A Performance Monitoring Interface for OpenMP

Bernd Mohr
Forschungszentrum Jülich
b.mohr@fz-juelich.de

Hans-Christian Hoppe
Frank Schlimbach
Pallas GmbH
hans-christian.hoppe@pallas.de
frank.schlimbach@pallas.de

Grant Haab
Jay Hoeflinger
Sanjiv Shah
Intel, KAI Software Lab
grant.haab@intel.com
jay.p.hoeflinger@intel.com
sanjiv.shah@intel.com

Allen Malony
University of Oregon
malony@cs.uoregon.edu

@cs.uoregon.edu
A Short History of OpenMP Instrumentation

• KAI KAP/Pro Toolset, GuideView
  • First commercial performance tool for OpenMP
• POMP OpenMP performance monitoring interface
  • Forschungszentrum Jülich, University of Oregon
  • Presented at EWOMP’01, LACSI’01, and SC’01
• OMPI Proposal of European IST Project INTONE
  • Development of OpenMP tools (incl. monitoring interface)
  • Pallas, CEPBA, Royal Inst. Of Technology, Tech. Univ. Dresden
  • http://www.cepba.upc.es/intone
• KSL-POMP
  • Development of OpenMP monitoring interface inside ASCI
  • Based on POMP, but further developed in other directions
Goals of this Work

• Define a common proposal for a Performance Monitoring Interface for OpenMP

• Document
  • List of technical issues to resolve
  • Discussions
  • Rationale
  • Open problems

• Provide a “base document” for future standardization through OpenMP ARB Tools Subcommittee
Objectives

• Allow alternative instrumentation methods (source-level, compile-time and binary-level, run-time)
• Allow alternative implementations of POMP interface (including profile and tracing-based measurement)
• Make interface portable across different platforms and languages and usable for different performance tools
• Do not prelude combination with other interfaces, especially PMPI
• Do not constrain the implementation of OpenMP compilers and run-time systems
• Do not constrain efficient implementations of POMP library
• Allow application programmers to specify and control instrumentation consistent with the use of OpenMP
Outline of the Proposal

• OpenMP Event Model
• Management of Event Context Information
• Instrumentation of
  • OpenMP Constructs and Directives
  • OpenMP API Functions
  • User Functions and Regions
• Monitoring Library Control
• Conditional Monitoring Code
• Instrumentation Control
  • Event Groups and Levels
  • Directives
• Nested Parallelism
OpenMP Event Model

• Three groups of events
  • OpenMP constructs and directives/pragmas
    - Enter/Exit around OpenMP construct plus Begin/End around associated body
    - Special case parallel loop: ChunkBegin/End, IterBegin/End or IterEvent instead of Begin/End
    - “single” events for small constructs like atomic or flush
  • OpenMP API calls
    - Enter/Exit events around omp_set_*_lock() functions
    - “single” events for all API functions
  • User functions and regions
    - Enter/Exit and “single” events

• Naming: <Construct>_[ enter | exit | begin | end | event ]
Management of Event Context Information (I)

- **Compile time context (CTC) information**
  - known at compile / instrumentation time
  - example: source code information or attributes of pragmas
  - make available for POMP library

- **Run time context (RTC) information**
  - has to be passed as parameters to POMP event functions
  - example: thread ID

- **Library data**
  - data associated with events by POMP library
  - example: library IDs or performance data
  - allocated + initialized by POMP library but needs instrumentor for associate library data to all events of a OpenMP construct
Management of Event Context Information (II)

- **Original POMP**
  - CTC stored in fixed data structure in static memory
  - Includes zero-initialized library handle
  - Library data can be allocated + initialized on first call
  - Fixed data structure hard to define portably across languages
  - Allocation + Initialization inside POMP event calls
    requires synchronization

- **OMPI**
  - CTC is passed to "context constructor" calls
  - Return handle to internal data structure
  - Solves portability problem
  - Allows partitioning of CTC information by using different handles
  - Still requires synchronization
Management of Event Context Information (III)

- CTC information is encoded in a string for maximum portability

- Standard instrumentation
  - POMP event calls get passed CTC string + zero-initialized library handle
  - POMP library responsible for portable + efficient synchronization

- Optimized instrumentation
  - If possible for instrumentor, POMP_Get_handle() called at program startup for each CTC string; returns library handle
  - POMP event call gets passed NULL string + library handle
  - Avoids / minimizes synchronization
## CTC Strings

### Content

<table>
<thead>
<tr>
<th>ALL entities</th>
<th>region type, start SCL, end SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus for parallel regions and workshare constructs</td>
<td>hasFirstPrivate, hasLastPrivate, hasReduction, hasNoWait, hasCopyin</td>
</tr>
<tr>
<td>plus for parallel loops</td>
<td>scheduleType, hasOrdered</td>
</tr>
<tr>
<td>plus for single</td>
<td>hasCopyprivate</td>
</tr>
<tr>
<td>plus for critical</td>
<td>name</td>
</tr>
<tr>
<td>plus for user functions and regions</td>
<td>name, group</td>
</tr>
</tbody>
</table>

