Supporting Nested OpenMP Parallelism in the TAU Performance System

Alan Morris, Allen D. Malony, and Sameer S. Shende

Performance Research Laboratory,
Department of Computer and Information Science
University of Oregon, Eugene, OR, USA,
{amorris,malony,sameer}@cs.uoregon.edu

Abstract. Nested OpenMP parallelism allows an application to spawn teams of nested threads. This hierarchical nature of thread creation and usage poses problems for performance measurement tools that must determine thread context to properly maintain per-thread performance data. In this paper we describe the problem and a novel solution for identifying threads uniquely. Our approach has been implemented in the TAU performance system and has been successfully used in profiling and tracing OpenMP applications with nested parallelism. We also describe how extensions to the OpenMP standard can help tool developers uniquely identify threads.

Keywords: OpenMP, nested parallelism, TAU

1 Introduction

OpenMP [1] research systems have supported nested parallelism since its introduction in the OpenMP standard (e.g., [2, 3]), and most commercial compilers now support nested parallelism in their products. Although some commercial packages provide tools for debugging and performance analysis in the presence of

nested parallelism (e.g., Sun Studio [4] and Intel's tools [5]), the recent OpenMP 2.5 specification [6] does not provide sufficient support for developing portable performance measurement and analysis tools with nested parallelism awareness. This deficiency is being discussed in the OpenMP tools community [7] and hopefully will be addressed in future OpenMP specifications.

In the meantime, there is interest in studying how performance measurement systems can determine nesting context during execution in order to capture performance data for threads and interpret the data vis à vis nesting level. In this paper, we present extensions to the TAU performance system [8] to support nested parallelism. In section 2, we describe our parallel performance system, TAU. We describe the current problems facing tool developers in supporting nested parallelism in Section 3 and discuss two possible solutions in Section 4. Based on this approach, we developed an improved, novel method for the TAU performance system. This is described in Section 5. The TFS application [9, 10] from RWTH Aachen is used as a case study for the TAU solution. Section 6 provides a detailed performance analysis of a nested parallel execution of TFS.

The present issues for portable performance measurement of OpenMP nested parallel execution are, as remarked above, hopefully temporary. In Section 7, we outline discussions underway in the OpenMP tools community and what might be expected in the future to address the problem. Conclusions are given in Section 8.

2 TAU Performance System

The TAU performance system, is composed of three main components: instrumentation, measurement, and analysis. The process of instrumentation consists of inserting measurement probes into an application. The measurement library, invoked by these probes, performs measurements during execution to collect per-

formance data. Analysis tools are used to process and study the collected data and to visualize it.

2.1 Instrumentation

In order to observe performance, additional instructions or probes are typically inserted into a program. This process is called *instrumentation*. From this perspective, the execution of a program is regarded as a sequence of significant events. As the events execute, they activate the probes which perform measurements. Thus, instrumentation exposes key characteristics of an execution.

TAU implements a flexible instrumentation model that permits a user to insert performance instrumentation hooks into the application at several levels of program compilation and execution. The C, C++, and Fortran languages are supported, as well as standard message passing (e.g., MPI) and multi-threading (e.g., POSIX threads, OpenMP) libraries, as well as hybrid executions.

2.2 Measurement

TAU provides a variety of measurement options that are chosen when TAU is installed. Each configuration of TAU is represented in a set of measurement libraries and a stub makefile representing the TAU configuration. Profiling and tracing are the two performance evaluation techniques that TAU supports. Profiling presents aggregate statistics of performance metrics for different events and tracing captures performance information in timestamped event logs for analysis. In tracing, we can observe, along a global timeline, when events take place in different processes and threads. Events tracked by both profiling and tracing include entry and exit from routines, interprocess message communication events, and other user-defined atomic events. Tracing has the advantage of capturing temporal relationships between event records, but at the expense of generating

large trace files. The choice to profile trades the loss of temporal information with gains in profile data efficiency.

Within the space of profiling, there is a wide range of data that can be collected. First, with standard flat profiling, we can track both wallclock time as well as hardware performance counters. Several data elements are recorded for each measured event, including the inclusive and exclusive time, the number of event invocations, and the number of instrumented child event invocations.

TAU also provides the capability to track *callpath* profiles. Here, the objective is to determine the distribution of measured metrics along the dynamic routine (event) calling paths of an application. We speak of the *depth* of a callpath as the number of parent routines included in each path. A callpath of depth 1 is a flat profile. A callpath of depth k represents a sequence of the last k - 1 routines called by a routine at the head of the callpath. The key concept to understand for callpath profiling is that a callpath represents a performance event. Just as a callpath of depth 1 will represent a particular routine and TAU will profile exclusive and inclusive values for that routine, every unique callpath of depth k in a program's execution will represent a unique performance event to be profiled.

