Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Numerical model for the thermal energy storage tank with integrated steam generator (TES-SG) based on the OpenFOAM software package

A.Charalampidou[a,b], P.Daoglou[a,b], J.Hertzer[c], E.V.Votyakov[d]*

[a]*Greek Research and Technology Network, Athens, Greece*
[b]*Scientific Computing Center, Aristotle University of Thessaloniki*
Thessaloniki 54124, Greece
[c]*HLRS, Nobelstr. 19, D-70569 Stuttgart, Germany*
[d]*The Cyprus Institute, 20 Konstantinou Kavafi Street*
2121 Aglantzia, Nicosia, Cyprus

Abstract

The project objective has been to develop and justify the OpenFOAM model for the simulation of a TES tank. In the course of the project we have obtained scalability results, which are presented in this paper. Scalability tests have been performed on HLRS Hermit HPC system using various combinations of decomposition methods, cell capacities and number of physical cpu cores.

1. Introduction

Within this project, a realistic 3D model which solves jointly Navier-Stokes and heat conduction equations including buoyancy effect with Boussinesq approximation has been developed based on the OpenFOAM CFD toolbox.

The CFD simulations have been carried out using the open source toolbox OpenFOAM (Open Field Operation and Manipulation) version 2.1.1. The OpenFOAM (Open Field Operation and Manipulation) CFD Toolbox [1] is a free, open source CFD software package, released under the GNU General Public Licence. It is developed by OpenCFD Ltd and distributed by OpenFOAM Foundation. OpenFOAM has an extensive range of features to solve anything from complex fluid flows involving chemical reactions, turbulence and heat transfer, to fluid dynamics and electromagnetics[2]. It is written in C++ and apart from being a ready to use CFD software, it can be also thought as a framework, which allows programmers to build their own code, as it provides them with the abstraction sufficient to think of a problem in terms of the underlying mathematical model [3].

Furthermore, OpenFOAM has parallel computing capabilities, which provide the opportunity to simulate problems of greater complexity, such as the one considered in this project, more quickly and with greater accuracy. The method of parallel computing used by OpenFOAM uses the public domain OpenMPI implementation of the standard Message Passing Interface (MPI). Parallelisation is robust and integrated into OpenFOAM at a low level, so in general, new applications are able to run in parallel by default, without the need of parallel specific coding.

Performance results in this study have been obtained from benchmark tests performed on HLRS *Hermit* [4] supercomputer. The initial motivation for this study on *Hermit* (CRAY) machine has been scalability results that were obtained on another CRAY machine beforehand for the OpenFOAM tutorial case [5]. Similar scalability has also been observed in this study.

2. Description of the model

The 3D numerical model that was designed and optimized with the usage of the OpenFOAM CFD toolbox is a thermal energy storage (TES) tank with integrated steam generator (SG). The model includes heat and mass transfer partial differential equations solved by means of the OpenFOAM. Specifically, the finite volume method (FVM) is used.

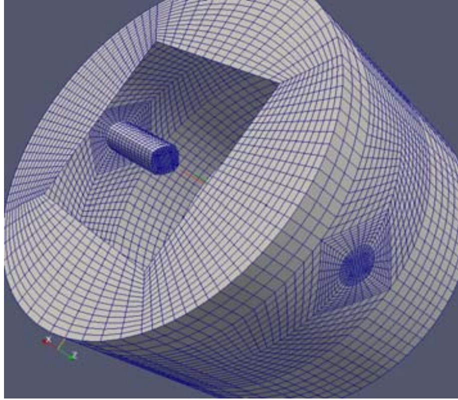


Figure 1: Thermal Energy Storage tank with integrated inlet pipe.

TES is a technology to store thermal energy in a reservoir for later use. The numerical model will be a part of the OPTS project (Optimization of Thermal Storage with integrated steam generator), which aims at developing a new TES system based on single tank configuration using stratifying Molten Salts (MS, Sodium/Potassium Nitrates 60/40 w/w) as the heat storage medium at temperatures reaching 550 °C. The TES tank is integrated with a Steam Generator (SG) to provide efficient, reliable and economic energy storage for the next generation of trough and tower plants.

3. Description of construction and solution steps

The use of the OpenFOAM utilities for the construction and solution of the model is briefly described here. The functions that have been used (in the order that they have been executed) are the following ones:

*a. **blockMesh**:* BlockMesh is a mesh generation utility. The utility reads the dictionary file **blockMeshDict**, generates the mesh and writes out the mesh data to points and faces, cells and boundary files [6].

