Integrating Performance Tools with Large-Scale Scientific Software*

Meng-Shiou Wu[1], Jonathan L. Bentz[1], Fang Peng[1], Masha Sosonkina[1],
Mark S. Gordon[1,2], Ricky A. Kendall[3]

[1]Scalable Computing Laboratory
Ames Laboratory, U.S. DOE
Ames, Iowa 50011, USA
{mswu,jnbntz,fangp,masha}@scl.ameslab.gov

[2] Department of Chemistry
Iowa State University
Ames, Iowa 50011, USA
mark@si.fi.ameslab.gov

[3]National Center for Computational Science,
Oak Ridge National Laboratory, PO Box 2008 MS6008,
Oak Ridge, Tennessee, 37831-6008, USA
kendallra@ornl.gov

Abstract

Modern performance tools provide methods for easy integration into an application for performance evaluation. For a large-scale scientific software package that has been under development for decades and with developers around the world, several obstacles must be overcome in order to utilize modern performance tools and explore performance bottlenecks. In this paper, we present our experience in integrating performance tools with one popular computational chemistry package. We discuss the difficulties we encountered and the mechanisms developed to integrate performance tools into this code. With performance tools integrated, we show one of the initial performance evaluation results, and discuss what other challenges we are facing to conduct performance evaluation for large-scale scientific packages.

1. Introduction

The development of high performance scientific computing software has a history of three decades or more. Some of these software packages are extremely popular today and have been under active development for multiple decades (thus even though they have a long history of development, they do not fit into the exact definition of "legacy code"). This software is usually developed with a certain programming language, using a particular programming model chosen when the project started. With the progress of parallel architectures, the programming language and programming model may no longer be the best choice to provide the best performance, thus modernization is required to provide optimized performance on the latest high performance computers.

When modernizing a scientific software package, the first step is to identify where performance bottlenecks may occur. Many large-scale scientific software packages were developed before performance technologies emerged, and usually have performance evaluations on a coarse-grain level only. Given the complexity of concurrent parallel architectures, it is not easy to identify the cause of performance bottlenecks with only coarse-grain level performance data, and detailed performance evaluations are required. With the maturation of modern performance tools, detailed performance evaluations of large-scale scientific software can be as simple as inserting the right Performance Evaluating Functions (PEF) in the right places. While this seems to be an easy task given that many modern performance tools provide automatic instrumentation mechanisms, there are several difficulties that must be overcome to incorporate modern performance tools into large-scale scientific software.

Existing performance tools such as TAU [12], PAPI[9],

*This research is supported by Iowa State University under Contract No. DE-AC02-07CH11358 with the U.S. Department of Energy and in part by the Scientific Discovery through Advanced Computing SciDAC 2006 award. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231

KOJAK [8], Paradyn [7], IBM's XProfiler and others [3, 1], each have different functionalities to provide different aspects of performance testing of software. For example, TAU provides aggregate profiling data, KOJAK has tracing and analyzing functionality, PAPI can be used to access hardware counter information, and Paradyn can be use for binary code instrumentation. To use these tools, we expect software developers to have a thorough understanding of the source code of target software, so that they know where to insert PEF. For a large-scale software package, a developer usually knows only part of the overall software. Moreover, the software development cycle for a large-scale scientific packages has usually been established for many years. Adding performance tools may increase the burden for developers and may require changes to the development cycle. While these problems can be categorized as software engineering problems, successfully overcoming these problems is vital to incorporating performance tools into large-scale scientific software.

One might argue that we could simply use automatic instrumentation mechanisms provided by some modern performance tools and this would solve the problems listed above. The drawback of this approach is that automatic instrumentation usually instruments all subroutines in a software package. In a large-scale scientific software package there are usually thousands of subroutines; instrumenting every subroutine is likely to incur a huge amount of overhead, and also produce performance information too difficult to manage. While some threshold may be set to reduce the number of subroutines instrumented to minimize instrumentation overhead, some expected information may also be lost. From our experience, in many scenarios *manual instrumentation is unavoidable*.