2.3 Analysis

The TAU performance system includes several analysis and visualization tools for performance data. It provides simple command line reporting tools as well as advanced GUI analysis tools and a performance data management system.

ParaProf[11] is the primary performance data viewer in TAU. It is capable of calculating simple statistics (e.g. mean, standard deviation) and displaying bar charts and histograms for flat profile data. With callpath data, ParaProf can display expandable tree-tables as shown in Figure 7, and call-graphs as shown in Figure 6. Additionally, large scale data can be displayed in three-dimensional interactive charts such as Figure 4.

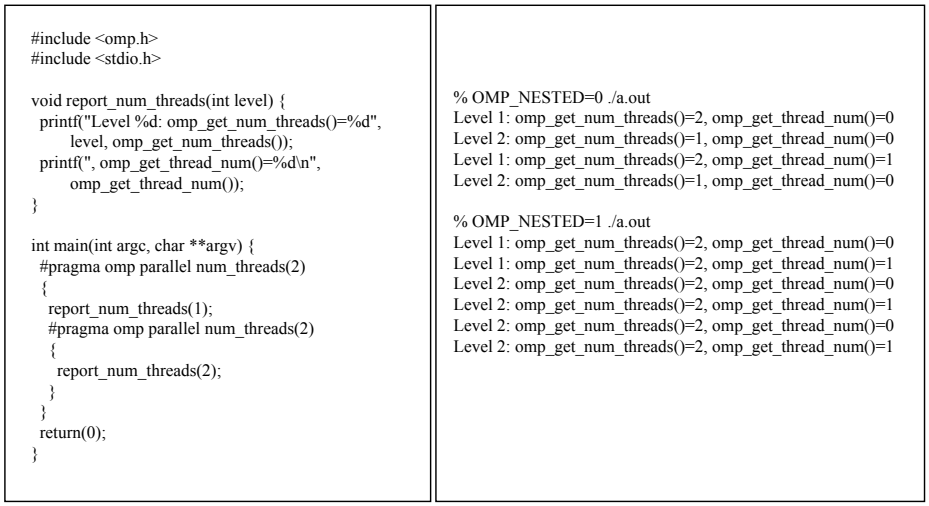
Empirical performance evaluation of parallel applications can generate significant amounts of performance data and analysis results from multiple experiments as performance is investigated and problems diagnosed. To better manage performance information, we developed the Performance Data Management Framework (PerfDMF) [12]. PerfDMF utilizes a relational database to store performance profiles. It provides an abstract profile query and analysis programming interface, and a toolkit of commonly used utilities for building and extending performance analysis tools.

Large scale performance studies may involve extremely large performance datasets from repeated executions across multiple platforms with varying parameters. The potential size of datasets and the need to assimilate results from multiple experiments makes it a daunting challenge to not only process the information, but discover and understand performance insights. In order to perform analysis on these large collections of performance experiment data, we developed PerfExplorer[13], a framework for parallel performance data mining and knowledge discovery. The framework architecture enables the development and integration of data mining operations that will be applied to large- scale parallel performance profiles. PerfExplorer operates as a client- server system and is built on PerfDMF to access the parallel profiles and save its analysis results. The analysis is integrated with existing analysis toolkits (R, Weka), and provides for analysis extensions in those toolkits.

3 Issues with Nested Parallelism in OpenMP

OpenMP allows for nested parallel regions during execution. Nested parallelism can be enabled and disabled through the use of the `OMP_NESTED` environment

variable or by calling the *omp_set_nested()* routine. A simple example is given below:[1]



```
#include <omp.h>
#include <stdio.h>

void report_num_threads(int level) {
  printf("Level %d: omp_get_num_threads()=%d",
        level, omp_get_num_threads());
  printf(", omp_get_thread_num()=%d\n",
        omp_get_thread_num());
}

int main(int argc, char **argv) {
  #pragma omp parallel num_threads(2)
  {
    report_num_threads(1);
    #pragma omp parallel num_threads(2)
    {
      report_num_threads(2);
    }
  }
  return(0);
}
```

```
% OMP_NESTED=0 ./a.out
Level 1: omp_get_num_threads()=2, omp_get_thread_num()=0
Level 2: omp_get_num_threads()=1, omp_get_thread_num()=0
Level 1: omp_get_num_threads()=2, omp_get_thread_num()=1
Level 2: omp_get_num_threads()=1, omp_get_thread_num()=0

% OMP_NESTED=1 ./a.out
Level 1: omp_get_num_threads()=2, omp_get_thread_num()=0
Level 1: omp_get_num_threads()=2, omp_get_thread_num()=1
Level 2: omp_get_num_threads()=2, omp_get_thread_num()=0
Level 2: omp_get_num_threads()=2, omp_get_thread_num()=1
Level 2: omp_get_num_threads()=2, omp_get_thread_num()=0
Level 2: omp_get_num_threads()=2, omp_get_thread_num()=1
```

