



## ParFlow performance assessment report

### Document Information

Reference Number	POP_AR_17
Author	Ilya Zhukov (JSC)
Contributor(s)	Brian Wylie (JSC)
Date	21.07.2016

**Notices:** The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 676553.



©2015 POP Consortium Partners. All rights reserved.



# Contents

<b>1</b>	<b>Background</b>	<b>3</b>
<b>2</b>	<b>Behavior and syntactic structure</b>	<b>3</b>
<b>3</b>	<b>Focus of interest (FOI)</b>	<b>5</b>
<b>4</b>	<b>Scalability</b>	<b>5</b>
<b>5</b>	<b>Memory</b>	<b>8</b>
<b>6</b>	<b>Parallel Efficiency Metrics</b>	<b>9</b>
<b>7</b>	<b>Load Balance</b>	<b>9</b>
<b>8</b>	<b>Serial performance</b>	<b>10</b>
<b>9</b>	<b>Communications</b>	<b>10</b>
<b>10</b>	<b>Summary of observations</b>	<b>10</b>



## Background

**Applicants Name:** Wendy Sharples

**Institution:** Jülich Supercomputing Center

**Application Name:** ParFlow, version 693

**Programming Language:** C, Fortran

**Programming Model:** MPI

**Source Code Available:** yes

**Input data:** idealized testcase

**Performance study:** Scalability

**User description:** ran into (perceived) memory issues when running a weak scaling test with a load balanced idealized test problem. I got stuck somewhere above 512 nodes, we are aiming for scalability up to at least 2048 nodes. We would really like to confirm what the bottleneck is. A place where to look perhaps for a (serious?) performance bottleneck in ParFlow, could be the "KINSolMatVec" routine where (maybe inefficient) matrix-vector multiplications are done.

**Application Description:** ParFlow (PARallel FLOW) is an integrated hydrology model that simulates surface and subsurface flow. ParFlow is a parallel simulation platform that operates in three modes, i.e. steady-state saturated, variably saturated and integrated-watershed flow. ParFlow is especially suitable for large scale problems on a range of single and multi-processor computing platforms.

**Testcase Description:** weak scaling testcase with computational grid of size 50x50x40 and 10 time steps without file IO.

**Machine Description:** Juqueen is a scalable IBM BlueGene/Q system consisting of 28 racks each with 1024 PowerPC A2 processors with 16 GB of memory and 16 cores supporting 4-way hardware threading, and a custom 5D torus interconnect. Application is running under Linux microkernels on compute nodes using IBM XL compilers and proprietary MPI library with GPFS filesystems.

**Analysis tools:** Score-P/2.0.2, Scalasca/2.3.1, PAPI/5.3.0 (following hardware counters were collected: PAPI\_TOT\_CYC, PAPI\_TOT\_INS, PAPI\_TOT\_IIS, PAPI\_FP\_OPS, PAPI\_VEC\_INS, PAPI\_L1\_DCM, PAPI\_BR\_MSP, PAPI\_BR\_PRC).

## Behavior and syntactic structure

The core component of ParFlow is a multigrid solver (`Solve`). Typical solver's workflow consists of several steps, i.e. initialization of the possible solver (`SolverRichardsInitInstanceXtra`), problem initialization for the specific input (`KINSolInit`) and actual solver loop. The last two steps reside in `KINSol` routine.

Solver loop performs the computational process of computing an approximate solution (`KINSpgrmrSolve`). Intermediate solution is updated every iteration (`KINLineSearch`) until desired convergence or tolerance will be reached (`KINStop`).

For simplicity reasons, those three aforementioned phases, i.e. initialization of the solver, problem initialization and solver loop will be annotated in the source code and called `init_solver`, `init_problem` and `solver_loop` respectively. Pseudo code is shown in the Listing 1, where we can see that `init_solver` called only once at the very beginning, whereas `init_problem` called as many times as required number of timesteps. And for each timestep `solver_loop` iterates until required tolerance is achieved.

