Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

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)

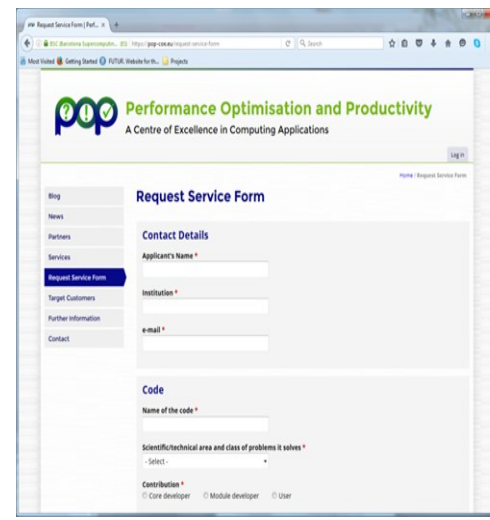
Proof-of-Concept

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

Reports

Software demonstrator

Apply @ **http://www.pop-coe.eu**



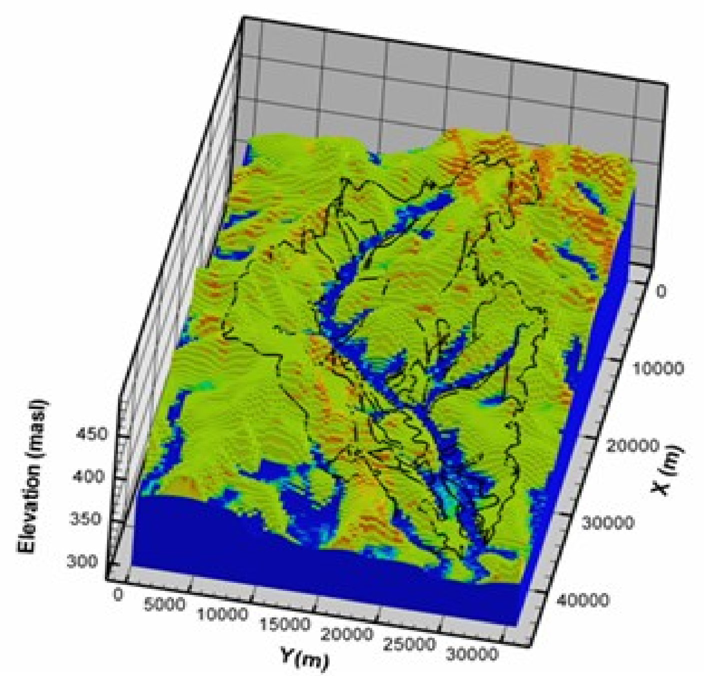
Performance audit of ParFlow

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

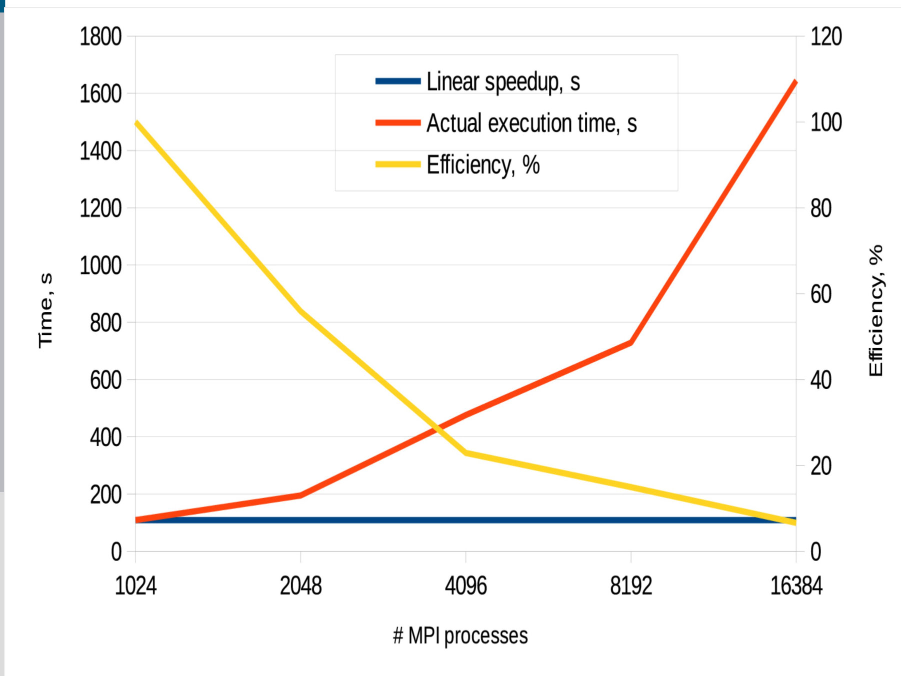
```
solve()
{
  init_solver();
  for( i=0; i < num_timesteps; i++ )
  {
    init_problem();
    solver_loop()
    {
      while( residual > tol )
      {
          nonlinear_iterative_solver();
      }
    }
  }
}
```

