A Performance Monitoring Interface for OpenMP

Bernd Mohr[a], Allen D. Malony[b], Hans-Christian Hoppe[c], Frank Schlimbach[c],
Grant Haab[d], Jay Hoeflinger[d], and Sanjiv Shah[d]

[a]Research Centre Jülich, ZAM
Jülich, Germany
b.mohr@fz-juelich.de

[b]University of Oregon
Eugene, Oregon USA
malony@cs.uoregon.edu

[c]Pallas GmbH
Brühl, Germany
hans-christian.hoppe@pallas.com
frank.schlimbach@pallas.com

[d]Intel Corporation, KAI Software Lab
Champaign, Illinois USA
grant.haab@intel.com
jay.p.hoeflinger@intel.com
sanjiv.shah@intel.com

Abstract. This paper reports on efforts to define a performance monitoring interface for OpenMP that merges the OMPI and POMP prototype interfaces developed in the past year. The primary goal is to define a clear and portable API for OpenMP that makes execution events visible to runtime monitoring tools, primarily tools for performance measurement. The proposed specification is presented, covering many relevant design issues and the result of discussions among the involved groups. Both successful convergence of ideas, leading to agreement on proposed specifications, as well as differences in opinion and remaining open issues are documented from our many discussions. The paper is intended to serve as a preliminary proposal for consideration by the OpenMP Architecture Review Board and recently formed Tools sub-committee.

1 Introduction

The OpenMP [5] parallel programming standard has been positively received in the scientific community as a means to achieve portable parallel execution on shared-memory multiprocessor (SMP) systems. The latest version of the OpenMP application programmer's interface (API) for C, C++, and Fortran offers a flexible, scalable, and moderately complete programming environment (compiler directives[1], library routines, and environment variables) for specifying SMP parallel operation. Unfortunately, the current OpenMP specification (Version 2.0) does not offer any support for performance observation beyond an interface for portable timer access. Lack of such support makes it difficult to observe performance behavior and to understand how performance properties map to the OpenMP parallel execution model.

Future acceptance and use of OpenMP will be greatly enhanced by the definition of a common interface for performance monitoring. Fortunately, the OpenMP Architecture Review Board (ARB) is interested in the specification of interfaces for performance monitoring and debugging. This interest is evidenced by the recent formation of an ARB "Tools" sub-committee which will review and discuss tool interface proposals.

Predating these ARB activities, three groups have designed and demonstrated prototypes of a performance monitoring interface for OpenMP. As part of the IST INTONE project [2], Pallas GmbH has developed the *OMPI* [9] "performance instrumentation interface." OMPI is supported by the INTONE Fortran compilation system based on the Nanos compiler by CEPBA [12] in Barcelona, and by the INTONE C compiler developed by KTH in Sweden. Both compilers can be downloaded from the INTONE project page. The *POMP* [4, 10] "performance tool interface" was developed by Forschungszentrum Jülich, the University of Oregon, and the KAI Software Lab (KSL) of Intel. POMP is integrated with the Expert [8] performance analyzer and the TAU [3] performance system, and work is underway at Intel KSL to implement a prototype of the POMP API as part of their ASCI Pathforward contract[2]. KAI

[1] We follow the OpenMP specifications using the term "directive" for both Fortran directives and C/C++ pragmas throughout the paper

[2] Department of Energy Accelerated Strategic Computing Initiative subcontract number B510239, "ASCI Pathforward Ultrascale Tools Ini-

Software Lab is uniquely qualified to propose standards for a performance API for OpenMP. Kuck and Associates, Inc (KSL before acquisition by Intel) develops the successful Kap/Pro Tool Set which includes GuideView, the first OpenMP-focused performance analysis tool.

Presentations on both OMPI and POMP were made to the OpenMP "Futures" committee at their SC 2001 meeting where there was positive feedback and encouragement to develop the ideas further. The respective groups have continued to evolve and refine the specifications for OMPI and POMP since that time.

Both proposals differ somewhat in objectives and approach and focus on different aspects of performance monitoring. Since the OpenMP ARB can only adopt one *performance monitoring interface* for OpenMP, Pallas, Forschungszentrum Jülich, the University of Oregon, and KAI Software Lab have decided to merge their specifications into a single proposal, combining the strengths of both and hopefully eliminating their respective weaknesses. (To remove ambiguity, we adopt the more correct term "performance monitoring interface" in this paper to refer to both OMPI and POMP.) The result of this effort is reported here.

In Section 2, we introduce the problems and objectives of a performance monitoring interface for OpenMP. Both projects addressed these in slightly different ways. Section 3 enumerates the issues which need to be addressed for an OpenMP monitoring interface and describes what we propose as a solution for these issues. Finally, in Section 4, we give conclusions and discuss future work.

2 Problem and Objectives

Experience with the MPI profiling interface (PMPI) [11] demonstrates that a standardized mechanism for instrumentation (in this case, library interposition) can facilitate the implementation of performance measurement and analysis tools (e.g., [7, 1, 3, 8]). Similar in spirit to PMPI, our goal is to define a simple and portable API for OpenMP that exposes execution events to runtime monitoring tools used for performance measurement and debugging. However, OpenMP is not simply a library that can be instrumented through routine interposition techniques. The transformation of OpenMP directives and runtime routines into a parallel program by an OpenMP compiler poses interesting instrumentation and monitoring challenges.

First, it is necessary to expose the OpenMP execution dynamics to the performance measurement system. However, because the OpenMP programming API merely "directs" parallelization and requires a compiler and runtime system to make the parallel operation explicit, the task of tracking execution dynamics requires instrumentation that is closely coupled with OpenMP directive processing. Second, OpenMP directives implicitly define the parallel execution model that the performance measurement system needs to observe. To correlate execution dynamics with this model, its semantics must be represented in the execution events. This requires the performance API to expose information about the dynamic context in which an execution event occurred. Lastly, there is a natural desire in an OpenMP programming environment to have other kinds of instrumentation (like user-level subroutines) be accomplished in a manner consistent with OpenMP. This suggests that a performance monitoring API should support a mechanism to instrument user-specified, arbitrary regions, including routines.

In the face of these challenges, both OMPI and POMP sought to define a performance monitoring interface with the following objectives:

- Allow application programmers to specify and control instrumentation consistent with the use of OpenMP.
- Allow alternative instrumentation methods, including source-level, compile-time, and run-time instrumentation.
- Do not constrain the implementation of OpenMP compilers or runtime systems.
- Allow alternative implementations of the monitoring interface (i.e., monitoring libraries), including for profile-based and trace-based performance measurement.
- Do not preclude combining OpenMP performance monitoring with other interfaces, in particular, PMPI.
- Make the interface portable across different platforms and usable for different performance tools.

tiative RTS – Parallel System Performance"

- Do not constrain efficient implementations of monitoring libraries.

