# 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

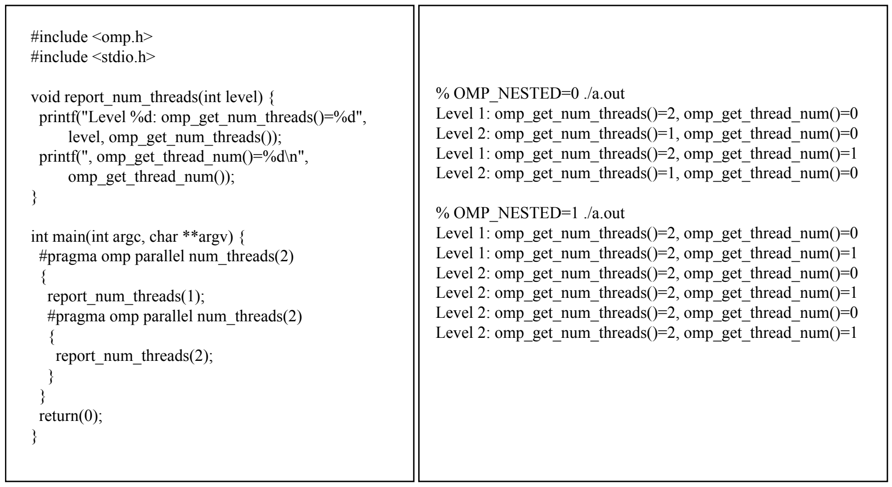
OpenMP research systems have supported nested parallelism since its introduction in the OpenMP standard (e.g., [12, 13]), and most commercial compilers now support nested parallelism in their products. Although some commercial packages provide tools for debugging and performance analysis in the presence of nested parallelism (e.g., Sun Studio [18] and Intel [20]), the recent OpenMP 2.5 specification [21] 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 [16] 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 the current problem in Section §2 and discuss two possible solutions in Section §3. Based on this approach, we developed an improved, novel method for the TAU performance system [1]. This is described in Section §4. The TFS application [14, 15] from RWTH Aachen is used as a case study for the TAU solution. Section §5 provides a detailed performance analysis of a nested parallel execution of TFS. The TAU parallel profile displays clearly show the thread nesting relationships.

The present issues for portable performance measurement of OpenMP nested parallel execution is, as remarked above, hopefully temporary. In Section §6 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 §7.

## 2 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. When nested parallelism is enabled, both the inner and outer regions will have 2 threads in each team, whereas without nested parallelism, only the outer region have 2 threads. Here, we also see that the *omp_get_thread_num()* runtime call cannot be used for unique thread identification.

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

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

appropriately assigned. Tools often require that the user configure them with the application's thread package, be it pthreads, sproc, or OpenMP. When the tool and the application use the same underlying thread package, proper thread identification can be done.

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 [4, 8] 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 provide thread local storage (TLS) that performance tools use to track thread identities. 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 number of threads in the current active team. TAU and other tools have traditionally used this call for thread identification. When an instrumented section of code is entered, the profiling library identifies the calling thread and performs measurements associated with that thread. We say unfortunately because this approach does not allow the performance measurement system to uniquely identify threads when nested parallelism is active. When using nested OpenMP parallelism, multiple teams may be active at any one time and more than one thread will return the *same* index from *omp_get_thread_num()*.

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 distinguished the performance measurements for each thread with the nesting context.

## 3 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.

### 3.1 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 [9] proposed an extension to the OpenMP specification in the form of a 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 abstractions for 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.

### 3.2 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 pthread library, the tool could use the pthreads API for thread identification, invoking functions such as *pthread_self()*. Regular thread local storage would be available as well.

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.

### 3.3 Additional OpenMP Instrumentation

Alexander Spiegel, RWTH [10] 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, code needs to be inserted such that the master stores the data, locks the shared space, then after a barrier, the entire team of threads reads the shared data, and after another barrier, the master unlocks it. In this way, each new parallel region inherits data from the parent region, and a proper mapping can take place.

A tool such as Opari can be extended to support the additional instrumentation required for this type of thread synchronization at the application-level. This approach has the advantage that it tracks the full nesting information, so at any given time, the performance tool can know which thread identifier from each team and each level of nesting is executing. This allows for a better mapping of thread level performance information back to the source code.

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 in some programs.

## 4 TAU's Solution

The TAU performance system supports performance evaluation of OpenMP programs at the region, construct, and routine level. The Opari tool is used to inserting instrumentation based on the POMP [4] interface at OpenMP regions and for OpenMP constructs. PDT [3] is used to instrument OpenMP source at the routine level. During measurement, TAU uses the OpenMP thread synchronization calls for updating the shared performance data structures. 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 select specific performance data for that context.

