
Oyarzun, G. [et al.] (2017) Memory aware poisson solver for peta-scale simulations with one FFT diagonalizable direction. *HPCS 2017: 2017 International Conference on High Performance Computing & Simulation: proceedings: 17-21 July 2017: Genoa, Italy*. [S.l.]: IEEE, 2017. Pp. 101-108. Doi: http://dx.doi.org/10.1109/HPCS.2017.26.

Oyarzun, G. [et al.] (2017) Memory aware poisson solver for peta-scale simulations with one FFT diagonalizable direction. HPCS 2017: 2017 International Conference on High Performance Computing & Simulation: proceedings: 17-21 July 2017: Genoa, Italy. [S.l.]: IEEE, 2017. Pp. 101-108. Doi: http://dx.doi.org/10.1109/HPCS.2017.26.

Memory aware Poisson solver for peta-scale simulations with one FFT diagonalizable direction

Guillermo Oyarzun
Heat and Mass Transfer
Technological Center
Technical University
of Catalonia,
Terrassa, Spain
guillermo@cttc.upc.edu

Ricard Borrell
Heat and Mass Transfer
Technological Center
Technical University
of Catalonia,
Terrassa, Spain
ricardb@cttc.upc.edu

F.Xavier Trias
Heat and Mass Transfer
Technological Center
Technical University
of Catalonia,
Terrassa, Spain
xavi@cttc.upc.edu

Assensi Oliva
Heat and Mass Transfer
Technological Center
Technical University
of Catalonia,
Terrassa, Spain
oliva@cttc.upc.edu

***Abstract*—Problems with some sort of divergence constraint are found in many disciplines: computational fluid dynamics, linear elasticity and electrostatics are examples thereof. Such a constraint leads to a Poisson equation which usually is one of the most computationally intensive parts of scientific simulation codes. In this work, we present a memory aware Poisson solver for problems with one Fourier diagonalizable direction. This diagonalization decomposes the original 3D system into a set of independent 2D subsystems. The proposed algorithm focuses on optimizing the memory allocations and transactions by taking into account redundancies on such 2D subsystems. Moreover, we also take advantage of the uniformity of the solver through the periodic direction for its vectorization. Additionally, our novel approach automatically optimizes the choice of the preconditioner used for the solution of each frequency subsystem and dynamically balances its parallel distribution. Altogether constitutes a highly efficient and robust HPC Poisson solver that has been successfully attested up to 16384 CPU-cores.**

I. Introduction

The early generation of supercomputers were large machines that relied on parallel vector processors. Such processors exploited a Single Instruction Multiple Data (SIMD) execution model, where each operation was executed simultaneously to a vector of data contiguously stored in memory. Despite the good performance, the machines were very costly to built and maintain due to overheating and large energy consumption constraints. The HPC community embraced a technology shift and moved towards the use of clusters based on microprocessor technology. During many years, the supercomputing systems performance constantly grew, and the evolution of the computer architecture was determined by two factors: rise in clock speed and increase of transistors density. However, the physical limitations of chip design have stopped the growth in the clock rate, and have forced the pursue of new technologies that increase the performance of the systems to keep the pace of Moore's law. Paradoxically, the new CPUs or the accelerator architectures have adopted vectorization (SIMD) in order to increase the number of instructions per cycle and boost the performance. The modern CPUs have included different vector extensions (up to 256-bit) that support the SIMD model (SSE, AVX, IBM QPX,etc). The accelerators have also included directly or indirectly the use of vectorization. The Intel Xeon Phi accelerator introduced the very wide (512-bit) SIMD unit to increase the FLOP/watt ratio. While the modern GPUs exploit a stream processing model that in the low level executes instructions simultaneously in a SIMD manner of 1024-bit wide registers. This leads to the conclusion that algorithms need to be adapted to the SIMD model in order to have a reasonable potential for future applicability. In this context, we focus our attention on the development of a parallel Poisson solver flexible enough to run efficiently on different kind of parallel systems.

The present work is restricted to CFD problems in which the Poisson equation is generally the most time consuming part of the code. In particular, direct numerical simulations (DNS) and large-eddy simulations (LES) of incompressible turbulent flows. The following four aspects, which are also relevant in the context of this paper, are commonly present in many DNS/LES applications:

- The Poisson equation has to be solved repeatedly with different right-hand-side terms (for DNS/LES problems the number of time-steps can easily reach $\mathcal{O}(10^6)$), while the system matrix remains constant. Hence, a preprocessing stage with large computing demands can be accepted.
- Wall-bounded flows and/or flows around internal obstacles are common in most of the applications. Therefore, in order to solve all relevant turbulent scales near the walls, arbitrary unstructured meshes are required.
- Periodicity in at least one direction is of interest in many cases.

For flows fulfilling the last property the Fourier diagonalization [1] in the periodic direction(s) is the best choice. By doing so, the three-dimensional (3D) Poisson equation is decomposed into a family of independent two-dimensional (2D) systems of equations. On this basis, several approaches can be adopted for (i) the parallelization strategy in the periodic direction and (ii) the choice of the parallel solver(s) for the 2D problems. Roughly speaking, their choice depend on the size of the problem and the computational architecture. For instance, the strategy adopted for small problems and reduced

number of CPUs was a sequential approach in the periodic direction and a direct Schur-complement (DSD) based solver for the 2D frequency subsystems [2], [3]. Then, for bigger problems it was necessary to adopt a hybrid strategy combining DSD for some frequencies with an iterative solver [4] for the others. This was mainly due to the RAM requirements of the DSD method. Alternatively, the range of applicability of the DSD could be extended by using an efficient parallelization in the periodic direction [5]. In both cases, scalability tests up to $\mathcal{O}(10^4)$ shown a good performance. These successive improvements in the Poisson solver led to the possibility to compute bigger and bigger simulations. Starting from the simulation of a turbulent air-filled differentially heated cavity at different Rayleigh (Ra) number [6], [7], many cutting-edge DNS simulations have been computed in the last decade [8][9].

