ompP 0.8.0 — MADMON v1.0
OpenMP Profiler
User Guide and Manual

Karl Fuerlinger
Ludwig-Maximilians-Universität München
Institut für Informatik, MNM-Team
http://www.karlfuerlinger.com

Project Webpage:
http://www.ompp-tool.com

December 2012

Contents

1 Introduction **2**

2 Installation **2**

3 Usage **2**

3.1 Instrumenting and Linking Applications with ompP 2
3.1.1 Instrumenting User-Defined Regions 3
3.1.2 Explicit Measurement Initialization 3
3.2 Running Applications 4
3.2.1 Disable the collection of performance data for certain types of OpenMP constructs 4
3.2.2 Selecting the Output Format 4
3.2.3 Specifying the Name of the Report File 5
3.2.4 Disable Output 5
3.2.5 Using Hardware Counters with ompP 5
3.2.6 Using evaluators with ompP 6
3.2.7 Incremental Profiling 6

4 The Contents of ompP's Profiling Report **7**
4.1 General Information 8
4.2 Region Overview 9
4.3 Callgraph 10
4.4 Flat Region Profiles 11
4.5 Callgraph Region Profiles 14
4.6 Overhead Analysis Report 15
4.7 Performance Properties Report 17

5 Analyzing ompP's Profiling Reports **19**

6 Known Issues **19**

7 Future Work **20**

1 Introduction

ompP is a profiling tool for OpenMP applications written in C/C++ or FORTRAN. ompP's profiling report becomes available immediately after program termination in a human-readable format. ompP supports the measurement of hardware performance counters using PAPI [1] and supports several advanced productivity features such as overhead analysis and detection of common inefficiency situations (performance properties).

2 Installation

Please see the file "INSTALL" for detailed instructions on how to install ompP on your system.

3 Usage

3.1 Instrumenting and Linking Applications with ompP

ompP is implemented as a static library that is linked to your application. To capture OpenMP execution events, ompP relies on Opari [3] for source-to-source instrumentation. A helper script `kinst-ompp` or `kinst-ompp-papi`[1] is included that hides the details of invoking Opari from the user. To instrument your application with Opari and link it with ompP's monitoring library simply prefix any compile or link command with `kinst-ompp` or `kinst-ompp-papi`. I.e., on a shell prompt:

[1]Based on similar scripts from the SCALASCA and KOJAK packages, courtesy of Bernd Mohr, FZ Juelich

```
$> icc -openmp foo.c bar.c -o myapp
```

becomes

```
$> kinst-ompp icc -openmp foo.c bar.c -o myapp
```

Similarly, to use ompP with Makefiles, simply replace the compiler specification like `CC=icc` with `CC=kinst-ompp icc`.

3.1.1 Instrumenting User-Defined Regions

User-defined regions (both on a block level as well as whole functions) can be instrumented with Opari's `pragma` handling mechanism. The pragma `#pragma pomp inst begin(name)` marks the begin of the region and `#pragma pomp inst end(name)` marks its end. For example:

```
int foo() {
#pragma pomp inst begin(foo)
  ...
#pragma pomp inst end(foo)
  return 1;
}
```

To mark an alternative exit point of a region (such as an addtional return statement) use the `#pragma pomp inst altend(name)` pragma. For FORTRAN the syntax is:

```
!$POMP INST BEGIN(name)
...
  [ !$POMP INST ALTEND(name) ]
...
!$POMP INST END(name)
```

3.1.2 Explicit Measurement Initialization

ompP will per default start monitoring when the first call to an OpenMP construct (or user-instrumented region) is made. To explicitly start monitoring (typically in `main()`), use the Opari `#pragma pomp inst init` pragma. This is especially useful for programs that do a significant amount of sequential work before entering the parallel regions. Placing `#pragma pomp inst init` at the beginning of `main()` will guarantee that some timing results (such as the parallel coverage, see Sect. 4.6) will be reported correctly by ompP in this case.

