2019-10-24: Profiling to enhance testing

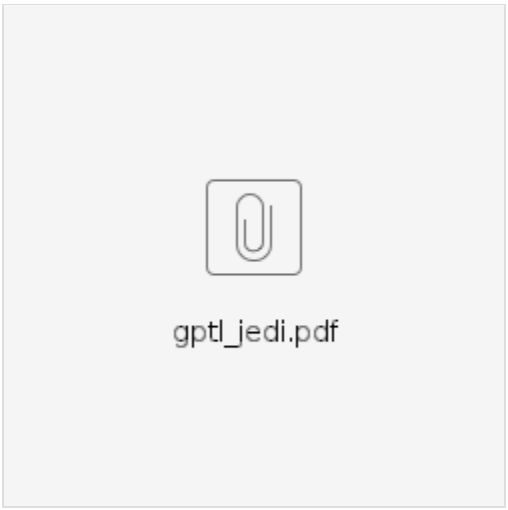
The focused topic chosen for today's meeting is the potential use of profiling in connection with our unit testing.

When using profilers to optimize code, it is often most useful for developers to work interactively with GUIs that are designed to display comprehensive performance metrics in graphical format. This will always be the case and it is not the focus of this meeting. Rather, the focus of this meeting is more to ask whether there are aspects of code profiling that we may wish to automate and include in the testing workflow as a way to identify bugs or performance bottlenecks.

In the JEDI meeting two weeks ago, 2019-10-10 and the discussion that followed we identified several possible profilers to consider:

- The **GPTL** profiling library developed by Jim R
- The profile tools developed by Mark O for the JEDI-Rapids workflow that are based on the python **psutil** package
- The **TAU** performance system from Univ. Oregon that Jong has been using to profile SOCA
- The intel **vtune/inspector/advisor** tools used by several team members, particularly Ryan
- The **MAP** profiler that is included with the DDT debugger in the Arm Forge package; Steve V has used this recently for MPAS

The discussion began with Jim R's description of GPTL:



For this and the other topics below, please consult the slides for details. What follows is jut a brief overview. Jim began by saying that GTPL was designed for analyzing the performance of large parallel (MPI/OpenMP) jobs but it might also be useful for the unit testing functionality described above. It is written in C and can be used for C, C++, and Fortran code. The GTPL libraries utilizes the PMPI library which provides a profiling layer for MPI. It can run in two different modes. The first is through explicit sentinels - code modifications that start and stop the profile timers for specific sections of code . The second mode is auto-instrumentation that uses compiler-generated entry and exit points enabled with the "-finstrument-functions" compiler option. Yannick asked if the user can mix these two modes and Jim responded yes - the example on the slides demonstrates this.

The red text in the example on slide 3 represent lines that must be added to the source code to enable either mode of operation. The green text highlights code that needs to be added for the sentinel mode, including start and stop points selecting some code segment. The profiling information is written out to a file with the GPTLpr(myrank) function. Each MPI task writes to a separate file. The user has the option of writing information for all MPI tasks or for only a selected subset. There is also a summary display as shown on slide 5 that summarizes statistics across all MPI tasks, such as the mean, min, and max time for particular functions/subroutines and the number of time each function/subroutine is called.

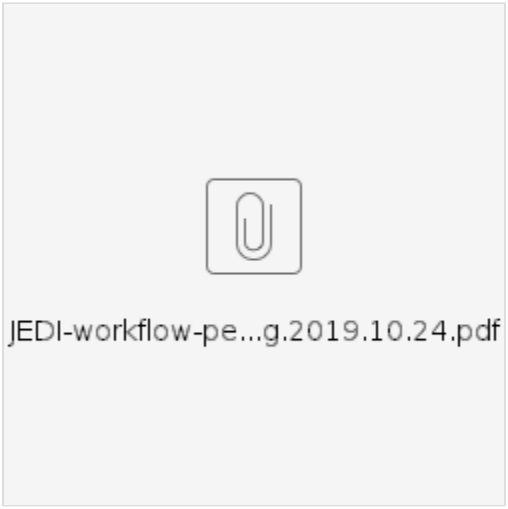
Slide 6 shows example output from a more substantial application, namely GSI. Jim noted in particular the entries that begin with synch_. These are synchronization points that come before calls to collective MPI functions such as MPI_Alltoall. The main purpose of the synchronization is to get robust results for the MPI timing statistics but the synchronization times reported can also give insight into load imbalance and isolate load imbalance from MPI communication costs.

The GTPL package also contains some tracking of memory usage as shown on slide 7. Output is generated if a function's memory grows and it can be useful to diagnose where the problem is when an application runs out of memory. Another feature, demonstrated on slide 8, is the tracking and reporting of what parent function called a particular function and how many times.