Listing 1: "Pseudo code of ParFlow's solver"



```

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

```

**Solve** routine in provided testcase on 32 nodes (1024 MPI processes) constitutes 99% of the total CPU time (375 seconds), whereas 8% of total execution time was spent in **init\_solver** (31 seconds), 3.3% in **init\_problem** (14 seconds), 88% in **solver\_loop** (351 seconds). Detailed information with approximate percentage of total execution time of ParFlow's specific regions is provided in the Table 1. The Table 1 shows that significant part of the total execution time was spent in the computation and only small amount of time in MPI communication, i.e. 5% in point-to-point operations within **solver\_loop**, and 2% in collective operations.

Table 1: Approximate percentage of total execution time of ParFlow's specific regions (1024 MPI processes)

Part of application	Percentage of total execution time, %			
	Computation	MPI communication operations		Rest of MPI
		point-to-point	collective	
<b>init_solver</b>	8	0	0	0
<b>init_problem</b>	3	0	0	0
<b>solver_loop</b>	80	5	2	1
Rest of the application	1	0	0	0
Application in total	92	5	2	1

Further breakdown shows that 5.4% of the total time spent in **GetGridNeighbors** (in **init\_solver**), 1.84% in **NlFunctionEval** (in **init\_problem**), 26.15% in **RichardsJacobianEval**, 20.23% in **PhaseRelPerm**, 11.34% in **Saturation**, 5.9% in **NlFunctionEval**, 4.5% in **PFMG**, 2.91% in **PFMGInitInstanceXtra** (in **solver\_loop**). Small amount of time was spent in MPI routines, i.e. 3.3% in **MPI.Waitall**, 2% in **MPI.Allreduce**, 1.3% in **MPI.Startall**. The most frequently called MPI routine was **MPI.Comm\_rank** (1198118 calls per process). **MPI.Comm\_rank** was called by **HYPRE\_PFMGSetup** and **HYPRE.StructGrid** routines from **HYPRE** library.

Besides, it was noticed that ParFlow frequently writes to the log files some debug output and regularly flush the stream with **fflush**. Such behavior can be very intrusive and can cause significant load imbalance.

## Focus of interest (FOI)

The Table 1 summarizes that provided testcase spent small amount of time in MPI communication (7% of total time). More insight can be achieved with analysis of computation, communication and load balance at bigger scale.

As it was mentioned in the previous paragraph there are three possible regions of interest. Although `init_solver` has relatively small execution time (8%), it executes only once at the initialization phase. On the contrary, `init_problem` and `solver_loop` are executing several times according to initial input.

In the Figure 1 execution timeline of ParFlow shows 10 time steps on one node (16 MPI processes), where green, yellow and dark blue colors represent `init_solver`, `init_problem`, and `solver_loop` respectively. `init_problem` was called 10 times (number of timesteps), whereas `solver_loop` executed 32 iterations (2-3 iterations per timestep, the number of iterations is not equally distributed since ParFlow uses non-linear solver). Further investigation of `solver_loop` iterations shows that the execution time is not equally distributed. More or less stable behavior can be reached after 5-6 time steps.