3.2 Running Applications

Invoke the instrumented OpenMP application like a regular OpenMP application. Make sure `OMP_NUM_THREADS` is set to the desired number of OpenMP threads. For an application called `myapp`, ompP will per default write the profiling report to `myapp.`n`-`m`.ompp.txt`, where n is the number of threads used and m is a consecutive number starting with 0. Consecutive numbering is used because ompP does not per default overwrite existing profiling reports. See Sect. 4 for the contents of ompP's profiling report.
The following Environment variables influence the execution of the monitored application:

3.2.1 Disable the collection of performance data for certain types of OpenMP constructs

By setting the environment variable `OMPP_DISABLE_`*`construct`*, the collection of performance data for this type of construct is disabled. Example:

```
$> export OMPP_DISABLE_ATOMIC=1
```

or

```
$> export OMPP_DISABLE_ATOMIC=yes
```

This will disable the collection of performance data for atomic constructs (both examples are for the bash shell). To re-enable data collection either un-set the environment variable or set it to 0. I.e,,

```
$> unset OMPP_DISABLE_ATOMIC
```

or

```
$> export OMPP_DISABLE_ATOMIC=0
```

Constructs which can be disabled are: `ATOMIC`, `BARRIER`, `CRITICAL`, `FLUSH`, `LOOP`, `MASTER`, `SECTIONS`, `SINGLE`, `WORKSHARE`, `USER_REGION`, and `LOCK`. [2]

3.2.2 Selecting the Output Format

Use the environment variable `OMPP_OUTFORMAT` to select the format of ompP's report. Two formats are currently specified plaintext ASCII (default) and comma separated values (CSV). To select output specify CSV or csv as the value of the `OMPP_OUTFORMAT` variable, e.g.:

```
$> export OMPP_OUTFORMAT=CSV
```

```
$> export OMPP_OUTFORMAT=csv
```

If a different string is specified or `OMPP_OUTFORMAT` is not specified at all, plaintext ASCII will be generated.

[2] Instead of specifying `OMPP_DISABLE_LOCK` you can also specify `OMPP_DISABLE_LOCKS` (plural form).

3.2.3 Specifying the Name of the Report File

`OMPP_APPNAME` can be used to specify the name of the target application. ompP uses the application name to infer the name of the report file (e.g., `myapp.`n-m`.ompp.txt`). Sometimes ompP fails to derive the correct name of the target application, e.g., when the application is not invoked directly but via another command like for example `dplace`. Use `OMPP_APPNAME` to specify the applications name in such cases.
`OMPP_OUTDIR` can be used the specify the directory in which to place the report files.
`OMPP_OVERWRITE` can be specified to overwrite existing report files instead of incrementally numbering them `report-1.ompp.txt` `report-2.ompp.txt`, etc.
If `OMPP_REPORTFILE` is set, this will be used as the name of the report file.

3.2.4 Disable Output

To force ompP to give no warning, error or status messages set the environment variable `OMPP_QUIET` to any value not equal to 0. No message will be given on output on `stdout` or `stderr`.

3.2.5 Using Hardware Counters with ompP

Hardware counters can be used with ompP by setting the environment variables `OMPP_CTR`n to the names of PAPI predefined or platform-specific event names. For example:

```
$> export OMPP_CTR1=PAPI_L2_DCM
```

The number of hardware counters that can be recorded simultaneously by ompP is a compile time constant set to 4 per default, see the definition of `OMPP_PAPI_MAX_CTRS` in file `ompp.h` if you want to increase this limit.
During startup ompP will display a message whether registering the specified counter(s) was successful:

```
ompP: successfully registered counter PAPI_L2_DCM
```

If the specified event name(s) are either not recognized or cannot be counted together, ompP will issue a warning:

```
ompP: PAPI name-to-code error
```

for an unrecognized event name, or:

```
ompP: Error adding event to eventset
```

for conflicting events that cannot be counted together by the underlying hardware.