In this context, the present work proposes several improvements and adaptations of the algorithm for Peta-scale simulations on modern HPC systems. The most remarkable new features are threefold: (i) the algorithm has been evolved to a fully iterative mode in order to avoid memory constraints derived from the memory requirements of direct factorization methods. (ii) it optimizes the memory allocations and transactions by taking into account redundancies on the set of 2D frequency subsystems, (iii) it also takes advantage of the uniformity of the solver through the periodic direction for its vectorization and (iv) automatically optimizes the choice of the preconditioner used for the solution of 2D problems and dynamically balances its parallel distribution. Altogether constitutes a highly efficient and robust HPC Poisson solver. This solver has been developed and included within TermoFluids code, a multi-physics CFD code developed for numerical simulation of complex flows on High Performance Computing platforms. The performance of TermoFluids has been demonstrated on up to 130K CPU cores on the Mira supercomputer of the Argonne Leadership computing facility (ALCF) [10], [11].

II. Poisson Equation: PCG + FFT

A. Math model

The simulation of turbulent incompressible flows of Newtonian fluids is considered. Under these assumptions the velocity field, u, is governed by the Navier-Stokes (NS) and continuity equations

$$\partial_t u + (u \cdot \nabla)u - \frac{1}{Re}\Delta u + \nabla p = 0, \quad (1)$$

$$\nabla \cdot u = 0, \quad (2)$$

where Re is the dimensionless Reynolds number.

In an operator-based formulation, the finite volume spatial discretisation of these equations reads

$$\Omega \frac{d\mathbf{u}_h}{dt} + C(\mathbf{u}_h)\mathbf{u}_h + D\mathbf{u}_h + \Omega G p_h = 0_h, \quad (3)$$

$$M\mathbf{u}_h = 0_h, \quad (4)$$

where $\mathbf{u}_h$ and p_h are the velocity and pressure fields defined in the nodes of the mesh $\mathcal{M}$, Ω is a diagonal matrix with the size of the control volumes, $C(\mathbf{u}_h)$ and D are the convective and diffusive operators and, finally, M and G are the divergence and gradient operators, respectively. The pressure-velocity coupling is solved by means of a classical fractional step projection method [12], [13], leading to a Poisson equation

$$-M\Omega^{-1}M^* p_h^{n+1} = M\left(\frac{\mathbf{u}_h^n}{\delta t} + \Omega^{-1}R\left(\frac{3}{2}\mathbf{u}_h^n - \frac{1}{2}\mathbf{u}_h^{n-1}\right)\right), \quad (5)$$

where $R(\mathbf{u}_h) = -C(\mathbf{u}_h)\mathbf{u}_h - D\mathbf{u}_h$, and that must be solved once at each time-step.

B. Discrete Laplace operator

The Laplacian operator of equation (5),

$$\mathsf{L} = -M\Omega^{-1}M^*, \quad (6)$$

is by construction symmetric and negative-definite. Its action on p_h is given by

$$[\mathsf{L}p_h]_k = \sum_{j \in Nb(k)} A_{kj}\frac{p_h(j) - p_h(k)}{\delta n_{kj}}, \quad (7)$$

where $Nb(k)$ is the set of neighbors of the k'th node. A_{kj} is the area of f_{kj}, the face between the nodes k and j, and $\delta n_{kj} = |n_{kj} \cdot v_{kj}|$, where v_{kj} and n_{kj} are the vector between nodes and the normal unit vector of f_{kj}, respectively. For details about the spatial discretization the reader is referred to [14]. The set $Nb(k)$ can be split into two subsets: $Nb(k) = Nb_{per}(k) \cup Nb_{2d}(k)$, where $Nb_{per}(k)$ and $Nb_{2d}(k)$ refer to the neighbor nodes along the periodic direction and in the same plane of the extrusion, respectively. In this way, the expression (7) becomes

$$[\mathsf{L}p_h]_k = \sum_{i \in Nb_{per}(k)} A_{ki}\frac{p_h(i) - p_h(k)}{\Delta_{per}} + \Delta_{per}\sum_{j \in Nb_{2d}(k)} a_{kj}\frac{p_h(j) - p_h(k)}{\delta n_{kj}}, \quad (8)$$

where a_{kj} is the length of the edge of f_{kj} contained in $\mathcal{M}_{2d}$, and Δ_{per} is the constant spatial step in the spanwise direction. This can be written in a more compact form by means of the Kronecker product of matrices. Using the *1D-block-order*, the Laplacian operator of the equation (8) reads

$$\mathsf{L} = (\Omega_{2d} \otimes \mathsf{L}_{per}) + \Delta_{per}(\mathsf{L}_{2d} \otimes \mathsf{I}_{N_{per}}), \quad (9)$$

where $\mathsf{L}_{2d} \in \mathbb{R}^{N_{2d} \times N_{2d}}$ and $\mathsf{L}_{per} \in \mathbb{R}^{N_{per} \times N_{per}}$ are the Laplacian operators discretised on the meshes $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$, respectively; $\Omega_{2d} \in \mathbb{R}^{N_{2d} \times N_{2d}}$ is the diagonal matrix representing the areas of the control volumes of $\mathcal{M}_{2d}$, and $\mathsf{I}_{N_{per}}$ is the identity matrix of size N_{per}. With the above-mentioned conditions (uniformly meshed periodic direction), L_{per} results into a symmetric circulant matrix of the form

$$\mathsf{L}_{per} = \frac{1}{\Delta_{per}} circ(-2, 1, 0, \cdots, 0, 1). \quad (10)$$

