Performance Profiling and Debugging on the K computer

● Keiichi Ida ● Yasuyuki Ohno ● Shunsuke Inoue ● Kazuo Minami

We have developed application-development support tools for the K computer. This paper describes profiling functions for raising the performance of applications and debugging functions for testing applications as main functions of these tools. In developing these tools, we first defined the work procedure that a user would follow for improving the performance of an application and testing it. We then investigated the form that these application-development support tools should take for each task in that procedure. We here introduce these tools in conjunction with those tasks. Additionally, while the large-scale configuration of the K computer is one of its major features, existing profilers and debuggers for large-scale applications still have problems that have yet to be solved, and we here describe new measures for addressing those problems. We also touch upon profiling functions specifically developed for the advanced hardware of the K computer such as the high-performance SPARC64 VIIIfx processor and Tofu interconnect.

1. Introduction

The K computer[note)] is a massively parallel supercomputer consisting of more than 80 000 nodes using advanced hardware such as the high-performance SPARC64 VIIIfx processor and Tofu interconnect. We have developed application-development support tools for this supercomputer.

This paper describes the main functions of these support tools: profiling functions for achieving high-performance applications and debugging functions for testing those applications. In setting out to develop these profiling and debugging functions, we first defined the work procedure that a user would follow for achieving and debugging a high-performance application. We begin this paper by presenting our development objectives and then introduce various development support tools corresponding to the defined work procedure.

2. Development objectives

We established three major objectives in our development of profiling and debugging functions: support large-scale parallel-processing applications and advanced hardware, make use of standard interfaces, and achieve advanced graphical user interface (GUI) functions.

Like the system itself, application software for the K computer is large-scale in nature and thus must use the cutting-edge hardware making up this supercomputer. For this reason, our first development objective was to support large-scale, parallel-processing applications and new functions geared to various types of new and advanced hardware. Though explained in detail later in this paper, these functions include profiling and debugging functions supporting applications running tens

note) "K computer" is the English name that RIKEN has been using for the supercomputer of this project since July 2010. "K" comes from the Japanese word "Kei," which means ten peta or 10 to the 16th power.

of thousands of processes in parallel, a large-scale data display function for presenting the performance information of tens of thousands of processes, self-check functions for the run time system (RTS) library and Message Passing Interface (MPI) library, profiling functions supporting the performance instrumentation counters of the SPARC64 VIIIfx processor, and a communications-cost display function taking the Tofu interconnect into account.

At the same time, the current environment surrounding development support tools is dominated by a trend toward standard interfaces, the use of open source software for tool components, and the construction of software platforms independent of hardware. With this in mind, our second development objective was to give due consideration toward open systems and implement as much as possible application-development functions on standard interfaces. To be more specific, we adopted the DWARF 2[1] debugging file format as an interface between the compiler and various tools, which makes it possible to employ a variety of debug tools provided in open-source format. Additionally, we adopted the Performance Application Programming Interface (PAPI)[2] as the interface for obtaining information from the SPARC64 VIIIfx performance instrumentation counters that enable various types of CPU performance, such as number of executing instructions and number of cache misses, to be precisely measured. Finally, to implement a log output function supporting various types of events, we adopted VampirTrace[3] specifications considering their status as a de facto standard.

A high-performance GUI is needed to display the profiling and debugging information of large-scale parallel-processing applications. Our third development objective was therefore to achieve advanced GUI functions. Many existing application-development support tools use X Window System (X11) as a GUI, but this system cannot easily take advantage of performance improvements in user terminals and is limited in GUI functions and performance. To resolve these issues, we adopted Adobe System's AIR[4] runtime environment as a new GUI platform owing to its foundation in Rich Internet Applications (RIA) technology and its rapid evolution in the field of Web applications. With AIR, an AIR client receives only as much data as needed from the server and locally performs all processing for redrawing—such as when the image viewpoint changes—so that the graphic performance of the user's terminal can be fully exploited.

