PerfRegions Documentation

Anna Mittermair, Silvia Mocavero, Andrew Porter, Martin Schreiber

July 7, 2025

`(Please add yourself to the list of authors in alphabetical order if you contributed something to PerfRegio`

Abstract

Gaining understanding in performance limitations is known to be a very challenging job. Even with the arise of automized commercial tools such as Intel Amplifier, or open source tools such as HPCToolkit, this still does not allow to get highly accurate statistics on the performance of selected areas of the codes from scientific computing area and to guarantee portability on several architectures. An alternative is to extend each program by hand with code which allows gaining insight into the performance, however this is very time consuming as well as error prone.

This document describes our development which is called PerfRegions which suggests an annotation of the program. This reveals detailed information on selected regions on the code and significantly reduces the time until insight is gained into the performance.

1 Targets

We focus on the following targets

1. **Language flexibility**: Supporting C and Fortran code
2. **Programmability**: The suggested code annotation should be easy to use
3. **Portability**: The only requirement is the PAPI library installed
4. **MPI support**: The development should support MPI-based parallelization and accumulation of the results
5. **All-in-one information**: The tool should be able to measure accurate timings as well as performance counters.
6. **No recompilation**: Since the number of performance counters to use is limited, changing the performance counters to be measured should not require recompilation.

2 Realization

The package includes 2 main folders:

1. the *src* folder, which contains the source code of perf regions. In particular, the folder includes (i) the header files, (ii) the source code for the performance measurement init and finalize, and for the region measurement start and stop and (iii) the interfaces to the PAPI libraries.
2. the *example* folder, which includes two example folders (C and Fortran examples).

3 Language extensions

The language extensions should be kept very flexible. This allows e.g. replacing existing annotations of sections/regions with the annotations of PerfRegions. This was in particular of our interest since this project was originally developed to replace timing constructs in the NEMO development in order to gain per-region information on hardware performance counters.

3.1 Example code in C

Due to the flexibility of the language extensions, we'd like to give a concrete C-code example:

```
#include <stdio.h>
#include <stdlib.h>

#pragma perf_regions include

double *a;
int size;

int main()
{
#pragma perf_regions init
        run_computations();
        return 0;
#pragma perf_regions finalize
}

void nested_region()
{
#pragma perf_regions start rec
    [... foo regions computations ...]
#pragma perf_regions stop rec
}

void run_computations()
{
  for (int k = 0; k < iters; k++)
  {
#pragma perf_regions start foo
    [... foo region computations ...]
    nested_region();
    nested_region();
#pragma perf_regions stop foo

#pragma perf_regions start bar
    [... bar region computations ...]
#pragma perf_regions stop bar
  }
}
[...]
```

In this version, we use #pragma as a marker for the preprocessor to identify which parts of the code to replace with the PerfRegion code. Since the preprocessor uses regular expressions to detect such regions, this can be basically any other language extension. More information is provided in the next section.

3.2 PerfRegion C-language extensions

Using PerfRegion on C-code, the program annotation is given by

```
#pragma perf_regions [identifier] [name]
```

The following table gives an overview of the language extensions:

Identifier	Description
include	This construct is replaced with PerfRegion header files
init	Initialize the PerfRegion library
init_mpi [communicator]	Initialize the PerfRegion library, and set the MPI communicator. This means results will be accumulated over all the communicator's ranks at finalize.
finalize	Finalize the PerfRegion and *output* a summary of the measured performance
start [name]	Annotation of the start of a region to run performance measurements. 'name' has to be a unique identifier.
stop [name]	Annotation of the end of a region to run performance measurements.'name' has to be a unique identifier and has to match to 'name' at the previous start annotation.

3.3 Fortran support

The Fortran support currently only supports replacing the timing constructs in the NEMO development. Since this shows the flexibility of PerfRegions, an example is given as follows:

```
 [...]
PROGRAM main
     !pragma perf_regions include
     CALL timing_init()
          ! or alternatively: CALL timing_init_mpi(MPI_COMM_WORLD)
     call test1
     CALL timing_finalize()
END PROGRAM main

SUBROUTINE test1
     !pragma perf_regions include
     CALL timing_start('FOOa')
     CALL timing_start('FOOb')
     CALL test2
     CALL timing_stop('FOOb');
     CALL timing_stop('FOOa');
end SUBROUTINE test1
 [...]
```

4 Using PerfRegions

4.1 Compiling of PerfRegions

PerfRegions can be compiled in it's main folder by typing 'make'.

