Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters

Massimiliano Culpo*

**CINECA, Via Magnanelli 6/3, Casalecchio di Reno (BO) I-40033, Italy*

Abstract

The scaling behavior of different OpenFOAM versions is analyzed on two benchmark problems. Results show that the applications scale reasonably well up to a thousand tasks. An in-depth profiling identifies the calls to the MPI_Allreduce function in the linear algebra core libraries as the main communication bottleneck. A sub-optimal performance on-core is due to the sparse matrices storage format that does not employ any cache-blocking mechanism at present. Possible strategies to overcome these limitations are proposed and analyzed, and preliminary results on prototype implementations are presented.

OpenFOAM

1. Introduction

OpenFOAM is an object-oriented toolkit for continuum mechanics written in C++ and released under the GNU-GPL license. More than being a ready-to-use software it can be thought as a framework for Computational Fluid Dynamics (CFD) 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 [1].

This key-feature fostered the wide acceptance of the code in the CFD community within both academic and industrial groups. As its usage spread over time OpenFOAM has been ported and tested over many different architectures, including massively parallel clusters [2]. The understanding of the scaling behavior of OpenFOAM on the latest PRACE Tier-0 and PRACE Tier-1 computers [3] becomes therefore essential to correctly exploit these resources and tackle in the best way possible scientific problems that would otherwise be of unmanageable size.

To provide our contribution to this understanding in the following we will analyze the scalability of two OpenFOAM solvers on problems that are considered to be good representatives for realistic production runs on Tier-0 systems. In Section 2 we will report on the architectures and the various installations used to test OpenFOAM, while in Section 3 we will present the actual benchmark results along with a first brief analysis. A more detailed discussion of the bottlenecks will be conducted in Section 4 with a strong emphasis on the linear algebra core libraries. A possible solution to overcome part of these bottlenecks will be discussed in Section 5, and some results on a prototype implementation will be given. Finally in Section 6 we will summarize the main points of the work, and propose possible future improvements to the code.

---

* Massimiliano Culpo. E-mail address: m.culpo@cineca.it

2. Installation notes

All the tests that will be presented in this document have been executed on two computers: CINECA PLX (PRACE Tier-1 system) [4] and TGCC CURIE (PRACE Tier-0 system) [5]. The particular choice of these two machines for the tests is dictated by the fact that the two clusters share similar processor technologies and node architectures. This will indeed permit a fair comparison of the results obtained on differently sized systems. In the following we will thus give the details of the OpenFOAM installations on both CINECA PLX and TGCC CURIE.

2.1. OpenFOAM on CINECA PLX

CINECA PLX is an IBM iDataPlex DX360M3 cluster currently ranked at the 82nd position in the TOP500 and 13rd position in the Green500. It consists of 274 IBM X360M2 12-way compute nodes with 2 Intel Xeon Esa-Core Westmere and 2 nVIDIA Tesla M2070 each. Every Intel processor has a clock frequency of 2.4 GHz, 24 GB of RAM and 12MB of L3 cache, while each one of the six cores is equipped with a 256KB private L2 cache and a 32KB L1 cache.

OpenFOAM versions 1.6-ext, 1.7.1 and 2.0.0 have been installed on CINECA PLX with executables built both with GNU 4.5.2 and Intel 12.1.0 C++ compilers. The standard MPI implementation linked to OpenFOAM is OpenMPI 1.4.4, even though Intel MPI 4.0 has also been tested. The code has been instrumented with the Integrated Performance Monitoring (IPM 2.0.0) library [6]. Notice that the version of the Linux kernel used in the compute nodes (2.6.18-238.el5) permits no gathering of the hardware counters. Thus, on CINECA PLX, IPM is not linked to the Performance Application Programming Interface (PAPI) library [7].

2.2. OpenFOAM on TGCC CURIE

TGCC CURIE is the first French supercomputer open to European scientists and one of the European PRACE Tier-0 system installations, currently ranked at the 9th position in TOP500 list. It consists of 360 S6010 BULLX fat-nodes and 16 BULLX B hybrid-chassis. The fat nodes are composed of 4 Intel Nehalem-EX X7560 processors with a frequency of 2.26GHz, 32GB of RAM memory and 32MB of L3 cache. Each of the eight cores of the processor has a 256KB private L2 cache and a 32KB L1 cache. The hybrid chassis contains 9 hybrid GPU B505 blades with 2 Intel Westmere and 2 Nvidia M2090 T20A.

OpenFOAM has been installed on TGCC CURIE with version 1.6-ext, 1.7.1 and 2.0.0 using GNU 4.5.1 and Intel 12.0.3 C++ compilers to build the executables. The MPI implementation to which the applications were linked is Bullx MPI 1.1.10.1. The code has been instrumented with IPM 2.0.0 and PAPI 4.1.3 libraries [6, 7].

3. Scalability results

In this section we will show the timing and performance of different runs on two test cases. The first one (Section 3.1) constitutes a simple and widely-used benchmark that will be thoroughly investigated to probe the impact of different choices on the scalability of the code. The second (Section 3.2) represents instead a more complex case that will be used to gain a better insight of what should be expected when solving real scientific and industrial problems.

