Proceedings of
11th European Conference on Turbomachinery Fluid dynamics & Thermodynamics
ETC11, March 23-27, 2015, Madrid, Spain

A HYBRID PARALLELIZATION STRATEGY OF A CFD CODE FOR TURBOMACHINERY APPLICATIONS

M. Giovannini[1] *– M. Marconcini*[1] *– A. Arnone*[1]
A. Dominguez[2]

[1] Department of Industrial Engineering,
University of Florence, Via S. Marta, 3, 50139 Florence, Italy
Matteo.Giovannini@unifi.it

[2]Intel Corporation, EMEA High Performance and Throughput Computing
"Les Montalets"- 2, rue de Paris, 92196 Meudon Cedex, France

ABSTRACT

This paper presents the serial optimization as well as the parallelization of the TRAF code, a 3D multi-row, multi-block CFD solver for the RANS/URANS equations. The serial optimization was carried out by means of a critical review of the most time-consuming routines in order to exploit vectorization capability of the modern CPUs preserving the code accuracy. The code parallelization was carried out for both distributed and shared memory systems, following the actual trend of computing clusters. Performance were assessed on several architectures ranging from simple multi-core PCs to a small slow-network cluster, and high performance computing (HPC) clusters. Code performance are presented and discussed for the pure MPI, pure OpenMP, and hybrid OpenMP-MPI parallelisms considering turbomachinery applications: a steady state multi-row compressor analysis and an unsteady computation of a low pressure turbine (LPT) module. Noteworthy, the present paper can provide code developers with relevant guidelines in the selection of the parallelization strategy without asking for a specific background in the parallelization and HPC fields.

NOMENCLATURE

ϵ	parallel efficiency	nyp	number of cells in the j direction
f_p	parallel fraction of the code	nzp	number of cells in the k direction
gain %	$(t_{original} - t_{current})/t_{original}$	p	number of processors
N_{block}	number of grid blocks in the computational domain	S_p	speedup, $t_{original}/t_{current}$
nxp	number of cells in the i direction	t	computational time

INTRODUCTION

In the last decades Computational Fluid Dynamics (CFD) has become widely used for both scientific and industrial applications. In the turbomachinery industry, CFD codes are essential design tools providing a deep insight into flow fields and accurate predictions of performance. Meanwhile computational power has rapidly increased so that clusters, collecting hundreds or thousands of powerful CPUs, have become more and more available at affordable costs. High performance computing (HPC) platforms, nowadays, consist of multi-socket, multi-core shared-memory nodes coupled via high-speed interconnections. In this context, it is mandatory for a modern CFD code to exploit such available computational power while taking into account the current developing trend of computer architectures. Moreover, a modern solver has to ensure great performance for everyday serial computations as well as high scalability for large-scale computations.

Steady-state analyses are the actual standard adopted in the turbomachinery industry to solve multi-row computations during the design process. With the modern computers, such kind of computations can be performed within the nighttime even with a serial approach. However, a serial code revision can provide a further reduction of the turn-around time with a relevant impact on turbomachinery design.

Thanks to the increased computational power, renewed and increasing attentions have been devoted to unsteady analyses. The importance of the introduction of unsteady computations into the design process has been asserted in several works (He et al. (2002); Blumenthal et al. (2011); Denton (2010)). However, until now the use of unsteady simulations remains limited to small fractions of the turbomachines (e.g. one single stage) and to the very last phases of the design process. It is worthy noting that this is not due to any theoretical difficulty but it is mainly caused by their large computational requirements. Even if several simplified, low computational cost, models have been recently developed (e.g. Hall and Ekici (2005); Gopinath and Jameson (2006); He (2010)), such kind of calculations can hardly be managed even with today's best serial computers. As clearly underlined by Holmes et al. (2011), a great extension of the feasible problem size and a relevant reduction of the solution time can be obtained only by scaling up a CFD flow solver to perform well on large parallel cluster. Only when such calculations will be completed on currently available computers in short enough time industries would introduce them into the design process. These background considerations justify the relevant efforts spent for code parallelization. Moreover, it is interesting to note how the parallelization strategies adopted in the last years have followed the development trend of HPC clusters. If distributed memory parallelism was a common choice in the first 2000s (Djomehri and Jin (1991); Chen and Briley (2001); Yao et al. (2001)), an increasing number of works dealing about shared memory parallelization (Simmendinger and Kuegeler (2010); Jackson et al. (2011)) as well solutions for graphics processing units (GPU) and microprocessors have been presented in the last years (Gorobets et al. (2013); Wang et al. (2013); Brandvik and Pullan (2011)).

The serial optimization as well as the parallelization of the TRAF code, an in-house developed CFD code, is presented and comprehensively illustrated in this paper. The code parallelization was carried out in a trade-off between the cost of parallelization and the expected performance benefits. Following the natural hierarchy of parallelism, the first step was aimed at improving the serial execution by exploiting vectorization capability, that is a sort of parallelism for instructions within a single CPU. Then, a second level of parallelism was introduced for intra-node parallelization and finally, a inter-node level was implemented. The OpenMP standard was adopted for the shared memory parallelization while the Message Passing Interface (MPI) was used to handle distributed memory systems. The coexistence of these different approaches permitted to improve the code scalability and flexibility as it can be easily tuned on specific clusters. Moreover this work underlines how an hybrid parallelism (MPI+OpenMP) allows to overcome some known limits of both paradigms. The present activity was focused on an industrial point of view. Code development was mainly aimed at improving performance for daily applications on computer architectures that are usually adopted for design purposes rather than at providing very high pick performance for low-level benchmarks on super computers.