### Format:

```
"<len>*type=<type>*name=<name>*...**"
```

- list of attribute=value pairs separated by "*"
- terminating empty field ("**")
- length useful for sizing buffers etc.
Instrumentation of OpenMP Constructs and Directives

- C/C++

  ```
  int32 POMP_<event>(POMP_handle_t* libhandle,
                      int32 thread_id,
                      ... /*other RTC info*/
                      char ctc_string[])`
  ```

- Fortran

  ```
  INTEGER*4 POMP_<event>(libhandle, thread_id,
                          ..., ctc_string)
  ```

  ```
  INTEGER*<ptrsize> lib_handle
  INTEGER*4 thread_id
  CHARACTER*(*) ctc_string
  ```

- `<event>` is Parallel_enter, Single_begin, atomic_event, ...
- Run time context (RTC) information passed as parameters
- CTC string passed only to 1<sup>st</sup> POMP call associated with construct
- All POMP routines return status code
- Open Issues: mixed-case spelling of names, status code definition

© 2002 Forschungszentrum Jülich, NIC-ZAM
RTC Information Parameters

- Run time context information in addition to thread_id:

<table>
<thead>
<tr>
<th>Event</th>
<th>RTC Parameters</th>
<th>NA value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel_enter</td>
<td>int32 num_threads</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>int32 if_expr_result</td>
<td>-1</td>
</tr>
<tr>
<td>Loop_enter</td>
<td>int64 chunk_size</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>int64 init_iter_value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int64 final_iter_value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int64 incr</td>
<td>0</td>
</tr>
<tr>
<td>Loop_chunk_begin</td>
<td>int64 init_iter_value</td>
<td></td>
</tr>
<tr>
<td>Loop_iter_begin</td>
<td>int64 iter_value</td>
<td></td>
</tr>
<tr>
<td>Loop_iter_event</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section_begin</td>
<td>int32 section_num</td>
<td>-1</td>
</tr>
</tbody>
</table>
Example: Standard Instrumentation

```c
1: int main() {
2:     int id;
3:     #pragma omp parallel private(id)
4:     {
5:       id = omp_get_thread_num();
6:       printf("hello from \%d\n", id);
7:     }
8:     #pragma omp parallel private(id)
9:     {
10:       int32 pomp_tid = omp_get_thread_num();
11:       POMP_Parallel_enter(&pomp_hdl1, pomp_tid, -1, 1,
12:         "49*type=pregion*file=demo.c*slines=4,4*elines=8,8**");
13:       id = omp_get_thread_num();
14:       printf("hello from \%d\n", id);
15:       POMP_Parallel_end(pomp_hdl1, pomp_tid);
16:     }
17:     POMP_Parallel_exit(pomp_hdl1, pomp_tid);
18: }
19: POMP_Finalize();
20: }
```
Example: Optimized Instrumentation

1: int main() { 
2:     int id; 
***     POMP_handle_t pomp_hd1 = 0; 
***     POMP_Init(); 
***     POMP_Get_handle(&pomp_hd1, 
***         "49*type=pregion*file=demo.c*slines=4,4*elines=8,8**"); 
3: 
***     { int32 pomp_tid = omp_get_thread_num(); 
***       POMP_Parallel_enter(&pomp_hd1, pomp_tid, -1, 1, NULL); 
4:     #pragma omp parallel private(id) 
5:     { 
***       int32 pomp_tid = omp_get_thread_num(); 
***       POMP_Parallel_begin(pomp_hd1, pomp_tid); 
6:       id = omp_get_thread_num(); 
7:       printf("hello from %d\n", id); 
***       POMP_Parallel_end(pomp_hd1, pomp_tid); 
8:     } 
***     POMP_Parallel_exit(pomp_hd1, pomp_tid); 
***     } 
***     POMP_Finalize();
9: }
Instrumentation of OpenMP API Functions

- `omp_set_lock, omp_set_nest_lock`
  - begin/end events to allow measurement of blocking time
- all OpenMP API function
  - “single” events to keep instrumentation overhead small

- Interface and naming like POMP calls for OpenMP constructs
- RTC information

<table>
<thead>
<tr>
<th>all functions</th>
<th>int32 thread_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock functions</td>
<td>omp_lock_t lock_id</td>
</tr>
<tr>
<td>get/set functions</td>
<td>value set/returned</td>
</tr>
</tbody>
</table>
Instrumentation of User Functions and Regions

- Function call sites
  - `POMP_Function_enter/_exit()`

- Function bodies
  - `POMP_Function_begin/_end()`
  - `POMP_Function_event()`

- User-specified regions (possibly nested, structured blocks)
  - `POMP_user_region_begin/_end()`
  - `POMP_user_event()`