Approximate percentage of total execution time (1024 MPI processes)

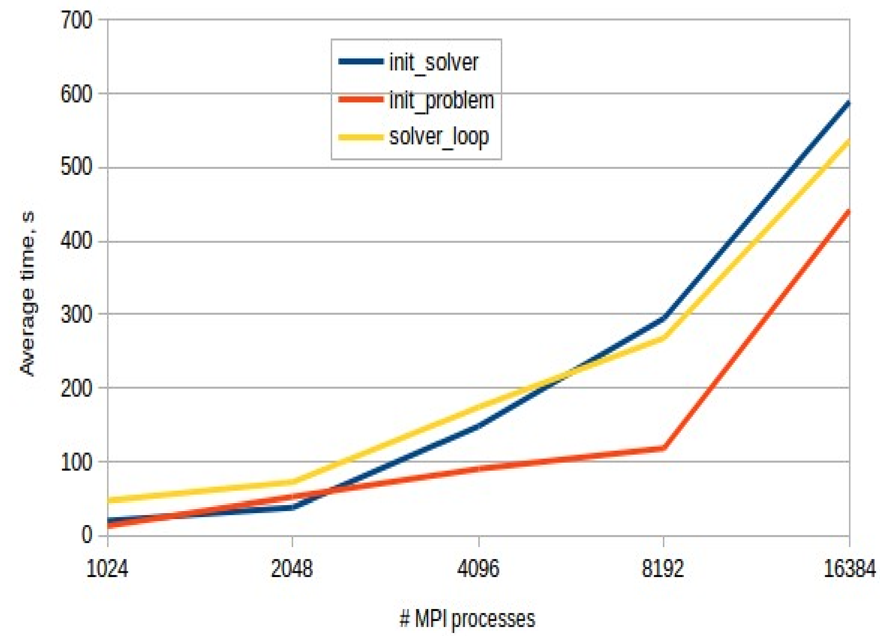
Part of application	Percentage of total execution time, %			
	Computation	MPI communication operations		Rest of MPI
		Point-to-point	collective	
init_solver	19	0	4	0
init problem	8	0	7	0
solver loop	31	20	3	3
Rest of the application	1	4	0	0
Application in total	59	24	14	3

Scalability

Total execution time



Average time of specific regions



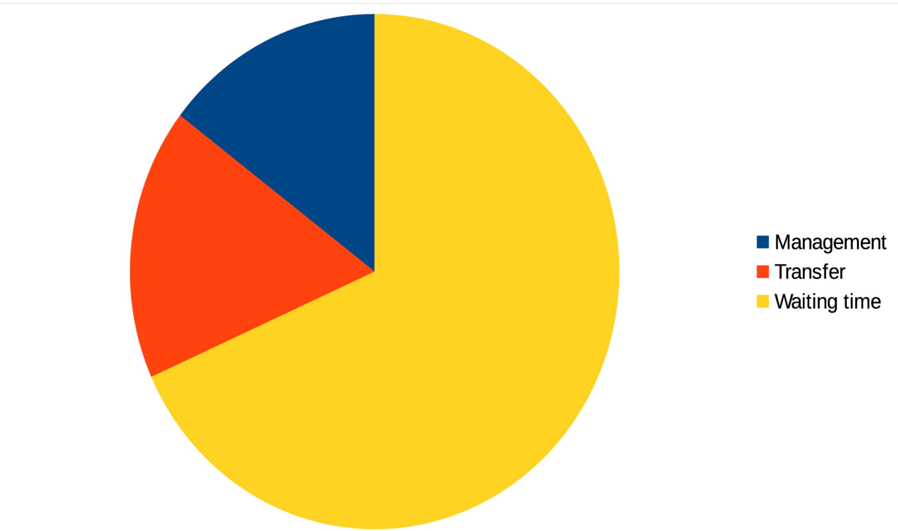
Efficiency

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

Efficiency metric	Part of application			
	Application in total	init_solver	init_problem	solver loop
Load balance (avg/max)	0.99	0.99	0.83	0.96
Serialization	0.72	0.83	0.51	0.76
Transfer	0.9	0.99	0.99	0.88
Communication efficiency	0.65	0.82	0.51	0.67
Parallel efficiency	0.64	0.81	0.42	0.64

Communication

Total MPI time



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.