3. Application performance improvement steps

The significance of a supercomputer lies in its ability to perform high-speed calculations, and determining the performance of an application running on a supercomputer is therefore an essential and important endeavor. Moreover, if the performance so determined turns out to be inadequate, it is imperative that the source of such performance degradation be analyzed and efforts be made to improve the application's performance. This work for determining and improving an application's performance consists of multiple steps beginning with performance characteristics analysis (basic profiling) and continuing with a set of mutually independent performance improvement steps (detailed

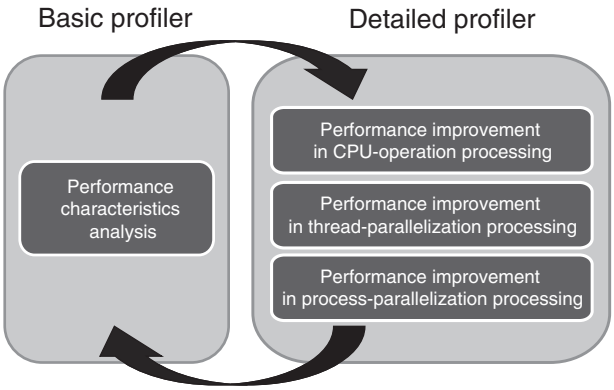

Figure 1
Application performance improvement steps.

profiling), as shown in **Figure 1**. Among the three steps in the detailed profiler, we here take up performance improvement in CPU-operation processing and performance improvement in process-parallelization processing.

3.1 Performance characteristics analysis

The objective of this first step is to obtain a comprehensive understanding of application performance under normal operating conditions. To this end, we developed a "basic profiler" with four basic functions.

1) Obtain performance information simply on the basis of highly optimized object code without re-compilation
2) Obtain basic performance information by simply adding an fipp command to an ordinary job-run script
3) Display processing conditions in terms of CPU-operation processing, thread-parallelization processing, and process-parallelization processing
4) Obtain performance information by sampling run-time data at fixed intervals, thereby minimizing and fixing overhead.

Once the basic profiler has uncovered the processing necessary to raise application performance, it is time to move on to the next steps. CPU-operation processing, thread-parallelization processing, and process-parallelization processing are mutually independent from the viewpoint of performance, and each one requires a separate approach to improving performance. Nevertheless, the detailed profiler that we have developed is used as the main tool for raising performance in each case. This profiler has three basic functions.

1) Specify the beginning and end of a measurement interval by calling a service library from within the application in order to perform detailed analysis of a specific program location
2) Obtain detailed and accurate performance information for every measurement interval by making rigorous measurements of the times that CPU processing, thread-parallelization processing, and process-parallelization processing are executed within a measurement interval in contrast to sampling
3) Individually analyze measured times instead of processing them as statistical averages, thereby supporting even cases in which performance characteristics change every time a user function is called.

3.2 Performance improvement in CPU-operation processing

The high-performance SPARC64 VIIIfx processor incorporates various types of performance instrumentation counters, which can be used to classify the total execution time of an instruction sequence in terms of CPU operating states (executing instructions, waiting for memory access, waiting for an operation to complete, etc.). This method, which is commonly called cycle accounting,[5] enables developers to determine where in the processor bottlenecks are occurring and to analyze and improve processor performance. If a performance problem is known to exist in CPU-operation processing, the developer can use the hardware monitor function of the detailed profiler to analyze the problem using the cycle accounting method (**Figure 2**).