THE TRAF CODE

The TRAF code (Arnone (1994)) is a Q3D/3D multi-row, multi-block CFD solver for the RANS-URANS equations written in conservative form in a curvilinear, body-fitted coordinate system. The source code is written in standard Fortran 90 and adopts dynamic memory allocation. The steady Navier-Stokes equations are reformulated to be handled by a time marching steady-state approach using a four-stage Runge-Kutta scheme. Residual smoothing, local time-stepping and multigridding techniques are employed to speed up the convergence rate.

The code features several turbulence closures, ranging from simple algebraic model, to one- and two-equation models (e.g. Wilcox (1998); Menter (1994)). Two different transition-sensitive, turbulence closures are also available (Pacciani et al. (2014)). Steady multi-row problems are handled by means of a Mixing-Plane model. Time-accurate calculations are performed using a dual-time stepping method. Both the Full-Annulus (Arnone and Pacciani (1996)) and the Phase-Lagged (Giovannini et al. (2014)) approaches are available to handle BCs in multi-row unsteady calculations.

SERIAL OPTIMIZATION

Following the current trend of the hardware architectures increasing attentions are devoted towards code parallelization as well as code scalability improvements. In this context, any effort spent in the serial optimization of a code may sound quite dated as it appears easier to exploit massive parallelism instead of improve code efficiency on a single core. Nevertheless, different conclusions can be drawn when considering real applications of CFD codes. Serial efficiency is clearly profitable for daily CFD computations reducing solution time for a given analysis or allowing more complex calculations in the same turnaround time. Noteworthy, an improved serial code can be relevantly capitalized in all that activities in which the computational challenges are caused more by the total amount of computations than the cost of a single computational, i.e. design or optimization campaigns. Finally, the serial performance will affect so relevantly the parallel code performance that serial optimization is the first necessary step towards a good parallelization. When seeking strong scalability, by using more and more processors, performance are mainly affected by the non-parallel part of the code. Therefore, the serial optimization can relevantly improve strong scalability by reducing the computational costs of the serial part of the code (Hager and Wellein (2010)). Approaching the code revision some pressing constraints were considered. No code reformulation could be accepted unless it had preserved the original numerical accuracy and the source code readability. This constraints, for example, prevent the use of more aggressive compiler optimization option that enable faster but less precise implementations (e.g. "-fast" option for Intel compiler). Several attentions, for example, were needed when rearranging arithmetic expressions or loops and when forcing compiler optimization due to the non-associativity of floating point operations. After each change, the code was tested on a wide range of test cases representative of real turbomachinery applications and the results were compared with the original ones. The optimized code was asked to provide a similar convergence history and to assess practically the same performance (efficiency, mass flow rate, blade loads...).

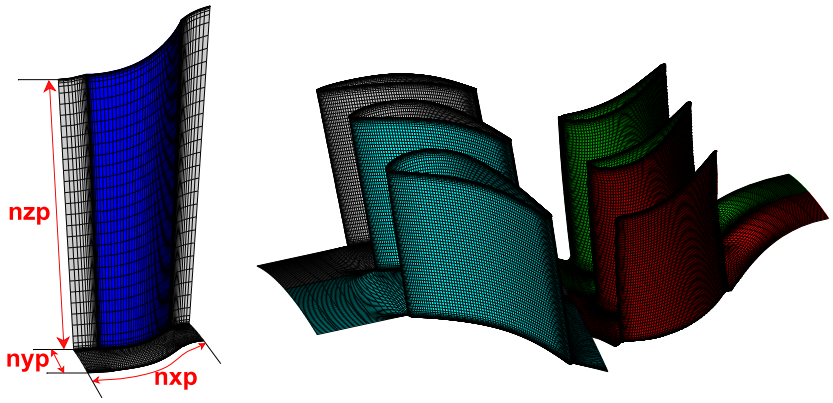
The first step of the serial optimization consisted in the code profiling. The runtime behaviour of the code was analyzed looking for the most time-consuming parts. As a first result, the code profiling proved that the original code was not memory bound thanks to its efficient access to memory. The memory storage of the main matrices as well as the loop ordering within the routines follows the column major order of Fortran program and are thought to improve both spatial data and temporal data locality ensuring an efficient access to memory. Therefore, the TRAF code features and high computational intensity, i.e. runtime performance are mainly limited by floating point operations. In such a situation the CPU's peak performance and the optimization capabilities of compilers can be profitably exploited. To improve performance, special attentions were paid to the use of transcendental functions that are high time consuming and have very long latency (e.g. several tens of cycles for square root, about 100 or more for trigonometric functions...). They are pipelined to a small level or not at all preventing compiler optimizations. Avoiding, or almost limiting, the use of such functions is thus a primary target for code optimization.

Even more relevant results were obtained by exploiting the "vectorization" capability (single instruction multiple data, SIMD) of modern CPUs. This consists in the parallel execution of instructions within a single processor that can process multiple data (vector) at a time instead of processing each scalar sequentially. Even if this task and other similar optimizations are generally demanded to the compiler, the programmer can help the compiler itself to perform loop fusion, loop unrolling, vectorization etc... During this phase, the log reports generated by the compiler were used to figure out where optimizations had been automatically performed and where performance could be further improved. The presence of transcendental operations, conditional branches as well as the presence of too few instructions within a loop, for example, retains the compiler from doing advanced optimizations. In the first case, the original loops were split in two different ones. Transcendental operations are performed in first small no-vectorized loop and results are stored in scratch arrays, while all the other floating point operations are carried out within a second vectorized loop.

model	speedup
SD	1.78
MD	2.27
Algebraic TM	1.78
1 eq. TM	1.72
2 eq. TM	1.78

Intel(R) Core(TM) i7 950 3.07 GHz

Table 1: **Serial optimization results: SD=scalar dissipation; MD=matrix dissipation; TM=turbulence model.**



(a) (b)

Figure 1: **Multi-block 3D mesh**

Analogous issues are due to the presence of conditional branches within a loop. To enable vectorization some routines were rewritten vanishing if-statements or moving them outside of the loops. No vectorization is allowed in case of loop dependencies that is when a loop iteration depends on the results of some other ones. Due to the presence of pointers or alias, compiler softwares disable vectorization each time that a "possible" loop dependency is detected. Each non-vectorized loop has to be careful analyzed. If dependencies really exist, the loop can not be vectorized unless a new formulation is provided. On the contrary, when there are not real loop dependencies specific directives can force the compiler to vectorize the loop.

