Profiling

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

Part I

Profiling

Profiling

- As HPC programmers, we want our code to run as fast as possible.
- How do we know which parts are causing the bottleneck?
- Good profiling is essential; it may not be the parts you expect.
- "Premature optimization is the root of all evil" - Donald Knuth

Things you should already have considered

- Correct algorithm
- Basic data layout for reasonable cache performance
- Do not copy data around more often than necessary; pass by reference.
- This talk is applicable to C/C++/Fortran only; other languages have their own profiling tools.
- Examples are given in C++ but the tools/APIs have C and Fortran interfaces.

Basic timing

- Basic profiling can often be achieved by:

```
clock_t start = clock();
// Do something expensive
clock_t end = clock();
std::cout << "Total time " << (double)(end - start) /
    CLOCKS_PER_SEC << "s" << std::endl;
```

- Using some well-placed macros and putting them around likely functions may be all you need.
- More advanced use may require summing the time taken for multiple calls to the same function.

gprof

- The next step is to use automatically instrumented profiling calls from the compiler: `https://sourceware.org/binutils/docs/gprof/`
- With `gcc`, use the `-pg` option.
- With `icc`, use the `-p` option.
- Run instrumented code as normal, slowdown: Less than 5%. Generates `gmon.out` file.
- Post-process using: `gprof ./MyCode ./gmon.out`

gprof

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.04      0.03     0.03                             dot(int,
  0.00      0.03     0.00        1     0.00     0.00  _GLOBAL_
  0.00      0.03     0.00        1     0.00     0.00  _GLOBAL_
```

valgrind

- http://valgrind.org/
- No recompilation needed (debugging symbols `-g` required).
- Essentially a CPU emulator; includes cache and branch-prediction simulation. Run normal code as:

  ```
  valgrind --tool=callgrind --callgrind:dump-instr=yes
  --cache-sim=yes --branch-sim=yes ./MyCode
  ```
- Slowdown: Factor of 30-50.
- Visualise using `kcachegrind`.

Score-P

- http://www.vi-hps.org/projects/score-p/
- Special compiler/linker wrapper required. Available on CSD3 as a module:

```
module load scorep/2.0.2/intel-impi-latest

scorep-cxx -c MyMPICode.C -O3 MyMPICode.o

export SCOREP_ENABLE_TRACING=1
export SCOREP_ENABLE_PROFILING=1
export SCOREP_EXPERIMENT_DIRECTORY=./MyMPICode_ScoreP_np1
./MyMPICode
```

- Slowdown: Varies but usually less than 5%
- Generates output in `$SCOREP_EXPERIMENT_DIRECTORY` in otf2 format.

Visualisation

Various visualisation tools are available for SCOREP output:

- Cube (GUI not brilliant)
- Periscope, TAU (did not compile immediately...)
- Vampir - commercial code - cheapest option about 500.

I have mainly used Vampir; seems to have the clearest UI.

MPI profiling

- If we simply profile individual MPI processes, we have no visibility of what causes an MPI function call to wait.
- The MPI standard allows for profiling functions/hooks to be implemented and labelled with the universal wall-clock time.
- Score-P does this for most MPI functions.

MPI Visualisation

Vampir comes into its own when applied to MPI codes.

Processor instructions

papi_avail:

```
PAPI_L1_DCM  Level 1 data cache misses
PAPI_L1_ICM  Level 1 instruction cache misses
PAPI_L1_TCM  Level 1 cache misses
...
PAPI_FP_OPS  Floating point operations
PAPI_SP_OPS  Floating point operations; optimized to count sca
PAPI_DP_OPS  Floating point operations; optimized to count sca
```

Note that these are only available on Xeon-class processors, not i7-class.

```
export SCOREP_METRIC_PAPI=PAPI_FP_OPS,PAPI_VEC_DP,PAPI_L1_TCM
```

See results in Vampir.

Score-P user instrumentation

- Although the `scorep-cxx` wrapper instruments code automatically, this may be overkill (e.g. in case of many small inlined functions).
- Better focused profiling may be achieved by turning off all function instrumentation and using macros:

```
int f(){
  SCOREP_USER_FUNC_BEGIN();
  SCOREP_USER_REGION_DEFINE(MyAlgPart1);
  SCOREP_USER_REGION_BEGIN( MyAlgPart1,
    "AlgorithmPart1",SCOREP_USER_REGION_TYPE_COMMON );
  /* Some code; */
  SCOREP_USER_REGION_END(MyAlgPart1);

  SCOREP_USER_REGION_DEFINE(MyAlgPart2);
  SCOREP_USER_REGION_BEGIN( MyAlgPart2,
    "AlgorithmPart2",SCOREP_USER_REGION_TYPE_COMMON );
  /* Some other code;*/
  SCOREP_USER_REGION_END(MyAlgPart2);

  SCOREP_USER_FUNC_END();
}
```

Unexpected things you may find

- Multiple small memory allocations - try using a pool of memory instead
- `pow` function in glibc used to run very slowly for certain inputs.
- Input/output performance