3.2.6 Using evaluators with ompP

Evaluators are a convenience feature of ompP to transform hardware performance counter based data into more readable forms directly in the profiling tool. An *evaluator* is an arithmetic formula in string form that can involve hardware counter data. For example:

```
$> OMPP_EVAL1=PAPI_FP_OPS/1000000  # compute the megaflop rate
$> OMPP_EVAL1=1-L2_MISSES/L2_REFERENCES # L2 hit rate (Itanium)
$> OMPP_EVAL1=1-(L3_MISSES-L3_WRITES_L2_WB_MISS)/
             (L3_REFERENCES-L3_WRITES_L2_WB_ALL) # L3 hit rate (Itanium)
```

ompP will extract hardware counter names from evaluator strings and program PAPI to collect the necessary data. In addition to PAPI event counter names, evaluators can contain numeric constants and they can contain references to `EXECT` and `EXECC`. Those two special variables denote the execution time and counts for the region for which hardware counter data was acquired.
ompP uses `libmatheval`[3] to evaluate the numeric values of the evaluator strings and those values appear as additional columns in the profiling report similar to plain hardware counters (see Sects. 4.4 and 4.5).

3.2.7 Incremental Profiling

Incremental profiling refers to the method of continuously capturing profiling reports while the program is running. While pure ("one-shot") profiling does usually not allow one to uncover and explain reason and temporal relationship of performance phenomena, this is possible with full event tracing and to some extent also with incremental profiling. Hence, incremental profiling can be a good compromise between full tracing and pure one-shot profiling as it usually is less intrusive and generates smaller amounts of performance data. Incremental profiling is enabled in ompP by setting the environment variable `OMPP_DUMP_-INTERVAL` to the desired duration (in seconds) between capture points. The duration is given in seconds and must be at least 0.1 seconds, shorter intervals would likely cause too much overhead to produce usable results.

```
$> export OMPP_DUMP_INTERVAL=1
```

The profiling reports are delivered in the same format as normal ompP profiling reports (see Sect. 4). The data contained in a profiling dump at time x contains performance data as it is available at time x. Since ompP's region statistics are always updated when regions are exited, the incremental profiling reports always only contains the fully executed region instances.

[3] `http://www.gnu.org/software/libmatheval/`

4 The Contents of ompP's Profiling Report

ompP's profiling report contains the following parts which will be discussed in detailed in this section.

- General Information (Header)
- Region Overview
- Callgraph
- Flat Region Profile
- Callgraph Region Profiles
- Overhead Analysis Report
- Performance Properties Report

4.1 General Information

Example:

```
---------------------------------------------------------------------
----     ompP General Information     ---------------------------------
---------------------------------------------------------------------
Start Date      : Tue Apr 17 18:45:57 2007
End Date        : Tue Apr 17 18:46:13 2007
Duration        : 16.46 sec
User Time       : 15.56 sec
System Time     : 0.63 sec
Max Threads     : 4
ompP Version    : 0.6.0
ompP Build Date : Apr 17 2007 18:36:24
PAPI Support    : available
Max Counters    : 4
PAPI Active     : yes
Used Counters   : 2
OMPP_CTR1       : PAPI_L2_DCM
OMPP_CTR2       : not set
OMPP_CTR3       : PAPI_TOT_INS
OMPP_CTR4       : not set
Max Evaluators  : 4
Used Evaluators : 0
OMPP_EVAL1      : not set
OMPP_EVAL2      : not set
OMPP_EVAL3      : not set
OMPP_EVAL4      : not set
```

This section of the profiling report lists general data about the profiling report. This includes the times of the start and end of the program run and its duration (in seconds wallclock time). The number of threads used for the execution, the date when the ompP library was built and the ompP version used. The ompP version string is represented as three numbers (major.minor.revision).
The "PAPI Support" line indicates if ompP was built with PAPI support "available" or not "not available". If PAPI support is present, the header also contains information about the used counters (if any). If no HW counters are used, the PAPI Active line will be "no", otherwise it will be "yes" and the reset of the header will show how many and which counters are used. The Max Counters lists the maximal number of counters supported by ompP. This is a compile-time constant and can be changed by adapting the definition of `OMPP_PAPI_MAX_CTRS` in file `ompp.h` of the source code distribution. For the usage of evaluators please see Sect. 3.2.6.