The focus of this paper is to show our approach to solving these software engineering problems that are closely related to the success of incorporating performance tools into large-scale software. We first discuss the difficulties we encountered in using modern performance tools on large-scale scientific software. We then discuss the tools and mechanisms we developed to overcome these difficulties. While we use our approach on one computational chemistry software package, the methodology is a generic one and should be available for application in any scientific code that requires detailed performance evaluation.

2. Large scale computational chemistry packages

Computational chemistry has become a vital element to chemistry research, an equal partner with experimental analysis tools. As the available computational methods increase in accuracy and breadth of capability, it becomes increasingly important to concurrently improve both the efficiency of the computations and the availability of the computational chemistry software, so that it is accessible to the widest set of possible problems. Computational chemistry codes have very broad applicability, ranging from combustion to homogeneous and heterogeneous catalysis, to solvent effects and surface science, to solid state chemistry and physics, to protein folding and biochemical analysis, and a whole host of other problem classes.

There are several large-scale computational chemistry packages available; some are commercial products while others are freely available. Three publicly available large scale computational chemistry software packages from DOE are GAMESS [11], NWChem [5] and MPQC [4]. Each package provides it own performance results, usually on a very coarse grain level such as the total runtime of major computations. If a computation does not scale well on a particular platform, it is usually not easy to identify performance bottlenecks with this coarse-grain level performance data. Our goal in this research is to incorporate modern performance tools into one of these large scale chemistry software packages, GAMESS, which has been under development for over two decades by many chemists around the world.

2.1 GAMESS

The General Atomic and Molecular Electronic Structure System (GAMESS) is an *ab initio* quantum chemistry program, which has been under development for more than twenty years. GAMESS is able to perform a wide range of quantum chemistry computations including Hartree-Fock (HF) wave functions (RHF, ROHF, UHF, GVB, and MCSCF) using the self-consistent field method. It is installed on many high performance computing systems, including those at most DOE, DOD, and NSF supercomputer centers, many academic institutions, and widely in the private sector. It is also part of the standard benchmark suites employed, for example, by NERSC, by the High Performance Computer Modernization Program, and by several computer companies (e.g., IBM). The number of GAMESS users is estimated to be on the order of 100,000 to 150,000. Most of the source code of GAMESS is written in FORTRAN 77 since it was the most popular programming language for scientific computing at the time the project started.

Most GAMESS computations can be run in parallel. GAMESS uses a specialized communication library, DDI (Distributed Data Interface), for parallel computation. DDI is designed with the goal of making GAMESS run on any parallel architecture. Its basic communication protocol is TCP/IP; it can also use MPI or another communication library if it is available on the target platforms. The details of the latest DDI architecture are described in [2, 10].

The GAMESS source code is developed mostly by

chemists worldwide. A GAMESS algorithm developer is usually in charge of a certain part of GAMESS, and hands the portion of the completed code to the project manager for testing. New versions of GAMESS are released once or twice a year, with intermittent minor revisions released as appropriate.

2.2 Integrating modern performance tools with GAMESS

In GAMESS, performance information is provided by the *TIMIT()* subroutine, which returns the runtime and CPU utilization between two consecutive *TIMIT()* calls. The performance information is provided on a coarse grain level: it shows the total run time and CPU utilization of a whole computation, or some major computations within a long computation. If we want to know how much time is spent in communication, how much time I/O takes or cache utilization for a certain code segment, we need to incorporate external performance tools to retrieve the performance data of interest. Developing those tools from scratch is not a choice since it may take years to develop and there are existing tools to use.

Performance tools developed during the last decade can provide more insight to find performance bottlenecks. However, we need to find the right PEF and instrument them in the right places. Some existing performance tools such as PDT [6] provide automatic instrumentation mechanisms that automatically instrument every subroutine. While this can save the trouble of finding the right places to instrument PEF, this cannot be used directly on large-scale software that contains thousands of subroutines, especially for scientific computing software where many subroutines are called millions of times within just a few minutes of a computation. Using naive automatic instrumentation not only may produce too much information to be useful, but the overhead of instrumented code can be too large, so it may significantly increase the runtime of the computation.