In our earlier work [4, 5], TAU relied upon the *omp_get_thread_num()* OpenMP API call 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.

Our approach is to use *#pragma threadprivate()*. 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 address space 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 done only when a given thread is seen for the first time, so there is no additional overhead at parallel region boundaries. 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 subsequence invocations, and this typically does not map back to any explicit parallelism in the source code. Nevertheless, as we see below, the method produces performance data that can be mapped back to the source code itself and does expose nested parallelism where it occurs.

## 5 A Case Study

To demonstrate TAU's support for nested parallelism, we conducted a case study with TFS [14, 15], a computational fluid dynamics code developed at Aerodynamics Institute at RWTH Aachen. TFS was initially parallelized using ParaWise [19] to generate an intra-block parallel version 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, multilevel version was developed which combined the intra and inter block version 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 [17].

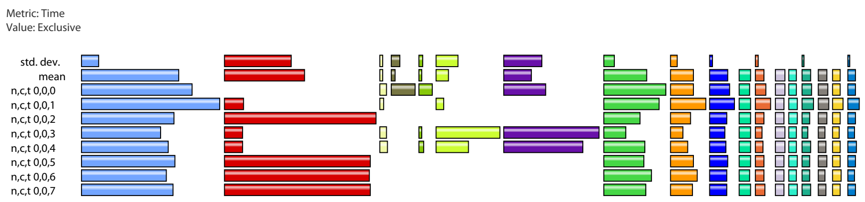


**Fig. 2.** Flat Profile for TFS

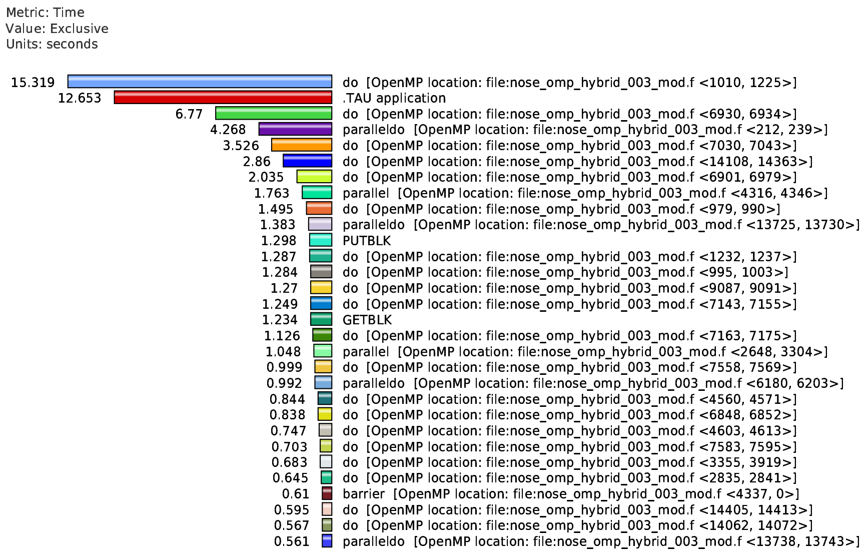


**Fig. 3.** Mean Profile for TFS

We integrated TAU in TFS and instrumented its source code using the TAU's compilation scripts [1]. This process required only a single modification to the TFS build system where the name of the compiler used in the make-file was changed from `FC=f90` to `FC=tau_f90.sh`. This script act as compiler wrapper, allowing for automatic instrumentation of Fortran programs. In the case of OpenMP code, it will automatically invoke 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 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