4.2 Region Overview

```
------------------------------------------------------------------------
----     ompP Region Overview     ----------------------------------
------------------------------------------------------------------------
PARALLEL: 7 regions:
 * R00016 zcopy.F (46-78)
 * R00007 muldoe.F (63-145)
 * R00004 muldeo.F (63-145)
 * R00013 zaxpy.F (48-81)
 * R00001 dznrm2.F (109-145)
 * R00019 zdotc.F (50-82)
 * R00022 zscal.F (42-69)

PARALLEL LOOP: 3 regions:
 * R00010 rndcnf.F (48-52)
...
```

This section lists all OpenMP regions identified by ompP. An ompP region is the lexical extent of an OpenMP language construct. All OpenMP constructs are supported by ompP and their accompanying regions are represented by region identifiers (e.g., R00016). The region overview lists the different region types in the report, gives their region identifies with their source code location. For most ompP regions the source code location is given as a file name and two line numbers (begin and end). For barrier regions the location is specified by only one number. Locks are identified by their memory address.

4.3 Callgraph

```
------------------------------------------------------------------------
----     ompP Callgraph     ------------------------------------------
------------------------------------------------------------------------

  Inclusive  (%)   Exclusive  (%)
  16.46 (100.0%)    5.45 (33.08%)           [310.wupwise: 4 threads]
   0.33 ( 2.00%)    0.33 ( 2.00%)  PARLOOP  |-R00010 rndcnf.F (48-52)
   0.65 ( 3.95%)    0.65 ( 3.95%)  PARLOOP  |-R00012 uinith.F (77-114)
   0.12 ( 0.74%)    0.12 ( 0.74%)  PARLOOP  |-R00011 rndphi.F (50-54)
   0.60 ( 3.68%)    0.00 ( 0.01%) PARALLEL  |-R00016 zcopy.F (46-78)
   0.06 ( 0.37%)    0.06 ( 0.37%)     LOOP  |  |-R00018 zcopy.F (72-76)
   0.54 ( 3.30%)    0.54 ( 3.30%)     LOOP  |  +-R00017 zcopy.F (53-57)
   4.09 (24.87%)    0.00 ( 0.01%) PARALLEL  |-R00007 muldoe.F (63-145)
   1.83 (11.13%)    1.83 (11.13%)     LOOP  |  |-R00008 muldoe.F (68-102)
...
```

The callgraph view of ompP's profiling report shows the callgraph or (-tree) of the execution of the application profiled by ompP. Note that the callgraph only shows the OpenMP regions visible to ompP and does not contain normal user functions unless the user manually instruments the functions (see Sect. 3.1.1).
The right part of the callgraph shows a graphical representation of child–parent relationships of the callgraph. Regions are identified by their region number and the source code location of the corresponding OpenMP construct. The root of the tree is implicitly the entire application. It is often advisable to manually instrument the `main()` function of an application (see Sect. 3.1.1). If this is done, the user region corresponding to `main()` will appear as the one and only child of the application and the other regions will appear below `main()`.
The left part of the callgraph view shows inclusive and exclusive times spent in each of the regions. Inclusive means the sum of the current region and all of its children, while exclusive is only the time of the current region (this is often called "self" time). The times reported in this section are "sequential" in the sense that, for example for a critical section the time listed is the summed execution time over all threads divided by the number of threads. This makes the execution times of the parallel execution comparable to the wallclock execution time of the total run. All percentages are computed with respect the wallclock duration of the entire run also.
Note, finally, that the shown callgraph is the "union" of all callgraphs encountered by OpenMP threads. That is, in principle OpenMP threads can execute independent constructs (of course synchronizing at implied synchronization points). The callgraph shown by ompP is the union of all individual callgraphs. Each node of the callgraph was visited by at least one thread, possibly more.

