# Design and Prototype of a Performance Tool Interface for OpenMP

Bernd Mohr[+], Allen D. Malony[*], Sameer Shende[*], and Felix Wolf[+]

[*]Dept. of Computer and Information Science
University of Oregon
{malony,sameer}@cs.uoregon.edu

[+]Research Centre Jülich, ZAM
Jülich, Germany
{b.mohr,f.wolf}@fz-juelich.de

**Abstract.** This paper proposes a performance tools interface for OpenMP, similar in spirit to the MPI profiling interface in its intent to define a clear and portable API that makes OpenMP execution events visible to runtime performance tools. We present our design using a source-level instrumentation approach based on OpenMP directive rewriting. Rules to instrument each directive and their combination are applied to generate calls to the interface consistent with directive semantics and to pass context information (e.g., source code locations) in a portable and efficient way. Our proposed OpenMP performance API further allows user functions and arbitrary code regions to be marked and performance measurement to be controlled using new OpenMP directives. To prototype the proposed OpenMP performance interface, we have developed compatible performance libraries for the EXPERT automatic event trace analyzer [17, 18] and the TAU performance analysis framework [13]. The directive instrumentation transformations we define are implemented in a source-to-source translation tool called OPARI. Application examples are presented for both EXPERT and TAU to show the OpenMP performance interface and OPARI instrumentation tool in operation. When used together with the MPI profiling interface (as the examples also demonstrate), our proposed approach provides a portable and robust solution to performance analysis of OpenMP and mixed-mode (OpenMP + MPI) applications.

## 1 Introduction

With the advent of any proposed language system for expressing parallel operation (whether as a true parallel language (e.g., ZPL [6]), parallel extensions to sequential language (e.g., UPC [4]), or parallel compiler directives (e.g., HPF [9])) questions soon arise regarding how performance instrumentation and measurement will be conducted, and how performance data will be analyzed and mapped to the language-level (high-level) parallel abstractions. Several issues make this an interesting problem. First, the language system implements a model for parallelism whose explicit parallel operation is generally hidden from the programmer. As such, parallel performance events may not be accessible directly, requiring instead support from underlying runtime software to observe them in full. When such support is unavailable, performance must be inferred from model properties. Second, the language system typically transforms the program into its parallel executable form, making it necessary to track code transformations closely so that performance data can be correctly mapped to the user-level source. The more complex the transformations, the more difficult the performance mapping will be. Last, high-level language expression of parallelism often goes hand-in-hand with an interest for cross-platform portability of the language system. Users will naturally desire the programming and performance tools to be portable as well.

For the performance tool developer, these issues complicate decisions regarding choice of tool technology and implementation approach. In this paper, we consider the problem of designing a performance tool interface for OpenMP. Three goals for a performance tool interface for OpenMP are considered:

- *Expose OpenMP parallel execution to the performance system.*
  Here we are concerned about what execution events and state data are observable for performance measurement through the interface.
- *Make the interface portable across different platforms and for different performance tools.*

Portability in this regard requires the definition of the interface semantics and how information is to be accessed.

- *Allow flexibility in how the interface is applied.*
  Since OpenMP compilers may implement OpenMP directives differently, including variations in runtime library operation, the performance interface should not constrain how it is used.

While our study focuses mainly on the instrumentation interface, as that is where events are monitored and the operational state is queried, clearly the type of performance measurement will determine the scope of analyses possible. Ideally, the flexibility of the interface will support multiple measurement capabilities.

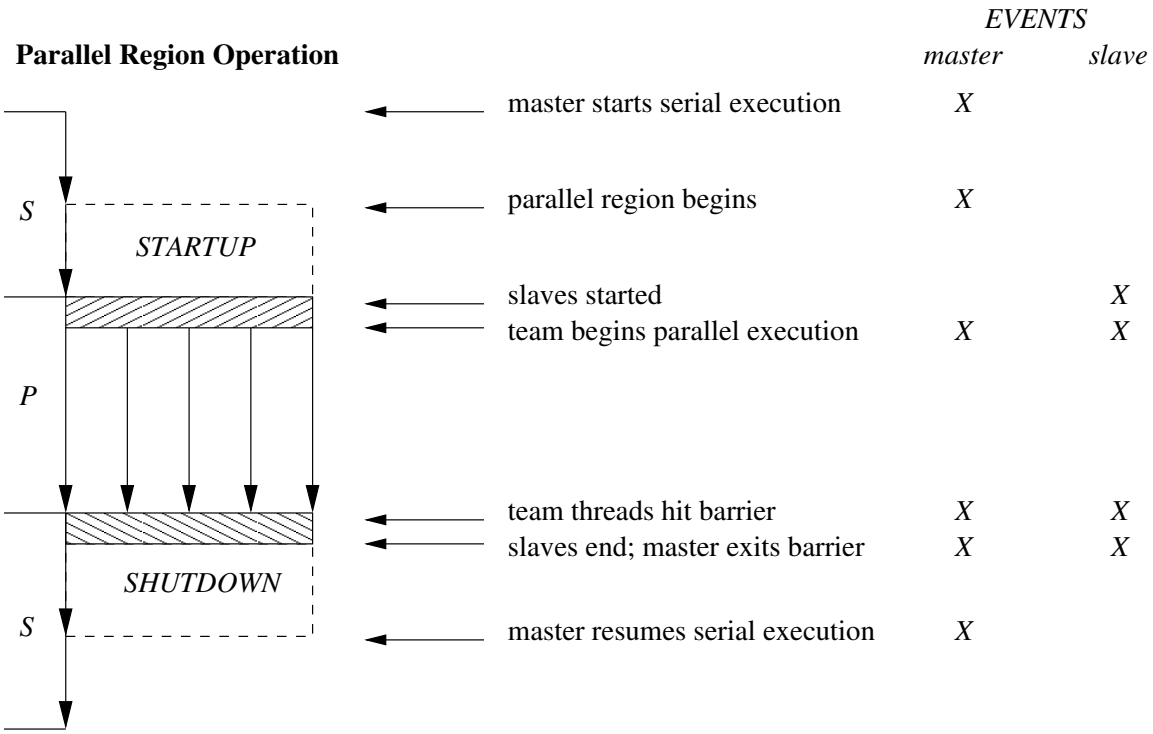


Figure 1: **OpenMP Parallel Region Operation Showing States and Events**

# 2 A General Performance Model for OpenMP

OpenMP is a parallel programming language system used to express shared memory parallelism. It is based on the model of (nested) fork-join parallelism and the notion of *parallel regions* where computational work is shared and spread across multiple threads of execution (a *thread team*); see Figure 1. The language constructs provide for thread synchronization (explicitly and implicitly) to enforce consistency in operation. OpenMP is implemented using comment-style compiler directives (in Fortran) and pragmas (in C and C++).

A performance model for OpenMP can be defined based on its execution events and states. We advocate multiple performance views based on a hierarchy of execution states where each level is more refined in focus:

- *Level 1*: serial and parallel states (with nesting)
- *Level 2*: work sharing states (per team thread)
- *Level 3*: synchronization states (per/across team threads)
- *Level 4*: runtime system (thread) states

In this way, performance observation can be targeted at the level(s) of interest using events specific to the level. Events are defined to identify points of state transitions (begin/end, enter/exit), allowing OpenMP programs to be thought of as multi-threaded execution graphs with states as nodes and events as edges. A performance instrumentation interface would allow monitoring of events and access to state information.

