Performance Profiling for OpenMP

Karl Fuerlinger, David Skinner
fuerling@eecs.berkeley.edu
dskinner@nersc.gov

http://www.ompp-tool.com

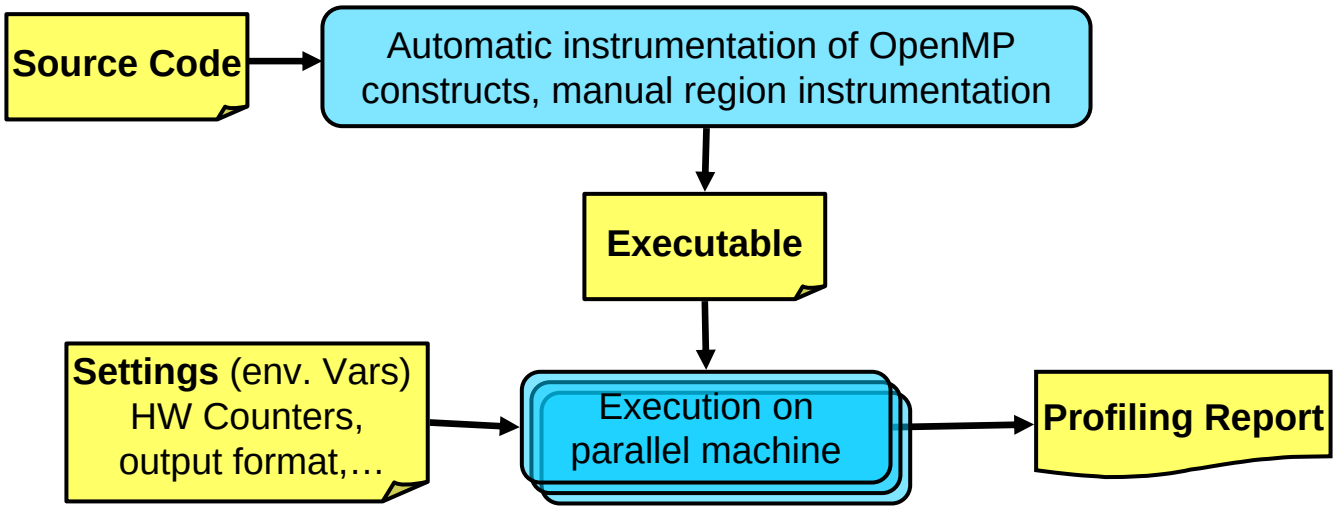
Outline

- ompP Basics:
  - Flat profiles and callgraph profiles
  - Overhead analysis

- ompP and Tasks
  - Instrumentation
  - Measurement
  - Data Analysis

- Conclusion / Future Work

OpenMP Performance Analysis with ompP



- ompP: Profiling tool for OpenMP
  - Based on source code instrumentation
  - Independent of the compiler and runtime used
  - Tested and supported: Linux, Solaris, AIX and Intel, Pathscale, PGI, IBM, gcc, SUN studio compilers
  - Supports HW counters through PAPI
  - Leverages source code instrumenter *opari* from the KOJAK/SCALASCA toolset
  - Available for download (GLP):

    http://www.ompp-tool.com

ompP's Profiling Report

- Header
  - Date, time, duration of the run, number of threads, used hardware counters,…
- Region Overview
  - Number of OpenMP regions (constructs) and their source-code locations
- Flat Region Profile
  - Inclusive times, counts, hardware counter data
- Callgraph
- Callgraph Profiles
  - With Inclusive and exclusive times
- Overhead Analysis Report
  - Four overhead categories
  - Per-parallel region breakdown
  - Absolute times and percentages

Profiling Data

- Example profiling data

Code:

```
#pragma omp parallel
{
 #pragma omp critical
 {
   sleep(1)
 }
}
```

Profile:

```
R00002 main.c (34-37) (default) CRITICAL
 TID     execT    execC     bodyT    enterT     exitT  PAPI_TOT_INS
   0      3.00        1      1.00      2.00      0.00          1595
   1      1.00        1      1.00      0.00      0.00          6347
   2      2.00        1      1.00      1.00      0.00          1595
   3      4.00        1      1.00      3.00      0.00          1595
 SUM     10.01        4      4.00      6.00      0.00         11132
```

- Components:
  - Region number
  - Source code location and region type
  - Timing data and execution counts, **depending on the particular construct**
  - One line per thread, last line sums over all threads
  - Hardware counter data (if PAPI is available and HW counters are selected)
  - Data is exact (measured, not based on sampling)

Flat Region Profile (2)

