

An approach for fine-grained profiling of mesh-based parallel programs

Amol S Deshmukh
ECECS Department,
University of Cincinnati
Cincinnati, OH 45220 USA
deshmua@ececs.uc.edu

Qingyuan Liu, Karen Tomko
ECECS Department,
University of Cincinnati
Cincinnati, OH 45220 USA
{liuqu, ktomko}@ececs.uc.edu

Abstract – Characterizing the dynamic behavior of parallel programs in terms of their execution profile helps to understand their behavior and suggest optimization strategies to improve the performance. Traditional event tracing techniques write the profiled data to trace files. Using the traditional approach for fine grained profiling not only yields large unwieldy trace files but often also gives skewed results due to the inaccuracies introduced by the profiling. This paper describes an approach to profile mesh-based parallel programs at a very fine level of granularity by measuring performance metrics at the level of each mesh element. The approach described in this paper is novel in that profile data is associated with mesh elements, not processors, so the profile data can be used, for example, to develop adaptive load balancing policies. A tool implementing the idea described in this paper is developed which provides an easy-to-use C API with Fortran 90 wrappers to enable fine grained profiling of mesh-based parallel applications.

1 INTRODUCTION

Profiling of parallel programs is important for two main reasons. Firstly, profiling aids in locating bottlenecks in the program where optimization efforts can be directed to improve the execution speed of the program. Secondly, by analyzing the trace data or using it to drive simulations, it is possible to measure or evaluate a system design [1]. As a result profiling allows characterization of programs which helps gain a better understanding of the programs and possibly predict their performance on different architectures and runtime environments.

Recent profiling efforts have focused on online performance analysis techniques in an attempt to use the performance analysis for runtime optimization of the program [2, 3, 4]. E.g., runtime analysis of performance parameters can be used for adaptive load balancing of mesh-based parallel applications [5, 6, 7]. In [8] the authors describe a tool that uses profiling to perform load balancing of parallel loops in an MPI application. Several algorithms for adaptive load balancing have already been developed [9, 10]. Profiling for online optimization imposes additional constraints on the profiling or performance-monitoring component of the online optimizer. Firstly, profiling should be lightweight so that

it does not add any significant overhead in terms of execution time of the target program. In addition, the profiling must be fast so that the performance data is available to the decision-making component of the load balancer without significant delay. As noted in [4], online profiling is suitable especially for long-running programs where the execution time is of the order of several hours or greater.

It should be noted that while online profiling has the above stated advantages; it is not a substitute for offline profiling. When used to drive simulations based on profiled data, offline profiling allows the user to actually replay the execution profile of the program any number of times, even allowing replaying in ‘slow motion’ [11] and replaying ‘backwards’ in time [12]. This affords the possibility of performing different analyses on trace data obtained from a single profiling run.

The tool implementing the approach described in this paper is designed with two objectives in mind. The primary objective is to collect data for characterizing mesh-based applications at a very fine level of granularity to allow the construction of a synthetic application that will mimic the execution of the target program at the level of individual mesh element. Thus, unlike traditional approaches to profiling which target the tracing of events or determining performance parameters for subroutines, the approach described in this paper gathers profiling data that maps to the structure of the mesh-based application. Towards this goal, the tool provides an easy to use API that can be used to profile a variety of mesh-based parallel programs so that the value of a chosen performance metric can be measured per mesh-element. In addition, the software architecture of the tool allows its use as an online performance analysis system. Initial results of using the tool for profiling of two parallel applications show that the tool operates with moderate to low overhead in terms of execution time and with appropriate choice of performance metric, even performs fast enough to allow its use in an online performance steering system.

1.1 Motivation

The various trade-offs involved in choosing the level of granularity when profiling are discussed in [1]. While coarse-grained profiling gives results that are easily

generalized they convey less information about the system. Generally speaking, fine-grained profiling provides more information but the results are difficult to generalize and incorporate in a simulation that is driven by the trace data. However, our belief is that in case of mesh-based parallel programs fine grained profiling at the level of each mesh element could provide a detailed view of the system which can be easily incorporated in a simulation that is driven by the trace data.