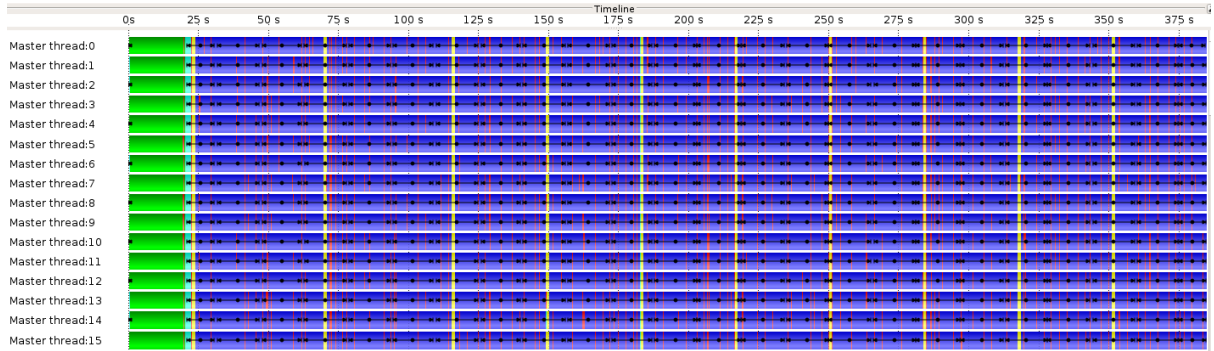


Figure 1: Timeline of ParFlow shows 10 time steps on one node (16 MPI processes), where green, yellow and dark blue colors represent `init_solver`, `init_problem`, and `solver_loop` respectively.

Thus, the main focus of interest will be concentrated on two regions, i.e. `init_problem` and `solver_loop`. Besides, aforementioned issue with writing debug output will be addressed. Detailed output will be commented out in `init_problem`, and `solver_loop`. Comparison of execution time for original and modified versions will be provided in the following paragraphs, where the modified version means a version of the ParFlow code where the logging statements were commented out in `init_problem` and `solver_loop`.

## Scalability

Figure 2 displays the weak scaling speedup and efficiency of entire execution time of ParFlow, for uninstrumented original and modified versions. Modified version scales slightly better than the original one. It is clear that the provided testcase is not scaling. Actual execution time (blue line) increases with the amount of processes that is far away from the expected behavior (red line).

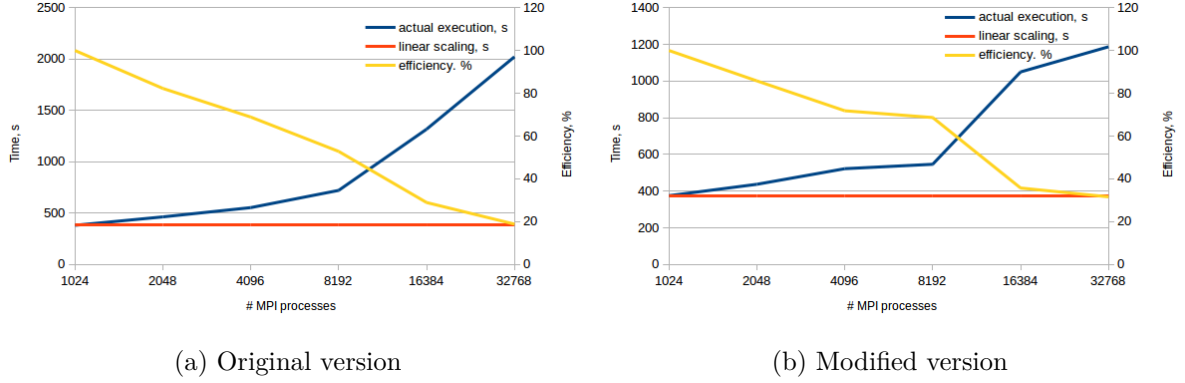


Figure 2: Speedup and efficiency of uninstrumented version of ParFlow (weak scaling). In the modified version the logging statements were commented out in `init_problem` and `solver_loop`.

Breakdown of regions of interest for uninstrumented versions of ParFlow provided in Figure 3. Behavior of all chosen regions are more or less similar, i.e. execution time increases with amount of processes. `init_problem` region (yellow line) shows slightly better performance than the others and it is close to perfect in the modified version. Modified version of `solver_loop` (green line) shows slightly better performance than the original one especially at the bigger scale.