Fig. 1. An example illustrating nested OpenMP parallelism, the output (right) is obtained by executing the program on the left

Figure 1 illustrates the effects of nested OpenMP parallelism. Without nested parallelism enabled, the inner region pragma has no effect. Each thread from the outer region will process the inner region in serial for a total of two executions of the inner region. When nested parallelism is enabled, both the inner and outer regions will be parallelized independently and will have two threads in each team, resulting in four executions of the inner region.

This example also illustrates that the *omp_get_thread_num()* runtime call cannot be used for unique thread identification within the process. While four threads are active at one time, the values from *omp_get_thread_num()* are only zero and one since the maximum size of a given team is two.

[1] Adapted from `http://docs.sun.com/source/819-0501/2_nested.html`.

Nested OpenMP parallelism poses a challenge to traditional performance analysis tools. The above example is useful in pointing out the issues. Typically, a performance tool will attempt to measure the performance for each thread individually. To do so, there must be an agreement between the application and performance tool for proper thread identification to occur and measured events appropriately assigned. Tools often require that the user configure them with the same thread package that the application is using, be it Pthreads, OpenMP, or another thread package. When the tool and the application use the same underlying thread package, proper thread identification can be done. Moreover, to provide proper thread safety, the measurement system must be aware of the thread environment to perform proper synchronization when necessary.

However, when nested parallelism is used in OpenMP, the nesting context is not available to the performance interface. It may appear that nested parallelism can be statically analyzed and a tool such as Opari [14, 15] could insert additional instrumentation providing nesting information. However, static analysis is insufficient to track threads as the varied interactions of execution paths at runtime can create arbitrary nesting depths. A runtime solution for thread identification is necessary.

Native thread libraries typically provide thread local storage (TLS) that performance tools use to track thread identities and other thread specific state information. The OpenMP runtime library provides the *omp_get_thread_num()* API call that returns the thread number or identifier within the active team of threads. Unfortunately, the value returned from this call lies between 0 and *omp_get_num_threads()-1*, which is not the total number of threads in the program, but the size of the team from which the call is made. TAU and other tools have traditionally used this call for thread identification. When an instrumented section of code is entered, the measurement library identifies the calling thread

and performs measurements associated with that thread. Thread identification is critical because the measurement system must internally maintain the callstack for each running thread. Each of these data structures are keyed by the thread identifier.

However, when using nested OpenMP parallelism, multiple teams may be active at any one time and more than one active thread will return the *same* index from *omp_get_thread_num()*. This will cause errors in the measurement system as it will observe what appears to are overlapping timers where a thread enters one routine, but exits from a different one.

Nested parallelism in OpenMP offers the additional challenge of mapping the per-thread performance data back to the nested parallelism abstractions in the source code. To do so, it is necessary to distinguish the performance measurements for each thread based on the nesting context. When done properly, the execution profile should resemble a tree-like hierarchy of threads.

4 Solutions

The problem of thread identification in nested OpenMP programs is widespread in the community, for purposes other than performance evaluation. As such, several solutions have been proposed.

4.1 Extending the OpenMP API

A promising solution to this problem is to extend the OpenMP specification itself to allow for more in depth query and knowledge of nested parallelism. Dieter an Mey, RWTH proposed an extension to the OpenMP specification in the form of runtime library calls that return the current nesting level, the number of threads within that level, and the current thread identifier at that level. This

provides enough information to uniquely identify threads for performance measurement purposes as well as information necessary for the proper mapping of the runtime execution to the application developer's model of nested parallelism in the application.

Ultimately, we hope that the OpenMP specification will be extended in this manner, and we will update TAU to use the new runtime calls when these become widely available.

4.2 Native Thread Library Hooks

Another method of tracking threads in the face of nested OpenMP parallelism involves bypassing the OpenMP runtime system entirely and instead tracking the threads based on the underlying thread implementation. For example, if the OpenMP thread package is implemented using the native POSIX thread library, the tool could use the Pthreads API for thread identification, invoking functions such as *pthread_self()* which provides a capability similar to the *getpid()* function for processes. Given that the measurement library is now hooked to the underlying thread library, it could then use the available thread local storage using the native thread API.

