Phase-Based Parallel Performance Profiling

Allen D. Malony[a], Sameer S. Shende[a], Alan Morris[a]

[a] Department of Computer and Information Science University of Oregon, Eugene, OR, USA

Parallel scientific applications are designed based on structural, logical, and numerical models of computation and correctness. When studying the performance of these applications, especially on large-scale parallel systems, there is a strong preference among developers to view performance information with respect to their "mental model" of the application, formed from the model semantics used in the program. If the developer can relate performance data measured during execution to what they know about the application, more effective program optimization may be achieved. This paper considers the concept of "phases" and its support in parallel performance measurement and analysis as a means to bridge the gap between high-level application semantics and low-level performance data. In particular, this problem is studied in the context of parallel performance profiling. The implementation of phase-based parallel profiling in the TAU parallel performance system is described and demonstrated for the NAS parallel benchmarks and MFIX application.

1. Introduction

When studying the performance of scientific applications, especially on large-scale parallel systems, there is a strong preference among developers to view performance information with respect to their "mental model" of the application, formed from the structural, logical, and numerical models used in the program. If the developer can relate performance data measured during execution to what they know about the application, more effective program optimization might be achieved. The general problem is one of *performance mapping* [13] – associating measured performance data to higher level, semantic representations of model significance to the user. The difficulty of making the association depends on what information is accessible from instrumentation and available in the measured performance data that can be used in performance mapping. There is the additional difficulty of how the performance information is to be presented to the users in model-based views.

What can be done to support the association between model and data in performance mapping? The concept of "phases" is common in scientific applications, both in terms of how developers think about the structural, logical, and numerical aspects of a computation, and how performance can be interpreted [16]. It is therefore worthwhile to consider whether support for phases in performance measurement can help to bridge the semantic gap in parallel performance mapping. Performance tracing has long demonstrated the benefits of phase-based analysis and interpretation [5]. However, performance tracing carries a heavy cost in trace size and complexity of post-mortem processing. The questions we pose in this paper concern support for phases in parallel performance profiling.

Section §2 describes the problem of phase-based profiling in more detail and outlines our objectives. The design and development approach we used to implement phase-based profiling in the TAU performance system are described in Section §3. The API for creating and controlling phases during profile measurements is also presented. We tested phase profiling on several of the NAS parallel benchmarks [1] and the MFIX (Multiphase Flow with Interphase eXchanges) [14,15] application. Section §4 shows how these programs were instrumented for phases and contrasts the phase profile results with those of standard profiling. A comparison of our work to other profiling efforts is given in Section §5. Conclusions and future work are given in Section §6.

2. Problem Description

An empirical parallel performance methodology collects performance data for *events* that occur during program execution. Thus, the events of interest determine what performance information is present in profiles or traces. Events have associated *semantics* and *context*. Semantics defines what the event represents (e.g., the entry to a routine), and context captures properties of the state in which the event occurred (e.g., the routine's calling parent). Most profiling tools can generate a *flat profile* showing how the computation performance is distributed over the program's static structure (i.e., its code points). The events usually represent routine entry and exit points, but can identify any code location. The "context" in flat profiling is the whole program. Unfortunately, a flat profile cannot differentiate performance with respect to computation dynamics.

Callgraph profiles are commonly used to separate performance with respect to parent-child event relationships. Computation dynamics is captured in the profile by identifying these relationships at the time of event occurrence. This "context" is used to essentially extend the event semantics and map performance to parents and children. Unfortunately, because callgraph profiling only captures local parent/child context, it will be difficult to map performance of deeply nested events to outermost, top-level parents. If a profiling tool supports full *callpath profiling*, the performance mapping improves because more ancestors are exposed on the calling path, but this still relies on "calling relationships" (i.e., calling contexts) to extend event semantics in the performance profile.

Given a set of instrumented events and a profile measurement infrastructure, we would like to be able to interrogate the context with each event occurrence as a way to decide how to map performance information for that event. Defining context based solely on event calling relationships is problematic because it requires events to be created for purposes of identifying context, whether or not they are of performance interest. Context relates to the notion of *state*, and its definition is richer than something that just occurs during execution (i.e., an event). Context can be logical as well as based on runtime properties. In this paper, we propose to define context in relation to the concept of a *phase*. Our goal is to then support phases as part of the instrumentation and measurement environment the developer uses to understand the performance of their application. The support should allow for phases to be created and events to be associated with phases, thereby distinguishing performance by phase identity. It should be possible to capture both *static* and *dynamic phases* in the performance measurement, and our implementation does so.

