Performance Profiling and Analysis of DoD Applications Using PAPI and TAU

Shirley Moore, David Cronk, and Felix Wolf
University of Tennessee-Knoxville, Knoxville, TN
{shirley, cronk, fwolf}@cs.utk.ed

Avi Purkayastha
University of Texas at Austin, Austin, TX
avijit@tacc.utexas.edu

Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava
University of Texas at El Paso, El Paso, TX
{pteller, raraiza, maguilera}@cs.utep.edu, jenava@utep.edu

Abstract

Large scientific applications developed as recently as five to ten years ago are often at a disadvantage in current computing environments. Due to frequent acquisition decisions made for reasons such as price-performance, in order to continue production runs it is often necessary to port large scientific applications to completely different architectures than the ones on which they were developed. Since the porting step does not include optimizations necessary for the new architecture, performance often suffers due to various architectural features. The Programming Environment and Training (PET) Computational Environments (CE) team has developed and deployed different procedures and mechanisms for collection of performance data and for profiling and optimizations of these applications based on that data. The paper illustrates some of these procedures and mechanisms.

1. Introduction

Developers and users of DoD applications running on high performance computing (HPC) platforms need easy-to-use tools for determining how well their applications are performing and for analyzing and improving application performance. Profile data, based on both time and hardware counter data, can be collected to determine the proportion of performance metrics that are attributable to different program regions, allowing performance analysis to focus where it will have the greatest effect and to help identify code hot spots and performance bottlenecks. Once problem areas are identified, detailed tracing of the application can give further insight into causes of the problems.

The PET Computational Environments (PET CE) team has developed and deployed mechanisms for automated collection of performance data using the PAPI cross-platform interface to hardware performance counters[1], developed at the University of Tennessee, and the TAU (Tuning Analysis and Utilities) performance tool suite[2], developed at the University of Oregon. Users need only make a couple of simple changes to their application makefiles in order to have TAU automatically instrument their parallel applications written in FORTRAN 77/90/95 or C/C++ to collect profile and/or trace data. The PAPI hardware metrics to be collected can be specified at runtime by setting environment variables. Once the instrumented application is run, the resulting performance data can be viewed and analyzed using the TAU command-line or graphical analysis tools, the KOJAK automated performance analysis system that uses a pattern matching approach to detect performance bottlenecks[3], or multivariate statistical analysis tools developed as part of the CE-KY4-002 project.

The PET CE team has worked with the HPCMP Benchmarking team to develop standard definitions for different levels of performance data and a standard database schema. A password-protected code profiling database has been set up at the University of Tennessee. Users may upload their performance data to this database so that the data can be analyzed in collaboration with the PET CE team. Alternatively, users can use the schema to implement their own local code profiling database. The TAU Performance Data Management Framework (PerfDMF) tools[4] can be used to upload data to the database and the analysis tools described above can be used to retrieve and analyze data stored in the database.

Systematic approaches towards collection and analysis of performance data and subsequent optimizations for a number of Department of Defense (DoD) applications have been carried out as part of the CE-KY4-003 Scalability and Performance Optimization Team (SPOT) project. As HPC systems grow in size and capability, they also continue to grow in complexity with respect to processor, cache, and node design, creating additional difficulties for code optimization on these systems. The SPOT team has demonstrated a systematic approach for profiling and optimizing large scientific

Proceedings of the Users Group Conference (DOD-UGC'05)
0-7695-2496-6/05 $20.00 © 2005 IEEE

applications in two application domains, showing how, despite the new difficulties, application performance can be optimal on different platforms.