Recent work in generating compact application signatures [13] describes a viable approach to characterizing the behavior of parallel programs by compaction of the traces as they are generated. The approach is impressive as described for characterizing the execution of the application process per node. Using the same approach for the problem of fine-grained profiling of mesh-based parallel programs however would result in creation of several traces (one per mesh element) on each node. With typical mesh sizes, this would mean generating tens of thousands of traces (even with sampling this would reduce only an order of magnitude and still mean several thousands of traces). Moreover, since the traces are generated as the execution of the program progresses, all the traces would have to be kept 'open' simultaneously till the program execution ends. For programs that have very long running times, this is not feasible. Hence this paper describes an adaptation of the approach described in [13] for trace generation and compaction, which uses trace generation in space rather than time. This results in individual traces per sampling time-step. Another concern in using the approach as described in [13] for fine-grained profiling is that the compaction algorithm runs in the same process as the application program, which is not suitable for fine grained profiling since the time taken in trace compaction could alter the execution time significantly. In our approach, a separate process (referred to as the LocalAgent process in this paper) is executed on each node that performs the compaction. Compact traces generated on each node are dispatched over the network to a separate node, that 'stitches' the traces received from different nodes running the parallel program.

1.2 Scope of the work

The proposed approach targets mesh-based parallel programs for which the per-element computations are performed individually as opposed to an approach, which parallelizes the solving of a global matrix for the entire mesh. [14] describes an element-by-element scheme for solving problems in finite-element systems. [15] describes implementations of the element-by-element approach to parallelizing and solving mesh-based systems.

Although the tool described in this paper can be used as a high-resolution probe for an online optimization system, the paper does not attempt to describe such an optimization system. The attempt is to provide a tool that can serve as a probe to supply the necessary performance-feedback information to an online optimization system, or generate trace data for offline analysis.

2 DESCRIPTION OF APPROACH

2.1 System Architecture

Figure 1 illustrates the runtime system architecture. The system consists of a daemon process executing on each node (called LocalAgent) that compacts and communicates the profiled trace information to a central node running another server process (called GlobalAgent). The application program to be profiled is hand-instrumented using an easy to use C API (with Fortran 90 wrappers) developed as a part of this work. The communication of profiling information from the application to the LocalAgent process is done via shared memory data exchange.

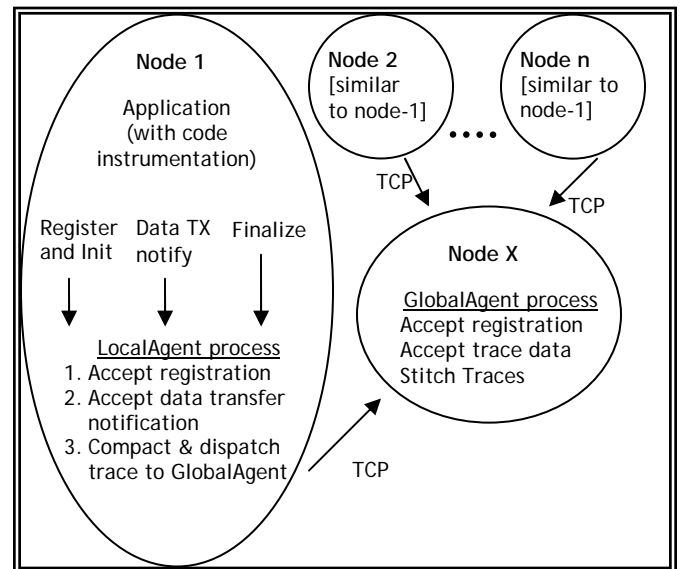


Figure 1: Runtime System

2.2 Code instrumentation in application program source code

The target application program is instrumented with calls to the library developed as an interface for the tool for profiling and communicating the profiled data to the 'LocalAgent' process running on that node. Figure 2 shows a sample of code being instrumented by the library calls. The code snippet is from the Wavetoy application from the Cactus framework [17] that is instrumented

using calls to the API developed as a part of the tool described in this paper.

