the experiment is implemented to evaluate the time to generate trace files of MessLeak. We ran HPL benchmark two times: in single and in integrating with MessLeak to compare execution times each other. The benchmark is run consecutively with 500, 1000, 1500 and 2000 processes in three options: 18, 27 and 36 iterations. The results are shown in following charts:

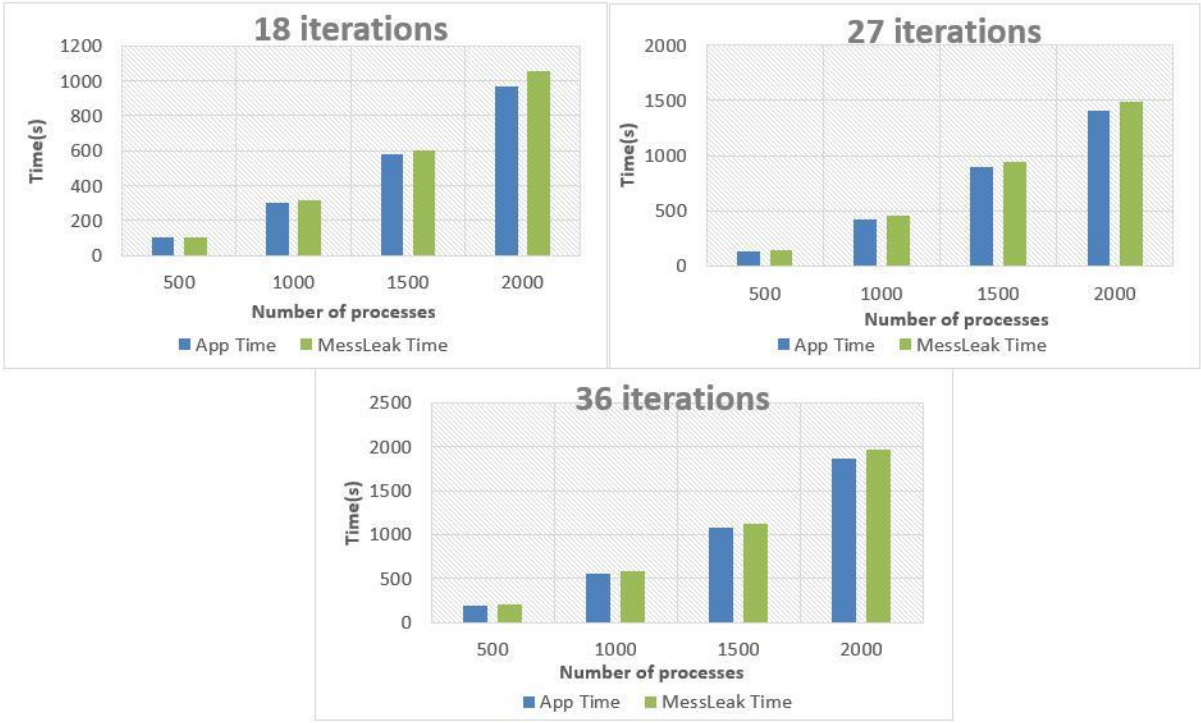

Figure 4. HPL's trace generating time in 36 iterations

From Figure 4, the time MessLeak used to generate trace files is not much different from cases running without MessLeak. The differential time if having just takes less than 5% the time this application finishes its execution. This proves that MessLeak is able to collect data to solve message leak problem without affecting much running time.

CONCLUSIONS AND FUTURE WORK

In this paper, we defined unusual behaviors manifesting in loops belonging to large-scale parallel applications. They are leaked messages which obviously can cause potential errors within loop iterations or loop cycles. In addition, we proposed a novel technique in order to help warn programmers about the message leak problem.

For future work, we have identified a number of research directions. We're going to research behaviors which use collective communication in order to cover all aspects of message passing communication. Moreover, MPI_ANY_SOURCE wild card could bring race condition, a programming fault producing non-deterministic program state and behavior due to un-synchronized parallel program executions. Race condition is very problematic to resolve in general and hence, we will also carry out the research so as to address it. Finally, we also plan to perform further experiments on more subject programs of larger size with a varying number of faults.

REFERENCES

AHN, D. H., DE SUPINSKI, B. R., LAGUNA, I., LEE, G. L., LIBLIT, B., MILLER, B. P. & SCHULZ, M. Scalable temporal order analysis for large scale debugging. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009. ACM, 44.

BAHMANI, A. & MUELLER, F. Scalable performance analysis of exascale mpi programs through signature-based clustering algorithms. Proceedings of the 28th ACM international conference on Supercomputing, 2014. ACM, 155-164.

LAGUNA, I., AHN, D. H., DE SUPINSKI, B. R., BAGCHI, S. & GAMBLIN, T. Probabilistic diagnosis of performance faults in large-scale parallel applications. Proceedings of the 21st international conference on Parallel architectures and compilation techniques, 2012. ACM, 213-222.

MITRA, S., LAGUNA, I., AHN, D. H., BAGCHI, S., SCHULZ, M. & GAMBLIN, T. Accurate application progress analysis for large-scale parallel debugging. ACM SIGPLAN Notices, 2014. ACM, 193-203.

THANH-PHUONG, P. & NAM, T. 2010. LiR: A Light Weight Replay Technique for Debugging Message Passing Programs. *International Conference on Advanced Computing and Applications.* Ho Chi Minh, Vietnam.

THANH-PHUONG, P. N., THOAI 2011. C2LiR: An approach to apply coordinated checkpointing to light weight replay technique. *International Conference on Advanced Computing and Applications.* Ho Chi Minh, Vietnam.

THOAI, N., KRANZLMÜLLER, D. & VOLKERT, J. ROS: The rollback-one-step method to minimize the waiting time during debugging long-running parallel programs. International Conference on High Performance Computing for Computational Science, 2002. Springer, 664-678.

WU, X. & MUELLER, F. Elastic and scalable tracing and accurate replay of non-deterministic events. Proceedings of the 27th international ACM conference on International conference on supercomputing, 2013. ACM, 59-68.