The primary contribution of our work is the realization of phase-based measurement in parallel performance profiling. Because the processing takes place online, phase support is harder to implement in profiling. Indeed, we believe our results are the first to report phase-based profiling in an actual, portable parallel performance system, TAU [9]. The following describes this implementation. We first describe the instrumentation API and how the measurement system operates to map event performance to associated phases. Examples of phase profiling with the NAS parallel benchmarks and the MFIX application are then given.

3. Implementation Approach

To understand the phase profiling approach, it is important to first discuss how flat and callpath profiling are implemented.

3.1. Flat Profiles

A flat profile shows performance data associated with different parts on an application code, typically the program routines. Profile instrumentation is placed at code points to mark the entry and

exit of execution segments and profile measurements (*inclusive* and *exclusive*) are made when these code execution events are encountered. Profile measurements are kept in a data structure TAU calls a *profile object* associated with an entry/exit event pair. A profile object can be created statically, in the sense that it will be reused, or dynamically, where a new instance will be created. A *name* is assigned to the profile object. The profile object names represent the "flat" context (i.e., it identifies only the code segement profiled). To deal with nesting, however, a flat profiling system must maintain some sort of an *event stack* (or *callstack*). However, flat profiles do not expose inter-event relationships, as no details of the program's calling order is preserved in the profiles.

3.2. Callpath Profiles

In contrast to flat profiles, callpath profiles show calling or nesting relationships between events (code segments), typically thought of as calling paths in a program's routine callgraph. In callpath profiling, events are instrumented in the same way as for flat profiling, but profile objects are created and maintained based on the context in which the events occurs (i.e., their calling or nesting context). Profile measurements are made with respect to the current callpath, which can be determined by querying the callstack on exit events. TAU encodes the callpath in the profile object's name. The *length* of a callpath is equal to the number of nested events represented in its name.

In TAU, a callpath is constructed by traversing the callstack, which exists as linked profile objects. Upon an event entry, a callpath is constructed looked up in a callpath map. If the callpath has executed before, the associated profile object is found and linked into the callstack. Otherwise, TAU creates a new profile object, initializes it with the callpath name, links it into the callstack, and adds the name to a list of events. To make the map lookup efficient, we construct its key as an array of $n + 1$ elements, where n is the length of the callpath. The top element in the array is an integer that represents the length. The next n elements are the profile object identifiers. TAU's comparison operator compares the length and the n elements successively and returns a boolean value at the first mismatch. Users can control the length of the callpath profiling to generate more a detailed or refined ancestoral view.

3.3. Phase-Based Profiles

Callpath profiling associates context with callstack state as determined by the instrumented events. However, a phase is an execution abstraction that cannot be necessarily identified by a callpath or determined by specific instrumented events. It generally represents logical and runtime aspects of the computation, which are different from code points or code event relationships. The intent of phase profiling is to attribute measured performance to the phases specified. That is, for all phases in the execution, show the performance for that phase. There are two issues to address: 1) how to inform the measurement system about a phase, and 2) how to collect the performance data.

TAU implements a phase profiling API that gives the user control to create phases and effect their transitions. The following shows the interface for C/C++ programs (TAU also supports phase profiling for Fortran):

```
TAU_PHASE_CREATE_STATIC(var, name, type, group)
TAU_PHASE_CREATE_DYNAMIC(var, name, type, group)
TAU_GLOBAL_PHASE(var, name, type, group)
TAU_GLOBAL_PHASE_EXTERNAL(var)

TAU_PHASE_START(var)
TAU_PHASE_STOP(var)
TAU_GLOBAL_PHASE_START(var)
TAU_GLOBAL_PHASE_STOP(var)
```

A phase object is constructed with a unique name when a phase is created. A phase object is similar in function to a profile object for an event. Profiling for a particular phase is started and stopped using calls that take the phase object handle. Like events (i.e., profile objects), a phase can be created as static, where the name registration and phase object construction takes place exactly once, or dynamic, where it is created and instantiated each time. Phases may be nested, in which case the "active" phase object follows scoping rules and is identified with the closest parent phase. Phases may not overlap, as exclusive time for an overlapping phase is not defined, just as it is not defined for overlapping events (profile objects). Each thread of execution in a parallel application has a default phase corresponding to the top level event, usually the main entry point of the program. This top level phase (like all phases) contains the performance of other routines and phases that it directly invokes but excludes routines called by child phases.