In the simplest case, five calls to the library are inserted in the source code at appropriate places. They are:

a. LA_register: This call registers the application process with the LocalAgent process. This call enables the LocalAgent process to setup data structures for the application and shares the shared memory and semaphore set keys that are used for subsequent communication of profiled data between the application process and the LocalAgent. (In case of MPI programs this call is inserted in source code after the call to MPI_Init.)

b. LA_startProbe and LA_stopProbe: These calls are used to measure the performance metrics at level of each mesh-element. Thus every measurement involves wrapping the code corresponding to the ‘measured’ element with calls to LA_startProbe and LA_stopProbe. These methods actually measure the performance parameter and store the measured values in the shared memory.

c. LA_dataXfer: This call is used once after every data collection cycle completes to notify the LocalAgent process of the data setup in the shared memory. This function notifies the LocalAgent process of the profiled-data ready in the shared memory.

d. LA_finalize: This function clears the shared memory segment and notifies the LocalAgent process to cleanup any resources allocated for the application program. This call is inserted before the program ends. (In the case of MPI programs, this call is inserted just before call to MPI_Finalize.)

The above five function calls are the most basic calls that every profiled application program must include. Additionally, calls have been provided in the API to handle cases where the mesh elements are traversed in staggered fashion or for cases where loop unrolling is required in the source code [16]. These calls were required for profiling the BenchADM application from the Cactus framework [17] as well as for profiling the MRTD modeling application [18].

2.3 LocalAgent process

The LocalAgent is a multithreaded process which runs two TCP services: one for accepting application process registrations (RegService) and another for accepting data transfer notification (DXService). This runtime architecture allows it to accept registrations from multiple applications programs executing simultaneously. The transfer of profiled-data is done asynchronously via shared memory and only the parameters required to access the shared memory are sent as a part of the data transfer notification. This allows the application program process to proceed quickly after setting up the data in the shared memory and sending the data-ready notification.

```

void WaveToyC_Evolution(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTS
  cell_id_type cellId = {0, 0, 0};
  for (k=kstart; k<kend; k++) {
    for (j=jstart; j<jend; j++) {
      for (i=istart; i<iend; i++) {
        cellId.i=i; cellId.j=j; cellId.k=k;
        LA_start_counter(cellId);
        index = CCTK_GFINDEX3D(cctkGH,i,j,k);
        phi[index] = factor * phi_p[index]
        -phi_p_p[index]+(dt2)
        *((phi_p[CCTK_GFINDEX3D(cctkGH,i+1,j,k)]
        +phi_p[CCTK_GFINDEX3D(cctkGH,i-1,j,k)])*dx2i
        +(phi_p[CCTK_GFINDEX3D(cctkGH,i,j+1,k)]
        +phi_p[CCTK_GFINDEX3D(cctkGH,i,j-1,k)
        ]))*dy2i
        +( phi_p[CCTK_GFINDEX3D(cctkGH,i,j,k+1)]
        +phi_p[CCTK_GFINDEX3D(cctkGH,i,j,k-
        1)])*dz2i);
        LA_stop_counter(cellId);
      } } }
  LA_data_xfer();
  return;
} /* end of WaveToyC_Evolution */

```

Figure 2: Source code instrumentation required in the file WaveToy.c in the Cactus Framework [7] (instrumented code is indicated by lines in bold font).

2.4 GlobalAgent process

The GlobalAgent process executes on a node that does not belong to the parallel computing cluster. In case of MPI programs this can be the front-end node that is usually not a part of the cluster. This is done so that none of the nodes in the parallel computing cluster is unfairly loaded due to the GlobalAgent process. The GlobalAgent process runs a TCP service to allow the LocalAgent processes on every node in the cluster to communicate the compact trace of the profiled data. The trace data from all the nodes in the cluster is gathered by the GlobalAgent and assembled together to give the complete trace at every profiling timestep.