As already mentioned, the optimized code was then tested and compared to the original one when applied to solve steady and unsteady flow field. The revisited code provided the same numerical results with almost the same memory requirements but with a relevant time saving. The main outcomes of the serial optimization are reported in tab. 1 in terms of speedup. Results are shown for different numerical schemes and turbulence models. Analogous performance were measured using different Intel CPUs (e.g. Core2 Q6600 2.40 GHz 8M Cache SSE3, Xeon 5650 2.67 GHz 12M Cache SSE4.2, i5 2500K 3.3 GHz 6M Cache SSE4.2, Xeon E5-2680V2 2.8 GHz 25M Cache SSE4.2,...) As reported, a relevant speedup of at least 1.7 was achieved.

CODE PARALLELIZATION

Considering RANS/URANS computations, two different reasons bring about code parallelization. On one side a single core may be to slow to fulfill the current requirements for increasing computational complexity and reduced turnaround time. On the other side, the memory requirements of large-scale unsteady computations may outgrow the amount of memory available on a single node.

The present activity was focused on both these targets looking for code flexibility and scalability. Dealing with steady-state analysis, small- or mid-scale domains are solved by using small multiprocessors systems or workstations. In the daily use of the solver, a wide range of domain sizes and load balancing number (Simmendinger and Kuegeler (2010)), CPU architectures, network connections etc. may be encountered and a parallel code has to exploit the available computational power even without an ideal parallel setup. In this context, code flexibility is of major concern. Code scalability, instead, becomes a crucial parameter for unsteady Full-Annulus computations as hundreds or thousands of CPUs are needed to obtain results in a profitable amount of time.

After meticulously examining the pros and cons of different strategies a rigorous parallelization hierarchy was followed. Different levels of parallelism were introduced one after the other, starting from the intra CPU parallelism of vectorization to multi-threading within a single node and ending with inter-node parallelism. Each level, therefore, included the previous ones and extended code capabilities

finally resulting in a multi-level hybrid parallel code. The Open Multi-Processing (OpenMP) standard was selected for the shared memory system. The Message Passing Interface (MPI) standard instead was adopted to handle communications for distributed memory systems. Each parallel paradigm can be singularly selected at compiling time resulting in a pure OpenMP or pure MPI parallel code or combined in a flexible hybrid solution.

Shared-memory parallelization

OpenMP is a set of compiler directives that a non-OpenMP-capable compiler would just read as comments within the source code. Hence, the source code of the parallel OpenMP version can be used as a valid serial program simply disabling OpenMP when compiling. The OpenMP parallelism was introduced following a "fork-join" model and exploiting loop parallelization. Since all the threads can share the whole data no relevant changes were required for the original program.

The code parallelization was carried out preserving the numerical structure of the serial version and was based on its multi-block feature. As shown in fig. 1, each grid block cover a single blade passage, while several blocks may be arranged both in the tangential and axial directions to solve multi-block (e.g. splitter blades, circumferential periodicity sector, ...) and multi-row problems. The code therefore features the loop-based structure reported in fig. 2 in which N_{block} is the number of solved blocks and nxp, nyp, nzp are the numbers of cells in the i, j, k directions of the current block.

Each loop level (blocks or cells) was considered for parallelization. The final solution rose up in a trade-off between scalability and flexibility. On one side indeed greater is the amount of the loop iterations greater is the code flexibility as an arbitrary and high number of processors can be efficiently adopted. Concerning scalability, instead, parallel performance increase as the computational work carried out by a single thread increases. An overhead occurs each time that a parallel region is initialized but its relative importance decreases when parallel work increases. To reach very high parallel performance, the loop over different grid blocks was adopted as a coarse-grain level of parallelism. This domain decomposition is highlighted by using different colors for each block in fig. 1b. The number of parallel threads, N_{OMP}, is set at runtime. This choice will be suitable when the number of blocks is large compared to the available cores, and all the blocks have similar sizes (e.g. Full-Annulus unsteady computations). On the contrary the number of processors can not exceed the number of blocks and load unbalancing problems can arise whenever the number of blocks is not an exact multiple of the number of processors and in case of blocks of very different size. To overcome these limitations, a second (nested) parallelism level was introduced in which the loop on grid cells (k-direction) is parallelized among N_{nested} threads, see fig. 3. This nested level can be used singularly or in conjunction with the previous one. In the latter case computation is firstly parallelized among N_{OMP} processes. Then a team of N_{nested} threads is activated by each of the previous N_{OMP} ones, thus resulting in a total of $N_{CPU} = N_{OMP} \cdot N_{nested}$ processes. Due the original structure of the code, the nested level of parallelism is not so efficient as the previous one. If the k-loops can be directly parallelized when integrating equations (e.g. compute fluxes) in i- and j-directions, loop dependencies prevent this possibility when integrating along k-direction without adopting more complex implementations (e.g. hyperplane tech-

```
do m=1,N_block
  do k=1,nzp
    do j=1,nyp
      do i=1,nxp
          ⋮
```

Figure 2: **Loop-based structure of the TRAF code.**

```
!$OMP PARALLEL DO ! 1st level
 do m=1,N_block
   !$OMP PARALLEL DO  ! nested level
   do k=1,nzp
     do j=1,nyp
       do i=1,nxp        ! SIMD vectorization
          ⋮
```

Figure 3: **Adopted levels of OpenMP parallelism.**

nique). This means that approximately only 2/3 of the code can be easily parallelized. More relevant code modifications were needed to overcome this issue and the cost of parallelization appeared too high if compared to the expected improvements. Nevertheless the nested parallel level allows the parallelization of single-passage computation and it can be used to speed up the solution of multi-block problems when the number of available CPUs is greater than the number of blocks.

