Parallel mesh partitioning based on Space Filling Curves

R.Borrell[a,*], J.C. Cajas[a], D. Mira[a], A. Taha[b], S. Koric[b], M. Vázquez[a], G. Houzeaux[a]

[a]*Barcelona Supercomputing Center*
[b]*NCSA, University of Illinois at Urbana Champaing, USA*

Abstract

Larger supercomputers allow the simulation of more complex phenomena with increased accuracy. Eventually this requires finer and thus also larger geometric discretizations. In this context, and extrapolating to the Exascale paradigm, meshing operations such as generation, deformation, adaptation/regeneration or partition/load balance, become a critical issue within the simulation workflow. In this paper we focus on mesh partitioning. In particular, we present a fast and scalable geometric partitioner based on Space Filling Curves (SFC), as an alternative to the standard graph partitioning approach. We have avoided any computing or memory bottleneck in the algorithm, while we have imposed that the solution achieved is independent (discounting rounding off errors) of the number of parallel processes used to compute it. The performance of the SFC-based partitioner presented has been demonstrated using up to 4096 CPU-cores in the Blue Waters supercomputer.

Key words: space filling curve, SFC, mesh partitioning, geometric partitioning, parallel computing

1. Introduction

Mesh partitioning is traditionally based on graph partitioning, which is a well-studied NP-complete problem generally addressed by means of multilevel heuristics composed of three phases: coarsening, partitioning, and un-coarsening. Different variants of them have been implemented in publicly available libraries such as Metis/ParMetis [1], Scotch/PT-Scotch [2] and

[*]Corresponding author, email: ricard.borrell@bsc.es

Zoltan [3]. All these libraries enable parallel partitioning, but with a limited parallel performance and a decreased quality of the parallel partition [4, 5]. Both aspects make graph-based partitioning a potential bottleneck in the simulation workflow. However, taking into account the evolution of the computing HPC systems, any potential bottleneck becomes an effective bottleneck if not addressed in time. Motivated by these circumstances, we present a fully parallel geometric partitioning alternative.

Geometric partitioning techniques obviate the topological interaction between mesh elements and perform its partition according to their spatial distribution. If we consider as unitary element the mesh cell, then its mass center can be used to determine its spatial location. A Space Filling Curve (SFC) is a continuous function used to map a multi-dimensional space into a one-dimensional space with good locality properties, i.e. it tries to preserve the proximity of elements in both spaces. The idea of geometric partitioning using SFC is to map the mesh elements into a 1D space and then easily divide the resulting segment into equally weighted sub-segments. A significant advantage of the SFC partitioning is that it can be computed very fast and it is easy to parallelize, especially when compared to graph partitioning methods. However, while the load balance of the resulting partitions can be guaranteed, the data transfer between the resulting subdomains, measured in terms of edge-cuts in the graph partitioning approach, cannot be explicitly measured and thus neither be minimized.

In this paper we describe and evaluate the performance of a parallel SFC based partitioner. We have avoided any computing or memory bottleneck that could limit the scalability of the algorithm, imposing that the solution achieved is independent (discounting rounding off errors) of the number of parallel processes used to compute it. The structure of the paper is as follows: Section 2 contains a short overview of the Hilbert SFC; in Section 3 we describe in detail the sequential and parallel versions of our SFC-based partitioner. In Section 4 the performance of the initial implementation is analyzed and some optimizations are evaluated; moreover, the partitions quality is compared with the quality of the equivalent ones computed with Metis. Finally, general conclusions are outlined section 5.

2. Hilbert Space Filling Curve

There are many possible definitions of SFC based on different mapping options, among them the well-known Peano [6] and Hilbert [7] approaches.

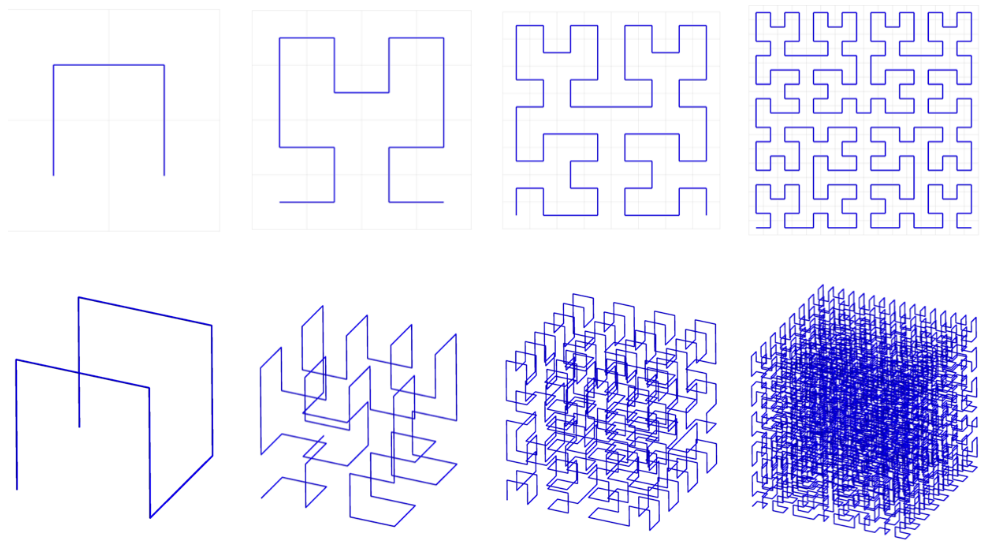


Figure 1: Hilbert SFC recursive generation, 1D (top) and 3D (bottom)

A complete overview is given in [8]. In this paper the Hilbert SFC is selected, however switching to another mapping option would be straightforward in our implementation.

