2016 Fourth International Symposium on Computing and Networking

A case study of the running time fluctuation of application

Kiyoshi Kumahata, Kazuo Minami
Software Development Team, Operations and Computer Technologies Division
RIKEN AICS
Kobe, Japan
kuma@riken.jp

***Abstract*—During the operation of a supercomputer, running time of an application occasionally becomes longer or shorter than that previously measured under the same conditions. This is called "running time fluctuation." Running time fluctuations disturb the efficient operation of a supercomputer and waste the consumption of precious computer resources. Thus, we have been investigating and resolving this issue on the K computer. These issues may occur for many applications and supercomputers. In this paper, we describe some causes of running time fluctuations that we encountered in the past.**

***Keywords*—running time; execution time; time fluctuation; reproductivity; de-normalized number**

I. Introduction

During the operation of a supercomputer, the running time of an application occasionally becomes longer or shorter than that previously measured under the same conditions (i.e., with the same load module and input parameters). We call this issue "running time fluctuation." Running time fluctuations disturb the efficient operation of a supercomputer controlling the execution of applications, such as a batch job using a job queueing system. Many supercomputers employ such job queueing systems. In such a situation, if the running time exceeds the specified maximum elapsed time given by a job script for the job queueing system, precious computational resources will be wasted because of the incomplete termination of the application. On the other hand, if the running time is shorter, the job running order becomes disordered. Usually, the job running order is optimized by the job queueing system using the specified elapsed time, number of nodes used, etc. as criteria. Therefore, when the running time is shortened, the job running order must be rearranged. However, such a sudden generated schedule is not necessarily appropriate. From the user's viewpoint, running time fluctuation leads to difficulty in predicting the required computational resources or suggests that the application may have a hidden bug in its calculation. We have been considering such performance issues [1] for the K computer. Although there were some relative work about performance issue of applications derived by noise [2][3][4], the noise is not necessarily a only cause of the running time fluctuation. In this paper, we present some causes of the running time fluctuations of applications that we have investigated and solved for the K computer. First, the running time fluctuation is introduced. Subsequent sections present the various causes: asynchronous input/output (I/O), access concentrations for the execution file when a program is started, de-normalized numbers, and random memory arrangement. These issues can occur for many numerical applications on other supercomputers. We believe that all supercomputer users should be aware of this issue.

II. What is Running Time Fluctuation?

Figure 1 shows an overview of running time fluctuation. Here, the execution file is made from the source file and compile options. Five runs are carried out using the same parameter. Usually, the running times should be same. However, the running times of the five runs are different. This is called running time fluctuation.

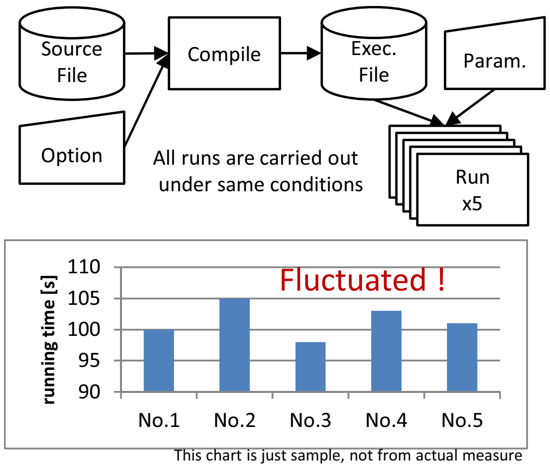


Fig. 1. Running time fluctuation

Today, many supercomputer systems manage applications as a batch job by using a job queuing system. All jobs are assigned CPU time and carried out under a previously defined schedule by the job queueing system. Therefore, running time fluctuations would disturb the efficient operation of the supercomputer. In such a situation, if the running time exceeds the specified maximum elapsed time given by a job script for the job queueing system, computational resources will be wasted due to the incomplete termination of the application.

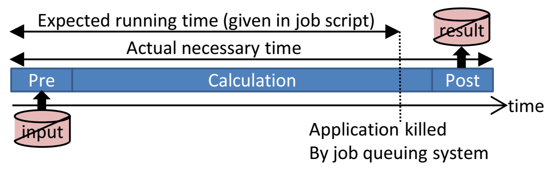


Fig. 2. Case where running time increases

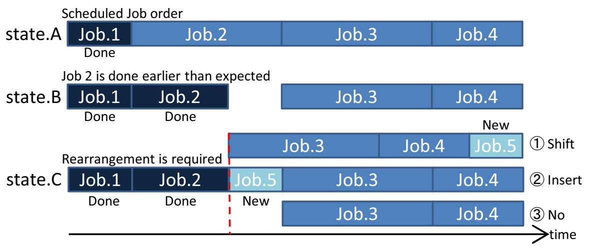


Fig. 3. Case where running time decreases

Figure 2 shows the case where the application running time increases. The application reads the input data, performs the calculation, and finally writes the output data. When the time for calculation is exceeded, the application consumes the expected running time previously set by the job script, and the application is suddenly terminated by the job queueing system. Because output data are not obtained, the consumed time has been wasted. Even though most actual calculations have a checkpoint mechanism, the wasted time should be present.