Distributed-memory parallelization

The final step towards the TRAF code parallelization concerned distributed-memory machines. It was carried out adopting a single-program-multiple-data (SPMD) strategy in which the same instruction sequence is carried out by each processors considering different sub-domains. Splitting the computation among different nodes, MPI parallelization offers the possibility to handle very-large domains whose memory requirements exceed the memory available on a single node. As a drawback, MPI parallelization is more involved than OpenMP, as it needs some modifications in order to set up the parallel environment, to distribute data and to perform parallel I/O and data communications.

The original domain decomposition of the code which considers one block for each blade passage was chosen as coarse-grain partitioning. Each MPI core works on one or more grid blocks exchanging information with the neighbor ones. By default, computational blocks are distributed among processors following a fill-up strategy as it proved to usually reduced communication overhead. However, it is possible to manually drive the load distribution by supplying a process-to-block table at runtime. As for the first OpenMP parallelism, the maximum number of processors that can be used is restricted to the number of blocks of the computation. Moreover, load balancing is strongly affected by the ratio between the number of grid blocks and the number of available cores. These features can limit the code capability to efficiently exploit the computational resources available for a given activity (flexibility). However, in the daily use, they did not revealed particularly limiting since distributed memory parallelization was mainly aimed at handling large-scale computations, e.g. Full-Annulus unsteady calculations, in which grid blocks are usually enough to ensure good performance. Moreover, it is known that a finer partitioning could improve code flexibility but it does not necessarily result in an higher scalability. The total number of phantom cells, and the amount of communications would increase as the number of sub-domains does with a negative impact on MPI efficiency. Furthermore, an high domain decomposition could have a detrimental effect on some convergence accelerating techniques (e.g. residual smoothing, multigridding, etc.) adopted in the TRAF code. For these reasons a further domain partitioning was not implemented and code flexibility was pursued adopting a hybrid parallelization strategy.

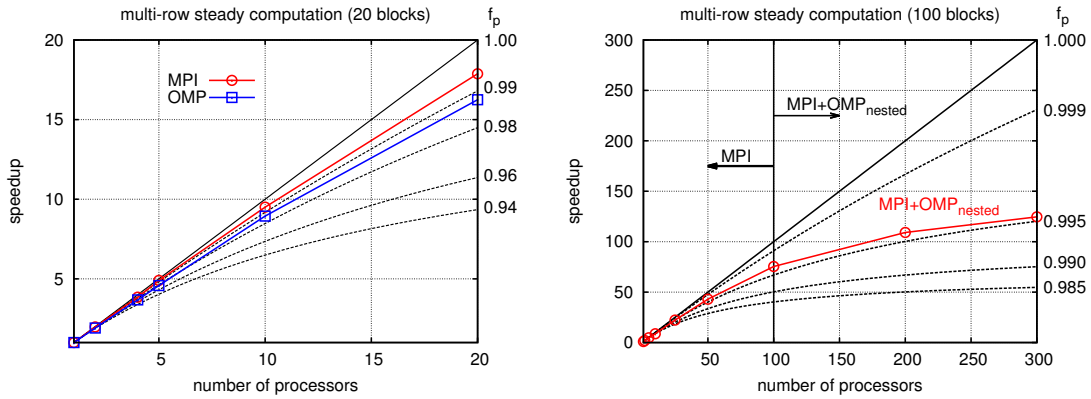
To obtain exactly the same results of the serial computation the original scheme of BCs was preserved. This results in a great number of communication occurrences that worsens scalability. It is well known, indeed, that communication overhead increases with the amount of data exchanged as well as with the number of times communications are set up. For the first issue, some routines of the serial code was reformulated in order to minimize data required for setting BCs. For example, an improved sliding interface was implemented for unsteady Full-Annulus computations. In the enhanced scheme, each processor exchange data with the neighbor blocks without the need of rebuilding and broadcasting a complete periodic sector. To reduce communication occurrences, instead, a new scheme of communication (hereafter named “frozen-BCs”) was implemented. This scheme reduces data exchange for coarse multi-grid levels and can be easily selected at runtime. More exactly, in all the coarse levels of the multigrid, communications are performed in the first Runge-Kutta step only, freezing BCs for the next steps. It is important to bear in mind that even if this scheme reduces time per iteration, this not ever results in a reduced solution time. Freezing the BCs during inner steps, does not alter the final results but can have a detrimental effect on accelerating techniques. This can lead to a slower convergence rate for the parallel computation and should be carefully evaluated in each specific case.

Hybrid parallelization

It is well known that efficient MPI codes can scale up to thousands of processes without relevant performance penalties on high speed network cluster. At the same time, the benefits of an hybrid parallelization are proved for slow network clusters even with few tens of processes (Rabenseifner et al. (2009)).

A hybrid solution was implemented following a master-only scheme. An MPI environment is initialized at the beginning of the calculation distributing computational blocks among N_{MPI} processes. Then, if $N_{MPI} < N_{block}$, OpenMP can be used to parallelize the computation of each MPI process considering multi-threading (N_{OMP} threads) over its own blocks. Finally each of these OpenMP processes can give rise to a parallel team of N_{nested} threads parallelizing k-loops. Therefore, the total number of CPUs adopted in the hybrid mode, N_{CPU}, results from the products of the CPU asked for each parallel level: $N_{CPU} = N_{MPI} \cdot N_{OMP} \cdot N_{nested}$. In this hybrid implementation, data packing and unpacking are carried out taking advantage of the multi-threading while only the master thread performs communications.

