Characterization and Comparison of Application Resilience for Serial and Parallel Codes

Kai Wu
kwu42@ucmerced.edu
U. of California, Merced

Qiang Guan, Nathan DeBardeleben
{qguan,ndebard}@lanl.gov
USRC, Los Alamos National Lab

Dong Li
dli35@ucmerced.edu
U. of California, Merced

ABSTRACT

Soft error of exascale application is a challenge problem in modern High Performance Computing. In order to quantify an applicationfis resilience and vulnerability, the application-level fault injection method is widely adopted by HPC users. However, it is not easy since users need to inject a large number of faults to ensure statistical significance, especially for parallel version program. Normally, parallel code is more complex and requires more hardware resources than its serial code. Therefore, it is essential that we can predict error rate of parallel application based on its corresponding serial version. In this poster, we characterize fault pattern in serial and parallel code. We find first there are same fault sources in serial and parallel code. Second, parallel code also has some ***unique*** fault sources compared with serial code. Those ***unique*** fault sources are important for us to understand the difference of fault pattern between serial and parallel code.

1 INTRODUCTION

Making system resilient to hardware and software faults is a critical design goal for future extreme scale systems. To implement resilient HPC, we must have a good understanding of application resilience in the existence of faults. Currently, the application level fault injection is the major method to understand application resilience. The application level fault injection triggers random bit flip in the operand or result of a random instruction. Typically, the statistical results of many fault injection tests, e.g., the percentage of the fault injection tests that have success application outcome, is used to evaluate the application resilience.

However, the application level fault injection can be very expensive, because HPC users need to inject a large number of faults to ensure statistical significance. Moreover, comparing with fault injection for the serial code, fault injection for the parallel code can be even more expensive. First, the parallel code needs more hardware resource than the serial code to deploy fault injection tests. Second, injecting faults into the parallel code can be more difficult, because there is a larger exploration space for fault injection.

In this poster, we explore the correlation between the parallel and serial codes regarding their resilience. Our ultimate goal is that by studying the resilience of the serial code we can derive the resilience of the parallel code without using expensive fault injection. We aim to answer two fundamental questions. First, does the application resilience remain the same across the serial and parallel codes? Second, if the application resilience is difference between the two codes, what code structure causes such difference? We use an application-level fault injection tool named PFSEFI [2] to randomly choose dynamic instruction and then randomly flip one bit in the instruction result. After enough fault injection tests, we characterize and compare the serial and parallel codes based on the fault injection results. We hope that our work can lay foundation to build a model to predict the resilience of the parallel code only based on fault injection results in the serial code.

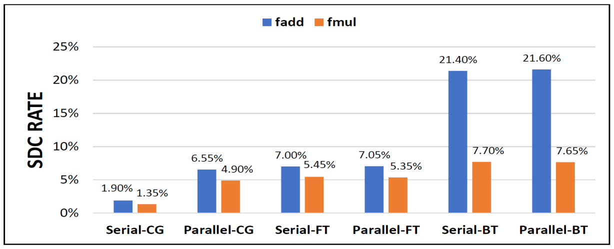


Figure 1: SDC Rate of NPB CG,FT, BT Benchmarks.