3.1. Lid-driven cavity flow

The lid-driven cavity flow benchmark[b] involves the solution (using the icoFoam solver) of a laminar, isothermal, incompressible flow in a three-dimensional cubic domain. All the boundaries are modelled as walls and are considered static, with the exception of the top one that moves in the x direction at a speed of 1 m/s. Notice that, despite its simplicity, this benchmark is of particular interest as it is widely

[b] This benchmark problem has been constructed applying straightforward modifications to the two-dimensional lid-driven cavity flow tutorial located in the $FOAM TUTORIALS/incompressible/icoFoam/cavity/ directory of any OpenFOAM distribution

employed at different sites for benchmarking purposes [2, 8, 9], thus permitting a direct comparison of different application set-ups.

Following [2], a structured, uniformly spaced mesh has been adopted in all the strong scaling tests that have been performed while two different spatial discretization steps (returning respectively 100×100×100 and 200×200×200 cells) have been considered. To avoid any ambiguity in the subdivision of the domain for a given number of processors, the simple algorithm has been dropped in favour of the scotch decomposition method with uniform weights. Both the coarse and fine mesh configurations have been tested without I/O for 40 time steps ($\delta t = 5 \times 10{-}3$ sec.).

Table 1 shows the results of a first battery of tests performed on CINECA PLX. The overall aim is to compare different stable releases of OpenFOAM using as a figure of merit the wall-clock time (in seconds) needed to run the benchmark case. Notice that the executables of each OpenFOAM version have been built with both GNU and Intel compilers, to check if any performance gain is obtained using native compilers. The percentage of the wall-clock time spent in MPI communication is also reported. To ensure that the execution speed was not affected by the IPM instrumentation, two timings of the same run have been gathered. The first one (plain text in Table 1) stems from an execution with a non-instrumented version of the code, while the second one (bold text in Table 1) derives from an execution with an IPM instrumented version of the code. Finally, the upper part of the table refers to the 100×100×100 test case while the lower to the 200×200×200.

It can be readily seen that the lightweight instrumentation with the IPM library does not indeed affect the execution time of the runs. Furthermore no evident performance gain is obtained using Intel instead of GNU compilers. An explanation for this behavior may reside in the storage format employed for sparse matrix representation, which does not permit an efficient use of SSE vectorization. More details on this will be given in Section 4. The same scaling trend is exhibited by all OpenFOAM versions: a substantial loss of scaling efficiency is seen around a hundred tasks for the 100×100×100 case and around five hundred tasks for the 200×200×200 case. It should be noted that the small fluctuations in the timing results are most likely due to the different load of the whole cluster and to the different mapping of tasks upon nodes for each particular run.

In Table 2 a second set of measurements is reported, addressing the parallel efficiency of the executable on a single node.[c] For applications that (like OpenFOAM) heavily rely on the iterative solution of sparse linear algebra kernels [10], this kind of analysis is particularly important to understand how much the memory-bound nature of the code impacts on the overall performance. The deviation from a linear scalability behavior inside the node will be in fact the main cause of a sub-optimal parallel efficiency for small to medium size jobs as it will reduce the good inter-node scalability (inferred from Table 1) by a constant factor. As expected, the intra-node scalability is far from being optimal due to the saturation of the bandwidth to memory as the number of tasks increases. The same issue can also be noted from Table 3, presenting the results of an inter-node scalability study where the variable parameter is now constituted by the number of tasks spawned on each computational node. Here the small gain obtained in terms of execution time when spawning one task per core does not fairly compensate the improvement in the available memory per task obtained when spawning 8 or 4 tasks per node. Furthermore, spawning less than 12 tasks per node limits the growth of the wall-time fraction spent in MPI communication. Thus, extrapolating the results in Table 3, we may even expect a faster execution time when using 4 or 8 tasks per node instead of 12, for a sufficiently large number of nodes.

Figure 1 compares the performance of two different MPI implementations on the same benchmark, showing that no substantial differences are to be expected. Finally, for the sake of completeness, it should be mentioned that a study concerning the hardware acceleration of MPI collective communications has been presented in literature [11], showing a performance boost that grows with the scale of the problem.

3.2. Droplet splash

The droplet splash benchmark involves the solution of a two-phase incompressible system that models the impact of a droplet against a wall. The solver being used is in this case interFoam. Details about the mathematical model implemented in this solver can be found in [12].

[c] The tests have in this case been performed using an instrumented version of OpenFOAM 1.7.1 built with Intel compilers and linked to OpenMPI 1.4.4. The benchmark case taken into consideration is the one with 200×200×200 cells