2.5 Choice of performance metric

The tool implementing the approach described in this paper is not limited to using a fixed set of performance functions. In fact any function that can measure a required performance parameter per mesh element can be used. Changing the approach to measure an arbitrarily selected performance metric involves registering custom probe functions with the tool by calling API functions: LA_setProbeStartFunc and LA_setProbeStopFunc both of which take the pointer to the corresponding custom probe function as an argument. It is even possible to switch between different probe functions that are available at runtime.

Currently the probe functions available as a part of the tool API include probe functions to measure CPU cycles, CPU instruction count and execution time per mesh element. The CPU cycle and instruction count measurement functions use the PAPI library [19] for the actual measurement. The timer is a nanosecond precision timer that uses a system call to get highly precise timing information.

2.6 Trace Compaction

Trace compaction is necessary prior to dispatching the data to the GlobalAgent process on the node external to the parallel computing cluster to avoid the overhead of transmitting large data over the network. Trace compaction is performed in space rather than time for reasons mentioned in section 1.1. Since this compaction is done on every node by the LocalAgent process, it has to be done in a fast and lightweight manner. Our current implementation adapts the polyline fitting trace compaction strategy used in [13] and is based on least squares linear fitting [20] algorithm. The case for using polyline fitting as an efficient and flexible method of curve fitting has been made in [13]. The main advantage of polyline fitting algorithm is that it allows compaction of the traces as they are being generated, i.e. the whole trace does not have to be generated before it is compacted. The coefficient of variation (CV) of the error due to trace compaction is taken as the compaction error metric and is a parameter to the compaction algorithm.

One problem with traces generated in space is that the compaction ratio may not be as good as compaction for traces generated in time since the computation levels in successive points in the trace depends on the sequence in which the mesh elements are traversed. Since the mesh traversal is specific to the implementation of the target program the profiling tool has no control over deciding the traversal order. In addition, as is evident from figures 3a, 3b & 3c, the choice of performance metric can also greatly influence the degree of trace compaction.

2.7 Trace stitching

Trace stitching involves assembling the trace data collected over the different nodes in the parallel computing cluster that belong to the same profiling cycle into a single complete trace for the application at that time step. To enable trace stitching at the global agent, the traces are identified by three identifiers, viz. ‘Node rank’, ‘Application Id’ and the ‘Profiling cycle tag’. Thus traces that are stitched together form a Trace-Set with all traces in the Trace-Set having the same ‘application identifier’ and the same ‘profiling cycle tag’.