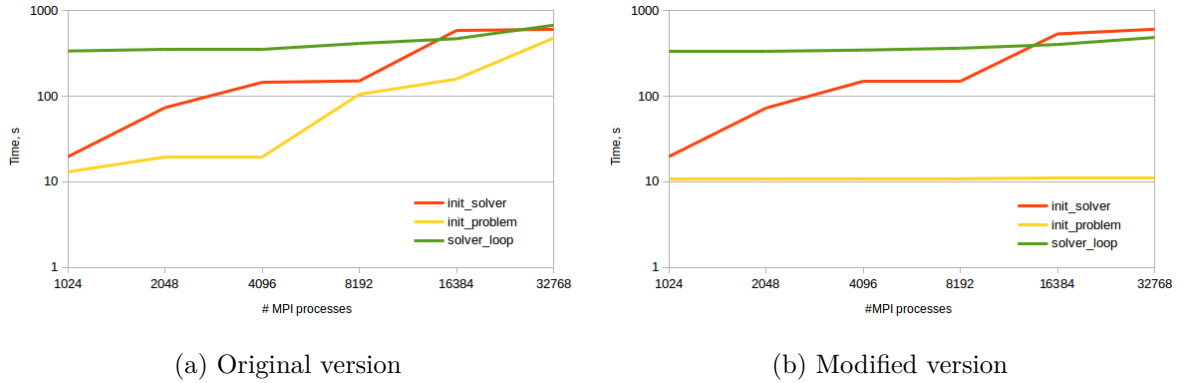


Figure 3: Breakdown of regions of interest of uninstrumented version of ParFlow. In the modified version the logging statements were commented out in `init_problem` and `solver_loop`.

For more insight of the application it was necessary to instrument it with Score-P. Instrumentation usually introduces additional overhead. To cut down instrumentation expenses only user instrumentation was used, and very frequently called MPI routines like `MPI_Comm_rank` were excluded from the measurement.

Figure 4 depicts overhead introduced by instrumentation with Score-P. Up to 4096 MPI processes overhead is negligible. Therefore measurements above at bigger scale should be considered very carefully.

Breakdown of regions of interest for instrumented versions of ParFlow provided in Figure 5. Behavior is very similar to what was shown in Figure 3 for uninstrumented measurements.

Customer expressed particular interest in behavior of `KINSolMatVec` routine. Therefore additional measurements of the execution time of `KINSolMatVec` have been done and showed

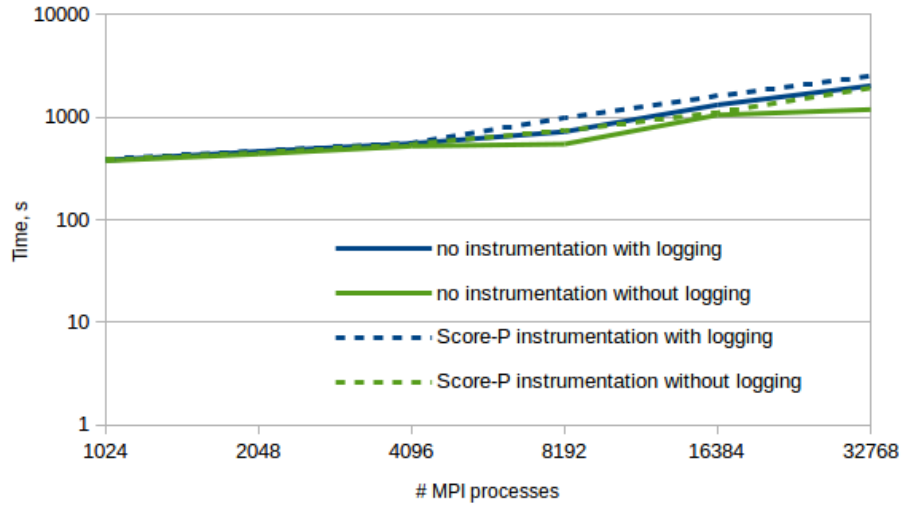


Figure 4: Measurement overhead introduced by Score-P is negligible up to 4096 MPI processes.

that the routine scales almost perfect for the weak testcase in the original and modified versions of ParFlow (blue line).

