This assignment covers profilers and performance analysis tools.

1 Superuseful application – GNU gprof (2P)

The GNU and Intel compiler have integrated support for profiling. If the option `-pg` is added to the compiler AND linker flags, the execution of the program will generate a profile data file `gmon.out`. The profile can be analysed using the gprof tool.

In this exercise, we will examine a super useful application, which is given in the file `superuseful.cpp`. This application has two kernels. In this exercise we want to determine, which kernel is a candidate for optimisation using only the information returned by the profiler. Compile the superuseful application for Intel Xeon Phi using optimisation level 3 and enable profiling.

1. Why do the interesting functions `kernel1` and `kernel2` do not show up in the call graph when the Intel 17 compiler is used? Would you expect this behaviour from the documentation of the Intel 17 compiler?

2. Use compiler options such that `kernel1` and `kernel2` show up in the call graph. Is the resulting profile still useful?

3. Add compiler pragmas such that the `kernel1` and `kernel2` show up in the call graph but without incurring significant profiler overhead.

4. Which function should be optimized first (according to the profile data)?

2 Quicksort – Intel VTune Amplifier XE (2P)

Intel VTune Amplifier XE is a profiler from Intel. It is available on the Linux Cluster and can be loaded with `module load amplifier_xe`. VTune comes with a graphical user interface (`amplxe-gui`) and a command line tool (`amplxe-cl`).

In this exercise we will optimize the parallel Quicksort from the second worksheet using VTune. The goal is, to find an optimal `final`-clause for the OpenMP tasks.

Setup a project for Quicksort with the VTune GUI and configure a "Basic Hotspots" analysis. **However, do not start the analysis from the GUI.** If you do this, the application will run on the login node. As a workaround, the GUI can export the command to run the analysis with `amplxe-cl`. Put the command in a SLRUM script and submit it. Once the application is finished, you can visualize the results of the analysis with the GUI.

1. Find the optimal `final`-clause for your OpenMP parallelization of Quicksort using Intel VTune Amplifier XE.

2. Look at the other analysis types besides "Basic Hotspots". Are they useful for the Quicksort application? What is the difference between them?

3. Does the optimal final clause depend on the number of threads or the array length?

Hint: You can download the generated profile data from the cluster and inspect it on your local machine using the Amplifier XE GUI. You may obtain a free student license of Amplifier XE from the intel website. As an alternative you can use X forwarding in SSH to start the Amplifier XE GUI directly on the cluster; this might be slow, however.

3 CG – Scalasca (3P)

In this exercise we will look at the MPI parallelization of the conjugate gradient method from the third worksheet. We will use Scalasca to find load imbalances in the parallelization. To instrument the code, you have to place the prefix `scalasca -instrument [options]` before the compiler and linker, e.g. `scalasca -instrument mpiCC poisson.cpp`.

With the `options` you can control which parts of the code you want to instrument. Start with `--mpp=mpi` to instrument normal functions (default) and MPI functions. The instrumented application will generate profile data in a folder called `scorep-*`. The data can be visualized with `cube`. Besides the graphical interface there is also a set of command line tools available to convert or dump the profile data which might be useful for automatic processing.

A nice feature of Scalasca is, that it does not only support automatic but also manual code instrumentation. The manual instrumentation can be enabled with the option `--user`. To define a "region" in your code, you can use the macros `SCOREP_USER_REGION_DEFINE( handle )`, `SCOREP_USER_REGION_BEGIN( handle, "foo", SCOREP_USER_REGION_TYPE_COMMON )` and `SCOREP_USER_REGION_END( handle )` defined in `scorep/SCOREP_User.h`. You will see your defined regions similar to normal functions in `cube`.

1. Scalasca is available as a module on CoolMUC2 but not on CoolMUC3. **For this exercise, you may choose to work on either cluster.** For CoolMUC3, you have to manually install scalasca using the standard `./configure; make; make install` approach. Check `./configure --help` for available compiliation options, e.g. for selecting GNU or Intel compiler. You should also have a close look at the `--prefix` option and you should set important environment variables in your `~/.bashrc`. (Especially `PATH, CPATH, CPPPATH, LIBRARY_PATH, LD_LIBRARY_PATH`.)

2. Profile your parallel CG implementation. Do you detect load imbalances for certain numbers of tasks? Explain the box plot in `cube`.