- Times and counts reported by ompP for various OpenMP constructs

	main		*enter*		*body*					*barr*	*exit*	
construct	execT	execC	enterT	startupT	bodyT	sectionT	sectionC	singleT	singleC	exitBarT	exitT	shutdwnT
MASTER	•	•										
ATOMIC	•	•										
BARRIER	•	•										
FLUSH	•	•										
USER REGION	•	•										
CRITICAL	•	•	•		•						•	
LOCK	•	•	•		•						•	
LOOP	•	•			•					•		
WORKSHARE	•	•			•					•		
SECTIONS	•	•				•	•			•		
SINGLE	•	•						•	•	•		
PARALLEL	•	•		•	•					•		•
PARALLEL LOOP	•	•		•	•					•		•
PARALLEL SECTIONS	•	•		•		•	•			•		•
PARALLEL WORKSHARE	•	•		•	•					•		•

____T: time

____C: count

Main =
enter +
body +
barr +
exit

Callgraph

- Callgraph View
  - ‘Callgraph’ or ‘region stack’ of OpenMP constructs
  - Functions can be included by using Opari’s mechanism to instrument user defined regions: #pragma pomp inst begin(…), #pragma pomp inst end(…)
- Callgraph profile
  - Similar to flat profile, but with inclusive/exclusive times
- Example:

```
main()
{
#pragma omp parallel
  {
    foo1();
    foo2();
  }
}
```

```
void foo1()
{
#pragma pomp inst begin(foo1)
   bar();
#pragma pomp inst end(foo1)
}
```

```
void foo2()
{
#pragma pomp inst begin(foo2)
   bar();
#pragma pomp inst end(foo2)
}
```

```
void bar()
{
#pragma omp critical
  {
     sleep(1.0);
  }
}
```

Callgraph (2)

- Callgraph display

```
    Incl. CPU time
  32.22 (100.0%)             [APP 4 threads]
  32.06 (99.50%)  PARALLEL    +-R00004 main.c (42-46)
  10.02 (31.10%)   USERREG       |-R00001 main.c (19-21) ('foo1')
  10.02 (31.10%)  CRITICAL       |  +-R00003 main.c (33-36) (unnamed)
  16.03 (49.74%)   USERREG       +-R00002 main.c (26-28) ('foo2')
  16.03 (49.74%)  CRITICAL          +-R00003 main.c (33-36) (unnamed)
```

- Callgraph profiles

```
[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
 TID    execT/I    execT/E      execC
   0       1.00       0.00          1
   1       3.00       0.00          1
   2       2.00       0.00          1
   3       4.00       0.00          1
 SUM      10.01       0.00          4

[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
[=03] R00003 main.c (33-36) (unnamed) CRITICAL
 TID      execT      execC    bodyT/I    bodyT/E     enterT      exitT
   0       1.00          1       1.00       1.00       0.00       0.00
   1       3.00          1       1.00       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       1.00       0.00
   3       4.00          1       1.00       1.00       3.00       0.00
 SUM      10.01          4       4.00       4.00       6.00       0.00
```

Overhead Analysis (1)

- Certain timing categories reported by ompP can be classified as overheads:
  - Example: **`exitBarT`**: time wasted by threads idling at the exit barrier of work-sharing constructs. Reason is most likely an **imbalanced** amount of work
- Four overhead categories are defined in ompP:
  - **Imbalance**: waiting time incurred due to an imbalanced amount of work in a worksharing or parallel region
  - **Synchronization**: overhead that arises due to threads having to synchronize their activity, e.g. **`barrier`** call
  - **Limited Parallelism**: idle threads due not enough parallelism being exposed by the program
  - **Thread management**: overhead for the creation and destruction of threads, and for signaling critical sections, locks as available

Overhead Analysis (2)

	main		*enter*		*body*					*barr*	*exit*	
construct	execT	execC	enterT	startupT	bodyT	sectionT	sectionC	singleT	singleC	exitBarT	exitT	shutdwnT
MASTER	•	•										
ATOMIC	•(S)	•										
BARRIER	•(S)	•										
FLUSH	•(S)	•										
USER REGION	•	•										
CRITICAL	•	•	•(S)		•						•(M)	
LOCK	•	•	•(S)		•						•(M)	
LOOP	•	•			•					•(I)		
WORKSHARE	•	•			•					•(I)		
SECTIONS	•	•				•	•			•(I/L)		
SINGLE	•	•						•	•	•(L)		
PARALLEL	•	•		•(M)	•					•(I)		•(M)
PARALLEL LOOP	•	•		•(M)	•					•(I)		•(M)
PARALLEL SECTIONS	•	•		•(M)		•	•			•(I/L)		•(M)
PARALLEL WORKSHARE	•	•		•(M)	•					•(I)		•(M)