4.4 Flat Region Profiles

```
------------------------------------------------------------------------
----     ompP Flat Region Profile (inclusive data)     ----------------
------------------------------------------------------------------------
R00010 rndcnf.F (48-52) PARALLEL LOOP
 TID      execT      execC      bodyT   exitBarT   startupT   shutdwnT
   0       0.33          1       0.33       0.00       0.00       0.00
   1       0.33          1       0.32       0.01       0.00       0.00
   2       0.33          1       0.33       0.00       0.00       0.00
   3       0.33          1       0.32       0.01       0.00       0.00
 SUM       1.32          4       1.29       0.02       0.01       0.00
...
```

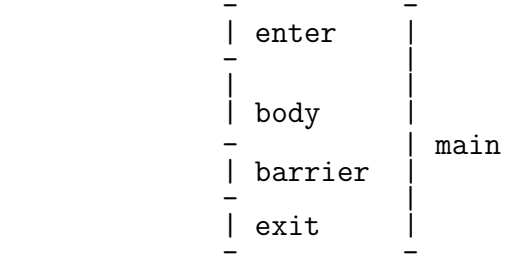
This section lists flat profiles for each OpenMP construct in a per-region basis. The profiles in this section are flat. I.e., the times and counts reported are those incurred for the particular construct, irrespective of how the construct was executed. For example, imagine a critical section `c` in a function `foo()`. If `foo()` is called from two different parallel regions `r1` and `r2`, the flat profile for `c` will show summary data for both calls. I.e., it is not possible to distinguish between the calls from `r1` or `r2` in this view. The callgraph region profiles section of ompP's profiling report offers the ability to distinguish data based on the callgraph (see Sect. 4.3).
The first line the flat region profile lists the region identifier and the source code location and region type. The next line is the header for the data entries, then data appears on a per-thread basis, the thread IDs correspond to those delivered by OpenMP's `omp_get_thread_num()` call.
The columns reported by ompP depend on the type of the OpenMP region. Timing entries end with a capital T (e.g., `execT`) while counts end with a capital C (e.g., `execC`). The table below lists which timing and count categories are reported for the various OpenMP

constructs.

(construct)	main	enter			body						exit	
	execT	execC	enter	startup	bodyT	sectionT	sectionC	singleT	singleC	exitBarT	exitT	shutdownT
MASTER	*C	*										
ATOMIC	*	*										
BARRIER	*	*										
FLUSH	*	*										
USER_REGION	*C	*										
CRITICAL	*	*	*		*C						*	
LOCK	*	*	*		*C						*	
LOOP	*	*			*C					*		
WORKSHARE	*	*			*C					*		
SECTIONS	*	*				*C	*			*		
SINGLE	*	*						*C	*	*		
PARALLEL	*	*		*	*C					*		*
PARALLEL_LOOP	*	*		*	*C					*		*
PARALLEL_SECTIONS	*	*		*		*C	*			*		*
PARALLEL_WORKSHARE	*	*		*	*C					*		*

This table shows which timing categories and counts are reported for the various OpenMP constructs. The 'C' indicates for which sub-region hardware counter data is reported (if any).

As a general rule, each region is thought of being composed of smaller sub-regions and this corresponds to the times and counts reported the composition as follows:



I.e., main corresponds to the whole construct (e.g., `zcopy.F (46-78)`) and contains all other sub-regions. The other sub-regions are not nested and do not overlap, hence main