To give an example of a performance-improvement procedure, more effective use of the cache would have to be achieved if it was found by cycle accounting that waiting for memory access was occupying a high percentage of execution time. Specifically, this could be done by changing the application's data structure (spatial measure) and the order of data referencing (temporal measure) and increasing the reusability of data (cache-line measure), thereby improving the cache hit rate and decreasing memory access. The SPARC64 VIIIfx extensions include single instruction multiple data (SIMD) instructions, which enable multiple operations to be executed

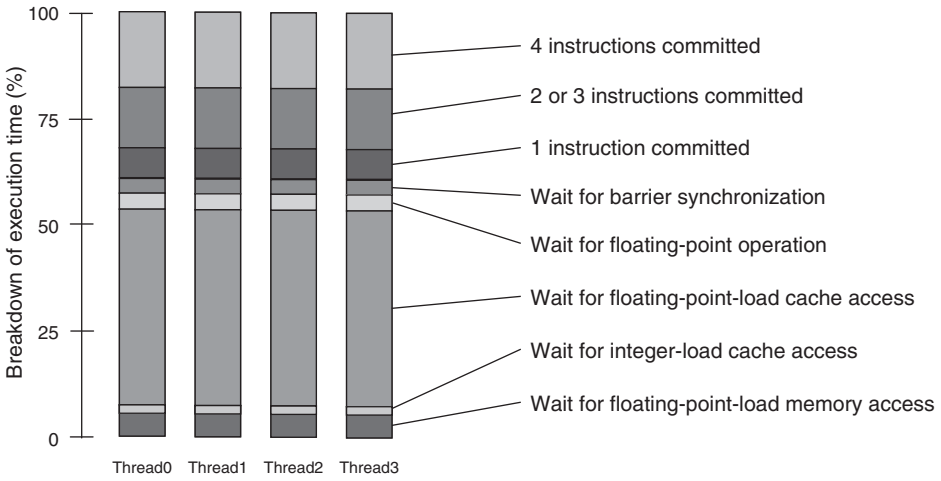

Figure 2
Example of analysis using cycle accounting method.

at one time. Attention should therefore be paid to the ratio of SIMD instructions used in order to obtain high performance. If it is determined that the SIMD-instruction ratio is low, efforts should be made to optimize the compiler by selecting appropriate compiler options and/or rewriting program source code.

3.3 Performance improvement in process-parallelization processing

As described earlier, the K computer is a massive system consisting of more than 80 000 nodes, but, in addition to this huge array of hardware, it also supports large-scale applications. For example, a job consisting of more parallel processes than the total number of nodes can be run on the K computer. There are two main approaches to improving performance in process-parallelization processing as described below.

The first approach is to achieve good load balance in process parallelization. In general, an MPI library is used to run application processes in parallel, but, if loads differ among the processes, communication and synchronization waits can prevent high operating performance from being achieved. Disorder in load balance can occur for various reasons. For example, it can be a problem inherent to the application, such as different degrees of computational complexity among the subdivided regions of the application created for parallel processing. This can cause subtle differences in the layout of virtual memory between nodes, resulting in variation in cache miss rates and chaotic application performance. Of course, an application that runs processes in parallel generates much communication processing, which means that communication-related variation can also occur due, for example, to collisions in the communication network.

It is important that a comprehensive understanding of current performance values be obtained to correct these various types of disturbances in load balance. Both the basic profiler and detailed profiler incorporate new means of providing large-scale visualization of various types of performance data to give developers an overall view of application conditions. Examples of cost-distribution displays as provided by the basic profiler are shown in **Figures 3** and **4**. In these screen shots, the basic profiler is analyzing performance when running the CG kernel (conjugate gradient, irregular memory access, and communication) the NAS Parallel Benchmarks.[6] The lower half of these screens shows sampling cost