To reduce the overhead caused by instrumented performance codes, modern performance tools such as TAU provide more flexible mechanisms for automatic instrumentation. It allows users to define a threshold to indicate that if a subroutine is called up to a certain number of times, we do not want to profile it at all. While this can greatly reduce runtime overhead with instrumented codes, some information of interest may be lost. For example, in GAMESS the subroutine *QOUT()* is used for I/O in some computations and it is called repeatedly. But if we simply set a threshold value we may not get the correct time spent in I/O. Another case that may be solved with manual instrumentation only is when we are interested in several code segments within a subroutine. The ENTRY statement in FORTRAN 77 must also be handled manually[1].

A software engineering problem we encountered is that GAMESS is developed by many scientists that do not have a computer science background. The complexity in modern parallel architectures results in many performance tools that provide a large set of PEF to explore different aspects of a system. Finding the right tools and functions to use can be a daunting task. While the learning curve for using basic performance subroutines is not high, asking every application scientist to learn all these tools to proceed with performance evaluation can be very counterproductive. Moreover, adding performance subroutines may complicate the software development procedure that has been used for years. On the other hand, performance tool developers or computer scientists usually do not have much knowledge of the details of scientific packages and can not really define the sections of code that are really of importance to the performance.

To sum up, our solution has to overcome the following difficulties:

1. We have to assume the worst-case scenario, i.e., performance tool developers have limited knowledge of GAMESS source code, and GAMESS algorithm developers have limited knowledge about performance tools and performance optimizations, or prefer to focus on chemistry research instead of spending time learning different performance tools.

2. The approach to be used must be able to utilize different existing performance tools; We want to relieve application scientists of the burden of learning different performance tools and be able to obtain performance data easily.

3. The tool we develop for GAMESS must be an add-on and has to cope with different versions of GAMESS releases. It must be able to handle the dynamic developing procedures of GAMESS and at the same time not interfere with the original GAMESS development procedure.

In the next section we discuss our tool GPERFI and mechanism for addressing the above difficulties. GPERFI is a complementary tool to the existing "off-the-shelf" performance tools in that it utilizes them whenever possible, but adds mechanisms when GAMESS requires more specificity than available tools provide.

[1]The ENTRY statement allows program execution to be transfered to the middle of a subroutine, directly from some other routine in the program

3 GAMESS performance tools integrator (GPERFI)

Our solution to overcome the difficulties discussed in the last section is to create a master instrumentation file for each GAMESS version and a flexible instrumentation program that can automatically instrument *any* PEF provided by existing performance tools. The master instrumentation file is constructed from many instrumentation files provided by GAMESS algorithm developers (application scientists), and GPERFI can decide which PEF or tools to use to generate expected data based on the master instrumentation file. The contributions of both GAMESS developers and GPERFI developers are described next.

3.1 Algorithm developers

Since our goal is that the algorithm developers (mainly chemists) should be able to acquire performance data easily, and we do not want the already established software development procedure to be interrupted, the changes on the algorithm developer side should be minimized.

In our procedure, algorithm developers need only to define the "area of interest". This includes a file or a group of files, a subroutine or a group of subroutines, or code segments. In the case of I/O, subroutines or READ/WRITE identifiers can be specified (details of this will be discussed in Section 4.1). This information can be specified in a file and sent to GPERFI developers. The original software development procedure does not need to be changed.

To help the algorithm developers facilitate the procedure of defining their instrumentation file, we have developed some useful tools. For example, we have a tool to show the function calling tree of a computation that an algorithm developer is interested in. Then he/she can decide how detailed this calling tree should be instrumented and an instrumentation file can be generated automatically containing all the files and subroutines used by that portion of the function calling tree.

Instrumentation files from the algorithm developers can be put together by GPERFI developers as a master instrumentation file. For a different version of GAMESS, the only thing required to be handled manually is the code segment section, which indicates what code segments within a subroutine are of interest. This can be maintained by either the algorithm developers or GPERFI developers.

3.2 GPERFI developers

On the performance tools developer side, we have to deal with two cases: GAMESS code sections *with* and *without* an instrumentation file. For sections of GAMESS code without instrumentation files, we use the automatic instrumentation function provided by PDT, with settings to exclude subroutines called more than one million times. We also use some statistical data from experiments to identify that some subroutines take very little time and are called rarely; in that case we do not instrument them at all.