2. Automated Collection of Performance Data

Application performance data are basically of two types: profile data and trace data. *Profile data* provide summary statistics for various metrics and may consist of event counts or timing results, either for the entire execution of a program or for specific routines or program regions. In contrast, *trace data* provide a record of time-stamped events that may include message-passing events and events that identify entrance into and exit from program regions, or more complex events such as cache and memory access events. The DoD High Performance Computing Modernization Program (HPCMP) has defined the following levels of performance data:

- *Level 1* consists of whole program profile data such as wall clock time, operation counts, and counts of cache and memory accesses and misses.
- *Level 2a* consists of profile data broken down to the routine level.
- *Level 2b* consists of profile data broken down to the loop and basic block and even statement levels.
- *Level 3* consists of time-dependent event traces.

One goal of the CE-KY4-001 project was to automate the collection of the various levels of performance data. Such capability releases users from an unreasonable burden, i.e., having to make extensive changes to their codes in order to collect such data. A related and frequently mentioned user requirement is that data collection tools must be easy to use. Accordingly, this project has resulted in the following tools:

- A command-line utility called papiex for collection of Level 1 profile data for unmodified executables.

papiex has been developed as part of the PAPI project[1]. It was originally written for Linux/x86 and Linux/IA-64 systems. It uses library preloading to automatically instrument an unmodified executable. papiex can be used with both shared-memory and message-passing parallel programs. It requires that the system have support for library preloading and shared libraries. Since IBM AIX systems do not support library preloading, a wrapper was written for the native hpmcount utility to emulate papiex on that platform. Similarly, on the Cray X1, which does not support shared libraries, a wrapper was written for the native pat_hwc utility to emulate papiex on that platform. Thus, papiex provides the capability of automated collection of Level 1 profile data on all major DoD platforms. Command-line options are used to specify the particular performance metrics to be collected for a given run.

- Scripts and utilities for using the TAU toolkit[3] to perform automated source code instrumentation for collection of Level 2a profile data and Level 3 trace data.

TAU uses the Program Database Toolkit (PDT) to perform automated source code instrumentation at the routine level for MPI programs. For OpenMP programs, automated instrumentation may be done down to the parallel region level using Opari. `TAU` support for automated loop and basic block level instrumentation using `PDT` is currently under development. Previous use of TAU's source code instrumentation required the user to make a number of changes to the application makefile. For the CE-KY4-001 project, a script called TAUCOMPILER was developed that automates much of this process. The user now needs only to make two small changes to the application makefile:

1. Include the appropriate makefile file stub (e.g., makefile.tau-auto-mpi) which defines the necessary constants and libraries, and
2. Precede the compiler or linker command with the TAUCOMPILER command.

The user can then compile and link in the usual manner to produce a TAU-instrumented version of the code. The particular performance metrics to be collected (e.g., wall clock time, operation counts, counts of cache and memory accesses or misses, TLB misses, etc.) are specified at runtime by setting environment variables. Scripts containing the settings for default sets of metrics have been developed as part of this project.

3. Performance Data Management.

As users begin to collect large quantities of performance data for their applications, they need a data management system that can help organize the data, preserve important metadata, and facilitate easy retrieval and analysis. Towards this end, the PET CE team has collaborated with the HPCMP benchmark team to define a standard schema for the different levels of performance data described above. For this project, TAU PerfDMF was extended to support the HPCMP code profiling schema. The TAU PerfDMF provides support for managing multi-experiment performance data stored in an underlying relational database.

Entities in the schema include, for example, Application, Experiment, and Trial. Since Oracle is used for the Code Profiling Database at ERDC MSRC, Oracle support also was added to TAU PerfDMF, which previously supported only PostgresSQL and MySQL. Metadata can be defined and performance data can be uploaded to PerfDMF using either command-line utilities or the graphical ParaProf parallel profile browser. ParaProf also can be used to retrieve and analyze stored

data. TAU PerfDMF not only supports uploading of TAU profile data files, but also supports uploading of profile data from other tools such as gprof, hpmcount, and mpiP. Third-party analysis tools also may be interfaced to PerfDMF. PerfDMF can be set up for individual use or for use by groups who wish to share data. PerfDMF can thus serve as the focal point for tool interoperability and collaboration on application performance tuning.