2 EVALUATION METHODOLOGY

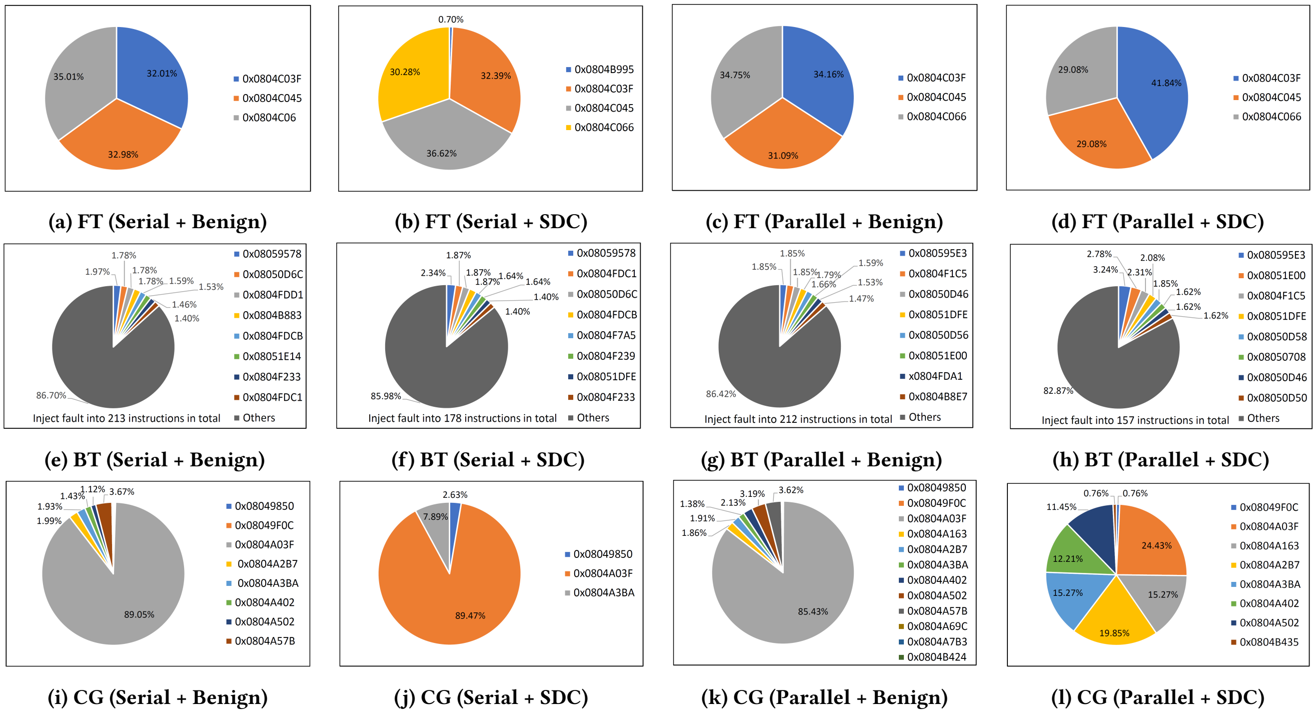
We employ a fault injection tool, PFSEFI to study three NAS benchmarks (CG, FT, BT) with the input problem S. For serial code fault injection, we only run one MPI process; For parallel code fault injection, we run four MPI processes and then randomly choose one MPI process for fault injection. We inject faults into the whole application and focus on two types of instructions, i.e., floating point addition (*fadd*) and floating point multiplication (*fmul*), because they are the most common ones in HPC applications. To ensure statistical significance for fault injection, we gradually increase the number of fault injection tests until the fault injection result becomes stable. The fault injection results are classified into three types: (1) Benign: the computation results of benchmarks pass the benchmarks' verification phase, it means the computation results are acceptable. But the computation results may be different from those without fault injection. (2) Silent data corruption (SDC): the computation results of benchmarks do not pass the benchmarks' verification phase; (3) Crashes: the benchmark cannot run to completion. Since the fault injection happens based on the random selection of dynamic instruction, we cannot know where the fault happens within the application code. But we can know the instruction address in the EIP register when the fault happens. We map the instruction address into the application code via PYELFTOOLS [1]. Based on the EIP information for all random fault injection points, we can know the occurrence frequency of each faulty instruction; also, we can analyze the code, and understand the difference or similarity of application resilience in serial and parallel codes.

3 EXPERIMENT RESULTS

Figure 1 shows the fault injection results (i.e., SDC rate of fault injection tests). We calculate the SDC rate every 1000 fault injection tests. The fault injection results become stable after 6,000 fault injection tests.

Figure 2 shows the faulty instruction distribution for the fault injection tests for floating point *add* instructions. In particular, we find that there are no crashes happened in tests of three benchmarks;

[*] LA-UR-17-26470



(a) FT (Serial + Benign) **(b) FT (Serial + SDC)** **(c) FT (Parallel + Benign)** **(d) FT (Parallel + SDC)**

(e) BT (Serial + Benign) **(f) BT (Serial + SDC)** **(g) BT (Parallel + Benign)** **(h) BT (Parallel + SDC)**

(i) CG (Serial + Benign) **(j) CG (Serial + SDC)** **(k) CG (Parallel + Benign)** **(l) CG (Parallel + SDC)**

Figure 2: The distribution of faulty floating point *add* instructions in fault injection tests.

thus we only show how frequent each instruction is selected when the fault injection results are benign and SDC.

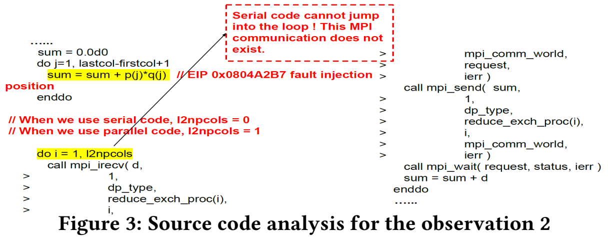
Figure 2 (a)-(d)shows that for FT, the randomly selected faulty instructions in the fault injection tests for the serial and parallel codes are the same, which explains why the fault injection results for the two codes are almost the same. The fact that the faulty instructions are the same for the serial and parallel codes mainly because of the code similarity between the serial and parallel codes.

