On the Use of Space Filling Curves for Parallel Anisotropic Mesh Adaptation

Frédéric Alauzet[1] and Adrien Loseille[2]

[1] INRIA Paris-Rocquencourt, Projet Gamma, Domaine de Voluceau, BP 105, 78153 Le Chesnay cedex, France
Frederic.Alauzet@inria.fr
[2] CFD Center, Dept. of Computational and Data Sciences, College of Science, MS 6A2, George Mason University, Fairfax, VA 22030-4444, USA
aloseill@gmu.edu

Abstract. Efficiently parallelizing a whole set of meshing tools, as required by an automated mesh adaptation loop, relies strongly on data localization to avoid memory access contention. In this regard, renumbering mesh items through a space filling curve (SFC), like Hilbert or Peano, is of great help and proved to be quite versatile. This paper briefly introduces the Hilbert SFC renumbering technique and illustrates its use with two different approaches to parallelization: an out-of-core method and a shared-memory multi-threaded algorithm.

Keywords: Space filling curves, shared-memory multi-threaded parallelization, out-of-core parallelization, anisotropic mesh adaptation, mesh partition.

1 Introduction

The efficient use of computer hardware is crucial to achieve high performance computing. No matter how clever an algorithm might be, it has to run efficiently on available computer hardwares. Each type of computer, from common PCs to fastest massively parallel machines, has its own shortcomings that must be accounted for when developing both algorithms and simulation codes. The wish to develop efficient parallel codes is thus driven by several requirements and practical considerations: the problem at hand that need to be solved, the required level of accuracy and the available computational power. The main motivation of this paper is to take advantage of today's ubiquitous multi-core computers in mesh adaptive computations.

Indeed, mesh adaptation is a method developed to reduce the complexity of numerical simulations by exploiting specifically the natural anisotropy of physical phenomena. Generally, it enables large complex problem to be solved in serial. However, the present hardware evolution suggests the parallelization of mesh adaptation platforms. Since 2004, first Moore's law corollary has plummeted from the 40% yearly increase in processor frequency, that it has enjoyed for the last 30 years, to a meager 10%. As for now, speed improvement

can only be achieved through the multiplication of processors, now called cores, sharing the same memory within a single chip.

Space filling curves (SFCs) are mathematical objects that enjoy nice proximity in space properties. These properties made then very useful in computer science and scientific computing. For instance, they have been used for data reordering [26, 31], dynamic partitioning [29], 2D parallel mesh generation [8] or all of these in the context of Cartesian adapted meshes [1].

In this paper, we present a straightforward parallelization of all softwares of a mesh adaptation platform where the pivot of the strategy is the **Hilbert space filling curve**. This strategy must be efficient in the context of highly anisotropic adapted meshes for complex real-life geometries. This platform is highly heterogeneous as it contains several software components that have different internal databases and that consider different numerical algorithms. It generally involves a flow solver, an adaptive mesh generator or an adaptive local remesher, an error estimate software and a solution interpolation (transfer) software. Two classes of parallelization are given.

The first one is an intrusive parallelization of the code using the **pthreads** paradigm for **shared-memory** cache-based parallel computers. One of the main assets of this strategy resides in a slight impact on the source code implementation and on the numerical algorithms. This strategy is applied to the flow solver and to the error estimate software. Parallelization is at the loop level and requires few modifications of the serial code. However, to be efficient this approach requires a subtle management of cache misses and cache-line overwrite to enable correct scaling factor for loop with indirect addressing. The key point is to utilize a Hilbert space filling curve based renumbering strategy to minimize them.

The second one is an **out-of-core** parallelization that considers the software as a black box. This approach is applied to a local remesher and the solution transfer software. It relies on the used of the Hilbert SFC to design a fast and efficient mesh partitioning. This partitioning method involves a correction phase to achieve connected partitions which is mandatory for anisotropic mesh adaptation. The mesh partitioner is coupled with an adequate management of the software on each partition. In this case, the code can be run in parallel on the same computer or in a distributed manner on an heterogeneous architecture.

As regards the meshing part, global mesh generation methods, such as Delaunay or Frontal approaches, are still hard to parallelize even if some solutions have already been proposed [7, 17, 20, 23]. Therefore, a local remeshing approach which is easier to parallelize thanks to its locality properties has been selected over a global mesher. The key point is how to adapt the partition borders [4, 10, 11, 18, 25, 27].

We illustrate with numerical examples that this methodology coupling anisotropic mesh adaptation, cache miss reduction and, out-of-core and pthreads parallelization can reduce the complexity of the problem by several

orders of magnitude providing a kind of "high performance computing" on nowadays multi-core personal computers.

This paper is outlined as follow. Section 2 recalls our mesh adaptation platform and Section 3 describes the test cases. Then, in Section 4, we present the Hilbert space filling curve based mesh renumbering. The shared-memory and the out-of-core parallelizations with their application to each stage of the mesh adaptation loop are introduced in Sections 5 and 6, respectively.

2 A Brief Overview of the Mesh Adaptation Platform

In the context of numerical simulation, the accuracy level of the solution depends on the current mesh used for its computation. And, for mesh adaptation, the size prescription, *i.e.,* the metric field, is provided by the current solution. This points out the non-linearity of the anisotropic mesh adaptation problem. Therefore, an iterative process needs to be set up in order to converge both the mesh and the solution, or equivalently the metric field and the solution. For stationary simulations, an adaptive computation is carried out *via* a mesh adaptation loop inside which an algorithmic convergence of the pair mesh-solution is sought. At each iteration, all components of the mesh adaptation loop are involved successively: the flow solver, the error estimate, the adaptive mesh generator and the solution interpolation stage. This procedure is repeated until the convergence of the mesh-solution pair is reached.

Our implementation of the mesh adaptation platform considers an independent dedicated software for each stage of the adaptation loop. As compared to the strategy where only one software contains all the stages of the mesh adaptation, we can highlight the following disadvantages and advantages. The main drawback is that between two stages, one software writes the data (e.g. the mesh and the fields) out-of-core and the next software reads them back and builds its internal database. This results in a larger part devoted to I/O as compared to the all-in-one approach. But, the CPU time for the I/O is generally negligible with respect to the global CPU time. The advantage of the proposed strategy is its flexibility. Each software can be developed independently with its own programming language and its own optimal internal database. For instance, the flow solver can keep a static database, the mesh generator can use specific topological data structures such as the elements neighbors, etc. Consequently, we may expect a higher efficiency in memory and in CPU time for each software. Moreover, each software is interchangeable with another one, only the I/O between the different softwares need to be compatible.