*b. **decomposePar**:* In order to run OpenFOAM in parallel on distributed processors, the mesh is decomposed using the OpenFOAM decomposePar utility. DecomposePar decomposes the domain according to **decomposeParDict** dictionary file which is present in the **system** subdirectory of the case, assigning one domain per process. The numberOfSubdomains variable specifies the number of processors used to solve the case [7]. The decomposition method applied is also declared within decomposeParDict file. Decomposition methods that have been tested in this study are **manual**, **scotch** and **simple**.

*c. **buoyantBoussinesqPimpleFoam**:* OpenFOAM **buoyantBoussinesqPimpleFoam** solver [8] is a transient solver for buoyant, turbulent flow of incompressible fluids. The buoyantBoussinesqPimpleFoam solver uses the Boussinesq approximation:

$$rho_k = 1 - beta(T - T_{ref})$$

where:

rho_k= the effective (driving) kinematic density beta = thermal expansion coefficient $[1/K]$
T = temperature $[K]$
T_{ref} = reference temperature $[K]$

Valid when:

$$\frac{beta(T - T_{ref})}{rho_{ref}}_k \ll 1$$

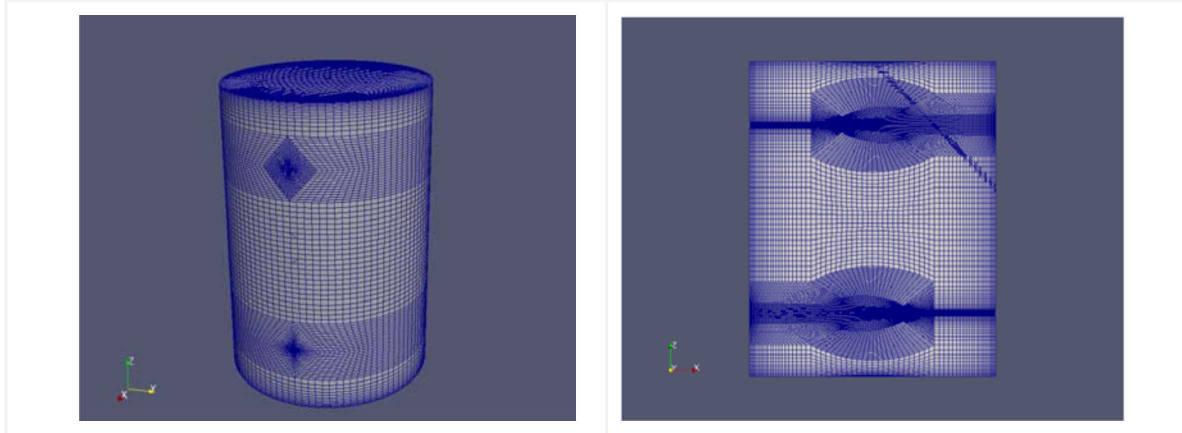
The solver step has been executed in parallel on Hermit on the allocated nodes for each run using the *aprun* application launcher.

*d. **reconstructPar**:* The reconstructPar utility is finally used to reconstruct the case that has been run in parallel [9]. All the sets of time directories from each processor directory are merged into a single directory.

4. Mesh construction and decomposition

In this section we describe the 3D structured mesh for the realistic system that has been developed.

The mesh consists of macroscopic hexahedral blocks of different shapes and geometric sizes. In OpenFOAM notation, hexahedra are polyhedral blocks with six faces, which might be curved and they are defined by eight vertices. The blocks are subdivided in cells by the OpenFOAM blockMesh utility. The amount of cells in each block is proportional to N^3, where N is a characteristic number (number of partitions of the typical edge in the system).



Figures 2,3: Mesh N=16 for the system studied

Hexahedra are described in a so called blockMesh dictionary file. This is an ASCII file formatted according to blockMesh specifications. It is important that the generation and partition of the hexahedra in cells are proceeded in the order given by the order of lines in the blockMesh dictionary file, that is, block 1, block 2, .. ,etc. So, in the trivial case block 1 can be assigned to core 1, block 2 to core 2, and so on.

To prepare a blockMesh dictionary file we used m4 pre-processor language, which allows us to parameterize the mesh. Then, any change of geometric sizes of the system, inlet location, boundary layer thickness, mesh resolution, and any other needed tank properties can be controlled by the parameters specified in one place. So, the work to generate the mesh was as follow: (1) the preparation of m4 script, which must be done once; (2) the blockMesh dictionary generation at the given mesh parameters; (3) mesh generation from the blockMesh dictionary file.

