Journal of Computational Science 5 (2014) 126–134

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Statistical assertion: A more powerful method for debugging scientific applications

Minh Ngoc Dinh [a,*], David Abramson [b], Chao Jin [a]

[a] Monash University, Clayton, Victoria, 3800, Australia
[b] University of Queensland, St Lucia, Queensland, 4072, Australia

ARTICLE INFO

Article history:
Received 5 October 2012
Received in revised form 9 December 2013
Accepted 15 December 2013
Available online 21 December 2013

Keywords:
Debugging
Assertion
Statistics
Parallel architecture

ABSTRACT

Traditional debuggers are of limited value for modern scientific codes that manipulate large complex data structures. Current parallel machines make this even more complicated, because the data structure may be distributed across processors, making it difficult to view/interpret and validate its contents. Therefore, many applications' developers resort to placing validation code directly in the source program. This paper discusses a novel debug-time assertion, called a "Statistical Assertion", that allows using extracted statistics instead of raw data to reason about large data structures, therefore help locating coding defects. In this paper, we present the design and implementation of an 'extendable' statistical-framework which executes the assertion in parallel by exploiting the underlying parallel system. We illustrate the debugging technique with a molecular dynamics simulation. The performance is evaluated on a 20,000 processor Cray XE6 to show that it is useful for real-time debugging.

1. Introduction

Debugging large-scale scientific codes is difficult due to several reasons. First, many scientific codes manipulate enormous multi-dimensional data structures, which are often distributed across parallel processes, and it is impractical for a user to trace problems by focusing on individual data elements. Second, the programs often embody complex algorithms making them difficult for non-experts to follow. Using sophisticated data-mining operations can help identify outlier values in large amounts of program states. However, sophisticated data-mining operations are often more expensive to execute on a single processor. That leads to the performance issue in parallel debugging. In this paper, we address these issues and introduce a simple, yet scalable debugging technique.

1.1. Motivation

As more data is produced and gathered, statistics and data patterning (or data mining) [1] is becoming an increasingly important concept for transforming data into information. For example, data mining is popular in a wide range of profiling practices, such as marketing, finance, climate modeling, and earth systems [1]. In addition, most highly performance software, not only generates raw data, but also produces patterning information in forms of histograms, probability distributions, or data models (i.e. mathematical functions). These statistics not only give the users great insights of the observed phenomena, but also sometimes display unusual details of the computation.

Our earlier work has demonstrated that ad hoc debug-time assertions can assist in verifying program states at runtime because it is not necessary to examine every value in a large data structure [2]. The study showed that in a number of cases, a parallel computer can be used to execute the assertion logic, making it efficient when used on large data structures and machines. Recently, we recognized the importance of extracting statistical information for debugging purposes while chasing an error in one of our software tools (the debugger, in fact). Specifically, we generated a performance model based on a set of simulations, and produced a plot that summarized the model behavior with a two-dimensional graph. This simple display highlighted an error, and we subsequently found a coding bug. Importantly, the error became obvious not through the detailed examination of process state, as supported by almost all debugging tools, but through a simple proxy – a graph showing one derived variable against another. The location of the bug could be deduced quite accurately without viewing the source code, because the graph contained sufficient information about the type of error. This example highlights the potential of using statistics instead of raw data to locate coding defects.

1.2. Statistical assertion

This paper introduces a new type of ad hoc debug-time assertion called a *Statistical Assertion*. A *statistical assertion* is defined as a predicate consisting of two *data models* in the form of either

* Corresponding author. Tel.: +61 3 9902 0388.
E-mail address: minh.ngoc.dinh@monash.edu (M.N. Dinh).

1877-7503/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jocs.2013.12.002

statistical primitives (e.g. mean or standard deviation values) or data models (e.g. histograms or density functions). Statistical assertions allow the user to compare data pattern information between two data structures, instead of comparing the exact values like the assertions used in previous work [2,3]. For example, it is possible to assert that the *mean* value of a large dataset is between certain bounds during the life of a function call, or the number of elements in an array needs to be in a specific range. More advanced statistical assertions allow the user to state that the *histogram* formed by all elements in a specific dataset must be equivalent to a *histogram* formed by another dataset, or assert that all elements in a dataset should be normally distributed.

Statistical assertions are useful in debugging large scale scientific problems because (1) they allow users to focus on the scientific *meaning* of the computations instead of the exact data values produced by them, and (2) they reduce the complexity in debugging *stochastic processes*, for example as found in Monte Carlo methods. Statistics can be used to reflect scientific knowledge behind a computation, thus by using statistical assertions; a user can integrate such knowledge into the debugging process and transform it into runtime invariants that ensure the correct execution of the code at runtime. Failure to comply with such expectations will lead the users to the incorrect computation. Furthermore, stochastic processes pose a nontrivial difficulty in testing and debugging, because the program state is often nondeterministic. However, if we disregard the exact data values, the data patterns extracted from those datasets are often deterministic. Statistical assertions allow the users to capture such determinism, and make debugging stochastic simulations more practical, while reducing the complexity of processing raw data.