The mesh adaptation platform described in this paper involves the flow solver `Wolf` [5], `Metrix` for the error estimate [21], the local remesher `Mmg3d` [12] and `Interpol` for the solution transfer [6].

3 The Considered Test Cases

The efficiency of all the presented algorithms will be analyzed independently on the same list of test cases in their own dedicated sections. The efficiency is demonstrated thanks to CPU times and speed-ups, the speedup being the ratio between the CPU time in parallel and the CPU time in serial. The list of test cases is composed of uniform, adapted isotropic and anisotropic meshes for a wide range of number of tetrahedra varying from 40 000 to 50 000 000:

- uniform mesh: a transonic flow around the **M6 wing** [13] and **Rayleigh-Taylor instabilities** (IRT) [3]
- adapted isotropic mesh: a blast in a **city** [3]
- adapted highly anisotropic mesh: supersonic flows around Dassault-Aviation **supersonic business jet** (SSBJ) [21] and a NASA **spike** geometry, and a transonic flow around Dassault-Aviation **Falcon** business jet.

Meshes associated with these test cases are shown in Figure 1 and their characteristics are summarized in Table 1. Note that the SSBJ and the spike test cases involve very high size scale factor and highly anisotropic adapted meshes. For instance, for the SSBJ, the minimal mesh size on the aircraft is 2mm and has to be compared with a domain size of 2.5km.

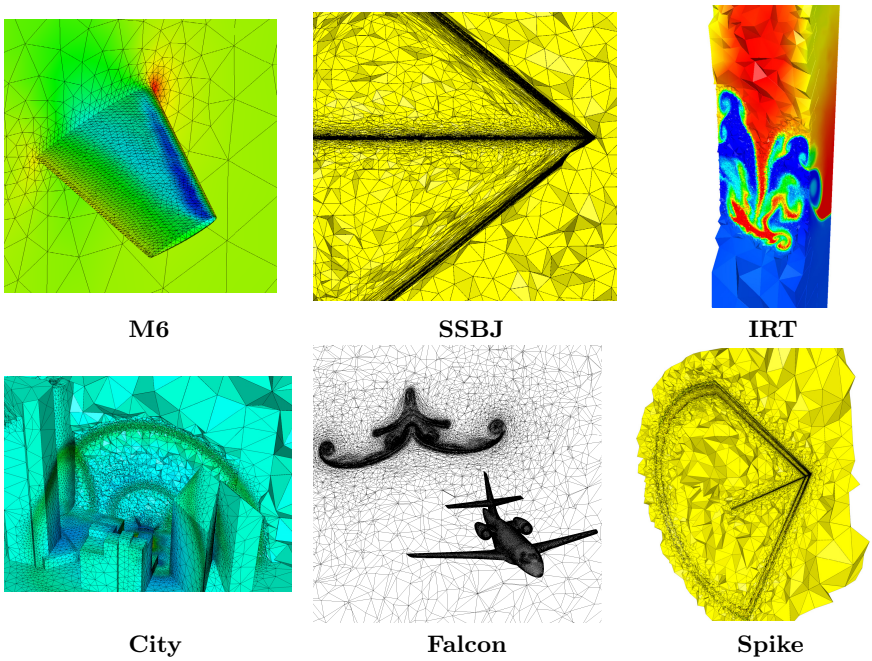

Fig. 1. View of the considered test cases meshes

Table 1. Characteristics of all test cases

Case	Mesh kind	# of vertices	# of tetrahedra	# of triangles
M6	uniform	7 815	37 922	5 848
IRT	uniform	74 507	400 033	32 286
City	adapted isotropic	677 278	3 974 570	67 408
Falcon	adapted anisotropic	2 025 231	11 860 697	164 872
SSBJ	adapted anisotropic	4 249 176	25 076 962	334 348
Spike	adapted anisotropic	8 069 621	48 045 800	182 286

All the runs have been done on a 2.8 GHz dual-chip Intel Core 2 Quad (eight-processor) Mac Xserve with 32 GB of RAM.

4 The Hilbert Space Filling Curve

The notion of space filling curves (SFCs) has emerged with the development of the concept of the Cantor set [9]. Explicit descriptions of such curves were proposed by Peano [28] and Hilbert [15]. SFCs are, in fact, fractal objects [22]. A complete overview is given in [30]. A SFC is a continuous function that, roughly speaking, maps a higher dimensional space, e.g. $\mathbb{R}^2$ or $\mathbb{R}^3$, into a one-dimensional space:

$$h : \{1, \ldots, n\}^d \mapsto \{1, \ldots, n^d\}\,.$$

These curves enjoy strong local properties making them suitable for many applications in computer sciences and scientific computing. The **Hilbert SFC** is a continuous curve that fills an entire square or cube. For the Hilbert SFC, we have [26]:

$$|h(i) - h(j)| < \sqrt{6}\,|i - j|^{\frac{1}{2}} \quad \text{for} \quad i, j \in \mathbb{N}\,.$$

The Hilbert SFC for the square or the cube is generated by recursion as depicted in Figure 2. Its discrete representation depends on the level of recursion. In our use of the Hilbert SFC, the curve is not explicitly constructed but its definition is used to calculate an index associated with each mesh entity by means of the recursive algorithm. In other words, in three dimensions, the SFC provides an ordered numeration of a virtual Cartesian grid of size 2^{3p} where p is the depth of recursion in which our computational domain is embedded. The index of an entity is then obtained by finding in which cube the entity stands. In the following, we will present how these indices are used to renumber a mesh or to partition it.