RESULTS



(a) (b)

Figure 4: **Multi-row steady-state results.**

The so-called "strong-scaling" is considered to assess code performance as the main goal of parallelization is the minimization of the solution time for a given problem. In this context, theoretical results provided by the well known Amdhal's law are used for comparison purposes. Amdhal's curves represent the theoretical parallel speedup, S_p, and the parallel efficiency, ϵ, as a function of the number of processors, p, and of the parallel fraction of the code, f_p:

$$S_p = \frac{t_s}{t_p} = \frac{1}{\frac{f_p}{p} + (1 - f_p)} \quad ; \quad \epsilon = \frac{\text{performance on } p \text{ CPUs}}{p \cdot \text{performance on one CPU}} = \frac{S_p}{p} \tag{1}$$

where t_s is the computational time of the serial execution while t_p refers to the parallel one.

Low-level benchmarks are powerful tools to get information about the basic capabilities of a parallel implementation, and were exploited during the code development. However, they can not be used to accurately predict code behaviour for "real" applications. Therefore, test cases representative of actual turbomachinery applications were used to foreseen more realistic performance. First test cases consisted of steady, 3D, multi-row viscous calculations. In this kind of application, the computational domain consists of few blocks and features a small amount of data communication. A single block per row is solved and few tens of rows can be found in turbomachines. Moreover, only the circumferential-averaged values are exchanged between adjacent blocks to set BCs at the inter-row interface.

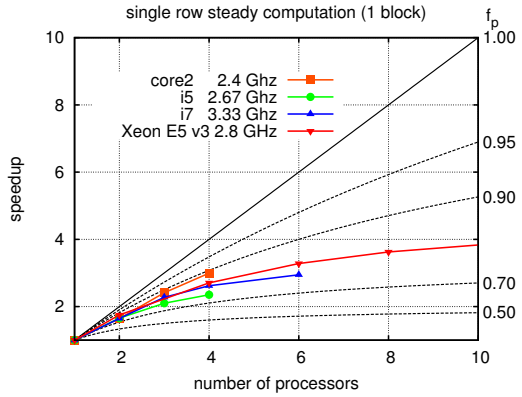


Figure 5: **Speedup for single-row steady-state test case by using OpenMP nested parallelism.**

The speedup obtained computing a 20-rows axial compressor is reported in fig. 4a where it is compared with theoretical curves with a parallel fraction, f_p, ranging from 94% to 100%. Computations were carried out on a cluster recently installed at the university of Florence (UNIFI-cluster[1]) by using up to 20 cores. An H-type grid with about 10^6 cells per blade passage was adopted for the computations. As clearly shown in fig. 4a, very good scaling was obtained. More exactly, MPI results are consistent with a parallel part of over 99%. OpenMP code scales well too, but, as expected, performance are little worse than the MPI ones due to the intrinsic overhead of the OpenMP loop-based parallelization. Nevertheless, both these paradigms appeared particularly interesting in order to reduce the solution time of steady-state calculations.

A specific test case consisting of 100 grid blocks was set up in order to definitively assess the performance for steady-state analysis. Even if it is greater than any real turbomachinery configuration it can provide a comprehensive overview of the code scaling for steady computations. OpenMP results obtained using 20 cores were substantially similar to that presented in fig. 4a and are not reported here. Pure MPI parallelism was tested using up to 100 processors. As reported in fig. 4b, a good scalability consistent with a parallel part over 99.5% was achieved. Due to the partitioning strategy adopted, however, there were no possibility to further increase the number of processors with the pure MPI paradigm and a maximum value of about ~ 75 was obtained using 100 cores. This limitation was then overwhelmed by means of hybrid parallelization. Introducing the OpenMP nested level very good performance were obtained by using up to 300 cores.

As far as steady state CFD computations are concerned, it is well known that CFD code are extensively used, not only for multi-row computations, but also for the analysis of single-row steady-state problems. In such a case neither the MPI nor the OpenMP applied to block-loop can speedup the calculation. On the contrary OpenMP nested level offers interesting possibilities. Nowadays, in an industrial environment, this kind of computation are daily carried out. Thanks to the small size of such analysis they are usually performed even in very small workstations. The performance of OpenMP nested level was therefore tested on a single node of the UNIFI-cluster as well as on current multi-core desktop computers as shown in fig. 5. As expected, parallel performance of the nested level is worse than that of the other level (MPI and OpenMP over blocks), featuring a part of parallelism of about 80%. As shown in fig. 5, no relevant differences rise up changing CPU architectures, showing that the implemented code can be fruitfully exported to different processors. Even if these values of speedup are not so interesting from an HPC point of view, different conclusions can be drawn considering daily applications on small systems, where this reduction in the solution time may be valuable.

As already mentioned, unsteady computations feature very different issues and pose strong challenges for code developers. A comprehensive comparison between the computational costs of steady

[1]UNIFI-cluster: 33 nodes with 66 CPUs (660 cores) Intel Xeon E5-V2@2.8 GHz (14.8 Tflops) with 2.1 TiB RAM.

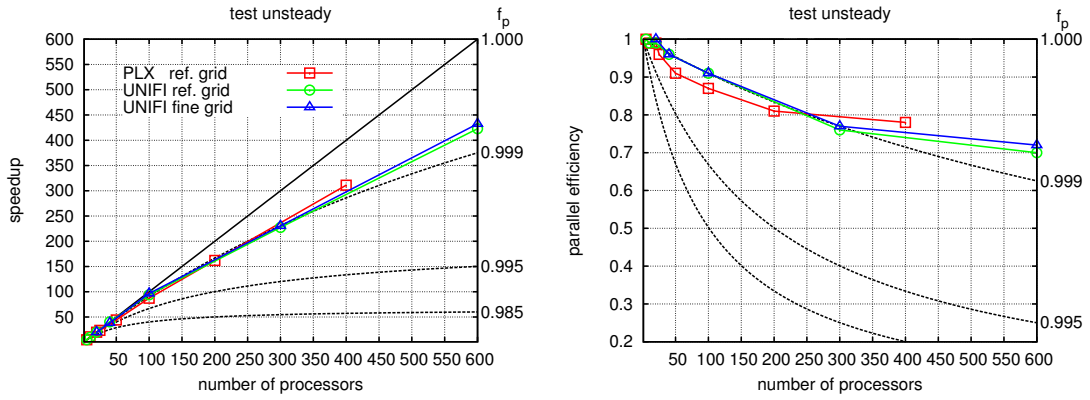


Figure 6: **Parallel performance of TRAF code on HPC clusters for large, multistage unsteady calculation (assumed full parallelism until 5 processors): a) speedup; b) parallel efficiency.**