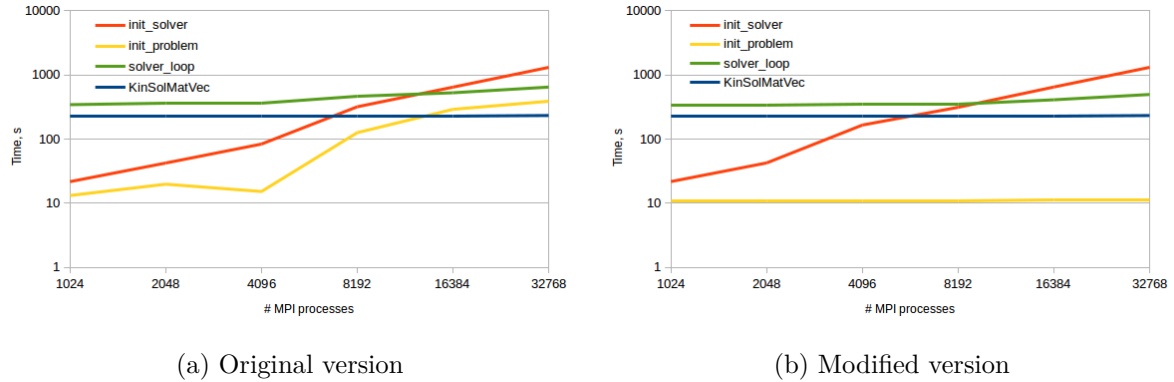


Figure 5: Breakdown of regions of interest of instrumented version of ParFlow. In the modified version the logging statements were commented out in `init_problem` and `solver_loop`.

Figure 6 displays in logarithmic scale breakdown of computation and communication of regions of interest based on Score-P measurements. In original version only computation part of `solver_loop` remains more or less constant (solid green line), whereas other regions are increasing. In the modified version both parts of `solver_loop` (green lines) and both parts of `init_solver` (red lines) are growing with increasing amount of processes. In modified version `solver_loop` parts are growing not so steep as in the original one.

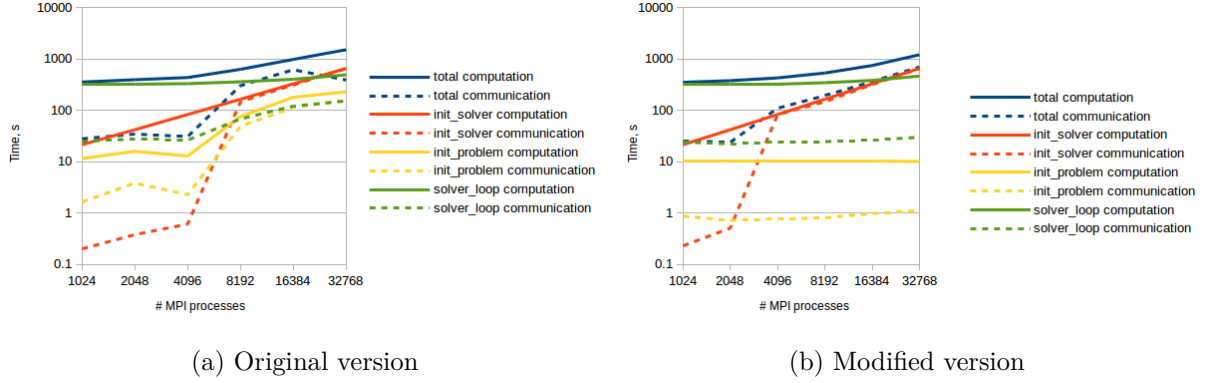


Figure 6: Breakdown of computation and communication of regions of interest. In the modified version the logging statements were commented out in `init_problem` and `solver_loop`.