Figure 1 shows a diagram of OpenMP parallel region operation. Identified are serial (*S*) and parallel (*P*) states, parallel startup (*STARTUP*) and shutdown (*SHUTDOWN*) states, and different events at different levels for master and slave threads. Based on this diagram, and given a workable performance instrumentation interface, we can develop measurement tools for capturing serial and parallel performance.

| **Before** | **After** |
|---|---|
| !$OMP PARALLEL<br>*structured block*<br>!$OMP END PARALLEL | call POMP_Parallel_fork(d)<br>!$OMP PARALLEL<br>call POMP_Parallel_begin(d)<br>*structured block*<br>call POMP_Barrier_enter(d)<br>!$OMP BARRIER<br>call POMP_Barrier_exit(d)<br>call POMP_Parallel_end(d)<br>!$OMP END PARALLEL<br>call POMP_Parallel_join(d) |

| **Before** | **After** |
|---|---|
| !$OMP DO<br>*do loop*<br>!$OMP END DO | call POMP_Do_enter(d)<br>!$OMP DO<br>*do loop*<br>!$OMP END DO NOWAIT<br>call POMP_Barrier_enter(d)<br>!$OMP BARRIER<br>call POMP_Barrier_exit(d)<br>call POMP_Do_exit(d) |

| **Before** | **After** |
|---|---|
| !$OMP WORKSHARE<br>*structured block*<br>!$OMP END WORKSHARE | call POMP_Workshare_enter(d)<br>!$OMP WORKSHARE<br>*structured block*<br>!$OMP END WORKSHARE NOWAIT<br>call POMP_Barrier_enter(d)<br>!$OMP BARRIER<br>call POMP_Barrier_exit(d)<br>call POMP_Workshare_exit(d) |
| !$OMP BARRIER | call POMP_Barrier_enter(d)<br>!$OMP BARRIER<br>call POMP_Barrier_exit(d) |

| **Before** | **After** |
|---|---|
| !$OMP SECTIONS<br>!$OMP SECTION<br>*structured block*<br>!$OMP SECTION<br>*structured block*<br>!$OMP END SECTIONS | call POMP_Sections_enter(d)<br>!$OMP SECTIONS<br>!$OMP SECTION<br>call POMP_Section_begin(d)<br>*structured block*<br>call POMP_Section_end(d)<br>!$OMP SECTION<br>call POMP_Section_begin(d)<br>*structured block*<br>call POMP_Section_end(d)<br>!$OMP END SECTIONS NOWAIT<br>call POMP_Barrier_enter(d)<br>!$OMP BARRIER<br>call POMP_Barrier_exit(d)<br>call POMP_Sections_exit(d) |

| **Before** | **After** |
|---|---|
| !$OMP CRITICAL<br>*structured block*<br>!$OMP END CRITICAL | call POMP_Critical_enter(d)<br>!$OMP CRITICAL<br>call POMP_Critical_begin(d)<br>*structured block*<br>call POMP_Critical_end(d)<br>!$OMP END CRITICAL<br>call POMP_Critical_exit(d) |

| **Before** | **After** |
|---|---|
| !$OMP SINGLE<br>*structured block*<br>!$OMP END SINGLE | call POMP_Single_enter(d)<br>!$OMP SINGLE<br>call POMP_Single_begin(d)<br>*structured block*<br>call POMP_Single_end(d)<br>!$OMP END SINGLE NOWAIT<br>call POMP_Barrier_enter(d)<br>!$OMP BARRIER<br>call POMP_Barrier_exit(d)<br>call POMP_Single_exit(d) |

| **Before** | **After** |
|---|---|
| !$OMP ATOMIC<br>*atomic expression* | call POMP_Atomic_enter(d)<br>!$OMP ATOMIC<br>*atomic expression*<br>call POMP_Atomic_exit(d) |

| **Before** | **After** |
|---|---|
| !$OMP MASTER<br>*structured block*<br>!$OMP END MASTER | !$OMP MASTER<br>call POMP_Master_begin(d)<br>*structured block*<br>call POMP_Master_end(d)<br>!$OMP END MASTER |

Table 1: **Proposed OpenMP Directive Transformations.**

# 3 Proposed OpenMP Performance Tool Interface

How should a performance interface be developed to meet the goals for OpenMP? Although different interfaces are possible (see [5, 10]), the basic idea behind our proposal is to define a standard API to a performance measurement library that can be used to instrument a user's OpenMP application program for monitoring OpenMP events. This instrumentation could be done by a source-to-source translation tool prior to the actual compilation or within an OpenMP compilation system. Performance tool developers then only need to implement the functions of this inter-

face to enable their tool to measure and analyze OpenMP programs. Different measurement modes (e.g., profiling [2] and tracing [5, 7, 10]) can easily be accommodated in this way. Hence, the proposed performance interface does not specify how measurements are performed, only the form of the library interface.

We call the OpenMP performance API the "POMP API" and the corresponding performance library the "POMP library," taking after the use of "PMPI" in the definition of the MPI profiling interface. In the following, we present various aspects of our proposal for a standardized performance tool interface using directive rewriting for its implementation. Fortran90 OpenMP 2.0 syntax is used in examples and tables. The transformations equally apply to C/C++.

## 3.1 OpenMP Directive Instrumentation

The POMP API is defined with respect to the semantics of OpenMP operation. Thus, we specify the instrumentation of OpenMP directives in terms of directive transformations because, first, this allows a description independent of the base programming language, and second, the specification is tied directly to the programming model the application programmer understands. Please note that this does not mean that the actual instrumentation has to be based on the technique of directive transformations; we only use it for specification purposes.

Our transformation rules insert calls to `POMP_Name_type(d)` in a manner appropriate for each OpenMP directive, where `Name` is replaced by the name of the directive, `type` is either `fork`, `join`, `enter`, `exit`, `begin`, or `end`, and `d` is a context descriptor (described in Section 3.5). `fork` and `join` mark the location where the execution model switches from sequential to parallel and vice versa, `enter` and `exit` flag the entering and exiting of OpenMP constructs, and, finally, `begin` and `end` mark the start and end of structured blocks used as bodies for the OpenMP directives. Table 1 shows our proposed transformations and performance library routines. To improve readability, optional clauses to the directives, as allowed by the OpenMP standards, are not shown.

As can be seen, the type and placement of POMP calls is intended to expose OpenMP events to the underlying performance measurement system. In some cases, it is necessary to transform the directive in such a way that the operation can be explicitly captured. For instance, in order to measure the synchronization time at the implicit barrier at the end of `DO`, `SECTIONS`, `WORKSHARE`, or `SINGLE` directives, we use the following method:

- If, as shown in the table, the original corresponding `END` directive does not include a `NOWAIT` clause, `NOWAIT` is added and the implicit barrier is made explicit.
- If there is a `NOWAIT` clause in the original `END` directive, then this step is not necessary.

To distinguish these barriers from (user-specified) explicit barriers, the `POMP_Barrier_###()` functions are passed the context descriptor of the enclosing construct (instead of the descriptor of the explicit barrier).

Unfortunately, this method cannot be used for measuring the barrier waiting time at the end of `PARALLEL` directives because they do not come with a `NOWAIT` clause. Therefore, we add an explicit barrier with corresponding performance interface calls here. For source-to-source translation tools implementing the proposed transformations, this means that actually two barriers get called. But the second (implicit) barrier should execute and succeed immediately because the threads of the OpenMP team are already synchronized by the first barrier. Of course, a OpenMP compiler can insert the performance interface calls directly around the implicit barrier, thereby avoiding this overhead.

Transformation rules for the combined parallel work-sharing constructs (`PARALLEL DO`, `PARALLEL SECTIONS`, and `PARALLEL WORKSHARE`) can be defined accordingly; see Table 2. They are basically the combination of transformations for the corresponding single OpenMP constructs after unfolding the combined constructs. The only difference is that clauses specified for the combined construct have to be distributed to the single OpenMP constructs in such a way that it complies with the OpenMP standard (e.g., `SCHEDULE`, `ORDERED`, and `LASTPRIVATE` clauses have to be specified with the inner `DO` directive).