4.1 Mesh Entities Renumbering

The Hilbert SFC can be used to map mesh geometric entities, such as vertices, edges, triangles and tetrahedra, into a one dimensional interval. In numerical

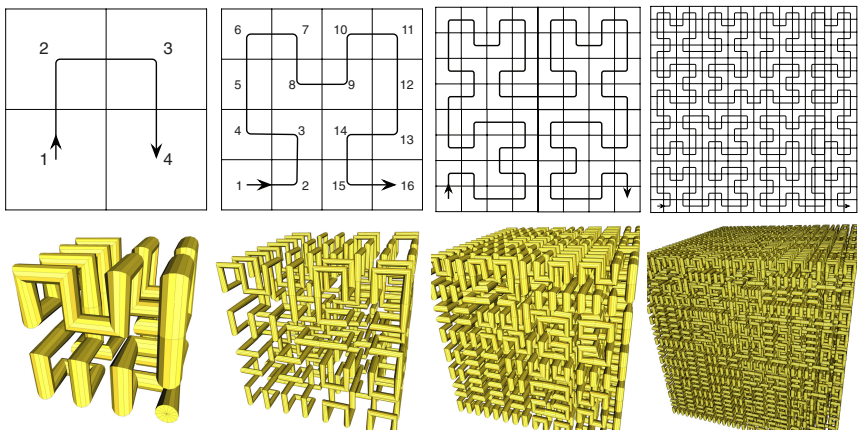

Fig. 2. Representation of the Hilbert curve in 2D and 3D after several recursions. Top, the 2D Hilbert SFC of the square after recursions 1, 2, 3 and 4. Bottom, the 3D Hilbert SFC of the cube after recursions 2, 3, 4 and 5.

Table 2. CPU time in seconds for sorting the vertices of the test cases meshes.

Case	M6	IRT	City	Falcon	SSBJ	Spike
SFC construction CPU in sec	0.004	0.038	0.338	1.002	1.894	3.632
Quicksort CPU in sec	0.001	0.010	0.104	0.974	0.668	1.518
Global CPU in sec	0.005	0.048	0.442	2.076	2.562	5.150

applications, it can be viewed as a mapping from the computational domain Ω onto the memory of a computer. The local property of the Hilbert SFC implies that entities which are neighbors on the memory 1D interval are also neighbors in the domain Ω. But, the reverse may not be true. Neighbors in the volume may be separated through the mapping. This approach has been applied to reorder efficiently Cartesian grids [1, 29] and its used for unstructured tetrahedral meshes has been indicated in [31]. Note that a large varieties of renumbering strategies exist which are commonly used in scientific computing [19] or in meshing [32]. Here, we apply and analyze such renumbering to unstructured, isotropic or anisotropic, adapted meshes.

First, the index is computed for each entity, this operation has a linear complexity. Then, the mesh entities have to be reordered to obtain the renumbering. This sort is done with standard C-library sorting routine such as `quicksort`, hence the $O(N \log(N))$ complexity of our method. Table 2 sums-up the CPU time for sorting all the test cases meshes on the Mac Xserve. Figure 3 illustrates an unstructured mesh of a scramjet (left) and how the vertices have been reordered in memory by following the line (right), this is the Hilbert SFC.

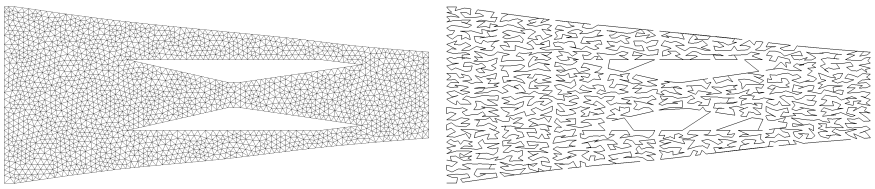

Fig. 3. Left, unstructured mesh of a scramjet. Right, the Hilbert SFC (red line) associated with the vertices. It represents how the vertices are ordered in memory.

Table 3. Number of CPU cycles required for typical operations on Intel Core 2.

Operations	mult/add	div	sqrt	cache miss L1	cache miss L2	mutex	condwait
Cycles	1	10	10	13	276	6 240	12 480

Renumbering strategies have a significant impact on the efficiency of a code. This is even more crucial for numerical methods on unstructured meshes. We can cite the following impact:

- reducing the number of cache misses during indirect addressing loops
- improving matrix preconditioning for linear system resolutions
- reducing cache-line overwrites during indirect addressing loops which is fundamental for shared memory parallelism, see Section 5.1
- may provide implicit hashing of the data.

Let us focus on the first item : cache misses that are due to indirect addressing. They occur when data are required for a computation and those data do not lie within the same cache line. For instance, such situation is frequent while performing a loop through the tetrahedra list and requesting for vertices data. It is worth mentioning that the cost of a cache miss is far more important than typical operations used in numerical applications, see Table 3.

To effectively reduce cache misses, all the mesh entities must be reordered and not, for instance, only the vertices. In our approach, the Hilbert SFC based renumbering is used to sort the vertices as their proximity depends on their position in space even for anisotropic meshes. As regards the topological entities, e.g. the edges, the triangles and the tetrahedra, the Hilbert SFC based renumbering can be applied by using their center of gravity. However, as they are topological entities, we prefer to consider a topological renumbering strategy based on vertex balls which also provides an implicit hashing of the entities. Note that similar wave renumbering strategies are described in [19]. This strategy reduces by 90% the number of cache misses[1] of the flow solver (Section 5.2) as compared to an unsorted mesh generated with a Delaunay-based algorithm.

[1] This statistic has been obtained with the Apple Shark code profiler.

For the test cases of Sections 5 and 6, we obtain in serial a speed-up up to 2.68, *i.e.,* up to almost three time faster, when all the entities are reordered as compared to the unsorted mesh. More precisely, speed-ups of 1.06, 1.62, 2.54 and 2.68 are obtained for the M6, IRT, City and Falcon test cases, respectively.

5 Exploiting Space Filling Curves for Efficient Shared-Memory Multi-threaded Parallelization

5.1 A Shared Memory Multi-threaded Parallelization

Our approach is based on posix standard threads (known as *pthreads*) thus taking advantage of *multi-core* chips and *shared memory* architectures supported by most platforms. Such an approach presents two main advantages: data do not need to be explicitly partitioned as with MPI based parallelism and its implementation requires only slight modifications to a previously developed serial code.

Symmetric parallelization of loops. In this case, a loop performing the same operation on each entry of a table is split into many sub-loops. Each sub-loop will perform the same operation (hence the name symmetric parallelism) on equally-sized portions of the main table and will be concurrently executed. It is the scheduler job to make sure that two threads do not write simultaneously on the same memory location. To allow for a fine load balancing, we split the main table into a number of blocks equal to 16 times the number of available processors.

