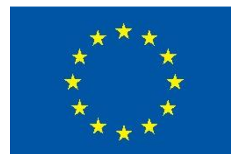
Incompressible immerflow

Fouzhan Hosseini, Numerical Algorithms Group

Fouzhan.hosseini@nag.co.uk, Sep 2019



EU H2020 Centre of Excellence (CoE)

1 December 2018 – 30 November 2021

Grant Agreement No 824080

Background

- Applicant: Marco Cisternino, Optimad, Italy
- Code: Incompressible immerflow (CFD), C++, MPI
- Platform: MareNostrum-IV(@BSC)
  - Dual Intel Xeon Platinum 8160 Skylake 48-core nodes
  - GCC-8.1.0, OpenMPI-4.0.1, PETSc-3.11.3, LAPACK-3.8.0
  - Testcase: lid_driven_cavity with load_balance_threshold -1.0
  - ~ 16 millions cells
  - 10, 100, 1000 iterations
- Scale: from 48 to 768 cores (1 to 16 nodes on MN-IV)
- Performance data collected using Scalasca/Score-P
  - Using compiler instrumentation filter and hardware counters

Strong Scaling

- Whole application and ROI (run_manager::computation())
  - run_manager::preprocess() is briefly discussed at the end

- Similar pattern for 10,100, 1000 iterations

- **Does not scale well beyond 400 cores**

Strong Scaling

ideal
80% of Ideal
Application-1000itr
ROI-1000itr
Application-10itr
ROI-10itr

Speedup
0 2 4 6 8 10 12 14 16

Cores
48 96 192 384 768

POP MPI Parallel Efficiency Metrics

Global Efficiency = *Parallel efficiency × Computational Scaling*

× Parallel Eff. = *average useful computation / runtime*

- For a perfectly parallelised code *runtime = average useful comp*
- Useful computation excludes time within MPI (or OpenMP, etc.)

× Load Balance = *average useful computation / maximum useful computation*

- Average and maximum is measured over all processes

Communication Eff. = *maximum useful computation / runtime*

- If time overhead of MPI is zero then runtime would equal maximum useful computation

× Transfer Eff. = *ideal runtime / runtime*

- Ideal runtime is calculated assuming infinite bandwidth and zero latency network

Serialisation Eff. = remaining cost of MPI

- Due to processes waiting idle for communications to start / end
- It excludes any contribution to inefficiency captured in 'load balance' (see above)

Computational Scaling = *Instruction Scaling × IPC Scaling × Frequency Scaling*

- Computed with reference to 1 node value

× Instruction Scaling | IPC Scaling | Frequency Scaling

POP Metrics

POP MPI Performance Metrics for Incompressible immerflow ROI - 10 iterations

Number of cores	48	96	192	384	768
Global Efficiency	0.91	0.93	0.97	0.86	0.69
↳Parallel Efficiency	0.91	0.87	0.87	0.75	0.58
↳ Load balance	0.99	0.98	0.97	0.94	0.91
↳ Communication Efficiency	0.92	0.89	0.89	0.8	0.64
↳ Serialisation	0.93	0.91	0.93	0.87	0.73
↳ Transfer efficiency	0.98	0.98	0.96	0.92	0.88
↳Computational Scaling	1.00	1.07	1.11	1.15	1.19
↳ Instruction Scaling	1.00	0.99	0.97	0.95	0.92
↳ IPC Scaling	1.00	1.08	1.16	1.27	1.44
↳ Frequency Scaling	1.00	1.00	0.99	0.96	0.90

POP Metrics (Cont.)

POP MPI Performance Metrics for Incompressible immerflow ROI- 100 iterations

Number of cores	48	96	192	384	768
Global Efficiency	0.93	0.94	0.93	0.84	0.76
↳ Parallel Efficiency	0.93	0.91	0.87	0.77	0.68
↳ Load balance	0.99	0.98	0.98	0.97	0.95
↳ Communication Efficiency	0.94	0.92	0.89	0.79	0.72
↳ Serialisation	0.95	0.94	0.92	0.85	0.81
↳ Transfer efficiency	0.99	0.99	0.97	0.94	0.89
↳ Computational Scaling	1.00	1.03	1.07	1.09	1.12
↳ Instruction Scaling	1.00	0.99	0.97	0.95	0.92
↳ IPC Scaling	1.00	1.05	1.10	1.18	1.27
↳ Frequency Scaling	1.00	1.00	1.00	0.98	0.96

POP Metrics (Cont.)