Since programmers are responsible for inserting OpenMP directives and calling OpenMP runtime routines in their application, it is reasonable to expect the performance monitoring API to be applied manually at the source level. Even when source-based directive rewriting tools (e.g., Opari [4]) or OpenMP compilers perform instrumentation automatically, programmers will still want to have control over instrumentation level and scope, and manual instrumentation will continue to be necessary for user-defined events. New instrumentation directives are useful for this purpose. Thus, because the performance monitoring interface will be seen and used by programmers at the source level, it is important that it is defined consistently with respect to the OpenMP programming model and API.

From the perspective of a tool implementing the performance monitoring interface (e.g., a performance measurement system), the interface must provide all the necessary event and context information to do its job. However, it is important that no specific tool can dictate how the interface is defined and how it operates. The design of the performance monitoring API must support different intended usages and has to avoid assumptions that would limit its application. At the same time, issues of implementation efficiency must be considered.

Achieving portability in the interface requires care in both how the API will function with the different languages OpenMP supports as well as how the API will be implemented in OpenMP compilers and on different execution platforms. Language concerns relate mainly to the calling interface structure and parameters, but also include the extension of the OpenMP directive set for performance monitoring. Portability with respect to OpenMP compilers is primarily an issue of adoption of the performance monitoring API in the compilers. Platform portability of the API deals with issues of getting the interface to function properly with different compilers and performance tools in different system environments.

3 Proposed Specification

The following list covers the topics which were discussed within the involved groups in order to define a common OpenMP monitoring API. We believe that this list covers all necessary issues. It tries to summarize the discussions and to document the changes/differences in reference to the original proposals [4, 9, 10]. Topics where we could not reach an agreement or we ran out of time to define a coherent proposal are marked as ”open issues”.

3.1 Event Definition

This section discusses the OpenMP execution events relevant for performance monitoring which are potential points in the execution where a POMP API call could be made. It is basically a super-set of the events defined in the original proposals. For an exact definition of the events see Appendix C.

The naming of the events follows the following schema: enter/exit events mark the event locations immediately before and after the execution of an OpenMP construct or function. begin/end events mark the begin and end of the function body or of the structured block controlled by the construct. To avoid high overhead for wrapping “small” constructs with a matching pair of API calls, we propose to define a set of *single events* (XXX_event) as an user selectable alternative to the pair of enter/exit events (indicated by [...]).

Events related to OpenMP Constructs and Directives:

Parallel_enter, Parallel_exit, Parallel_begin, Parallel_end,
Loop_enter, Loop_exit, Loop_chunk_begin, Loop_chunk_end, Loop_iter_begin, Loop_iter_end, [Loop_iter_event],
Workshare_enter, Workshare_exit, Workshare_begin, Workshare_end
Sections_enter, Sections_exit, Section_begin, Section_end,
Critical_enter, Critical_exit, Critical_begin, Critical_end,
Single_enter, Single_exit, Single_begin, Single_end,
Barrier_enter, Barrier_exit, Implicit_barrier_enter, Implicit_barrier_exit,
Master_begin, Master_end,
Ordered_enter, Ordered_exit, Ordered_begin, Ordered_end,
Atomic_event, Flush_event

Events related to OpenMP API calls:

Set_lock_enter, Set_lock_exit, [Set_lock_event],
Set_nest_lock_enter, Set_nest_lock_exit, [Set_nest_lock_event],
Init_lock_event, Unset_lock_event, Test_lock_event, Destroy_lock_event,
Init_nest_lock_event, Unset_nest_lock_event, Test_nest_lock_event, Destroy_nest_lock_event,
Set_num_threads_event, Get_num_threads_event, Get_max_threads_event, Get_thread_num_event,
Get_num_procs_event, In_parallel_event, Set_dynamic_event, Get_dynamic_event,
Set_nested_event, Get_nested_event

Events related to User Function and Regions:

Function_enter, Function_exit, Function_begin, Function_end, [Function_event],
User_region_begin, User_region_end, [User_event]

3.2 Context Information Passing

This section describes how context information is associated with the POMP events and how this information is made available by the *instrumentor* to the POMP monitoring API. Please note that by instrumentor we mean a program (e.g. compiler, source-to-source or binary translator) or person (i.e. application developer) which or who inserts calls to the POMP API into the application code.

The context information actually is the combination of three parts:

1. **compile time information:** context information known at compile / instrumentation time (e.g., source code information but also "fixed" attributes of OpenMP constructs like the scheduling strategy). Needs to be passed by the instrumentor to the POMP library API calls.
2. **run time information:** context information only known at run time (e.g., number of threads). The instrumentor has to arrange that this is passed to the POMP library at run time as parameters.
3. **"library" data:** data/information associated with the events by a POMP library (e.g., performance data like counters or library identifiers). This data is allocated and initialized by the POMP library but needs help of the instrumentor to provide access to the library data for all events corresponding to the same OpenMP construct (e.g., enter/exit/begin/end).

The original POMP proposal [4] stored the compile time context information in a standardized fixed data structure in static memory. This descriptor also provides a zero-initialized library handle. The descriptor and the run time context information is passed to the POMP event routines. A POMP library can then construct the library data structures inside the POMP event calls when called for the first time using the compile time and the run time information. This approach has two main drawbacks: First, it is very difficult to define a fixed data layout which is portable across different programming languages. Second, the initialization of the library data inside the POMP event routines requires synchronization, as these routines are possibly called inside parallel regions, which can be a source of large overhead.

The INTONE project follows a different approach: The compile time information is passed as arguments to *context constructor* API routines, which inside the POMP library setup the library data structures (using the passed compile time information) and bind them to a returned *handle*. This approach nicely separates construction and usage of context information in different routines. Passing compile time information as arguments avoids the portability problems of the POMP approach. However, as the context constructor routines can be called inside parallel regions, they still require synchronization.

Based on the experiences gained with our existing implementations, we therefore propose the following procedure for passing context information:

- Compile time context (CTC) information is encoded in a string for maximum portability. Arguments to POMP event routines are the CTC string, run time context information, and a library handle. Normally, the instrumentor passes a valid CTC string plus a zero-initialized library handle to the POMP event routines. In this case, the POMP event routine can initialize the library handle when called for the first time (i.e., if library

handle is zero). As already described, this requires synchronization so all threads get the same, unique handle. The POMP library implementor is responsible for correct, efficient, and portable synchronization.

- If the instrumentor can arrange to call separate register/define handle calls (`POMP_Get_handle`) at program startup (or in other serial phases of the program), it would pass the CTC string to these routines which return a new *pre-initialized* library handle. The POMP event routine then gets passed this pre-initialized library handle, and its CTC string argument can be invalid (better: NULL). This approach minimizes/avoids synchronization but is not always applicable. Because of the performance benefits instrumentors are strongly suggested to use it wherever possible.