S: Synchronization overhead

I: Imbalance overhead

M: Thread management overhead

L: Limited Parallelism overhead

ompP's Overhead Analysis Report

```
---------------------------------------------------------------------
----     ompP Overhead Analysis Report    ---------------------------
---------------------------------------------------------------------
Total runtime (wallclock)   : 172.64 sec [32 threads]
Number of parallel regions  : 12
Parallel coverage           : 134.83 sec (78.10%)

Parallel regions sorted by wallclock time:
         Type                                Location      Wallclock (%)
R00011  PARALL                    mgrid.F (360-384)      55.75 (32.29)
R00019  PARALL                    mgrid.F (403-427)      23.02 (13.34)
R00009  PARALL                    mgrid.F (204-217)      11.94 ( 6.92)
...
                                                  SUM     134.83 (78.10)

Overheads wrt. each individual parallel region:
         Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00011  1783.95   337.26 (18.91)    0.00 ( 0.00)  305.75 (17.14)    0.00 ( 0.00)   31.51 ( 1.77)
R00019   736.80   129.95 (17.64)    0.00 ( 0.00)  104.28 (14.15)    0.00 ( 0.00)   25.66 ( 3.48)
R00009   382.15   183.14 (47.92)    0.00 ( 0.00)   96.47 (25.24)    0.00 ( 0.00)   86.67 (22.68)
R00015   276.11    68.85 (24.94)    0.00 ( 0.00)   51.15 (18.52)    0.00 ( 0.00)   17.70 ( 6.41)
...

Overheads wrt. whole program:
         Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00011  1783.95   337.26 ( 6.10)    0.00 ( 0.00)  305.75 ( 5.53)    0.00 ( 0.00)   31.51 ( 0.57)
R00009   382.15   183.14 ( 3.32)    0.00 ( 0.00)   96.47 ( 1.75)    0.00 ( 0.00)   86.67 ( 1.57)
R00005   264.16   164.90 ( 2.98)    0.00 ( 0.00)   63.92 ( 1.16)    0.00 ( 0.00)  100.98 ( 1.83)
R00007   230.63   151.91 ( 2.75)    0.00 ( 0.00)   68.58 ( 1.24)    0.00 ( 0.00)   83.33 ( 1.51)
...
   SUM  4314.62  1277.89 (23.13)    0.00 ( 0.00)  872.92 (15.80)    0.00 ( 0.00)  404.97 ( 7.33)
```

Number of threads, parallel regions, parallel coverage

Wallclock time x number of threads

Overhead percentages wrt. this particular parallel region

Overhead percentages wrt. whole program

Outline

- ompP Basics:
  - Flat profiles and callgraph profiles
  - Overhead analysis
- ompP and Tasks
  - Instrumentation
  - Measurement
  - Data Analysis
- Conclusion / Future Work

Tasking and OpenMP

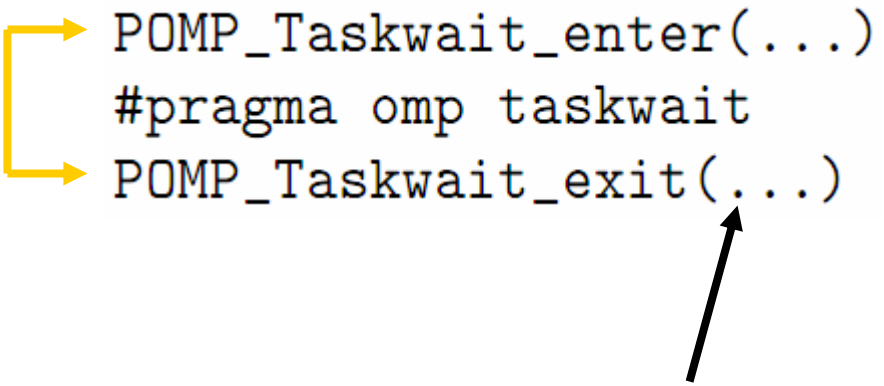
- Two new constructs:
  - **task**
  - **taskwait**
- Tasks can be **tied** (the default) or **untied**
- Threads suspend and resume execution at Task Scheduling Points (TSPs)
  - **Tied**: same thread resumes execution
  - **Untied**: resuming thread can be a different one
- TSPs
  - Tied: At a set of specified locations in tied tasks
  - Untied: TSPs can be anywhere in untied tasks

Instrumentation (1)

- Extended Opari to instrument *task* and *taskwait* constructs
  - Opari: Instrumenter from the KOJAK/SCALASCA toolset [FZ Juelich]
  - Source-to-source instrumenter, adds calls according to the POMP specification and generates region descriptors; POMP_Parallel_enter, POMP_Parallel_exit

- Taskwait:

```
#pragma omp taskwait
```



Region descriptor:
- File name, line number, ...