A major drawback to this approach is the lack of portability. The underlying thread library must be known and accessible to the performance tool. On some systems, the underlying thread substrate may be inaccessible and such an approach cannot be guaranteed to work universally. We favor approaches that follow the higher-level abstract OpenMP API. Additionally, this approach makes it difficult to determine parent child relationships arising from nested parallelism.

4.3 Additional OpenMP Instrumentation

Alexander Spiegel, RWTH proposed another solution to this problem, in which the master and worker threads of each parallel team exchange information through

a global shared space which is locked by the master. At the start of parallel regions, additional code must be inserted to perform this task.

The additional code does roughly the following. At the beginning of the parallel region, the master thread, that is, the thread that was executing the code prior to the parallel section, obtains a lock on a shared storage space. Next, all of the threads synchronize (e.g. *#pragma omp barrier*) and read the buffer. After the data has been read by all threads and another sychronization occurs (to ensure that all threads have consumed the data), the master can then unlock the shared storage space.

The data that the master disseminates to the children will include the current nesting context including that execution callstack's thread identifier within each parent's parallel region. In this way, each new parallel region inherits the relevant context information from the parent, and a proper mapping can take place.

A tool such as Opari could be extended to support the additional instrumentation required for this type of thread synchronization at the source level. This approach has the advantage that it tracks the full nesting information, so at any given time, the performance tool can identify which team member from each team and each level of nesting is invoking a measurement point. This allows for a better mapping of thread level performance information back to the source code and application developer's understanding of nested parallelism.

The drawback of this approach is of course the additional synchronization and locking at parallel region boundaries. We have not performed studies to measure the overhead involved, but we estimate that it might be significant for some programs.

5 OpenMP support in TAU

The TAU performance system supports performance evaluation of OpenMP programs at the region, construct, and routine level. Additionally, the user may manually instrument regions of code using the TAU API. For OpenMP-based applications, the Opari tool is used to insert instrumentation based on the POMP [14] interface at OpenMP regions and for OpenMP constructs. The Program Database Toolkit (PDT)[16] is used to instrument OpenMP source at the routine level.

Construct level measurement refers to the tracking of OpenMP constructs by themselves. The resulting profile will show time spent in each type of OpenMP construct. For example, the time spent in "`for enter/exit [OpenMP]`" and "`barrier enter/exit [OpenMP]`" represents the aggregate time spent in all OpenMP `for` and `barrier` constructs.

Region level measurement goes beyond construct measurement to provide specific information about each separate parallel region. For example, a timer called "`for [OpenMP location: file:mandel.cpp <96, 109>]`" represents the time spent in the particular OpenMP `for` region from lines 96 to 109 in the file `mandel.cpp`. Thus, the construct measurements are partitioned based on the region to which they apply.

During measurement, TAU uses the OpenMP thread synchronization calls for updating shared performance data structures when necessary. Construct-based measurement uses globally accessible timers to aggregate construct-specific performance costs over all OpenMP regions. For region-based measurement, the region descriptor is used to partition the construct information for each region.

In our earlier work [14, 17], TAU relied upon the OpenMP runtime API call, *omp_get_thread_num()*, to distinguish threads for indexing into the runtime performance data structures. Unfortunately, this method is inadequate for nested

parallelism due to the issues discussed above. Instead, we need a mechanism to uniquely identify the current thread.

5.1 Nested Parallelism Support

We propose a new scheme for thread identification. Our approach involves the use of the OpenMP directive, *#pragma threadprivate()*. The basic idea is to identify and use some piece of data that is unique for each thread and persists within a thread's execution such that the same identifier can be found at the start of a thread's execution of a region until the end of the region. Though the values of a threadprivate variable are not guaranteed to persist between parallel regions, we are at least guaranteed that no two currently active threads will point to the same memory location for a given threadprivate variable. Using this scheme, TAU can then uniquely identify threads even in the presence of nested parallelism.

The approach requires a single threadprivate variable that is initialized inside the TAU library (when TAU is built using OpenMP threading). This variable and/or its address can be used to distinguish it from other threads executing concurrently. When the TAU runtime system encounters a thread that it has not seen before, it registers the thread and assigns it an identifier on a first come, first serve basis.

In contrast to the other proposed approaches, this method has the advantage of faster speed, as no runtime API calls are made in identifying a thread. The thread registration in the TAU runtime system is performed only when a given thread is seen for the first time, so there is no additional overhead at parallel region boundaries. Additionally, this method requires no knowledge of the underlying native thread library, it is based solely on the OpenMP system. The main drawback with this method is that we are unable to identify the nesting depth or specify a team identifier for a given thread. A thread is assigned a

unique identifier, but not necessarily the same identifier between regions, and this typically does not map back to the nested teams within the application source code. Nevertheless, as we see below, this new method produces performance data that can be mapped back to the source code and does expose nested parallelism where it occurs.