4. Multivariate Statistical Analysis of Hardware Counter Data

Once data have been uploaded to a performance database, they can be retrieved and analyzed using a variety of performance analysis tools, including the multivariate statistical analysis tools developed as part of the CE-KY4-002 project. Instrumentation of contemporary applications to collect performance data yields huge multidimensional data sets, the size of which depends on the number of processors involved in the execution, the number of instrumentation points, and the number and type of monitored events. Multivariate statistical techniques can help focus the user's attention on important metrics and show the distribution of those metrics across parallel tasks and code regions. Multivariate data are difficult to analyze and often need to be simplified. Principal Component Analysis (PCA), for example, reduces dimensionality by representing the data as a linear combination of principal components and eliminating those that contribute the least to the variance. k-means clustering is a method for identifying groups of similar data points that can be treated uniformly and, thus, provides data reduction[5].

Tools were developed to carry out statistical analysis of both hardware counter data and communication data for parallel applications. The hardware counter data to be analyzed are assumed to be stored in a relational database. Thus, as part of this project, a MATLAB-SQL connection script was provided to facilitate MATLAB access to an SQL database. Also provided, as part of the project, were step-by-step directions that a user can follow to extract required data from a MySQL database using the MySQL query browser and then upload the extracted data file to MATLAB. The data can be extracted using the script or the necessary steps from either the ERDC MSRC Counters Database or a TAU PerfDMF database that uses the DoD HPCMP Code Profiling schema. Once the hardware counter data for an execution of a parallel application have been uploaded to MATLAB, all the multivariate statistical analysis tools provided by the MATLAB Statistics Toolbox can be applied. However, in order to make it easy for non-experts to use multivariate statistical analysis, the following MATLAB convenience functions were developed as part of this project:

- findPrincipalCounters – finds the principal components and the percentage of variance in the data contributed by each and maps the principal components back to the original counter events.
- findPrincipalCountersAdvanced – finds the principal components and the percentage of variance in the data contributed by each and maps the principal components back to the original counter events. In addition, this function returns a bitmap marking which events map to each principal component.
- findClusters – performs k-means clustering and evaluates the clustering quality.
- plotClusters – plots up to seven clusters that were found by k-means clustering.

The MATLAB multivariate statistical analysis tools were used to analyze whole-program hardware performance counter data collected for the HPCMP benchmark codes COBALT, RF-CTH, OOCORE, and HYCOM. The results of the analysis for HYCOM are summarized below. Full details of the exact steps carried out and the full results are contained in a separate report that was submitted for publication on the Online Knowledge Center (OKC).

Table 1. Principal component analysis of HYCOM *(Marcellus, O4, 256)* hardware counter data

Principal Component	Principal Event	Contributing Events	Explanation
1	PM_CYCLES	PM_INST_CMPL, PM_INST_DISP, PM_FXU_FIN, LD_REF_L1, ST_REF_L1	Total cycles and instructions
2	PM_FPU_ALL	PM_FPU0_FIN, PM_FPU1_FIN, PM_FPU_FDIV, PM_FPU_FIN, PM_FPU_FMA, PM_FPU_STF	Floating-point operations
3	PM_LD_MISS_L1	PM_ST_MISS_L1, INST_FETCH_CYC, L1_WRITE_CYC, PM_DTLB_MISS	Level 1 cache misses
4	PM_INST_FROM_L3	PM_ITLB_MISS, PM_BR_MPRED_CR	
5	PM_INST_FROM_L2		
6	PM_FPU_FSQRT	PM_FPU_DENORM	Floating-point square root
7	PM_DTLB_MISS		Data TLB misses

All hardware counter data for multiple runs of an application with the same configuration can be extracted from the performance database. In this example, hardware counter data for the HYCOM application executed on the IBM POWER4 (Marcellus) with 256 processors, optimization level O4, and the large test case

were extracted. Data collected for the same counter over multiple runs were averaged. After eliminating a counter with zero counts (loads from Level 3.5 cache), the counts for 38 metrics (events) on 256 processors remained. Using Principal Component Analysis, seven principal components were found to account for 87 percent of the variability in the data. These principal components may be characterized as shown in Table 1.