**Before**

```
!$OMP PARALLEL DO clauses ...
    do loop
!$OMP END PARALLEL DO
```

**After**

```
call POMP_Parallel_fork(d)
!$OMP PARALLEL other-clauses ...
    call POMP_Parallel_begin(d)
    call POMP_Do_enter(d)
    !$OMP DO schedule-clauses, ordered-clauses,
             lastprivate-clauses
        do loop
    !$OMP END DO NOWAIT
    call POMP_Barrier_enter(d)
    !$OMP BARRIER
    call POMP_Barrier_exit(d)
    call POMP_Do_exit(d)
    call POMP_Parallel_end(d)
!$OMP END PARALLEL
call POMP_Parallel_join(d)
```

**Before**

```
!$OMP PARALLEL SECTIONS clauses ...
!$OMP SECTION
    structured block
!$OMP END PARALLEL SECTIONS
```

**After**

```
call POMP_Parallel_fork(d)
!$OMP PARALLEL other-clauses ...
    call POMP_Parallel_begin(d)
    call POMP_Sections_enter(d)
    !$OMP SECTIONS lastprivate-clauses
    !$OMP SECTION
        call POMP_Section_begin(d)
        structured block
        call POMP_Section_end(d)
    !$OMP END SECTIONS NOWAIT
    call POMP_Barrier_enter(d)
    !$OMP BARRIER
    call POMP_Barrier_exit(d)
    call POMP_Sections_exit(d)
    call POMP_Parallel_end(d)
!$OMP END PARALLEL
call POMP_Parallel_join(d)
```

**Before**

```
!$OMP PARALLEL WORKSHARE clauses ...
    structured block
!$OMP END PARALLEL WORKSHARE
```

**After**

```
call POMP_Parallel_fork(d)
!$OMP PARALLEL clauses ...
    call POMP_Parallel_begin(d)
    call POMP_Workshare_enter(d)
    !$OMP WORKSHARE
        structured block
    !$OMP END WORKSHARE NOWAIT
    call POMP_Barrier_enter(d)
    !$OMP BARRIER
    call POMP_Barrier_exit(d)
    call POMP_Workshare_exit(d)
    call POMP_Parallel_end(d)
!$OMP END PARALLEL
call POMP_Parallel_join(d)
```

Table 2: **Proposed OpenMP Combined Parallel Work-sharing Directive Transformations.**

## 3.2 OpenMP Runtime Library Routine Instrumentation

To monitor OpenMP runtime library routine calls, the transformation process replaces these calls by calls to the performance tool interface library. For example, an OpenMP API call to `omp_set_lock()` is transformed into a call to `POMP_Set_lock()`. In the implementation of the performance interface function, the original corresponding OpenMP runtime library routine must be called. Performance data can be obtained before and after. Currently, we think it is sufficient to use this procedure for the `omp_###_lock()` and `omp_###_nest_lock()` routines because they are most relevant for the observation of OpenMP performance behavior.

## 3.3 Performance Monitoring Library Control

In addition to the performance library interface, we propose to add a new directive to give the programmer control over when the performance collection is done:

```
!$OMP INST [INIT | FINALIZE | ON | OFF]
```

For normal OpenMP compilation this directive is ignored. Otherwise, it is translated into calls of `POMP_Init()`, `POMP_Finalize()`, `POMP_On()`, and `POMP_Off()` calls when performance instrumentation is requested. If compatibility with existing OpenMP compilers is essential, new directives with the sentinel `!$POMP` could be used. Yet another approach (which does not extend the set of OpenMP directives) would be to have the programmer add the performance tool interface calls directly, but this then requires either stub routines, conditional compilation, or the removal of the instrumentation to be used when performance monitoring is not desired. Our proposed new directive approach is more portable, effective, and easier to maintain.

| **PARALLEL** | **MASTER** | **SINGLE** |
|---|---|---|
| POMP_Parallel_fork(d)<br>POMP_Parallel_begin(d)<br>POMP_Parallel_end(d)<br>POMP_Parallel_join(d) | POMP_Master_begin(d)<br>POMP_Master_end(d) | POMP_Single_enter(d)<br>POMP_Single_begin(d)<br>POMP_Single_end(d)<br>POMP_Single_exit(d) |
| **DO / FOR** | **WORKSHARE** | **SECTION / SECTIONS** |
| POMP_Do_enter(d)<br>POMP_Do_exit(d)<br><br>POMP_For_enter(d)<br>POMP_For_exit(d) | POMP_Workshare_enter(d)<br>POMP_Workshare_exit(d) | POMP_Sections_enter(d)<br>POMP_Section_begin(d)<br>POMP_Section_end(d)<br>POMP_Sections_exit(d) |
| **BARRIER** | **CRITICAL** | **ATOMIC** |
| POMP_Barrier_enter(d)<br>POMP_Barrier_exit(d) | POMP_Critical_enter(d)<br>POMP_Critical_begin(d)<br>POMP_Critical_end(d)<br>POMP_Critical_exit(d) | POMP_Atomic_enter(d)<br>POMP_Atomic_exit(d) |
| **Runtime Library** | **Control** | **User Code** |
| POMP_Set_lock()<br>POMP_Unset_lock()<br>POMP_Set_nest_lock()<br>POMP_Unset_nest_lock | POMP_Init()<br>POMP_Finalize()<br>POMP_On()<br>POMP_Off() | POMP_Begin()<br>POMP_End() |

Table 3: **POMP OpenMP Performance API.**

## 3.4 User Code Instrumentation

For large application programs it is usually not sufficient to just collect OpenMP related events. The OpenMP compiler should also insert appropriate `POMP_Begin()` and `POMP_End()` calls at the beginning and end of each user function. In this case the context descriptor describes the user function. In addition, users may desire to mark arbitrary (non-function) code regions. This can be done with a directive mechanism similar to that described in the last subsection, such as

```
!$OMP INST BEGIN ( <region_name> )
    structured block
```

```
!$OMP INST END ( <region_name> )
```

The directives are translated into `POMP_Begin()` and `POMP_End()` calls. Again, techniques can be used to avoid defining new directives, but with the same disadvantages as described in the last section. Furthermore, the transformation tool / compiler cannot generate the context descriptor for this user defined region in this case, so another (less efficient) mechanism would have to be used here.

The full proposed OpenMP performance API is shown in Table 3.

## 3.5 Context Descriptors

An important aspect of the performance instrumentation is how the performance tool interface routines get access to context information, in order to relate the collected performance information back to the source code and OpenMP constructs. We propose the following. For each instrumented OpenMP construct, user function, and user-specified region, the instrumentor generates a *context descriptor* in the global static memory segment of the compilation unit containing the construct or region. All monitoring function calls related to this construct or region are passed the address of this descriptor (called `d` in Tables 1 and 2). The proposed definition of the context descriptor (in C syntax) is shown below:

```
typedef struct ompregdescr {
    char* name;
    char* sub_name;
    int   num_sections;
    char* file_name;
    int   begin_line1, begin_lineN;
    int   end_line1, end_lineN;
    void* data;
    struct ompregdescr* next;
} OMPRegDescr;
```

The fields of the context descriptor have the following meaning: `name` contains the name of the OpenMP construct or the string `"region"` for user functions and regions. `sub_name` stores the name of named critical regions or the name of user functions and regions. In case of the `sections` OpenMP directives, `num_sections` provides the number of sections, otherwise it is set to 0. The next five fields (`file_name`, `begin_line1`, `begin_lineN`, `end_line1`, `end_lineN`) describe the source code location of the OpenMP construct or user region: the source file name, and the first and last line number of the opening and of the corresponding `END` OpenMP directive. The field `data` can be used by the performance tool interface functions to store performance data related to this construct or region (e.g., counters or timers). Finally, the `next` component allows for chaining context descriptors together at runtime, so that at the end of the program the list of descriptors can be traversed and the collected performance data can be stored away and analyzed.

This approach has many advantages over other methods (e.g., using unique identifiers):

1. Full context information, including source code location, is available to the performance tool interface functions.
2. Runtime overhead for implementing this approach is minimal. Only one address is passed as an argument.
3. The context data is kept together with the (instrumented) executable so it avoids problems of locating (the right) separate context description file(s) at runtime.
4. Finally, it allows for separate compilation. This is important for today's large complex application codes.

An alternative approach would be to define a POMP API function which gets the context information passed in as separate parameters and returns the address of an internal descriptor. The instrumentation process would arrange that for each region this POMP function would be called exactly once. The returned descriptor would then be passed in to the other POMP functions as usual. This would have the advantage that the layout of a context descriptor is not fixed and a POMP library implementation can define it in an optimal way, e.g., adding fields to store library specific (performance) data. The disadvantage is that instrumentation becomes more difficult because it has to be ensured

that such a function is only called exactly once, e.g., using additional locking and synchronisation calls, resulting in more measurement overhead.

## 3.6 Conditional Compilation

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 the Sections 2.1.3.1 and 2.1.3.2 of the OpenMP Fortran Application Program Interface, Version 2.0, November 2000.