6 A Case Study

To demonstrate TAU's support for nested parallelism, we conducted a case study of TFS [9, 10], a computational fluid dynamics code developed at the Aerodynamics Institute at RWTH Aachen. TFS was initially parallelized using ParaWise [18] to generate an intra-block parallelization where the parallel loops operated over a single dimension within a block. Then, a second version was developed that used parallel loops to iterate over blocks. Finally, a hybrid, multi-level version was developed which combined both the intra- and inter- block parallelizations using nested OpenMP parallelism. In recent performance testing, the developers reported a speedup of 21 for TFS using nested OpenMP parallelism on a SunFire 25K system [19].

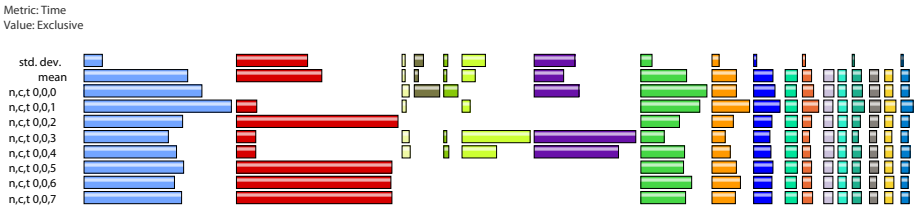


Fig. 2. Flat Profile for TFS

We analyzed the TFS code by first integrating TAU into the TFS build system and instrumented its source code using the TAU's compilation scripts.

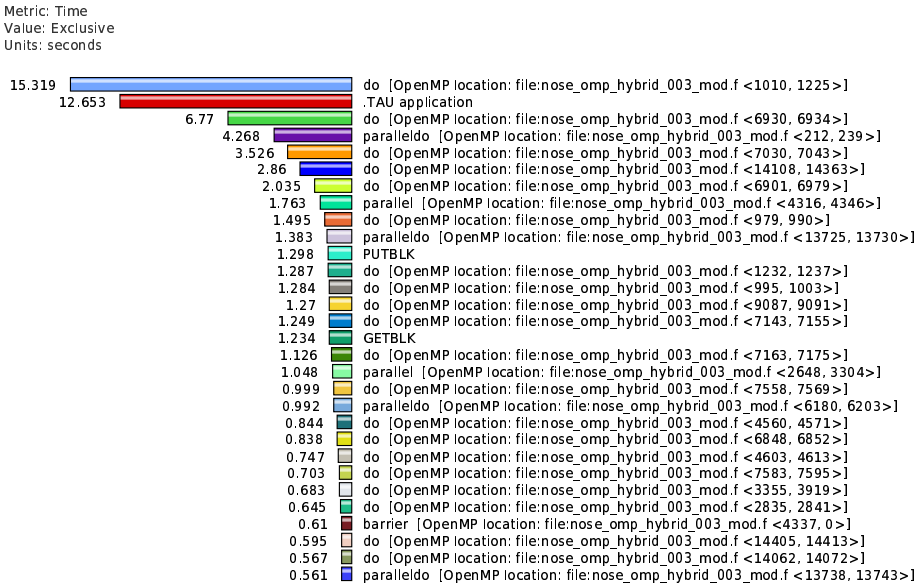


Fig. 3. Statistical Mean Profile for TFS

This process required only a single modification to the TFS build system where the name of the compiler used in the makefile was changed from `FC=f90` to `FC=tau_f90.sh`. This script acts as a compiler wrapper, allowing for automatic instrumentation of Fortran programs (similarly, `tau_cc.sh` and `tau_cxx.sh` are used for C and C++ compilation respectively) . In the case of OpenMP code, it automatically invokes the Opari tool to instrument OpenMP constructs and regions by rewriting OpenMP directives using the POMP interface. The code is then parsed by PDT to create PDB files that contain source-level information about the routine names, and their respective entry and exit locations. The script then instruments the entry and exit points using the *tau_instrumentor* utility. Finally, the instrumented code is linked with the TAU library.