and unsteady calculations can be found in Giovannini et al. (2014). Full-Annulus unsteady calculations are usually characterized by very large computational domains (tens or hundreds blade passages) and long computational times. Therefore, exploiting distributed memory systems is the only viable possibility to manage calculations with such high memory requirements. Moreover, relevant performance are asked for the parallel code in order to reduce the computational time. Such targets are even more ambitious considering the large amount of data that must be exchanged between different blocks.

The Full-Annulus unsteady computation of an industrial axial compressors was taken into account to assess code performance. Tests were carried out on the already mentioned UNIFI cluster and on the PLX cluster[2]. Two main test cases were set up starting from a real multistage axial compressor. The first one was analyzed on PLX cluster and consisted of 9 rows (IGV + 4 stages) resulting in a computational domain of 400 blocks. Two more stages were added in the second configuration resulting in 13 rows and 600 blocks. This configuration was tested on the UNIFI cluster considering two different grid sizes: a reference mesh consisting of about 10^6 cells per blade passage and a fine mesh with a double number of grid points.

Speedup and parallel efficiency are shown in fig. 6 as a function of the number of processors. Parallel efficiency is used to assess how effectively given resources (cores) are used in the parallel program. Due to the high memory requirements it was not possible to run any calculation using a single processor and a different definition was used for calculating speedup values. More exactly, as at least 5 nodes were necessary to meet memory requirements, a full parallelism ($S_p = 1$) was assumed until 5 processors. A parallel fraction of over 99.9% is found with a very good scalability up to the maximum number of tested processors. Moreover, such results are not relevantly affected by both the mesh size and cluster architecture. Up to 400 cores, the pure MPI approach, with the original scheme for BCs, features the best performance. The hybrid solution as well as the "frozen-BCs" scheme did not ensure any benefit. A different scenario was found with 600 cores, as shown in tab. 2 where a comprehensive comparison between different parallel setups are shown. Due to both the simple communication scheme and the master-only approach adopted, in conjunction with the large amount of data to be exchanged in a Full-Annulus unsteady computations, the pure MPI release of the code features some communication overhead. In this context, both the hybrid parallelism and the "frozen-BCs" scheme could provide better results reducing the amount of communications. Hybrid solution with $N_{MPI} = 150$ and $N_{OMP} = 4$ resulted more efficient than the pure MPI one. Even better results were obtained by introducing "frozen-BCs". For the present test case convergence rate was not

[2]PLX is an IBM iDataPlex DX360M3 Cluster equipped with 274 IBM X360M2 12-way compute nodes. Each node consists of 2 Intel Xeon Westmere six-core E5645 processors, with a clock of 2.40GHz.

N_{MPI}	N_{OMP}	N_{nested}	BCs	speedup
600	1	1	orig	369
600	1	1	frozen	412
300	2	1	orig	387
300	1	2	orig	322
150	4	1	orig	398
150	4	1	frozen	422

Table 2: **Speedup for different parallel setups with 600 CPUs.**

worsened while communication overhead was fruitfully reduced. These results definitively proved that the adoption of a hybrid solution may help to improve code scaling overcoming some limitations that rise up when parallelizing an already existent code without scheduling a complete rewrite. This is even more strategic when using a slower network computational clusters. Several tests, not reported here for the sake of brevity, were carried out on a small cluster with an Ethernet connection (1Gbps) showing that hybrid parallelism becomes useful starting from about 70 parallel processes.

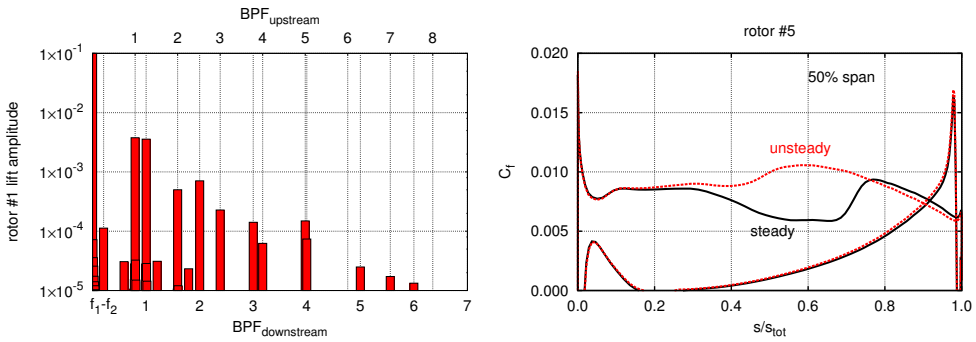
LPT module computation

The parallel release of the TRAF code was used for the detailed analyses of a 6-stage LPT module. An accurate discussion of the results, as well as the complete analysis of the LPT performance, are very beyond the scope of the present work. Geometries, operating conditions as well as the detailed results are all strictly confidential. The present paragraph is only aimed at showing how the code parallelization, together with the current computational power, drastically improves the possibility to solve problems of very high interest for the aeronautical industry.