Further breakdown of communication routines based on Score-P measurements (see Figure 7) revealed the main scalability breakers. In the original version those are `MPI_Allreduce` in `init_solver` (red line) and `init_problem` (yellow line), and `MPI_Waitall` in `solver_loop` (solid green line). In the modified version `MPI_Allreduce` in `init_solver` behaves similar, whereas `MPI_Allreduce` in `init_problem` and `MPI_Waitall` in `solver_loop` looks more smooth than in the original version. Improvement of `MPI_Allreduce` in `init_solver` can improve overall scalability.

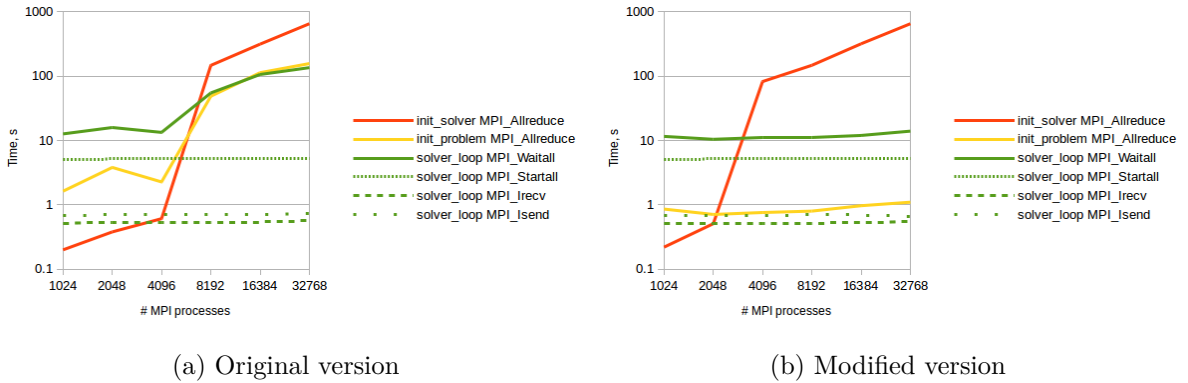


Figure 7: Further breakdown of communication of regions of interest.

As modified version showed better performance, we will consider it in the following paragraphs.

## Memory

In the audit request customer mentioned memory issues. Therefore Score-P 2.0.2 was used to track memory usage of the ParFlow. Figure 8 shows that memory consumption is increasing with the increasing number of MPI processes, e.g. approximately 2.3GB of memory was used per node for 1024 processes and 5.87GB for 32768 processes, whereas 16GB per node is available. The largest testcase with 65536 processes crashed at the beginning due to memory allocation failure. But as it is shown in Figure 8 used memory is growing and could approach the limit with 65536 MPI processes. Memory usage within compute node was almost constant. In addition,



it was detected that `NewGrid` routine has some memory leaks. The most memory consuming routines are: `GetGridNeighbors`, `PFMGInitInstanceXtra`, `KinSolPC`, `AllocateVectorData`.

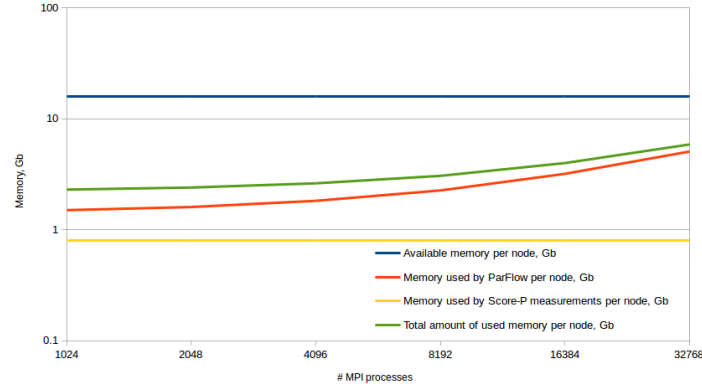


Figure 8: Memory usage of ParFlow

## Parallel Efficiency Metrics

Basic parallel efficiency metrics are shown in the Table 2. The higher the value the better is the efficiency. *Load balance* is a relation between average computation and maximal computation time. *Communication* is a relation between maximal computation and maximal executing time. *Parallel efficiency* was computed as a product of *Load balance* and *Communication*.