  An example instrumentation is shown in Appendix A. The optimized scheme is actually quite often applicable: Binary re-writing instrumentors can do this easily. Source-to-source translators or compilers can use the procedure described in Appendix B.

- When the library handle is NULL, the lifetime of the CTC string must be at least the duration of the POMP API call. One of library handle or CTC string must be non-NULL at every POMP API call.

Open Issues:

- What context information should be described by the CTC string?

construct	context information
ALL	region type start SCL [Source Code Location (file name + line numbers)] end SCL
plus for parallel regions and workshare constructs	hasFirstPrivate hasLastprivate hasReduction hasNowait hasCopyin
plus for parallel loops	scheduleType hasOrdered
plus for single	hasCopyprivate
plus for critical	name (if defined)
plus for functions and user regions	name group (e.g., class, namespace, or module name)

- Format of the CTC string

  The string could be a list of ”attribute=value” pairs which are separated by a star (`'*'`) character (which is very unlikely to appear in function and file names).

  ```
  "<length>*type=<type>*name=<name>*file=<file>*lines=<start>,<end>*...**"
  ```

  Notice that the string is terminated by an empty field, e.g. a double star ”**”. The length is required, all other fields are optional. The terminating double stars are required. Strictly speaking, the length is extraneous given the termination, but it can be useful to size buffers etc. The length should count only the number of characters between the first star and the last, inclusive, without counting the C termination `\0` or any blank fill characters in Fortran.

  As a consequence, the * character cannot appear inside any of the fields, although we could define an escape sequence for it following the C trigraph convention.

  So the shortest possible string is ”2**”.

3.3 Instrumentation of OpenMP Constructs and Directives

At the defined event locations around OpenMP constructs and directives, the instrumentor inserts calls to POMP monitoring API routines. The POMP event routines are named `POMP_<event>()` where `<event>` is one of the values defined in Section 3.1.

Parameters to the event routines are a library handle and the run time context (RTC) information. The RTC includes at least the OpenMP thread ID. If the event routine is the "first" POMP call associated with an OpenMP construct (typically the enter event), it also gets passed the CTC string as the last argument (for maximum portability). All POMP calls return error codes. So, the typical signatures for POMP event routines look like this:

```
C/C++:     int32 POMP_<event> (POMP_Handle_t* lib_handle,
                               int32 thread_id,
                               ...  /*other RTC parameters*/
                               char[] ctc_string)
Fortran:   INTEGER*4 POMP_<event> (lib_handle,
                                   thread_id,
                                   ...,
                                   ctc_string)
           INTEGER*4          thread_id
           INTEGER*<ptrsize>  lib_handle
           CHARACTER*(*)      ctc_string
```

We propose to use a mixed case naming scheme for the C interface (very much like the one used for MPI) because this is the only way to ensure that the external name of the Fortran and the C version of the POMP API are different. This allows to implement the most efficient versions for each language but still enables a shared (common) implementation if desired. However, this does not agree with the current definition of the OpenMP API routines, which are all lowercase.

The following table lists the RTC information (besides the thread ID) for OpenMP construct events:

Event	RTC parameters	NA value
Parallel_enter	int32 num_threads int32 if_expr_result	-1 -1
Loop_enter	int64 chunk_size int64 init_iter_value int64 final_iter_value int64 incr	-1 0
Loop_chunk_begin	int64 init_iter_value int64 final_iter_value	
Loop_iter_begin Loop_iter_event	int64 iter_value	
Section_begin	int32 section_num	-1

Open Issues:

- Should it be allowed to pass a set of user-specified values (e.g., program variables or hardware counter) to user event functions? The CTC string could provide a list of names. The POMP user event functions would have two additional parameters specifying the number of values and an array of int64 values.
- Need to define error codes for all POMP API functions and document their meaning.

3.4 Instrumentation of OpenMP API Functions

In addition to the OpenMP directives, calls to OpenMP API functions need to be visible to a monitoring tool. It is left unspecified how the instrumentation is done. Possible methods are, for example, defining wrapper routines or

pre-instrumented vendor runtime routines which arrange to call the necessary event routines at the right places.

As the OpenMP run time functions are "small", generating enter and exit events for them would produce a large overhead. Therefore we suggest to only use single events for monitoring OpenMP API functions (see Section 3.1). Since setting and getting locks can be a more costly operation and a monitoring tool might be interested in measuring the waiting time for the lock, we define optional enter and exit events for lock functions.

In general, the naming and arguments are handled like for the POMP event routines for directives. The RTC information is the thread ID plus the omp_lock ID for the OpenMP lock functions. OpenMP test lock calls provide an indicator whether a lock was set. All routines which set or return a value (e.g., `omp_get_dynamic`) have a parameter supplying the value set or returned.

3.5 Instrumentation of Functions and User Regions

Experience shows that it is usually not sufficient to just collect OpenMP related events: the end-user needs a way to relate performance information to user defined functions or even arbitrary structured blocks.

We propose that an instrumentor (typically, the application developer) insert calls to `POMP_Function_XXX()` and `POMP_User_XXX()` POMP calls. Functions are subroutines, functions, procedures, methods, etc. defined by the programming language. The Function_enter/_exit events are for call site instrumentation, while Function_begin/_end events can be used for the instrumentation of a function body. Also, functions typically have multiple entry and exit points. In contrast, user regions can be arbitrary structured blocks. User regions can be nested but are not allowed to overlap.

The user has to pass a zero-initialized library handle, the thread ID, and a valid CTC string directly to the event routine **or** optionally use the `POMP_Get_handle()` calls. If the instrumentor is an automatic tool, providing automatic instrumentation of all or at least a reasonable set of user functions would be a very useful feature. While this optional, it should use the proposed API if provided. Finally, to simplify manual instrumentation, it would be advantageous to allow the definition of user regions through instrumentation directives (see Section 3.9).

Open Issues:

- How to specify (e.g., through additional directives) which user functions should be instrumented (blindly instrumenting every single function will certainly produce unacceptable overhead).
- Is it useful to distinguish between function call site and function body instrumentation (i.e., having separate Function_enter/_exit and Function_begin/_end events)?
- Define a method for the user to pass a set of values to the `POMP_User_region_XXX()` routines.

3.6 Monitoring Library Control

In addition to POMP functions for defining and recording of execution events, we propose functions for initialization/finalization (to be inserted by the instrumentor at appropriate places) and to signal start/stop of monitoring activities to the POMP library (inserted by the user).

```
POMP_Init(), POMP_Finalize(), POMP_On(), POMP_Off()
```

If possible, the instrumentor should call `POMP_Init` at program startup and `POMP_Finalize` just before program termination on the master thread only. However, a conforming POMP monitoring library must be able to auto-initialize and auto-finalize.

Open Issues:

- Should the places where `POMP_On()` and `POMP_Off()` can be called be restricted? For example, what shall happen to event pairs that are not properly matched at the time of an `POMP_Off()` call?