The essence of this approach is to (1) diminish the substantial amount of raw data to manageable "blocks"; (2) alleviate the complexity in managing data decomposition across a large number of processors; and (3) leverage parallelism to make the system fast enough for real time debugging. Statistical assertions can test the 'statistical' state of a large distributed array, and can be refined iteratively by the user in order to locate the source of an error. Because evaluation is likely to be expensive, we propose a scheme that executes the assertion in parallel, making assertions over large data structures feasible. We also discuss how partial statistical results are aggregated, and compared. These ideas are implemented in an existing parallel debugger, Guard [4].

The paper is structured as follows. In Section 2, we discuss related work and recent research in supporting scalable debugging operations. We also describe several notable efforts in using statistics for testing and verifying software correctness at runtime. Section 3 presents a statistical-debugging framework which supports the creation of user-defined statistical models and functions. Section 4 describes the required extension to the based parallel-debugger Guard. Section 5 delivers case studies to demonstrate sample use of the proposed techniques in debugging a molecular dynamics application. The analysis and evaluation of the performance obtained on a Cray XE6 system is provided in Section 6. We conclude the paper in Section 7 with some discussion about future enhancement.

2. Related work

The work discussed in this paper is a good example of Zeller's more general [5] *scientific debugging* method, in which a user invents a *hypothesis* about program behavior, and then tests it against an *observation*. Related work on the use of statistical hypothesis includes Zhou et al. [6], Daikon [7], and DIDUCE [8]. They demonstrate that *statistics-rule-based* approaches are very promising in detecting bugs that do not violate any programming rules. However, they only work with sequential programs and are not evaluated in parallel. DMTracker [9] also employs the statistics-rule-based technique and provides a solution for parallel applications. The tool can automatically detect the cause of phenomena such as data corruption or deadlocks by observing data movements between parallel processing threads. A more notable example of applying statistics for debugging is the S*tatistical Debugging* technique developed by Liblit et al. [10]. The author argues that stochastic failures can be reported multiple times and the information, extracted from reporting data via various statistical and modeling techniques, can be used to deduce the likely location of the bugs. However, the goal of DMTracker and statistical debugging technique is only to isolate a certain class of bugs namely program *runtime failure*. They obviously cannot be used to identify bugs that do not abort the operation of the program but silently corrupt the final results.

For interactive debugging paradigm, DDT [11] supports the use of a simple bar-code like snapshot to show the spread of values for a given variable, while TotalView debugger [12] provides users with a limited number of statistical functions including max/min, mean, median, standard deviation, quartiles: first and third, and upper/lower adjacent. These are statistical functions of the type we envisage. In addition, TotalView's *TVScript* allows users to perform arbitrary actions at breakpoints, similar to the GDB's *conditional breakpoints* [13]. These are promising tools for monitoring statistical features of the application. However, with these tools, it is difficult to reason about the collective state of a parallel program. Our implementation in this work provides a parallel solution implicitly.

3. Design of statistical assertions

The support for statistical assertions requires a framework that addresses two issues. First, it needs to support a wide range of useful statistics, and it is desirable to compute these in parallel in order to provide real time debugging of large datasets. Second, the framework should allow users to create arbitrary user-defined data models. The following texts focus on these issues, respectively. Details regarding the implementation are discussed in Section 4.

3.1. Split-phase statistical operation

The parallel computation of basic statistics such as average, max, min is relatively straight forward; however, more complex statistics require special handling. For example, given a dataset X, the typical standard *two-pass algorithm* for computing standard deviation [14] is not efficient in parallel, because it requires all elements in X to be examined twice. Even though there are one-pass algorithms that compute the standard deviation value, some of them are numerically unstable. However, the *pair-wise algorithm* [14] provides much better accuracy. Given $X = A \cup B$, one can compute the variance in one pass by computing sample mean value μ and the *sum of squares of differences* from μ, denoted as $M_{2,i}$, for A and B independently. These values can later be combined to calculate the overall standard deviation value using the following formulas

$$M_{2,X} = M_{2,A} + M_{2,B} + \delta^2 \frac{n_A n_B}{n_X} \quad (1)$$

$$stdev = \sqrt{\frac{M_{2,X}}{n_X}} \quad (2)$$

This algorithm can be executed in two phases, and the first phase can be parallelized (as depicted in Fig. 1). Therefore, we design the statistical reduction process as a *split-phase* operation, as below. As it turns out, this design actually maps well onto the client/server architecture of our host debugger, as discussed in Section 4 [2].