The Hilbert SFC is defined by means of a geometric recursion. For the 2D case, on the $p'th$ level of the recursion a discrete function is obtained:

$$h : \{1, ..., 2^{2p}\} \longrightarrow \{1, ..., 2^{p}\}\{1, ..., 2^{p}\} \tag{1}$$

determining the ordering of the curve through a $2^p \times 2^p$ cartesian grid. The four initial steps of this recursion are represented in Figure 1(top). On the first step, the curve has a "⊓" shape traversing a 2×2 grid. This is refined into the curve that traverses the 4×4 grid corresponding to the second step. This 4×4 grid can be decomposed into four 2×2 sub-grids, where the curve takes the shapes "⊐", "⊓", "⊓" and "⊏", respectively. Following a specific refinement pattern for each of these basic 2×2 shapes we obtain the third level, and this process is continued until the desired level of refinement is achieved. Therefore, the process of generating a Hilbert curve, is based on recursively applying a specific refinement pattern to the basic 2×2 shapes generated at each level of the recursion. This algorithm can be implemented by cyclic lookups to two arrays: the first storing the orderings which define the basic 2×2 shapes, the second its refinement patterns, also referred as orientations. Details of this implementation approach can be found in [9]. The same idea is followed in the 3D case, where the basic shapes are defined on $2 \times 2 \times 2$ sub-grids. The first four steps of the 3D recursion are depicted in Figure 1 (bottom). Note that a Hilbert SFC can be defined on any rectangular or cubic domain, by using a grid with 2^p elements on each axis.

A Hilbert SFC can also be generated in parallel following a multilevel approach. An initial coarse grid is defined, and the SFC-index of each of its elements, hereafter referred as coarse bins, determines the rank of the parallel process continuing the recursion within it. If the proper orientation is followed within each coarse bin, the resulting SFC generated in parallel is the same that would be obtained sequentially. This process is illustrated in Figure 2. Generating an SFC in parallel has two advantages: first it is faster and second, since there is more memory available for the overall SFC, a more refined one can be generated. Recalling that the ordering of the coarse bins is based on an SFC, this imposes a restriction to the number of parallel processes, P, generating the SFC: P, must be a power of 2^d, where $d \in \{2, 3\}$ is the dimension of the domain.

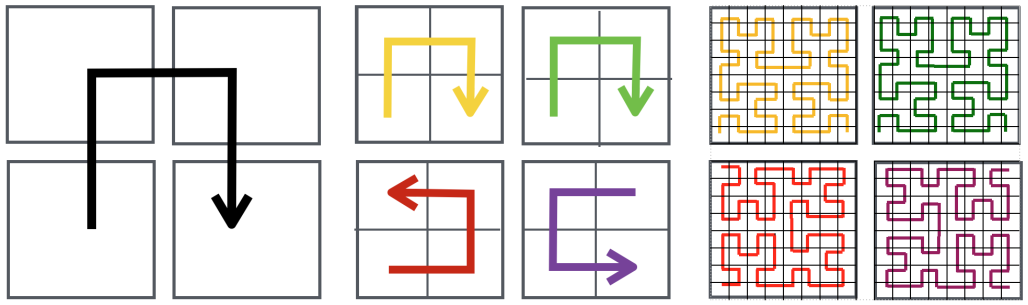


Figure 2: Hilbert SFC parallel generation. a) Initial coarse grid definition, b) orientation of each parallel process to continue the recursion, c) recursive generation of SFC in each parallel process.

3. SFC based mesh partitioning

3.1. Sequential version

Consider the problem of partitioning a mesh in M subsets using the Hilbert SFC described in Section 2. A sequential algorithm to obtain the desired partition is as follows:

1. Define a minimum bounding box enclosing the mesh.
2. Define a cartesian grid with 2^p elements in each direction, its cells are referred as bins. Note that p recursion steps will be necessary to generate the corresponding Hilbert SFC.
3. Embed the mesh elements into the grid bins. For each mesh element a weight is added to the grid bin containing it. A mesh partition can be based on the distribution of nodes (vertices) or cells, in this paper we consider cells. We use the mass center to determine the bin that contains each cell, and the weight of a cell is defined as its number of nodes.
4. Traverse the cartesian grid and label each bin with a partition number. The SFC ordering is used to traverse the grid. The associated mapping function requires p steps for each evaluation, therefore if $N = (2^d)^p$ the cost of this step is $O(Nlog(N))$. While the cartesian grid is traversed, the bins weight is accumulated on a variable and, each time the *objective weight* is reached, the partition number assigned to the bins is incremented and the accumulation variable is reinitialized to zero. The *objective weight* is initially defined as the total weight accumulated in the grid divided by M. Then it is re-evaluated each time a subset is completed, considering the remaining weight and subsets to complete, to avoid the accumulation of discrete errors.
5. Infer the mesh partition from the bins partition. This step is straightforward, the partition number of each mesh element is the one of the bin containing it.

The steps of the sequential partitioning algorithm are illustrated in Figure 3.

3.2. Parallel version

The parallel counterpart of the algorithm is described below. As initial conditions, we assume that there is a list of N_c cells distributed among P parallel processes. We assume that P is a power of 2^d, i.e. $P = 2^{dq_c}$, where q_c are the levels of the coarse SFC recursion. The initial distribution of the mesh cells among the parallel processes could come from: i) a parallel read of the mesh file, carried out at the initialization phase of the simulation; or ii) a previous partition, when a mesh re-partitioning is considered, e.g. for load balancing purposes. The parallel SFC-based mesh partitioning algorithm has the following steps:

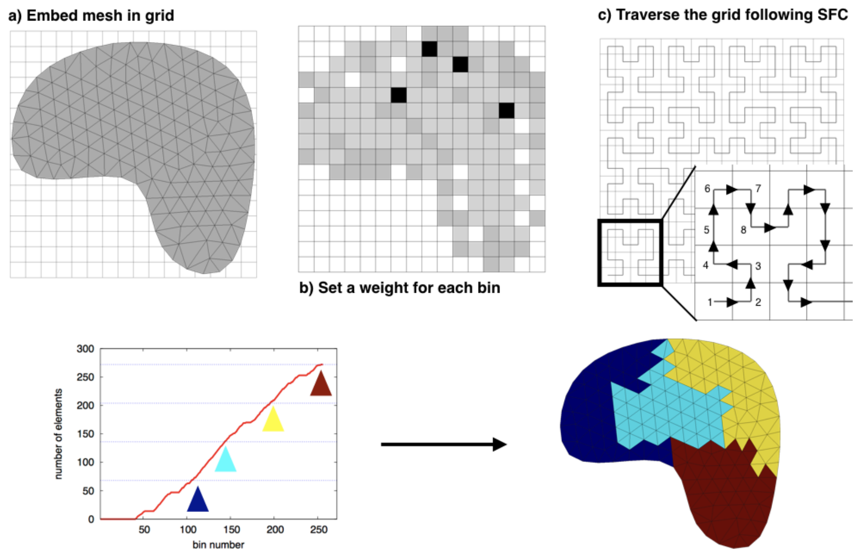

a) Embed mesh in grid
b) Set a weight for each bin
c) Traverse the grid following SFC
number of elements
bin number

Figure 3: Sequential algorithm for mesh partitioning using the Hilbert SFC. a) The mesh is embedded in a cartesian grid, b) a weight is assigned to each element of the cartesian grid ("bins") depending on the number of elements contained in it, c) the cartesian grid is traversed in the SFC order, accumulating the weights of the bins and assigning subset numbers depending on the objective weights of each subset.