For BT(see figure 2(e)-(h)), we find that faulty instructions are widely spread across the parallel and serial codes. There is almost no instruction similarity in those faulty instructions between the serial and parallel codes. It is because BT has complicated computation. There is no dominant computation phase where the faulty instructions can repeatedly happen.

For CG(see figure 2(i)-(l)), we find that faulty instructions are limited to a few instructions, which is very different from the cases of BT. Also, the fault injection results for the serial and parallel codes are quite different. To understand the reason for such difference, we map the faulty instructions into the source code of CG and have the following observations.

Observation 1: The instruction at 0x0804A03F is the most frequently selected instruction for fault injection. Such instruction appears in all cases (serial+benign), (serial+SDC),(parallel+benign) and (parallel+SDC). This instruction is used so often in the benchmark, such that most of faults are injected into it. Also, the corruption of this instruction seems to easily cause SDC.

Observation 2: Some instructions only appear in (serial+benign), (parallel+benign) and (parallel+SDC), but do not appear in (serial+SDC). Those instructions include those at 0x0804A2B7, 0x0804A3BA, 0X0804A402 and 0x0804A502. Those instructions cause fault injection result difference between the serial and parallel codes. Figure 3 shows the related code segment for 0x0804A2B7. In particular, the serial and parallel codes have a different value for the variable *l2npcols*, which leads to different code structure (particularly the MPI synchronization) for serial and parallel code. Such difference in the code structure makes the faulty injection at 0x0804A2B7 behave differently in the serial and parallel code.



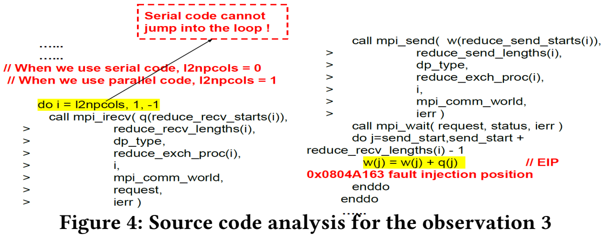
```
......
 sum = 0.0d0
 do j=1, lastcol-firstcol+1
   sum = sum + p(j)*q(j)   // EIP 0x0804A2B7 fault injection
position
 enddo

// When we use serial code, l2npcols = 0
// When we use parallel code, l2npcols = 1

 do i = 1, l2npcols
   call mpi_irecv( d,
>               1,
>               dp_type,
>               reduce_exch_proc(i),
>               i,
>               mpi_comm_world,
>               request,
>               ierr )
   call mpi_send(  sum,
>               1,
>               dp_type,
>               reduce_exch_proc(i),
>               i,
>               mpi_comm_world,
>               ierr )
   call mpi_wait( request, status, ierr )
   sum = sum + d
 enddo
......
```

Serial code cannot jump into the loop ! This MPI communication does not exist.

Figure 3: Source code analysis for the observation 2



```
......
......
// When we use serial code, l2npcols = 0
// When we use parallel code, l2npcols = 1

 do i = l2npcols, 1, -1
   call mpi_irecv( q(reduce_recv_starts(i)),
>               reduce_recv_lengths(i),
>               dp_type,
>               reduce_exch_proc(i),
>               i,
>               mpi_comm_world,
>               request,
>               ierr )
   call mpi_send(  w(reduce_send_starts(i)),
>               reduce_send_lengths(i),
>               dp_type,
>               reduce_exch_proc(i),
>               i,
>               mpi_comm_world,
>               ierr )
   call mpi_wait( request, status, ierr )
   do j=send_start,send_start +
reduce_recv_lengths(i) - 1
      w(j) = w(j) + q(j)          // EIP
0x0804A163 fault injection position
   enddo
 enddo
......
```

Serial code cannot jump into the loop !

Figure 4: Source code analysis for the observation 3

Observation 3: The instruction at 0x0804A163 is only shown in (parallel+bengin) and (parallel+SDC), and such instruction only exists in the parallel code because of the following reason: the variable *l2npcols* has a different value in the serial and parallel code. Hence the two codes behave differently, shown in Figure 4.

4 CONCLUSIONS AND FUTURE WORK

This work is a preliminary study to explain the reason for similar or different application resilience between the serial and parallel codes. For the future work, we will investigate more benchmarks and establish a model to predict application resilience for the parallel code based on the fault injection results for the serial code.

REFERENCES

[1] PYELFTOOLS. https://github.com/eliben/pyelftools.

[2] Q. Guan, N. BeBardeleben, P. Wu, S. Eidenbenz, S. Blanchard, L. Monroe, E. Baseman, and L. Tan. Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications. In *the 9th EAI International Conference on Simulation Tools and Techniques*, Prague, Czech Republic, 2016.