3.7 Conditional Monitoring Code

We also propose to support user source code lines to be compiled conditionally if POMP instrumentation is requested. If the OpenMP compiler or POMP transformation tool supports a macro preprocessor (e.g., `cpp` for C, C++, and sometimes Fortran), it must define the symbol `_POMP` to be used for conditional compilation. Following OpenMP standard conventions, this symbol is defined to have the decimal value `YYYYMM` where `YYYY` and `MM` are the year and month designations of the version of the POMP API that the implementation supports. This allows users to define and use application-, user-, or site-specific extensions to POMP by writing:

```
#ifdef _POMP
   ... arbitrary user code ...
#endif
```

The `!P$` sentinel can be used for conditional compilation in Fortran compilation or transformation systems. In addition, `CP$` and `*P$` sentinels are accepted only when instrumenting Fortran fixed source form. During POMP instrumentation, these sentinels are replaced by three spaces, and the rest of the line is treated as a normal Fortran line. These sentinels also have to comply with the specifications defined in [6].

This is an optional feature as binary / library instrumentation cannot support this. But if a platform does support this feature, it should do it as described above.

3.8 Instrumentation Control

We think it is important to allow the user to specify the amount of monitoring. There needs to be a way to specify to what extent a user program is instrumented and also a way to activate/deactivate events at runtime. Ideally, the specification for both uses is based on the same principles / terms.

Open Issues:

- There is no complete proposal for this feature yet. A starting point could be the following: events can be categorized into the following groups. For each group, a level of instrumentation/activation is defined:

Group	Constructs	Levels
Parallel	Parallel	none, EnterExit, BeginEnd
Loop	Do/For	none, EnterExit, (Chunks *or* IterEvent *or* IterBeginEnd)
Workshare	Section, Workshare, Single	none, EnterExit, BeginEnd
Sync	Critical, Ordered	none, EnterExit, BeginEnd
	Barrier	none, EnterExit
	Master	none, BeginEnd
	Atomic, Flush	none, Event
User	Function, User regions	none, (Event *or* BeginEnd)
Runtime	OpenMP run-time library routines	none, (Event *or* EnterExit)

  Levels separated by "*or*" are alternatives and don't include instrumentation for each other. Levels separated by "," do include instrumentation for previous (lower) levels, e.g., for group "Parallel" this means "none" would specify to monitor no events at all, "EnterExit" specifies to monitor enter and exit events only, and "BeginEnd" would result in the monitoring of the enter, exit, begin, and end events for parallel regions.

  The activation/deactivation can be specified through environment variables (`POMP_<group>`) and through corresponding POMP API functions.

- While the above proposal works nicely for activation/deactivation of events at run time, it is a pretty clumsy way to specify the amount of instrumentation for an automatic instrumentor. A more natural way would be to

define a new directive that lets the user specify the extent of instrumentation (as most of OpenMP is specified by directives).

```
!$POMP INST group level [file]
```

This directive applies only to the function body or the static extend of the OpenMP construct immediately following the directive unless ”file” is specified in which case it applies to the rest of the compilation unit.

However, it is very controversial whether defining new directives is a good idea and acceptable to the OpenMP ARB. A possible solution would be to mark them as optional, i.e., a POMP compliant instrumentor is not required to implement them but if it supports them, it has to be done in the way described above.

Disadvantages:

- Directives require new compiler implementation work for industrial compilers. The rest of this proposal can be handled by minor adjustments to the existing handling of OpenMP directives and new OpenMP runtime libraries.
- Controlling instrumentation through directives might be too confusing / complex. Is it clear to the user which OpenMP constructs are exactly affected (especially if they can be arbitrarly nested)?
- Directives require recompilation to switch from collecting performance data to not collecting, or the other way around. Ideally, only re-linking or a new binary instrumentation should be required.

Advantages:

- The core of OpenMP is defined by directives. So, additional features should use the same mechanism.
- The instrumentation is specified inside the program source text to which it applies to. No separate file or specification is necessary requiring extra functionality/tools to manage it.
- Directives are the most effective way to allow instrumentation control for selective parts of an application. Using environment variables or command line switches only allow to control the instrumentation of a compilation unit (or even the program) as a whole.
- In some cases, it is desirable to be able to completely remove the instrumentation overhead. A programmer does not want to have to live with extra instrumentation calls in the code if instrumentation is turned off (especially if it is in inner loops).

3.9 Directives for User Defined Events

By combining the features from Sections 3.5 and 3.7, a user can make arbitrary structured blocks visible to the monitoring tool. However, since the user has to arrange the correct definition of things like CTC strings and library handles, the scheme seems unnecessarly complex.

The following new directives would greatly simplify the specification / generation of user defined events:

```
!$POMP USERREGION START ( <name> [: <variable-or-value-list>] )

!$POMP USERREGION STOP  ( <name> [: <variable-or-value-list>] )

!$POMP USERREGION EVENT ( <name> [: <variable-or-value-list>] )
```

These would be translated into the necessary CTC string and library handle generation and event routines by the instrumentor. Of course, the remarks about new directives stated in the last section also apply here.

3.10 Monitoring of Applications which use nested parallelism

Currently, it is not completely clear how to make sure the proposed POMP monitoring API is able to correctly monitor OpenMP applications which use nested parallelism. We need some concept of OpenMP team here in order to gather performance information about threads within a specific team. This probably needs to be defined by the OpenMP standard before we tackle it here.

4 Conclusions and Future Work

The definition of a robust, open, and implementable performance monitoring interface for OpenMP is not an easy task. The efforts among the groups represented in this paper point to the difficult technical issues which must be resolved, as well as to the differences of opinion that can arise, for example, from experiences with performance tool users. Nevertheless, substantial progress was achieved in the ten areas reported above, and our work will provide a solid context for future discussion.

With relatively minor open issues remaining, we have reached agreement on event definition, mechanisms for context information passing, instrumentation of OpenMP directives and API functions, instrumentation of user functions/regions, monitoring library code, and conditional monitoring code. The issue of context information has been particularly interesting, since from a quite contentious debate a solution that is clearly superior to both original schemes has emerged. The areas of new directives for instrumentation control and for user defined events and nested parallelism are more challenging and we did not attempt to resolve them here.

Although prototype tools can be developed based on the specification we proposed, which was done before for the INTONE and POMP work, the long-term goal is to have an API approved by the OpenMP Architecture Review Board (ARB) and implemented in real compiler systems. The next steps in the process are to work within the ARB Tools sub-committee to define a monitoring interface specification. To aid in this endeavor, we offer our current outline of a specification as initial input.

References

[1] J. Hoeflinger et al., "An Integrated Performance Visualizer for MPI/OpenMP Programs," *Workshop on OpenMP Applications and Tools (WOMPAT 2001)*, July, 2001.

[2] INTONE (INnovative Tools for OpenMP for Non-Experts). `http://www.cepba.upc.es/intone/`.