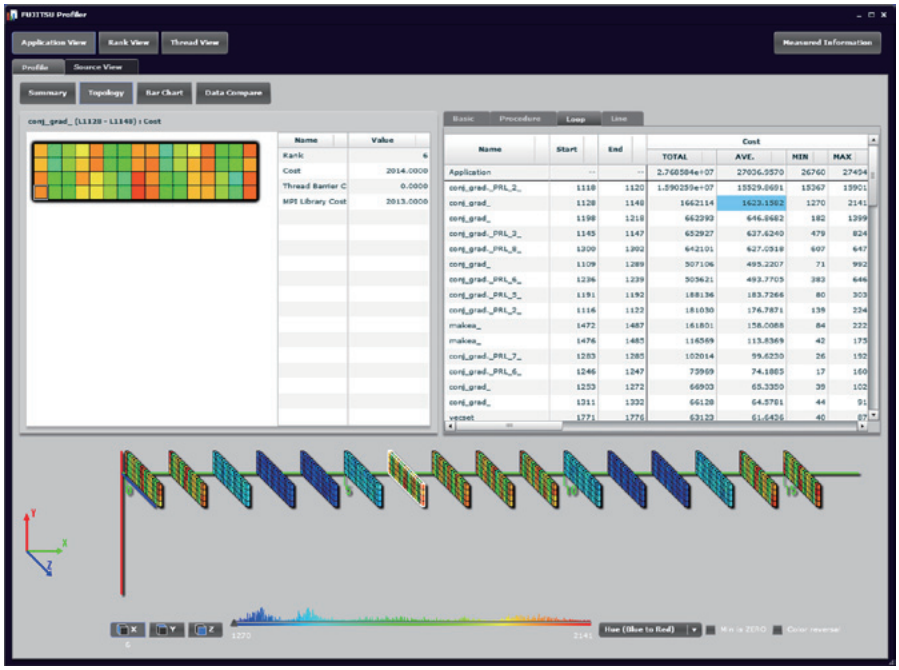

Figure 3
Screen shot 1 of cost analysis by basic profiler (topology display).
Upper-left frame: cost information for each process (partial close-up)
Upper-right frame: list of performance values for each program loop
Bottom frame: cost information for each process (complete 3D topology display)

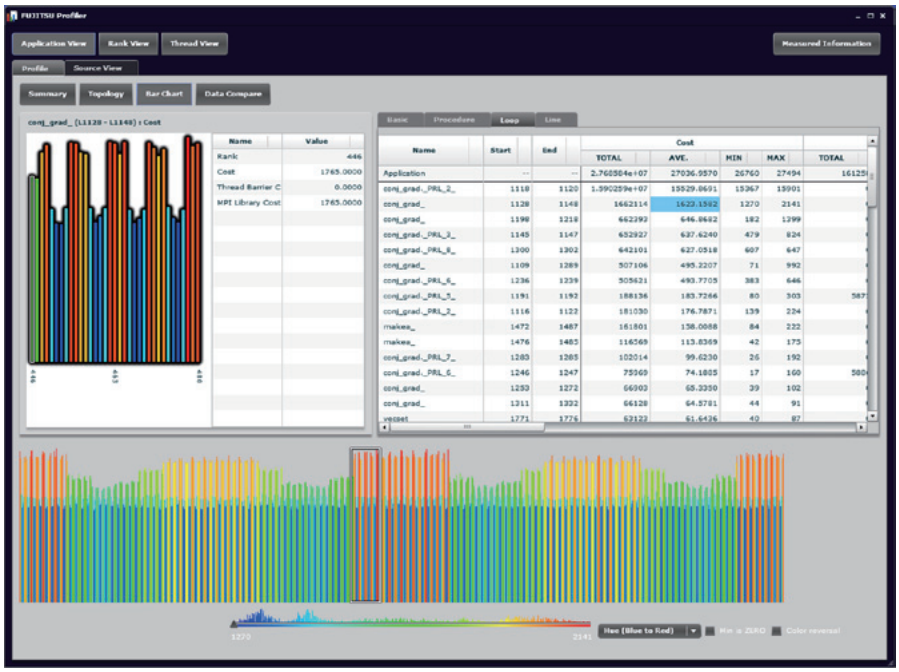

Figure 4
Screen shot 2 of cost analysis by basic profiler (1D display).
Upper-left frame: cost information for each process (partial close-up)
Upper-right frame: list of performance values for each program loop
Bottom frame: cost information for each process (complete 1D display)

corresponding to elapsed time of one program loop (rows 1128–1148 of conj_grad) in a parallel application running on 1024 processors. In Figure 3, the format of the display is "application topology" having a 16×4×16 three-dimensional torus structure running along the x-axis (i.e., in the horizontal direction in the figure). Here, the magnitude of sampling cost for each process is expressed in color. An application topology display of this type makes it easy to see how the cost of executing this loop changes along the x-axis. In Figure 4, the same data is converted into one dimension and shown in the form of a bar graph, which enables a developer to accurately determine the extent of load imbalance among the processes. The lower portion of this screen shows a histogram representing sampling-cost distribution above a legend explaining the color usage. Displaying performance data in this way (application topology, one-dimensional display, or histogram) gives the developer a bird's-eye view of cost-balance conditions across the entire application.