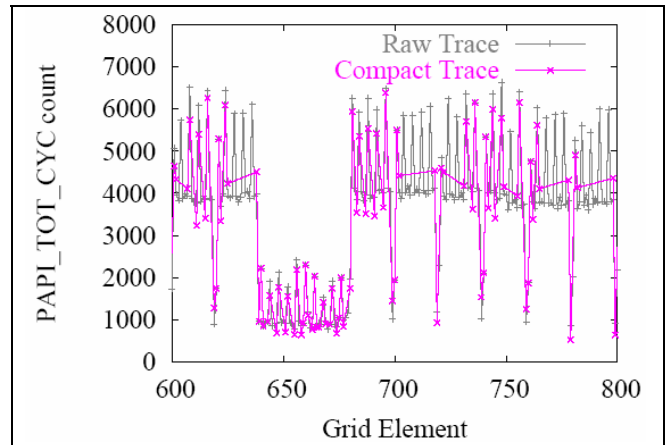


Figure 3a

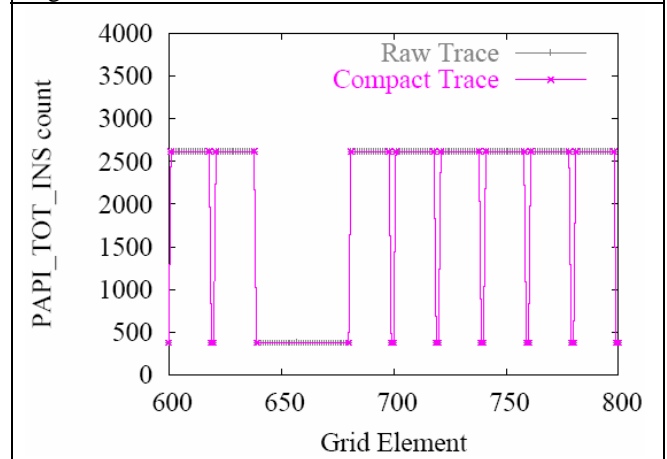


Figure 3b

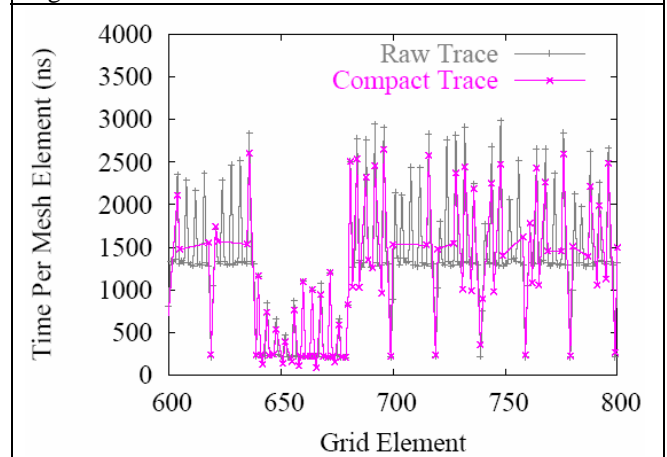


Figure 3c

Figures 3a, 3b & 3c: Sample trace snippets for the Cactus BenchADM with different performance metrics (CPU Cycles, CPU Instructions, Wall clock time) for the same profiling iteration over the same set of mesh elements (measured over different executions).

3 EXPERIMENTAL EVALUATION

3.1 Benchmark Applications

For the experimental evaluation, we chose the MRTD (Multiresolution Time Domain) modeling application and two scientific codes from the Cactus Code framework [21]. The MRTD modeling application performs the time domain modeling of large-scale co-site interference problems arising in wireless communications. Cactus is a popular and actively maintained framework for numerical simulations. The applications chosen from the Cactus framework were WaveToy and BenchADM. All the codes perform element-by-element calculations and are thus suitable to be profiled using the approach described in this paper.

Mesh sizes in all the applications were chosen to be of the order of 104 to 105 elements per processor. The experiments were conducted on a Beowulf cluster consisting of 8 nodes with dual AMD Athlon MP-1800 processors running the MPICH-1.2.5.2 MPI environment. Sample trace snippets for the BenchADM application from the Cactus framework with different performance metrics are shown in Figures 3a, 3b & 3c.

Cactus Wavetoy: Profiling of WaveToy was possible with minimal code instrumentation in the source code. Specifically the files changed were WaveToy.c and ProcessEnvironment.c. The file WaveToy.c was instrumented to contain code to measure computation values in each element of the mesh.

Cactus BenchADM: The code for Cactus BenchADM is more complex in the way the iteration over the set of mesh elements is performed as compared to the WaveToy application. As compared to the other applications, Cactus BenchADM has more computations per mesh element.

Multiresolution Time Domain Application: In the case of this application, only one source file was changed to include the calls to the tool API. This application is peculiar since it has multiple loops for performing different calculations in the source code that traverse the same set of cells in single time step iteration. This causes the fine-grained profiling to incur a greater overhead since the value of the performance metric for each mesh element has to be ‘measured’ multiple times in a single profiling iteration.