PLX	GNU compilers (Gcc 4.5.2)						Intel compilers (Icc 12.1.0)					
	1.6-ext		1.7.1		2.0.0		1.6-ext		1.7.1		2.0.0	
Task #	Time	% MPI	Time	% MPI	Time	% MPI	Time	% MPI	Time	% MPI	Time	% MPI
1	1045	-	987	-	1121	-	1018	-	1258	-	1014	-
12	285(**287**)	**8.69%**	222(**226**)	**4.74%**	277(**277**)	**3.70%**	288(**285**)	**8.54%**	220(**223**)	**4.59%**	276(**276**)	**3.11%**
24	127(**126**)	**15.01%**	123(**129**)	**10.36%**	119(**121**)	**8.95%**	124(**122**)	**13.87%**	121(**122**)	**9.19%**	118(**118**)	**7.30%**
48	48(**50**)	**25.04%**	36(**42**)	**19.52%**	45(**47**)	**18.33%**	46(**46**)	**21.88%**	34(**36**)	**15.59%**	45(**45**)	**14.88%**
96	24(**23**)	**33.48%**	25(**23**)	**32.96%**	20(**22**)	**28.68%**	22(**22**)	**33.01%**	17(**17**)	**25.96%**	20(**21**)	**26.26%**
192	22(**25**)	**65.22%**	24(**23**)	**55.01%**	22(**26**)	**63.51%**	21(**21**)	**65.76%**	18(**17**)	**59.35%**	24(**21**)	**60.84%**
1	21218	-	19259	-	19204	-	19120	-	18565	-	13321	-
12	5466(**5419**)	**7.26%**	3991(**4005**)	**2.56%**	5083(**5128**)	**1.53%**	5438(**5477**)	**7.27%**	3992(**4020**)	**2.43%**	5097(**5198**)	**2.25%**
24	2069(**2095**)	**10.58%**	2602(**2608**)	**4.10%**	1958(**1958**)	**4.36%**	2081(**2089**)	**9.94%**	2613(**2616**)	**4.36%**	1944(**1962**)	**4.04%**
48	1360(**1345**)	**10.53%**	1328(**1326**)	**6.88%**	1301(**1297**)	**5.88%**	1356(**1352**)	**11.14%**	1308(**1311**)	**6.05%**	1288(**1297**)	**5.79%**
96	615(**619**)	**13.98%**	479(**482**)	**10.24%**	592(**593**)	**8.36%**	610(**606**)	**12.74%**	470(**469**)	**9.19%**	594(**593**)	**8.37%**
192	281(**288**)	**23.11%**	213(**218**)	**20.68%**	269(**278**)	**18.65%**	270(**269**)	**19.38%**	206(**210**)	**18.71%**	163(**265**)	**15.51%**
384	140(**124**)	**40.29%**	133(**123**)	**37.17%**	119(**131**)	**43.41%**	116(**114**)	**38.61%**	122(**107**)	**31.20%**	117(**117**)	**36.51%**
768	-	-	119(**92**)	**68.67%**	107(**103**)	**69.95%**	-	-	92(**109**)	**71.09%**	-	-

Table 1: Timing of the 100×100×100 cells (upper part of the table) and of the 200×200×200 cells (lower part of the table) benchmark cases. The wall-clock time is reported in seconds. Results in bold text are gathered from versions of the code instrumented with IPM library.

PLX			CURIE					
Task #	Time	Efficiency	Task #	Time	Efficiency	Task #	Time	Efficiency
1	19497	-	1	26855	-	17	4417	0.356
2	12202	0.799	2	19188	0.700	18	4864	0.307
3	10305	0.631	3	13191	0.679	19	4580	0.309
4	7638	0.638	4	10937	0.614	20	4298	0.312
5	6916	0.564	5	9265	0.580	21	2876	0.445
6	6820	0.476	6	8921	0.502	22	2585	0.472
7	6994	0.398	7	8144	0.471	23	2613	0.447
8	4083	0.597	8	6071	0.553	24	2500	0.448
9	6780	0.320	9	9330	0.320	25	2451	0.438
10	5562	0.351	10	8632	0.311	26	2334	0.443
11	5332	0.332	11	7268	0.336	27	2321	0.429
12	3993	0.407	12	5814	0.385	28	2130	0.450
			13	6287	0.329	29	1580	0.586
			14	6019	0.319	30	1969	0.455
			15	5440	0.330	31	1943	0.446
			16	5522	0.304	32	1882	0.446

Table 2 : Intra-node scaling results for the 200 × 200 × 200 cells test case. The wall-clock time is reported in seconds.

Node #	Task #	Time	% MPI	Task #	Time	% MPI	Task #	Time	% MPI
1	4	7074	1.73%	8	5129	2.24%	12	4020	2.43%
2	8	3066	9.98%	16	3255	8.04%	24	2616	4.36%
4	16	2258	15.82%	32	1528	10.93%	48	1311	6.05%
8	32	1006	15.08%	64	517	13.04%	96	469	9.19%
16	64	313	14.57%	128	296	16.41%	192	210	18.71%
32	128	179	14.96%	256	120	23.66%	384	107	31.20%

Table 3 : Scaling results for the 200×200×200 cells case, spawning a different number of tasks per node. The wall-clock time is reported in seconds.