The second approach to improving performance in process-parallelization processing is to reduce MPI library cost. When running a parallel application that involves frequent communication among processes, MPI-library processing increases, which increases costs and degrades performance. The countermeasure to this problem is to revise the method of using the MPI library such as by reducing communication processing and reducing the number of messages (through message aggregation). In the detailed profiler, specifying a certain interval returns detailed information on MPI-library functions executed in that interval (such as number of calls, processing time, and average message length for each function). This information can be used to check the effect of the performance-improvement measures described above. In addition to using the detailed profiler, MPI library cost can also be obtained via the "MPI statistical information" function in the MPI library. This function logs information on MPI-library internal processing and outputs statistical information when execution of the target application completes.

Optimizing communication processing in the K computer requires that the 3D torus network (Tofu interconnect) and advanced barrier communication function of this supercomputer be taken into account. For this reason, the detailed profiler and MPI statistical information function have additional routines for measuring communication distance (number of communication hops in the torus) and for measuring the performance of the advanced barrier communication function.

Finally, the state of packet processing on the Tofu interconnect must be understood to optimize MPI-library communication processing. To this end, we developed a function called "Tofu performance analysis (PA) information" for measuring the number of send/receive packets, the number of send/receive bytes, the remaining capacity of send/receive buffers, etc. for each network router on the Tofu interconnect. These measurements can be initiated by specifying a certain interval in the detailed profiler. Analyzing the results of these measurements and taking appropriate measures to balance out communication processing in the interconnect network and avoid packet collisions can raise the interconnect usage efficiency and therefore improve the performance of the application.

4. Application testing steps

As mentioned above, the significance of a supercomputer lies in its ability to perform high-speed calculations, but it must also be able to perform those calculations correctly. It can be said that calculations in an application are being performed correctly if the application is operating in line with the specifications established at design time. Checking that this is so is called "testing." For ultra-large-scale parallel systems like the K computer, the work of testing becomes increasingly difficult as system complexity

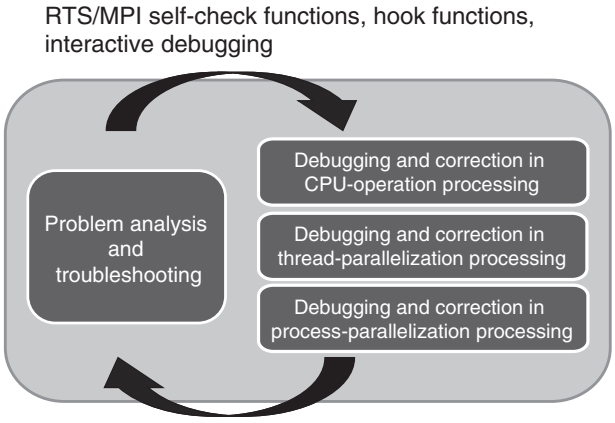

Figure 5
Application debugging steps.

increases. This situation calls for new measures such as the self-check functions described below.

Among the steps making up application testing, the work of debugging, which begins with the occurrence of a bug or bugs, consists of problem analysis and troubleshooting followed by several mutually independent debug steps (**Figure 5**). We here introduce a debugging technique made up of these steps and take an in-depth look at debugging of CPU-operation processing and process-parallelization processing.