This allows to use a Fourier diagonalisation algorithm in the periodic direction. Note that this particular solution corresponds to a second-order finite volume discretization (see [14],

for instance). However, the proposed algorithm relies on the fact that the Laplacian operator has the structure given in Eq.(9). This is the case for most of the existing numerical approximations of the Laplacian operator for problems with one periodic direction.

$$\mathsf{L}x_i = b_i \qquad i = 1, ..., N_t, \tag{11}$$

where the Laplacian operator, L, remains constant during the simulation and N_t is the total number of time-steps. Since the couplings in the periodic direction are circulant matrices, the initial system (11) can be diagonalized by means of a Fourier transform. The spectral Laplacian operator reads,

$$\widehat{\mathsf{L}}_k = \lambda_k \Omega_{2d} + \Delta_{per} \mathsf{L}_{2d} \qquad k = 0, ..., N_{per} - 1. \tag{12}$$

Note that the matrices $\widehat{\mathsf{L}}_k$ only differ in the eigenvalue, λ_k, multiplying the diagonal contribution Ω_{2d}. A general expression for the eigenvectors can be found in [15], [16], in this particular case

$$\lambda_k = -\frac{2}{\Delta_{per}} \left(1 - \cos\left(\frac{2\pi k}{N_{per}} \right) \right) \qquad k = 0, ..., N_{per} - 1. \tag{13}$$

Therefore, the original system (11) is decomposed into a set of N_{per} mutually independent 2D systems

$$\widehat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d} \qquad k = 0, ..., N_{per} - 1, \tag{14}$$

where each system, hereafter denoted as *frequency system*, corresponds to a frequency in the Fourier space. In summary, the process to solve the Poisson system is detailed in Algorithm 1.

Algorithm 1:

1) Transform the right-hand-side b, $\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}}) b$
2) Solve the *frequency systems*, $\widehat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d}$
3) Restore the solution vector: $x = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}}) \hat{x}$

C. Domain decomposition

The parallelisation of the solver is based on a geometric domain decomposition into P subdomains, one for each parallel process. The partition of $\mathcal{M}$ is carried out by dividing $\mathcal{M}_{2d}$ and $\mathcal{M}_{per}$ into P_{2d} and P_{per} parts respectively, being $P = P_{2d} P_{per}$. This is referred as a $P_{2d} \times P_{per}$-partition. Since a distributed memory parallelisation for the FFT is out of consideration, the *span-wise* component of the mesh is not partitioned. Thus, $\mathcal{M}_{2d}$ is divided into P subdomains and a $P \times 1$-partition of $\mathcal{M}$ follows. In this way, the *span-wise* subvectors of any field are not split between different processes, and a sequential FFT algorithm [17] can be used. An identical reasoning is applied to the change-of-basis from the *spectral* to the *physical* space, choosing the same $P \times 1$-partition. To obtain a balanced partition of $\mathcal{M}_{2d}$, the graph partitioning tool METIS [18] is used. Note that the $P \times 1$-partition chosen for the steps 1 and 3 can be sub-optimal if P_{2D} is too large according to the strong scalability of the linear solver being used for the frequency systems. Thus, partitions with $P_{per} > 1$ may be necessary to keep P_{2d} in the region of linear scalability of the linear solver. In this case, the N_{per} frequencies to be solved are divided into P_{per} subsets, and groups of $P_{2D} = P/P_{per}$ processes are used to solve the frequencies of each subset. The number of frequency systems contained in the k'th subset is referred as $N_{per,k}$. Therefore, since in general the optimal partitions for the change-of-basis (steps 1 and 3) and for the solution of the *frequency systems* (step 2) are different, two partitions are used in the parallelisation. As a consequence, two redistributions of data between those partitions are needed. Hence, the following algorithm replaces Algorithm 1:

Algorithm 2:

1) Evaluate $\hat{b} = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}^*_{N_{per}}) b$ on the $P \times 1$-partition.
2) Redistribute $\hat{b}$ from the $P \times 1$- to the $P_{2d} \times P_{per}$-partition (collective comm.).
3) Solve the *frequency systems*, $\widehat{\mathsf{L}}_k \hat{x}_k^{2d} = \hat{b}_k^{2d}$, on the $P_{2d} \times P_{per}$-partition.
4) Redistribute $\hat{x}$ from the $P_{2d} \times P_{per}$- to the $P \times 1$-partition (collective comm.).
5) Evaluate $x = (\mathsf{I}_{N_{2d}} \otimes \mathsf{F}_{N_{per}}) \hat{x}$ on the $P \times 1$-partition.

In order to simplify the redistributions of data (steps 2 and 4), a multilevel partition strategy is used. To do so, the P-partition of $\mathcal{M}_{2d}$, used in steps 1 and 5, is obtained from the P_{2d}-partition used in the step 3 by dividing each of its subdomains in P_{per} parts. In the general case, P_{2d} independent transmissions need to be done, and P_{per} parallel processes are involved in each of them. These collective communications are performed by means of the MPI_Alltoall routine.