Figure 3 shows the case where the application running time is shortened. In state A, jobs are scheduled in the shown order. Job 1 is already finished, and job 2 is running. If job 2 is suddenly finished in state B, the job queuing system should reschedule state C. There are three patterns. In the "Shift" pattern, the execution start times for subsequent jobs are simply moved ahead simply. In the "Insert" pattern, a newer job 5 is inserted immediately after the time for job 2. Job 5 is chosen from queuing jobs. In this simple figure, inserting a new job seems simple. In actual cases, however, this is not so easy because many criteria (e.g., time, number of nodes, node shape) are involved in creating the job schedule. For the pattern "No," no suitable job for insertion is found. In this pattern, no jobs are performed until the time to start job 3 is reached. Thus, time is wasted. Running time fluctuations disturb the efficient operation of a supercomputer, as described above. The subsequent sections present cases of running time fluctuations that we encountered in applications on the K computer.

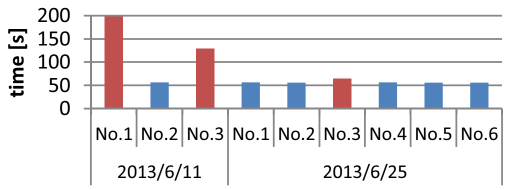


Fig. 4. Measured fluctuation of an FEM application

III. Asynchronous I/O

Figure 4 shows the running times of a finite element method (FEM) application on the K computer. Executions were carried out two different days. On both days, the times on this chart were measured under the same conditions. That is, the same execution file was derived from the same source file and used with the same compile option, input data, and running parameters. Running time fluctuation was observed on June 11. Then, we inserted detailed timer routines into the application to measure the times for numerous regions in the application.

We carried out six executions on June 25 while using the application with a timer. Most running times were nearly the same, but some running times became longer. The running time fluctuations on the two execution days had different frequencies.

Figure 5 shows the time for a range of application per call for execution No. 3 on June 25. In this chart, the horizontal axis represents a call number of this range. The vertical axis indicates the time consumed by a call in this range. Many peaks were found; however, these peaks were not derived from the difference in the number of operations between each call of this range. In this application, each routine or range had the same operations for every call. Furthermore, this application allocated a memory space for the working array once in the initial process of the main routine when the program started. The main routine assigned parts of the memory space as working arrays for subroutines. A memory address referred to by a routine did not change when called by the routine. Thus, the frequency of cache thrashing also did not change when called by a routine. Therefore, if cache thrashing led to slowdown of a range, this chart must be flat because all calls would be slowed down by the same degree. This implies that this running time fluctuation was not due to memory factors.

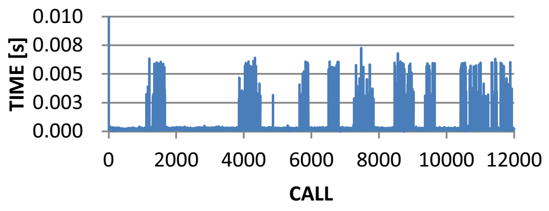


Fig. 5. Time of range in applications per call

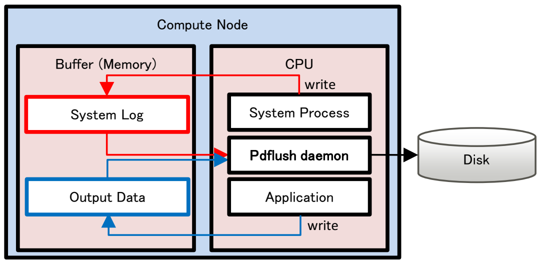


Fig. 6. Schematic chart of the asynchronous I/O

In fact, these peaks of time were derived from jitters due to the pdflush daemon of the Linux operating system. The pdflush daemon plays the asynchronous I/O. Figure 6 shows a schematic chart of the asynchronous I/O of Linux OS. The data written by an application or system process are temporarily stored in a buffer. The pdflush daemon periodically checks the buffer, and if the buffer has sufficient stored data, the pdflush daemon moves the data into a disk. Such pdflush daemon behavior disturbs the application as a jitter. The pdflush daemon was running when the peaks in Figure 5 occurred. To avoid this problem, we suggest inserting a sleep command before the application to wait for the pdflush daemon to finish moving data from system processes to prepare for executing the application. If a parallel application uses many processes, a reasonable choice is to avoid writing small output from all of the processes when the application is executed (e.g., debug output or convergence log). Many small outputs activate the data move operation of the pdflush daemon. Each process is disturbed by the pdflush daemon in each period. As a result, the synchronization between all processes requires a long time. Eventually, this causes the application to slow down.