As we mentioned in the previous section, the decomposePar utility is used in order to split the mesh into a number of sub-domains and allocate them to separate processors. OpenFOAM provides the user with the choice of several decomposition methods. We analyse here shortly the methods that were used in this study:

a. manual

This decomposition method provides the user with the option to specify directly the allocation of cells to a particular processor.

Using this method we have achieved an **equally loaded decomposition** of our mesh. Thus, all cores may handle an equal number of cells, and the communication time between adjacent cores (blocks) is minimal. Computationally, to work on such a structured mesh is the same as to work on the equally distant cubic grid.

In order to implement this specific decomposition strategy, before the decomposition step, a script was used in order to calculate the number of cells per process for each different mesh capacity and setup the files system/decomposeParDict, system/decomposeParBlock and constant/cellDecomposition.

b. scotch

Scotch decomposition method does not require geometric input from the user and attempts to minimise the number of processor boundaries [9].

d. simple

In the case of simple geometric decomposition, the domain is split into pieces by x, y and z directions. For example, the table below demonstrates the configurations that were used for mesh with characteristic number N=32 and different numbers of processors.

Configuration Number	Number of processors	#Sub-domains X direction	#Sub-domains Y direction	#Sub-domains Z direction
1	512	4	2	64
2	512	4	4	32
3	512	8	4	16
4	512	8	2	32
5	512	2	2	128
6	1024	8	2	64
7	1280	8	2	80
8	2048	16	2	64

Table 1: Simple decomposition method - configurations for mesh N=32

5. Scalability tests using manual decomposition method

For the simple square grid and canonical problem (cavity flow) almost ideal scalability has been demonstrated on a CRAY machine [5] as long as the number of cells for one core is sufficiently large.

In order to check our model's scalability, benchmark tests have been performed on the meshes of total capacity equal to $320N^3$cells with N=16, 32, 64. The largest mesh (N=64) consists of 83,886,080 cells.

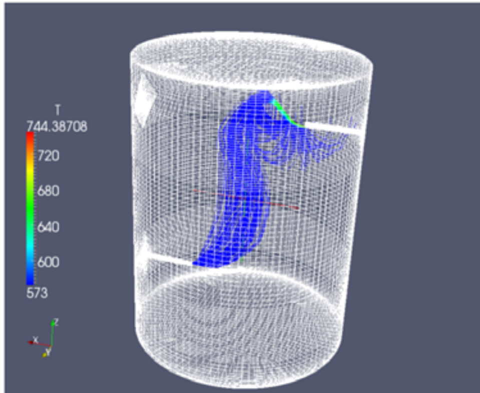


Figure 4: Typical flow structure from the inlet (upper pipe) to the outlet.

Though the geometry of our problem is not rectangular, because it consists of large cylindrical tank with small inlet and outlet pipes, we were able to map this complex computational domain onto blocks of proportional size. Thus, we have obtained the speedup in parallel simulations similar to those observed by other authors with OpenFOAM for a simple rectangular 2D problem. This speedup has been observed when the number of cells for one core is larger than 10^4. For reasons given below, work with double refined mesh with the same computational time requires factor 16 more cores, hence, this provides natural utilization of cores with larger mesh by keeping the same structure of the problem.

Scalability results are shown in figure 5 and the table below. The simulation runs, used to obtain the scalability data, correspond to laminar flow. For each N, the runs were started with the same initial condition and stopped when the physical time of the simulations reached time equal to 0.1sec. Double refinement of the mesh (each new N is larger with factor 2 than previous N) means that the new mesh is eight times larger because we are working on a 3D system. Moreover, in order to have the same stability factor (the same Courant number) the time integration step must be also twice less. Altogether, it results in 16 times slowing down when keeping the same number of cores in the double-refined mesh. In this respect, the system studied is highly suitable for massive parallel simulation, since the refinement of the mesh just means an appropriate increase of processors for parallel computations.