[3] A. Malony, S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," *Proc. 3rd Workshop on Distributed and Parallel Systems*, DAPSYS 2000, (Eds. G. Kotsis, P. Kacsuk), pp. 37–46, 2000.

[4] B. Mohr, A. Malony, S. Shende, F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP," *The Journal of Supercomputing*, 23, 105–128, 2002.

[5] OpenMP Forum, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," October, 1997. http://www.openmp.org.

[6] OpenMP Forum, "OpenMP Fortran Application Program Interface, Version 2.0," November 2000. Sections 2.1.3.1 and 2.1.3.2

[7] Pallas GmbH, "VAMPIR: Visualization and Analysis of MPI Resources". `http://www.pallas.de/pages/vampir.htm`.

[8] F. Wolf, B. Mohr, "Automatic Performance Analysis of SMP Cluster Applications," Tech. Rep. IB 2001-05, Research Centre Jülich, 2001.

[9] Pallas GmbH, "Public OpenMP Instrumentation Interface Specification," INTONE – Innovative OpenMP Tools for Non-Experts, Version 1.7, 2002.

[10] B. Kuhn, A. Malony, B. Mohr, S. Shende, "A Performance Tool Interface for OpenMP," Report for Accelerated Strategic Computing Initiative (ASCI), ASCI Path Forward program, Ultrascale Tools Initiative, RTS - Parallel System Performance, submitted by KAI Software Labs, A Division of Intel America, Inc., August 28, 2001.

[11] Message Passing Interface Forum, "MPI: A MessagePassing Interface Standard," *International Journal of Supercomputer Applications*, Vol. 8, 1994. Special issue on MPI.

[12] European Center for Parallelism of Barcelona (CEPBA). `http://www.cepba.upc.es`.

A Instrumentation Example

Consider the following example C OpenMP program.

Original Example Program `demo.c`

```
 1:   #include <stdio.h>
 2:   #include <omp.h>
 3:
 4:   int main() {
 5:      int id;
 6:
 7:      #pragma omp parallel \
 8:                  private(id)
 9:      {
10:         id = omp_get_thread_num();
11:         printf("hello from %d\n", id);
12:      }
13:   }
```

The following figure shows how the un-optimized (standard) instrumentation of the example program with POMP API calls could look like. New inserted lines are marked with `***` at the left side. Besides inserting `POMP_Init()` and `POMP_Finalize()` calls at the beginning and end of the main program, the parallel region is instrumented with enter, exit, begin, and end events.

Example Standard Instrumentation of `demo.c`

```
 1:   #include <stdio.h>
 2:   #include <omp.h>
***   #include "pomplib.h"
 3:
 4:   int main() {
 5:      int id;
***      POMP_Init();
 6:
***      {  POMP_Handle_t pomp_hd1 = 0;
***         int32 pomp_tid = omp_get_thread_num();
***         POMP_Parallel_enter(&pomp_hd1, pomp_tid, -1, 1,
***            "51*type=pregion*file=demo.c*slines=7,8*elines=12,12**");
 7:      #pragma omp parallel \
 8:                  private(id)
 9:      {
***         int32 pomp_tid = omp_get_thread_num();
***         POMP_Parallel_begin(pomp_hd1, pomp_tid);
10:         id = omp_get_thread_num();
11:         printf("hello from %d\n", id);
***         POMP_Parallel_end(pomp_hd1, pomp_tid);
12:      }
***         POMP_Parallel_exit(pomp_hd1, pomp_tid);
***      }
***      POMP_Finalize();
13:   }
```

In addition, there are Implicit_barrier_enter, Implicit_barrier_exit, and Get_thread_num_event events, probably initiated by the OpenMP run time system. All POMP event routines get passed a POMP library handle and the thread ID. The first event of the parallel region (Parallel_enter) gets passed a CTC string describing the parallel region. It is also ensured it gets passed a zero-initialized library handle.

The next figure shows the optimized version of the instrumentation of the same program. Library handles are initialized at program startup using the `POMP_Get_handle()` function which gets passed the CTC string and returns the initialized library in the first parameter. This saves unnecessary synchronization inside the POMP event routines.

Example Optimized Instrumentation of `demo.c`

```
 1:   #include <stdio.h>
 2:   #include <omp.h>
***   #include "pomplib.h"
 3:
 4:   int main() {
 5:      int id;
***      POMP_Handle_t pomp_hd1 = 0;
***      POMP_Init();
***      POMP_Get_handle(&pomp_hd1,
***         "51*type=pregion*file=demo.c*slines=7,8*elines=12,12**");
 6:
***      {  int32 pomp_tid = omp_get_thread_num();
***         POMP_Parallel_enter(&pomp_hd1, pomp_tid, -1, 1, NULL);
 7:      #pragma omp parallel \
 8:                  private(id)
 9:      {
***         int32 pomp_tid = omp_get_thread_num();
***         POMP_Parallel_begin(pomp_hd1, pomp_tid);
10:         id = omp_get_thread_num();
11:         printf("hello from %d\n", id);
***         POMP_Parallel_end(pomp_hd1, pomp_tid);
12:      }
***         POMP_Parallel_exit(pomp_hd1, pomp_tid);
***      }
***      POMP_Finalize();
13:   }
```

Of course this optimization is easy to do in this simple example. Things get more complicated if the main function and the instrumented OpenMP construct are in different compilation units. The next section demonstrates how a source-to-source translator or compiler could implement this alternatives while still maintaining separate compilation.

B Optimized Instrumentation for Multiple Compilation Units

This section describes how a source-to-source translator or compiler can implement the optimized instrumentation scheme while still maintaining separate compilation. In general, in the following, ### needs to be replaced by some unique identifier, e.g., a prefix followed by the inode number of the source file.

The described procedure even handles mixed language applications. Another advantages is, that the necessary library handles can always be numbered from 1 to N as they are file-local global variables or in separate modules/common blocks.

For each C/C++ file xxx.c:

1. Add the following as the first line to the file:

   ```
   #include "xxx.c.pomp.inc"
   ```

2. Then do the POMP instrumentation as usual, i.e., inserting

   ```
   POMP_Whatever(pompid#, ...);
   ```

   where needed. "#" is replaced by a number from 1 to N as needed. Of course, t is necessary to store all the compile time context information internally.

3. Then, at the end, generate the `"xxx.c.pomp.inc"` file based on the collected compile time context data like this:

   ```
   #include "pomplib.h"
   static int pompid1;
   ...
   static int pompidN;

   void POMP_Init_###()
      POMP_Get_handle(&pompid1, "ctc string 1");
      ...
      POMP_Get_handle(&pompidN, "ctc string N");
   ```

For each Fortran90 file xxx.f90:

1. Insert the following as the first line of each program, function and subroutine. Finding this line should be doable even in fuzzy parsers as one only need to look for e.g., `"subroutine XXX(...)"`.