1. Define the bounding box and its cartesian grids. A minimum bounding box is defined enclosing the mesh. This requires a collective reduction communication (`MPI_Allreduce`), to find the maximum and minimum value for each coordinate among the mesh nodes. Then a *coarse grid* is defined dividing the box. Each parallel process will account for the mesh portion contained in one of the coarse bins, in particular the one with SFC-index equal to its rank. At this point it is also determined the depth of the Hilbert recursion used within the coarse bins, referred as q_l. Consequently, the size of the *local grids* will be $N_g^l = (2^d)^{q_l}$, and the size of the *fine grid*, resulting from joining the *local grids*, will be $N_g^f = (2^d)^{q_l q_c}$.
2. Evaluate and redistribute of bins weight. Each parallel process evaluates the weight of the elements of its sublist accumulating it into the corresponding fine bins. Then the weights accumulated in the fine bins

are redistributed such that each parallel process ends up with the ones contained in its local grid. In order to accumulate the fine bins weight while looping trough the sublist of cells, it would be required an array of size N_g^f, or an sparse data structure such as a hash table. Alternatively, in this paper we split this evaluation into different sub-steps. These are detailed below from the perspective of one of the parallel processes:

- Identify which coarse bins contain elements of its sublist (loop around sublist required).
- For each of these:
  - Allocate a buffer of size N_g^l to accumulate the weights in the bins of the corresponding local grid.
  - Loop around the mesh elements sublist accumulating weights on the buffer.
  - Store the non-zero weight bins on a new buffer (a loop around the first buffer of size N_g^l is required).
- Collective communication (`MPI_Alltoall`) so that each parallel process gets the size of the buffers it will receive from other processes
- Point-to-point communications (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait`) to re-distribute weighted bins.
- Accumulate the bins weights in the local grid. The local bins weight may have been received from other parallel processes or may have been evaluated from the same process sublist. Note that, the processes without any mesh element contained in its coarse bin will end up with an empty local grid.

3. Gather the coarse bin weights to all processes. A collective `MPI_Allgather` communications is carried out to obtain the weights distribution across the parallel process.
4. Local grids partition. From the coarse weights distribution obtained in the previous step, each parallel process can evaluate the relative weight of its coarse bin, γ_i, and the relative weight accumulated by the parallel processes with lower rank, $\Gamma_i = \sum_{j<i} \gamma_j$. Note $\Gamma_i \leq 1$ for all the parallel processes. Recall M are the subsets desired for the final partition, then $\Gamma_i M$ subsets will be completed by processes of rank lower than i. In order to generate the partition in a coherent way, the assignation of the starting subset for each parallel process has to be done carefully.

Consider the general case where $\Gamma_i M$ is not an integer number and define the *residual subset* r_i as $r_i = \Gamma_i M - [\Gamma_i M]$, where [] stands for the floor function. The starting subset for parallel process i will be $[\Gamma_i M] + 1$ and the objective weight for this subset will be scaled by the factor $(1 - r_i)$, given that the parallel processes with ranks lower than i have to cover the r_i fraction of it. Once the starting partition number and the objective weight for the first subset are known, the parallel process can start its local partition. SFC partitioning methods are flexible enough to generate subsets of different sizes. This situation is illustrated in Figure 4, where a mesh is partitioned by four parallel processes into five subsets. Each parallel process starts assigning subset numbers according to the coarse weights accumulated by the parallel processes of lower ranks. In the example, the first subset is filled by ranks 0 and 1, the second by ranks 1 and 2, the third by 2 and 3, and finally the fourth and fifth subsets are filled by rank 3. Note that joining the local SFC partitions generated in parallel, the result is the same that would be obtained sequentially.

5. Fine bins redistribution. The point-to-point communications (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait`) opposite to those made in step 2 are performed. As a result each parallel process obtains the partition number of the bins containing cells of its sublist.
6. Infer mesh partition. This step is straightforward, the partition number of each element of the sublist is the one of the fine bin containing it (received in previous step).

Note that each parallel process does not account for more than $1/P$ of the mesh elements of the fine grid at any step of the algorithm. Therefore, the algorithm does not present any memory or computational bottleneck. Regarding data transfers, there are three collective communications: in steps 1 (`MPI_Allreduce`), in step 2 (`MPI_Alltoall`) and in step 3 (`MPI_Allgather`); and two point-to-point communications: steps 2 and 5 (`MPI_Isend`, `MPI_Irecv`, `MPI_Wait`). Its influence on the parallel performance is analyzed in the following section.

Regarding the restriction of P being a power of 2^d, imposed by the Hilbert SFC, there are alternative SFC without such restriction, e.g. the Peano or Morton SFCs [8]. In our case, we fix P as the largest power of 2^d minor or equal than M (the number of processes used for the simulation). An alternative option is to use a coarse grid with more bins than partitioning processes and assign more than one coarse bin to each parallel process.

The sequential and parallel executions of the algorithm produce an almost identical partition if the fine grid resulting from joining the local grids is the same than the one used for the sequential partition. However, since a larger grid can result in a better balanced partition, the parallel execution can produce more accurate results. An opposite behavior is observed with graph partitioners, where sequential and parallels results are generally different and the parallelization worsens the result [4, 5]. In case of identical grids, there can be a minor difference between the parallel and sequential SFC algorithms. This is because of the *objective weight* re-evaluation used to distribute the discrete errors at generating the subsets (see step 4 of the sequential algorithm). In the sequential case this re-evaluation is performed $P-1$ times and affects all the subsets, while in the parallel case it is performed locally within each coarse bin, the number of repetitions will depend on the portion of mesh elements partitioned by each parallel process.