D. Conjugate Gradient

The PCG algorithm is very well-known and can be found in [19]. However, there are several features of the set of problems given in Eq.(14) that may affect the convergence of the algorithm. The number of iterations needed to converge a Krylov-subspace method like PCG is closely related with its condition number κ. Well-conditioned systems (κ keeps close to unity) converge easily whereas they tend to degrade quickly when the system becomes ill-conditioned ($\kappa \gg 1$). In our particular case, the frequency systems are ordered by descending the condition number and since the matrices $\widehat{\mathsf{L}}_k$ are symmetric and positive semidefinite, the condition number κ is given by

$$\kappa(\widehat{\mathsf{L}}_k) = \frac{\max_j \left\{ \lambda_j(\widehat{\mathsf{L}}_k) \right\}}{\min_j \left\{ \lambda_j(\widehat{\mathsf{L}}_k) \right\}}, \tag{15}$$

where $\lambda_j(\widehat{\mathsf{L}}_k) \in \mathbb{R}_0^+$ denotes the eigenvalues of $\widehat{\mathsf{L}}_k$. From Eqs.(13) and (12), it follows that $\widehat{\mathsf{L}}_0 = \Delta_{per} \Delta_{2d}$. Therefore, the condition number corresponding to $k = 0$ is

$$\kappa(\widehat{\mathsf{L}}_0) = +\infty, \tag{16}$$

with a zero eigenvalue whose associated eigenvector is the unity vector

$$\widehat{\mathsf{L}}_0 1 = 0. \tag{17}$$

In practice, this singularity is easily removed by changing one element on the main diagonal that corresponds to an arbitrary inner node. This modification fixes the value in this particular node to zero. For $k > 0$, the condition number can be bounded by the following inequality

$$\kappa(\widehat{\mathsf{L}}_k) \leq 1 + \frac{2}{\sin^2(\pi(k-1)/N_{per})}, \tag{18}$$

assuming a second-order discretization on a uniformly distributed mesh with periodic boundary conditions (see [4] for details). This gives an approximate idea of how well-conditioned are the systems to be solved as a function of the relative number of plane defined as

$$\xi(k, N_{per}) \equiv \frac{2(k-1)}{N_{per}} \tag{19}$$

Convergence analysis of the CG algorithm provides an upper bound for the convergence rate (see [19], for instance)

$$\|e_k^n\|_{\widehat{\mathsf{L}}_k} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^n \|e_k^0\|_{\widehat{\mathsf{L}}_k}, \tag{20}$$

where $\|e_k^n\|_{\widehat{\mathsf{L}}_k} = \widehat{x}_i^n - (\widehat{\mathsf{L}}_k)^{-1}\widehat{b}_k$ is the solution error after n iterations and the A-norm is defined as $\|e\|_A = (e^t A e)^{1/2}$. Then, after some straightforward calculations, the convergence rate ω can be bounded above as a function of the relative number of plane ξ

$$\omega(\xi) \leq \frac{1}{1 + S\sqrt{S^2+2} + S^2}, \tag{21}$$

where $S = \sin(\xi\pi/2)$.

III. Vectorization

A. SIMD execution model

The Single Instruction Multiple Data (SIMD) is an execution model that aims at taking advantage of data parallelism. This means that we can pick the data in chunks, so called vectors, and perform operations on it within one clock cycle. The vectors are unidimensional arrays of data sequentially placed in memory. Its size depends on the architecture capabilities, and determines the number of operations that can be executed at once. For instance, an architecture that supports 256-bit vector wide registers performs 4 double precision (64-bit) vectorized operations simultaneously. In the case of operations between two vectors (add, multiplication, etc), the instructions are executed in a pair-wise manner.

B. Main considerations

- **Data Layout:** Data needs to flow to and from the vector instructions without excessive overhead. Efficiency on data movement depends on data layout, prefetching and efficient store operations. When data is loaded utilizing vector instructions the performance is optimal because reduces number of instructions (scalar instructions) and data access (everything is loaded at once). If the data will be used more than once, then it is important to reorganize the layout in order to maximize the data reuse.
- **Data Alignment:** The valid chunks of data that can be loaded in the vector registers are linked with the words in memory. The rule is that the beginning of a vector must correspond with the beginning of a word. This issue is known as data alignment, and it is key to obtain maximum performance in the vector load/store operations. In the case of a misaligned chunk of memory (when stored in two contiguous words), fetching in a vector register requires to perform more operations to organize, permute, or reorder the data and fulfill the alignment condition. Misaligned data requests introduce additional load instructions and shift or permute operations that degrade the performance of the vectorized code.
- **Loop Unrolling:** Normally, a single operation of a loop operates on one element of an array at a time. But with vectorization SIMD instructions can be utilized to operate on multiple components simultaneously. It is important that the loop iterations are independent to each other, otherwise problems like vector aliasing can be observed invaliding the output results.

C. Bluegene/Q intrinsics

In the case of Blue Gene/Q systems, the IBM XL C/C++ compiler supports 256-bit SIMD extensions. The vectorization can be generated by adding some compilation flags (automatically) or by using vector data types and the corresponding SIMD instructions (manually). To handle automatic vectorization is used the compiler flag `-qsimd=auto`, which indicates the compiler to enable vector registers when possible. In addition, the pragma directives `# pragma disjoint()` need to be embedded in the code, specifying the parts where there is no pointer aliasing. On the other hand, manual vectorization requires of code refactoring, the critical parts of the code are rewritten using the `vector4double` data type (quad) and vector intrinsic operations such as:

- `vec_ld` : loads data from a regular data type into quads
- `vec_st` : stores data from quads to a regular data type
- `vec_mult` : performs the multiplication of two quads
- `vec_madd` : executes the axpy operation with quads

These functions perform 4 double precision instructions in a single CPU-core cycle taking advantage of coalesced memory accesses.