The analysis indicates that it may be possible to characterize the performance of HYCOM using representative counters from the above principal components. Since these representative counters can be allocated using three counter groups on the POWER4, e.g., groups G27, G53, and G56, it would be possible to capture the necessary data using three runs instead of the six runs required to capture all 39 counters. These three groups would also capture the data necessary to calculate derived metrics such as cycles per instruction (CPI), floating point operations per second (FLOPS), and the floating point to memory operation ratio (F:M), which could then be compared across platforms.

Table 2 summarizes the results of applying the same methodology to the COBALT, RF-CTH, and OOCORE applications, for which 39 events were originally monitored. Column 2 shows the groups of events that collect 90% of the variability in the data in a POWER4 system regardless of the number of processors involved in the execution of the program. The last column shows the reduced number of events that could be monitored.

Table 2. Results of applying PCA for reduction of dimensionality of hardware counter data

Application	POWER4 Hardware Groups	Reduced Number of Events
COBALT	G3, G15, G27, G56	9
RF-CTH	G3, G27, G56	5
OOCORE	G3, G15, G20, G27, G53, G56	10

5. Performance Optimization of Scientific Applications

Two strategic DoD scientific application projects, in the Computational Chemistry and Material Sciences (CCM) and Computational Electronics and Nano-electronics (CEN) application domains, were chosen for performance optimization. The PAPI and TAU tools were used to profile the applications, and the performance data were uploading to a PerfDMF Code Profiling Database. The profiling helped identify code hotspots and algorithmic bottlenecks. First-order code-rewrites and (minor) algorithmic changes were then made wherever feasible to allow for enhanced efficiency and increased parallel scalability. Additional suggestions, including those for major code-rewrites for improving application performance, were passed on to the application developers in a detailed developers' guide document. Both applications were built, profiled, and optimized on the 512-node TACC Cray-Dell Linux *Lonestar* system, which consists of a mix of 3.06 and 3.2 GHz Xeon dual-processor nodes.

For the study of a CCM application area, the open-source (GPL licensing) molecular dynamics code *Socorro*, was chosen. *Socorro* is a modular, object-oriented code for performing self-consistent electronic-structure calculations for fundamental understanding of materials such as semiconductors and metals; it predicts how these materials will behave under various conditions. It was developed jointly by researchers from Sandia National Laboratories, Vanderbilt University, and Wake Forest University[6].

Socorro was profiled with PAPI (v3.0 beta2) using TAU for source code instrumentation. The events for which profiling information was obtained were floating-point instructions and total cycles. In the following figures, the relative percentages of the exclusive floating-point operation count for each function are tabulated and averaged over the number of processors on which the data were obtained. In Figure 1, functions are listed in order of decreasing magnitude for 16-processor runs. It can be observed in this figure that the **FFT_3D_SERIAL INDEXED** function executes the largest percentage of floating-point operations at 51.6%, followed by **DIAGONALIZE_I** at 27. The main computational function inside **DIAGONALIZE_I** is **zheev**, which is a LAPACK function computing all the eigenvalues and optionally the eigenvectors of a complex Hermitian matrix. Here, as the problem size remains fixed, each processor does the computations due to **zheev** on the same data, so the increased workload may represent the overhead for the sum of that work. This issue needs to be investigated further. The other PAPI profile information obtained was the total number of cycles used by each function. Figure 2 and Figure 3 show the percentage of total cycles for each function averaged over runs on 16 and 128 processors, respectively. The interesting observation from these figures is the increase in the number of cycles used by MPI calls like **MPI_Allreduce**. The percentage of this reduction function went from 6% in a 16-processor run to about 25% in a 128-processor run. Clearly the trend is likely to increase for runs on larger numbers of processors. In general, this is to be expected, since the synchronization overhead from using collective operations like MPI_Allreduce is likely to increase with additional processors. Simultaneously, the average total cycle count of **DIAGONALIZE_I** increases from 4% of total cycles in 16-processor runs to 12% of total cycles in a 128-processor run. The latter observation is not surprising, since it is validated by the dominance of this function in the floating-point operations count, as shown previously.