Finally, repartitioning and load balancing based on SFC has a significant advantage: the partition adjustments consist in moving the cutting points along the 1D segment defined by the SFC. Therefore, changes occur in the extremes of the sub-segments obtained from the former partition and, in general, most of mesh elements can be retained in the same subdomain.

The steps of the parallel partition algorithm are outlined in Algorithm 1.

Algorithm 1 Parallel SFC-based partitioning

```
1. Definition of a bounding box and its cartesian grids
2. Evaluation and redistribution of fine bins weight
3. Gather coarse bin weights to all processes
4. Local grids partition based on SFC
5. Fine bins redistribution
6. Infer mesh partition
```

4. Computational tests and optimizations

In this section we present the computational tests carried out and the optimizations implemented according to the intermediate results obtained. We have considered two test cases, the first one is the simulation of a sniff flow in the respiratory system. The heterogeneous and anisotropic mesh

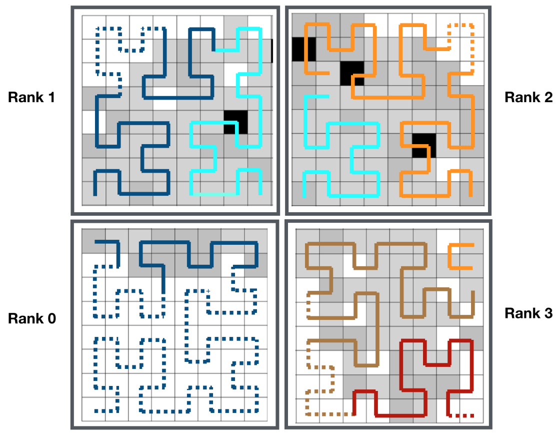


Figure 4: Parallel mesh partitioning using SFC.

illustrated in Figure 5 (left) is used for this case. It is mainly composed of prisms in the boundary layer and tetrahedra in the rest of the domain, while some pyramids are required on the transition between both regions. In total it has 17.7M cells. For the numerical solution of the sniff flow a large-eddy simulation (LES) formulation of the Navier Stokes equations is considered. This is coupled with a Lagrangian particle tracking solver to evaluate depositions. An illustrative image of the results is shown in Figure 5 (right). Further details about this case can be found in [10, 11].

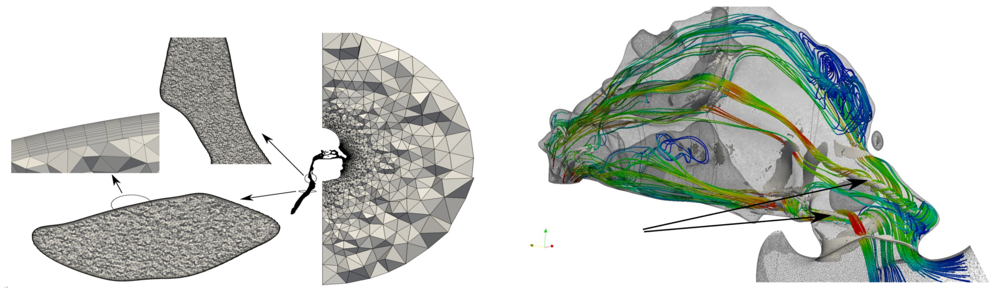


Figure 5: Geometric discretization of the respiratory system (left). Streamlines colored by velocity magnitude and iso-surface of Q-criterion pointing by arrows in the nasal cavities and throat(right).

The second case under consideration is the numerical simulation of a swirling combustor. The computational domain is also discretized with a highly anisotropic mesh composed of prisms, tetrahedra and pyramids; details are shown in Figure 6 (left). The mesh is composed of 28.9M cells in total. The numerical approach for the fluid is also a LES while the turbulent combustion model is based on laminar premixed flamelets. Further details on this case can be found in [12], an illustrative image of the flow dynamics is presented in Figure 6 (right).

The computational tests have been carried out on the Blue Waters supercomputer from the National Center for Supercomputing Applications (NCSA) at the University of Illinois (U.S.A). In particular we employed the XE nodes composed of two 16-core AMD 6276 "Interlagos" processors (nominal clock speed of at least 2.3 GHz), and connected by the Cray Gemini torus network.

The parallel SFC based partitioner developed in this paper and the physics solvers used in the test cases are both integrated in Alya [13, 14],: the high performance computational mechanics code developed in the Barcelona Supercomputing Center. The physics solvable with the Alya system include: incompressible/compressible flow, solid mechanics, chemistry, particle transport, heat transfer, turbulence modeling, electrical propagation, etc. Alya is designed for massively parallel supercomputers [15], its parallelization includes both the MPI and OpenMP frameworks, as well as heterogeneous options including accelerators. Alya is one of the twelve simulation codes of the Unified European Applications Benchmark Suite (UEABS) of PRACE and thus complies with the highest standards in HPC.

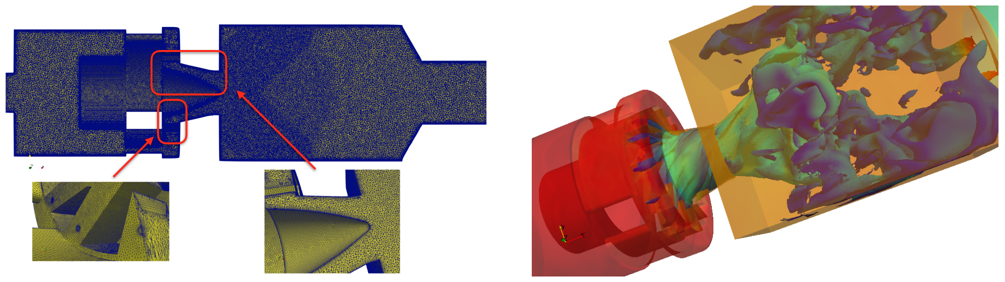


Figure 6: Geometric discretization of a swirling combustor (left). Instantaneous iso-surface of stoichiometric mixture fraction colored by pressure (right).