IV. Implementation details

A. Unified Approach

The data structures used in the PCG algorithm are sparse matrices and vectors. As shown in section II-A, after the Fourier diagonalization, the Laplacian matrix is decomposed into a set N_{per} matrices that differ only in the eigenvalue λ_k multiplying the diagonal contribution Ω_{2d}.

$$\widehat{\mathsf{L}}_k = \lambda_k \Omega_{2d} + \Delta_{per}\mathsf{L}_{2d} \qquad k = 0, ..., N_{per} - 1. \tag{22}$$

A naive implementation would be storing independently each one of the N_{per} frequency systems. This approach facilitates the implementation of the algorithm because it relies on standard functions and data structures. However, it does not take advantage of the data redundancies resulting from the Fourier diagonalization. Our strategy consists in storing all the frequency systems on a single data structure avoiding redundancies, and creating the corresponding unified kernel which has a higher FLOP per by ratio. The set of matrices of frequency systems is represented by the Laplacian matrix $\Delta_{per}\mathsf{L}_{2d}$ and the diagonal matrix Ω_{2d}, both of dimension N_{2d}; and the vector λ_k of dimension N_{per} containing the eigenvalues of the Fourier operator. The vectors involved in

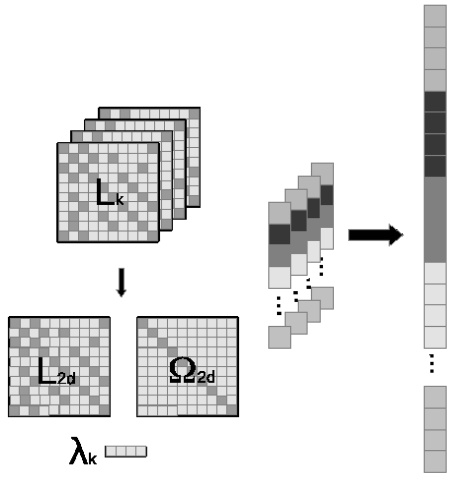

Figure 1: Left: Unified storage format. Right: Vectors arrangement

the solution of the frequency systems are dense data structures, therefore the only variant regarding its storage is in the ordering of the variables. This arrangement propitiates regular accesses to memory through the periodic direction. Figure 1 illustrates the unified storage format for the frequency systems and the corresponding vectors. A compressed storage of any sparse matrix requires at least the individual storage of the coefficients values and its corresponding columns indexes. The number of the non-zero entries is determined by the mesh geometry and the discretization scheme. In our application context, a 2.5D mesh is composed by triangles extruded through the periodic direction forming prismatic cells. We use a second order scheme for the spatial discretization, consequently, the 2D systems resulting from the decomposition contain 4 entries per row. Table I shows in detail the minimal number of bytes necessary to represent the set of frequency systems with both the naive and our novel approach. Generally the N_{2d} is several times larger than N_{per}, for this cases the new approach is almost N_{per} times more efficient in memory usage than explicitly storing each frequency system. Hence, the low memory footprint makes suitable the new approach when running simulations on systems-on-chip platforms, like Bluegene/Q, where the memory space is a scarce resource. The

Data Structure	Naive	Unified approach
Values (double)	$N_{per} \times (4N_{2d}) \times 8$	$(4N_{2d}) \times 8$
Columns (Int)	$N_{per} \times (4N_{2d}) \times 4$	$(4N_{2d}) \times 4$
Eigenvalues (double)	–	$N_{per} \times 8$
Total (bytes)	$(48\times\ N_{2d}\times\ N_{per})$	$(8N_{per} + 48N_{2d})$

Table I: Summary of SpMV memory usage for both storage approaches

SpMV kernel has to be adapted to the new storage format. Our implementation consists in computing the product by $\Delta_{per}\mathsf{L}_{2d}$ simultaneously for all the N_{per} frequency systems, while adding the corresponding contribution to the diagonal of each frequency system. This process is explicitly described in Algorithm 3:

Algorithm 3 Unified SpMV: $y = Lx$

```
1:  for i ∈ N_2d do
2:    for j ∈ [L_2d]_i do
3:      [y]_i = 0
4:      for k ∈ [0, N_per) do
5:        [y]_ik += L_2dij x_jk
6:        if i = j then
7:          [y]_ik += λ_k Ω_2di x_jk
8:        end if
9:      end for
10:   end for
11: end for
```

Where b and x are arrays of dimension $N_{per}N_{2d}$, and the subindex ik refers to the position $i \times N_{per} + k$ in the arrays. On the other hand, $[\mathsf{L}_{2d}]_i$ refers to the set of column indexes of the non-zero entries in the ith row of L_{2d}. Apart of reducing the global memory requirements, a key aspect of this implementation is the reduction of the memory traffic. Each coefficient of the matrix L_{2dij} is only fetched once from the RAM to the cache for all the frequency systems, this derives in important computing time savings as shown in Section V.

B. Communication reduction

As mentioned in the previous section, the N_{per} frequency systems that result from the Fourier diagonalization, are divided into P_{per} sub-sets containing $N_{per,k}$ subsystems each, and P_{2d} processors are assigned to the solution of each subset. Therefore, for the solution of each frequency system are engaged P_{2d} parallel processes. In our approach each operation of the PCG solver is performed simultaneously for the $N_{per,k}$ frequency systems composing a subset. As a consequence, the the all-to-all and point-to-point communications required by the norm, dot and SpMV operations are grouped and executed synchronously for all the P_{2d} processes, the benefit of this strategy is the reduction of the inter-core communications by a factor of $N_{per,k}$.