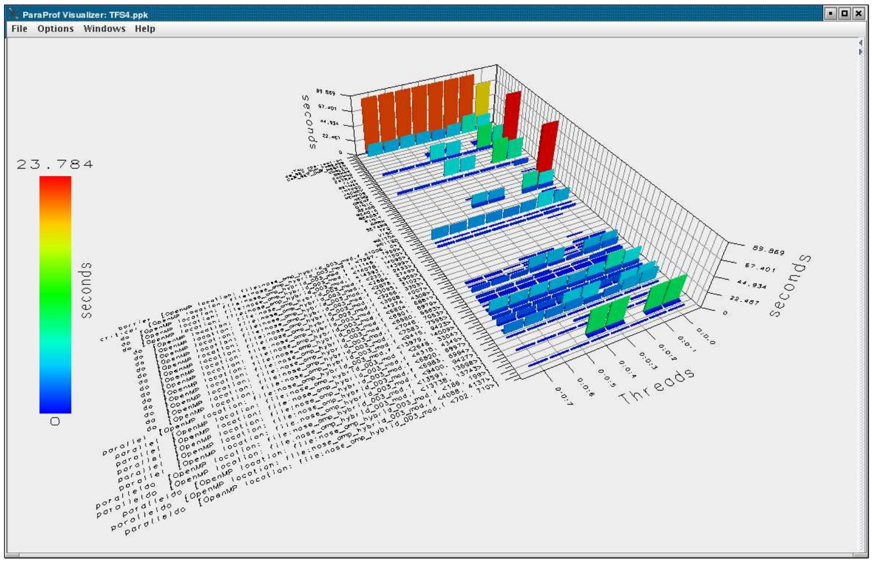


**Fig. 4.** 3D display of TFS performance data (inclusive time)

TAU's *tau_instrumentor* utility. Finally, the instrumented code is linked with the TAU library.

We ran TFS on the Sun Fire 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 previously. 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 second column timer (red) is `.TAU application`, which in this case represents the global time that a thread spent idle (not doing useful work).

Figure 3 shows the mean data for 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.

There is a clear pattern in the data wherein threads 0, 1, 3, and 4 do similar processing, and threads 2, 5, 6 and 7 are also very similar. This pattern is also visible in the three dimensional 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 [6] tool is able to automatically discover patterns such as this. PerfExplorer is a performance data mining package that operates on a relational database using statistical packages such as Weka and R. Shown in Figure 5, PerfExplorer performs a correlation analysis and splits the threads into clusters. These are the same clusters that we observed from the flat profile bar graphs.

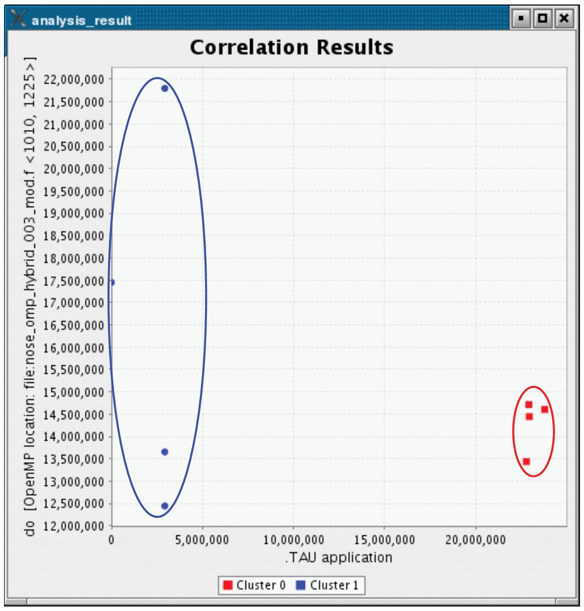


**Fig. 5.** Clustering of threads in TFS

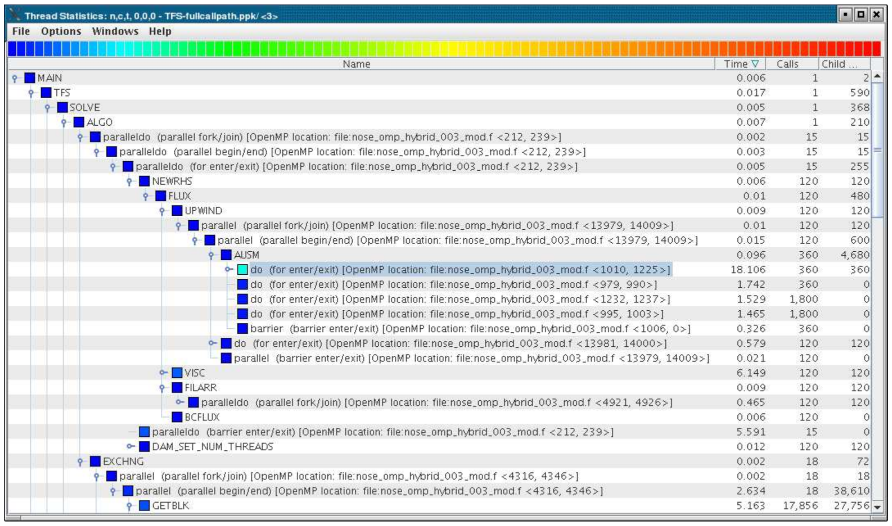
For a total runtime of 90 seconds, the second cluster which includes threads 2, 5, 6, and 7 are idle for about 22 seconds each, whereas the other slave threads, numbers 1, 3, and 4 spend only about 3 seconds idle. Each cluster executes different functions that may be seen in ParaProf's callgraph displays (not shown). With this knowledge, the application developer can note 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 6 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.