4.1. Parallel performance of the partitioning process

4.1.1. Initial results

The first test carried out is the measurement of the time required for the partition of the respiratory system mesh (17.7M cells), varying the number

#CPU	initial version	no redist.	no comm.
8	1,31	0,45 (34%)	1,27 (97%)
64	0,31	0,057 (18%)	0,30 (97%)
512	0,18	0,0078 (4%)	0,064 (36%)
4096	0,20	0,0037 (2%)	0,023 (11%)

Table 1: Partitioning time (sec.) with different numbers of CPU-cores. For the initial version of the partitioning algorithm (column 2). Skipping data redistributions (column 3) and skipping only MPI calls inside redistributions (column 4). In parenthesis percentage of the total time. Respiratory system mesh (17.7M).

of CPU-cores from 8 up to 4096. Even when the execution with 4096 CPUs is quite unrealistic, given that a load of $4.3K$ cells per CPU-core is very low, it is useful to show the scalability barriers of the algorithm. Starting with 8, the number of CPUs is incremented by a factor 8 on the successive cases considered. In this way a SFC can be used to assign the coarse bins to the parallel processes. While the size of the coarse grid is multiplied by 8 in the successive cases, the local grid size is divided by 8, therefore the fine grid is kept constant. The partitions generated have 512 subsets, however, the partitioning time is independent to this number. Results are shown in column 2 of Table 1, and are depicted in Figure 8 (left), under the label "SFC initial", for its comparison with some optimizations described below. The partitioning process is very fast: a minimum time of 0.18 seconds is observed for partitioning this 17.7M elements mesh. However the scalability is very limited: with 512 CPU-cores the parallel efficiency is only 11% and 1% with 4096 CPU-cores. Despite the algorithm may be fast enough in some contexts we aim to further understand its scalability limitations, having in mind its potential usage for repartitioning and load-balancing processes.

4.1.2. Analysis

Following the well known Amdahl's law principle, the first step to do is to find the part of the algorithm which dominates the execution costs. First, we have skipped steps 2 and 5 of Algorithm 1, in which data redistributions are carried out. As shown in column 3 of Table 1, the rest of the algorithm takes only 34% of the partitioning time with 8 CPUs-cores, and ends up representing only 2% with 4096 CPU-cores. This percentage decrease reflects a better parallel efficiency of the part being considered, which can be calculated from Table 1: 90% with 512 CPU-cores and 24% with with 4096 CPU-cores. In conclusion, our SFC partitioning algorithms is dominated by the cost of the redistribution steps, the rest of the algorithm scales well - the scalability achieved with 4096 is good considering the low workload per CPU - and is extremely fast. Consequently, in the rest of this section we focus on the analysis and the optimization of steps 2 and 5 of Algorithm 1.

Algorithm 2 Step 2 of Algorithm 1

```
1. Loop around the sublist of mesh elements to count #elements to be sent to other proc.
2. For each proc. $q$ to which messages have to be sent:
    (a) Allocate buffer of size $N_g^l$ (local grid of proc. $q$)
    (b) Loop around the sublist to accumulate weight on the buffer
    (c) Store in the "send to proc. q" buffer the bins with non-zero weight
3. MPI_Alltoall to generate the communication scheme
4. Point-to-point:
    (a) For each other process sending a buffer to me MPI_Irecv
    (b) Send all the buffers evaluated in step 2, MPI_Isend
    (c) MPI_Waitall
5. Accumulate weight in local grid
```

The step 2 of Algorithms 1 is shown in detail in Algorithm 2. Step 5 is just the point-to-point communications opposite to those made in step 2.4 (step 4 of Alg. 2). The next test considered is the execution of the Algorithms 1 discarding only the MPI calls of steps 2.3, 2.4 and 5, respectively. Results are shown in column 4 of Table 1. The cost of the three communication episodes eliminated is minimal (3%) up to 64 CPU-cores, but represents 64% and 89% of the partitioning time with 512 and 4096 CPU-cores, respectively. It is thus clear that the principal scalability bottleneck are these three MPI communications that have been omitted in this last test.

4.1.3. Optimizations

The first optimization that we have considered is hiding communications under computations in Algorithm 2. In step 2.3 there is a `MPI_Alltoall` communication used to get on each parallel process the size of the buffers that it will receive from the others. The size of each buffer is proportional to the number of weighted local bins to be sent to the corresponding process. In step 2.1 are counted the number of elements of the initial sublist contained in each coarse bin. This is an upper bound for the number of bins which are finally sent, since more than one element can be contained in the same local bin. Therefore, it is not possible to know the exact size of the buffers until they are generated (step 2.2). However the upper bound evaluated in step 2.1 is enough to start the communications because, according to MPI specifications, the non-blocking `MPI_Irecv` instruction can be started using as "count" parameter the "maximum number of elements in receive buffer". In conclusion, communications can be started before the buffering process is carried out.

The corresponding optimized algorithm is shown in Algorithm 3. The `MPI_Alltoall` communication has been advanced just after step 3.1 - the #elements in each coarse bin are exchanged instead of the send buffers size. Subsequently the `MPI_Irecv` instructions are started (step 3.3). Finally, the non-blocking instructions `MPI_Isend` (step 3.4.d) are interleaved with the weight evaluation and buffering (steps 3.4.a-3.4.c). On the other hand, in the step 5 of Algorithm 1, such an overlapping is not possible because only are performed a point-to-point communications without any substantial computation.

Algorithm 3 Optimized step 2 of Algorithm 1: overlapping

```
1. Loop around the sublist of mesh elements to count #elements to be sent
   to other proc.
2. MPI_Alltoall to generate the communication scheme
3. Point-to-point: for each other process sending a buffer to me MPI_Irecv
4. For each proc. $q$ to which messages have to be sent:
   (a) Allocate buffer of size $N_g^l$ (local grid of proc. $q$)
   (b) Loop around the sublist to accumulate weight on the buffer
   (c) Store in the "send to proc. q" buffer bins with non-zero weight
   (d) Point-to-point: send the buffer MPI_Isend
5. Point-to-point: MPI_Waitall
6. Accumulate weight in local grid
```

The latency of the partitioning algorithm with this new version of step 2 is shown in column 3 of Table 2. With 8 and 64 CPU-cores the new algorithm is slightly slower than the initial version. As previously shown in Table 1, the communication costs represent a minimal part of the execution

