Dr. Pekka Manninen
CSC - IT Center for Science
Finland

Performance Optimization of Scientific Software

Part I: Performance Analysis

CSC Webinar Oct 30, 2018

WEBINAR SERIES INTRODUCTION

A day in life at CSC

CSC customer

me

I’m performing simulations with my Fortran code. It seems to perform much worse with MKL library in the new system than with IMSL library in the old system.

Have you profiled your code?

No

A day in life at CSC

- I profiled the code: 99.9% of the execution time was being spent on these lines:

```
do i=1,n
  do j=1,m
    do k=1,fact(x)
      do o=1,nchoosek(x)
         where (ranktypes(:,:)==k)
            ranked(:,:,o)=rankednau(o,k)
         end where
      end do
    end do
  end do
end do
```

A day in life at CSC

- Removing the unnecessary loops

```
do i=1,n
  do j=1,m
    do k=1,fact(x)
      do o=1,nchoosek(x)
         where (ranktypes(:,:)==k)
            ranked(:,:,o)=rankednau(o,k)
         end where
      end do
    end do
  end do
end do
```

...reduced the execution time from 17 hours to 3 seconds

Performance optimization of scientific software

- **Part I: Performance Analysis (today)**
- **Part II: Node-level performance tuning (Nov 6)**
- **Part III: Improving Application Scaling (Nov 20)**
- The assumed platform here is CSC's Cray XC40 supercomputer Sisu, most of the content and considerations are applicable and transferable to other platforms as well
- Please be prepared that these will be a bit longer than typical webinars
- Questions preferably at the end of the session
- An optional hands-on exercise provided

Improving application performance

- Obvious benefits
  - Better throughput => more science
  - Cheaper than new hardware
  - Save energy, compute quota etc.
- ..and some non-obvious ones
  - Potential cross-disciplinary research
  - Deeper understanding of application

Performance optimization

- Adapting the problem to the underlying hardware
- Key factors to application performance
  - Effective algorithms, doing things in a more clever way
    - e.g. $O(n \log(n))$ vs $O(n^2)$
  - High CPU cycle utilization
  - Efficient memory access
  - Parallel scalability
  - File I/O efficiency

Performance optimization

- Important to understand dependencies
  - Algorithm – code – compiler – libraries – hardware
- Performance is not portable
- Optimize *only* the parts of code that are relevant for the total execution time!
  - ”The 90/10 rule”: most of the time (~90%) is typically being spent in executing a very limited number of code lines (~10%)

PERFORMANCE ANALYSIS: FIRST CONSIDERATIONS

Application timing

- Most fundamental information: total wall clock time
  - Built-in timers in the program (e.g. MPI_Wtime)
  - System commands (e.g. time) or batch system statistics
- Built-in timers can provide also more fine-grained information
  - Have to be inserted by hand
  - Typically no information about hardware related issues
  - Information about load imbalance and communication statistics of parallel program is difficult to obtain

Performance analysis tools

- *Instrumentation* of code
  - Adding special measurement code to binary
  - Normally all routines do not need to be measured
- *Measurement*: running the instrumented binary
  - Profile: sum of events over time
  - Trace: sequence of events over time
- *Analysis*
  - Text based analysis reports
  - Visualization

Some performance analysis tools

- CrayPAT (available on Sisu)
  https://docs.cray.com
  see also ”`man intro_craypat`” on Sisu
- Scalasca
  http://www.scalasca.org/
- Paraver
  https://tools.bsc.es/
- Intel VTune Amplifier
  https://software.intel.com/en-us/vtune

Profiling

- Purpose of *profiling* is to find the "hot spots" of the program
  - Determine, which routines consume the most of the execution time (or the metric we are optimizing for)
- Usually the code has to be recompiled or relinked, sometimes also small code changes are needed
- Often several profiling runs with different focus are needed for a proper analysis

Profiling: sampling

The application execution is interrupted at constant intervals and the program counter and call stack is examined

Pros

- Lightweight
- does not interfere the code execution too much

Cons

- Not always accurate
- Difficult to catch small functions
- Results may vary between runs

Profiling: tracing

Hooks are added to function calls (or user-defined points in program) and the required metric is recorded