- Same pattern for 10 and 100 iterations
- Parallel efficiency drops for higher number of cores
  - Serialisation is the major contributing factor

- Computational Scaling
  - IPC scaling
    - Compares how fast instructions are executed for different number of processes
    - Improves on higher number of cores
    - Likely due to better cache access

  - Instruction scaling
    - Compares total amount of work for different number of processes
    - Drops by about 8% on 768 cores but still good
      - Duplicated computation?
      - Extra computation needed to manage parallelism?

- **Main conclusion: serialisation is the main factor that limits scalability**
  - This assessment focuses on serialization

Critical Path Profile

- "`StencilScalarSolver::solve()`" is the most expensive activity on the critical path
  - More than 70% of "`computation()`" time was spent in this function (for all cases)
  - Further analysis showed that the cost is associated with the call to PETSc solver

Own root percent

Call tree | Flat view

- 1.71 gloria
  - 1.69 MPI_Init
  - 0.00 MPI_Comm_size
  - 0.00 MPI_Comm_rank
  - 17.81 void preprocess()
  - 78.78 void computation()
  - 0.00 void postprocess()
  - 0.01 MPI_Allreduce
  - 0.00 MPI_Barrier
  - 0.00 MPI_Finalized
  - 0.00 MPI_Comm_free
  - 0.00 MPI_Finalize

75.99 void stepAdvance()

↳ 62.08 void solvePressureSystem(double)

`incompressible::solvePressureSystem()`

↳ 60.69 void m_pressureSolverWrapper()

`StencilScalarSolver::solve()`
`Called as m_pressureSolver->solve()`

Inclusive values (▶)
the value for all sub-elements are aggregated
Exclusive values (▼)
values cannot be subdivided further

- 31.96 void m_pressureSolverWrapper()
  - 0.30 MPI_Comm_size
  - 0.00 MPI_Comm_rank
  - 19.28 MPI_Allreduce
  - 0.00 MPI_Allgather
  - 0.47 MPI_Startall
  - 1.99 MPI_Start
  - 6.40 MPI_Waitany
  - 0.29 MPI_Waitall

Numbers report percentage of total wall-clock time spent by each activity on the application critical path
~16 millions cells, 100 iterations, **768 cores**
Total wall-clock time: 204.53 s, Preprocess() time: 36.42 s, Computation() time: 161.12 s

POP Metrics

POP MPI Performance Metrics for `m_pressureSolver->solve()` - 100 iterations

Number of cores	48	96	192	384	768
Global Efficiency	0.91	0.96	0.97	0.80	0.62
↳Parallel Efficiency	0.91	0.89	0.84	0.63	0.44
↳ Load balance	0.99	0.99	0.98	0.97	0.95
↳ Communication Efficiency	0.91	0.9	0.85	0.66	0.46
↳ Serialisation	0.93	0.92	0.89	0.72	0.54
↳ Transfer efficiency	0.98	0.98	0.96	0.91	0.86
↳Computational Scaling	1.00	1.08	1.16	1.28	1.41
↳ Instruction Scaling	1.00	0.99	0.96	0.94	0.91
↳ IPC Scaling	1.00	1.09	1.20	1.37	1.58
↳ Frequency Scaling	1.00	1.00	1.00	0.99	0.98

Poor serialisation limits scalability of this activity (and consequently the whole application) on higher number of cores

Delay Cost Analysis

- Now the question is "*what causes low serialisation efficiency in this code*?"
  - Knowing the cause helps the decision on the optimization approach
  - POP metrics confirms that overall load balance is good

- Serialisation
  - typically happens due to at least one process arriving early/late at synchronization point

- Scalasca calculates a delay cost metric
  - This metric highlights the root causes of serialization
  - Attributes processes' waiting time to the routines causing serialization
    - For exact definition see https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/help/scalasca_patterns-2.5.html#delay

- Additional analysis was undertaken:
  - Traced immerflow + bitpit
  - And analyzed *Delay Costs* in `computation()` phase

Cause of Serialisation

```
▼ 0.32 void solvePressureSystem(double)
  ▼ 0.00 void m_pressureSolverWrapper()
    ▶ 93.27 void solve()
```

SystemSolver::solve() from bitpit

```
▼ 0.32 void solvePressureSystem(double)
  ▼ 0.00 void m_pressureSolverWrapper()
    ▼ 21.09 void solve()
        0.61 MPI_Comm_size
        0.00 MPI_Comm_rank
        54.66 MPI_Allreduce
        0.00 MPI_Allgather
        0.57 MPI_Startall
        1.69 MPI_Start
        14.22 MPI_Waitany
        0.43 MPI_Waitall
```

Inclusive values (▶)
Exclusive values (▼)