Indirect memory access loops. Using meshes and matrices in scientific computing leads inevitably to complex algorithms where indirect memory accesses are needed. For example, accessing a vertex structure through a tetrahedron leads to the following instruction:

```
TetTab[i]->VerTab[j];
```

Such a memory access is very common in C, the compiler will first look for tetrahedron i, then vertex j, thus accessing the data indirectly. In this case, after splitting the main tetrahedra table into sub-blocks, tetrahedra from different blocks may point to the same vertices. If two such blocks were to run concurrently, memory write conflict would arise. To deal with this issue, an asynchronous parallelization is considered instead of a classic gather/scatter technique usually developed on distributed memory architectures, *i.e.,* each thread writes in its own working array to avoid memory conflict and then the data are merged. Indeed, this asynchronous parallelization has the following benefits: there is no memory overhead and for all the test cases on 8 cores this method was 20 to 30 % faster than the gather/scatter method. The main difficulty is then to minimize the synchronization costs that are expensive in CPU cycles, cf. Table 3. To this end, the scheduler will carefully

choose the set of concurrently running blocks so that they share no common tetrahedra and no common vertices. In case no compatible blocks are to be found, some threads may be left idling, thus reducing the degree of parallelization. This method can even lead to a serial execution in the case of a total incompatibility between blocks.

The collision probability of any two blocks sets the mesh inherent parallelization factor. Meshes generated by advancing front or Delaunay techniques feature a very low factor (2 or 3 at best), while octree methods far much better. Renumbering the mesh elements and vertices as described in Section 4.1 dramatically enhances the inherent parallelism degree (by orders of magnitude). Applying such renumbering is *de facto* mandatory when dealing with indirect memory access loops. The block collision statistics for the test cases of Section 3 on 8 processors without and with the renumbering strategy of Section 4.1 are reported in Table 4.

As regards the scheduler cost, the operation of locking/unlocking a thread needs one mutex and one condwait, see Table 3. As these two operations are needed when launching and when stopping a thread, the resulting cost is approximatively 37 000 CPU cycles which is very expensive as compared to standard floating point operations or cache misses timings.

This approach has been implemented in the *LP2* library [24]. The purpose of this library is to provide programmers of solvers or automated meshers in the field of scientific computing with an easy, fast and transparent way to parallelize their codes. Thus, we can implement directly in parallel.

A sketch of the modifications of a serial code parallelized with the LP2 is given in Figure 4. Left, the dependencies of the tetrahedra array with respect to the vertices array are set. Right, the modification of the routine `Solve` which processes a loop on tetrahedra. This routine is called in parallel with two additional parameters `iBeg` and `iEnd` that are managed by the LP2. This illustrates the slight modifications that occur for the serial code. For instance,

Table 4. Collision percentage between blocks of entities when the list is not sorted or sorted.

Cases		Edges List		Tetrahedra List		Triangles List	
		Avg	Max	Avg	Max	Avg	Max
M6	no sort	2.96%	6.10%	7.25%	9.83%	0.35%	0.61%
	sort	0.94%	1.57%	0.95%	1.61%	0.37%	0.61%
IRT	no sort	24.06%	35.94%	45.21%	55.36%	1.56%	2.71%
	sort	1.00%	2.03%	1.00%	1.79%	0.42%	0.92%
City	no sort	73.90%	98.64%	93.29%	99.51%	1.21%	2.78%
	sort	1.09%	3.95%	1.10%	3.60%	0.25%	0.97%
Falcon	no sort	99.55%	100.00%	99.65%	100.00%	0.15%	1.02%
	sort	1.81%	4.63%	1.81%	4.24%	0.19%	0.59%

```
BeginDependency(Tetrahedra,Vertices);
for (iTet=1; iTet<=NbrTet; ++iTet) {
  for (j=0; j<4; ++j) {
    AddDependency( iTet, Tet[iTet].Ver[j] );
  }
}
EndDependency(Tetrahedra,Vertices);
```

```
Solve(Tetrahedra,iBeg,iEnd) {
  for (iTet=iBeg; iTet<=iEnd; ++iTet) {
    // .... same as serial
  }
}
```

Fig. 4. Modification of the serial code for shared-memory parallelization.

for our flow solver (see Section 5.2) most of (98.5 %) the explicit resolution part has been parallelized in less than one day.

5.2 Parallelizing the Flow Solver

The parallelized flow solver is a vertex-centered finite volume scheme on unstructured tetrahedral meshes solving the compressible Euler equations. To give a brief overview, the HLLC approximate Riemann solver is used to compute the numerical flux. The high-order scheme is derived according to a MUSCL type method using downstream and upstream tetrahedra. A high-order scheme is deduced by using upwind and downwind gradients leading to a numerical dissipation of 4^{th} order. To guarantee the TVD property of the scheme, a generalization of the Superbee limiter with three entries is considered. The time integration is an explicit algorithm using a 5-stage, 2-order strong-stability-preserving Runge-Kutta scheme. We refer to [5] for a complete description.

A shared-memory parallelization of the finite volume code has been implemented with the pthreads paradigm described in Section 5.1. It uses the entities renumbering strategy proposed in Section 4.1. It took only 1 day to parallelize most of the resolution part with less than 2% of the resolution remaining in serial. More precisely, 6 main loops of the resolution have been parallelized:

- the time step evaluation which is a loop on the vertices without dependencies
- the boundary gradient[2] evaluation which is a loop on the tetrahedra connected to the boundary
- the boundary conditions which is a loop on the boundary triangles
- the flux computation which is a loop on the edges
- the source term computation which is a loop on the vertices without dependencies
- the update (advance) in time of the solution which is a loop on the vertices without dependencies.

The speed-ups, as compared to the serial version, obtained for each test case from 2 to 8 processors are summarized in Table 5. These speed-ups

[2] For this numerical scheme, the element gradient used for the upwinding can be computed on the fly during the flux evaluation.

Table 5. Speed-ups of the flow solver as compared to the serial version for all the test cases from 2 to 8 processors.