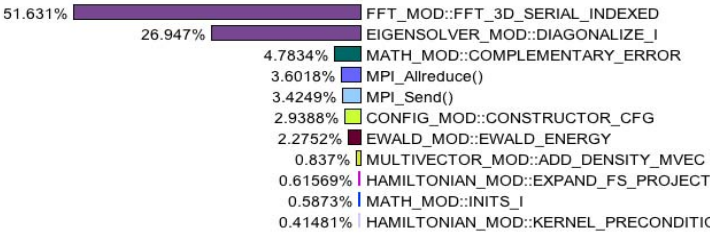

Figure 1. PAPI FP instructions per routine sorted by exclusive count averaged over 16 processors

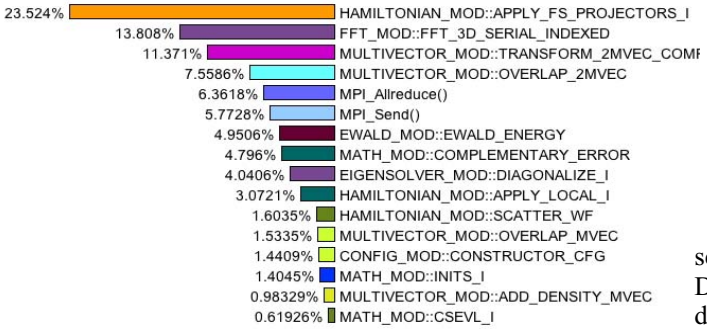

Figure 2. Total cycles per routine sorted by exclusive count for a 16-processor run

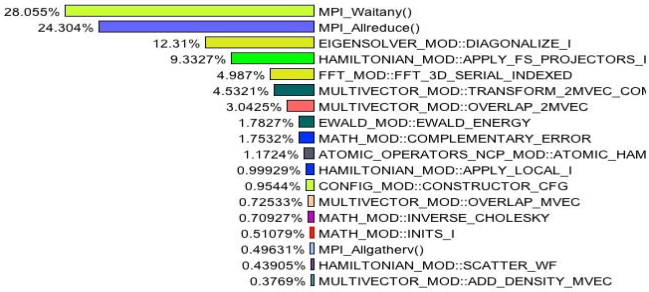

Figure 3. Total cycles per routine sorted by exclusive count for a 128-processor run

On the basis of profiling information, a number of source changes were made and tested but further testing and scalability tests are necessary. For reasons of brevity, a few of the more important developer suggestions are included here.

- Since the **zheev** LAPACK function is a major consumer of floating-point cycles with increases in processor counts, an alternative approach is to look at incorporation of a parallel mathematical library, particularly for larger processor counts, when the gains from parallel processing are greater than the overhead of using it, particularly when each processor has computations of smaller and smaller granularity. The exact processor count where such a crossover from a serial to a parallel library has to occur needs to be determined heuristically, at least at the initial stages. One such option for consideration is using parallel libraries for distributing the **zheev** function operation. Adoption of a parallel library would have a direct impact in reducing the total cycles consumed by **DIAGONALIZE_I** and thus reducing wall-clock time or cycles.
- Other areas of review and possible redesign, based on the above profile information, are the issues of synchronization and cycles consumed that are raised from **MPI_Allreduce** and **MPI_Waitany** calls, particularly for runs on high processor counts. Clearly, review of the parallel design is necessary and any subsequent code changes affect the whole application's performance and not just that of a few procedures; these would involve a nontrivial effort. Towards this end, it is highly recommended that the developers obtain and view trace files.