Mesh	#cores	Solver Cpu Time (seconds)	speedup
N=16	32	232	1
	64	90	2.57
	128	42	5.52
	256	27	8.59
N=32	256	845	1
	512	395	2.14
	1024	217	3.89
	1280	186	4.54
	2048	224	3.77
N=64	1024	5842	1
	2048	2559	2.28
	4096	1669	3.50

Table 1: timing results for solver step, using manual decomposition method

In figure 5, one can see that the achieved speedup is close to the ideal one. For the mesh N=32 (red curve with open circles), the speedup for 2048 cores is worse than what we have for 1280 cores because the number of cells for one core is too small for this mesh, and the communication cost between subdomains is large compared to the time of the internal simulation inside each subdomain. Therefore, in order to utilize properly the larger number of cores, one has to use a larger mesh, that is, mesh N=64 (blue curve with closed squares) which exhibits a fairly good scalability pattern. Actually, for the mesh N=64, one can see almost perfect scalability up to 4096 cores. We have found that speedup in our problem is observed when the number of cells for one core is larger than , which corresponds to 8192 cores for the mesh N=64.

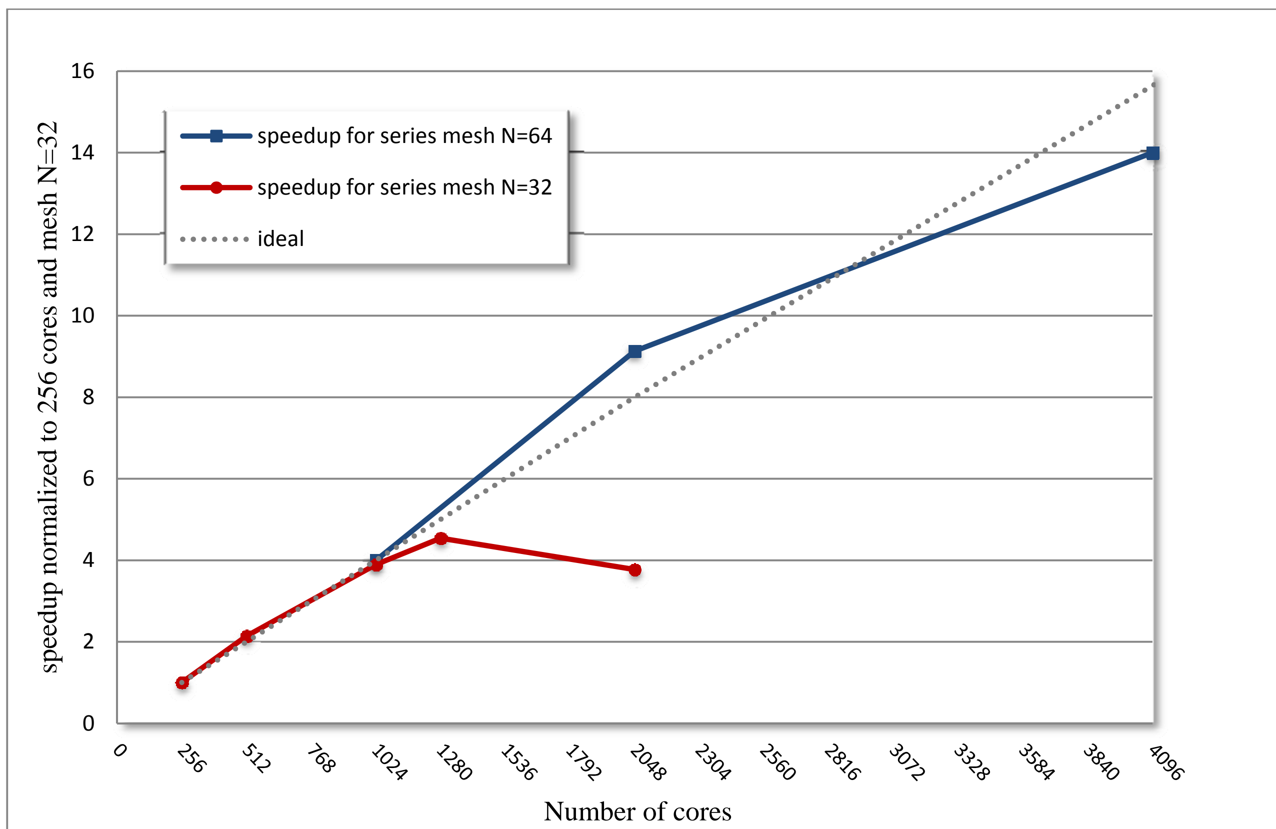


Figure 5: Speedup for mesh N=32 and N=64

6. Comparison of decomposition methods

Timing results for N=32, using 3 different methods for decomposition, are shown in Figure 6. As can be seen, for our case study, the scotch decomposition method provides best scaling overall. Similar results are obtained for mesh N=64. For this specific case (N=64), only the manual and scotch decomposition methods have been tested. When implementing the scotch approach in the decomposition step the solver scales better with an increasing number of cores. It should be noted that the results given in this section include only the timing results of the solver (parallel) step (not the decomposition and reconstruct steps).

