MULTITHREAD PARALLELISM FOR NUMERICAL SIMULATION

How to easily accelerate any F.E. or F.V. solver on multicore CPUs

Loïc Maréchal / DLP february 2025

Motivations

- CPU sequential speed started to stall at beginning of the century. It used to double every two years (corollary to Moore's Law) since the 70s but doubling the speed takes around 6 or 7 years now.
- There are two reasons for this slow down: the heat wall (the heat dissipated per square mm grows with the square of the frequency) and the memory wall (memory latency is stuck to 60 ns -> 16 MHz).
- Since 2004 (free lunch time is over), the only way to keep on with Moore's Law is to use some kind of concurrent computing: distributed parallelism (MPI clusters), multithreading (pthreads on multicore CPUs), FPGS, GPU, vector accelerators.

Parallel architectures: distributed vs. shared

- Distributed memory: a cluster of multiple machines connected with a network. Machines cannot access each other’s memory, they need to explicitly ask for data transfer with MPI (explicit memory access). This architecture can scale to millions of cores but usually requires to completely rewrite the software.

- Shared memory: a single machine with a CPU containing several cores. The machine's memory is accessible from every core (implicit memory access). A sequential code can be multithreaded step by step rather easily.

- Hybrid systems: DSHM (Distributed Shared Memory), a single machine hosting one to eight CPUs connected with a high-speed memory bus. It can be programmed in tha same way as a shared memory system but getting an optimal scaling requires to completely rewrite the code)