Instrumentation (2)

- Task

```
#pragma omp task
{
   // user's tasking code
{
```

```
POMP_Task_enter(...)
#pragma omp task
{
  POMP_Task_begin(...)
   // user's tasking code
  POMP_Task_end(...)
{
POMP_Task_exit(...)
```

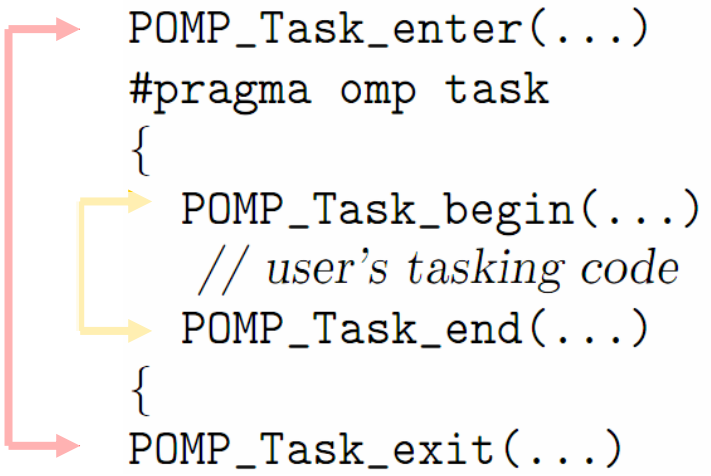
Region descriptor:
- File name, line number, ...

- Untied tasks:
  - **POMP_Utask_{enter,exit,begin,end}** instead

Measurement

- ompP implements the calls
  - **POMP_Task_{enter,exit,begin,end}**
  - **POMP_UTask_{enter,exit,begin,end}**
  - **POMP_Taskwait_{enter,exit}**

- **A task is represented by two separate data structures at runtime**
  - A region of type **TASK** for the task "definition".
  - A region of type **TASKEXEC** for the actual execution of the task
  - TASK and TASKEXEC regions show up in the callgraph and profile displays like other regions…



TASK / TASKEXEC example (1)

```
void main(int argc, char* argv[])
{
#pragma omp parallel
{
  int i;
#pragma omp single nowait
  {
  for( i=0; i<5; i++ )
    {
#pragma omp task
      {
        mytask();
      }
    }
  }
}
```

```
void mytask() {
  sleep(1);
}
```

- Region/Call-Graph:

```
PARALLEL  +-R00001
  SINGLE     |-R00002
    TASK     |  +-R00003
TASKEXEC     +-R00003
```

- TASK / TASKEXEC profiles:

R00003 main.c (22-23) TASK

TID	execT	execC
0	0.00	0
1	0.00	5
SUM	0.00	5

R00003 main.c (22-23) TASKEXEC

TID	execT	execC
0	3.00	3
1	2.00	2
SUM	5.00	5b

TASK / TASKEXEC example (2)

```
void main(int argc, char* argv[])
{
#pragma omp parallel
{
  int i;
#pragma omp single nowait
  {
  for( i=0; i<5; i++ )
    {
#pragma omp task if(0)
      {
        mytask();
      }
    }
  }
}
```

```
void mytask() {
  sleep(1);
}
```

- Region/Call-Graph:

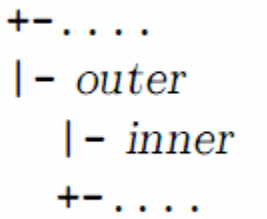
```
PARALLEL  +-R00001
  SINGLE     +-R00002
    TASK        +-R00003
TASKEXEC           +-R00003
```

(example from previous slide:)

```
PARALLEL  +-R00001
  SINGLE     |-R00002
    TASK     |  +-R00003
TASKEXEC     +-R00003
```