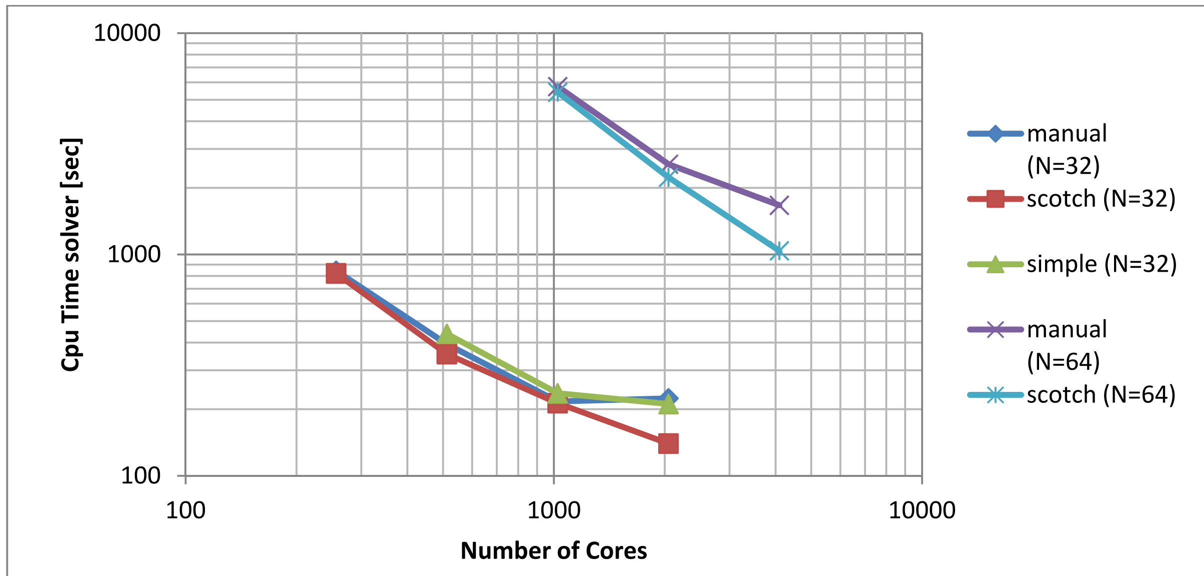
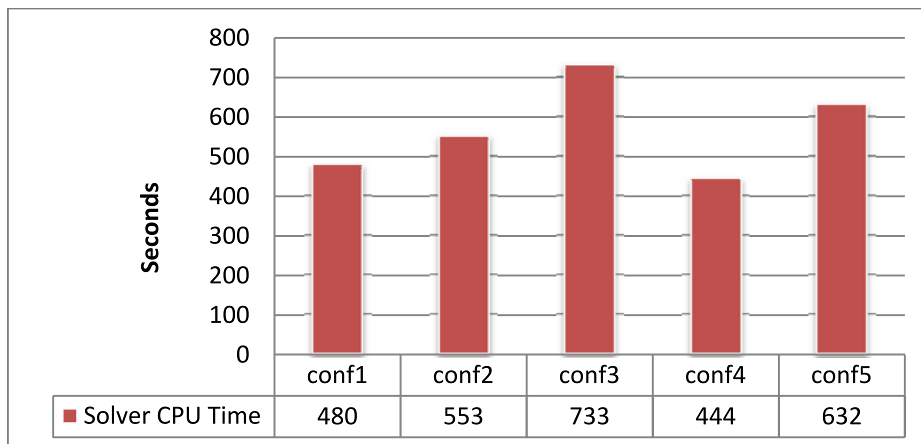


Figure 6: cpu time for solver step using different decomposition methods for mesh N=32 and N=64

Regarding simple decomposition method, figure 6 only includes results for configurations 4, 6 and 8 that are presented in table 1. In figure 7 the timing results for the solver using the simple decomposition method and in specific configurations 1-5 from table 1 are presented, showing that the best results are achieved when the domain is split in a greater number of sub-domains in the z direction.



	conf1	conf2	conf3	conf4	conf5
Solver CPU Time	480	553	733	444	632

Figure 7: cpu time for solver running in parallel on 512 processors when simple method has been used for decomposition (mesh N=32)

Regarding the time the decomposition step takes to run, some significant observations are presented in tables 3 and 4.