= enter + body + barrier + exit.
The `enter` subregion corresponds to entering or starting a construct. For critical sections this is the time required to enter the sections and for parallel regions and combined worksharing-parallel regions this is the time for thread startup.
Similarly, the `exit` subregion corresponds to leaving a construct (signalling the critical section or lock as available) or for shutting down threads.
The `barrier` subregion is present for worksharing regions and represents the time waiting at implicit exit barriers of those worksharing constructs (unless a `nowait` clause is specified)
The `body` is the part of the construct where the actual work gets done. Some constructs such as `USER_REGION` do not have enter, barrier or exit sub-regions. In this case only main is reported.
Finally, the counts and times for each sub-region are reported with different names to reflect their meaning for different OpenMP constructs. For example, the time in the enter subregion is reported as `enterT` for locks and critical sections but as `startupT` for parallel regions. Please refer to the table above for a `detailed` list of reported times and counts for all OpenMP constructs.
Also note that the times and counts are always reported left to right such that main is followed by body, followed by barrier, followed by enter, and exit. `execT` and `execC` are reported for any construct and the time reported by `execT` should always be the sum of all other times reported for all threads...
If hardware counter events are selected for monitoring (see Sect. 3.2.5), the hardware counter data appears in the form of additional columns in the profiling report:

```
R00018 zcopy.F (72-76) LOOP
 TID      execT              PAPI_L2_DCM       PAPI_TOT_INS
   0       0.06   (other              82         15.736.041
   1       0.06    columns            74         15.736.049
   2       0.06    skipped)           93         15.736.074
   3       0.06                      108         15.736.080
 SUM       0.25                      357         62.944.244
```

The counter are displayed in the order in which they are specified (`OMPP_CTR1` to `OMPP_CTR4`, for example), unset counters are skipped. Note that for easier human interpretation the numbers are grouped by thousands, millions, etc. This grouping is not performed in the CSV (comma separated values) output format (see Sect. 3.2.2).
ompP always collects counter data for the sub-region where the actual work is performed (i.e., the region that contains user-code). Some constructs do not actually contain user code, such as the `barrier` construct. For other regions it would not make sense to acquire counters for performance reasons, an example for this is the `atomic` construct. The table above marks the sub-region for which hardware counter data is reported with a `C`.

4.5 Callgraph Region Profiles

```
[*00] 310.wupwise
[+01] R00016 zcopy.F (46-78) PARALLEL
[=02] R00017 zcopy.F (53-57) LOOP
 TID      execT      execC    bodyT/I    bodyT/E   exitBarT
   0       0.54          9       0.53       0.53       0.02
   1       0.54          9       0.53       0.53       0.01
   2       0.54          9       0.54       0.54       0.01
   3       0.54          9       0.53       0.53       0.01
 SUM       2.17         36       2.12       2.12       0.05
```

The data displayed in this section is largely similar to the flat region profiles. However, the data displayed represents the summed execution times and counts only of the current execution graph.
The path of the root of the callgraph to the current region is shown as the first lines for each region. The lines have the format `[`*s*`xy]`, where `xy` denotes the level in the hierarchy, starting with 0 (the root) and the symbol *s* has the following meaning: `*` stands for root of the callgraph `+` denotes that this entry has children in the call-graph, while `=` denotes that this region has no child entries in the callgraph (it is a leaf of the callgraph).
The data entries displayed for callgraph region profiles are similar to the ones shown for flat profiles. However, for selected columns both inclusive and exclusive data entries are displayed. Inclusive data represents this region and all descendants, while exclusive data excludes any descendants. In the example shown above the data is displayed for a leaf node and hence inclusive and exclusive times for `bodyT` are the same.
Hardware counter data is handled similar to timing data, i.e., a `/I` or a `/E` is appended to the counter name, for example `PAPI_L2_DCM/I` and `PAPI_L2_DCM/E`.