   ```
   use POMP_###
   ```

2. Then do the POMP instrumentation as usual, i.e., inserting

   ```
   call POMP_Whatever(pompid#, ...);
   ```

   where needed.

3. Then, at the end, generate a `"POMP_###.f90"` file like this:

   ```
   module POMP_###
      integer, public :: pompid1
      ...
      integer, public :: pompidN
   end module POMP_###
   ```

4. At the end of this file, also add

```
subroutine POMP_Init_###()
   use POMP_###
   call POMP_Get_handle(pompid1, "ctc string 1")
   ...
   call POMP_Get_handle(pompidN, "ctc string N")
end subroutine POMP_Init_###
```

5. Of course, in addition and before the modified f90 source, the `POMP_###.f90` file has to be compiled.

For each Fortran77 file xxx.f:

Here, basically do the same as for Fortran 90, just that instead of a module `POMP_###`, use a common block `/POMP_###/` and then a "include" statement to insert this in every function.

1. Insert the following as the first line of each program, function and subroutine.

```
include 'xxx.f.pomp.inc'
```

2. Then do the POMP instrumentation as usual.
3. At the end of the file, add

```
subroutine POMP_Init_###()
  include 'xxx.f.pomp.inc'
  call POMP_Get_handle(pompid1, "ctc string 1")
  ...
  call POMP_Get_handle(pompidN, "ctc string N")
end subroutine POMP_Init_###
```

4. Then, at the end, generate a `"xxx.f.pomp.inc"` file like this:

```
  integer pompid1
  ...
  integer pompidN
  common /POMP_###/ pompid1, ..., pompidN
```

Linking

1. A modified linker or special pre-linker then scans all passed-in object files and libraries (e.g., using `nm`) for functions named `POMP_Init_*`. Then, a temporary file "pomp.init.c" is generated with the following contents:

```
extern void POMP_Init_###1();
...
extern void POMP_Init_###n();

void POMP_Init_Handles()
   POMP_Init_###1();
   ...
   POMP_Init_###n();
```

   This gets compiled and it is linked in addition to the POMP library to the program.

2. Finally, `POMP_Init()` calls `POMP_Init_Handles()`.

C Performance Event Model for OpenMP

Our goal for a performance event model for OpenMP is to define a set of *performance events* that will a) represent completely the semantics of OpenMP parallel execution and b) allow general program-level and user-defined events to be specified. The performance interface is based on these performance events and is associated with the execution of OpenMP constructs or function calls/returns.

For OpenMP constructs, there are typically four standard events:

- **enter**: execution of an OpenMP construct is starting
- **begin**: execution of the body of an OpenMP construct is starting
- **end**: execution of the body of an OpenMP construct has finished
- **exit**: execution of an OpenMP construct has finished

The events are assigned to *event levels* according to the level of detail or granularity they represent for a particular construct. Enter/exit events provide the boundary of the overall execution of a given construct. The begin/end events provide the boundaries of the execution of the user code within the constructs. The exact level assignment depends also on the OpenMP construct involved. Levels are useful to control the detail of event instrumentation.

The following tables describe the performance events and routines that will be called. The tables are grouped according to the event groups of Section 3.8. The event is described in the first column along with a prototype of the POMP function that will called when the event occurs. This prototype is for C programs. The Fortran prototype may be obtained as described in Section 3.3. The second column describes the event. The third column gives the group to which the event belongs and the fourth column shows the event levels under which the function should be called.

OpenMP event / POMP API	Description - called when ...	Group is	and Level is
`Parallel` *Enter*	Event occurs only in the master thread immediately before a team of threads is formed to start parallel execution of the `parallel` region.	Parallel	EnterExit or BeginEnd
`POMP_Parallel_enter( POMP_Handle_t* handle, int32 thread_id, int32 num_threads, int32 if_expr_result, char ctc[] )`			
`Parallel` *Begin*	Event occurs in a team thread immediately before that thread begins its execution in the `parallel` region	Parallel	BeginEnd
`POMP_Parallel_begin( POMP_Handle_t handle, int32 thread_id )`			
`Parallel` *End*	Event occurs in a team thread immediately after that thread ends its execution in the `parallel` region	Parallel	BeginEnd
`POMP_Parallel_end( POMP_Handle_t handle, int32 thread_id )`			
`Parallel` *Exit*	Event occurs only in the master thread, immediately after all team threads finish parallel execution of the `parallel` region and the master thread leaves the join barrier	Parallel	EnterExit or BeginEnd
`POMP_Parallel_exit( POMP_Handle_t handle, int32 thread_id )`			
`for/do` *Enter*	Event occurs in a thread immediately before that thread enters the OpenMP parallel loop	Loop	EnterExit or Chunks or IterEvent or IterBeginEnd
`POMP_Loop_enter( POMP_Handle_t* handle, int32 thread_id, int64 chunk_size, int64 init_iter, int64 final_iter, int64 incr, char ctc[] )`			
`for/do` *Chunk Begin*	Event occurs in a thread immediately before that thread enters the chunk of iterations to execute	Loop	Chunks
`POMP_Loop_chunk_begin( POMP_Handle_t handle, int32 thread_id, int64 init_iter, int64 final_iter )`			
`for/do` *Iter Begin*	Event occurs in a thread immediately before that thread enters the indicated iteration	Loop	IterBeginEnd
`POMP_Loop_iter_begin( POMP_Handle_t handle, int32 thread_id, int64 iter )`			
`for/do` *Iter Event*	Event occurs in a thread immediately before that thread enters the indicated iteration	Loop	IterEvent
`POMP_Loop_iter_event( POMP_Handle_t handle, int32 thread_id, int64 iter )`			
`for/do` *Iter End*	Event occurs in a thread immediately after that thread completes the indicated iteration	Loop	IterBeginEnd
`POMP_Loop_iter_end( POMP_Handle_t handle, int32 thread_id )`			
`for/do` *Chunk End*	Event occurs in a thread immediately after that thread completes the chunk of iterations to execute	Loop	Chunks
`POMP_Loop_chunk_end( POMP_Handle_t handle, int32 thread_id )`			
`for/do` *Exit*	Event occurs in a thread immediately after that thread exits the OpenMP parallel loop construct	Loop	EnterExit or Chunks or IterEvent or IterBeginEnd
`POMP_Loop_exit( POMP_Handle_t handle, int32 thread_id )`			