A phase-based profile shows the performance profile of the execution for each phase of an application, as determined by the instrumented events encountered. The user uses the phase profiling API to create, start, and stop phases. The implementation of phase-based profiling in TAU effectively will show the user a callpath performance profile of depth 2 for each uniquely identified phase. When TAU is configured with support for tracking phase profiles, the phases are activated and TAU must keep track of mappings between the instrumented events and the phase that they belong to. By default, phases are mapped by TAU to normal profile events when phase-based profiling is disabled, so as to incur no additional overhead.

The real innovation comes in the how performance data is collected for phases. To implement phase-based profiling in TAU, we leveraged our work in mapping performance data [13] and used callpath profiling (as described above) as the underlying mechanism. Here, a caller-callee relationship is used to represent phase interactions. At a phase or event entry point, we traverse the callstack until a phase is encountered. Since the top level event is treated as the default application phase, each event invocation occurs within some phase. To store the performance data for a given event invocation (in a profile object), we need to determine if the current $(event, phase)$ tuple has executed before. To do this, we construct a key array that includes the identities of the current (event) profile object and the parent phase. This key is used in a lookup operation on a global map of all phase, event relationships. If the key is not found, a new profile object is created with the name that represents the parent phase and the currently executing event or phase. In this object, we store performance data relevant to the phase. If we find the key, we access the (already existing) profile object and update its performance metrics. As with callpath profiling, a reference to this object is stored to avoid a second lookup at the event exit.

4. Application

To illustrate the use of phase-based profiling, we experimented with the NAS parallel benchmarks and other parallel applications. Our goal was to show how phase profiling can provide more refined profile results specific to phase localities than standard flat or callpath profiling. Our choice of phases and phase instrumentation boundaries was informed only by what we could learn from the application documentation and code analysis. Developers could better apply their understanding of the computational models used in the code to define phases.

4.1. NAS Parallel Benchmarks

The NAS parallel benchmark applications [1] presented convenient testcases for phase profiling. We instrumented BT, CG, LU, and SP with phases. We present only our results from BT phase profiling. The BT benchmark emulates a CFD application that solve systems of equations resulting from

an approximately factored implicit finite-difference discretization of the Navier-Stokes equations. It solves three sets of uncoupled systems of equations: first in the X, then in the Y, and finally in the Z direction. The system is block tridiagonal (hence the name) with 5x5 blocks. A multi-partition approach is used to solve the systems, since it provides good load balance and uses coarse-grained communication. BT requires a square number of processors to be used.

A natural phase analysis of the BT code would highlight performance for each solution direction. These directions are identified in the code by the three main functions `x_solve`, `y_solve`, and `z_solve`. We used static phases to see how performance varied with each direction by encapsulating the calls to these functions with phase instrumentation. Figure 1 shows the NAS Parallel Benchmark program BT run on 9 processors. The parallel system used in this case was a 16-processor SGI Altix. The top window of the figure shows a flat profile. The middle window shows the exclusive execution profile of the top-most phase, `MPBT`. It consists of the three phases (each shown separately in the bottom windows) as well as any functions called outside of any other phase, primarily `MPI_Waitall`. The `MPI_Wait` call is highlighted in all of the bottom windows enabling this routine's performance to be compared across each of the phases. We can see that `MPI_Wait` consumes different portions of the runtime between nodes in one phase, and these times are distributed differently across nodes in another phase. The portions and distributions cannot be distinguished in the

Figure 1. BT Phase Profile showing phases for x, y, and z solution directions.

flat profile.

4.2. MFIX

From the experience with the NAS benchmarks, one may conclude that callpath profiling can generate that same information as might be obtained from phase profiling. This is true *only* if the *full* callpath profile is obtained. For programs with a relatively small number of nodes in the callgraph, as is the case for BT above, a full callpath profile may be preferred. However, phase profiling can be more intuitive and closely matched to complex applications based on computational models with natural phase design and behavior, and overall less expensive than a full callpath profile. The Multiphase Flow with Interphase exchanges (MFIX [14,15]) application is such a case.

MFIX is being developed at the National Energy Transfer Laboratory (NETL) and is used to study hydrodynamics, heat transfer, and chemical reactions in fluid-solid systems. It is characteristic of large-scale iterative simulations where a major outside loop is performed as the simulation advances in time. We initially ran MFIX on a simulation testcase modeling the Ozone decomposition in a bubbling fluidized bed [4]. The flat profile for this simulation showed that over 28% of the total wallclock time was spent in the `MPI_Waitall` routine. This accounted for 70 minutes of wallclock time spent waiting for interprocess communication requests. There were 93 iterations performed by the outer simulation loop in MFIX. We were interested in knowing if all iterations took the same amount of time and how that time was distributed among the instrumented events encountered within each iteration, especially how `MPI_Waitall` varied from iteration to iteration.