4.6 Overhead Analysis Report

```
---------------------------------------------------------------------
----     ompP Overhead Analysis Report     --------------------------
---------------------------------------------------------------------
Total runtime (wallclock)   : 16.38 sec [4 threads]
Number of parallel regions  : 10
Parallel coverage           : 10.93 sec (66.73%)

Parallel regions sorted by wallclock time:
            Type                              Location       Wallclock (%)
R00004  PARALLEL                     muldeo.F (63-145)       4.01 (24.45)
R00007  PARALLEL                     muldoe.F (63-145)       3.99 (24.34)
R00013  PARALLEL                       zaxpy.F (48-81)       0.70 ( 4.30)
R00012   PARLOOP                     uinith.F (77-114)       0.66 ( 4.03)
R00016  PARALLEL                       zcopy.F (46-78)       0.61 ( 3.71)
...

Overheads wrt. each individual parallel region:
       Total     Ovhds (%) = Synch (%)+ Imbal  (%)+ Limpar (%)+   Mgmt (%)
R00004 16.02  0.16 ( 1.02) 0.00 (0.00) 0.16 (1.01) 0.00 (0.00) 0.00 (0.01)
R00007 15.95  0.22 ( 1.40) 0.00 (0.00) 0.22 (1.39) 0.00 (0.00) 0.00 (0.01)
...

Overheads wrt. whole program:
       Total     Ovhds (%) = Synch (%)+ Imbal  (%)+ Limpar (%)+   Mgmt (%)
R00007 15.95  0.22 ( 0.34) 0.00 (0.00) 0.22 (0.34) 0.00 (0.00) 0.00 (0.00)
R00004 16.02  0.16 ( 0.25) 0.00 (0.00) 0.16 (0.25) 0.00 (0.00) 0.00 (0.00)
...
```

The overhead analysis report offers various interesting insights into where an application wastefully spends its time. The first part of the overhead analysis report shows the total runtime (this corresponds to the duration report of overview section of the profiling report (Sect. 4.2). Then the total number of parallel regions is reported (this includes combined worksharing-parallel regions) and then the parallel coverage is listed. The parallel coverage is the amount of execution time spent in parallel regions. A low parallel coverage limits the speedup that can be achieved by parallel execution.
The next part of the overhead analysis report lists all parallel regions, sorted by their wallclock execution time. Again, both parallel regions and combined worksharing-parallel regions are included here. The list is sorted with decreasing wallclock execution time. The percentage gives the amount of total time spent in a particular parallel region.
The following two sections lists overheads according to a classification scheme described in detail in [2]. The difference between these two sections is the order in which parallel regions

are listed and the way in which percentages are computed. The absolute times reported are the same for both sections. In the "Overheads wrt. each individual parallel region" section, the parallel regions appear in the same order as in the "Parallel regions sorted by wallclock time" part (i.e., with decreasing overall execution time). The first column `Total` is the wallclock time times the number of threads used in the parallel region and hence corresponds to the overall *totally consumed execution time* of the parallel region. The listed overheads are similarly summed over all threads. Four overhead category are distinguished (synchronization, imbalance, limited parallelism and thread management). The `Ovhds` column is the sum of all the other four overhead categories and all percentages are computed with respect to each individual parallel region. That is, in the shown example, R00004 the summed execution time over all threads is 16.02 seconds, and of this 0.16 seconds (again summed over all threads) is spent in some overheads. This is a percentage of 1.02 lost execution time.
The "Overheads wrt. whole program" sections lists the same overhead times but the percentages are computed with respect to the total execution time (i.e., duration x number of threads) in this section. Also, the regions are sorted by their overheads in this part. Hence this section allows the easy identification of those regions that cause most overheads. In the example, `R00007` contributes most overheads to this application's execution with

0.22 seconds (or 0.34 percent of total execution time).

```
Overhead classification scheme of ompP:
 - M is thread managemen overhead
 - S is synchronization overhead
 - L is limited parallelism overhead
 - I is imbalance overhead
```

(construct)	main		enter		body					exit	
	execT	exCT	entT	startupT	bodyT	sectionT	seciontC	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

4.7 Performance Properties Report

```
------------------------------------------------------------------
----     ompP Performance Properties Report     ----------------
------------------------------------------------------------------
```

```
Property P00001 'ImbalanceInParallelLoop' holds for
     'LOOP muldoe.F (68-102)', with a severity (in percent) of 0.1991

...
```