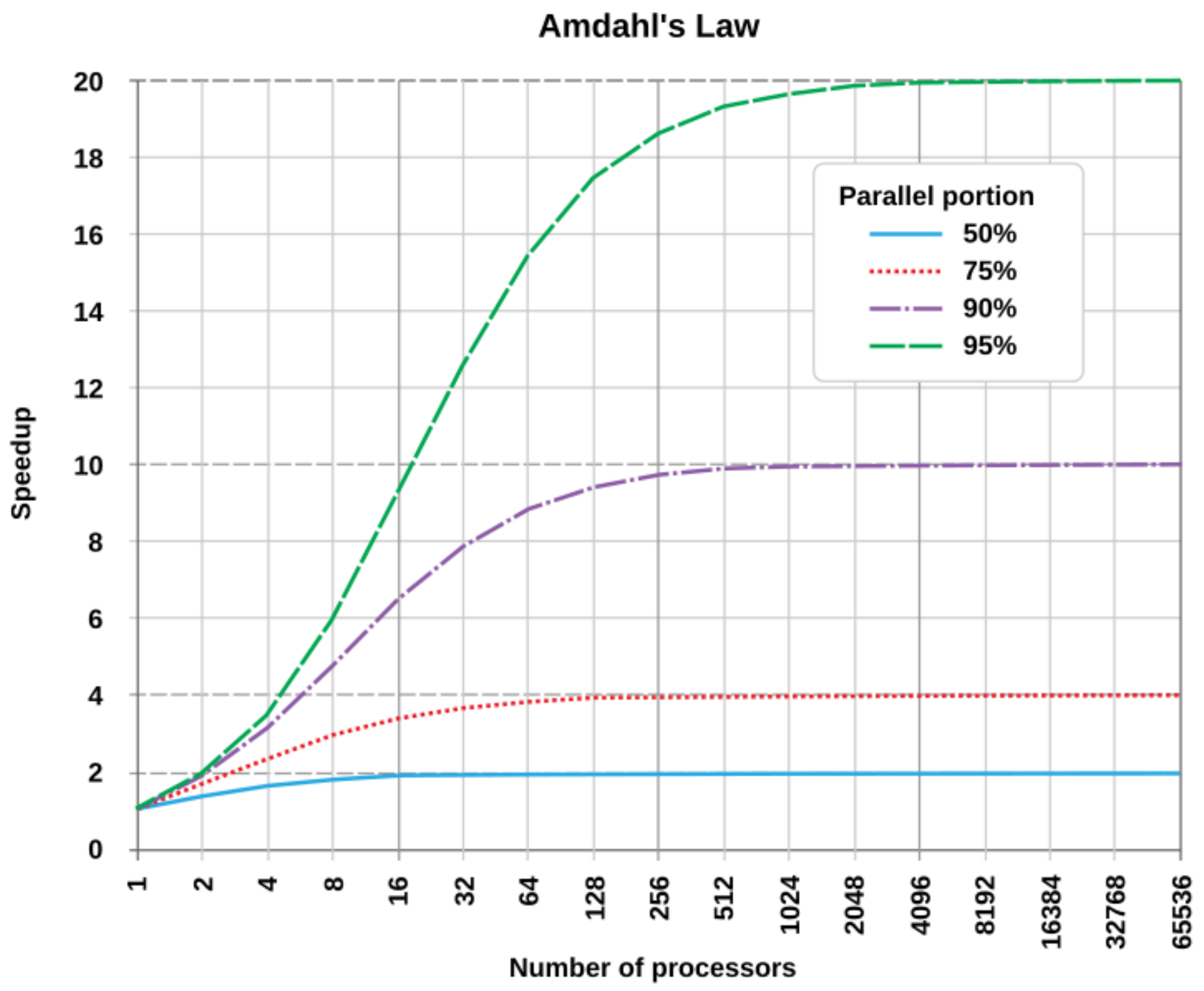
Fundamental laws of parallelism

- Amdahl's Law: if a code's normalized runtime S + P = 1, then it's theoretical maximum speedup is 1/S (1967 Gene Amdahl / IBM).

- Scaling vs. Speedup: comparing two versions of the same code, SC (Sequential Code) and PC(n) (// code run on n processors).

- Scaling: ratio between the runtimes of PC(1) and PC(n).

- Speed-up: ratio between the runtimes of SC and PC(n).



Fundamental laws of parallelism

- Handling the parallelism usually adds some compute and memory overhead which means that PC(1) may be slower than SC. Consequently, achieving a high speed-up is more complex than a high scaling.
- Strong scaling vs. Weak scaling:
- If a parallel code takes time T to process some data of size S using P processors, it should take the same time to process 2S data on 2P processors. This is called weak scaling, such parallelism is not meant to lower the runtime but to process bigger data. This is the kind of performance that is expected from distributed parallelism since adding more compute capacity adds more memory at the same time.
- If the same parallel code takes T/2 time to process the same amount of data S using 2P processors, it is called strong scaling. Such scaling is more difficult to obtain and is usually the aim of multithreaded parallelism as the data size is bounded by a single machine's memory.

Parallelism strategies

- Embarrassingly parallel: use concurrent computing only when it is straightforward. If not, don't waste your time on parallelism but use it on other software assets like debugging, UI, documentation and optimization.
- Parallel by design: since all hardware are parallel in some way, all software should adapt to it from its inception regardless of efficiency.
- Opportunistic parallelism: evaluate the cost/benefit of the different concurrent programing schemes. Multithreading can increase the speed by an order of magnitude at a reasonable cost.

Basic memory access patterns

- One of the biggest issues with parallel algorithms are concurrent memory writes at the same location which requires costly synchronizations.
- Direct memory read or write: **i=1..n; U(i) = V(i);** Uses the full memory bandwidth and straightforward to parallelize as there are no concurrent writes.
- Indirect memory reads with direct memory write: **i=1..n; U(i) = V(W(i));** No concurrent writes issues but much slower. Because of the memory indirection, the runtime depends on the memory latency (60 ns) not on the memory throughput (<1 ns).
- Indirect memory writes: **i=1..n; U(V(i)) = W(i);** memory writes may occur simultaneously at the same location causing wrong calculation. Each loop iteration must perform a costly synchronization before writing to memory.

Dealing with concurrent writes accesses

- Scatter gathers: each parallel thread writes only in its own local buffer and after all calculations are done, a parallel gathering process sums up all local contributions. This scheme works fine but adds useless runtime and memory consumption.

- Atomic operations: a lightweight synchronization to make sure that two threads will never write at the same memory location at the same time. It is very easy to use, but only a single instruction can be atomic, not a sequence of instruction. The hidden synchronization operations grow as the square of the number of threads !

- In place memory writes: partitions the data into more pieces as there are threads and make sure that all running threads write to different memory locations at the same time.

Access patterns with mesh data structures

- Loop over elements and read face, edge or node-based data, make some calculation and store the result inside an element based structure: indirect memory reads, direct memory writes. No issue with parallelization but inefficient memory access pattern.

- Loop over elements, make some calculations based on elements' data and add the contribution to its faces, edges or nodes: these indirect memory writes must be dealt with.

- Stack based, recursive or arbitrary path algorithm: very challenging to parallelize with any scheme !