#cores	Decomposition Method	Decomposition cpu time (seconds)	Solver cpu time (seconds)
1024	manual	85	5734
	scotch	1890	5410
2048	manual	87	2559
	scotch	2121	2232
4096	manual	88	1669
	scotch	2409	1040

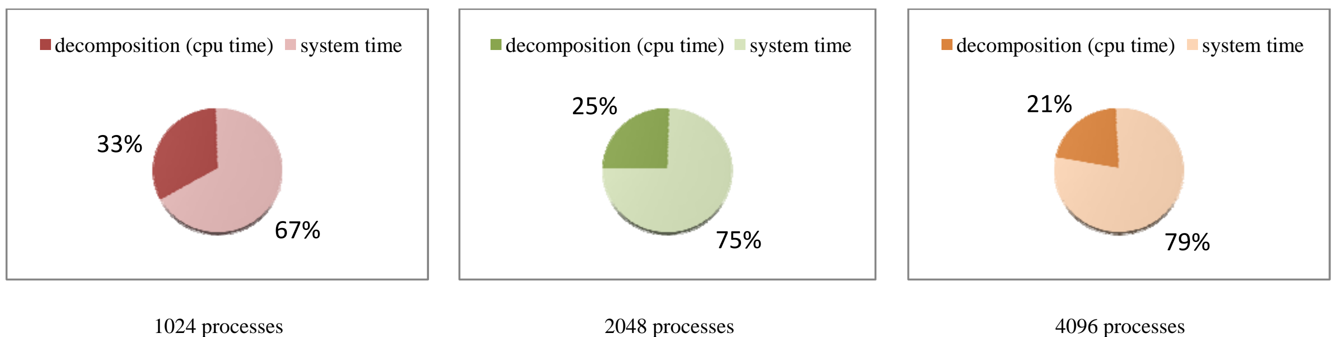
Table 3: cpu for decomposition and solver (mesh N=64)

It is clear that although the solver part had less time consumption when using the scotch decomposition method, the decomposePar utility takes much more cpu time to complete in this case than when using manual decomposition method. However, assuming the overall absolute wall time is the critical metric, timing results do not differ that remarkably, taking into consideration the results presented in table 4. Due to the limited resources in terms of cpu hours for this project, wall time results for manual decompositions were not obtained for 2048 and 4096 number of cores. However, we assume that manual decomposition method might be preferable.

#cores	Decomposition Method	Decomposition Setup wall time (seconds)	Decomposition wall time (seconds)	Solver wall time (seconds)	Total wall time (seconds)
1024	manual	1253	3990	5767	11010
	scotch	-	5744	5436	11180
2048	scotch	-	8335	2256	10591
4096	scotch	-	11291	1077	12368

Table 4: wall time for decomposition and solver (mesh N=64)

It was also found that system time (file creation and I/O) for decomposition step has been remarkably time consuming when compared to the overall wall time. Figure 8 shows that system time is increased with the number of processors used, due to the increasing number of files that are created and processed.



1024 processes 2048 processes 4096 processes

Figure 8: percentage of system time over total time spent for mesh decomposition (mesh N=64, scotch decomposition method)

Future work may involve further investigation on system time consumption for decomposition step.

7. Conclusions

OpenFOAM with this application proved to be suitable for current Tier-0 systems up to several thousand cores. The efficiency depends on the number of cells of the finite volume model used, increasing significantly with more detailed models, requiring greater number of cells. Therefore, for more complicated problems, which require more detailed models, OpenFOAM is expected to scale well on higher core numbers also.

References

[1] OpenFOAM main site, http://www.openfoam.com/
[2] OpenFOAM Foundation, http://www.openfoam.com/features/index.php
[3] Massimiliano Culpo, Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters, PRACE-1IP Whitepaper, http://www.prace-project.eu
[4] HLRS, Hermit User Guide, https://wickie.hlrs.de/platforms/index.php/Cray_XE6
[5] Computational Fluids Dynamics (CFD) Simulations at Scale. OpenFOAM open source applications. HPC|Scale Working Group, Sep 2010, http://www.hpcadvisorycouncil.com/pdf/OpenFOAM_at_Scale.pdf
[6] http://www.openfoam.org/docs/user/blockMesh.php
[7] http://openfoamwiki.net/index.php/DecomposePar
[8] http://foam.sourceforge.net/docs/cpp/a02826.html#details
[9] http://www.openfoam.org/docs/user/running-applications-parallel.php

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.