For the CEN application domain, the open-source, serial, three-dimensional (3-D) Finite Domain Time-Difference (FDTD) code was used. It solves the time-dependent Maxwell's equation in curl form[7], based on the method for the Perfectly Matched Layer (PML) for the appropriate absorbing boundary conditions[8]. Since this is a serial code, the primary profiling effort went into finding the code hotspots and associated optimizations. The porting of this code to a parallel version is currently taking place but could not be completed in time for this project. The serial version was built on the *Lonestar* system with all compiler flags turned on.

Based on cache-miss profiling with PAPI, the code listings in the **calc_H*** and **calc_E*** functions were closely observed for possible optimization. These functions update or compute the magnetic (or electric) field for each cell in each of six coordinate combinations i.e., x-y, y-x, x-z, z-x, y-z, and z-y. Each cell at a given time-step is updated in a given direction for the current iteration with information from the other directions from the previous iteration. In code form, this transforms to computations over a triply nested **kji** loop, where each of i, j, and k are being referred to each Cartesian direction. For 3-D statically stored C-arrays, this path for data access is not the most optimal from a code optimization standpoint, although it may be for FORTRAN arrays. The loop orderings were changed to **ijk** and **ikj**, both of which appeared to be an improvement over the given **kji** data layout. The initial conjecture of the non-optimality of the **kji** data ordering was validated from the cache-miss results that were obtained. Although both of the **ijk** and **ikj** data layout strategies were better than the **kji** layout, the **ijk** layout is the most optimal. The improvements in reducing L1 data cache misses were reflected in wall-clock time (not shown here). The degree and order of improvement are also a function of the number of time steps. For example, for 1024 time steps, the wall-clock time for the original code is 575 seconds, while using the **ijk** strategy takes about 45 seconds. This is more than a

10x improvement in run time. The code developers have observed a similar improvement in the order of the wall-clock time using the proposed source code enhancements on different computer system architectures, but in general the degree of improvement also depends on architectural issues such as cache sizes as well as the capabilities of the compiler and its use with relevant optimization flags. Apart from the significant improvement in the original serial version of the FDTD code obtained from the SPOT optimization, there are a few other suggestions for optimizations that should also be considered by the developers for the 3-D FDTD code, such as strip-mining for cache lengths, although they are not likely to lead to improvements as dramatic as those obtained here. The same loop ordering is recommended for the parallel port of the code, since the data layout issues remain the same.

Acknowledgments

This publication was made possible through support provided by DoD High Performance Computing Modernization Program (HPCMP) Programming Environment and Training (PET) activities through Mississippi State University under the terms of Contract No. N62306-01-D-7110.

References

1. PAPI website, http://icl.cs.utk.edu/papi.

2. TAU website, http://www.cs.uoregon.edu/research/paracomp/tau/.

3. Mohr, B. and F. Wolf, "KOJAK–A Tool Set for Automatic Performance Analysis of Parallel Programs." *Proceedings of the Euro-Par Conference*, 2003, pp. 1301–1304.

4. Huck, K., A. Malony, R. Bell, and A. Morris, "Design and Implementation of a Parallel Performance Data Management Framework." *Proceedings of ICCP05*, June 2005.

5. Ahn, D. and J. Vetter, "Scalable Analysis Techniques for Microprocessor Performance Counter Metrics." *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society Press, 2002, pp. 1–16.

6. Socorro web-page: http://dft.sandia.gov/Socorro/mainpage.html.

7. Ayubi-Moak, J., S. Goodnick, S. Aboud, M. Saraniti, and S. El-Ghazaly, "Coupling Maxwell's Equations to Full Band Particle-Based Simulators." *Journal of Computational Electronics,* 2, Dec 2003, pp.183–190.

8. Berenger, J., "Perfectly Matched Layer for the FDTD Solution of Wave-structure Interaction Problems." *IEEE Transactions on Antennas Propagat.*, 44, 1996, p. 110.