When the instrumentation file is available for a certain GAMESS code section, our automatic instrumentation program is used to instrument PEF based on what kind of performance data is expected. Since different PEF from different performance tools may be required, these functions are stored as templates, and instrumented by GPERFI in the places specified by algorithm developers in the instrumentation file. At this time, we have templates which include some PEF from TAU, PAPI and our own PEF. If we need any new performance subroutines from a different performance tool, all we need to do is include the subroutine as a template. Then our program can instrument it into the GAMESS source code.

We first developed GPERFI based on the July 2005 release of GAMESS, then we applied it to the February 2006 release. All we need to do is check the correctness of the "code segments" section in the instrumentation file, then we can apply the tool on the newer version of GAMESS. With this procedure, the GAMESS algorithm developers only have to define their own instrumentation file. GPERFI developers can utilize whatever performance tools necessary for the tasks, and the performance evaluation can be conducted without interfering with GAMESS development by scientists.

The process of generating an instrumentation file is a collaborative procedure whereby the GPERFI developers and GAMESS developers must decide on a set of subroutines and/or code segments that are important for performance evaluation. This procedure may take some time for a scientific package of the scope of GAMESS. When we can clearly define an instrumentation file for the whole GAMESS package, we can put "tags" in the GAMESS source code and the instrumentation program can check those "tags" to instrument performance codes. As we mentioned, GPERFI is a complementary tool to existing performance tools in that we want to utilize functionalities provided by different "off-the-shelf" performance tools whenever possible. We have observed the "co-habitation" of some performance tools, e.g., we can use PAPI subroutines within TAU. Figure 1 presents an overall view of the GAMESS performance evaluation cycle when GPERFI is applied.

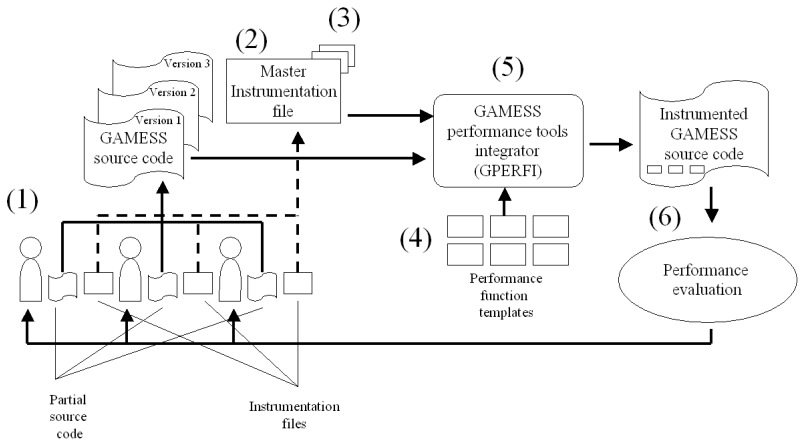

Figure 1. The procedure of performance evaluation in GAMESS. (1) Algorithm developers provide instrumentation files. (2) A master instrumentation file is constructed. (3) For a different version of GAMESS only a small section of the master instrumentation file has to be changed. (4) PEF from different performance tools are stored as templates. (5) GPERFI instruments GAMESS source code with templates according to the master instrumentation file. (6) Performance evaluation and the results are sent back to GAMESS algorithm developers.

4 Incorporating performance tools with GAMESS

Before we started integrating performance tools with GAMESS, we consulted the GAMESS algorithm developers concerning the performance data that are most interesting to have from the point of view of a chemist, and the following has emerged.

(a) Time spent in I/O.
(b) Time spent in major routines (SCF, Correlation Methods, CI, C.C., MP2, etc.).
(c) Parallel vs. serial execution.
(d) Cache utilization.
(e) CPU utilization.
(f) Memory, CPU, I/O tracing.
(g) Time spent in communications.

The first four items in the list are profiling metrics, while (e) and (f) fall into the tracing category, and (g) may be viewed as either profiling or tracing metrics. Some of the performance data can be generated by directly incorporating performance measurements on the function level while others need special handling. In this section, we describe those that require special design and help of performance evaluation tools to generate performance data for GAMESS.