We ran TFS on the SunFire machines at RWTH Aachen using 8 threads. With TAU's support for nested OpenMP parallelism, we can run the instrumented version of TFS without the thread identifier clash that occurred previ-

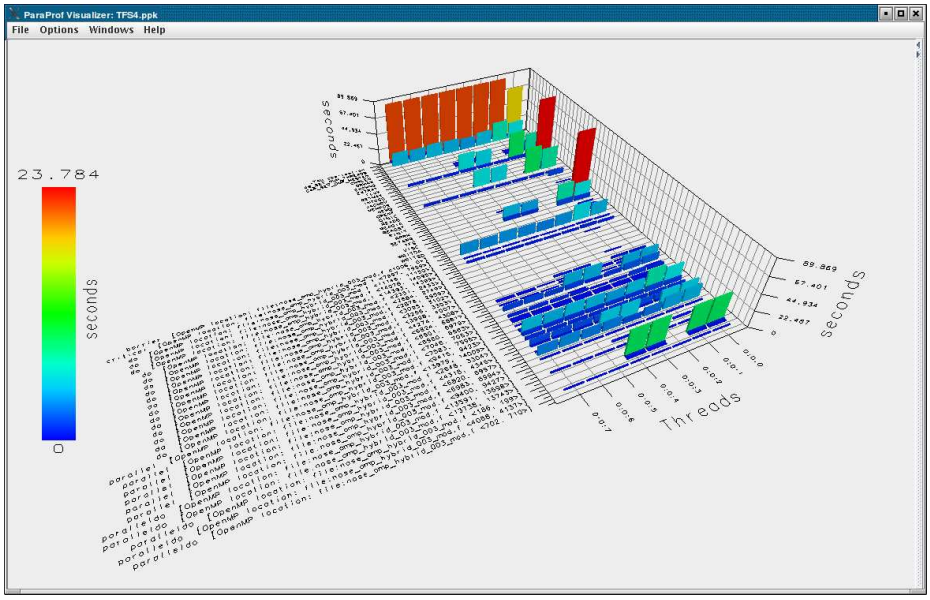


Fig. 4. 3D display of TFS performance data (inclusive time)

ously. The flat profile for TFS is shown in Figure 2. Each thread is represented by a row in the graph, and each timer/region is a column. The event in the second column (colored red) is `.TAU application`, a special timer created at the start of thread execution which is then stopped at the end of program execution. The effect is to show roughly how much time each thread spends idle since its exclusive time is roughly the wall clock time of the application minus parallel regions.

Figure 3 shows the statistical mean exclusive value across all threads. Note that the timer names for parallel regions and constructs contain the source filename and line number. This data is provided by the Opari tool through the POMP interface.

By looking at the flat profile we find that there is a clear pattern in the data wherein threads 0, 1, 3, and 4 have a similar execution profile, and threads 2, 5, 6 and 7 are also very similar. This pattern is also visible in the three dimensional

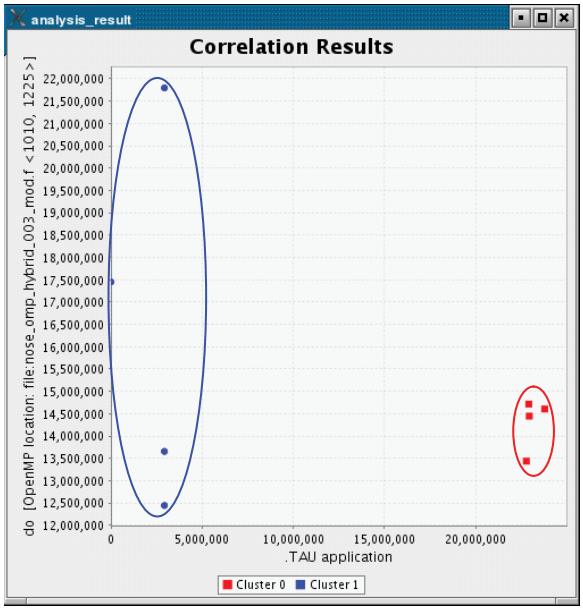


Fig. 5. Clustering of threads in TFS

display of ParaProf shown in Figure 4. The three axes are the threads, the timers (functions), and the exclusive time spent in the given timer.

TAU's PerfExplorer data mining tool is able to automatically discover patterns such as these. Shown in Figure 5, PerfExplorer displays the result of a correlation analysis which clearly shows two clusters of threads. These are the same clusters that we observed from the flat profile bar graphs.

Figure 6 shows the callgraphs from each observed thread of execution. Again, we see two groups of four very similar threads. The main thread is slightly different in that it executes all of the non-parallelized code.