Table 1: **Parallel and Loop Event Groups**

OpenMP event / POMP API	Description - called when ...	Group is	and Level is
`Workshare` *Enter*	Event occurs in a thread immediately before that thread enters the OpenMP `workshare` construct	Workshare	EnterExit or BeginEnd
`POMP_Workshare_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Workshare` *Begin*	Event occurs in a thread immediately before that thread begins the OpenMP `workshare` structured block	Workshare	BeginEnd
`POMP_Workshare_begin( POMP_Handle_t handle, int32 thread_id )`			
`Workshare` *End*	Event occurs in a thread immediately after that thread ends the OpenMP `workshare` structured block	Workshare	BeginEnd
`POMP_Workshare_end( POMP_Handle_t handle, int32 thread_id )`			
`Workshare` *Exit*	Event occurs in a thread immediately after that thread exits the OpenMP `workshare` construct	Workshare	EnterExit or BeginEnd
`POMP_Workshare_exit( POMP_Handle_t handle, int32 thread_id )`			
`Sections` *Enter*	Event occurs in a thread immediately before that thread enters the OpenMP `sections` construct	Workshare	EnterExit or BeginEnd
`POMP_Sections_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Section` *Begin*	Event occurs in a thread immediately before that thread begins its execution of the `section` structured block	Workshare	BeginEnd
`POMP_Section_begin( POMP_Handle_t handle, int32 section_num, int32 thread_id )`			
`Section` *End*	Event occurs in a thread immediately after that thread ends its execution of the `section` structured block	Workshare	BeginEnd
`POMP_Section_end( POMP_Handle_t handle, int32 thread_id )`			
`Sections` *Exit*	Event occurs in a thread immediately after that thread exits the OpenMP `sections` construct	Workshare	EnterExit or BeginEnd
`POMP_Sections_exit( POMP_Handle_t handle, int32 thread_id )`			
`Single` *Enter*	Event occurs in a thread immediately before that thread enters OpenMP `single` construct	Workshare	EnterExit or BeginEnd
`POMP_Single_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Single` *Begin*	Event occurs in a thread immediately after that thread has obtained exclusive access to begin its execution of the `single` structured block	Workshare	BeginEnd
`POMP_Single_begin( POMP_Handle_t handle, int32 thread_id )`			
`Single` *End*	Event occurs in a thread immediately after that thread ends its execution of the `single` structured block	Workshare	BeginEnd
`POMP_Single_end( POMP_Handle_t handle, int32 thread_id)`			
`Single` *Exit*	Event occurs in a thread immediately after that thread exits the OpenMP `single` construct	Workshare	EnterExit or BeginEnd
`POMP_Single_exit( POMP_Handle_t handle, int32 thread_id )`			

Table 2: **Workshare Event Group**

OpenMP event / POMP API	**Description - called when ...**	**Group is**	**and Level is**
`Critical` *Enter*	Event occurs in a thread immediately before that thread enters the OpenMP `critical` construct	Sync	EnterExit or BeginEnd
`POMP_Critical_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Critical` *Begin*	Event occurs in a thread immediately after that thread has obtained exclusive access to begin its execution of the `critical` structured block	Sync	BeginEnd
`POMP_Critical_begin( POMP_Handle_t handle, int32 thread_id )`			
`Critical` *End*	Event occurs in a thread immediately after that thread ends its execution of the `critical` structured block	Sync	BeginEnd
`POMP_Critical_end( POMP_Handle_t handle, int32 thread_id )`			
`Critical` *Exit*	Event occurs in a thread immediately after that thread exits the OpenMP `critical` construct	Sync	EnterExit or BeginEnd
`POMP_Critical_exit( POMP_Handle_t handle, int32 thread_id )`			
`Master` *Begin*	Event occurs only in the master thread immediately before it begins its execution of the `master` structured block	Sync	BeginEnd
`POMP_Master_begin( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Master` *End*	Event occurs only in the master thread immediately after it ends its execution of the `master` structured block	Sync	BeginEnd
`POMP_Master_end( POMP_Handle_t handle, int32 thread_id )`			
`Barrier` *Enter*	Event occurs in a thread immediately before that thread enters the explicit OpenMP `barrier` construct	Sync	EnterExit
`POMP_Barrier_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Barrier` *Exit*	Event occurs in a thread immediately after that thread exits the explicit OpenMP `barrier` construct	Sync	EnterExit
`POMP_Barrier_exit( POMP_Handle_t handle, int32 thread_id )`			
Implicit Barrier Enter	Event occurs in a thread immediately before that thread enters an implicit OpenMP barrier occurring in `parallel`, `do/for`, `workshare`, `sections`, and `single` constructs	Sync	EnterExit
`POMP_Implicit_barrier_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
Implicit Barrier Exit	Event occurs in a thread immediately after that thread exits the implicit OpenMP barrier	Sync	EnterExit
`POMP_Implicit_barrier_exit( POMP_Handle_t handle, int32 thread_id )`			
`Flush` *Event*	Event occurs in a thread immediately after the `flush` operation completes for that thread	Sync	Event
`POMP_Flush_event( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Atomic` *Event*	Event occurs in a thread immediately after that thread exits the OpenMP `atomic` region	Sync	Event
`POMP_Atomic_event( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			

Table 3: **Sync Group Events for** `critical`, `atomic`, `master`, `barrier` **and** `flush`

OpenMP event / POMP API	Description - called when ...	Group is	and Level is
`Ordered` *Enter*	Event occurs in a thread immediately before that thread enters OpenMP `ordered` construct	Sync	EnterExit or BeginEnd
`POMP_Ordered_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
`Ordered` *Begin*	Event occurs in a thread immediately after that thread has entered the code in the `ordered` structured block	Sync	BeginEnd
`POMP_Ordered_begin( POMP_Handle_t handle, int32 thread_id )`			
`Ordered` *End*	Event occurs in a thread immediately after that thread ends execution of the code in the `ordered` structured block	Sync	BeginEnd
`POMP_Ordered_end( POMP_Handle_t handle, int32 thread_id )`			
`Ordered` *Exit*	Event occurs in a thread immediately after that thread exits the OpenMP `ordered` construct	Sync	EnterExit or BeginEnd
`POMP_Ordered_exit( POMP_Handle_t handle, int32 thread_id )`			

Table 4: **Sync Group Events for** `ordered`

OpenMP event / POMP API	Description - called when ...	Group is	and Level is
Function Enter	Event occurs in a thread immediately before that thread executes a call to a function	User	EnterExit
`POMP_Function_enter( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
Function Begin	Event occurs in a thread immediately before that thread begins execution of the function body, after the function is called	User	BeginEnd
`POMP_Function_begin( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
Function End	Event occurs in a thread immediately after that thread ends execution of the function body, before the function returns	User	BeginEnd
`POMP_Function_end( POMP_Handle_t handle, int32 thread_id )`			
Function Exit	Event occurs in a thread immediately after the thread returns from a called function	User	ExitEnter
`POMP_Function_exit( POMP_Handle_t handle, int32 thread_id )`			
Function Event	For a very small function, event occurs on entry to the function	User	Event
`POMP_Function_event( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
User Region Begin	Event occurs in a thread immediately before that thread begins execution of the user-defined structured block	User	BeginEnd
`POMP_User_region_begin( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			
User Region End	Event occurs in a thread immediately after that thread ends execution of the user-defined structured block	User	BeginEnd
`POMP_User_region_end( POMP_Handle_t handle, int32 thread_id)`			
User Event	Signal a user-defined event	User	Event
`POMP_User_event( POMP_Handle_t* handle, int32 thread_id, char ctc[] )`			