3.2 Analysis of results

Effect of choice of performance metric

Since fine-grained profiling involves measuring the value of the performance metric for every mesh element,

several thousands of measurements occur in one iteration. As a result, if the profiling is to be used in an online performance steering system, care should be taken to choose a performance metric whose measurement causes low overhead in terms of execution time. Results demonstrate that using ‘execution time per mesh-element’ as the performance metric results in very low increase in the overall execution time of the program, whereas using the PAPI calls to measure the CPU cycles and CPU instructions results in significant increase in the overall program execution time (Figure 4).

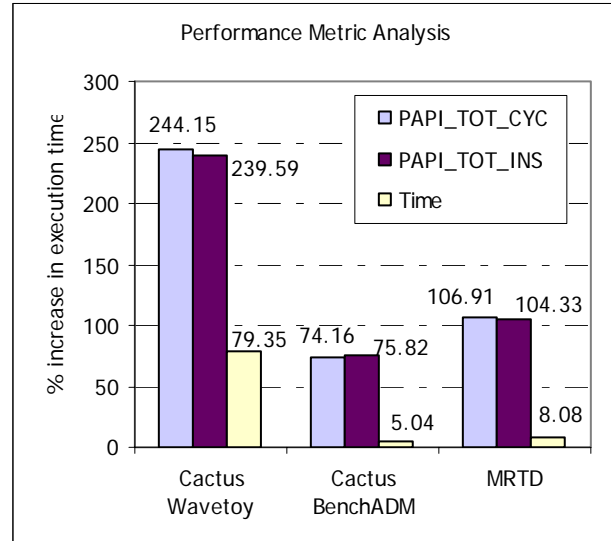


Figure 4: Increase in execution time with different performance metrics (profiling period=15, CV of error due to trace compaction=0.1)

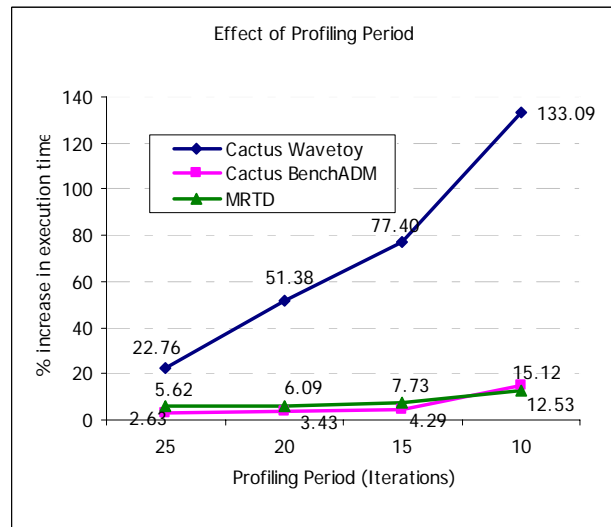


Figure 5: Increase in execution time over different values of profiling period (performance metric='time', CV of error due to trace compaction=0.1)

Effect of profiling frequency

For cases involving very few computations per mesh element, the trace compaction algorithm runs slower than the performance data collection, which causes the application program to stall until the data from the previously sampled iteration is read and compacted by the LocalAgent process. Results demonstrate that increasing profiling frequency significantly increases the overall execution time (Figure 5).

Effect of trace compaction error

The polyline fitting algorithm described in [20] takes the coefficient of variation in error due to trace compaction as a parameter. A smaller value implies a larger size of the compacted trace and hence the effect is greater network overhead in dispatching the trace data to the GlobalAgent process. This effect is seen in figure 6.