IV. Access Concentration for the Execution File

Figure 7 shows the time of a subroutine of a parallel application for first several steps. The horizontal axis indicates the MPI rank and the vertical axis indicates time. Lines indicate the measured time for first 5 time steps. Even though each step has same operations, only the first time step consumes larger time than other time steps. Such type of running time fluctuation may occur because of the access concentration for the execution file. If a parallel application use numerous amounts of process, all processes access the execution file at same time when the application begin. Therefore I/O gets slowdown.

V. De-normalized Number

Recently, most computers employ the IEEE754 format for floating point numbers [5]. Figure 8 shows the bits expression of a single-precision four-byte floating point number bits and the formulation. For a normal number, the least significant bit (LSB) of the exponent bits is fixed to 1. Even if all fraction bits are 0, the fraction value in decimals is at least 1.0 because an implicit 1.0 is added to the fraction in the formulation. By this definition, problems may appear for sensitive calculations due to the behavior "flush to zero." For example, the subtraction between two very small numbers may be zero, which may involve an unexpected division by zero if the result of subtraction is used as a denominator. To avoid this, IEEE754 defines the state of "de-normalized number." In this state, smaller values than the normal state can be described, which lowers the possibility of unexpected division by zero. In general, the calculation of normal numbers is carried out by hardware. On the other hand, the calculation of de-normalized numbers is carried out by software in most computers. The K computer also processes de-normalized numbers with software. Therefore, calculations containing a de-normalized number are slower than calculations consisting only of normal numbers.

Figure 9 compares the calculation times of the cases with only normal numbers case and only de-normalized numbers. Both calculations contain the same number of arithmetic operations. In the case with only de-normalized numbers, however, all arithmetic operation should be carried out by software. To avoid this slowdown, the user checks to makes sure that the calculation requires de-normalized numbers. Furthermore, the programmer should check that all memory accesses refer to the correct address and confirm that all arrays and variables are properly initialized. If these are wrong, the application will use incorrect bits as a floating point number. In many cases, these incorrect bits differ for every application execution. Therefore, the slowdown aspect changes for every application execution. Many compilers have a compile option to facilitate such checks and a switch allowing or preventing the use of de-normalized numbers.

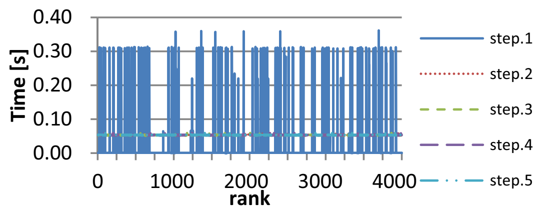
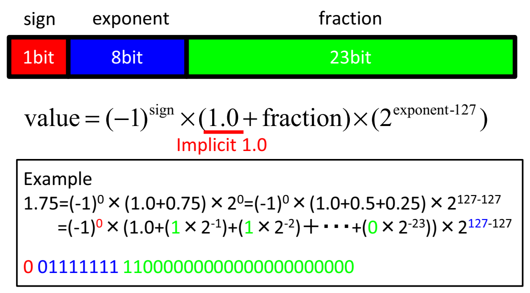


Fig. 7. Time of a subroutine of a parallel application for first several steps



$$\text{value} = (-1)^{\text{sign}} \times (\underline{1.0} + \text{fraction}) \times (2^{\text{exponent-127}})$$

$$1.75 = (-1)^0 \times (1.0+0.75) \times 2^0 = (-1)^0 \times (1.0+0.5+0.25) \times 2^{127-127}$$
$$= (-1)^0 \times (1.0 + (1 \times 2^{-1}) + (1 \times 2^{-2}) + \cdots + (0 \times 2^{-23})) \times 2^{127-127}$$