For a total runtime of 90 seconds, the second cluster which includes threads 2, 5, 6, and 7 are idle for about 22 seconds each, whereas the other slave threads, numbers 1, 3, and 4 spend only about 3 seconds idle. Each cluster executes different functions that may be seen in ParaProf's callgraph displays. With this

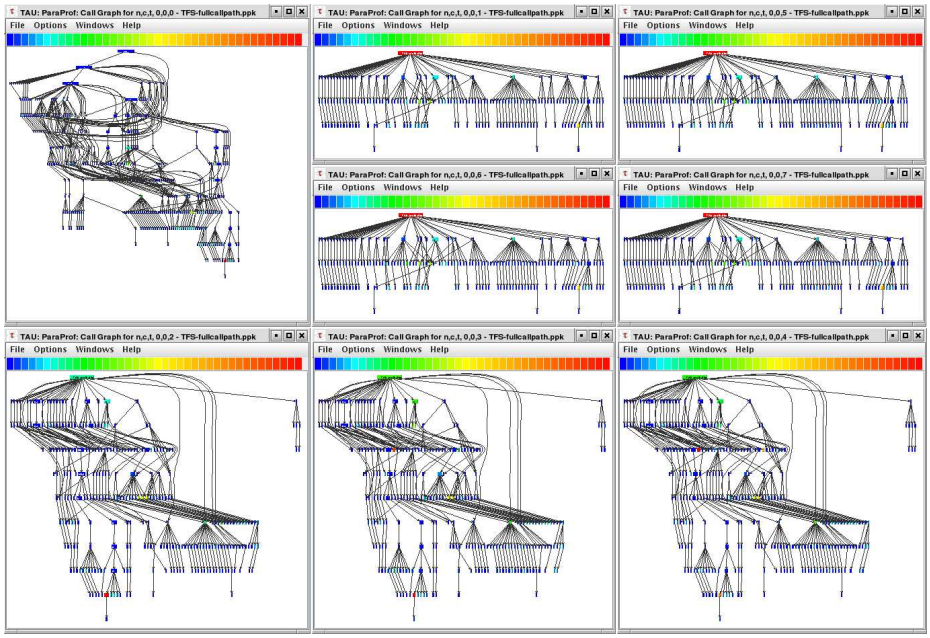


Fig. 6. The callgraph of each thread from an 8 thread run of TFS

knowledge, the application developer can identify the regions that each thread executes and map the execution back to the source code.

Using TAU's callpath profiling capability, the nested parallelism present in TFS is easily decomposed. Figure 7 shows where time was spent at each level of nesting. The function `ALGO` started a parallel region, and deeper in the callpath, the function `AUSM` also started a parallel region.

7 Future Work

As noted earlier, there is active discussion underway in the OpenMP tools forum as to what the appropriate interface should be for runtime tools such as performance measurement libraries. We hope that runtime system functions will be made available for thread identification and nesting context to be queried.

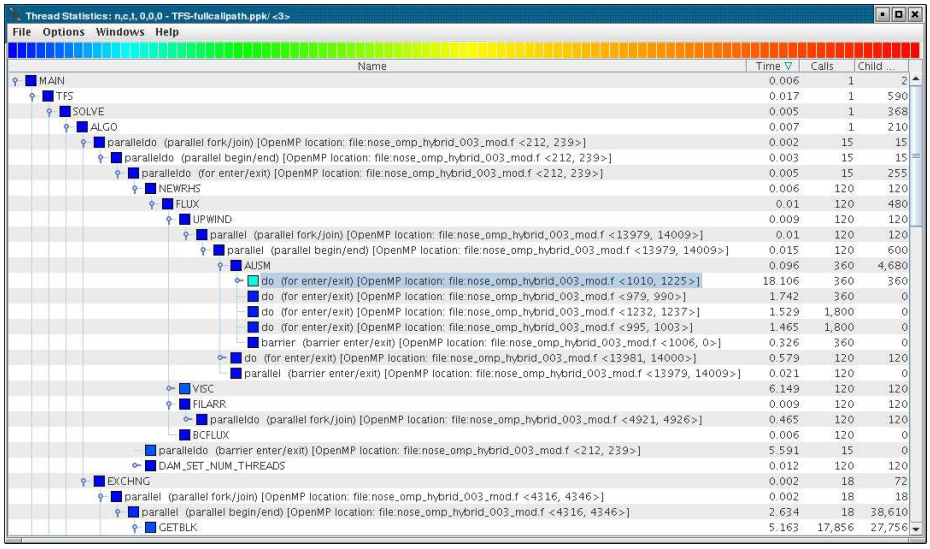


Fig. 7. Nested Parallelism in TFS

For the purposes of enhancing the TAU performance system, we would like to support a higher level mapping of thread identifiers back to the application developer's model of nested parallelism. In the case of non-nested parallelism, TAU already provides a clear picture of the execution performance of each thread in each team. This picture is currently not as clear in the nested case because we have only a single number to identify a thread. We anticipate adding support for thread naming in TAU wherein a thread is labeled in the nested OpenMP case by its nesting depth and identifier, or by the identifier in each team where it originated (such as "thread $0 \rightarrow 3 \rightarrow 2$"). This will allow us to display performance instrumentation based on the hierarchical structure of each thread team. The runtime, hopefully provided in a future OpenMP specification, will provide the necessary information to maintain these structures in the measurement system.