4.1.1 Release mode

The release mode is automatically used if compiling with 'make'

```
$ make MODE=release
```

4.1.2 Debug mode

This mode should be used to test the PerfRegions library. This includes certain validation checks which reduced the number of potential bugs.

```
$ make MODE=debug
```

4.1.3 MPI support

When using MPI you can track the wallclocktime for every MPI process individually and in the end accumulate the results for all ranks. To be able to do this, you need to switch it on in the following way:

```
$ make USE_MPI=1
```

4.2 Preprocessing annotated code

The preprocessor is realized with a python script. Example scripts can be found in the example directory, e.g. 'examples/array_test_c/perf_regions_instrumentation.py'. This script instructs the preprocessor where to find the code and how to preprocess the code.

4.2.1 Preprocess

To start preprocessing the code, the script is executed with "preprocess" as parameter:

```
$ ./perf_regions_instrumentation.py preprocess
```

4.2.2 Reverting to original code (if required)

The preprocessing generates code which can be reverted to its original one. To revert the PerfRegion code, call the script with "cleanup" parameter:

```
$ ./perf_regions_instrumentation.py cleanup
```

4.3 3rd party library, compiling and linking

PerfRegions requires the PAPI library installed.

4.3.1 Linker flags:

```
-lpapi -L[path to perf regions]/build -lperf_regions
```

4.3.2 Compile flags:

```
-I[path to perf regions]/src
```

4.4 Executing performance measurements

First of all, the environment variable LD_LIBRARY_PATH has to be set to the path of the PerfRegion build directory:

```
$ export LD_LIBRARY_PATH=[path to perf regions]/build:$LD_LIBRARY_PATH
```

Each platform might have a different set of performance counters. The available performance counters identifiers can be determined via:

```
$ papi_avail
```

Only a limited number of performance identifiers can be specified. PerfRegions allows to specify a list of performance counter identifiers via the environment variable such as:

```
$ export PERF_REGIONS_COUNTERS=PAPI_L1_TCM,PAPI_L2_TCM,PAPI_L3_TCM
```

Also the wallclock time can be measured by adding "**WALLCLOCKTIME**" to this list which would add a separate column in the output for the wallclock time:

```
$ export PERF_REGIONS_COUNTERS=PAPI_L1_TCM,PAPI_L2_TCM,PAPI_L3_TCM,WALLCLOCKTIME
```

4.5 Example output

An example output looks as follows:

```
Performance counters profiling:
----------------------------
Section  PAPI_L1_TCM    PAPI_L2_TCM    PAPI_L3_TCM    PAPI_TOT_INS   SPOILED  COUNTER
FOOA     2.5846895e+08  1.9985350e+08  9.4317611e+07  8.8859247e+10  1        1
FOOB     2.4577002e+08  1.8729369e+08  8.6322805e+07  8.5007949e+10  1        1
BARA     1.7902568e+08  1.3522808e+08  6.1375034e+07  6.1987130e+10  1        1549
BARB     6.2486545e+07  4.7860065e+07  2.2210664e+07  2.1407764e+10  0        1549
```

4.6 Accumulation of results for multiple MPI processes

When using MPI you can track the wallclocktime etc for every MPI process individually and then in the end accumulate the results for all ranks. This is activated by using the *init_mpi* annotation instead of *init*. The results give the total mean, minimum, maximum and variance over all wallclocktime measurements of a given section for all ranks and additionally all the results for rank 0.

An example output looks as follows:

```
[PERF_REGIONS] section name: STEP
[PERF_REGIONS] [STEP].total_samples: 44
[PERF_REGIONS] [STEP].min_wallclock_time: 1.6641617e-04
[PERF_REGIONS] [STEP].max_wallclock_time: 9.1576576e-04
[PERF_REGIONS] [STEP].mean_wallclock_time: 3.0701811e-04
[PERF_REGIONS] [STEP].variance_wallclock_time: 6.0474020e-08
Perf_regions, results for rank 0:
Performance counters profiling:
----------------------------
Section  PAPI_L1_TCM    SPOILED  WALLCLOCKTIME  MIN            MAX            MEAN           VAR            COUNTER
STEP     1.6625300e+05  0        3.3783913e-03  1.6784668e-04  9.1433525e-04  3.0712648e-04  5.9292986e-08  11
```

When just using *init*, normal non-accumulated results are given for all MPI processes.

5 Conclusions

[Let's see how it goes and then fill in this section...]