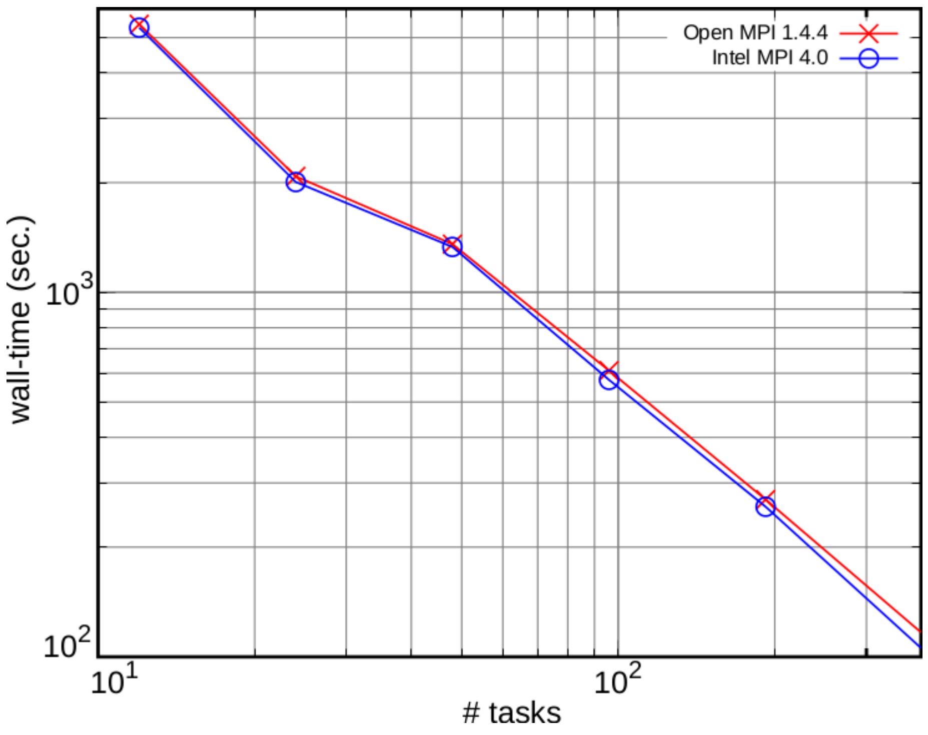

Figure 1: Comparison of two different MPI implementations on the 200×200×200 cells case

In the following we will consider a case with a droplet size of 0.6mm, an impact velocity of 50 m/sec and a static finite volume mesh. Notice that due to the complexity of the physics being modelled this benchmark resembles more a real industrial case than the lid-driven cavity flow treated in Section 3.1.

A plot of the strong scaling results on PLX is shown in Figure 2 where meshes of respectively 320000 cells, 1296000 cells and 4800000 cells have been used. The scaling trends are similar to the ones obtained in Section 3.1. This result suggests that the current bottleneck may reside in core algorithms that are relevant for both the lid-driven cavity and the droplet splash test case. In the next section we will therefore discuss which part of the solver is preventing a greater scalability and why this effect is arising.

4. Analysis and discussion

The results presented in Section 3 clearly show that the size of the problems that can be handled on a HPC cluster lies beyond the limitations imposed by smaller in-house clusters. Still, the scalability of OpenFOAM is such to permit a proper usage of Tier-1 rather than Tier-0 architectures. Furthermore the computational performance on the single node is limited by the bandwidth to memory available on the node itself. As already pointed out in literature [13], the scalability and performance issues are both related to the sparse linear algebra core libraries.d To motivate the last statement the Preconditioned Conjugate Gradient (PCG) method, shown schematically in Algorithm 1, will be briefly analyzed in the following as a representative of the class of Krylov subspace iterative solvers. The most relevant operations performed during each PCG iterative cycle are scalar products, preconditioning steps and matrix-vector multiplications. It is worthwhile to stress that these operations are common to all the methods based on Krylov subspaces, and therefore their optimization will have a positive impact on the whole set of linear solvers. In the following paragraphs we will thus give some brief comments

[d] Here and in the following it is assumed that the reader is familiar with the ”Zero-Halo Layer Decomposition” employed by OpenFOAM to divide the computation among a pool of tasks. If this is not the case [13] should provide the required knowledge.

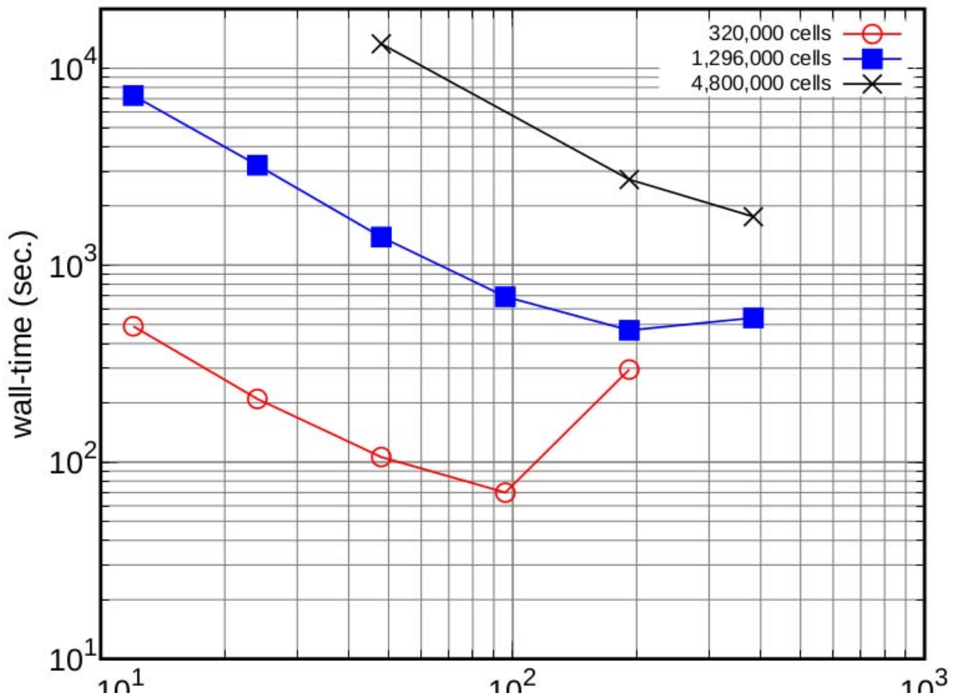

Figure 2 : : Strong scaling results for the droplet splash test case. Three different meshes of increasing size have been used. The scalability behavior resembles that of the lid drivencavity flow benchmark

on the role and impact of each of these operations.