8 Conclusion

Performance evaluation of OpenMP codes is critical in making the greatest use of SMP-capable machines, both in the the small and large scale. Application developers rely on performance tools such as TAU to study the performance of their programs under varying conditions.

Performance tools that measure per-thread performance data must be able to uniquely identify threads of execution at runtime in order to assign resulting measurements. Additionally, they must support concurrent multi-threaded execution with simultaneous event invocations across all the threads in the process.

This process is complicated in the presence of nested parallelism in OpenMP when thread identities queried from the runtime system only represent identities within a subset of the currently active threads (the local team). Additionally, there is no support in the specification for accessing information about nesting context. In this paper, we describe the nested parallelism problem currently faced by tools for portable OpenMP performance analysis. Several possible solutions are discussed. The approach we implemented in the TAU performance system uses thread private storage to create a unique thread identifier. The approach is portable and has been validated with both Sun and Intel's OpenMP compilers. We demonstrated TAU's new capability to track nested OpenMP parallelism in the TFS application and analyzed a sample execution.

9 Acknowledgments

Research at the University of Oregon is sponsored by contracts (DE-FG02-05ER25663, DE-FG02-05ER25680) from the MICS program of the U.S. Dept. of Energy, Office of Science.

References

1. OpenMP, http://www.openmp.org/drupal/.
2. Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa, "Performance Evaluation of OpenMP Applications with Nested Parallelism," in Languages, Compilers, and Run-Time Systems for Scalable Computers, pp. 100–112, 2000.
3. M. Gonzalez, E. Ayguade, X. Martorell, J. Labarta, N. Navarro, and J. Oliver, "NanosCompiler: Supporting Flexible Multilevel Parallelism Exploitation in OpenMP," *Concurrency - Practice and Experience*, 12(12):1205–1218, 2000.
4. Sun Studio compilers, http://developers.sun.com/prodtech/cc, 2006.
5. Intel compilers, http://www.intel.com/cd/software/products/asmo-na/eng/compilers, 2006.
6. OpenMP API Specification 2.5, http://www.openmp.org/drupal/mp-documents/spec25.pdf, May 2005.
7. OpenMP tools mailing list, Omp-tools@openmp.org, http://openmp.org/mailman/listinfo/omp-tools.
8. S. Shende and A. D. Malony, "The TAU Parallel Performance System," International Journal of High Performance Computing Applications, SAGE Publications, 20(2):287-331, Summer 2006
9. E. Fares, M. Meinke, W. Schröder, "Numerical Simulation of the Interaction of Flap Side-Edge Vortices and Engine Jets," *Proceedings of the 22nd International Congress of Aeronautical Sciences, ICAS 0212*, Sep. 2000.
10. E. Fares, M. Meinke, W. Schröder, "Numerical Simulation of the Interaction of Wingtip Vortices and Engine Jets in the Near Field," *Proceedings of the 38th Aerospace Sciences Meeting and Exhibit*, AIAA Paper 20002222, January 2000.
11. R. Bell, A. D. Malony, and S. Shende, "A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis", Proc. EUROPAR 2003 conference, LNCS 2790, Springer, Berlin, pp. 17-26, 2003.
12. K. A. Huck, A. D. Malony, R. Bell, and A. Morris, "Design and Implementation of a Parallel Performance Data Management Framework," Proc. International Conference on Parallel Processing (ICPP 2005), IEEE Computer Society, 2005.

13. K. A. Huck, and A. D. Malony, "PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing," In Proceedings of SC 2005 conference, ACM, 2005.
14. B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting," Proceedings of Third European Workshop on OpenMP, (EWOMP 2001), Sep. 2001.
15. B. Mohr, and F. Wolf, "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications," *Proc. of the European Conference on Parallel Computing (EuroPar)*, Springer-Verlag, LNCS 2790, pp. 1301-1304, August 26-29, 2003.
16. K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates," SC 2000 conference, 2000.
17. B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP," The Journal of Supercomputing, 23, Kluwer, pp. 105–128, 2002.
18. ParaWise, `http://www.parallelsp.com/parawise.htm`, 2006.
19. I. Hörschler, S. P. Johnson, D. an Mey, "100 (Processor) Years Simulation of Flow through the Human Nose using OpenMP," `http://www.rz.rwth-aachen.de/computing/events/2005/sunhpc_colloquium/07_Hoerschler.pdf`, 2006.