The MFIX code isolated iterate computation in the Fortran subroutine `ITERATE`, making it convenient for use to instrument iterations with dynamic phases. For experimentation purposes, we ran MFIX on four processors. We then analyzed the profiles to partition the total time spent in the `MPI_Waitall` routine based on the application phase. Figure 2 shows the inclusive time contributed by each iteration. Clearly, the times spent in each iteration and in `MPI_Waitall` are not uniform. Seeing the performance profiles with respect to phases in this way provides valuable information in identifying the causes for poor performance as it relates to application specific parameters. `MPI_Waitall` acts an indicator of poor performance which can be explored in more detail by delving into the rest of the profile for the suspect phase. Figure 2 also shows the inclusive time for `SOLVE_LIN_EQ` which displays the same performance shape profile.

As this example demonstrates, phase profiling provides performance information that has previously been available only in tracing. Even if there were no difficulties generating large trace files, standard tracing tools provided no easy means to relate the performance of a group of invocations of a particular routine (such as `MPI_Waitall`, which executes over a million times on each processor in MFIX) to the application phases by simply looking at a global timeline display. The phase profiles quickly highlight the variability in performance of the `ITERATE` subroutine, which led the developers to a better understanding of the convergence criterion used for solving the set of linear equations iteratively at each time step of the computation. The performance of several routines executed within a phase are affected by convergence rate within a phase (e.g., `MPI_Waitall` and `SOLVE_LIN_EQ`). Phase-based profiling helped to track temporal differences in performance with respect to the computational abstraction the scientists understand, enabling them to be more productive in their performance problem solving.

5. Related Work

Early interest in interpreting parallel performance in relation to phases of execution grew out of two research directions. First, there was the recognition that visual patterns of performance data can

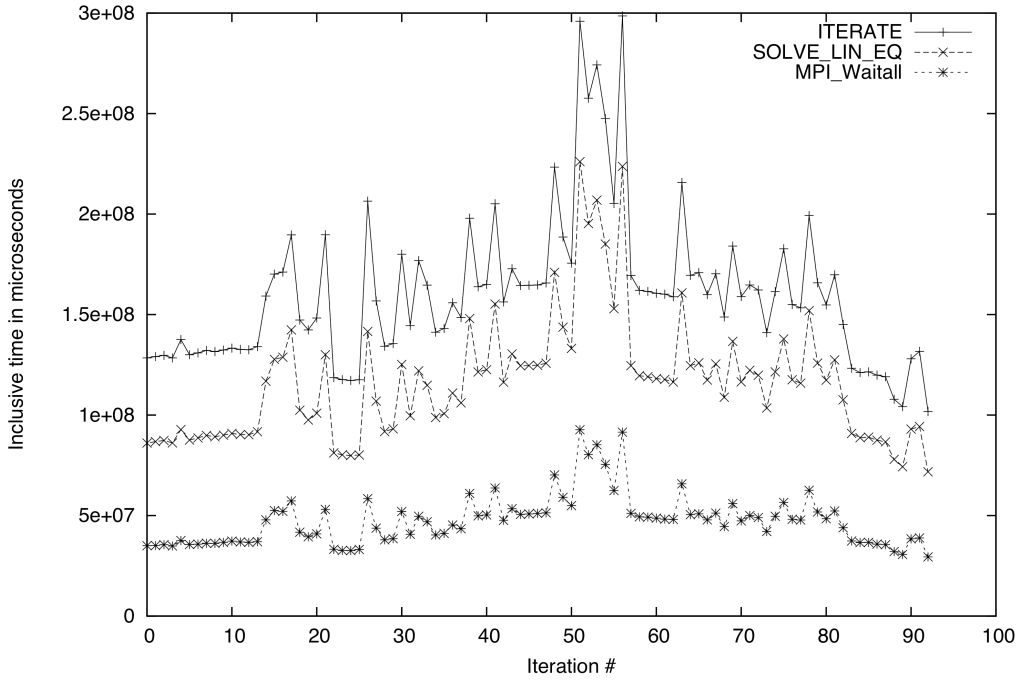

Figure 2. `MPI_Waitall` and `SOLVE_LIN_EQ` performance in each iteration

be related to application behavior [5,6,8]. The Paragraph [5] tool was the first to demonstrate how phase patterns can be seen in the visualization of performance traces. Other tools sought to represent low-level trace data in more abtract views [8] tied to application semantics. These techniques carry on in modern day trace visualization tools such as Vampir [3].