Preconditioning Preconditioning may heavily impact on both scalability and performance, and constitutes a current topic of research. A discussion of the trade-offs of different preconditioning techniques is anyhow too technical to enter in the scope of this report. The interested reader is therefore referred to [14] and the references therein for a survey on the state-of-the-art of the subject.

Algorithm 1 Preconditioned Conjugate Gradient Algorithm

```
1: $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
2: $\mathbf{z}_0 \leftarrow \mathbf{M}^{-1}\mathbf{r}_0$
3: $\mathbf{p}_0 \leftarrow \mathbf{z}_0$
4: $k \leftarrow 0$
5: repeat
6:     $\alpha_k \leftarrow \frac{\mathbf{r}_k^T\mathbf{z}_k}{\mathbf{p}_k^T\mathbf{A}\mathbf{p}_k}$
7:     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{p}_k$
8:     $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
9:     $\mathbf{z}_{k+1} \leftarrow \mathbf{M}^{-1}\mathbf{r}_{k+1}$
10:    $\beta_k \leftarrow \frac{\mathbf{z}_{k+1}^T\mathbf{r}_{k+1}}{\mathbf{z}_k^T\mathbf{r}_k}$
11:    $\mathbf{p}_{k+1} \leftarrow \mathbf{z}_{k+1} + \beta_k\mathbf{p}_k$
12:    $k \leftarrow k + 1$
13: until $(\|\mathbf{r}_{k+1}\| < tol)$
```

Scalar productThe implementation of the scalar product operation in the Zero-Halo Layer approach is shown in Listing 1: the local contributions to the product are computed first and then the partial sums are reduced among tasks. Notice that the presence of the MPI_Allreduce routine means that scalar products act as implicit communication barriers in step 6, 10 and 13 of Algorithm 1. This constraint, imposed by the domain decomposition strategy in order to minimize the amount of replicated data, cannot be easily avoided and renders the scalar product the operation that most likely limits the scalability of the code.

Listing (1): *Zero-Halo Layer Decomposition: scalar product*

```
scalar SumProd     = 0;
scalar partialSum = 0;
// Local part of the product
for(label ii = 0; ii < max; ii++)
  partialSum += f1p[ii]*f2p[ii];
// Gather other tasks contribution
MPI_Allreduce(&SumProd,&partialSum,1,MPI_SCALAR,MPI_SUM,MPI_COMM_WORLD);
```

Matrix-vector multiplication Matrix-vector multiplication is performed in the Zero-Halo Layer approach in three sequential steps: an interface initialization, a task local matrix-vector multiplication and an interface update. The first and last operations manage communications among different subdomains, while the task local multiplication performs the actual mathematical operation on the diagonal matrix blocks associated with each subdomain. For medium size problems this core step of the algorithm may result the point where most of the CPU-time is spent and is one of the causes of the memory bound behavior noticed in the scaling tests on a single node. To provide an adequate base of understanding for this statement the storage format used for the sparse matrices will be introduced next. Sparse matrices in OpenFOAM are stored adopting the LDU storage format, otherwise known as storage by faces. In the LDU format a generic matrix A is stored as an aggregate of three different components:

$$A = L + D + U$$

where D is a diagonal matrix while L and U are respectively strictly lower and upper triangular. The

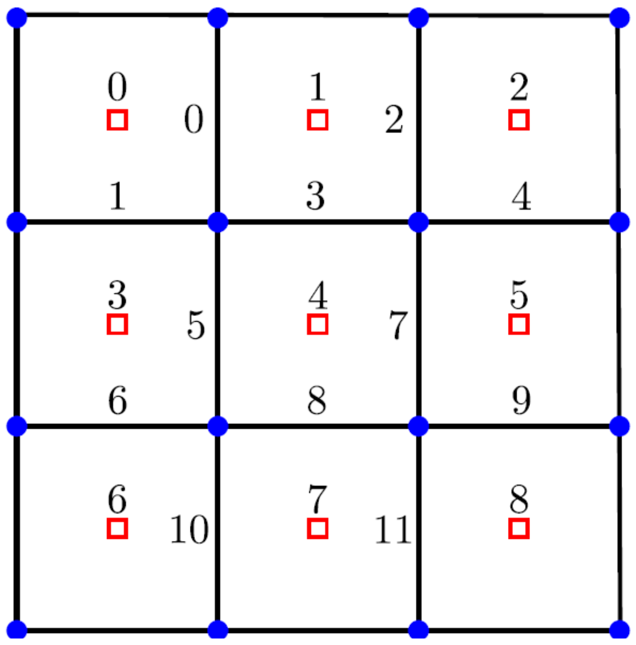

Figure 3 : Mesh composed of 9 cells and 12 internal faces. The number of each face is internal to the owner cell. To better adhere to the actual C++ implementation cells and faces are numbered employing zero-based ordering.

diagonal matrix D is represented by a scalar[e] vector of ncells elements. The upper and lower triangular parts L and U are instead stored by means of 2 label vectors and 2 scalar vectors of nfaces elements. The first of these two set of vectors will be used to map off-diagonal entries to their respective position in the matrix, while the latter will contain the entries themselves. Notice that the decision to have vectors of length nfaces for the off-diagonal contributions implies the assumption for A to be structurally symmetric. Further, it limits the approximation of the fluxes across cells only to methods based on first neighbor contributions. To provide a concrete example of the addressing we will make use of the mesh appearing in Figure 3 with ncells = 9 and nfaces = 12. The two label vectors used for the addressing will read then:

$$\text{lPtr} = 0\ 0\ 1\ 1\ 2\ 3\ 3\ 4\ 4\ 5\ 6\ 7$$
$$\text{uPtr} = 1\ 3\ 2\ 4\ 5\ 4\ 6\ 5\ 7\ 8\ 7\ 8$$

and will serve the purpose of mapping each face to the owner (lPtr) and neighbor (uPtr) cell.[f] Notice that lPtr is always globally ordered, while uPtr is always locally ordered (i.e. for each set of faces that have the same owner, the neighbor is stored in increasing order).