## 6 Future Work

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

For TAU's application, 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 provides a clear picture of the 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



```
Name                                                                                              Time    Calls   Child ...
MAIN                                                                                              0.006       1        2
  TFS                                                                                             0.017       1      590
    SOLVE                                                                                         0.005       1      368
      ALGO                                                                                        0.007       1      210
        paralleldo  (parallel fork/join) [OpenMP location: file:nose_omp_hybrid_003_mod.f <212, 239>]  0.002  15   15
          paralleldo  (parallel begin/end) [OpenMP location: file:nose_omp_hybrid_003_mod.f <212, 239>]  0.003  15  15
            paralleldo  (for enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <212, 239>]  0.005  15  255
              NEWRHS                                                                              0.006     120      120
                FLUX                                                                              0.01      120      480
                  UPWIND                                                                          0.009     120      120
                    parallel  (parallel fork/join) [OpenMP location: file:nose_omp_hybrid_003_mod.f <13979, 14009>]  0.01  120  120
                      parallel  (parallel begin/end) [OpenMP location: file:nose_omp_hybrid_003_mod.f <13979, 14009>]  0.015  120  600
                        AUSM                                                                      0.096     360    4,680
                          do  (for enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <1010, 1225>]  18.106  360  360
                          do  (for enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <979, 990>]  1.742  360  0
                          do  (for enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <1232, 1237>]  1.529  1,800  0
                          do  (for enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <995, 1003>]  1.465  1,800  0
                          barrier  (barrier enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <1006, 0>]  0.326  360  0
                        do  (for enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <13981, 14000>]  0.579  120  120
                        parallel  (barrier enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <13979, 14009>]  0.021  120  0
                  VISC                                                                            6.149     120      120
                  FILARR                                                                          0.009     120      120
                    paralleldo  (parallel fork/join) [OpenMP location: file:nose_omp_hybrid_003_mod.f <4921, 4926>]  0.465  120  120
                  BCFLUX                                                                          0.006     120        0
              paralleldo  (barrier enter/exit) [OpenMP location: file:nose_omp_hybrid_003_mod.f <212, 239>]  5.591  15  0
              DAM_SET_NUM_THREADS                                                                 0.012     120      120
        EXCHNG                                                                                    0.002      18       72
          parallel  (parallel fork/join) [OpenMP location: file:nose_omp_hybrid_003_mod.f <4316, 4346>]  0.002  18  18
            parallel  (parallel begin/end) [OpenMP location: file:nose_omp_hybrid_003_mod.f <4316, 4346>]  2.634  18  38,610
              GETBLK                                                                              5.163  17,856   27,756
```

**Fig. 6.** Nested Parallelism in TFS

anticipate adding support for thread naming in TAU wherein a thread is identified 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"). The runtime, hopefully provided in a future OpenMP specification, will provide the necessary information.

## 7 Conclusion

Performance tools that measure per-thread performance data must be able to uniquely identify threads of execution at runtime. This is complicated in the presence of nested parallelism when thread identities queried from the runtime system are not unique and do not provide 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 its capabilities with the TFS application.

## 8 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. S. Shende, and A. Malony, "The TAU Parallel Performance System," In International Journal of High Performance Computing Applications, ACTS Collection Special Issue, Summer 2006.
2. A. Malony, S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," In G. Kotsis, P. Kacsuk (eds.), *Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Third Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, Kluwer, pp. 37–46, 2000.
3. 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.
4. 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.
5. 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.
6. 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.
7. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, **14**(3):189–204, Fall 2000.
8. 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.
9. Dieter an Mey, "Proposed Light Weight Extensions to the OpenMP Specification," Private Communication, Dec. 2005.
10. Alexander Spiegel, "Proposed Solution to Identifying Threads Uniquely in Nested OpenMP threads," Private Communication, Oct. 2005.
11. OpenMP, `http://www.openmp.org/drupal/`.
12. 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.
13. 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.
14. 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.
15. 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.
16. OpenMP tools mailing list, `Omp-tools@openmp.org`, `http://openmp.org/mailman/listinfo/omp-tools`.
17. 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.

18. Sun Studio compilers, `http://developers.sun.com/prodtech/cc`, 2006.
19. ParaWise, `http://www.parallelsp.com/parawise.htm`, 2006.
20. Intel compilers,
`http://www.intel.com/cd/software/products/asmo-na/eng/compilers`, 2006.
21. OpenMP API Specification 2.5,
`http://www.openmp.org/drupal/mp-documents/spec25.pdf`, May 2005.