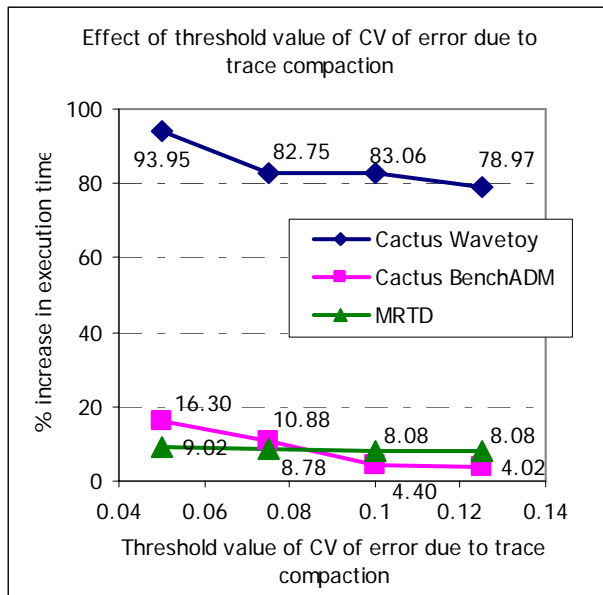


Figure 6: Increase in execution time over different values of CV of error due to trace compaction. (profiling period=15, performance metric='time')

4 FURTHER WORK

Two ways to further this work are being explored currently. One is to use the tool in a global adaptive mesh repartitioning load-balancer that utilizes the trace information from all the nodes in the same profiling iteration to determine the computational workload distribution in the mesh and accordingly suggest a repartitioning in case great imbalance occurs.

Another approach under consideration is to use the tool in generating compact trace data that can serve as an 'application signature' that maps to the structure of the mesh rather than to the workload of individual nodes in

the parallel computing cluster. This representation of the program execution profile would aid in developing a synthetic application that mimics the profiled application program performance using the trace data as input.

5 RELATED WORK

The main difference between the approach suggested in this paper and the other currently available profiling tools for high performance computing applications [11, 22, 23] is that this tool profiles mesh-based parallel applications by calculating the workload at the level of each mesh element. As a result, this approach conveys profiling information to a very fine level of granularity, but is not generic enough to profile all high performance applications.

Additionally, the approach proposed in this paper does not attempt to determine the communication related performance parameters. Photon MPI is a modified MPI library that helps to determine the performance overhead incurred due to communication, at a fine level of granularity yet being lightweight [24]. The approach proposed in this paper complements the Photon MPI system in providing a complete view of the parallel application performance at a very fine level of granularity.

While the software architecture of the tool draws inspiration from the ActiveHarmony architecture, it should be emphasized that both systems have significant differences. ActiveHarmony is not a fine-grained profiling tool (although it is a complete online performance steering system) and it does not use separate process running on each node to communicate with the Harmony server. Indeed in the case of ActiveHarmony having such a process would add unnecessary overhead since there is no processing of the data involved at individual nodes. In the problem domain addressed by this paper, trace data needs to be compacted and dispatched over the network to the node running the GlobalAgent. This is made more efficient by delegating the trace compaction to a separate process. Finally, the trace compaction algorithm is a direct adaptation of the algorithm used in [13] with the difference that our traces are generated in space rather than time.

6 CONCLUSION

We have developed a tool for profiling mesh-based parallel programs by measuring values of performance parameters at the level of individual mesh elements. The current implementation of the tool offers a very flexible choice of performance metric and has been demonstrated on three benchmark applications measuring three different

performance metrics. As demonstrated by the results, in case of applications involving significant computations the tool operates with overheads of less than 8% to measure execution time per mesh element at a sample frequency of 1/15 thus making fine-grained profiling feasible.

ACKNOWLEDGEMENTS

The authors would like to express their thanks to the LINC lab at the University of Cincinnati for permission to use the Beowulf cluster and also to Michal Kouril who continues to be a tremendous help in resolving cluster-related issues while conducting the experiments on the LINC Beowulf cluster. The research work of Qingyuan Liu and Karen Tomko is supported in part by National Science Foundation (Grant No. ACI-0305532).