3. The automatic instrumentation usually generates too much profile data. Disable it with `--nocompiler` and enable the manual instrumentation. Define suitable regions in the code to get detailed information without to much overhead.
   Hint: You can use the header from the SeisSol project to easily compile with and w/o manual instrumentation. The header defines dummy macros if manual instrumentation is not enabled:
   `https://github.com/SeisSol/SeisSol/blob/master/src/Monitoring/instrumentation.fpp`

4. Parallelize the important loops in the application with OpenMP. Does the hybrid MPI/ OpenMP parallelization give you a speedup? Besides manual, compiler and MPI instrumentation, Scalasca can also instrument OpenMP pragmas with `--thread=omp`. Does the OpenMP instrumentation help you optimizing your program?

Hints: The Cube viewer might not compile on CoolMUC3 due to the missing Qt library. Once you have generated a profile on CoolMUC3, you may switch to the login node of CoolMUC2 and use the installed Cube viewer.

On CoolMUC2 you have to add `-Mmpp2` to the SLURM commands to view information about available nodes. E.g. `sinfo -Mmpp2`.

4 Likwid-perfctr (3P)

Likwid-perfctr (`module load likwid`) is a tool to gather hardware performance counters for an application. As LIKWID is unavailable on CoolMUC3, **please use CoolMUC2**.

In contrast to many profilers, it does not instrument the code but relies on performance counters from the CPU and the kernel. Thus, it will not detect how long a code spends in a specific function but hardware such as total number of instructions, TLB misses, cache misses, or branch miss predictions.

1. The file `game.cpp` could be part of a physics engine for a computer game. As in classic object oriented programming styles, it creates a heap-allocated object for every rigid body. Here, we want to analyse its performance characteristics using LIKWID.

   a. Using the marker API of LIKWID allows the measuring of specific code regions [2]. Introduce a marker for the `gravity` function. What do you need to do in order for the markers to work (compilation options, command line arguments)?

   b. Use `likwid-perfctr -a` to get a list of all available event groups. Which event groups are relevant for this kind of application and which are not? Give a brief explanation for every event group.

   Hint: Check the architecture specific notes for Haswell EP [2].

   c. Run the application and measure relevant event groups. What would be your suggestions for improving the performance of the application?

2. The file `dtrmv.cpp` contains a routine for upper-triangular matrix times vector multiplication. We want to investigate possible load imbalances, as the routine does not scale well to multiple cores.

   a. Calculate the number of flops for every core manually and explain the load imbalance.

   b. Instead of hand calculation, it would be convenient to use hardware counters for the number of flops. However, these are known to be inexact since Sandy Bridge. Insert a marker for the `dtrmv` routine and measure the number of flops using LIKWID. Repeat the measurement for different numbers of threads. Are the measurements useful to find the load imbalance?

c. Is `INSTR_RETIRED_ANY` a useful alternative?

d. Modify the program such that the load is approximately balanced.

Deliverables

The following deliverables have to be handed in no later than 08:00 AM, Monday, 18 December 2017. If there is no submission until this deadline, the exercise sheet is graded with 0 points. Small files (<1 MB in total) can be send as an attachment directly to *uphoff AT in.tum.de* and *chaulio.ferreira AT tum.de*. Larger files have to be uploaded at a place of your choice, e.g. `https://gitlab.lrz.de/`, `http://home.in.tum.de/`, `https://syncandshare.lrz.de`. In either case inform us about the final state of your solution via e-mail.

- A short report which describes your work and answers all questions in this assignment.
- All of your code.
- Slides for the presentation during the next meeting.
- Output of all runs. Figures (e.g. scaling graphs) if applicable.
- Documentation how to build and use your code.
- Output/Screenshots from the profilers.

Literature

[1] GNU gprof. `https://sourceware.org/binutils/docs/gprof/`. Accessed: 2017-12-1.

[2] LIKWID. `https://github.com/RRZE-HPC/likwid/wiki`. Accessed: 2017-12-1.

[3] Scalasca user manual. `https://apps.fz-juelich.de/scalasca/releases/scalasca/2.3/docs/UserGuide.pdf`. Accessed: 2017-12-1.

[4] Score-P user manual. `https://silc.zih.tu-dresden.de/scorep-current/pdf/scorep.pdf`. Accessed: 2017-12-1.