#CPU	initial version	opt. 1	opt. 2
8	1,31	1,37 (104%)	1,37 (104%)
64	0,31	0,33 (106%)	0,33 (106%)
512	0,18	0,078 (43%)	0,071 (39%)
4096	0,20	0,28 (140%)	0,037 (18%)

Table 2: Partitioning time (sec.) with different optimizations of the partitioning algorithm. Opt. 1: overlapping communications and computations. Opt 2: overlapping + initial redistribution. Respiratory system mesh (17.7M).

in these two cases (3%), therefore, there is not much to overlap. In fact, as the results demonstrate, it is better to perform all the communications together instead of interleaving them with calculations. In the execution with 512 CPU-cores, the benefit is noticeable, the partitioning time is reduced by 57%. Finally, in the extreme case with 4096 the new algorithm performs much worse than the initial one (40%). In this case the communications represent 89% of the partitioning time (see Table 1). Thus, like for the first two cases, the ratio between communications and computations is very unbalanced and there is not much to overlap. However, in this case the dominant part are the communications instead of computations. Again it is better to perform computations and communicators in two consecutive blocks instead of interleaving them. Results in terms of strong speedup are shown in Figure 8 (left).

A second issue observed regarding the parallel performance is the load imbalance. In concordance with the previous analysis, we are focusing on the step 2 of Algorithm 1 and, in particular, on the communication operations performed in it. The `MPI_Alltoall` operation does not produce any imbalance because has the same cost for all the parallel processes involved in it. However, the number of calls to the `MPI_Irecv` and `MPI_Isend` functions can be different among parallel processes and, eventually, produce an imbalance.

In particular, the number of calls to the `MPI_Isend` function performed by one parallel process, depends on the number of mesh elements composing its initial sublist, and on how these are distributed in the domain. The more the collisions in the same coarse bin the less the number of messages to be sent. The more the collisions within bins of the local sub-grids the more are reduced the buffer sizes. In our tests, we have distributed the mesh elements into sublists of the same size, using the default ordering coming from the mesh generator.

On the other hand, the number of calls to the function `MPI_Irecv` by each parallel process, depends on the number of mesh elements contained in its coarse bin, and on how these are distributed among the initial sublists. This is the most critical part in terms of load imbalance. The coarse bins enclosing zones where the mesh is more refined, contain more mesh elements and receive more messages from other parallel processes. In our example, the mesh of the respiratory system is very anisotropic and is specially dense on the nasal cavities and throat. Consequently the processes accounting for these zones receive much more messages than the others. This produces a significant imbalance. In fact, there are some parallel processes having an associated coarse bin without mesh elements - corresponding to an empty zone of the bounding box - and others receiving messages from more than 75% of the rest of parallel processes.

In order to reduce the imbalance generated in the data transmissions, we have designed a multilevel strategy. If P is the number of partitioning processes (or coarse bins), with $P = 2^{dp}$. We define a new grid, referred as *vast grid*, with $Q = 2^{dq}$ elements, where $q < p$. The P partitioning processes can be divided into $P/Q = 2^{d(p-q)}$ subsets with Q processes each. The ideas is to redistribute the elements of the initial sublists within each subset of Q processes, grouping the elements contained in the same bin of the vast grid. With this redistribution, the number of calls to the functions `MPI_Isend` and `MPI_Irecv` is bounded by P/Q, one per subset of Q processes. Moreover, this initial redistribution benefits the collision of element into bins of the coarse and local grids. This optimization can be seen as a modification to the input data of Algorithm 3. Q is chosen as a relatively low number and the redistribution is performed with the `MPI_Alltoallv` function. The new version of the step 2 of Algorithm 1 is presented in Algorithm 4. To keep the coherence with this new version of the step 2, in the step 5 of Algorithm 1 are carried out first the point-to-point communications opposite to the `MPI_Irecv/Isend` instructions of steps 4.4 and 4.5.d. And then the one opposite to the `MPI_Alltoallv` of step 4.1. This multilevel approach is illustrated in Figure 7.

The result of this optimization is shown in column 4 of Table 2. We have fixed $Q = 128$, in the first two cases the redistribution is not carried out since the number of partitioning processes is lower than Q. With 512 CPU-cores the partitioning time is reduced by 61% with respect to the initial version

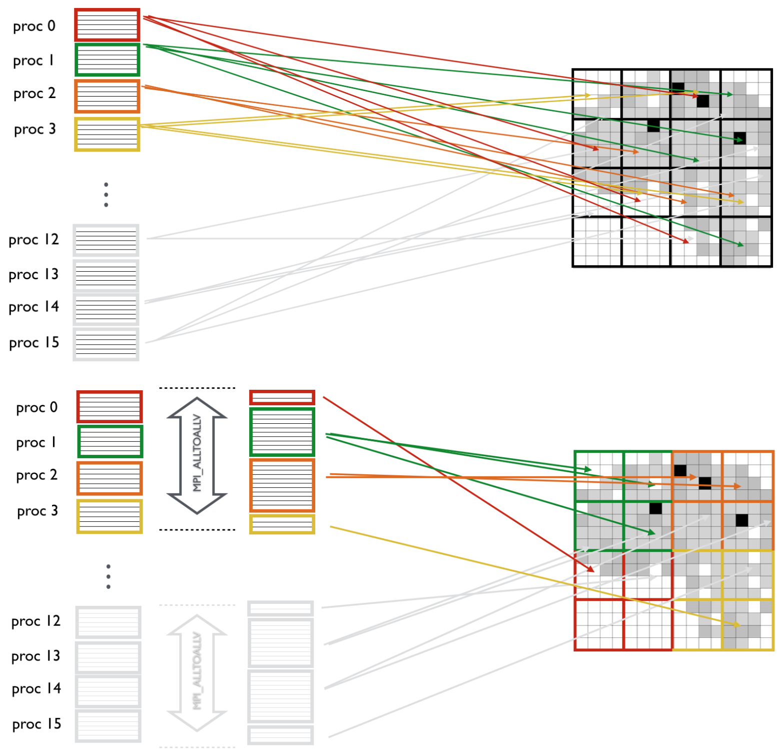


Figure 7: Illustration of step 2 of Algorithm 1: initial version (top) optimized version with redistribution of sublist elements (bottom).