REFERENCES

- [1] A. Downey and D. Feitelson, "The Elusive Goal of Workload Characterization," Technical report, Hebrew University, Jerusalem, 1999.
- [2] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," Proc. International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, 2003, pp. 265-275.
- [3] C. Tapus, I-H. Chung, and J. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," Proc. SC'02, 2002, pp. 1-11.
- [4] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool," IEEE Computer, vol. 28(11) pp. 37-46, 1995.
- [5] K. Devine, B. Hendrickson, E. Boman, M. St.John and C. Vaughan, "Zoltan: A dynamic load-balancing library for parallel applications; user's guide," Technical Report SAND99-1377, Sandia National Laboratories, 1999.
- [6] A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten and C. Walshaw, "Dynamic Multi-Partitioning For Parallel Finite Element Applications," Parallel Computing: Fundamentals & Applications, Linear systems and associated problems, vol. 27(7), 2001, pp. 869-881.
- [7] G. Karypis and V. Kumar, "A Coarse-Grain Parallel Formulation of Multilevel k-Way Partitioning Algorithm," Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.
- [8] R. Carino and Ioana Banicescu, "A Load Balancing Tool for Distributed Parallel Loops," International Workshop on Challenges of Large Applications in Distributed Environments, June 2003, pp. 39.
- [9] K. Devine, J. Flaherty, S. Wheat and A. Maccabe, "A massively parallel adaptive finite element method with dynamic load balancing," Proc. ACM/IEEE Conference on Supercomputing, pp. 2-11, 1993.
- [10] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Proc. ACM/IEEE Conference on Supercomputing, pp.28-es, 1995.
- [11] Vampir, Pallas.
<http://www.pallas.com/e/products/vampir/index.htm>
- [12] M. Heath and J. Finger, "ParaGraph: A Performance Visualization Tool for MPI," <http://www.csar.uiuc.edu/software/paragraph/userguide.pdf>, 2003.
- [13] C-D. Lu and D. Reed, "Compact Application Signatures for Parallel and Distributed Scientific Codes," Proc. SC'02, p. 1-10, 2002.
- [14] G. Carey, F. Barragy, R. McLay and M. Sharma, "Element-by-element vector and parallel computations," Communications in Applied Numerical Methods, 1988, pp. 299-307.
- [15] G. Carey, A. Boze, B. Davis, C. Harle and R. McLay, "Parallel Computation of Viscous Flows," Proc. of HPC '97, 1997.
- [16] A. Deshmukh, "An Approach For Profiling of Parallel Applications," Masters Thesis, Univ. of Cincinnati, 2004.
- [17] Cactus, <http://www.cactuscode.org>.
- [18] C.D. Sarris, K. Tomko, P. Czarnul, S.-H. Hung, R.L. Robertson, D. Chun, E.S. Davidson and L. P. B. Katehi, "Multiresolution Time Domain Modeling for Large Scale Wireless Communication Problems", Proc. of 2001 IEEE AP-S International Symposium on Antennas and Propagation, vol. 3, 2001, pp. 557-560.
- [19] K. London, S. Moore, P. Mucci, K. Seymour and R. Luczak, "The PAPI Cross-Platform Interface to Hardware Performance Counters," Proc. Department of Defense Users' Group Conference, 2001.
- [20] P. Worley "Modeling Histogram Data with Piecewise Polynomials," Technical Report ORNL/TM-11637, Oak Ridge National Laboratory, 1990.
- [21] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Masso and Thomas Radke, Edward Seidel and John Shalf "The Cactus Framework and Toolkit: Design and Applications," Vector and Parallel Processing - VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science, 2002.
- [22] J. Yan, S. Sarukkai, and P. Mehra, "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit," Software Practice & Experience vol. 25(4) , 1995, pp. 429-461.
- [23] J. Vetter and D. Reed, "Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids," The International Journal of High Performance Computing Applications, vol. 14(4), 2000, pp. 357-366.
- [24] J. Vetter, "Dynamic Statistical Profiling of Communication Activity in Distributed Applications," Proc. ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, 2002, pp. 240-250.