Cases		M6	IRT	City	Falcon	SSBJ	Spike
Speed-up	1 Proc	1.000	1.000	1.000	1.000	1.000	1.000
	2 Proc	1.814	1.959	1.956	1.961	1.969	1.975
	4 Proc	3.265	3.748	3.866	3.840	3.750	3.880
	8 Proc	5.059	6.765	7.231	6.861	7.031	7.223

contains the time of I/Os which is negligible for the flow solver. These results are very satisfactory for the largest cases for which an enjoyable speed-up around 7 is attained on 8 processors. The slight degradation observed between 4 and 8 processors is in part due to a limitation of the current hardware of the Intel Core 2 Quad chip. However, the speed-ups are lower for the smallest case, the M6 wing with only 7 815 vertices. This small test case with a light amount of work points out the over-cost of the scheduler for pthreads handling which is a weakness of the proposed approach. More precisely, we recall that launching and stopping a thread cost approximatively 37 000 CPU cycles. As the parallelization is at the loop level, obtaining correct speed-ups requires that the cost of handling each thread must remain negligible compared to the amount of work they are processing. But, this is not the case for the M6 test case. Indeed, if we analyze the time step loop, on 8 processors, this loop deals with 977 vertices each requiring 100 CPU cycles. Consequently, the management of the thread costs the equivalent of 38% of the total cost of the operations.

The speed-up of a parallel code is one criteria, but it is also of utmost importance to specify the real speed of a code. As regards flow solvers, it is difficult to compare their speed as the total time of resolution depends on several parameters such as the RK scheme (for explicit solver), the mesh quality that conditions the time step, the chosen CFL, etc. Therefore, for the same computation, high variations could appear. To specify the relative speed of the solver, we choose to provide the CPU time per vertex per iteration:

$$speed = \frac{CPU\,time}{\#\,of\,vertices \times \#\,of\,iterations}\,.$$

For the test cases on the Mac Xserve, the serial speed of the solver varies between 3.4 and 3.87 microseconds (μs) while a speed between 0.47 μs (for the spike) and 0.76 μs (for the M6) is obtained on 8 processors. To give an idea, a speed of 0.5 μs is equivalent to performing one iteration in half a second for a one million vertices mesh. In conclusion, the faster a routine, the harder it is to parallelize it with a satisfactory speed-up.

The flow solver has also been run on a 128 processors SGI Altix computer at the Barcelona Supercomputing Center to make a preliminary analysis on using a large number of processors. Such a machine uses a ccNUMA architecture suffering from high memory latency. In this context, minimizing main

Table 6. Speed-ups of the flow solver on a 128 processors SGI Altix computer.

# of proc	1	2	4	8	16	32	64	100
Speed-up	1.000	1.954	3.235	6.078	9.815	17.258	26.649	36.539

Table 7. Speed-ups of the error estimate code as compared to the serial version for all the test cases from 2 to 8 processors. Upper part, speed-up with respect to the whole CPU time. Lower part, speed-up of the parallelized part.

Cases		M6	IRT	City	Falcon	SSBJ	Spike
Total CPU in sec.	1 Proc	0.226	3.365	29.315	101.14	252.76	433.91
Speed-up	2 Proc	1.44	1.58	1.47	1.42	1.53	1.33
	4 Proc	1.77	2.19	1.76	1.71	2.10	2.01
	8 Proc	1.85	2.69	2.19	2.05	2.66	2.61
Gradation CPU in sec.	1 Proc	0.131	2.417	18.414	59.79	186.47	299.45
Speed-up	2 Proc	1.87	1.89	1.88	1.90	1.92	1.89
	4 Proc	3.19	3.55	3.44	3.53	3.62	3.47
	8 Proc	3.85	6.00	5.84	6.12	6.59	5.90

memory accesses through Hilbert SFC based renumbering, is all the more important. The considered test case is a large anisotropic adapted mesh containing almost 400 million tetrahedra. The speed-ups from 1 to 100 processors are given in Table 6. These first results are very encouraging and we are thus confident for the obtention of good speed-ups up to 128 processors in a near future.

5.3 Parallelizing the Error Estimate

The error estimate software will use the same parallelization methodology as the flow solver. This stage of the mesh adaptation platform is very inexpensive. In our experience, its cost is between 1 and 2 % of the whole CPU time. For instance, computing the metric with the estimate of [21] coupled with the mesh gradation algorithm of [2] for the spike test cases cost 7 minutes in serial (I/O included).

However, if the CPU time is carefully analyzed, we observed that the error estimate represents 3% of the CPU, the mesh gradation 69%, and the I/O plus building the database 27%. Consequently, we have only parallelized the mesh gradation algorithm as the error estimate computation is extremely inexpensive[3]. The speed-ups obtained for the whole CPU time (upper part) and for the mesh gradation phase (lower part) are given in Table 7. Concerning, the mesh gradation phase nice speed-ups are obtained for all the test cases. The impact on the whole CPU time is a speed-up between 2 and 2.6 on 8 processors.

[3] Its parallelization is not expected before running on 100 processors.

6 Exploiting Space Filling Curves for Efficient Out-of-Core Parallelization

6.1 A Fast Mesh Partitioning Algorithm

Mesh partitioning is one of the crucial task for distributed-memory parallelism. A critical issue is the minimization of the inter-processors communications, *i.e.,* the interfaces between partitions, while keeping well-balanced partitions. Indeed, these communications represent the main over-head of the parallel code. This problem is generally solved using graph connectivity or geometric properties to represent the topology of the mesh. These methods have now attained a good level of maturity, see ParMETIS [16].

However, in our context of anisotropic parallel mesh adaptation, the goal is different. The mesh partitioning is considered for purely distributed tasks without any communication, but for meshing or local remeshing purposes each partition must be connected. This requirement is not taken into account by classical partitioner. Therefore, we aim at designing the fastest possible algorithm that provides well-balanced connected partitions. We choose a Hilbert SFC based mesh-partitioning strategy applied to unstructured meshes, such as the ones proposed in [1, 29] which have been applied to Cartesian grids. This strategy is then improved to handle highly anisotropic unstructured meshes.

A Hilbert SFC based mesh partitioning. As the Hilbert SFC provides a one-dimensional ordering of the considered three-dimensional mesh, partitioning the mesh is simply equivalent to partitioning a segment. Given a 3D input mesh, the algorithm to create k subdomains is threefold:

1. get the index on the Hilbert SFC of each vertex/tetrahedron gravity center as stated in Section 4.1. An implicit Hilbert SFC is built
2. sort the vertices/tetrahedra list to get a new ordered list of elements
3. subdivide uniformly the sorted list of elements, *i.e.,* the Hilbert SFC, into k sublists. The sublists are the subdomains.