The computation was carried out by using a Phase-Lagged model (Giovannini et al. (2014)) since the limited number of available processors (600 CPUs on the UNIFI-cluster) prevented the adoption of the Full-Annulus approach. The original blade count ratio was considered and two blade passages per row were solved, resulting in a domain of 24 blade passages. It can be noticed that over 1700 blade passages would be needed to solve the same case with a Full-Annulus approach. An H-type grid with about $8 \cdot 10^5$ cells per blade passage was considered to be adequate for the present analysis. The value of y^+ for the first grid point above the wall, was approximately 1 for all the blade rows. The low-Reynolds version of the $k - \omega$ Wilcox's model was adopted for the turbulence closure. Based on the past experiences on rotor-stator interaction analyses, 50 time steps per blade passing period were considered sufficient to assure time accuracy. A good level of periodicity was reached after 5-6 periods, then three more cycles were performed to check the stability of the results.
As an example, the spectrum analysis of the lift coefficient is reported in fig. 7a for the first rotor, while fig. 7b shows the time-averaged skin friction of the fifth rotor blade at midspan in comparison with steady calculation. As can be noticed in fig. 7a, in each row, the flow field is mainly dominated by the neighbor BPFs, their higher harmonics, and their linear combinations. As shown in fig. 7b, the time-averaged location of the transition moves upstream in the unsteady results due to the presence of the incoming wakes. Unsteady computation therefore was particularly fruitful for a detailed analysis of the flow field (see also fig. 9) as well as for evaluating LPT performance. Though not reported in this paper, the performance differences between steady and unsteady results were comparable to the performance deltas currently pursued in the turbomachinery design.

Present computation was carried out on the UNIFI cluster by using 96 cores: 4 nested threads for each of the 24 MPI processes (one for each grid block). About 9 h (wall clock time) were needed for each time period, resulting in a solution time of about 3 days. The present configuration was also adopted to test code performance on the Intel Sandy Bridge E5 Grizzly Silver cluster[3]and on a small,

[3]Sandy Bridge E5 Grizzly Silver cluster consists of 16 nodes based on Intel Xeon E5 @ 2.7GHz (Gepner et al. (2012))



(a) (b)

Figure 7: **LPT module: a) harmonic content of the 1st rotor unsteady blade load; b) comparison of the computed skin friction distributions between steady and unsteady results (5th rotor).**

Computational time [days]			
CPUs	UNIFI	slow-network	Grizzly
1 (serial)	166	265	202
12 (MPI)	14.6	27	18.7
24 (MPI)	7.9	12	10.4
96 (hybrid)	2.8	8.5	5.6

Table 3: **Computational time for the unsteady analysis of a 6-stage LPT module on different clusters.**

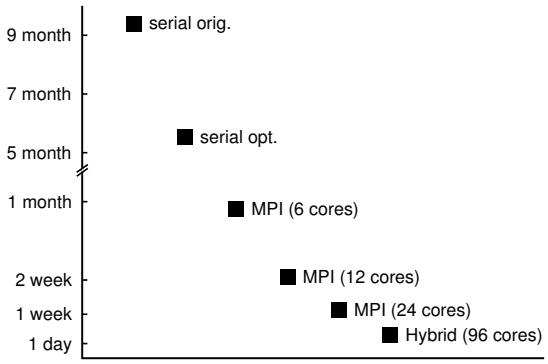


Figure 8: **Time requirements of unsteady analysis of a 6-stage LPT module.**

low-network cluster [4]. Results are reported in tab. 3. Noteworthy, as shown in fig. 8, more than 9 months would be needed to solve this problem before the code revision, 5 months would be required with the serial-optimized code, while solution time is drastically reduced even using few tens of CPUs whatever is the cluster architecture.

CONCLUSIONS

In the last years computational power has been increased opening new possibilities to exploit CFD techniques for turbomachinery design. In search for a way out of the power-performance dilemma the major processor vendors are developing multi-core platforms. This leads to new challenges for CFD code developers and the efficient use of parallel programming becomes more and more pressing.

These background considerations supported the overall motivation for the revision and parallelization of the TRAF code. Code development was carried out following a hierarchical parallelization resulting in a multi-level hybrid code. In the first level, instructions are parallelized within a single CPU by means of the SIMD vectorization. In the second step, an OpenMP block-based parallelism was implemented for sheared memory systems. A nested OpenMP parallelism was then adopted as a third level to improve load balancing. Finally, parallelization was extended to distributed memory system by the fourth MPI level. Code performance were assessed considering real turbomachinery applications. A multi-row axial compressor was considered to figure out code performance when handling steady and unsteady, 3D, viscous calculations. Results showed how relevant speedup (e.g. ~ 18

[4] 16 CPUs (96 cores) Intel Xeon X5650 2.67 GHz and a Gigabit Ethernet connection

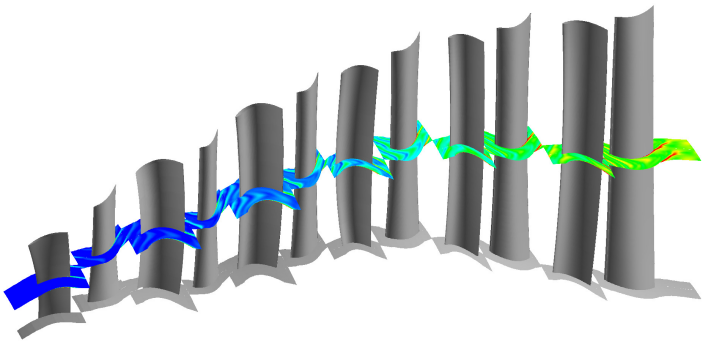


Figure 9: **LPT module: unsteady entropy contours.**

on a single node: 20 cores) can be obtained in calculations that are daily carried out during the design. Moreover, the nested level proved to be particularly suitable to improve code flexibility. Then, the code capability to scale to large numbers of processors was presented and discussed for unsteady calculations on different clusters. The parallel release of the TRAF code was finally exploited to carry out an unsteady computation of a 6-stage LPT module. More than 9 months would be required to solve such a calculation by using the original code. The serial optimization and the code parallelization allow a drastic reduction of the solution time so that less than 3 days was needed using 96 cores.