Algorithm 4 Optimized step 2 of Algorithm 1: overlapping and sublists redistribution

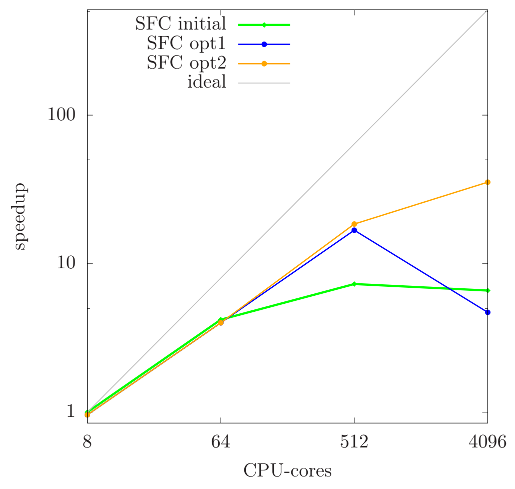
```
1. if($P \geq Q$) First level sublists redistribution: MPI_Allotoallv
2. Loop around the sublist of mesh elements to count #elements to be sent
   to other proc.
3. MPI_Alltoall to generate the communication scheme
4. Point-to-point: for each other process sending a buffer to me MPI_Irecv
5. For each proc. $q$ to which messages have to be sent:
   (a) Allocate buffer of size $N_g^l$ (local grid of proc. $q$)
   (b) Loop around the sublist to accumulate weight on the buffer
   (c) Store in the "send to proc. q" buffer bins with non-zero weight
   (d) Point-to-point: send the buffer MPI_Isend
6. Point-to-point: MPI_Waitall
7. Accumulate weight in local grid
```

#CPU	initial version	opt. 2
8	2,11	2,11 (100%)
64	0,58	0,57 (98%)
512	0,32	0,081 (25%)
4096	0,57	0,046 (8%)

Table 3: Partitioning time (sec.) with the initial version of the partitioning algorithm and its final optimization, for the swirling combustor mesh (28.9M).

and, with 4096 by 82%. This final algorithm is capable of partitioning the $17.7M$ mesh of the respiratory system in less than 4 cents of second using 4096 CPU-cores. Results in terms of strong speedup are shown in Figure 8 (left).

Finally, in Figure 8 (right) are compared the strong scalability of the initial and final versions of the algorithm for the second example, the mesh of the swirling combustor with 28.9M elements. The time measurements are shown in Table 3. The final version of the algorithms is more than 10× faster than the initial version using 4096 CPU-cores, in particular the mesh is partitioned in only 5 cents of second.



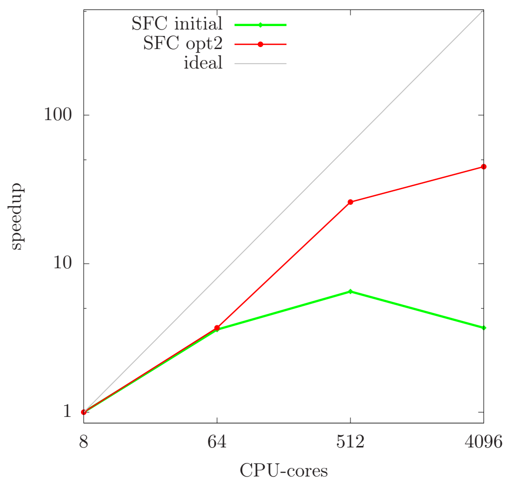


Figure 8: Strong speedup of the initial version of the partitioning algorithm and two optimizations: overlapping communications and computations (opt1), overlapping and initial redistribution (opt2). Left: mesh of the respiratory system (17.7M). Right: mesh of the swirling combustor (28.9M).

4.2. Partition quality

The present work is focused on the efficiency of the parallel partition algorithm. However, given that the ultimate goal of mesh partitioning is to enable the parallel execution of simulation codes, we compare the partition quality obtained with the presented SFC algorithm with the one obtained with Metis. To this end the Metis k-way algorithm with its default parameters was selected, which is very common in practice. In Figure 9 we compare the average time-step duration with different partitions for the two test cases under consideration: the respiratory system (left) and the swirling combustor (right). In each test case a different set of equations is solved, but in both of them the time-step is composed of an explicit part, in which the linear system is assembled, and an implicit part in which a linear solver is applied.

Results for both partitions and test cases are rather similar. Metis aims to generate balanced partitions minimizing also communication costs, while the SFC approach does not consider communication costs explicitly, but can be very precise on the load balancing. In general, in a system like the Blue Waters one, a load of $\sim$ 35K mesh elements per CPU-core would be considered optimal for a simulation with Alya. Therefore, the most representative executions of this test are the ones with 512 and 1024 CPU-cores. In the res-

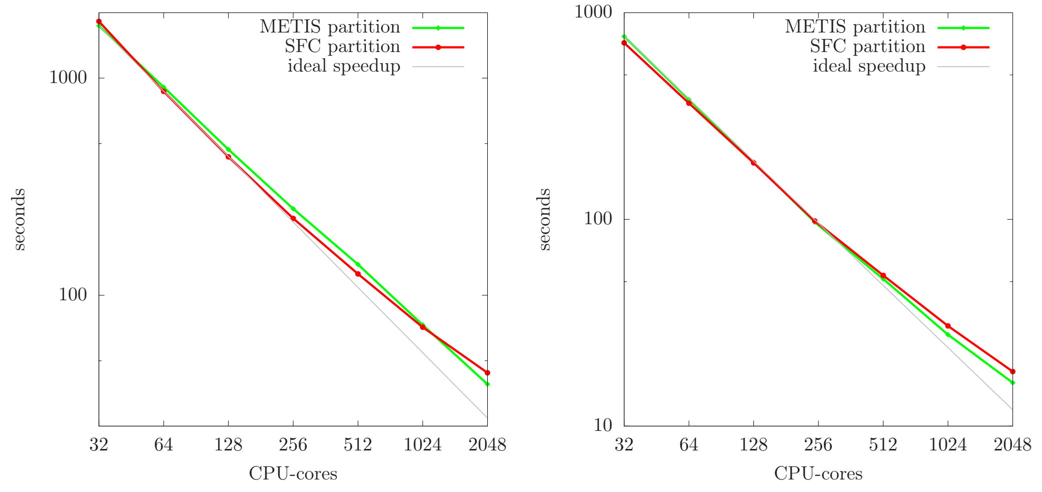


Figure 9: Average latency per time-step on the simulations of the sniff in the respiratory system (left) and the combustion in a swirling combustor (right), for partitions generated with SFC and Metis.

piratory system the SFC partition outperforms the Metis partition by 10% and 3%, respectively; contrarily, for the combustor the Metis partition outperforms SFC partition by 4% and 9%, respectively. The graph obtained from the discretization of the irregular respiratory system geometry makes difficult the balancing for Metis. When the load per CPU is very low communication costs become critical and graph partitions are a better option.