This algorithm is extremely fast and consumes little memory. The partitioning is equivalent to a renumbering plus a sort.

If this algorithm works perfectly for structured grids or uniform meshes, it needs, as standard partitioner, to be corrected when dealing with anisotropic meshes to obtain connected partitions. Indeed, two consecutive elements on the Hilbert SFC are close in the domain but they may be linked by a single vertex or edge, see Figure 5, and not by a face. In this case, the resulting subdomains are non-connected. A correction phase is then mandatory to ensure that each partition is connected. This is done in two steps. The first step consists in detecting each connected component of each subdomain thanks to a coloring algorithm [14]. A first non-colored tetrahedron initializes a list. Then, the first tetrahedron of the list is colored with the connected component index and removed from the list, and its non-colored neighbors (tetrahedra adjacent by a face) are added to the list. The process is repeated

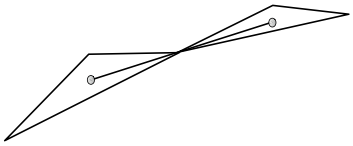

Fig. 5. Case where two consecutive elements in Hilbert SFC based numbering are linked by a vertex leading to the creation of a non-connected partition.

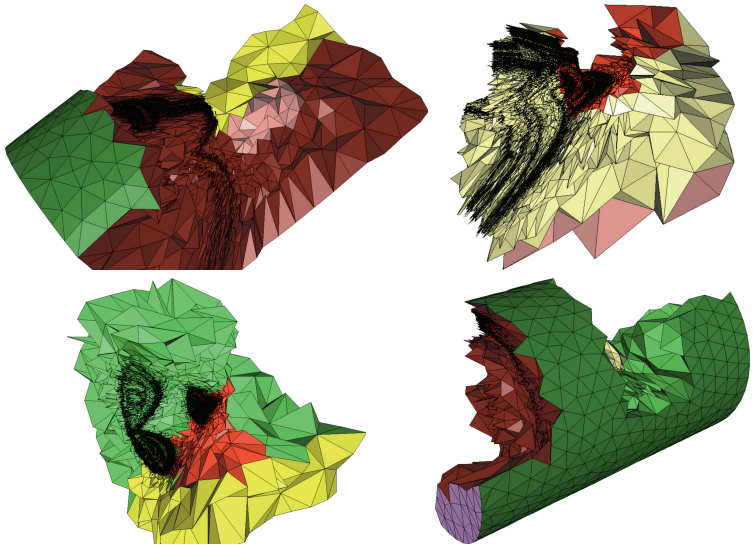

Fig. 6. Some partitions of a complex anisotropic adapted mesh.

until the list is empty. This algorithm requires as supplementary data structure the neighbors tetrahedra table. If each subdomain is composed of only one connected component then no correction is needed. Otherwise, in the second step, elements are reattributed to neighboring partitions in order to ensure that all partitions are connected. Figures 6 shows some partitions of a complex anisotropic adapted mesh.

Domain gathering. Domain gathering is done on the fly by reading each partition one after another and by updating a global table of vertices lying on interfaces. Consequently, only all the interface meshes are loaded in memory and no partition elements need to be kept in memory. The partition meshes are read and written element by element on the fly. Therefore, the whole mesh is never allocated. It results that this operation can gather a large number of partitions while requiring little memory. The key point is the algorithm to recover the one-to-one mapping between two interfaces. It consists in using a wave front approach based on the topology of the interface meshes to map

Table 8. Number of tetrahedra per partition for the spike test case.

1	2	3	4	5	6	7	8
6 005 604	6 006 014	6 006 041	6 005 838	6 005 737	6 005 026	6 005 757	6 005 783

Table 9. Statistics for partitioning into 8 blocks. CPU times is in seconds and memory is in MB.

Case	I/O CPU	Partition CPU	Total CPU	Max memory	Size variation
M6	0.10	0.09	0.191	5	7.68%
IRT	0.86	1.25	2.114	47	3.17%
City	10.89	12.05	22.94	469	0.04%
Falcon	50.47	41.68	92.15	1396	0.22%
SSBJ	98.43	89.35	187.78	2947	0.04%
Spike	120.22	168.35	288.57	5608	0.02%

one after another each vertex [19]. This algorithm is purely topologic. It is thus exact and not subjected to any floating point precision.

Numerical experiments. Let us analyze the spike test case for 8 partitions. The time to create the 8 connected partitions and to write the corresponding meshes is about 288s. The maximal memory allocated in this case is 5.6GB. The partitions are well-balanced, indeed the size in balance between partitions is no more than 0.02% as summarized by Table 8. We now give the detailed CPU time for each phase of the domain decomposition algorithm:

- Reading input data: 69s
- Create an initial Hilbert SFC based partition: 91s
- Create neighboring structure: 24s
- Correct partitions: 52.2s
- Writing output data: 50s.

As regards the partitions gathering, the algorithm consumes very low amount of memory as only the interfaces of the meshes are stored. For this example, the complete gathering step requires 42s and the maximal allocated memory is 156MB.

Results obtained for all the test cases are summarized in Table 9. All the CPU times are in seconds and the maximum allocated memory is in MB. An excellent load balancing is obtained for all the cases, except for the two smallest ones.

6.2 Parallelizing the Local Adaptive Remesher

The strategy to parallelize the meshing part is a out-of-core parallelization that uses the adaptive mesh generator as a black box. This method is completely distributed without any communications, thus it can be applied to

shared-memory or heterogeneous architectures. The advantage of this method is its simplicity and its flexibility, as any adaptive mesh generators can be used. Here, the local remesher of [12] is utilized. The drawback is an I/O and build database over-cost.

In the context of parallel anisotropic mesh adaptation, the objectives are different from the solver ones. Apart from the traditional scaling of the parallel algorithm, the interest is in the possibility of improving the serial remeshing algorithm by:

- reducing the cache misses for efficiency
- reducing the scale factors for robustness purposes
- improving the local quadratic search algorithms that could occur in Delaunay-based mesh generators.

Previous points are necessary to foresee the generation of highly anisotropic meshes with dozens of million of elements.

