Providing Observability for OpenMP 3.0 Applications

Yuan Lin, Oleg Mazurov
Overview

• Objective
• The Data Model
• OpenMP Runtime API for Profiling
• Collecting Data
• Examples
• Overhead
Objective

• Provide observability of OpenMP applications in terms of OpenMP user model
  > Implementation independent
    > Compiler transformations
    > OpenMP runtime internals

• Practically usable: variety of tools, low overhead
  > Data volume
  > Multithreaded environment
  > Task oriented execution model
Data model

- User callstack (logical call path)
- OpenMP parallel region tree
- OpenMP task tree
- OpenMP thread state / OpenMP metrics
User callstack

- Implementation agnostic
- Functions from user code only
  - Hides compiler generated outlined functions
  - Hides OpenMP runtime frames
- Maintains caller-callee relationship as specified by user's program
- Allows 'apples-to-apples' comparison:
  - serial vs. parallel
  - one OpenMP implementation vs. another
Dynamic function call graph

- All callstacks aggregated by common nodes
- Every leaf node represents a callstack
- Multiple call sites are preserved
OpenMP parallel region tree

- Reflects parallel region creation (with nesting), thread teams
- Levels of detail:
  - Full dynamic tree
  - Sampled dynamic tree
  - Aggregated tree
    - Full path
    - Only parent-child relationship
OpenMP task tree

- Reflects task creation
- Same levels of detail:
  > Full
  > Sampled
  > Aggregated

Task from Implicit ParReg
Data model (summary)

- An arbitrary program event is described by:
  > OpenMP thread state
  > A node in the parallel region tree
  > A node in the task tree
  > User callstack

- Non-OpenMP data:
  > Timestamp
  > System thread ID
  > CPU ID
  > Machine callstack
OpenMP Runtime API for Profiling

- Formal definition at
  > http://www.compunity.org/futures/omp Collector api.h

- White paper at
  > http://www.compunity.org/futures/omp-api.html

- Last updated on Oct 31, 2007

- Dynamically loaded agent
  > Sends requests to the OpenMP runtime
  > Receives event notifications from the OpenMP runtime
Meta API

```c
int __omp_collector_api( void *arg )
```

- Request layout

<table>
<thead>
<tr>
<th>Size</th>
<th>Request #</th>
<th>Error code</th>
<th>Rsize</th>
<th>Mem</th>
</tr>
</thead>
</table>

- Multiple requests

| Request 1 | Request 2 | ... | Request n | 0   |
Meta API

- Dynamic binding (versionless)
- One call – many requests (consistency)
- Multi-value returns
- Variable size returns (strings, arrays, etc.)
- Full memory control on the consumer side
- Extensible
Profiling API – examples

- **Requests**
  > OMP_REQ_START – start profiling session
  > OMP_REQ_REGISTER – register an event callback
  > OMP_REQ_STATE – report current thread state

- **Events**
  > OMP_EVENT_FORK – a new parallel region started
  > OMP_EVENT_JOIN – a parallel region finished
Profiling API – extension

#define OMP_REQ_SUNEXTENSION \ ((OMP_COLLECTORAPI_REQUEST)0x4A415641)

#define OMP_REQ_XREQ1 ((OMP_COLLECTORAPI_REQUEST)-1)
#define OMP_REQ_XREQ2 ((OMP_COLLECTORAPI_REQUEST)-2)
...
#define OMP_EVENT_XEVENT1 ((OMP_COLLECTORAPI_EVENT)-1)
#define OMP_EVENT_XEVENT2 ((OMP_COLLECTORAPI_EVENT)-2)
...
Collecting data

- Collecting user callstacks
- Collecting parallel region tree information
- Collecting task tree information
- Collecting thread states and computing metrics
Collecting user callstacks

```c
int main ( int argc, char *argv[] ) {
    foo();
}

void foo () {
    #pragma omp task
    {
        ...
        #pragma omp task
        {
            ...
            #pragma omp task
            {
                ...
                bar();
            }
        }
    }
    #pragma omp taskwait
}
```
Collecting user call stacks

Thread 1

main()
foo()
_omp_task_wait_()
_omp_execute_task_()
_mf_foo_2_()
bar()
...

Thread 2

_omp_slave_thread_()
_omp_execute_task_()
_mf_foo_1_()
_omp_create_task_()
Collecting user call stacks

Thread 1

main()

foo()

_omp_task_wait_()

_omp_execute_task_()

_mt.foo.2_()

bar()

...