This section reports so-called "performance properties" that are detected automatically for the application. Performance properties capture common situations of inefficient execution, they are based on the profiling data that is reported for each region.
Properties have a name (`ImbalanceInParallelLoop`) and a context for which they have been detected (`LOOP muldoe.F (68-102)`). Each property carries a severity value, which represents the negative impact on overall performance a property exhibits. The severity value is given in percentage of total accumulated execution time (over all threads) and the properties are reported with decreasing severity values.
The code that checks the for the `ImbalanceInParallelLoop`' property sums the `exitBarT` for all threads and the severity is this given by the ratio of this sum to the total execution time of the application. Hence it detects situations with imbalanced (wrt. threads) amounts of work. Here is a list of all properties defined for ompP:

`WaitAtBarrier`

This property checks for idle threads at explicit (programmer-added barriers)

`ImbalanceInParallelRegion`
`ImbalanceInParallelLoop`
`ImbalanceInParallelWorkshare`
`ImbalanceInParallelSections`

Those properties check for situations of imbalanced work in parallel regions and in worksharing regions.

`ImbalanceDueToNotEnoughSections`,
`ImbalanceDueToUnevenSectionDistribution`,

Those two properties try to uncover the reason for a situation of imbalanced execution in a Sections construct more precisely. `ImbalanceDueToNotEnoughSections` is detected when the number of individual section constructs in an enclosing sections construct is not sufficient to provide all threads with work. Conversely, `ImbalanceDueToUneven-SectionDistribution` is detected if there are enough sections but they can not be evenly distributed among all threads.

`CriticalSectionContention`
`LockContention`

Those properties check for contention for entering critical sections or for acquiring locks. They are based ont he `enterT` of the corresponding profiling data structures.

`FrequentAtomic`

This property is detected if an atomic construct is executed with a frequency higher than a predefined threshold

`InsufficienWorkInParallelLoop`

This property checks the amount of work (i.e., execution time) that is performed in a parallel region. Usually the amount of work should be big enough to amortize the cost of spawning threads. The `InsufficienWorkInParallelLoop` property warns if this not the case.

`UnparallelizedInMasterRegion`
`UnparallelizedInSingleRegion`

Those two properties detect situations of serialized execution in master and single regions respectively. In a single region, only one thread executes code, other threads have to wait at the exit barrier unless a `nowait` is specified. For master the situation is somewhat different. Only the master thread executes the master region but there is no synchronization point implied at the end of worksharing threads. Hence it is not know if the other threads are idle or performing useful work while the master thread executes inside the master construct. Hence this property does not actually point out an actual inefficiency situation but merely points to potential case for such a case.

5 Analyzing ompP's Profiling Reports

A set of utilities (perl scripts) will be included in a forthcoming releas to allow for easier analysis and visualization of ompP's profiling reports.

6 Known Issues

Pathscale C/C++ compiler incompatibility with expanding preprocessor macros in OpenMP pragmas:

```
user@host$ kinst-ompp pathcc -mp -o test c_simple.c
c_simple.c: In function 'main':
c_simple.c:7: error: expected '#pragma omp' clause before 'POMP_DLIST_00001'
```

Workaround: Invoke the `kinst-ompp` script as

```
user@host$ kinst-ompp -nodecl -- pathcc -mp -o test c_simple.c
```

instead. `-nodecl` is an option to `kinst-ompp` to not use preprocessor definitions and `--` signifies the end of options to kinst-ompp.

7 Future Work

- Support for other additional output formats, e.g., XML.
- Write converter of profiling reports into the CUBE format.
- Allow named evaluators, for example `missrate=L2_MISSES/MEM_REFERENCES`, use name of evaluator as the column head then.
- Instead of displaying inclusive and exclusive data in callgraph region profiles, display data differentiating into self / descendants (children). This might make manual reasoning easier. Example `bodyT/S` and `bodyT/C`.

References

[1] Shirley Browne, Jack Dongarra, N. Garner, G. Ho, and Philip J. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[2] Karl Fürlinger and Michael Gerndt. Analyzing overheads and scalability characteristics of OpenMP applications. In *Proceedings of the Seventh International Meeting on High Performance Computing for Computational Science (VECPAR'06)*, pages 39–51, Rio de Janeiro, Brasil, 2006. LNCS 4395.

[3] Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, September 2001.