4.1 Problem analysis and troubleshooting

Bug phenomena can be classified into a variety of types such as abnormal termination caused by a SIGnal SEGmentation Violation (SIGSEGV) or other event, deadlock, and mistakes in calculation results. The process of analyzing such phenomena in detail and isolating the position where the bug occurred is called "troubleshooting." Here, we first consider troubleshooting of CPU-operation processing, thread-parallelization processing, and process-parallelization processing.

The troubleshooting procedure begins by analyzing the bug, such as by executing the program while varying compilation-time options and run-time options or using a debugger utility. The procedure can therefore be complicated, but it is systematic just the same. Typical troubleshooting techniques are presented below.

For an abnormal termination arising from a SIGSEGV error, the location of the abnormal occurrence can be displayed by using a "traceback map," which shows the function calling relationships in the application. Using it, a developer can troubleshoot the problem on the basis of the calling relationships. For example, if the traceback map shows that an abnormal termination occurred in the MPI library, the developer can conclude that the problem is one inherent to process-parallelization processing.

Next, one effective means of dealing with an application deadlock is to attach an interactive debugger to the application using the job ID. With this technique, the interactive debugger is attached to the halted application from the outside, and control of the application is passed to the debugger. This enables the developer to use interactive operations to investigate why the application has stopped running. A screen shot of debugging using an interactive debugger is shown in **Figure 6**. The upper-left frame shows the stop location for each halted process or thread. The developer can troubleshoot the cause of the deadlock by analyzing this information.

4.2 Debugging of CPU-operation processing

As described above, the work of testing becomes increasingly difficulty as system complexity increases in a large-scale application, which calls for new testing measures. As one such measure, we focused on enhancing and improving the performance of the "built-in debugging function," which is used to automatically detect problems such as mistakes in array subscripts and variables, initialization leaks, and inappropriate memory release. We have developed a new, high-speed detection mode that improves the performance of this function by several tens of times in actual

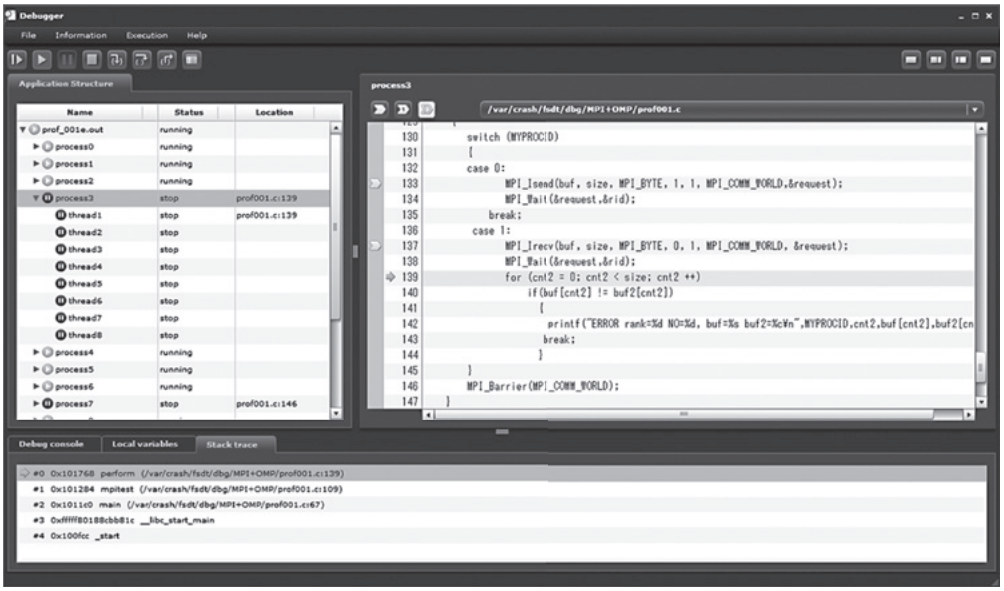

Figure 6
Screen shot of MPI-application debugging using an interactive debugger.
Upper-left frame: run state of each parallel process and thread (showing location of stop where applicable)
Upper-right frame: breakpoint locations and present location of stop in source code
Bottom frame: stack trace

applications, as described elsewhere in this issue.[7] This enhancement enables developers to apply the built-in debugging function to even more applications in actual operations.