The second direction was in parallel performance modeling of scientific application codes [16]. Natural models represent the computational components and the parameters affecting their performance, as well as the sequence (phases) of their use. Phase-based performance views made it possible to relate both implementation and performance constraints between phases, versus ones that just evaluated component kernels in isolation. The approach also helped to identify performance artifacts of phase transitions, such as extra data copying or communication [2].

Irvin and Miller [7] introduced the concept of mapping low-level performance data back to source-level constructs for high-level parallel languages. They created the *Noun-Verb* (NV) model to describe the mapping from one level of abstraction to another. A *noun* is any program entity and a *verb* represents an action performed on a noun. Sentences, composed of nouns and verbs, at one level of abstraction, map to sentences at higher levels of abstraction. The *ParaMap* methodology defined three different types of mappings: static, dynamic, and one based on the *set of active sentences* (SAS). Shende's [13] *Semantic Entities, Associations, and Attributes* (SEAA) model builds upon the NV/SAS mapping abstractions to support user-level mapping, more accurate performance capture, and asynchronous operation. Our phase-based profiling research is founded in these mapping principles.

6. Conclusion

Linking performance data to execution semantics requires mapping support in the measurement system. We have developed techniques to map performance profiles to phases of a computation.

Phases are defined by the application developer, using an API for phase creation and control. Phases can be used to represent structural, logical, and execution time aspects of the program. Phases can also be hierarchically related. Parallel profiles produced at the end of an execution encode phase relationships. For any particular phase, its profile data reflects the parallel performance when that phase was active in the computation.

We have implemented phase-based profiling in the TAU parallel performance system and this paper demonstrates the implementation for the NAS benchmarks and the MFIX application. In addition, we have applied phase profiling to several large-scale applications including a high-fidelity simulation of turbulent combustion with detailed chemistry (the S3D code [11]), a simulation of astrophysical thermonuclear flashes (the FLASH code [10]), and the Uintah computational framework for the study of computational fluid dynamics [12]. Our future objectives are to improve the phase profile data analysis and provide to better visualization of phase performance.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, "The NAS Parallel Benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[2] J. Brehm, P. Worley, and M. Madhukar, "Performance Modeling for SPMD Message-Passing Programs," *Concurrency: Practice and Experience*, **10**(5):333–357, April 1998.

[3] H. Brunst, M. Winkler, W. Nagel, H.-C. Hoppe, "Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach," In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, K. Tan, (eds.), *International Conference on Computational Science*, Part II, LNCS 2074, Springer, pp. 751–760, 2001.

[4] C. Fryer and O.E. Potter, "Experimental Investigation of models for fluidized bed catalytic reactors," AIChE J., **22**:38–47, 1976.

[5] M. Heath and J. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, **8**(5):29–39, September 1991.

[6] M. Heath, A. Malony, and D. Rover, "The Visual Display of Parallel Performance Data," *IEEE Computer*, pp. 21–28, November 1995.

[7] R. Irvin and B. Miller, "Mapping Performance Data for High-Level and Data Views of Parallel Program Performance," *International Conference on Supercomputing (ICS)*, May 1996.

[8] T. LeBlanc, J. Mellor-Crummey, and R. Fowler, "Analyzing Parallel Program Executions using Multiple Views," *Journal Parallel and Distributed Computing*, **9**(2):203–217, June 1990.

[9] A. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon, "Advances in the TAU Performance System," in *Performance Analysis and Grid Computing*, (Eds. V. Getov, et. al.), Kluwer, Norwell, MA, pp. 129-144, 2003.

[10] R. Rosner et al., "Flash Code: Studying Astrophysical Thermonuclear Flashes," *Computing in Science and Engineering*, **2**:33, 2000.

[11] http://scidac.psc.edu/

[12] S. Shende, A. Malony, A. Morris, S. Parker, J. de St. Germain, "Performance Evaluation of Adaptive Scientific Applications using TAU," *International Conference on Parallel Computational Fluid Dynamics*, 2005.

[13] S. Shende, "The Role of Instrumentation and Mapping in Performance Measurement," Ph.D. Dissertation, University of Oregon, 2001.

[14] M. Syamlal, W. Rogers, and T. O'Brien, "MFIX documentation: Theory Guide," Technical Note, DOE/METC-95/1013, 1993.

[15] M. Syamlal, "MFIX documentation: Numerical Technique," EG&G Technical Report, DE-AC21-95MC31346, 1998.

[16] P. Worley, "Phase Modeling of a Parallel Scientific Code," *Scalable High-Performance Computing Conference (SHPCC)*, pp. 322–327, 1992.