## 3.7 Conditional / Selective Transformations

Finally, to allow a user to (temporarily) disable the POMP instrumentation process for specific files or parts of files, we propose to provide the following new directives:

```
!$OMP NOINSTRUMNENT
!$OMP INSTRUMNENT
```

(or `!$POMP ...`, as necessary) which disable and re-enable POMP instrumentation (similar to common existing compiler directives for disabling optimizations). POMP instrumentation disabled by `!$OMP NOINSTRUMENT` is in effect until end of file or until the next `!$OMP INSTRUMENT` whichever comes first.

## 3.8 C/C++ OpenMP Pragma Instrumentation

The transformations for Fortran OpenMP directives described in Tables 1 and 2 apply equally to C/C++ OpenMP pragmas. The main difference is that the extent of C/C++ OpenMP pragmas is determined by the structured block following it, and not by an explicit `END` pragma as in Fortran. This has the following consequences for pragma instrumentation:

- Instrumentation for the "closing" part of the pragma follows the structured block.
- Adding a `nowait` clause (to allow the make implicit barriers explicit) has to be done for the "opening" part of the pragma.
- The *structured block* of a C/C++ OpenMP pragma (`#pragma omp ###`) will be transformed by wrapping it with `POMP_###_begin(d)` and `POMP_###_end(d)` calls which in turn are enclosed in a block (i.e., using {...}).

All other changes are simple differences in language (e.g., no `call` keyword and using `#pragma omp` instead of `!$OMP`). The new monitoring control and user code instrumentation would appear as:

```
#pragma omp inst [init | finalize | on | off]

#pragma omp inst begin ( <region_name> )
    structured block
#pragma omp inst end ( <region_name> )

#pragma omp noinstrument
#pragma omp instrument
```

## 3.9 Implementation Issues

It is clear that an interface for performance measurements must be very efficient and must minimize its influence on the dynamic behavior of the user code under investigation. We designed our POMP library interface with efficiency in mind from the very beginning:

- We choose to define a larger set of performance monitoring API functions according to the pattern

  `POMP_Name_type(descr)`

  instead of using an interface like

  `POMP_Event(POMP_Name, POMP_type, filename, linenumber, value, ...)`

  which would have been much simpler to implement. But it would make it necessary to use costly `IF` or `SWITCH` statements inside the `POMP_Event` routine, resulting in run-time overhead.

- Argument passing overhead is minimized (while still supporting full context information) by using context descriptors. This was already discussed in Section 3.5.

- If further optimization is required, inlining of the POMP interface calls through the use of macros would be possible. In C/C++, this is possible without any changes to the proposal as outlined above as the function call and macro call syntax is the same. Fortran OpenMP instrumentation only requires the generation of the `CALL` keyword during OpenMP directive transformation to be suppressed. However, the structure of a single POMP call is so simple (i.e., a function call with one constant argument) that current compiler technology should be able to inline the calls without the "manual" use of macros.

- The granularity of OpenMP constructs can be much finer than, for example, MPI functions. This is especially true for the `ATOMIC`, `CRITICAL`, `MASTER`, and `SINGLE` OpenMP constructs as well as for the OpenMP API functions for locking. Implementers of the POMP interface must take special care when implementing the corresponding POMP functions. In addition, we propose that the instrumentation of these constructs can be selectively disabled by specifying the following command line option to the OpenMP compiler or POMP transformation tool:

  `--pomp-disable=construct[,construct ...]`

  where construct is the name of one OpenMP construct listed above or `"sync"` to disable all of them. Finally, if the instrumentation of these (or other) constructs is critical only in specific parts of the code, our `!$OMP NOINSTRUMENT` directive (see Section 3.7) can be used to disable them selectively.

## 3.10 Open Issues

While we think that the proposal outlined so far is reasonably complete, there are additional issues which need further discussion. Here, we try to briefly summarize the issues we are aware of:

- Do we need instrumentation for the OpenMP `ORDERED` and `FLUSH`?
- Do we need instrumentation of single iterations of `PARALLEL DO` (`parallel for`) loops? This would potentially allow the influence of OpenMP loop iteration scheduling policies to be measured but may introduce a large measurement overhead if iterations are not of a sufficient granularity.
- Should the context descriptor also contain additional information about parallel loops, e.g., the scheduling policy, the chunk size, and whether there is a reduction defined for this loop?
- Do we need a way to allow users to pass additional information to the POMP interface? Besides using POMP conditional compilation (see Section 3.6), we might consider extending the `!$OMP INST BEGIN` and `END` directive (see Section 3.4) to optionally allow to pass the address of a user variable:

  ```
  !$OMP INST BEGIN ( <region_name> [, <variable>] )
      structured block
  !$OMP INST END ( <region_name> [, <variable>] )
  ```

  The `POMP_Begin` and `POMP_End` routines would have an additional `void*` typed second parameter which would be `NULL` if the user did not specify a variable. This way, the user could pass additional information to the POMP API functions, which would be ignored in general implementations but could be utilized in (user-supplied) special versions. It is also possible to allow passing user information to the other (all) POMP API functions, but this requires further directives / pragmas.
- Do we need additional OpenMP runtime level instrumentation or is it enough to observe OpenMP behavior on the source code level?

# 4 Prototype Implementation

To integrate performance tools with the proposed OpenMP performance interface, two issues must be addressed. First, the OpenMP program must be instrumented with the appropriate performance calls. Second, a performance library must be developed to implement the OpenMP performance API for the particular performance tool. The following describes how two performance tools, EXPERT and TAU have been integrated with the proposed OpenMP performance interface. In each case, both OpenMP applications and hybrid (OpenMP+MPI) applications are supported. The latter demonstrates the ability to combine the OpenMP performance interface with other performance interface mechanisms in a seamless manner.

## 4.1 Automatic Instrumentation

As a proof of concept and a means for experimentation, we implemented OPARI (**O**penMP **P**ragma **A**nd **R**egion **I**nstrumentor). It is a source-to-source translator which performs the OpenMP directive and API call transformations described in this paper, including the new proposed directives and `!$POMP INST` alternative sentinel. The current prototype implements full Fortran77 and Fortran90 OpenMP 2.0 and full C/C++ OpenMP 1.0 support. The instrumentation of user functions (based on PDT [12]) is under way. The tool consists of about 2000 lines of C++ code.

### 4.1.1 Limitations due to Fuzzy Parsing

Being just a source-to-source translator based on a (very) fuzzy parser, and not a full compiler, OPARI has a few small limitations:

**Fortran**

- The `!$OMP END DO` and `!$OMP END PARALLEL DO` directives are required (not optional, as described in the OpenMP standard).
- The *atomic expression* controlled by a `!$OMP ATOMIC` directive has to be on a line all by itself.

**C/C++**

- *Structured blocks* describing the extent of an OpenMP pragma need to be either compound statements ({...}), or simple statements. In addition, *for loops* are supported only after `omp for` and `omp parallel for` pragmas.
- Complex statements like *if-then-else* or *do-while* need to be enclosed in a block ({...}).

We did not find these limitations overly restrictive during our tests and experiments. They rarely apply for well-written code. If they do, the original source code can easily be fixed. Of course, it is possible to remove these limitations by enhancing OPARI's parsing capabilities.

Finally, if the performance measurement environment does not support the automatic recording of user functions entries and exits, and therefore cannot automatically instrument the program's `main` function, the OPARI runtime measurement library has to be initialized by a `!$OMP INST INIT` directive / pragma prior to any other OpenMP pragma.

### 4.1.2 Limitations due to Source-to-source Translation

In addition, because of some subtleties in the OpenMP standard specifications, the transformations performed by OPARI on the source code level can differ from the same instrumentation done by a real OpenMP compiler. Here is the list of limitations we currently know about:

- OPARI makes implicit barriers explicit. Unfortunately, this method cannot be used for measuring the barrier waiting time at the end of `PARALLEL` directives because they do not allow a `NOWAIT` clause. Therefore, we add an explicit barrier with corresponding performance interface calls here. For OPARI, this means that actually two barriers get called. But the second (implicit) barrier should execute and succeed immediately because the threads of the OpenMP team are already synchronized by the first barrier.
- The OpenMP standard (unfortunately) allows compilers to ignore `NOWAIT`s, which means that in this case OPARI inserts an extra barrier and the POMP functions get invoked on this extra (and not the real) barrier.
- OPARI cannot instrument the (required) internal synchronization inside `!$OMP WORKSHARE`.
- We were told that some compilers use different implementations (with different characteristics) for implicit and explicit barriers. If OPARI changes implicit to explicit barriers, we measure the wrong behavior when using these compilers.

