An Implementation of the POMP
Performance Monitoring for OpenMP
based on Dynamic Probes

Luiz DeRose
IBM Research
ACTC
Yorktown Heights, NY
USA
laderose@us.ibm.com

Bernd Mohr
Forschungszentrum Jülich
ZAM / NIC
52425 Jülich
Germany
b.mohr@fz-juelich.de

Seetharami Seelam
IBM Research
ACTC
Yorktown Heights, NY
USA
srseelam@us.ibm.com
Outline

- What is POMP
- What is DPCL
- IBM compiler and run-time library features that makes dPOMP possible
- Implementation
  - POMP not supported features (and why)
  - Changes and extensions
  - Overhead measurements
- Examples of use
- Conclusions
A Brief History of OpenMP Instrumentation

- POMP1 OpenMP performance monitoring interface
  - Forschungszentrum Jülich, University of Oregon
- European IST Project INTONE
  - Development of OpenMP tools (includes Monitoring interface)
  - Pallas, CEPBA, Royal Inst. Of Technology, Tech. Univ. Dresden
  - http://www.cepba.upc.es/intone/
- Intel KAI Software Laboratory (KSL) - POMP
  - Development of OpenMP monitoring interface inside ASCI
  - Based on POMP, but further developed in other directions
- Joint proposal presented at EWOMP’02
  - ⇒ POMP2 == POMP
What is POMP?

- Proposed standard for a performance monitoring interface for OpenMP
  - Portable cross-platform/cross-language API to simplify the design and implementation of OpenMP tools
  - Three groups of events
    - **OpenMP constructs and directives/pragmas**
      - Enter/Exit around each OpenMP construct
      - Begin/End around associated body
      - Special case for parallel loops:
        - 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**
  - Allows application programmers to specify and control amount of instrumentation
Example: Standard Instrumentation

```c
int main() {
    int id;
    POMP_Init();
    
    {  
        int32 pomp_tid = omp_get_thread_num();  
        POMP_Parallel_enter(&pomp_hd1, pomp_tid, -1, 1,  
               "49*type=pregion*file=demo.c*slines=4,4*elines=8,8**");
    
    #pragma omp parallel private(id)
    {  
        int32 pomp_tid = omp_get_thread_num();
        POMP_Parallel_begin(pomp_hd1, pomp_tid);
        id = omp_get_thread_num();
        printf("hello from %d\n", id);
        POMP_Parallel_end(pomp_hd1, pomp_tid);
    }  
    POMP_Parallel_exit(pomp_hd1, pomp_tid);
    }  
    POMP_Finalize();
} 
```
Example:  **Optimized Instrumentation**

```c
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:  }
```
dPOMP Motivation

- POMP under review by the OpenMP ARB!
  - May take too long to be implemented (if accepted)

- Our Approach
  - A POMP implementation based on dynamic probes
    - Built on top of DPCL
      - Modifies the binary with performance instrumentation
      - No source code or re-compilation required
    - A validation infrastructure for the POMP proposal
What Is DPCL?

- C++ Based Class Library
  - IBM Poughkeepsie Unix Development Lab
  - 11 Classes, Plus Additional API's
- Dynamic Instrumentation - Software Probes
  - Based on DynInst and Paradyn
- Language/Programming Model Independent
  - Supports Fortran, Fortran 90, C, C++
  - Requires only information from the executable (a.out)
- Provides a general purpose infrastructure for:
  - Serial, shared memory, and message passing
- A Platform to Enable Tools Developers To Build Tools With Less Time And Effort
DPCL Probes

- DPCL allows tools to insert data, functions, and code patches (probes) into a program dynamically
  - Call site
  - Call entry
  - Call exit
- Probes can collect and report program information, program state, or modify the program execution
- Probes may be placed at specific locations in the program and can be activated:
  - Whenever execution reaches that location
  - By expiration of a timer
  - Exactly once
dPOMP: DPCL based POMP

- The IBM compiler and run-time library

Source code

```c
main() {
    A() {
        OMP parallel
        OMP loop
        OMP end parallel
    }
}
```

Compiler generated

```c
main() {
    A() {
        POMP_Parallel_begin
        POMP_Parallel_exit
    }
}
```

run-time library

A@0L1

master thread

all threads

A@0L1@OL2

do I=start,end
loop body
endo
dPOMP

- Input parameters:
  - OpenMP application (or mixed-mode)
  - POMP compliant monitoring library
  - List of user functions to instrument (optional)
    
    \[ \text{dpomp [-f function.lst] libpomp a.out} \]

- Performs binary instrumentation
  - Amount of instrumentation can be controlled by
    
    Environment variables and/or
    
    Set of POMP calls available in the monitoring library

- Executes instrumented application
Limitations

- 63 out of 68 POMP events supported!
- Limitations due to compiler issues
  - POMP_Loop_iter_(begin, or end, or event)
  - POMP_Implicit_barrier_(end, or exit)
  - OMP Parallel Loop NOT = OMP Parallel / OMP Loop
  - Compile Time Context (CTC)
    - hasFirstPrivate, hasLastPrivate, hasNowait, hasCopyin, schedule, hasOrdered, and hasCopypriv not available
- Limitations due to DPCL issues
  - Loop iteration values (init, final, incr, chunk)
- Limitations due to lack of time …
  - C++ support
Changes and Extensions Due to Open Issues

- Fully defined attribute and values for CTC string
- Event handler is always passed by reference
- Finer instrumentation control
  - **User defined functions**
    - Function calls in “main” program (outside parallel regions) are instrumented by default
    - User can provide a file with functions to instrument
  - **POMP Events**
    - Only events supplied in the monitoring libraries are instrumented
Overhead Measurements

- Minimal code for a typical POMP event routine

```c
int32 POMP_Parallel_begin(POMP_Handle_t handle, int32 thread_id)
{
    if ( pomp_active && is_activated[MYPOMP_PAR_BEGIN] )
        mypompdescr* d = (mypompdescr*) handle;
    /* -- perform monitoring for parallel begin event here -- */
    return 0;
}
```

- Measured Overhead: 3$\lambda$s of CPU time per call
  - On IBM POWER4+ at 1.45 GHz
KOJAK POMP Library

- POMP monitoring library which generates EPILOG event traces
  - Processed by KOJAK’s automatic event tracer analyzer EXPERT
EPILOG Trace Converted to VTF3

- EPILOG-to-VTF3

Maps OpenMP constructs into VAMPIR symbols and activities
POMP Profiler Library

- Generates a profile of the time spent in OpenMP loops and user functions
  - Viewed with the PeekPerf Graphical Interface from IBM ACTC
Conclusions

- DPCL based implementation of the POMP performance monitoring interface for OpenMP
  - Easy to use
  - Low instrumentation overhead

Test it at the OMP Lab

- Two POMP Libraries
  - KOJAK POMP Library
  - POMP Profiler Library
  - or build your own library