4.1 Profiling of I/O operations

Profiling/tracing the cost of I/O can be very easy or very difficult, depending on the method that I/O is implemented. In GAMESS, one source file *iolib.src* contains many I/O subroutines for in GAMESS computations, but many subroutines do not use it for I/O in GAMESS. In some computations READ/WRITE statements are used for I/O instead of using subroutines in *iolib.src*. For example, the coupled cluster algorithm uses *WRITE(INTG, xxx)* or *READ(INTG, xxx)* for I/O (INTG is an integer and it is what we call a READ/WRITE identifier in this paper). Some developers even write their own I/O subroutines to handle I/O.

We provide approaches for each scenario in I/O instrumentation. First, subroutines in iolib.src can be instrumented automatically with different levels of granularity. For example, we can instrument only the top level read subroutine such as *DAREAD*(which calls *RAREAD*, which in turns calls *RARD*, then *READ*), or instrument every subroutine in iolib.src.

If the algorithm developers indicate which READ/WRITE identifiers are of interest, we can just put them in a list. GPERFI will automatically insert performance code before and after each READ/WRITE statement with the identifiers of interest. For example, to measure how much time is spend in I/O in the coupled cluster algorithm, we only need to put INTG in the instrumentation list, and then every statement with READ(INTG) or WRITE(INTG) will be instrumented.

If we already know a certain subroutine is in charge of I/O for a computation, and for I/O only, we can put the subroutine name in the instrumentation list. For example, put *QOUT, PKFILE, DAREAD* and *RAREAD* into the instru-

mentation list and then we can obtain the total I/O cost during a conventional HESSIAN computation.

In our performance results, all the subroutines or code segments that are related to I/O are then put together to evaluate the total time spent in I/O for a computation. This data can then be used to determine which parts of the code may need optimization.

4.2 Profiling of cache utilization

Acquiring cache utilization, together with MFLOP information requires access to hardware counters. This can be achieved by instrumenting PAPI functions. While PAPI subroutines can also be accessed through TAU, in our system we put PAPI functions as templates and use those templates directly.

4.3 Parallel vs. serial execution

Existing performance tools such as TAU can show how much time is spent exclusively in sequential execution. By using *paraprof* provided with TAU, we can observe subroutines that are executed sequentially by a single process and subroutines that are executed in parallel. Incorporating KOJAK to provide tracing data of parallel vs. serial execution is currently under development.

4.4 Profiling of communications

With PEF provided by tools such as TAU, aggregate communication cost is probably the easiest performance metric to develop. By adding TAU's instrumentation code to the entry and exit points of each DDI subroutine the instrumentation procedure is complete.

4.5 Tracing of CPU utilization

Some performance tools provide tracing functionality. While those can be used in GAMESS, we also developed our own tracing subroutines in case we need performance information not provided by these performance tools. For example, we want CPU utilization currently used by GAMESS in tracing results. This is basically the same information as provided by the *TIMIT()* subroutine in GAMESS. It has been used by GAMESS for many years and we expect many GAMESS users would like to keep this performance measuring ability.

4.6 Tracing of memory usage

A function called *rmemory* in TAU can provide the available physical memory. However, it is not available on Linux platforms. To access the available physical memory, we design our own subroutines to retrieve available physical memory. Computational chemistry software usually requires a very big chunk of memory. When there is not enough physical memory the program will not even start. Displaying the available physical memory during runtime can help us determine the best method to use for a certain part of computation.

4.7 Tracing of communications

Besides the profiling data of communications from TAU, we also want tracing results such as those provided by VAMPIR for MPI based programs. While tools such as MPE [1] or such as VAMPIRTRACE (now Intel Cluster Tools [3]) can instrument MPI based programs and provide tracing data, we can not use those tools for DDI directly. Currently we are exploring approaches to instrument MPE or KOJAK subroutines to DDI, so that tracing data of communications can be available to application scientists.

5 Performance evaluation of GAMESS

Performance testing of computational chemistry software is extremely complex and difficult. In particular, GAMESS has many different functionalities which utilize complex algorithms. Traditional run-time complexity analysis depends on a single input parameter N, the input size. Chemistry computations have multiple input parameters. The input molecule is one input parameter, and the execution time is roughly proportional to the size of the molecule. The basis set is another input parameter and again the execution time is proportional to the basis set[2]. But there are also other more detailed parameters, such as SCF iterative convergence threshold values. If high accuracy convergence thresholds are chosen then an iterative calculation may have to execute many times before convergence is achieved. Thus, changing a threshold value (and leaving the rest of the input parameters the same) may result in a large (unforeseen) increase in execution time. An exhaustive performance test of every computation type with a reasonable subset of molecules and basis sets would be cost-prohibitive. How to conduct performance evaluation for GAMESS with integrated performance tools and provide useful information for optimizations is an ongoing research project. In this section, we show what type of performance bottlenecks may be found when GPERFI is employed.