Clementine asked whether one can get performance summaries for a subset of MPI tasks and Jim responded yes - one can pass an MPI communicator to the summary function to get information for a particular group. Mark O asked about overhead. Jim responded that one of the significant sources of overhead can be to call the intrinsic gettimeofday() routines to do the timings. When called many times, potentially billions of times in a large application, this can add up. Jim avoids this when possible. Mark O mentioned accessing the hardware counters through, for example, the papi library. Jim agreed this can be useful if you're careful about how the hardware counters are reset. In any case, the overhead will depend on how many times a particular sentinel is called.

To turn off profiling, one must recompile without the "-finstrument-functions" compiler option and without linking to the GPTL library.

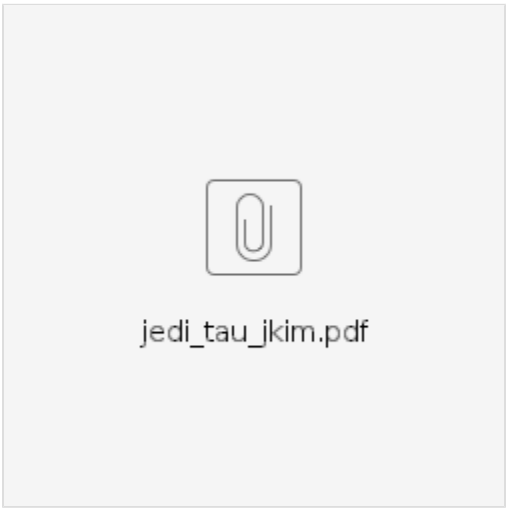
Mark O then discussed how he is using the python psutil tools:



This is very different from the GTPL package Jim described and could be complementary. In contrast to GTPL, it does not provide function-level profiling information. Rather, it queries the system hardware much as the linux "top" package does. So, it can provide time series of cpu and memory usage by each MPI task and by the application as a whole. Due to the nature of how it operates, it does not require any code instrumentation and it can work with any programming language. Another nice feature is that it writes all its profiling information to "python pickle" output files that can then be post-processed however one wishes. So, for example, one can compare the timing statistics of different versions of an application or one can plot further details of any potential problem areas that may be identified automatically. One can also use these tools to identify applications for function-level profiling as enabled by GPTL or tau. Mark O also mentioned that he is now running it on one node but it could be extended to multi-node jobs. This would probably involve writing a separate pickle file for each node and then processing them together.

One feature of this tool is that the execution of the application is handled by python, so python calls mpirun. So, if we were to include this in ctests, we would need to call python wrappers rather than the applications directly.

We then turned to Jong who described SOCA profiling with TAU:



Jong began by emphasizing that Tau is a heavy-weight, complex application that is probably overkill for the unit testing use case described above. But, it can be an extremely powerful and useful tool when run interactively with the GUI. There are different levels of instrumentation possible. The simplest and least invasive it to just compile the code with the "-g" option and run it with the "tau_exec" executable and event-based sampling, e.g.

```
mpirun -np tau_exec -ebs <myapp>
```

You can add a "-memory" option to track memory leaks and other memory diagnostics. More detailed profiling information can be obtained by inserting code instrumentation as shown on slide 2 and compiling with wrappers that are provided as part of the tau package.

Then, because the hour was almost up, we decided to stop the discussion there and revisit it at the next JEDI focused topic discussion in two weeks (Nov 7). We will begin that discussion with a demonstration by Ryan on the use of vtune/inspector/advisor and then we will proceed to discuss how we might implement one or more of these tools in our CI testing.

Before the meeting closed there were a few other comments. Travis asked Mark O whether psutil might miss peak resource usage because it operates in discrete sampling intervals. Mark O said yes, that could be a problem. One can increase the sampling frequency but this generates much more data that must be processed.

Steve V mentioned that he has used the MAP tool from Arm FORGE on Cheyenne and finds it to be very useful and easy to use. But, being proprietary and GUI-heavy, this might make it unsuitable for incorporating into our CI testing.

Ryan did give a brief description of the intel tools. He mentioned that these are freely available (note error in Mark M's meeting announcement that said these required an intel license - they do not). Jim R added that these tools can be extremely useful. He mentioned in particular a race condition in threads that was quickly diagnosed with inspector that would have been very difficult to track down otherwise.

It was mentioned that OOPS has timers that are in use now which provide coarse timing information (basically the total time for each test). It might make sense to start with these timing data for experimenting with automatic timing analysis during unit testing.

The meeting closed with a reminder from Yannick to please suggest more topics for focused JEDI meetings in the future. You can do this by sending an email to one of us on the core team or by directly editing the JEDI Bi-weekly Discussion ZenHub board.