C. Auto-tuning

In section II-D, it is shown that not all the frequency systems have the same solution costs. This means that a

uniform partition of the set of frequency systems would derive in a significant imbalance, being the subset with the lowest frequencies the most expensive. Our approach to solve this problem is a dynamic load balance. The cost for the solution of each subset of frequency systems is monitored and the partition is adapted periodically, however a residual imbalance is tolerated in order to avoid an excessive overhead produced by the balance process. In particular, in our application context, once the flow reaches the statistically stationary regime, the solution cost (i.e. iterations) of the frequency systems remains almost constant for the rest of the simulation. Therefore, once a balanced state is achieved, additional balance operations are rarely required for the remaining steps of the simulation. Using an asymmetric partition of the frequency systems requires some changes in the communication pattern, because different processors need to send and receive different amounts of data. In particular, the change of basis from the physical to the spectral space and vice-versa requires substituting the `MPI_Alltoall` communications by `MPI_Alltoallv`. Changing the distribution of the frequency systems requires also to reevaluate the preconditioner for the systems that are redistributed, this produces an overhead but, as mentioned, in our application context the load balance process could be considered as "runtime preprocessing" which is only executed on the initial stages of the time integration process.

The second aspect that is automatically tuned is the choice of the preconditioner for each subset of frequency systems. A unique preconditioner is adopted for each subset in order to favor the vectorization. However, different subsets may be more efficiently solved by different preconditioning methods. In general, for the lowest frequencies is more optimal an accurate preconditioner since those are more ill-conditioned systems. While, the highest frequencies are much more diagonal dominant, therefore the Jacobi diagonal scaling performs very well. In the current version of our algorithm we combine two preconditioners, the sparse Approximate Inverse (AIP) [20] and the Jacobi diagonal scaling. A greedy algorithm based in alternate the use of the preconditioners and its parameters is utilized to estimate the best configuration. In the same way than the balancing, the preconditioners tuning takes place during the first steps of the simulation and its costs become negligible in our application context.

V. Performance results

The numerical experiments of this study were performed on the Blue Gene/Q Vesta supercomputer of the Argonne Leadership Computing Facility (ALCF), this is a test and development platform used as a pad for researchers to the largest ALCF supercomputer Mira (ranked 5th in the Top500 list). Vesta has two computer racks that sum up a total of 32,768 cores with a peak performance of ≈ 0.5 PFlops.

A. Vectorization

Figure 2 shows the GFLOPS achieved with the vectorized SpMV using our novel storage format versus the GFLOPS achieved with the naive approach that consists in multiplying separately one frequency system after the other. The test case is the Laplacian matrix discretized on meshes generated by the extrusion of a 2D mesh with 577K nodes, N_{per} varies from 4 to 32. The performance of the naive approach is independent of N_{per}, because increasing N_{per} is just repeating more times the same kernel. On the contrary, the performance of the unified SpMV improves with N_{per}, because the larger it is N_{per} the larger is the uniform part of the problem in which the vectorization and memory prefetching produce acceleration.

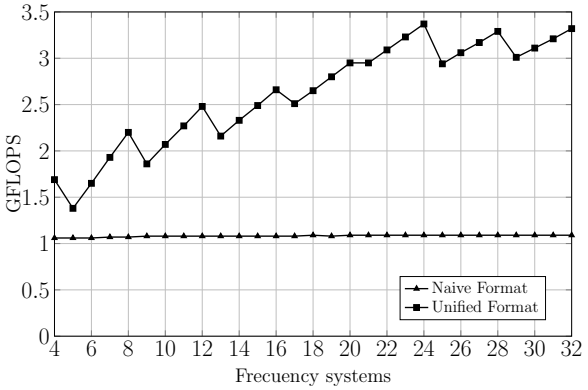

Figure 2: Unified SpMV versus naive approach for meshes generated by the extrusion of a 2D grid with 577K nodes

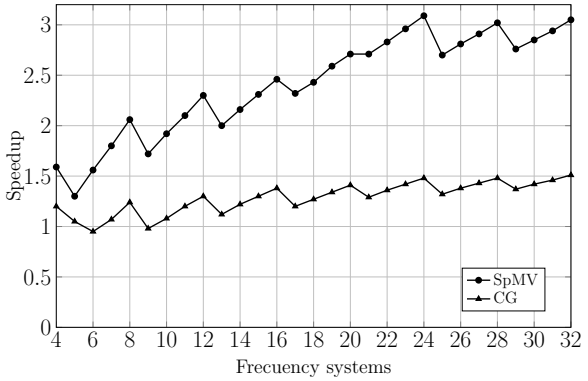

Figure 3: The speedup of cg and unified SpMV for meshes generated by the extrusion of a 2D grid with 577K nodes.

Since the vectorization is performed using vector data types composed of four doubles (quads). When N_{per} is multiple of 4 there is a perfect alignment and all the components of the multiplying vector fetched to the cache are effectively used, this results on the peaks of performance observed in the figure. The speedup of the unified SpMV versus the naive approach ranges between $1.4\times$ and $3.5\times$. Figure 3 shows the speedup of the unified versus the naive approach of both the SpMV and the CG with diagonal scaling (CG-diag). The performance improvement on the SpMV benefits the CG solver, but the improvement is limited by Amdahl's law since the relative weight of the SpMV on the CG-diag is around 60%. Using other methods such as the Approximate Inverse Preconditioner results in higher speedups versus the naive approach.