Note that the SFC partitions have been generated in parallel, setting P (the number of partitioning processes) as the highest power of 8 lower then M (the number of subdomains generated), in other words, using the largest possible subset of the parallel processes engaged on the simulation for the mesh partitioning. On the other hand, Metis partitions have been generated sequentially but some studies show a degradation of the partition quality with its parallel version [5].

Finally, Figure 10 illustrates the partition of the respiratory system mesh. The partition number assigned to each non-zero weighted bin of the fine grid is plotted, this represents the output of step 4 of Algorithm 1.

5. Concluding Remarks

In this paper, a new SFC-based parallel mesh partitioner has been developed and implemented. The analysis and optimization of the algorithm

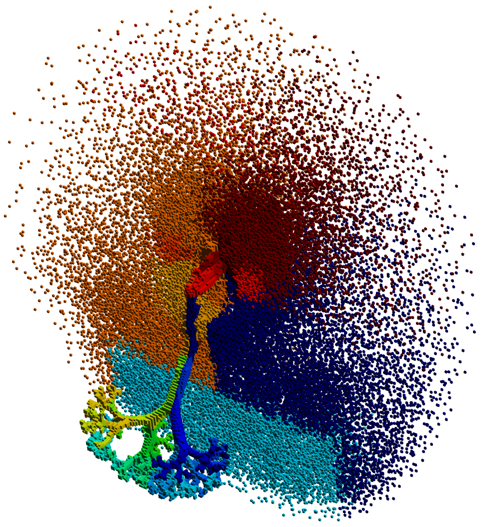


Figure 10: SFC-partition of the respiratory system mesh (17.7M) elements.

have been presented in detail. The final version is fully scalable and does not suffer any memory or computational bottlenecks, being able to partition a mesh of $28.9M$ cells in 5 cents of second using 4096 CPU-cores.

Moreover, our method maintains or even improves the quality of the partition at increasing the number of partitioning processes. We have compared it with Metis, both methods generate partitions for which the simulations performance is similar, differences obtained for different representative test cases are below 10%. Even when a much more comprehensive study is still necessary on this aspect, the SFC method seems to be competitive approach for parallel mesh partitioning with a clear potential for dynamic load balancing.

Acknowledgments

The research leading to these results has received funding from the European Union Horizon 2020 Programme (2014-2020) and the Brazilian Ministry of Science, Technology and Innovation through Rede Nacional de Pesquisa (RNP) under the HPC4E Project (grant agreement No. 689772). This work is also part of the PRAC "Simulations of Aircraft Engines" supported by the National Science Foundation. It has also been financially supported by the PRACE preparatory access projects funded in part by the EU Horizon 2020 research and innovation programme (2014-

2020) under grant agreement 653838. J.C. Cajas acknowledges the financial support of the ‘Consejo Nacional de Ciencia y Tecnología (CONACyT, México)’ grant number 231588 290790. Ricard Borrell and Daniel Mira acknowledge the Juan de la Cierva postdoctoral grants with codes IJCI-2014-21034 and IJCI-2015-26686, respectively.

References

[1] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[2] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *CoRR*, abs/0907.1375, 2009.

[3] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[4] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, January 1998.

[5] Zhi Shang. *Performance analysis of large scale parallel cfd computing based on codesaturne. Computer Physics Communications*, 184(2):381 – 386, 2013.

[6] G. Peano. *Sur une courbe, qui remplit toute une aire plane*, pages 71–75. Springer Vienna, Vienna, 1990.

[7] David Hilbert. *Über die stetige Abbildung einer Linie auf ein Flächenstück*, pages 1–2. Springer Berlin Heidelberg, Berlin, Heidelberg, 1970.

[8] Hans Sagan. *Space-Filling Curves*. Springer New York, New York, NY, 1994.

[9] J. Luitjens, M. Berzins, and T. Henderson. Parallel space-filling curve generation through sorting. *Concurrency Computation Practice and Experience*, 19(10):1387–1402, 2007.

[10] H. Calmet, A. M. Gambaruto, A. J. Bates, M. Vázquez, G. Houzeaux, and D.J. Doorly. Large-scale cfd simulations of the transitional and turbulent regime for the large human airways during rapid inhalation. *Computers in biology and medicine*, 69:166–180, 2016.

[11] H. Calmet, C. Kleinstreuer, G. Houzeaux, A.V. Kolanjiyil, O. Lehmkuhl, E. Olivares, and M. Vázquez. Subject-variability effects on micron particle deposition in human nasal cavities. *Journal of Aerosol Science*, 115(Supplement C):12 – 28, 2018.

[12] Simon Gövert, Daniel Mira, Jim B. W. Kok, Mariano Vázquez, and Guillaume Houzeaux. The effect of partial premixing and heat loss on the reacting flow field prediction of a swirl stabilized gas turbine model combustor. *Flow, Turbulence and Combustion*, Sep 2017.

[13] G. Houzeaux, M. Vázquez, R. Aubry, and J.M. Cela. A massively parallel fractional step solver for incompressible flows. *Journal of Computational Physics*, 228(17):6316 – 6332, 2009.

[14] G. Houzeaux, R. Aubry, and M. Vázquez. Extension of fractional step techniques for incompressible flows: The preconditioned orthomin(1) for the pressure schur complement. *Computers & Fluids*, 44(1):297–313, 2011.

[15] M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, E. Dering Burness, J. M. Cela, and M. Valero. Alya: Multiphysics engineering simulation towards exascale. *J. Comput. Sci.*, 14:15–27, 2016.