Fig. 8. Single-precision four-byte floating point number

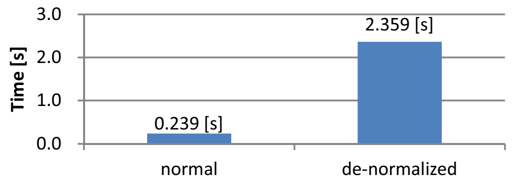


Fig. 9. Comparison of normal and de-normalized operations

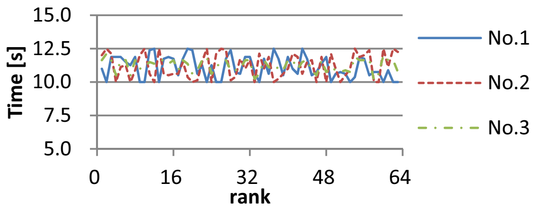


Fig. 10. Running time difference between three executions (before)

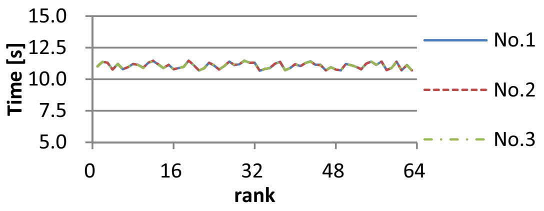


Fig. 11. Running time difference between three executions (after)

VI. Random memory arrangement

Figure 10 shows the total running time for a subroutine of a parallel application for three executions under the same conditions. The horizontal axis means the MPI rank. The vertical axis means the time. The lines indicate the measurement results. Even though the conditions were the same for all executions, the measurement results did not match.

The subroutines had different running times for three executions. After a detailed investigation, we found that the TLB thrashing occurring ratio and cache thrashing occurring ratio differed between the three executions. We used the same input data for each execution, so the memory access patterns should have been the same. Thus, the TLB/cache thrashing occurring ratio should have been the same. However, the memory addresses of arrays using the application were different for the three executions. Because the array pointers were different, even though the access patterns were the same, the TLB/cache thrashing occurring ratio differed for different executions. Therefore, the running time of this subroutine fluctuated.

A dynamic linker allocates important memory blocks of a process (e.g., heap, stack, dynamic libraries) when a program starts. At this time, those memory blocks are assigned in a random manner. This approach is called address space layout randomization (ASLR) [6] and is employed to maintain security. With this technique, applications can avoid risks from buffer overflow attacks because the return address is variable for each execution. However, the modern CPU has a no eXecute bit (NX bit) function, which can disturb buffer overflow attacks. Thus, we felt that there should be no problems without ASLR, especially for a supercomputer.

After the system was modified so that the dynamic linker would allocate memory blocks onto the same address in the memory space even if many executions were carried out, the running time fluctuation disappeared. Figure 11 shows the result of this improvement. The subroutines had the same running time for the three executions. Although there were differences between ranks, this difference was due to the unbalanced number of calculations processed by each rank, which is not a problem.

VII. Summary

Running time fluctuation may disturb the efficient operation of a supercomputer and waste computer resources. During the operation and user support of the K computer, we encountered some running time fluctuations. In this paper, we introduce some causes, asynchronous I/O, access concentration for the execution file when starting a program, the de-normalized number defined by IEEE754, and random memory arrangement. Runtime fluctuations derived from communication time fluctuation are outside the scope of this paper.

Acknowledgment

We received a great deal of support for this study from Fujitsu. We thank them and our colleagues in the Operations and Computer Technologies Division of RIKEN AICS.

References

[1] A. Kuroda, Y. Sugisaki, S. Chiba, K. Kumahata, M. Terai, S. Inoue, K. Minami, "Performance impact of TLB on the K computer applications" (in Japanese), JSPS Transactions of Advanced Computing System, Vol. 6. No. 3, 2013, pp. 1-11.

[2] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, M. Valero, "A Quantitative Analysis of OS Noise", Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, Anchorage, AK, 2011, pp. 852-863.

[3] F. Petrini, D. J. Kerbyson, S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q", Supercomputing, 2003 ACM/IEEE Conference, Phoenix, AZ, USA, 2003, pp. 55-55.

[4] T. Hoefler, T. Schneider, A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation", Supercomputing, 2010 ACM/IEEE Conference, New Orleans, LA, 2010, pp. 1-11.

[5] W. Kahan, "Lecture notes on the status of IEEE754 for binary floating-point arithmetic," https://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF, 1997.

[6] H. Shachan, M. Page, B. Pfaff, E. Jin Goh, N. Modaugu. D. Boneh, "On the effectiveness of address-space randomization," Proceedings of the 11th ACM Conference on Computer and Communications Security, 2004, pp. 298–307.