Parallel languages and libraries

- OpenMP: general purpose, efficient with simple memory access patterns but not optimized for algorithms working on meshes.
- Atomic operations: efficient but not general purpose and can be tricky to use when working with complex data structures.
- Compilers' automatic parallelization capacity: general purpose but inefficient except for very basic algorithms.
- Pthreads: very efficient and general purpose but requires a lot of knowledge. Quite invasive in the source code.
- LPlib: a library dedicated to multithreading loops working on unstructured meshes. Efficient and easy to use but not for general purpose.

The LPlib

- One ANSI C file to compile and link along your code and one header file to include.
- No need to rewrite your code, an existing serial code can be easily parallelized.
- Light library memory overhead, no user's data duplication.
- Threads run asynchronously, avoiding synchronization barriers.
- High concurrency thanks to dynamic scheduling.
- Based on open source and standard pthread library that is available on all systems (Linux, macOS, Windows)

Direct loops: partitions

- 2 threads -> 2 blocks
- Block 1: elements 1,2,3,4,5,6
- Block 2: elements 7,8,9,10,11,12

3	6	9	12
2	5	8	11
1	4	7	10

Direct loops: data structures

```
typedef struct{

double t, coordinates[2];

int num;

}VerStruct;
```

```
typedef struct{

VerStruct *VTab[4];

double t;

}QuadStruct;
```

```
typedef struct{

int nbv, nbq;

VerStruct VTab[ nbv ];

QuadStruct QTab[ nbq ];

}MeshStruct;
```

Direct loops

```
main()
{
  for(i=0; i<mesh->nbq; i++)
    for(j=0; j<4; j++)
      mesh->QTab[i]->t += mesh->QTab[i]->VTab[j]->t;
}
```

```
void AddTemperature(int begin, int end, int thread,
void *arguments) {
  MeshStruct *mesh = (MeshStruct *)arguments;
  for(i=begin; i<end; i++)
    for(j=0; j<4; j++)
      mesh->QTab[i]->t += mesh->QTab[i]->VTab[j]->t;
}
main() {
  LibIdx = InitParallel(2);
  QuadType = NewType(LibIdx, mesh->nbq);
  LaunchParallel(LibIdx, QuadType, 0, AddTemperature, mesh);
  StopParallel(LibIdx);
}
```

Indirect loops

```
main() {
   for(i=0; i<mesh->nbq; i++)
      for(j=0; j<4; j++)
         mesh->QTab[i]->VTab[j]->t += mesh->QTab[i]->t;
}
```

```
void AddTemperature(int begin, int end, int thread, void *arguments) {
   MeshStruct *mesh = (MeshStruct *)arguments;
   for(i=begin; i<end; i++)
      for(j=0;j<4;j++)
         mesh->QuadTab[i]->VerTab[j]->t += mesh->QuadTab[i]->t;
}
main() {
   LibIdx = InitParallel(2);
   QuadType = NewType(LibIdx, mesh->nbq);
   VerType = NewType(LibIdx, mesh->nbv);
   BeginDependency(LibIdx, QuadType, VerType);
   for(i=0; i<mesh->nbq; i++)
      for(j=0; j<4; j++)
         AddDependency(LibIdx, i, mesh->QTab[i]->VTab[j]->num);
   EndDependency(LibIdx);
   LaunchParallel(LibIdx, QuadType, VerType, AddTemperature, mesh);
   StopParallel(LibIdx);
}
```

Indirect loops: partitioning

3	6	9	12
2	5	8	11
1	4	7	10

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 0 & 0 \\ 2 & 1 & 1 & 1 & 0 \\ 3 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 \end{array}$$

- 2 threads -> 4 blocks !
- Block 1: elements 1,2,3
- Block 2: elements 4,5,6
- Block 3: elements 7,8,9
- Block 4: elements 10,11,12

- 1 / the two threads will process
- blocks 1 and 3 concurrently
- 2 / then they will process blocks 2 and 4

Sample runtimes and scaling

Code	Mesh size	Sequential	Parallel with 1 thread	Parallel with 64 threads	Speed-up	Scaling
basic test	**100M**	11.80		0.25	47	
indirect write	**925M**	718.72		19.77	36	
build edges	**925M**	74.66	88.43	2.86	26	31
setup neighbours	**925M**		138.53	8.91		15
Wolf	**33M**	815.67		18.33	44	
P1toPk	**10M**		10.97	0.32	34	