Of course, an OpenMP compiler can insert the POMP calls directly around the implicit barriers, thereby avoiding the described overheads and discrepancies.

## 4.2 Integration into EXPERT

The EXPERT tool environment [17, 18] is aimed at automatically uncovering performance problems in event traces of MPI, OpenMP, or hybrid applications running on complex, large SMP clusters. The work on EXPERT is carried out within the KOJAK project [11] and is a part of the ESPRIT working group APART [1].

EXPERT analyzes the performance behavior along three dimensions: *performance problem category*, *dynamic call tree position*, and *code location*. Each of the analyzed dimensions is organized in a hierarchy. Performance problems are organized from more general ("*There is an MPI related problem*") to very specific ones ("*Messages sent in wrong order*"). The dynamic call tree is a natural hierarchy showing call stack relationships. Finally, the location dimension represents the hierarchical hardware and software architecture of SMP clusters consisting of the levels machine, node, process, and thread.

The range of performance problems known to EXPERT are not hard-coded into the tool but are provided as a collection of *performance property specifications*. This makes EXPERT extensible and very flexible. A performance property specification consists of

- a compound event (i.e., an event pattern describing the nature of the performance problem),
- instructions to calculate the so-called *severity* of the property, determining its influence on the performance of the analyzed application,
- its parent performance property, and

- instructions on how to initialize the property and how to display collected performance data or property related results.

Performance property specifications are abstractions beyond simple performance metrics, allowing EXPERT to explain performance problems in terms of the underlying programming model(s). Specifications are written in the event trace analysis language EARL [16], an extension of the Python scripting language. EARL provides efficient access to an event trace at the level of the abstractions of the parallel programming models (e.g., region stack, message queue, or collective operation) making it easy to write performance property specifications.

EXPERT's analysis process relies on event traces as performance data source. Event traces preserve the temporal and spatial relationship among individual events, and they are necessary to prove certain interesting performance properties. Event traces are recorded in the newly designed EPILOG format that, in contrast to traditional trace data formats, is suitable to represent the executions of MPI, OpenMP, or hybrid parallel applications being distributed across one or more (possibly large) clusters of SMP nodes. It supports storage of all necessary source code and call site information, hardware performance counter values, and marking of collectively executed operations for both MPI and OpenMP. The implementation of EPILOG is thread safe, a necessary feature not always present in most traditional tools.

Traces can be generated for C, C++, and Fortran applications just by linking to the EPILOG tracing library. To intercept user function calls and returns, we use the internal profiling interface of the PGI compiler suite [15] being installed on our LINUX SMP cluster testbed. For capturing OpenMP events, we implemented the POMP library functions in terms of EPILOG tracing calls, and then use OPARI to instrument the user application. For example, the `POMP_For_enter()` and `POMP_For_exit()` interface implementation for instrumentation of the `#pragma omp parallel for` directive for C/C++ would look like the following in EPILOG:

```
void POMP_For_enter(OMPRegDescr* r) {
  struct ElgRegion* e;
  if (! (e = (struct ElgRegion*)(r->data[0])))
    e = ElgRegion_Init(r);
  elg_enter(e->rid);
}

void POMP_For_exit(OMPRegDescr* r) {
  elg_omp_collexit();
}
```

What is important to notice is how the region descriptor is utilized to collect performance data per OpenMP construct. For hybrid applications using OpenMP and MPI, MPI-specific events can also be generated by an appropriate wrapper function library utilizing the MPI standard profiling interface.

## 4.3 Integration into TAU

The TAU performance system [13] provides robust technology for performance instrumentation, measurement, and analysis for complex parallel systems. It targets a general computation model consisting of shared-memory *nodes* where *contexts* reside, each providing a virtual address space shared by multiple *threads* of execution. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Because TAU enables performance information to be captured at the node/context/thread levels, this information can be mapped to the particular parallel software and system execution platform under consideration.

TAU supports a flexible instrumentation model that allows access to a measurement API at several stages of program compilation and execution. The instrumentation identifies code segments, provides for mapping of low-level execution events to high-level computation entities, and works with multi-threaded and message passing parallel execution models. It interfaces with the TAU measurement model that can capture data for function, method, basic block, and statement execution. Profiling and tracing form the two measurement choices that TAU provides. Performance experiments can be composed from different measurement modules, including ones that access hardware performance monitors. The TAU data analysis and presentation utilities offer text-based and graphical tools to visualize the performance data as well as bridges to third-party software, such as Vampir [14] for sophisticated trace analysis and visualization.

As with EXPERT, TAU implements the OpenMP performance API in a library that captures the OpenMP events and uses TAU's performance measurement facility to record performance data. For example, the POMP implementation of the same functions as in Section 4.2 would look like the following in TAU:

```
TAU_GLOBAL_TIMER(tfor,``for enter/exit'', ``[OpenMP]'',OpenMP);

void POMP_For_enter(OMPRegDescr* r) {
#ifdef TAU_AGGREGATE_OPENMP_TIMINGS
  TAU_GLOBAL_TIMER_START(tfor);
#endif
#ifdef TAU_OPENMP_REGION_VIEW
  TauStartOpenMPRegionTimer(r);
#endif
}

void POMP_For_exit(OMPRegDescr* r) {
#ifdef TAU_AGGREGATE_OPENMP_TIMINGS
  TAU_GLOBAL_TIMER_STOP();
#endif
#ifdef TAU_OPENMP_REGION_VIEW
  TauStopOpenMPRegionTimer(r);
#endif
}
```

TAU supports construct-based as well as region-based performance measurement. Construct-based measurement uses globally accessible timers to aggregate construct-specific performance cost over all regions. In the case of region-based measurement, the region descriptor is used to select the specific performance data for that context. Following this instrumentation approach, all of TAU's functionality is accessible to the user, including the ability to select profiling or tracing, enable hardware performance monitoring, and add MPI instrumentation for performance measurement of hybrid applications.

# 5 Example Applications

To demonstrate the viability and robustness of the proposed interface and instrumentation tools we have developed, the OPARI tool and the POMP libraries for EXPERT and TAU are applied to example mixed-mode (OpenMP + MPI) applications. These types of application present an analysis challenge for performance tools because of the need to capture and present performance data for different parallel modes and their interaction.

## 5.1 Weather Forecasting

The REMO weather forecast application from the DKRZ (Deutsches Klima Rechenzentrum, Germany) is an excellent testcase for the performance API. The code is instrumented using OPARI for OpenMP events and the MPI profiling library for MPI events. The measurement system uses the EPILOG tracing facility, with the POMP library calling EPILOG trace routines, as described above. The EXPERT system then processes the events traces and displays the performance analysis results.

In EXPERT, the presentation of the results is also based on three dimensions: performance problem, call graph, location. Each dimension is displayed using *weighted trees*. A weighted tree is a tree browser that labels each node with a weight. EXPERT uses the performance property's severity as this weight. The weight is displayed simultaneously using both a numerical value as well as a color coded icon. The color allows to identify nodes of interest easily even in a large tree. By expanding or collapsing any of the three trees it is possible to analyze the performance behavior of a parallel application on different levels of granularity.