Thread 2

_omp_slave_thread_()

_omp_wait_for_work_()
Collecting user callstacks

- User callstack can be reconstructed from machine callstack segments representing user code
- Not all segments are necessarily present
- Store user callstack at task spawn within the task
- To reconstruct user callstack (UC) at an arbitrary point, combine user callstack at task spawn (TSUC) with the current local segment (LS):

\[ UC = TSUC + LS \]
Collecting user callstacks

- New helper mechanism: requests originated on the OpenMP runtime side
Collecting user callstacks

- User callstack is represented by a 64-bit hash code
- Collector records all necessary info to ensure reverse mapping at analysis time
- Collector minimizes recording duplicate information
- OMPX_REQ_CONTEXT – get all information about the local segment (agent request)
- OMPX_HLP_UCTX – get user callstack for specified segment (runtime request)
Collecting user callstacks

- Special cases
  - OpenMP runtime overhead on behalf of user's task
  - OpenMP runtime user API: `omp_get_thread_num()`, ...
  - Calling a copy constructor for `firstprivate` C++ objects
  - Immediate task execution: if() clause, internal conditions
Parallel region tree information

- New requests
  - OMP_REQ_PREG_NLVLS – tree depth for the current parallel region
  - OMP_REQ_PREG_IDN – 64-bit ID for parallel region at level N
  - OMP_REQ_PREG_THRIDN – OpenMP thread ID at level N
  - OMP_REQ_PREG_TMSZDN – thread team size at level N
  - OMPX_REQ_PREG_PPCN – instruction address representing parallel region at level N
Parallel region tree information

• For each encountered parallel region:
  > Parallel region ID
  > Thread team size
  > Representative address
  > Immediate parent's parallel region ID
  > Immediate parent's thread ID

• Keep a table of already recorded IDs
Collecting task tree information

- New requests
  - OMP_REQ_TASK_NLVLS – task tree depth for the current task
  - OMP_REQ_TASK_IDN – 64-bit task ID at level N
  - OMPX_REQ_TASK_PPCN – instruction address representing task at level N
- For each encountered task: ID, representative address, immediate parent's ID
OpenMP states and metrics

- **OMP_REQ_STATE** – calling thread OpenMP state:
  > THR_WORK_STATE, THR_OVHD_STATE, THR_IBAR_STATE, THR_IDLE_STATE, THR_REDUCE_STATE, THR_CTWT_STATE, THR_TSKWT_STATE, ...

- Aggregate time spent in various states:
  > OMP Work: work, reduction, serial, no state
  > OMP Wait: idle, overhead, barriers, waits, etc.

- Attribute to various program objects:
  > Functions, threads, parallel regions, tasks, etc.
Example – quick sort

```c
void quick_sort( int lt, int rt, float *data ) {
    if ( rt - lt < LOWLIMIT ) {
        insertion_sort( lt, rt, data );
        return;
    }
    int md = partition( lt, rt, data );
    #pragma omp task firstprivate(lt, md, data)
    quick_sort( lt, md - 1, data );
    #pragma omp task firstprivate(md, rt, data)
    quick_sort( md + 1, rt, data );
}
```
Example – integer partitioning

```c
long intpart( long sum, long *arr, int len ) {
    if ( len == 0 ) return ( sum == 0 ) ? 1L : 0L;
    if ( len <= 20 ) return intpart_ser( sum, arr, len );

    int idx = len - 1;
    long res = 0;
    #pragma omp task shared(res) firstprivate(sum, arr, idx)
    res += intpart( sum + arr[idx], arr, idx );
    #pragma omp task shared(res) firstprivate(sum, arr, idx)
    res += intpart( sum - arr[idx], arr, idx );
    #pragma omp taskwait
    return res;
}
```
Quick sort – machine view
Quick sort – user view
Integer partitioning – user view
OpenMP tasks with metrics
Data collection overhead

• Systems
  > T5440, UltraSPARC T2+, 4x8x8=256 hw threads, 1.4 Mhz
  > M8000, SPARC64 VI, 12x2x2=48 hw threads, 2.28 Mhz

• Programs
  > Quick sort (memory bound)
  > Integer partitioning (CPU bound)

• Clock profiling, sampling each thread at 10ms, recording to /tmp

• Timestamps before/after the main parallel region
Integer Partitioning: OpenMP task + serial (T5440)

Graph showing the ratio of overhead collected over threads.
Quick sort: OpenMP task + serial (T5440)
Integer Partitioning: OpenMP task + serial (M8000)

- Linear
- Parallel
- Parallel+Collect

Y-axis: Speedup
X-axis: Threads

Graph shows the speedup as a function of the number of threads, with different methods compared.
Integer Partitioning: OpenMP task if (M8000)

- Serial
- Parallel
- Parallel+Collect

Time (ms)

Threads
THANK YOU

- Oleg Mazurov

oleg.mazurov@sun.com