The main difference between different parallelizations of local remeshing algorithms resides in how the partitions interfaces are handled. In some parallel adaptation implementations connectivity changes are performed in the interior of the partition and migration is used to make border regions interior [4, 10, 11, 18, 25]. In [27], tetrahedron connectivity changes are performed on purely-local and border tetrahedra in separate operations without migration. This difficulty generally comes from the use of a all-in-one method. Here, as each software is independent and a local remeshing strategy is employed, the necessity of remeshing partitions interfaces is not strictly necessary. Indeed, the parallel remeshing algorithm can be thought as an iterative procedure. The only constraint is then to ensure that from one step to another the boundaries of interfaces change to adapt them. Consequently, reducing the size of the interfaces is no more the most critical issue.

On the contrary, we prefer to generate well-balanced partitions for anisotropic remeshing. Note that in the context of remeshing, well-balanced partitions does not mean having the same number of vertices or elements. Indeed, the estimate time CPU of a mesh generator depends more of the current operation: insertions, collapses or optimization. The CPU time of these operations is not always linear with the number of vertices of the input mesh. We did not propose yet any improvements to deal with these non linearities.

The proposed method is a divide and conquer strategy using the mesh partitioner of Section 6.1 which is given by the following iterative procedure:

1. Split the initial mesh: each partition is renumbered using Hilbert SFC based strategy
2. Adapt each partition in parallel with only vertex insertion, collapses, swaps
3. Merge new adapted partitions and split the new adapted mesh with random interfaces: each partition is renumbered using Hilbert SFC based strategy
4. Optimize each partition in parallel with swaps and vertices movement

5. Merge new adapted partitions
6. return to 1.

Generally, two iterations are performed. Using this technique makes the anisotropic remeshing time satisfactory as compared to the flow solver CPU time in the adaptive loop. However, it is very difficult to quantify the CPU time of the meshing part as it depends on a large number of parameters, for instance, do we coarse the mesh, optimize it or insert a lot of vertices, etc.

Our experience on a large number of simulations with dozens million of elements shows that managing efficiently the cache misses leads to acceleration between 2 and 10 in serial. As regards the out-of-core parallelization, after adequate renumbering, satisfactory speed-ups are obtained. The speed-ups for the City and SSBJ test case are given in Table 10. These speed-ups are coherent as the partitions are balanced with respect to their size and do not take into account the future work of the mesh generator. Sometimes, the remeshing of one of the partitions is twice more costly than the remeshing of any of the other ones. This degrades considerably the speed-up. It can be improved by increasing the number of partitions for a fixed number of processors. For instance, for the SSBJ test case on 8 processors and 32 partitions the speed-up increases to 6.

Overall, this strategy combining cache miss management and out-of-core parallelization can provide speed-ups up to 40 on 8 processors (the speed-up may even be greater than the number of processors) as compared to the original code alone. But large fluctuations in the obtained speed-ups are observed and are highly dependent on the considered case.

Table 10. Speed-ups of the local remesher as compared to the serial version.

Cases	1 Proc	2 Proc	4 Proc	8 Proc
City	1.00	1.56	2.43	2.61
SSBJ	1.00	1.36	2.37	4.50

6.3 Parallelizing the Solution Interpolation

After the generation of the new adapted mesh, the solution interpolation stage consists in transferring the previous solution fields obtained on the background mesh onto the new mesh. This stage is also very fast if cache misses are carefully managed thanks to the Hilbert SFC based renumbering. For instance, the solution fields of the spike test case are interpolated in 107 seconds. Detailed CPU times are 47s for the I/Os and sort, 15s for building the database and 45s for the interpolation method. We notice that the I/Os and building the database are taking more than 50% of the CPU time. Thus, the expected speed-ups for a parallel version are limited.

The algorithm to efficiently parallelize the interpolation method of [6] with the pthreads paradigm is equivalent to partition the domain. But, partitioning is slower than the interpolation. This way has thus not been chosen. Nevertheless, this stage can be parallelized in the context of mesh adaptation with a distributed out-of-core strategy. The clue point is that the new mesh has already been partitioned and renumbered for mesh adaptation. Therefore, before merging all partitions, the interpolation can be applied in parallel to each new adapted partition separately. The over-cost of partitioning and gathering the mesh is already included in the mesh adaptation loop. Otherwise, it will be faster to run in serial. However, the expected speed-ups are limited by the I/Os and building the database associated with the background mesh which is not partitioned.

This method has been applied to all the test cases. Each pair mesh-solution of Section 3 are interpolated on a new (different) mesh of almost the same size, *i.e.,* a size variation of less than 10%. The CPU time in seconds for each case in serial is given Table 11. In parallel, no gain is observed for the smallest cases: the M6 and the IRT. For larger test cases, speed-ups between 1.3 and 2 are obtained on 2 processors and they are moderately higher with 4 processors. Unfortunately, CPU time degrades for 8 processors. This is mainly due to the fact that I/Os degrade because eight process run concurrently on the same computer while requesting access to the disk at the same time. Fortunately, this effect diminishes (or cancels) during an adaptive computations as the interpolation on each partition immediately follows the mesh adaptation. Indeed, the mesh adaptation of each partition finishes at different time.

Table 11. CPU times in seconds to interpolate the solution fields in serial.

Cases	M6	IRT	City	Falcon	SSBJ	Spike
CPU time in sec.	0.081	0.88	14.69	48.45	56.51	107.48

7 Conclusion

In this paper, we have presented a first step in the parallelization of the mesh adaptation platform. It has been demonstrated that the use of the Hilbert SFC authorizes a cheap and easy parallelization of each stage of the mesh adaptation platform. The parallelization can be shared-memory multi-threaded or out-of-core. The Hilbert SFC is the core of the renumbering strategy and the mesh partitioner. It also importantly reduces the code cache misses leading to important gain in CPU time. As already mentioned in [31], many code options that are essential for realistic simulations are not easy to parallelize on distributed memory architecture, notably local remeshing, repeated h-refinement, some preconditioners, etc. We think that this strategy can provide an answer even if it is not the optimal one.

The weaknesses of this approach are I/Os and build database over-cost, especially on the fastest stages as the error estimate or the interpolation. The I/Os time is incompressible, it depends on the hardware. Indeed, solutions exist, like fast RAIDs. Improving the building database part require to parallelize complex algorithm such as hash table. The other point of paramount importance for the proposed shared-memory multi-threaded parallelization is the cost of locking/unlocking thread which can be prohibitive for a loop with a little amount of work.