Numbers report percentage of total delay cost in `computation()` phase
~16 millions cells, 10 iterations, **768 cores**

- In this run, 93.27% of the delay costs is attributed to the `SystemSolver::solve()`
  - The cost is related to PETSc solver, `KSPSolve(...)`
    - `vectorsReorder(PETSC_TRUE)` was not called and therefore does not contribute any cost
- The breakdown of the delay costs is shown on the right
  - With MPI_Allreduce being the most significant one (54.66%)
  - Followed by the computation part (21.09%)
- **The MPI collective calls and imbalanced computation regions within the PETSc solver are the main causes of the serialisation on 768 cores**

Conclusion

- Computation() phase
  - Poor serialisation limits scalability
    - Regions of computational load imbalance, between MPI synchronization points
    - Growing waiting time in MPI calls especially collectives
  - The PETSc solver call
    - is the most expensive call on the application critical path
      - 77.04% of computation() time on critical path, on 768 cores with 100 iterations
    - shows poor communication efficiency (46%) on higher number of cores, mostly due to 54% serialisation
      - Main factor that limits the scalability of the application
    - shows about 9% drop in Instruction scaling, not perfect but still very good
    - **To improve the strong scalability, you need to optimize this part of the computation**
      - **e.g. alternative preconditioner/solver**

Recommendation

- Adjustable solver Parameters affect performance
  - No good guidance on how to use them, but experimenting with these parameters can pay off
- The **restart** parameter:
  - Larger values improve convergence rate (fewer iterations) but make each iteration more expensive
  - GMRES requires roughly O(restart^2) reductions per iteration.
  - Given that MPI reduction operations contribute significantly in the serialisation, setting the GMRES restart to a lower value might improve the scalability of the application.
- The **Additive Schwarz overlap** parameter:
  - Larger values improve converge rate
  - The trade-off is the increase in point-to-point message sizes between neighbors in the communication graph.
- It is worthwhile to experiment with
  - Smaller values of ***GMRES restart***, e.g. 5 or 10 rather than the default value of 30
  - Larger values of ***Additive Schwarz overlap*** rather than the default value of 1

Recommendation (Cont.)

- Is flexible GMRES (FGMRES) needed for your problem?
  - Two main reasons to use a flexible solver rather than the standard version:
    1. The preconditioner step is itself another krylov method
    2. The preconditioner will change between iterations

- If GMRES and FGMRES take the same number of iterations, then GMRES is usually faster
  - Each iteration of standard GMRES requires less memory reads and writes (about half as many)

- Can you use **BiCGSTAB** instead?
  - BiCGSTAB fluctuates a lot, but it only used 4-5 reduction operations per iterations
  - Flexible BiCGStab (KSPFBCGS) is also available

POP Metrics – Preprocess()

- In our tracing, preprocess() time increased up to 100 sec for 1000 iterations on 768 cores
  - not negligible with respect to total runtime
- Optimization may help depending on configurations for production runs
  - POP metrics highlight the main issues (instruction scaling, load imbalance and serialisation)
  - Starting point for optimization: `solver::initializeGrid(const run_status_t &)`

preprocess() - POP metrics	48	96	192	384	768
Global Efficiency	0.71	0.60	0.44	0.22	0.06
↳ Parallel Efficiency	0.71	0.71	0.64	0.54	0.4
↳ Load balance	0.74	0.8	0.77	0.71	0.7
↳ Communication Efficiency	0.96	0.89	0.83	0.76	0.57
↳ Serialisation	0.97	0.92	0.88	0.83	0.65
↳ Transfer efficiency	0.99	0.97	0.94	0.91	0.87
↳ Computational Scaling	1.00	0.85	0.69	0.41	0.15
↳ Instruction Scaling	1.00	0.82	0.71	0.51	0.29
↳ IPC Scaling	1.00	1.05	1.08	1.09	1.13
↳ Frequency Scaling	1.00	0.98	0.90	0.74	0.47

Potential Work for POP Proof of Concept

- Scaling acceptable to ~ 16 nodes (compressible and incompressible)
- If Improved performance is desired …

POP Proof-of-Concept targeting:

- immerflow
  - Identify cause of instruction scaling losses (both)
  - Investigate cause of IPC scaling losses (compressible)
- Bitpit
  - Investigate `KSP` solver performance improvements (incompressible)
  - Improve data buffering efficiency (both)
  - Load balance improvements? (both)
- Re-analyse with different computational problem(s)

Performance Optimisation and Productivity

A Centre of Excellence in HPC

Contact:

https://www.pop-coe.eu

mailto:pop@bsc.es

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 676553 and 824080.