We see some of this interactive analysis in Figure 2 and 3. The three tree views are shown. The first view lists the different performance properties. The numbers at the nodes show the percentage of CPU allocation time spent on

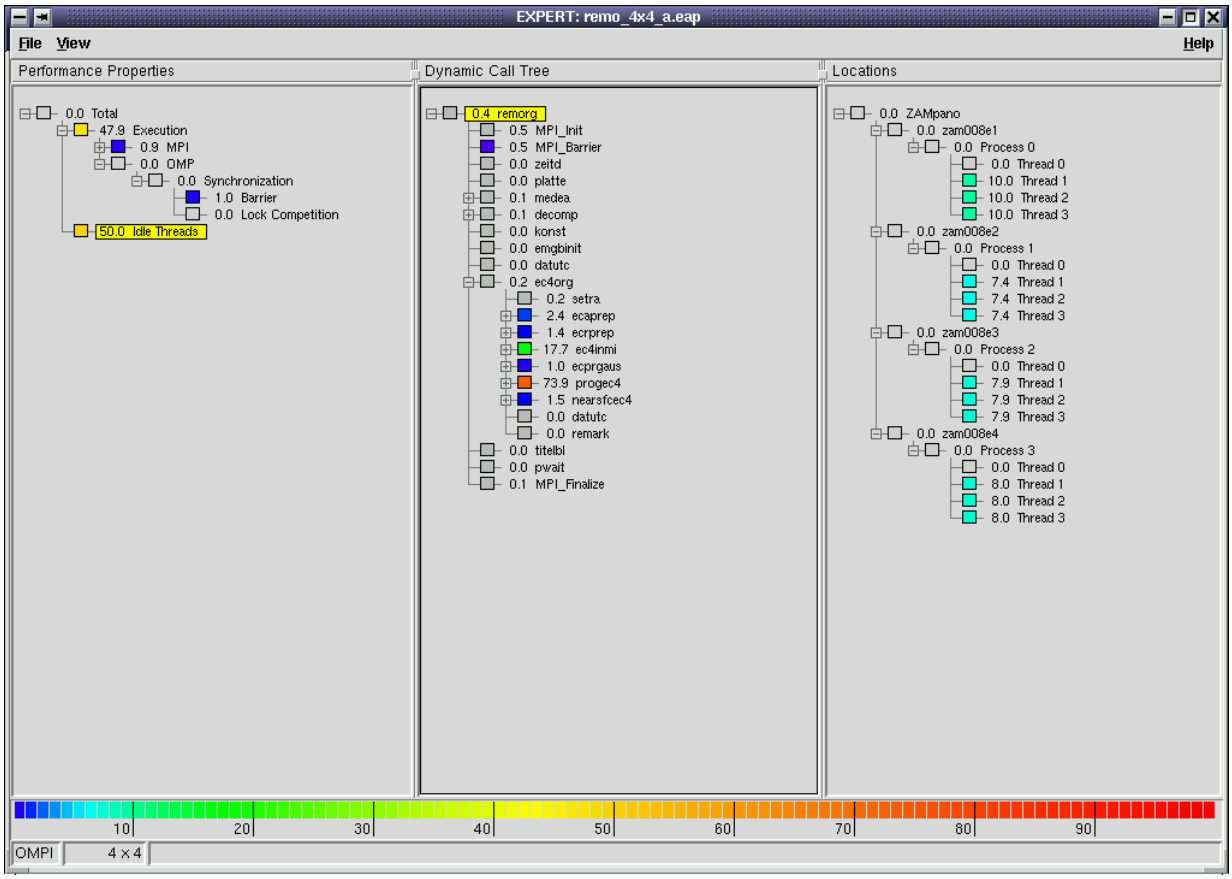


Figure 2: **EXPERT Performance Analysis of OpenMP/MPI Weather Forecasting Application Instrumented with OPARI**

that property or lost as a result of that property. When the node is collapsed, the inclusive value is shown, when it is expanded only its exclusive value. Colors represent the numeric value graphically.

Figure 2 shows that 50% of CPU allocation time was lost as a result of sequential execution or was "used by idle slave threads." Although idle threads do not execute any code, the idle time is mapped to the middle (call tree) view (i.e., the idle slave threads are considered to be executing the same code as the corresponding master threads). This allows the user to find code regions that spent a very long time on sequential execution. The numbers in the middle view refer to the selection in the left view, so 73.5% of the 50.0% are spent in `/remo/ed4org/progec4`.

The right view shows the distribution of idle times across the different threads. Here all values refer to the selection in the left neighbor, so the sum of all values correspond to the 73.9% from the middle view. Of course, only the slave threads have idle times, the master thread shows always 0.0%.

Figure 3 refers to the property "OpenMP Barrier." The call tree shows that nearly all barrier time is spent on an implicit barrier (`!$omp ibarrier`) belonging to a parallel do (`!$omp do`). The distribution of overhead across the different threads is shown in the right view.

## 5.2 Ocean Circulation

To demonstrate the use of the OpenMP performance tool interface with TAU, we applied it to a two-dimensional Stommel ocean current application from the San Diego Supercomputing Center. The application code models wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom, and Coriolis force. A 5-point stencil is used to solve partial differential equation on a grid of points. Table 4 shows the source code for a more compute-intensive `for` block, before and after instrumentation

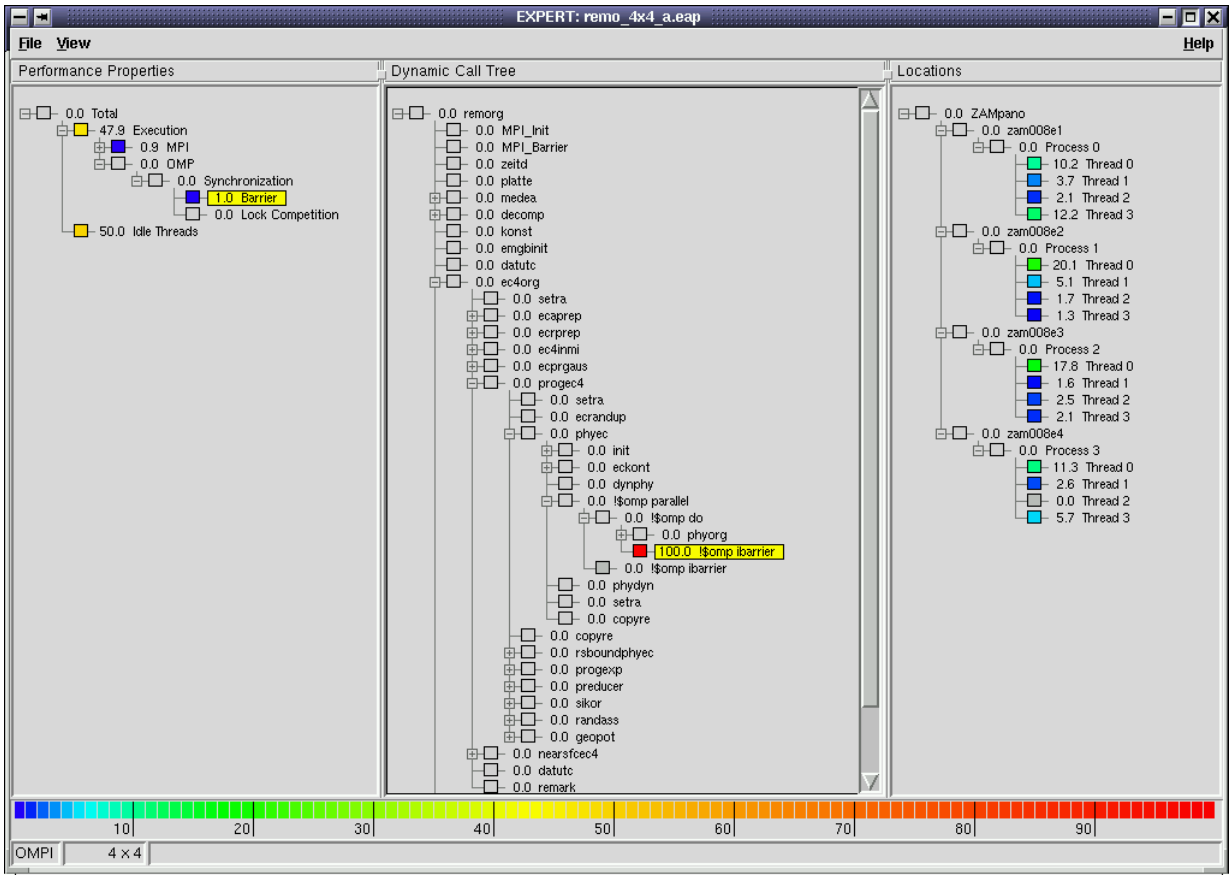


Figure 3: **Barrier Performance Analysis of REMO**

with OPARI. By linking with the TAU-specific POMP library and a user-configured TAU measurement package, the performance data for OpenMP and MPI events can be captured and displayed.

Figure 4 presents profiling data for the Stommel application. Shown is a region-based performance view where individual parallel loops are distinguished. The `for` block shown in Table 4 is highlighted in the "n,c,t 0,0,0 profile" display (representing node 0, context 0, and thread 0) and is seen to take a significant percentage of time to execute. The execution time for this block across all threads is shown in the "for

$$OpenMPlocation : file.stommel.c < 252, 260 >$$

profile" display. Clearly, there is a work imbalance between the two threads within each process, but the distribution is consistent across nodes (i.e., processes). Notice how the MPI performance data is integrated with the OpenMP data in the display. It is also possible for TAU to be used to obtain construct-based performance data.