In spite of that the proposed parallel adaptive methodology provides a kind of "high performance computing" on nowadays multi-core personal computers by reducing the complexity of the problem by several orders of magnitude.

Several improvements of the proposed approach are still in progress.

Regarding the shared-memory parallelization, the scheduler has to be parallelized to keep its cost constant whatever the number of processors and, at the loop level, some algorithm can be enhanced to improve their scalability. The out-of-core parallelization can be enhanced by parallelizing the mesh partitioner and by deriving a fine load-balancing that takes into account the future work on each partition of the local remesher thanks to the metric specification. And finally, for fast codes, the error estimate and the interpolation, we will have to tackle the problem of parallelization of database construction which involves hash tables.

References

1. Aftosmis, M., Berger, M., Murman, S.: Applications of space-filling curves to cartesian methods for CFD. AIAA Paper 2004-1232 (2004)
2. Alauzet, F.: Size gradation control of anisotropic meshes. Finite Elem. Anal. Des. (2009) doi:10.1016/j.finel.2009.06.028
3. Alauzet, F., Frey, P., George, P.-L., Mohammadi, B.: 3D transient fixed point mesh adaptation for time-dependent problems: Application to CFD simulations. J. Comp. Phys. 222, 592–623 (2007)
4. Alauzet, F., Li, X., Seol, E.S., Shephard, M.: Parallel anisotropic 3D mesh adaptation by mesh modification. Eng. w. Comp. 21(3), 247–258 (2006)
5. Alauzet, F., Loseille, A.: High order sonic boom modeling by adaptive methods. RR-6845, INRIA (February 2009)
6. Alauzet, F., Mehrenberger, M.: P1-conservative solution interpolation on unstructured triangular meshes. RR-6804, INRIA (January 2009)
7. Alleaume, A., Francez, L., Loriot, M., Maman, N.: Large out-of-core tetrahedral meshing. In: Proceedings of the 16th International Meshing Roundtable, pp. 461–476 (2007)
8. Behrens, J., Zimmermann, J.: Parallelizing an unstructured grid generator with a space-filling curve approach. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 815–823. Springer, Heidelberg (2000)
9. Cantor, G.: über unendliche, lineare punktmannigfaltigkeiten 5. Mathematische Annalen 21, 545–586 (1883)

10. Cavallo, P., Sinha, N., Feldman, G.: Parallel unstructured mesh adaptation method for moving body applications. AIAA Journal 43(9), 1937–1945 (2005)
11. DeCougny, H.L., Shephard, M.: Parallel refinement and coarsening of tetrahedral meshes. Journal for Numerical Methods in Engineering 46(7), 1101–1125 (1999)
12. Dobrzynski, C., Frey, P.J.: Anisotropic Delaunay mesh adaptation for unsteady simulations. In: Proceedings of the 17th International Meshing Roundtable, pp. 177–194. Springer, Heidelberg (2008)
13. Frey, P.J., Alauzet, F.: Anisotropic mesh adaptation for CFD computations. Comput. Methods Appl. Mech. Engrg. 194(48-49), 5068–5082 (2005)
14. Frey, P., George, P.-L.: Mesh generation. Application to finite elements, 2nd edn. ISTE Ltd and John Wiley & Sons, Chichester (2008)
15. Hilbert, D.: über die stetige abbildung einer linie auf ein flächenstück. Mathematische Annalen 38, 459–460 (1891)
16. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (1998)
17. Larwood, B.G., Weatherill, N.P., Hassan, O., Morgan, K.: Domain decomposition approach for parallel unstructured mesh generation. Int. J. Numer. Meth. Engng. 58(2), 177–188 (2003)
18. Lepage, C., Habashi, W.: Parallel unstructured mesh adaptation on distributed-memory systems. AIAA Paper 2004-2532 (2004)
19. Löhner, R.: Applied CFD techniques. An introduction based on finite element methods. John Wiley & Sons, Ltd., New York (2001)
20. Löhner, R.: A parallel advancing front grid generation scheme. Int. J. Numer. Meth. Engng 51, 663–678 (2001)
21. Loseille, A., Dervieux, A., Frey, P., Alauzet, F.: Achievement of global second-order mesh convergence for discontinuous flows with adapted unstructured meshes. AIAA paper 2007-4186 (2007)
22. Mandelbrot, B.B.: The Fractal Geometry of Nature. W.H. Freedman and Co., New York (1982)
23. Marcum, D.: Iterative partitioning for parallel mesh generation. In: Tetrahedron Workshop, vol. 2 (2007)
24. Marechal, L.: The LP2 library. A parallelization framework for numerical simulation. Technical Note, INRIA (2009)
25. Mesri, Y., Zerguine, W., Digonnet, H., Silva, L., Coupez, T.: Dynamic parallel adaption for three dimensional unstructured meshes: Application to interface tracking. In: Proceedings of the 17th International Meshing Roundtable, pp. 195–212. Springer, Heidelberg (2008)
26. Niedermeier, R., Reinhardt, K., Sanders, P.: Towards optimal locality in mesh-indexings. Discrete Applied Mathematics 7, 211–237 (2002)
27. Park, M., Darmofal, D.: Parallel anisotropic tetrahedral adaptation. AIAA Paper 2008-0917 (2008)
28. Peano, G.: Sur une courbe, qui remplit toute une aire plane. Mathematische Annalen 36, 157–160 (1890)
29. Pilkington, J., Baden, S.: Dynamic partitioning of non-uniform structured workloads with spacefilling curves. IEEE Transactions on Parallel and Distributed Systems 117(3), 288–300 (1996)
30. Sagan, H.: Space-Filling Curves. Springer, New York (1994)

31. Sharov, D., Luo, H., Baum, J., Löhner, R.: Implementation of unstructured grid GMRES+LU-SGS method on shared-memory, cache-based parallel computers. AIAA Paper 2000-0927 (2000)
32. Shontz, S., Knupp, P.: The effect of vertex reordering on 2D local mesh optimization efficiency. In: Proceedings of the 17th International Meshing Roundtable, pp. 107–124. Springer, Heidelberg (2008)