At the same time, bugs that originate in faulty software logic such as mistakes in calculation results may often go undetected by the built-in debugging function. To deal with bugs of this type, checking on the application side must be intensified, and one effective means of doing so has been the "printf debug" technique. Using the printf function, an application programmer displays intermediate values to test calculation results on the basis of execution logic (also called "Debugging by WRITE statements" in Fortran language). As applications increase in scale with a degree of parallelism in excess of 1000, the effectiveness of debugging using an interactive debugger utility becomes limited. This has forced developers to embed test code in the application itself in many cases. To therefore simplify the printf debug technique, we have prepared a new "hook function." This is a mechanism by which the language processing system (compiler or RTS) automatically calls a specific user subroutine at appropriate times such as at the entrance/exit points of subroutines, the beginning and ending of thread parallel processes, and the calling of MPI functions. Executing application test code at hook locations in this way enables more accurate printf-debug results to be achieved while minimizing the number of changes to program code.

4.3 Debugging of process-parallelization processing

If troubleshooting has determined that the problem in question is inherent to process-parallelization processing, there is a high possibility of a bug in parallelization processing using the MPI. For this reason, we have incorporated a new function in the MPI library called "dynamic debug during MPI program execution."

This dynamic debug function detects deadlocks (suspension of communication

waiting), monitors for corruption of the MPI communication buffer by overwriting, and checks the arguments of each MPI function call. It enables mistakes in the use of the MPI library to be automatically detected on the MPI-library side.

Furthermore, to facilitate the tracing of MPI function calls, we have developed an "MPI function hook" as a means of applying the printf debug function described above to MPI processing. This hook provides a mechanism for calling a specific subroutine at the time of an MPI function call. It enables even source-program line numbers to be obtained, making it possible to create advanced test code for debugging purposes. We have also prepared a VampirTrace event log function that outputs MPI-library usage history in Open Trace Format (OTF)[8] so that testing can be performed by checking a trace log.

5. Conclusion

This paper introduced application-development support tools for the K computer. The development of these tools began by first defining the work procedure that a user would follow for achieving and testing a high-performance application and investigating the form that the tools should therefore take. We found that approaches that addressed the ease-of-use of the GUIs for the profilers, interactive debugger, and other tools and the diverse problems associated with advanced hardware and a large-scale configuration made it possible to achieve our initial objectives.

We take pride in the fact that these application-development support tools will contribute to the enhanced performance and testing of the world-class "Grand Challenge" applications slated to be run on the K computer.

References

1) The DWARF Debugging Standard. *http://dwarfstd.org/*
2) Performance Application Programming Interface (PAPI). *http://icl.cs.utk.edu/papi/*
3) VampirTrace. *http://www.tu-dresden.de/zih/vampirtrace*
4) Adobe AIR. *http://www.adobe.com/jp/products/air.html*
5) SPARC64 VIIIfx Extensions (April 26, 2010). *http://www.fujitsu.com/downloads/TC/sparc64viiifx-extensions.pdf*
6) NAS Parallel Benchmarks. *http://www.nas.nasa.gov/publications/npb.html*
7) K. Taki et al.: Compiler Technology That Demonstrates Ability of the K computer. *Fujitsu Sci. Tech. J.*, Vol. 48, No. 3, pp. 317–323 (2012).
8) Open Trace Format (OTF). *http://www.tu-dresden.de/zih/otf/*

Keiichi Ida
Fujitsu Ltd.
Mr. Ida is engaged in the development of application development support tools.

Shunsuke Inoue
RIKEN
Mr. Inoue is engaged in research and development of application software for speeding-up.

Yasuyuki Ohno
Fujitsu Ltd.
Mr. Ohno is engaged in the development of application development support tools.

Kazuo Minami
RIKEN
Mr. Minami is engaged in research and development of application software for high parallelization and high performance.