The default GAMESS execution model is to compute most of the data (used in iterative calculations) once, store it on disk, and retrieve it whenever it is needed. This is

[2]When discussing the run-time complexity of computational chemistry algorithms, the value of N chosen is often taken to be the number of basis functions.

called *conventional* execution. There are a number of calculations where one can specify an optional *direct* execution, which eliminates the need to store the data on disk, but instead recalculates the data "on-the-fly" whenever it is needed. It is clear that the direct method results in more CPU cycles being used for the entire calculation, especially when this data is needed in an iterative calculation. However, in many cases—especially in parallel execution—the extra CPU usage is preferred to using disk I/O since it may lead to a greater scalability. The performance evaluation results in Figures 2 and 3 are from a *conventional* calculation of the molecular system (hereafter referred to as CARB), which is an unpublished result of a Berry pseudo-rotation in the 6-membered carbaphosphatrane system. The molecular system size is $C_{10}O_3PH13$ and the basis set has 370 atomic orbitals, performing an RHF gradient calculation. The two testing platforms are listed as follows:

1. Seaborg: IBM cluster where each node has 16 power3 processors running at 375 MHz with 16 GB memory.
2. Bassi: IBM cluster where each node has 8 power5 processors running at 1.9 GHz with 32 GB memory.

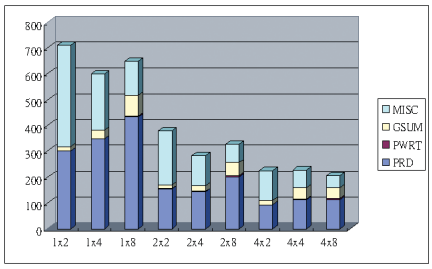

Figure 2. Performance of conventional CARB computations on Bassi. Parallel read is the major performance bottleneck on SMP machines.

For presentation clarity, Figures 2 and 3 distinguish only subroutines that are relevant to performance bottlenecks. In particular, the performance bottlenecks exist in the parallel read (shown as *PRD*) and parallel write (shown as *PWRT*) routines of GAMESS. The time spent in computational rather than bottleneck routines is combined under *MISC* in both Figures. We use $i \times j$ to represent the execution configuration that uses *i* nodes with *j* processors per node.

On Bassi, the run time of *PRD()* increases with the increase in processors per node. On Seaborg when the number of processors is increased to 16 per node, the latency of parallel write *PWRT()* is more than half of the total runtime.

Parallel read/write were developed in GAMESS before the MPI-IO standard emerged and implemented in a straightforward way, such that, every process reads/writes its own portion of data to/from the disk. With more processes used per node, the total volume of data may exceed I/O bandwidth even though the amount of data for read/write diminishes.

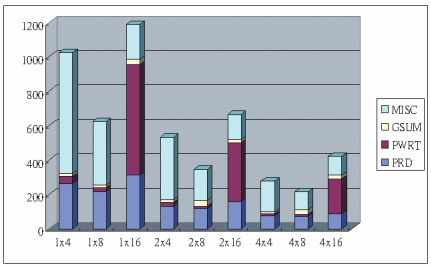

Figure 3. Performance of conventional CARB computations on Seaborg. Besides parallel read as the performance bottleneck, the latency of parallel write jumps dramatically when all the 16 processors are used within an SMP node.

We also observed that, on Bassi, the parallel read and DDI global sum (bars*PRD* and *GSUM*) are not performing better than those on Seaborg, given the same execution configuration (e.g., 1×4 or 1×8), although Bassi is a more powerful computer than Seaborg. While performance tools are useful in exploring performance bottlenecks of a computation, we are also interested in learning what parameters may be affecting performance the most under different execution configurations. To accomplish this, we would need to collect the data generated by performance tools, put this data into a database, and analyze it.