The LDU format is obviously quite convenient for the assembly of sparse matrices stemming from a finite volume discretization on a generic unstructured grid, as it will require a single cycle on faces to stamp flux related quantities. Nonetheless it will not always provide the best performances in the execution of matrix-vector multiplications. To see where issues may arise it is worthwhile to discuss the code snippet in Listing 2, which shows the implementation of the core matrix-vector multiplication operation. The computation of the output vector in two steps clearly reflects the structure of the LDU format. Anyhow there is a clear difference in the memory access pattern that renders the computation of the off-diagonal contributions significantly slower than that of the diagonal. In fact while in the first cycle data is accessed sequentially, in the second the data pointed by ApsiPtr and psiPtr is accessed indirectly through the mapping vectors lPtr and uPtr. This indirect access very likely compromises the prediction capabilities of the prefetcher and enhances the probability of having cache misses [15]. Furthermore it does not permit an efficient use of vectorization where SIMD registers are present, as the time necessary to move non-sequential data to the registers will largely overcome the gain of performing the operation simultaneously on multiple data.

Listing (2): *LDU core matrix-vector multiplication: serial implementation*

```
// Diagonal contributions
const label nCells = diag().size();
for (label cell = 0; cell < nCells; ++cell) {
  ApsiPtr[cell] = diagPtr[cell]*psiPtr[cell];
}

// Off-diagonal contributions
const label nFaces = upper().size();
for (label face = 0; face < nFaces; ++face) {
  ApsiPtr[uPtr[face]] += lowerPtr[face]*psiPtr[lPtr[face]];
  ApsiPtr[lPtr[face]] += upperPtr[face]*psiPtr[uPtr[face]];
}
```

Possible improvements to the code The analysis conducted in the previous paragraphs suggests two orthogonal ways to improve the performance of OpenFOAM solvers. The first is the implementation of cache blocking techniques to reduce the number of cache misses in the core operations due to the random access patterns [15]. This may require strong efforts as the basic matrix class must be revised to allow for the storage of small, contiguous blocks of scalar type as "unit" entries of the format. The second approach is the modification of the basic linear algebra routines in a way that makes them multi-threaded. This will indeed mitigate the increase in the time spent inside MPI routines as well designed multi-threaded tasks can ideally exploit the resources provided by the largest shared memory portion of the machine.

[e] In OpenFOAM the type scalar is usually a typedef to float or double, while label is a typedef of integer

[f] In OpenFOAM the owner cell of a face is defined to be the cell with the lower index next to the face. The cell with higher index will be denoted instead as neighbor.

5. Multi-threaded hybridization of the linear algebra libraries

To give an idea of the modifications that might be necessary when moving from a single threaded process to a multi-threaded one we will discuss the multi-threaded matrix-vector multiplication core algorithm. Then we will show preliminary benchmarks obtained with a prototype multi-threaded implementation and compare them against the current baselines for an equal amount of physical resources.

Multi-threaded matrix-vector multiplication The implementation shown in Listing 2 makes it evident that while the diagonal contribution to the product can be easily parallelized with a work-share directive, the same does not hold true for the off-diagonal contributions. The problem that arises is a possible write concurrency in the body of the second cycle: to avoid the indirect access to the lvalues the cycle over faces is therefore to be mapped on a cycle over cells. One way to obtain this is to exploit the property of lPtr to be globally ordered. It is possible in fact to construct a label vector owPtr of size (ncells + 1) that maps back cells to faces:

owPtr = 0 2 4 5 7 9 10 11 12 12

Using owPtr it is quite simple to rewrite the upper triangular contribution to the product:

```
for (label face = 0; face < nFaces; ++face) {
  ApsiPtr[lPtr[face]] += upperPtr[face]*psiPtr[uPtr[face]];
}
```

as:

```
#pragma omp for
for (label cell=0; cell < nCells; ++cell) {
  for (label fidx = owPtr[cell]; fidx < owPtr[cell+1]; ++fidx)
    AxPtr[cell] += upperPtr[fidx]*xPtr[uPtr[fidx]];
}
```

To adopt a similar strategy for the lower triangular contributions we need a label vector mapping the current case in which lPtr is globally ordered to the case where uPtr has this property:

reshape = 0 2 1 3 5 4 7 6 8 10 9 11

plus the corresponding cell to face mapping:

loPtr = 0 0 1 2 3 5 7 8 10 12

Notice that these vectors may be retrieved using methods provided by the lduAddressing class.

Lid-driven cavity flow benchmark Preliminary results on PLX are shown in Figure 4, where the lid-driven cavity flow test case has been taken into consideration. In the multi-threaded case all the BLAS-like operations used in the PCG algorithm have been parallelized with OpenMP. Differently from the tests performed in Section 3.1, a diagonal preconditioner has been used for the linear systems stemming from the pressure equation. This is definitely not the most efficient choice for real problems, but provides the most favorable case for the hybrid implementation to compare against the standard one and extract relative performance upper-bounds. Finally, as no special allocators have been written to deal with NUMA architectures, each **task** has been bound to a single socket when the multi-threaded version has been tested.