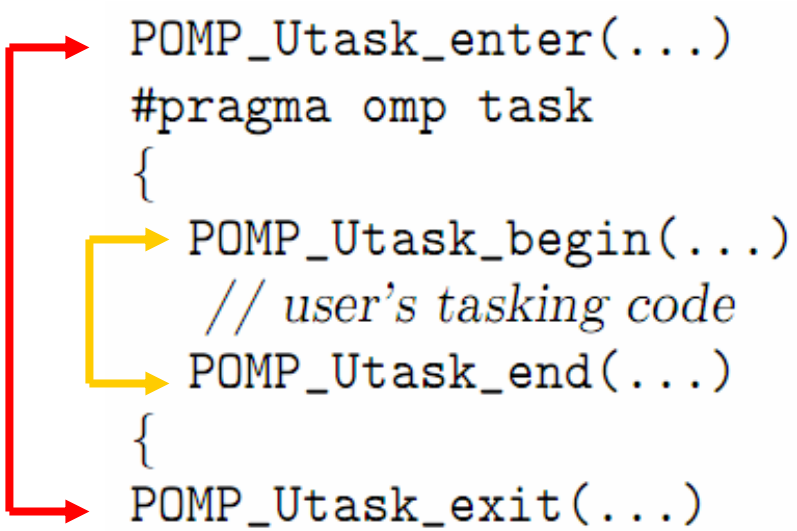
Which nestings are possible?

	inner region		
outer region	[U]TASK	TASKEXEC	TASKWAIT
[U]TASK	–	× 1	–
TASKEXEC	× 2	× 3	× 4
TASKWAIT	–	× 5	–



1. Immediate execution of the task, either because if() clause evaluates to false or because the runtime decides to do so on its own
2. Encountering another task construct while executing tasks
3. Task switching
4. Encountering a taskwait
5. Only thing that can happen during a taskwait is execution of tasks

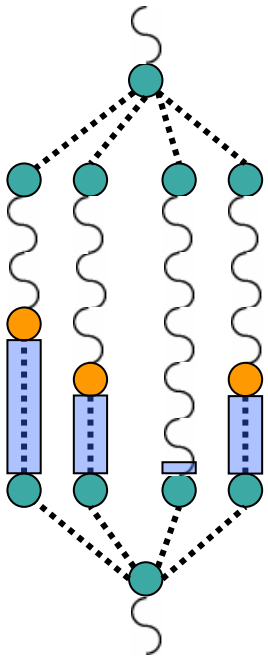
Untied Tasks



- Thread executing **begin** could be different from the one executing **end**
- No way to tell for ompP if this has happened without information from the runtime (callback)
- Not much we can do here...
  - Set **DISABLE_UNTIED** environment variable to disable monitoring of untied task entirely
  - Monitor untied tasks like tied tasks and hope for the best (the default)

Tasking Data Analysis

- Overhead analysis report:
  - Time spent in implicit exit barrier of parallel region is reported as imbalance overhead
  - Threads aren‘t doing anything useful on behalf of the application w/o tasking



- This changes with tasking:
  - Threads can go and grab tasks to execute
  - Time in exit barrier is only partially idle time
- Accounting in ompP
  - Subtract tasking time from the idle time in the exit barrier
  - This informatin is contained in the region nesting / callgraph of ompP

Example

```
Overheads wrt. each individual parallel region:
      Total     Ovhds (%) =  Synch(%) + Imbal   (%) + Limpar (%) +  Mgmt (%)
R00001 6.00 1.00 (16.68) 0.00 (0.00) 1.00 (16.66)  0.00 (0.00) 0.00 (0.02)

Overheads wrt. whole program:
      Total    Ovhds (%) =  Synch(%) + Imbal   (%) + Limpar (%) +  Mgmt (%)
R00001 6.00 1.00 (15.64) 0.00 (0.00) 1.00 (15.63)  0.00 (0.00) 0.00 (0.02)
   SUM 6.00 1.00 (15.64) 0.00 (0.00) 1.00 (15.63)  0.00 (0.00) 0.00 (0.02)
```

- Same simple code as before: 5 tasks are generated (1 second “work”) each, 2 threads execute the tasks
- Leads to an imbalance of 1 second in the containing parallel region

Example contd.

- So where did the time go?
  - Need a new timing column **taskT**

```
R00001 main.c (15-26) PARALLEL
 TID    execT    execC    bodyT   exitBarT  startupT  shutdwnT    taskT
   0     3.00        1     0.00       0.00      0.00      0.00     3.00
   1     3.00        1     0.00       1.00      0.00      0.00     2.00
 SUM     6.00        2     0.00       1.00      0.00      0.00     5.00
```

  - **taskT records time spent executing tasks while in the implicit exit barrier**

Conclusion

- First stab at implementing tasking support in the OpenMP profiling tool
  - Support for untied tasks is incomplete and it is unclear how it can be improved in an
  - Need to leverage call-back mechanism from the runtime to notify us when the
  - SUN whitepaper extensions?

- Future work:
  - Test on actual applications w/ tasking
  - Integration with SUN API

- Available for Download (Tasking support in upcoming release)
  - www.ompp-tool.com

Thank you for your attention!