Table 5: **User Event Group**

OpenMP event / POMP API	**Description - called when ...**	**Group is**	**and Level is**
`omp_init_lock` *Event*	Event occurs in a thread when the `omp_init_lock()` is called by that thread	Runtime	EnterExit or Event
`POMP_Init_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_lock_t lock_id, char ctc[] )`			
`omp_destroy_lock` *Event*	Event occurs in a thread when the `omp_destroy_lock()` is called by that thread	Runtime	EnterExit or Event
`POMP_Destroy_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_lock_t lock_id, char ctc[] )`			
`omp_set_lock` *Enter*	Event occurs when `omp_set_lock()` is called by the thread	Runtime	EnterExit
`POMP_Set_lock_enter( POMP_Handle_t* handle, int32 thread_id, omp_lock_t lock_id, char ctc[] )`			
`omp_set_lock` *Exit*	Event occurs in a thread immediately after the `omp_set_lock()` completes	Runtime	EnterExit
`POMP_Set_lock_exit( POMP_Handle_t handle, int32 thread_id, omp_lock_t lock_id )`			
`omp_set_lock` *Event*	Event occurs in a thread immediately after the `omp_set_lock()` completes	Runtime	Event
`POMP_Set_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_lock_t lock_id, char ctc[] )`			
`omp_unset_lock` *Event*	Event occurs in a thread when the `omp_unset_lock()` is called by that thread	Runtime	EnterExit or Event
`POMP_Unset_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_lock_t lock_id, char ctc[] )`			
`omp_test_lock` *Event*	Event occurs in a thread immediately after `omp_test_lock()` completes	Runtime	EnterExit or Event
`POMP_Test_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_lock_t lock_id, int32 was_lock_set, char ctc[] )`			

Table 6: **Runtime Group Events for Lock API functions**

OpenMP event / POMP API	**Description - called when ...**	**Group is**	**and Level is**
`omp_init_nest_lock` *Event*	Event occurs in a thread when the `omp_init_nest_lock()` is called by that thread	Runtime	EnterExit or Event
`POMP_Init_nest_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_nest_lock_t lock_id, char ctc[] )`			
`omp_destroy_nest_lock` *Event*	Event occurs in a thread when the `omp_destroy_nest_lock()` is called by that thread	Runtime	EnterExit or Event
`POMP_Destroy_nest_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_nest_lock_t lock_id, char ctc[] )`			
`omp_set_nest_lock` *Enter*	Event occurs when `omp_set_nest_lock()` is called by that thread	Runtime	EnterExit
`POMP_Set_nest_lock_enter( POMP_Handle_t* handle, int32 thread_id, omp_nest_lock_t lock_id, char ctc[] )`			
`omp_set_nest_lock` *Exit*	Event occurs in a thread immediately after the `omp_set_nest_lock()` completes	Runtime	EnterExit
`POMP_Set_nest_lock_exit( POMP_Handle_t handle, int32 thread_id, omp_nest_lock_t lock_id )`			
`omp_set_nest_lock` *Event*	Event occurs in a thread immediately after the `omp_set_nest_lock()` completes	Runtime	Event
`POMP_Set_nest_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_nest_lock_t lock_id, char ctc[] )`			
`omp_unset_nest_lock` *Event*	Event occurs in a thread when the `omp_unset_nest_lock()` is called by that thread	Runtime	EnterExit or Event
`POMP_Unset_nest_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_nest_lock_t lock_id, char ctc[] )`			
`omp_test_nest_lock` *Event*	Event occurs in a thread immediately after the `omp_test_nest_lock()` completes	Runtime	EnterExit or Event
`POMP_Test_nest_lock_event( POMP_Handle_t* handle, int32 thread_id, omp_nest_lock_t lock_id, int32 was_lock_set, char ctc[] )`			

Table 7: **Runtime Group Events for Nested Lock API functions**

OpenMP event / POMP API	**Description - called when ...**	**Group is**	**and Level is**
`omp_set_num_threads` *Event*	Event occurs in a thread when `omp_set_num_threads()` is called by that thread	Runtime	Event
`POMP_Set_num_threads_event( POMP_Handle_t* handle, int32 thread_id, int32 num_threads, char ctc[] )`			
`omp_get_num_threads` *Event*	Event occurs in a thread when `omp_get_num_threads()` is called by that thread	Runtime	Event
`POMP_Get_num_threads_event( POMP_Handle_t* handle, int32 thread_id, int32 num_threads, int32 char ctc[] )`			
`omp_get_max_threads` *Event*	Event occurs in a thread when `omp_get_max_threads()` is called by that thread	Runtime	Event
`POMP_Get_max_threads_event( POMP_Handle_t* handle, int32 thread_id, int32 num_threads, char ctc[] )`			
`omp_get_thread_num` *Event*	Event occurs in a thread when `omp_get_thread_num()` is called by that thread	Runtime	Event
`POMP_Get_thread_num_event( POMP_Handle_t* handle, int32 thread_id, int32 thread_num, char ctc[] )`			
`omp_get_num_procs` *Event*	Event occurs in a thread when `omp_get_num_procs()` is called by that thread	Runtime	Event
`POMP_Get_num_procs_event( POMP_Handle_t* handle, int32 thread_id, int32 num_procs, char ctc[] )`			
`omp_in_parallel` *Event*	Event occurs in a thread when `omp_in_parallel()` is called by that thread	Runtime	Event
`POMP_In_parallel_event( POMP_Handle_t* handle, int32 thread_id, int32 is_parallel, char ctc[] )`			
`omp_set_dynamic` *Event*	Event occurs in a thread when `omp_set_dynamic()` is called by that thread	Runtime	Event
`POMP_Set_dynamic_event( POMP_Handle_t* handle, int32 thread_id, int32 is_dynamic, char ctc[] )`			
`omp_get_dynamic` *Event*	Event occurs in a thread when `omp_get_dynamic()` is called by that thread	Runtime	Event
`POMP_Get_dynamic_event( POMP_Handle_t* handle, int32 thread_id, int32, is_dynamic, char ctc[] )`			
`omp_set_nested` *Event*	Event occurs in a thread when `omp_set_nested()` is called by that thread	Runtime	Event
`POMP_Set_nested_event( POMP_Handle_t* handle, int32 thread_id, int32 is_nested, char ctc[] )`			
`omp_get_nested` *Event*	Event occurs in a thread when `omp_get_nested()` is called by that thread	Runtime	Event
`POMP_Get_nested_event( POMP_Handle_t* handle, int32 thread_id, int32 is_nested, char ctc[] )`			

Table 8: **Runtime Group Events for Miscellaneous API functions**