Figure 4 shows a single node scalability test on a 200×200×200 cells mesh. As we can see both the MPI and the MPI+OpenMP implementations deviate considerably from the ideal scaling performance when using more than 4 cores: this reveals again the memory bound nature of the code. The best performance of the hybrid code results to be 10% slower than the best performance of the pure MPI code. Anyhow while the latter spawns 12 serial tasks on the node, the former spawns only 2 multi-threaded tasks (one for each node socket). This reduces the time spent in inter-task communication and improve the maximum memory available per task, the last issue being of utmost importance for many Tier-0 architectures (see for instance [16] where a single compute card may support 64 physical threads but provides only 16GB of RAM).

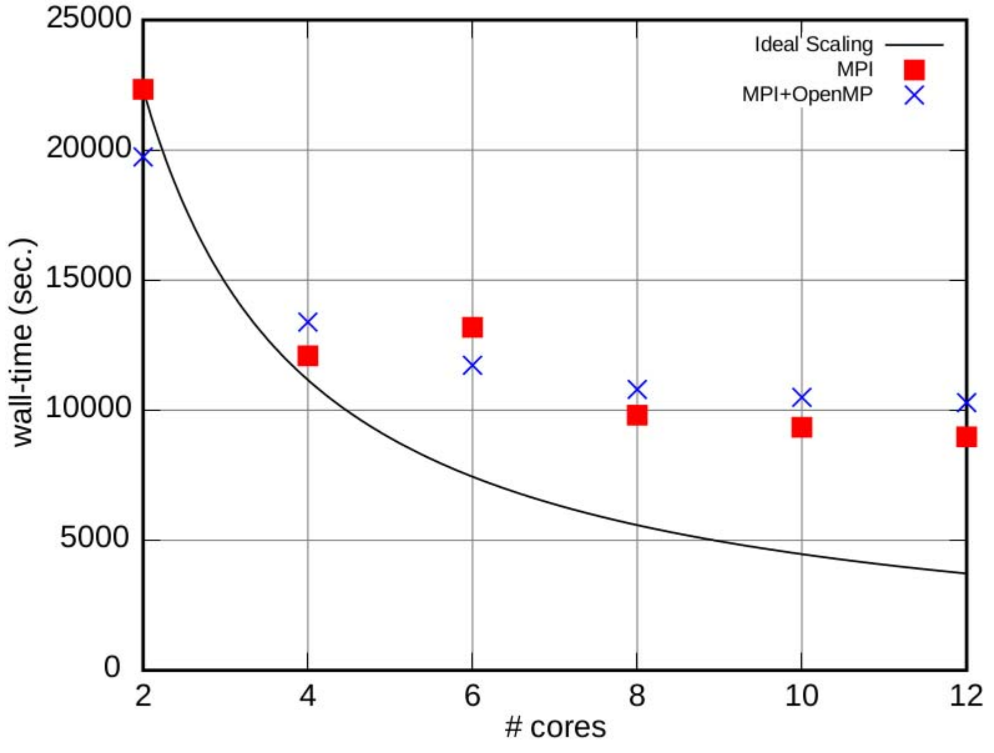

Figure 4 : Scaling results obtained on a single node for the 200 × 200 × 200 cells case. A diagonal preconditioner has been used instead of an Incomplete Cholesky Factorization for the PCG algorithm.

Finally Figure 5 shows the scalability test for a 100×100×100 cells mesh on multiple nodes. As expected the best performance (29 sec.) obtained by the hybrid code is 20% faster than the best pure-MPI run (36 sec.) and occurs at double the number of cores. Notice that though these timings stems from an ad-hoc test case, they suggest that a more careful hybridization of the linear algebra core libraries may improve both the scalability and the usability of OpenFOAM on many HPC architectures. What could be expected in the best case for a production run is a substantial reduction of the intra node MPI communication and, as a consequence, a better scalability of the application.

6. Conclusion

The scalability of OpenFOAM has been extensively tested on two benchmark cases, probing the impact of different set-ups and varying the input data-sets. Results show that the code can scale without any modification to the sources up to a thousand tasks on Intel NUMA architectures. For each test, the wall-clock time needed to execute a particular run proved to be independent of the MPI implementation or of the particular compiler used to build the application binaries.

The bottlenecks preventing further scalability have been tracked down to the linear algebra core libraries and their origin has been discussed. In particular the scalar product proved to be the most communication intensive function for large number of tasks, due to the necessity of an MPI Allreduce call for each scalar product call. Two viable strategies to improve the performance of the code have been proposed, namely a cache-aware storage format for sparse matrices and a multi-threaded hybridization of the linear algebra core libraries. The first of these modifications should increase the on-core performance, at the expense of a greater fill-in in the sparse matrices. The second should instead reduce the communication time by adding a