6 Discussion and future work

In this paper, we present an approach to incorporating modern performance tools into a large-scale computational chemistry package. While existing performance tools provide mechanisms for easy integration with applications, several difficulties have to be overcome in order to use the functionality provided by performance tools on a large-scale scientific software package. We developed GPERFI, such that it helps to keep the application development procedure almost intact, to minimize the burden of the performance tools usage, and at the same time, to proceed with performance evaluation without excessive overhead typically incurred by instrumented performance codes. Although currently GPERFI can handle only FORTRAN 77 code, we plan to extend its capabilities to F90/C/C++ to enable its usage by a variety of scientific packages.

While conducting experiments, we realized that more

tools are needed to conduct performance evaluations for large-scale applications. For the existing state-of-the-art performance tools, the generated performance data still has to be managed manually by users. For example, with a large number of computations provided by GAMESS, handling manually the performance data is error-prone and time consuming. Thus, our next step is to develop tools to handle performance data automatically, so that the performance evaluation process can be facilitated.

Performance comparisons between different architectures and different execution configurations are needed but currently can only be handled manually too. The focus of performance research has been mainly on the development of tools for data collection and evaluation. With the complexity of current parallel architectures and the amount of performance data, tools to help scientists to handle performance data should not be ignored. In the case of GAMESS performance evaluation, the complexity of chemistry computations may require building a database and standard testing procedures to make a comprehensive performance evaluation possible. This is a long-term effort and will require collaborations between the computational chemistry community and performance tools developers.

References

[1] CHAN, A., GROPP, W., AND LUSK, E. Scalable log files for parallel program trace data. *(draft)* (2000).

[2] FLETCHER, G. D., SCHMIDT, M. W., BODE, B. M., AND GORDON, M. S. The Distributed Data Interface in GAMESS. *Comp. Phys. Comm., vol.128* (2000), 190–200.

[3] INTEL. Intel Cluster Tools, http://www.intel.com.

[4] JANSSEN, C. L., SEIDL, E. T., AND COLVIN, M. E. Object-Oriented Implementation of Parallel Ab Initio Programs. *Parallel Computers in Computational Chemistry* (1995).

[5] KENDALL, R., APRA, E., BERNHOLDT, D., BYLASKA, E., DUPUIS, M., FANN, G., HARRISON, R., JU, J., NICHOLS, J., NIEPLOCHA, J., STRAATSMA, T., WINDUS, T., AND WONG, A. High Performance Computational Chemistry: An Overview of NWChem a Distributed Parallel Application. *Comp. Phys. Comm., vol.128* (2000), 260–283.

[6] LINDLAN, K. A., CUNY, J., MALONY, A. D., SHENDE, S., MOHR, B., RIVENBURGH, R., AND RASMUSSEN, C. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. *Proceedings of SC2000: High Performance Networking and Computing Conference* (2000).

[7] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer 28, 11, Special issue on performance evaluation tools for parallel and distributed computer systems.* (1995), 37–46.

[8] MOHR, B., AND WOLF, F. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003),* (Klagenfurt, Austria,, June 2003).

[9] MOORE, S., CRONK, D., WOLF, F., PURKAYASTHA, A., TELLER, P., ARAIZA, R., AGUILERA, M., AND NAVA, J. Performance Profiling and Analysis of DoD Applications using PAPI and TAU. In *Proceedings of DoD HPCMP UGC 2005, IEEE* (Nashville, TN, United States, June 2005).

[10] OLSON, R. M., SCHMIDT, M. W., GORDON, M. S., AND PENDELL, A. P. Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Model. In *Proceedings of Supercomputing* (Phoenix, Arizona, United States, November 2003).

[11] SCHMIDT, M., K.K.BALDRIDGE, J.A.BOATZ, ELBERT, S., GORDON, M., JENSEN, J., KOSEKI, S., MATSUNAGA, N., NGUYEN, K., SU, S., WINDUS, T., DUPUIS, M., AND MONTGOMERY, J. General Atomic and molecular Electronic Structure System. *Journal of Computational Chemistry, vol.14* (1993), 1347–1363.

[12] SHENDE, S., AND MALONY, A. D. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications, Vol.20* (2006), 287–331.