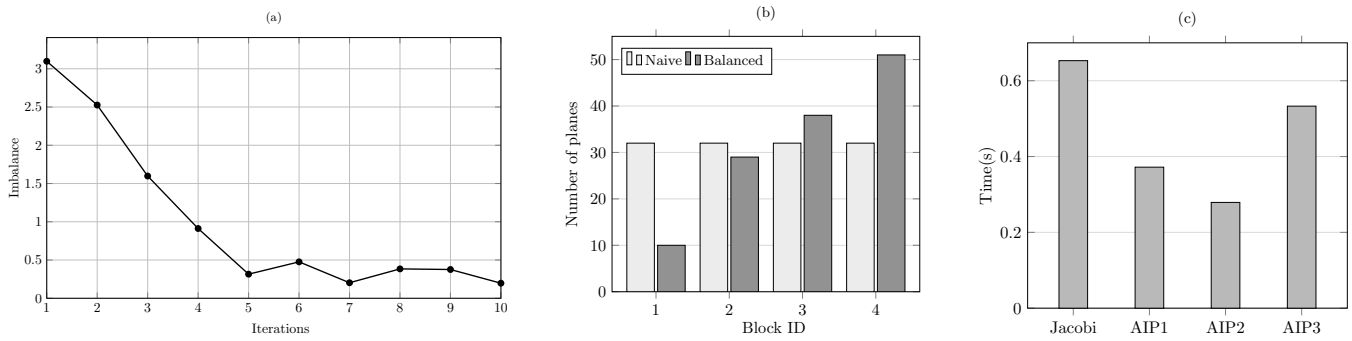

Figure 4: Auto-tuning process: (a) Load Imbalance reduction, (b) Initial and final distribution of frequency systems, (c) Preconditioner choice.

B. Autotuning

Figure 4 illustrates the auto-tuning process. Tests have been performed on a mesh of about 90 Million nodes composed of 12 planes of 700K nodes each. In particular, the results shown are for a 512×4 partition. In (a) is shown the imbalance reduction achieved with the auto-balance process, in this case the Diagonal Jacobi preconditioner is used for all blocks of frequency systems. The imbalance is measured as the maximum difference on the solution of different subsets of frequency systems, divided by the average time. In (b) is shown the initial and final distribution of planes, as expected, the block containing the larger frequencies are more loaded that the blocks containing the lower ones since the solution costs grow with the frequency number. Finally (c) shows the solution time obtained for the same test case with different preconditioning configurations. The first column corresponds to the case where Jacobi diagonal scaling is used for all the blocks of frequency systems, AIP1 refers to the case where only the first block is solved with the AIP preconditioner and the rest with the Jacobi diagonal, a AIP2 the two first blocks are solved with the AIP preconditioner and for AIP3 the first three blocks. The auto-tuning process ends when at increasing the number of blocks solved with the AIP preconditioner the solution time does not reduce. Therefore, in this particular case, the final configuration would be AIP2. Changing the preconditioning configuration has some associated costs: the set up for the new preconditioner and re-balance the distribution of frequency systems. But this can be considered as a runtime setup that does not require interruptions on the simulation process. In particular in this case the speedup achieved from the initial to the final configuration is $2.3\times$.

C. Scalability

A strong scalability test has been performed comparing our previous direct approach FFT-DSD, in which the frequency systems where solved by means of a Direct Schur-complement based Decomposition (DSD) [5], and the iterative solution here proposed (FFT-PCG). The same test case of the previous subsection is used here. The direct approach has been used in other supercomputers for LES simulations on meshes with up to 300M nodes [21], however 2GB of RAM per core where used in those simulations. The reduced size of of 1GB per core on Vesta supercomputer, allowed us to use only 8 of the 16 CPU-cores of the Vesta nodes. For this reason, we could only attest the performance of the FFT-DSD approach up to 8192 CPU-cores. In fact, this limitations are one of the reasons to evolve our solver. Results show that in the

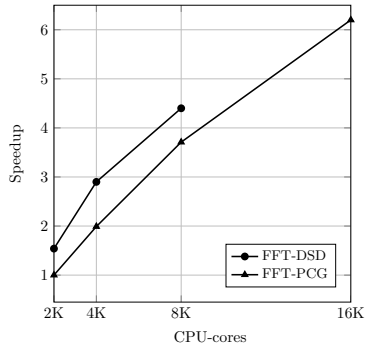

Figure 5: Strong speedup test for the direct (FFT-DSD) and iterative (FFT-PCG) approaches.

region where the FFT-DSD is applied (wasting half of cores allocated) it is faster than our new iterative approach. However the iterative approach scales better, initially (using 2048 CPU-cores) the FFT-DSD is 35% faster but with 8192 CPU-cores this difference reduces at 16%, finally using 16384 CPU-cores the iterative approach overcomes the FFT-DSD by 29%. The overall parallel efficiency of the FFT-PCG approach is 77%. Note that the load per CPU-core in the last case is only of about 5500 cells, which is a very small load for the Vesta CPU-cores, this fact glimpses a great scalability potential of the code.

VI. Conclusions

This article presents the efforts to evolve our Poisson solver for simulations with one FFT diagonalizable direction in order to align our strategy with the evolution of supercomputing systems. This evolution brings larger vector registers and less RAM memory per parallel process. We have adapted our previous solver into a purely iterative strategy, by solving all the frequency systems by means of a Preconditioned Conjugate Gradient method. In this new implementation we have

focused on optimizing the memory allocations and transactions and on taking advantage of the regularity of the memory accesses and operations through the periodic direction for its vectorization. The speedup achieved with the vectorization of the SpMV kernel, for which a specific format has been developed, averages $1.6\times$, with peaks of about $3\times$ when perfect alignment is achieved and enough frequency systems are operated simultaneously. Since the SpMV is the dominant kernel of the simulation code, a potential acceleration of all the code turns up, specially on the explicit parts of it. In the Poisson solver, we have observed that the acceleration of the Jacobi preconditioned CG averages $1.2\times$ with peaks of $1.5\times$, with respect of solving each frequency system separately. This result is consistent with the relative weight of the SpMV within the linear solver.