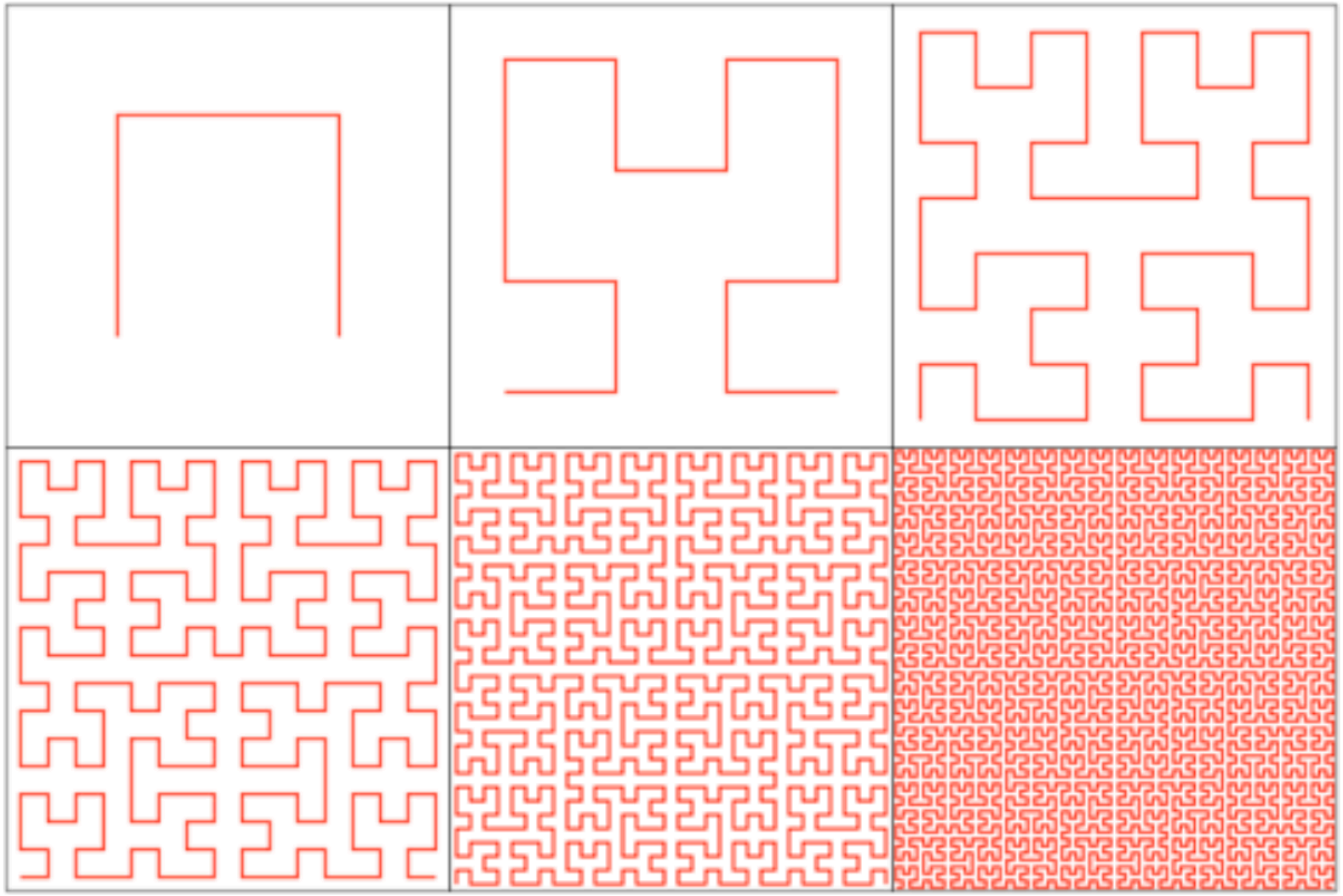
Hilbert renumbering: pattern

Level 1

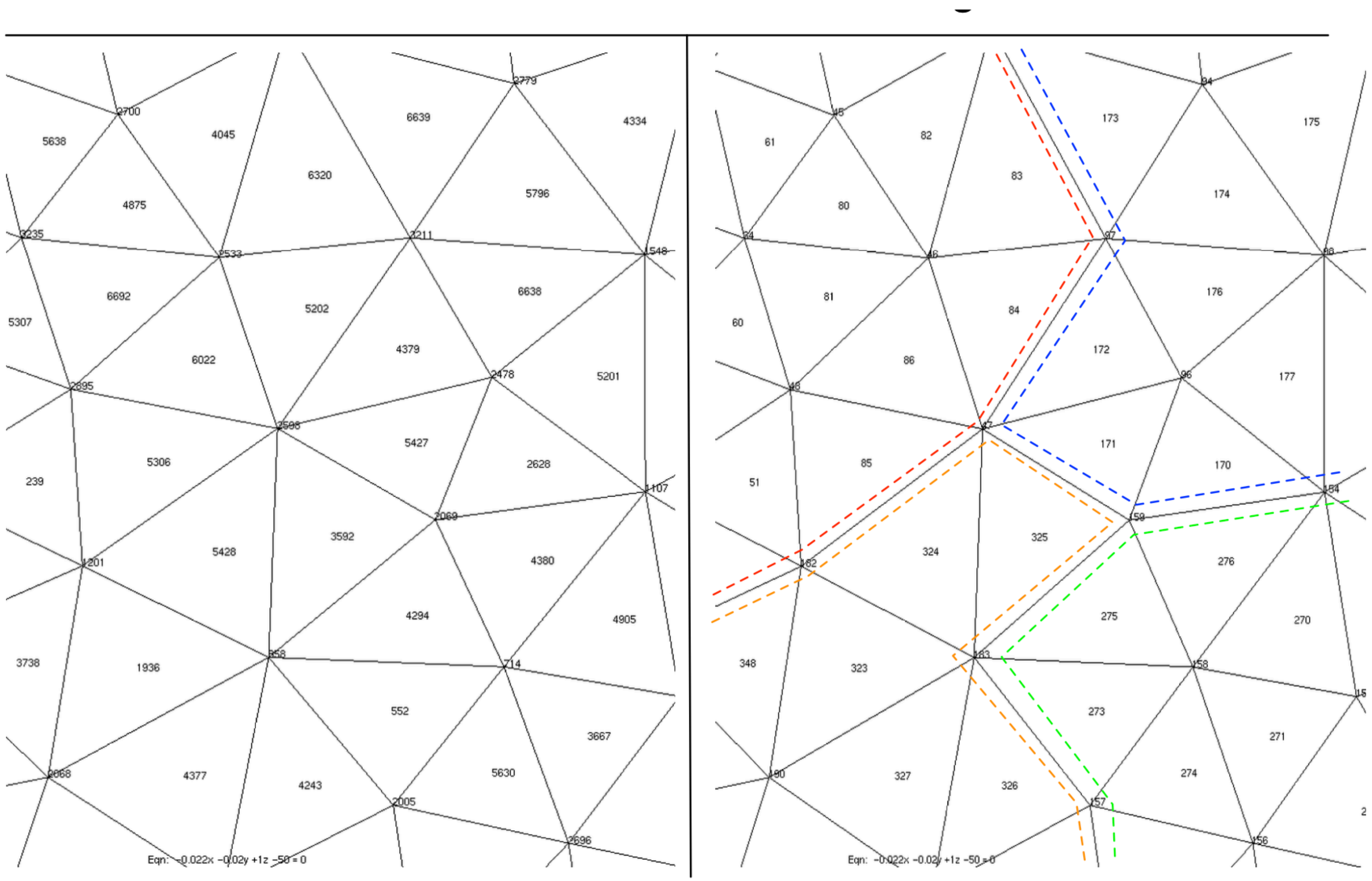
2	3
1	4

Level 2

6	7	10	11
5	8	9	12
4	3	14	13
1	2	15	16

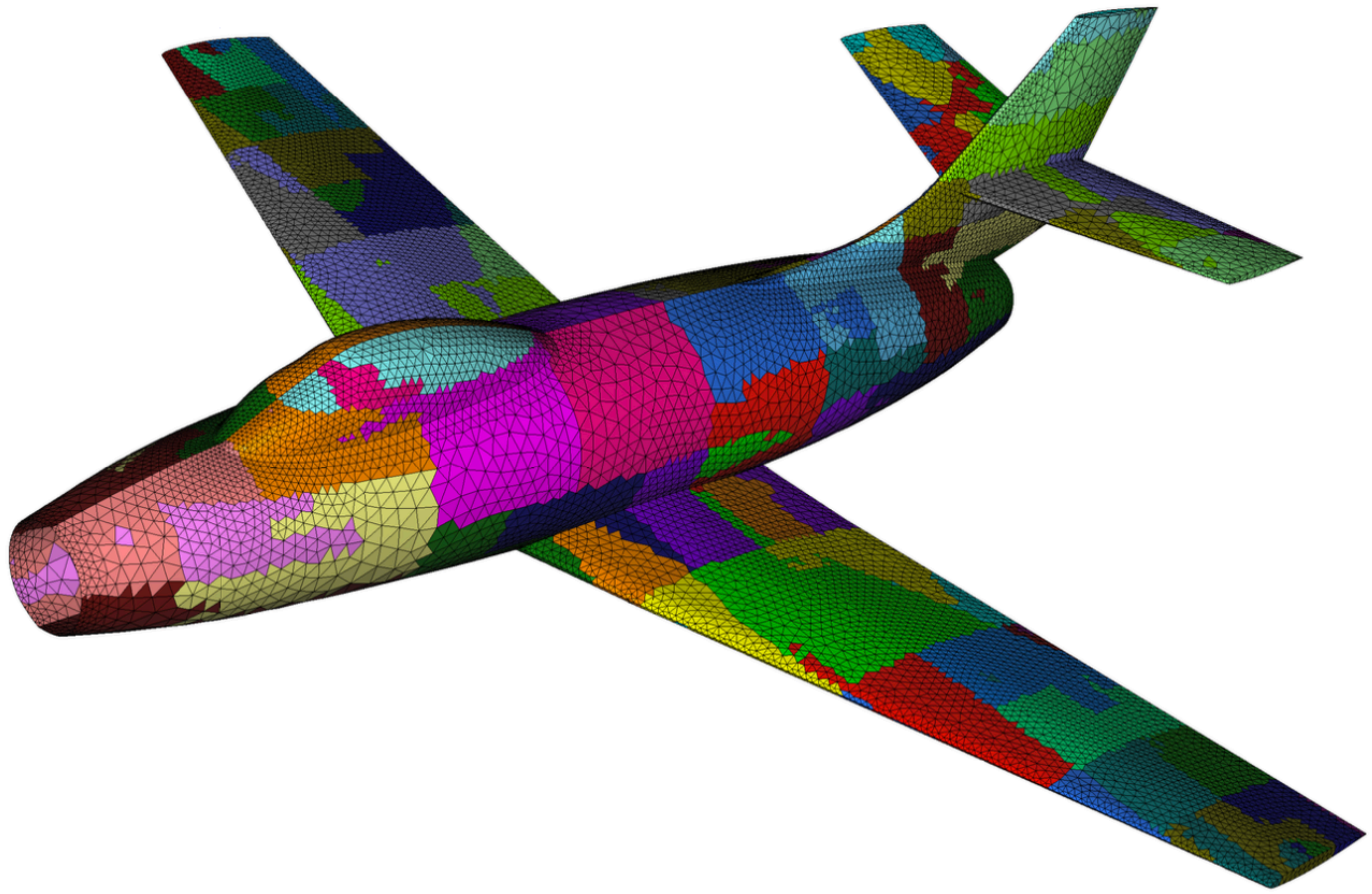


Hilbert Renumbering: a close look



Eqn: -0.022x -0.02y +1z -50 = 0
Eqn: -0.022x -0.02y +1z -50 = 0

Hilbert Renumbering: visual partitioning



Hilbert acceleration

- Hilbert or any SFC renumbering reduces cache misses and greatly increases memory throughput in indirect memory reads loops.
- It also enables an implicit mesh partitioning that is suitable for in-place parallelism to perfectly handle indirect memory writes loops.