The present results have proved how the development of high performing parallel code can play a key role in exploiting current computer power and in enabling the use of unsteady calculation in the design process.

The authors well know that code scalability could be further enhanced by adopting better communication schemes or removing the master-only approach. However, this paper underlines and discusses the promising performance that can be obtained on an existing code preserving its main structure without any need for a complete rewrite of the source code.

ACKNOWLEDGMENTS

The authors would like to thanks Eng. F. Bertini, Eng. E. Spano and Eng. D. Sternini from AVIO Aero for encouraging and supporting the joint project between the university of Florence and Intel Corporation and allowing the publication of the present results. Authors would also like to thanks Dr. Bellini from Intel Corporation for providing access to the Intel Swindon HPC. The PRACE research infrastructure and Cineca research center are gratefully acknowledge for the use of the PLX cluster.

REFERENCES

Arnone, A. (1994). Viscous analysis of three–dimensional rotor flow using a multigrid method. *ASME J. Turbomach.*, 116(3):435–445.

Arnone, A. and Pacciani, R. (1996). Rotor-stator interaction analysis using the Navier-Stokes equations and a multigrid method. *ASME J. Turbomach.*, 118(4):679–689.

Blumenthal, R., Hutchinson, B., and Zori, L. (2011). Investigation of transient CFD methods applied to a transonic compressor stage. *ASME paper GT2011–46635*.

Brandvik, T. and Pullan, G. (2011). An accelerated 3D Navier–Stokes solver for flows in turbomachines. *ASME J. Turbomach.*, 133(2):021025.

Chen, J. P. and Briley, W. R. (2001). A parallel flow solver for unsteady multiple blade row turbomachinery simulations. *ASME paper 2001–GT–0348*.

Denton, J. (2010). Some limitations of turbomachinery cfd. *ASME paper GT2010-22540*.

Djomehri, M. J. and Jin, H. H. (1991). Hybrid MPI+OpenMP programming of an overset cfd solver and performance investigations. Technical report, MIT Dept. of Aero. and Astro. GTL–205.

Gepner, P., Fraser, D. L., and Gamayunov, V. (2012). Performance evaluation of intel xeon e5-2600 family

cluster using scientific and engineering benchmarks. In *Parallel Distributed and Grid Computing, 2012 2nd IEEE International Conference*, pages 230–235.

Giovannini, M., Marconcini, M., Arnone, A., and Bertini, F. (2014). Evaluation of unsteady computational fluid dynamics models applied to the analysis of a transonic high-pressure turbine stage. *Proceedings of the Institution of Mechanical Engineers, Part A: Journal of Power and Energy*, 228(7):813–824.

Gopinath, A. K. and Jameson, J. (2006). Application of the time spectral method to periodic unsteady vortex shedding. In *AIAA Paper 06–0449*. 44rd AIAA, Reno, Nevada, Jenuary 9–12 2006.

Gorobets, A., Trias, F., and Oliva, A. (2013). A parallel MPI+OpenMP+OpenCL algorithm for hybrid supercomputations of incompressible flows. *Computers & Fluids*, 88(0):764 – 772.

Hager, G. and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

Hall, K. C. and Ekici, K. (2005). Multistage coupling for unsteady flows in turbomachinery. *AIAA Journal*, 43(3):624–632.

He, L. (2010). Fourier methods for turbomachinery applications. *Prog. in Aerospace Sciences*, 46(8):329–341.

He, L., Chen, T., Wells, R. G., Li, Y. S., and Ning, W. (2002). Analysis of rotor-rotor and stator-stator interferences in multi-stage turbomachines. *ASME J. Turbomach.*, 124(4):564–571.

Holmes, D. G., Moore, B. J., and Connell, S. D. (July 10–14 2011). Unsteady vs steady turbomachinery flow analysis: exploiting large scale computations to deepen our understanding of turbomachinery flows. In *SciDAC conference*, Denver, CO, USA.

Jackson, A., Campobasso, M., and Ahmadi, M. B. (2011). On the parallelization of a harmonic balance compressible Navier–Stokes solver for wind turbine aerodynamics. *ASME paper GT2011–45306*.

Menter, F. R. (1994). Two–equations eddy viscosity turbulence models for engineering applications. *AIAA Journal*, 32(8):1598–1605.

Pacciani, R., Marconcini, M., Arnone, A., and Bertini, F. (2014). Predicting high-lift low-pressure turbine cascades flow using transition-sensitive turbulence closures. *ASME J. Turbomach.*, 136(5):051007.

Rabenseifner, R., Hager, G., and Jost, G. (2009). Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the 2009 17th EICPDNP*, PDP '09, pages 427–436, Washington, DC, USA. IEEE Computer Society.

Simmendinger, C. and Kuegeler, E. (2010). Hybrid parallelization of a turbomachinery cfd code: Performance enhancements on multicore architectures. In *V European Conference on Computational Fluid Dynamics ECCOMAS CFD*.

Wang, Y.-X., Zhang, L.-L., Liu, W., Che, Y.-G., Xu, C.-F., Wang, Z.-H., and Zhuang, Y. (2013). Efficient parallel implementation of large scale 3d structured grid CFD applications on the tianhe-1a supercomputer. *Computers & Fluids*, 80(0):244–250.

Wilcox, D. C. (1998). *Turbulence Modeling for CFD*. DCW Industries Inc., La Cañada, CA, USA, 2nd edition. ISBN 1-928729-10-X.

Yao, J., Jameson, A., Alonso, J. J., and Liu, F. (2001). Development and validation of a massively parallel flow solver for turbomachinery flows. *J. Propulsion and Power*, 17(3):659–668.