By linking the Stommel application with a trace-configured performance library, OpenMP and MPI events can be displayed using the Vampir [14] visualization tool. Figure 5 displays an event timeline showing the overlaps of OpenMP and MPI events.

# 6 Related Work

Given the interest in OpenMP in the last few years, several research efforts have addressed performance measurement and analysis of OpenMP execution, but none of these efforts have considered a common performance tool interface in the manner proposed in this paper. The OVALTINE tool [2] helps determine relevant overheads for a parallel OpenMP program compared to a serial implementation. It uses the Polaris Fortran 77 parser to build a basic

Original code block

```
#pragma omp for schedule(static) reduction(+: diff) private(j)
        firstprivate (a1,a2,a3,a4,a5)
  for(i=i1;i<=i2;i++) {
      for(j=j1;j<=j2;j++){
          new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] + a3*psi[i][j+1] + a4*psi[i][j-1]
                        - a5*the_for[i][j];
          diff=diff+fabs(new_psi[i][j]-psi[i][j]);
       }
  }
```

Code block after OPARI instrumenation

```
POMP_For_enter(&omp_rd_2);
#line 252 "stommel.c"
#pragma omp for schedule(static) reduction(+: diff) private(j)
        firstprivate (a1,a2,a3,a4,a5) nowait
  for(i=i1;i<=i2;i++) {
      for(j=j1;j<=j2;j++){
          new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] + a3*psi[i][j+1] + a4*psi[i][j-1]
                        - a5*the_for[i][j];
          diff=diff+fabs(new_psi[i][j]-psi[i][j]);
       }
  }
POMP_Barrier_enter(&omp_rd_2);
#pragma omp barrier
POMP_Barrier_exit(&omp_rd_2);
POMP_For_exit(&omp_rd_2);
```

Table 4: **Directive Instrumentation for Stommel Code.**

abstract syntax tree which it then instruments with counters and timers to determine overheads for various OpenMP constructs and code segments. The nature of the OVALTINE performance measurements suggests that our OpenMP performance API could be applied directly to generate the OpenMP events of interest, allowing to use a greater range of performance tools for the overhead analysis.

OMPtrace [5] is a dynamic instrumentation package used to trace OpenMP execution on SGI and IBM platforms. It provides for automatic capture of OpenMP runtime system (RTS) events by intercepting calls to the RTS library. User functions can also be instrumented to generate trace events. The main advantage of OMPtrace is that there is no need to re-compile the OpenMP program for performance analysis. In essence, OMPtrace uses the RTS interface as the performance tool interface, relying on interception at dynamic link time for instrumentation. Unfortunately, this approach relies on OpenMP compiler transformations that turn OpenMP constructs into function calls, and on dynamic shared library operation. To bypass these restrictions, the OpenMP performance interface we propose could provide a suitable target for the performance tracing part of OMPtrace. A compatible POMP library would need to be developed to generate equivalent OMPtrace events and hardware counter data. In this manner, the Paraver [7] tool for analysis and visualization of OMPtrace data could be used without modification.

The VGV tool combines the OpenMP compiler tools (Guide, GuideView) from KAI with the Vampir/Vampirtrace tracing tools from Pallas for OpenMP performance analysis and visualization. OpenMP instrumentation is provided by the Guide compiler for both profiling and tracing, and the Guide runtime system handles recording of thread events. Being compiler-based, the monitoring of OpenMP performance can be quite detailed and tightly integrated in the execution environment. However, the lack of an external API prevents other performance tools for observing OpenMP execution events. The performance interface we proposed could be applied in the VGV context in the same manner as above. The POMP calls could be implemented in a library for VGV, mapping the OpenMP actions to Vampir state transition calls at appropriate points. Another approach might be to have the Guide compiler generate the POMP instrumentation, allowing other POMP-compatible performance interface libraries to be used.

Lastly, the JOMP [3] system is a source-to-source compiler that transforms OpenMP-like directives for Java to multi-thread Java statements that implement the equivalent OpenMP parallel operations. It has similarities to our

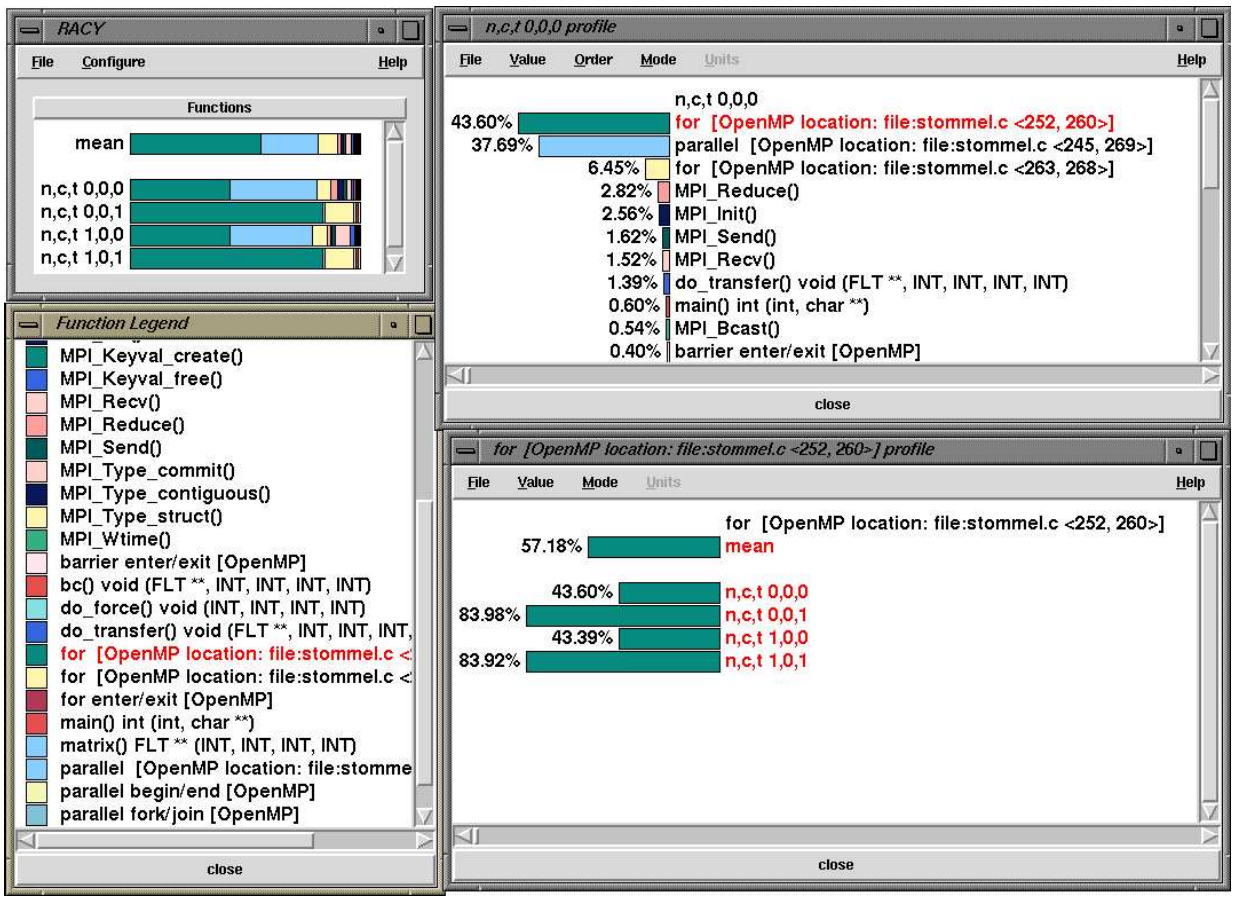


Figure 4: **TAU Performance Profile of an OpenMP/MPI 2D Stommel Model of Ocean Circulation Instrumented with OPARI**

work in that it supports performance instrumentation as part of its directive transformation [8]. This instrumentation generates events for analysis by Paraver [7]. In a similar manner, the JOMP compiler could be modified to generate POMP calls. In this case, since JOMP manages its own threads to implement parallelism, it may be necessary to implement runtime support for POMP libraries to access thread information.