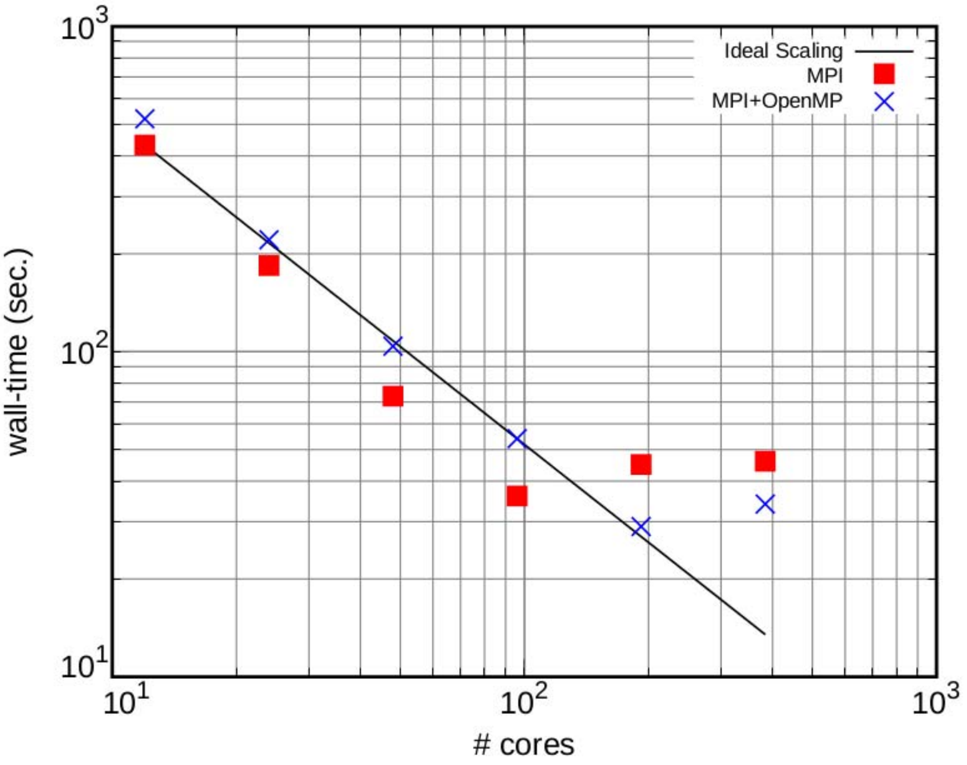

Figure 5 : Scaling results obtained on multiple nodes for the 100 × 100 × 100 cells case. A diagonal preconditioner has been used instead of an Incomplete Cholesky Factorization for the PCG algorithm.

multi-threaded layer to each MPI process. Preliminary results of a prototype multi-threaded implementation showed promising performance on an ad-hoc test case. More extensive studies on this approach and on the effect of cache-aware storage formats will be conducted in the framework of PRACE-2IP WP9. As this work will require careful and extensive modifications to the OpenFOAM linear algebra libraries, during PRACE-1IP a lot of effort has been devoted in establishing and maintaining contacts with the Italian OpenFOAM developer communities and with the Wiki head developer (Prof. Hrvoje Jasak). These contacts will be essential during the design phase of the modifications to be implemented in PRACE-2IP-WP9, due to the complexity of the OpenFOAM sources.

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-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources CURIE (TGCC, France) and PLX (CINECA, Italy).

References

1. L. Mangani, Development and Validation of an Object Oriented CFD Solver for Heat Transfer and Combustion Modeling in Turbomachinery Applications. PhD thesis, Universit`a degli Studi, Firenze, 2008.
2. G. J. Pringle, “Porting OpenFOAM to HECToR. A dCSE project,” tech. rep., EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, UK, 2010.
3. T. Priol, “HPC in Europe,” in 2nd workshop of the Joint Laboratory for Petascale Computing, 2009.
4. CINECA, PLX User Guide. https://hpc.cineca.it/content/ibm-plx-user-guide
5.] TGCC, TGCC CURIE Specifications. http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm.
6. K. Fürlinger, D. Skinner, S. Hunold, et al., Integrated Performance Monitoring (IPM) library.

http://tools.pub.lab.nm.ifi.lmu.de/web/ipm/.

7. P. Mucci, J. Dongarra, et al., Performance Integrated Programming Interface (PAPI) library. http://icl.cs.utk.edu/papi/.. Priol, “HPC in Europe,” in 2nd workshop of the Joint Laboratory for Petascale Computing, 2009.
8. E. Jarvinen, “OpenFOAM performance on Cray XT4/XT5,” tech. rep., CSC - IT Center for Science Ltd. , Espoo, Finland, 2009.
9. P. Calegari, K. Gardner, and B. Loewe, “Performance study of OpenFOAM v1.6 on a Bullx HPC cluster with a Panasas parallel file system,” in Open Source CFD conference, November 2009.
10. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel com- puting research: A view from Berkeley,” Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
11. HPC Advisory Council, “OpenFOAM performance with MPI collective accelerations,” in HPC Advisory Council Best Practices, 2011.
12. 12. H. Jasak, “Surface capturing in droplet breakup and wall interaction,” in Seminar at the ”Technische Universitaet Darmstadt”, Wikki, 2005.
13. H. Jasak, “HPC deployment of OpenFOAM in an industrial setting,” in PRACE Seminar: Industrial Usage of HPC, 2011.
14. M. Benzi, “Preconditioning techniques for large linear systems: A survey,” J. COMPUT. PHYS, vol. 182, pp. 418–477, 2002.
15. U. Drepper, “What every programmer should know about memory,” 2007.
16. A. Grant, “Blue Gene/Q: Next Step on the Roadmap to Exascale,” in 22nd Machine Evaluation Workshop, 2011.