Performance Tools
Use Case

June 2016  |  Ilya Zhukov
POP CoE

EU H2020 Center of Excellence
- On Performance Optimization and Productivity
- Promoting best practices in performance analysis and parallel programming

Providing Services
- Precise understanding of application and system behavior
- Suggestion/support on how to refactor code in the most productive way

Horizontal
- Transversal across application areas, platforms, scales

For academic AND industrial codes and users!
Partners

Who?

- BSC (coordinator), ES
- HLRS, DE
- JSC, DE
- NAG, UK
- RWTH Aachen, IT Center, DE
- TERATEC, FR

A team with

- Excellence in performance tools and tuning
- Excellence in programming models and practices
- Research and development background AND proven commitment in application to real academic and industrial use cases
Motivation

Why?

• Complexity of machines and codes
  - Frequent lack of quantified understanding of actual behavior
  - Not clear most productive direction of code refactoring

• Important to maximize efficiency (performance, power) of compute intensive applications and the productivity of the development efforts

Target

• Parallel programs, mainly MPI/OpenMP … although can also look at CUDA, OpenCL, Python, …
3 levels of services

? Application Performance Audit
  • Primary service
  • Identify performance issues of customer code (at customer site)
  • Small Effort (< 1 month)

! Application Performance Plan
  • Follow-up on the service
  • Identifies the root causes of the issues found and qualifies and quantifies approaches to address the issues
  • Longer effort (1-3 months)

⚠️ Proof-of-Concept
  • Experiments and mock-up tests for customer codes
  • Kernel extraction, parallelization, mini-apps experiments to show effect of proposed optimizations
  • 6 months effort

Apply @ http://www.pop-coe.eu
Performance audit of ParFlow

What to analyze:
- ParFlow parallel, three-dimensional, variably saturated groundwater flow code
- MPI
- Fortran and C
- weak scaling testcase

Where: Juqueen

What is the problem:
- Scalability?
- Memory?

Tools for analysis:
- Score-P 1.4.2 and 2.0.1
- Scalasca 2.2.1
- PAPI 5.3.0
Behavior and Structure

Syntactic structure

solve()
{
    init_solver();
    for( i=0; i < num_timesteps; i++)
    {
        init_problem();
        solver_loop()
        {
            while( residual > tol )
            {
                nonlinear_iterative_solver();
            }
        }
    }
}
Scalability

Total execution time

Average time of specific regions
Efficiency

- Parallel efficiency metrics based on time
- Values from 0 to 1 (the higher the better)

<table>
<thead>
<tr>
<th>Efficiency metric</th>
<th>Application in total</th>
<th>init_solver</th>
<th>init_problem</th>
<th>solver loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load balance (avg/max)</td>
<td>0.99</td>
<td>0.99</td>
<td>0.83</td>
<td>0.96</td>
</tr>
<tr>
<td>Serialization</td>
<td>0.72</td>
<td>0.83</td>
<td>0.51</td>
<td>0.76</td>
</tr>
<tr>
<td>Transfer</td>
<td>0.9</td>
<td>0.99</td>
<td>0.99</td>
<td>0.88</td>
</tr>
<tr>
<td>Communication efficiency</td>
<td>0.65</td>
<td>0.82</td>
<td>0.51</td>
<td>0.67</td>
</tr>
<tr>
<td>Parallel efficiency</td>
<td>0.64</td>
<td>0.81</td>
<td>0.42</td>
<td>0.64</td>
</tr>
</tbody>
</table>
Communication

Wait-state analysis:
- Late Sender (14.6% of total time)
- Wait NxN (13.19% of total time)

Root cause analysis:
- init_solver
- solver_loop
- GetGridNeighbors

Delay analysis:
- init_solver
- solver_loop
- GetGridNeighbors
- PFMG
Audit Summary

Observations:
- Provided testcase is certainly communication bound.
- Waiting time is dominating in communication.
- Most of the waiting time is spent in "Late Sender" and "Wait at NxN" wait-state patterns.
- Application has load imbalance.
- Some memory leaks were detected.

Recommendations:
- Try to avoid logging and intermediate flushes.
- To remove "Late Sender" and "Wait at NxN" wait-states examine/refactor following routines: `init_solver`, `solver_loop`, `PFMG` and `GetGridNeighbors`
- Verify if it is really necessary to call `MPI_Comm_rank` so often.
- Examine `NewGrid` for memory leaks.