- A simple mesh reordering addresses both memory indirection challenges.
- It even speeds up sequential algorithms that work with meshes and were not meant to use any kind of SFC renumbering.

	Random	**Hilbert**	**Speed-up**
libHOM Compute 25M P2 tets	11.64	9.96	1.17
Vizir open 100M hexes	8.32	5.85	1.42
Optet optimize 25M tets	182.64	91.29	2.00
Build face neighbours of 25M tets	36.00	11.68	3.08

Software availability

- Open source with BSD3 license: you can do whatever you want with it.
- Available on GitHub: https://github.com/LoicMarechal/LPlib
- Documentation and examples are provided: it is easier to start from one of the examples and adapt it to your needs.
- Requires only the pthread library: link with -lpthread on Linux, pthread.dll on Windows and nothing to add on macOS ;-)
- Posix Threads has been the de facto standard for more than 30 years and is here to stay.
- Hardware, system and compiler agnostic: no vendor lock-in.

Perspective

- Slide n°6 sums it all !
- Set a speed increase objective beforehand: x10 or x1000 imply completely different paths.
- The sequential algorithm you are developing may be facing competitors that use a 384 multicore CPU (2xAMD Epyc 9965), 7.6 million cores (Fujitsu Fugaku) or 600 million GPU compute units (HPE-Cray El Capitan).
- Concurrent programing imposes a constant burden on the development, maintenance and evolution process.
- But it's so much fun !