# 7 Conclusion and Future Work

This paper proposes a portable performance interface for OpenMP to aid in the integration of performance tools into OpenMP programming environments. Defined as a library API, the interface exposes OpenMP execution events of interest (e.g., sequential, parallel, and synchronization events) for performance observation, and passes OpenMP context descriptors to inform the performance interface library of region-specific information. Because OpenMP uses compiler directives (pragmas) to express shared memory parallelism, our definition of the performance tool API must be consistent with the operational semantics of the directives. To show how this is accomplished, we describe how the API is used in rewriting OpenMP directives in functionally equivalent, but source-instrumented forms. The OPARI tool can perform this OpenMP directive rewriting automatically, inserting POMP performance calls where appropriate.

The benefits of the proposed performance interface are manifold. First, it provides a performance API target for source-to-source instrumentation tools (e.g., OPARI), allowing for instrumented OpenMP codes that are portable across compilers and machine platforms. Second, the performance library interface provides a target for tool developers to port performance measurement systems. This enables multiple performance tools to be used in OpenMP

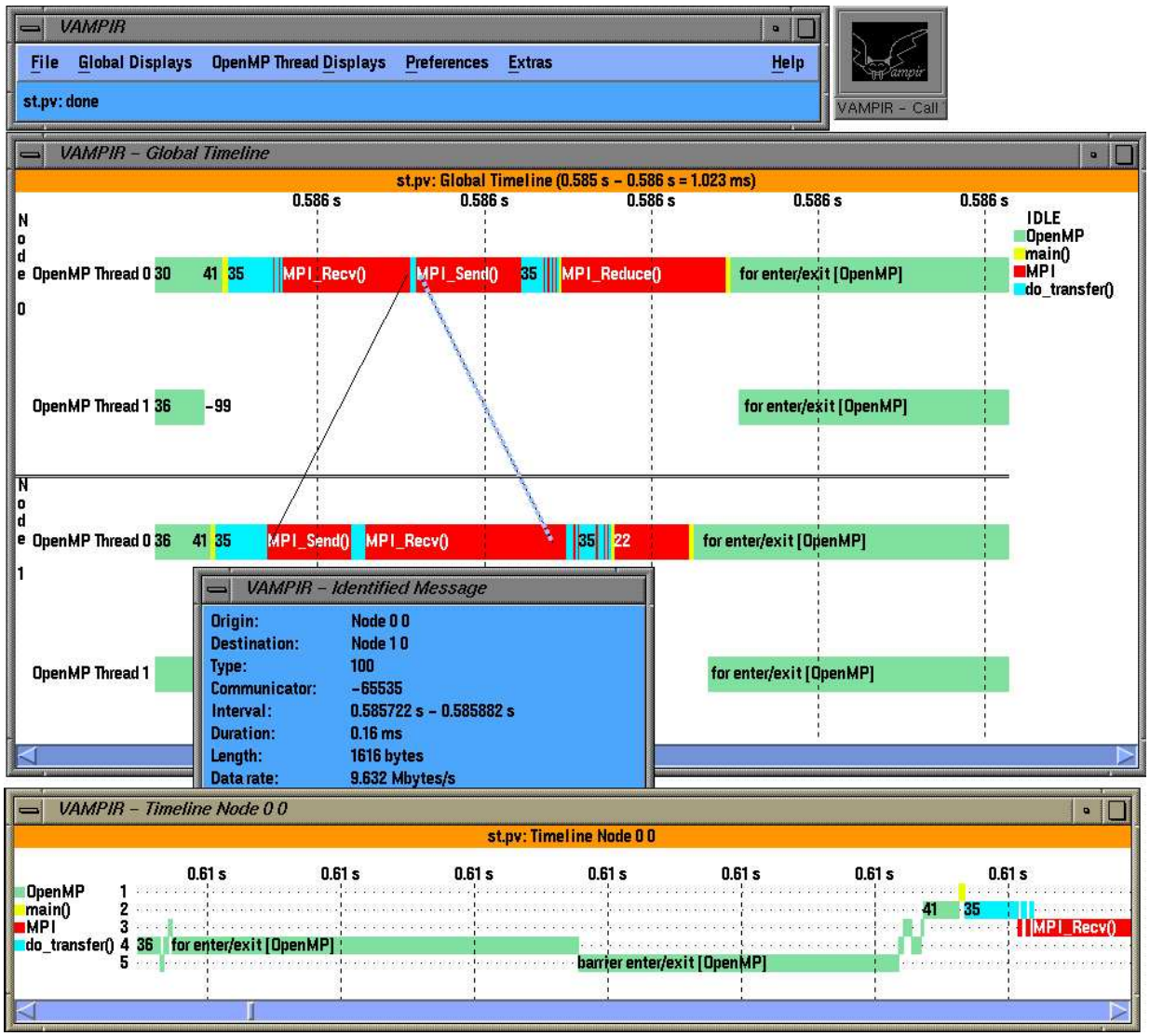


Figure 5: **TAU Performance Trace of Stommel Application**

performance analysis. We show how this is accomplished for EXPERT and TAU by implementing the POMP calls. Third, the API also offers a target for OpenMP compilers to generate POMP calls that can both access internal, compiler-specific performance libraries and external performance packages. Finally, if the OpenMP community could adopt an OpenMP performance interface such as the one we proposed, it would significantly improve the integration and compatibility between compilers and performance tools, and, perhaps more importantly, the portability of performance analysis techniques.

In the future, we hope to work with the OpenMP ARB to promote the definition for a performance tool API, offering our proposal here for consideration. We will enhance the OPARI source-to-source instrumentation approach with support for user function instrumentation using PDT [12]. Other opportunities are also possible with the integration of the API in OpenMP compilers and the use of other performance technologies for instrumentation and measurement. We hope to work with KAI and Pallas to investigate the use of our proposed performance tool interface in the KAP/Pro Guide compiler with Vampirtrace as the basis for the POMP performance library implementation.

# References


[1] ESPRIT Working Group APART (Automatic Performance Analysis: Resources and Tools). http://www.fz-juelich.de/apart/.

[2] M. Bane and G. Riley, "Overheads Profiler for OpenMP Codes," *European Workshop on OpenMP (EWOMP 2000)*, September, 2000.

[3] J. Bull et al., "Towards OpenMP for Java," *European Workshop on OpenMP (EWOMP 2000)*, Sep. 2000.

[4] W. Carlson et al. "Introduction to UPC and Language Specification," Technical Report CCS-TR-99-157, George Mason University, May, 1999.

[5] J. Caubet et al., "A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications," *Workshop on OpenMP Applications and Tools (WOMPAT 2001)*, July, 2001.

[6] B. Chamberlain et al., "The Case for High Level Parallel Programming in ZPL," *IEEE Computational Science and Engineering*, 5(3):76-86, 1998.

[7] European Center for Parallelism of Barcelona (CEPBA), *Paraver – Parallel Program Visualization and Analysis Tool – Reference Manual*, November, 2000. http://www.cepba.upc.es/paraver.

[8] J. Guitart et al., "Performance Analysis Tools for Parallel Java Applications on Shared-memory Systems," *Int'l. Conf. on Parallel Processing (ICPP'01)*, September, 2001.

[9] HPF. http://softlib.rice.edu/HPFF/.

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

[11] KOJAK (Kit for Objective Judgment and Knowledge-based Detection of Bottlenecks). http://www.fz-juelich.de/zam/kojak/.

[12] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, "Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates," *Proc. Supercomputing 2000*, Dallas/Texas, USA, November, 2000.

[13] 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.

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

[15] Portland Group Inc. Private Communication.

[16] F. Wolf, B. Mohr, "EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs," *Proc. of the 7th Int'l. Conf. on High-Performance Computing and Networking*, HPCN'99, A. Hoekstra and B. Hertzberger, eds., Amsterdam (The Netherlands), 1999, pp. 503–512.

[17] F. Wolf, B. Mohr, "Automatic Performance Analysis of MPI Applications Based on Event Traces," *Proc. of the European Conf. on Parallel Computing*, Euro-Par 2000, Munich (Germany), August 2000, pp. 123–132.

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