Table 2: Parallel efficiency metrics, %

# MPI processes	Efficiency metrics		
	Load balance	Communication	Parallel efficiency
1024	96.33	96.84	93.29
2048	97.19	96.81	94.08
4096	97.15	81.95	79.61
8192	97.88	74.98	73.39
16384	98.01	68.71	67.34
32768	98.32	64.45	63.36

The Table 2 gives an overview of the parallel efficiency of provided testcase and shows that application scales relatively good up to 2048 processes. From 4096 processes it has severe communication issue. As it was shown in the Figure 7 that issue was mostly caused by `MPI_Allreduce` in `init_solver`.

## Load Balance

As it was shown in the Table 2 load balance is good in general. Therefore it worth to concentrate on problematic part, i.e. communication and serial performance.



## Serial performance

Serial performance analysis for provided testcase with 1024 MPI processes was done by means of measurements with various hardware counters. Low IPC values, e.g. `solver_init` (0.31 out of 2) and `solver_loop` (0.31), indicate that there is space for improvement of serial performance. Potential reasons for low IPC can be pipeline stalls, cache misses, mispredicted branches etc. Therefore additional measurements with hardware counters were collected.

Analysis shows particular high values of cache misses in L1, stalls and mispredicted branches in the following routines `RichardsJacobianEval`, `PhaseRelPerm`, `Saturation` and `NlFunctionEval`.

It is necessary to mention that Juqueen has in-order instruction execution, i.e. instructions are fetched, executed and committed in compiler-generated order. If one instruction stalls, all instructions behind will stall. Branching on Juqueen is very expensive and can cause pipeline stalls. Therefore routines with big amount of mispredicted branches will have a big amount of stalls and low value of IPC.

## Communications

As it was shown in the Figures 6 and 7, communication time grows with increasing amount of processes. The main contributor to communication time is `MPI_Allreduce` in `init_solver`. To reduce communication and overall performance it is necessary to reduce time in aforementioned routine.

For example communication time (testcase with 32768 MPI processes) constitutes 36.64% of the total time. One MPI call has the biggest impact, i.e. `MPI_Allreduce` in `init_problem` constitutes 34.08% of total time (values per process: 676.34 seconds maximal time, 652.44 seconds average time, 3 calls, 0.5 MB transferred). Scalasca trace analysis identified major wait-state pattern, i.e. "Wait at NxN" which appears in `init_solver` (`MPI_Allreduce` called by `HYPRE_StructGrid`).

## Summary of observations

From the performance analysis of the ParFlow it is possible to conclude the following:

- Provided testcase has relatively small amount of communication at small scale, e.g. with 1024 MPI processes. Communication time grows considerable at scale. The main contributor to communication time is `MPI_Allreduce` in `init_solver`.
- Scalasca trace analysis identified major wait-state pattern, i.e. "Wait at NxN" which appears in `init_solver` (`MPI_Allreduce` called by `HYPRE_StructGrid`).
- Testcase with less verbose logging and stream flushes has better performance.
- Analysis did not detect any significant memory issues. Memory consumption is increasing at scale. Memory leak was detected in `NewGrid`.
- Value of IPC is rather low. It can be caused by pipeline stalls, L1 cache misses, mispredicted branches.

To improve overall performance of the application the following changes are recommended:



- Try to modify application to avoid logging and explicit stream flushes. Include it only for debug runs.
- Potential candidates to improve serial performance: `RichardsJacobianEval`, `PhaseRelPerm`, `Saturation` and `NlFunctionEval`. Examine those routines, and reduce amount of mis-predicted branches if possible.
- Prefetching options for L1 cache which are available on Juqueen may improve amount of cache hits.
- `MPI_Comm_rank` called very often by `HYPRE_PFMGSetup` and `HYPRE_StructGrid` routines from HYPRE library.
- Examine `NewGrid` for memory leaks.