The second focus of our algorithm design has been the auto-tuning capabilities. The iterative solution of the frequency systems has variable cost according to the conditioning of each system. In general, the lower frequencies couple larger parts of the domain and require more iterations. On the other hand, the optimal preconditioning requirements of the frequency systems differ, the higher frequencies are strongly diagonal dominant and Jacobi diagonal scaling performs very well but the lower require a more accurate approximation. In order to deal with these variable situation, that depends on the physical problem being considered and the computing system engaged, we have developed a run-time auto-tuning that adjusts both aspects on the time integration process of the simulation without requiring user intervention neither the simulation interruption. Finally the strong scalability of the new algorithm has been successfully attested up to 16384 CPU-cores. The reduced memory requirements of our new approach, its demonstrated scalability and auto-tuning capabilities, and its good performance compared with the direct approach previously used in several HPC systems, make it a highly efficient and portable code adapted to the characteristics of ongoing HPC systems.

ACKNOWLEDGEMENTS

This work has been financially supported by the Ministerio de Ciencia e Innovación, Spain (ENE- 2014-60577-R), CONICYT Becas Chile Doctorado 2012, the Juan de la Cierva posdoctoral grant (IJCI-2014-21034), the Ramón y Cajal postdoctoral contract (RYC-2012-11996) and the Initial Training Network SEDITRANS (GA number: 607394), implemented within the 7th Framework Programme of the European Commission under call FP7-PEOPLE-2013-ITN. Calculations have been performed on the Vesta supercomputer at the Argonne Leadership Computing Facility. The authors thankfully acknowledge these institutions.

REFERENCES

[1] R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," *Journal of the Association for Computing Machinery*, vol. 12, pp. 95–113, 1965.

[2] M. Soria, C. D. Pérez-Segarra, and A.Oliva, "A Direct Schur-Fourier Decomposition for the Solution of the Three-Dimensional Poisson Equation of Incompressible Flow Problems Using Loosely Parallel Computers," *Numerical Heat Transfer, Part B*, vol. 43, pp. 467–488, 2003.

[3] F. X. Trias, M. Soria, C. D. Pérez-Segarra, and A. Oliva, "A Direct Schur-Fourier Decomposition for the Efficient Solution of High-Order Poisson Equations on Loosely Coupled Parallel Computers," *Numerical Linear Algebra with Applications*, vol. 13, pp. 303–326, 2006.

[4] A. Gorobets, F. X. Trias, M. Soria, and A. Oliva, "A scalable parallel Poisson solver for three-dimensional problems with one periodic direction," *Computers & Fluids*, vol. 39, pp. 525–538, 2010.

[5] R. Borrell, O. Lehmkuhl, F. X. Trias, and A. Oliva, "Parallel Direct Poisson solver for discretizations with one Fourier diagonalizable direction," *Journal of Computational Physics*, vol. 230, pp. 4723–4741, 2011.

[6] M. Soria, F. X. Trias, C. D. Pérez-Segarra, and A. Oliva, "Direct numerical simulation of a three-dimensional natural-convection flow in a differentially heated cavity of aspect ratio 4," *Numerical Heat Transfer, part A*, vol. 45, pp. 649–673, April 2004.

[7] F. X. Trias, M. Soria, A. Oliva, and C. D. Pérez-Segarra, "Direct numerical simulations of two- and three-dimensional turbulent natural convection flows in a differentially heated cavity of aspect ratio 4," *Journal of Fluid Mechanics*, vol. 586, pp. 259–293, 2007.

[8] I. Rodríguez, O. Lehmkuhl, R. Borrell, and A. Oliva, "Direct numerical simulation of a naca0012 in full stall," *International Journal of Heat and Fluid Flow*, vol. 43, pp. 194 – 203, 2013.

[9] I. Rodríguez, O. Lehmkuhl, J. Chiva, R. Borrell, and A. Oliva, "On the flow past a circular cylinder from critical to super-critical reynolds numbers: Wake topology and vortex shedding," *International Journal of Heat and Fluid Flow*, vol. 55, pp. 91 – 103, 2015.

[10] O. Lehmkuhl, C. Perez-Segarra, R. Borrell, M. Soria, and A. Oliva, *TermoFluids: A new Parallel unstructured CFD code for the simulation of turbulent industrial problems on low cost PC Cluster*, 2009, pp. 275–282.

[11] R. Borrell, J. Chiva, O. Lehmkuhl, G. Oyarzun, I. Rodriguez, and A. Oliva, "Optimising the termofluids cfd code for petascale simulations," *Int. J. Comput. Fluid Dyn.*, vol. 30, no. 6, pp. 425–430, Jul. 2016.

[12] A. J. Chorin, "Numerical Solution of the Navier-Stokes Equations," *Journal of Computational Physics*, vol. 22, pp. 745–762, 1968.

[13] N. N. Yanenko, *The Method of Fractional Steps*. Springer-Verlag, 1971.

[14] F. X. Trias, O. Lehmkuhl, A. Oliva, C.D. Pérez-Segarra, and R.W.C.P. Verstappen, "Symmetry-preserving discretization of Navier-Stokes equations on collocated unstructured meshes," *Journal of Computational Physics*, vol. 258, pp. 246–267, 2014.

[15] R. M. Gray, "Toeplitz and Circulant Matrices: A review," *Foundations and Trends in Communications and Information Theory*, vol. 2, pp. 155–239, 2006.

[16] P.J.Davis, *Circulant Matrices*. Chelsea Publishing, New York, 1994.

[17] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[18] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1999.

[19] Y. Saad, *Iteratives Methods for Sparse Linear Systems*. SIAM, 2003.

[20] Y. S. E. Chow, "Approximate inverse preconditioners via sparse-sparse iterations," *SIAM Journal of Scientific Computing*, vol. 19, pp. 995–1023, 1998.

[21] O. Lehmkuhl, I. Rodríguez, R. Borrell, J. Chiva, and A. Oliva, "Unsteady forces on a circular cylinder at critical Reynolds numbers," *Physics of Fluids*, p. 125110, 2014.