Pros

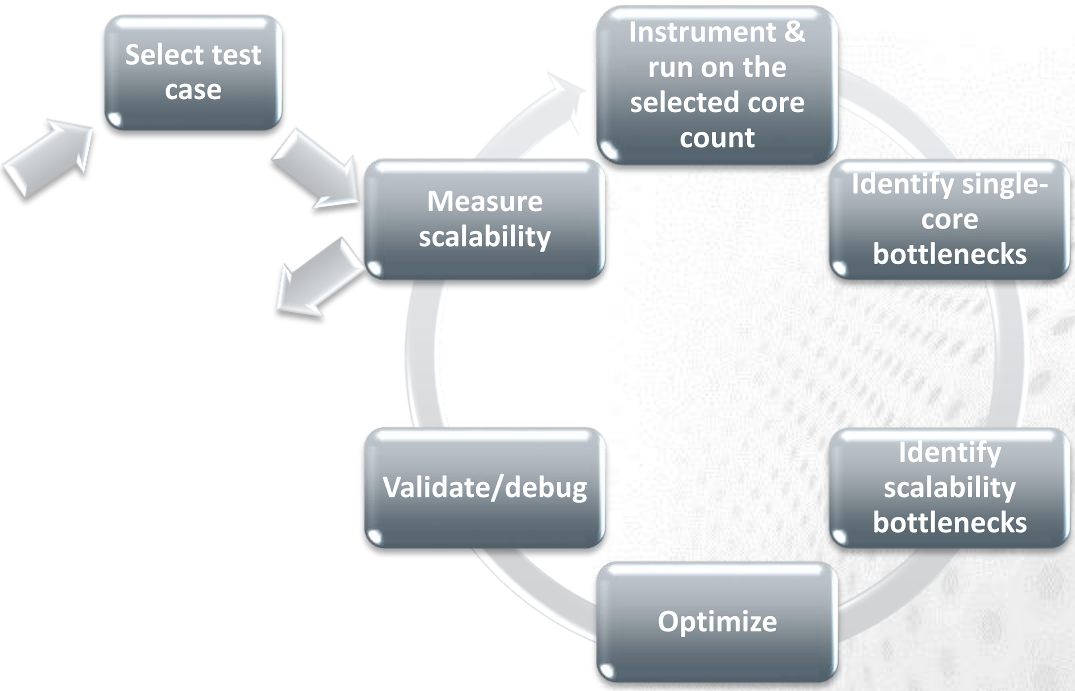
- Can record the program execution accurately and repeatably

Cons

- More intrusive
- Can produce prohibitely large log files
- May change the performance behaviour of the program

CODE OPTIMIZATION PROCESS

Code optimization cycle

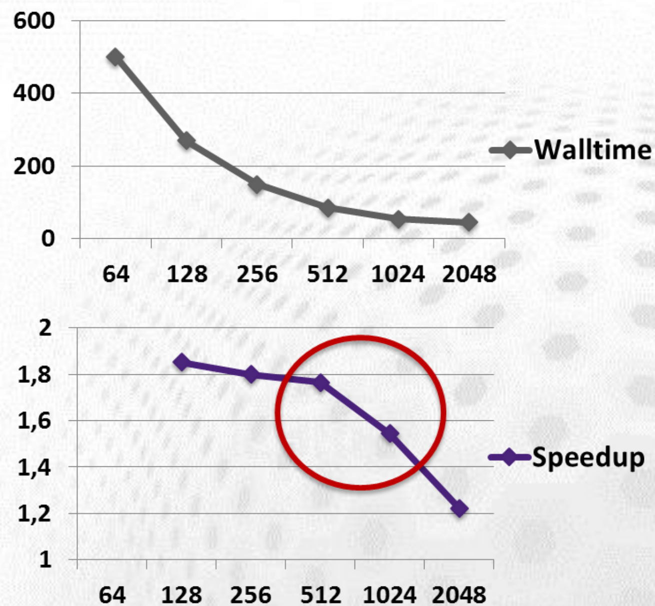


Step 1: Choose a test problem

- The dataset used in the analysis should
  - Make sense, i.e. resemble the intended use of the code
  - Be large enough for getting a good view on scalability
  - Complete in a reasonable time
  - For instance, with simulation codes almost a full-blown model but run only for a few time steps
- Remember that initialization/finalization stages are usually exaggerated and exclude them in the analysis

Step 2: Measure scalability

- Run the uninstrumented code with different core counts and see where the parallel scaling stops
- Often we look at strong scaling
  - Also weak scaling is definitely of interest



Step 3: Instrument & run

- Profile the code with
  - The core count where the scalability is still ok
  - The core count where the scalability has ended

  and compare these side-by-side: what are the largest differences between these profiles?

Step 4: Find single-core hotspots

- Remember to focus only on user routines that consume significant portion of the total time
- Collect the key hardware utilization details, for example
  - Cache & TLB metrics from the performance analysis tool
  - See the compiler output: are the hotspot loops being optimized, especially vectorized by the compiler?
- Trace the math intrinsics to see if expensive operations (exp, log, sin, cos,...) have a significant role

Step 4: Find single-core hotspots

- Signature: Low L1 and/or L2 cache hit ratios
  - <96% for L1, <99% for L1+L2
  - Issue: **Bad cache utilization**
- Signature: Low vector instruction usage
  - Issue: **Non-vectorizable (hotspot) loops**
- Signature: Traced ”math” group featuring a significant portion in the profile
  - Issue: **Expensive math operations**

Step 5: Identify scalability bottlenecks

- Signature: User routines scaling but MPI time blowing up
  - Issue: **Not enough to compute in a domain**
    - Weak scaling could still continue
  - Issue: **Expensive collectives**
  - Issue: **Communication increasing as a function of tasks**
- Signature: MPI_Sync times increasing
  - Issue: **Load imbalance**
    - Tasks not having a balanced role in communication?
    - Tasks not having a balanced role in computation?
    - Synchronous (single-writer) I/O or stderr I/O?

Part I concluding remarks

- Profile your code before optimizing anything
  - ”Premature code optimization is the root of all evil”
- Do the profiling yourself
  - Do not believe what the others claim about your code
- Profile the code on the hardware you are going to run it
  - Hotspots & bottlenecks will differ between your laptop and a supercomputer
- Profile with a representative test case
  - The hotspots of a toy problem are different to those of the real-world case
- Reprofile the code after every optimization

Optional lab

- To put the contents to practice, there is a self-consistent lab exercise available on the webinar page
  - Instructions in labs.pdf, a sample code in labs.tar.gz
- The first four sections will relate to this first part of the webinar series