- Open issues
  - Automatic instrumentation of functions by instrumentor?
  - How to specify extent of automatic function instrumentation?
  - Separate events for function call site and body?
  - Allow passing a set of values to `POMP_user_region()` calls?
Monitoring Library Control

• Calls to initialize and finalize POMP library
  • to be inserted by the instrumentor in the master thread code
  • at program startup and before program termination
  • But POMP library must be able to auto-initialize and auto-finalize
    
    POMP_Init()
    POMP_Finalize()

• Calls for the user to signal start/stop monitoring

    POMP_On()
    POMP_Off()

• Open Issues
  • Restrictions on placement of POMP_On() / POMP_Off()?
Conditional Monitoring Code

- C, C++, [Fortran, if supported]

```c
#ifdef __POMP
    arbitrary user code
#endif
```

- Fortran Free Form

```fortran
!P$ arbitrary user code
```

- Fortran Fixed Form

```fortran
CP$ arbitrary
*P$ user
!P$ code
```

- Usual restrictions apply

- **Optional**, as non source-code based instrumentors cannot support this
## Runtime Instrumentation Control: Event Groups/Levels

- By setting environment variables `POMP_<group>`
- By calling `POMP_Set_group()`

<table>
<thead>
<tr>
<th>Group</th>
<th>Constructs</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Parallel</td>
<td>None, EnterExit, BeginEnd</td>
</tr>
<tr>
<td>Loop</td>
<td>Do/For</td>
<td>None, EnterExit, (Chunks</td>
</tr>
<tr>
<td>Workshare</td>
<td>Section, Single Workshare</td>
<td>None, EnterExit, BeginEnd</td>
</tr>
<tr>
<td>Workshare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sync</td>
<td>Critical, Master, Barrier, Ordered</td>
<td>None, EnterExit, BeginEnd, Event</td>
</tr>
<tr>
<td></td>
<td>Atomic, Flush</td>
<td>None, Event</td>
</tr>
<tr>
<td>User</td>
<td>Function, Region</td>
<td>None, (Event</td>
</tr>
<tr>
<td>Runtime</td>
<td>API calls</td>
<td>None, (Event</td>
</tr>
</tbody>
</table>

- Levels also include events from levels listed left from them
Instrumentation Control: (Optional) Directives

- Environment variables/API calls only practical for run time control
- New directive would be helpful for instrumentation-time control

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

- **Pro/Contra new directives:**
  - ☺ Core of OpenMP uses directives, too
  - ☺ Instrumentation control is specified within program not separate
  - ☺ Only practical way of controlling parts of program
  - ☺ Only way to completely remove instrumentation overhead
  - 😞 New directives heavy burden for vendors
  - 😞 Might be confusing / complex for user, e.g., extent of directive
  - 😞 Need re-compile for change in instrumentation control
  - 😞 Cannot be supported by non source-code based instrumentors

⇒ controversial issue
Directives for User Defined Events (I)

- Example: Instrumentation for user-specified region

```c
#define _POMP
{
    POMP_handle_t my_hdle = 0;
    int my_id = omp_get_thread_num();
    POMP_user_region_begin(&my_hdle, my_id,
                           "31*type=userregion*name=myphase**");

#define _POMP
    ...
    ...

#define _POMP
    POMP_user_region_end(my_hdle, my_id);
#define _POMP
```
Directives for User Defined Events (II)

- Simplify specification of user-defined events

```
!$POMP USERREGION BEGIN(<name>)
... structured block ...  
!$POMP USERREGION END(<name>)

!$POMP USERREGION EVENT(<name>)
```

- Arguments about new directives also apply here
  ⇒ Even more controversial
Nested Parallelism

• Not completely clear yet how to make sure our proposal can handle nested parallelism

• Need some concept of OpenMP team on OpenMP API level
  • `omp_get_global_thread_num()`?
  • `omp_get_team_num()`?
  • `omp_get_team_master()`?

• Needs to be tackled by OpenMP ARB first
Summary

• Agreement
  • OpenMP event model
  • Context information management
  • Instrumentation of OpenMP directives and API functions
  • Instrumentation of user functions and regions
  • Instrumentation control by event groups and levels
  • Monitoring library control
  • Conditional monitoring code

• Disagreement
  • Need for new directives

• Open issues
  • Monitoring of nested parallelism
Resources

• Original POMP proposal (TJS Aug 2002)
  • www.cs.uoregon.edu/research/paraducks/papers/tjs02/tjs02.pdf.gz
  • www.cs.uoregon.edu/research/paraducks/papers/tjs02/tjs02.ps.gz
• Prototype src-to-src instrumentor OPARI
  Implements the original POMP proposal
  • www.fz-juelich.de/zam/kojak/opari/

• The OMPI proposal (European INTONE project)
  • www.cepba.upc.es/intone/papers/specification.pdf
• INTONE Prototype compilers
  • www.cepba.upc.es/intone/