THE LP3 LIBRARY

Easy and efficient multithreading
of serial codes that deal with meshes

Part 1: from the user's point of view
Part 2: multiple chips machines
Part 3: Hilbert renumbering

Loïc Maréchal, INRIA, jan 2015, MSU

LP3 Part 1: motivations

Shared memory parallelism:
+little knowledge needed
+step by step parallelization of serial code
+short development time
-limited speed up (an order of magnitude)
-limited data size

Distribute memory parallelism:
+strong scaling capabilities (3 orders of magnitude)
+memory capacity grows with compute capacity
-strong knowledge is needed if you want to do it all by yourself
-it takes to completely parallelize the code before you can run it on a cluster
-scaling is not guarantied

It is a choice between a low risk, low reward approach
and a high risk, high reward one.

LP3 Part 1: from the user's point of view

LP3 = Loop Parallelism Library Version 3

Works on shared memory computers: since CPU frequency stalled in recent years and distributed memory parallelism requires a lot of time and knowledge, we think that multi-threading is the best way to go.

Enables loop parallelism in C programs

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

LP3 Part 1: simple direct access loop

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

2 threads -> 2 blocks

Block 1: elements 1,2,3,4,5,6

Block 2: elements 7,8,9,10,11,12

LP3 Part 1: simple direct access loop

```
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;
```

Serial loop

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

Parallel loop

```
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);
}
```

LP3 Part 1: indirect access loop

```
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);
}
```

LP3 Part 1: indirect access loop

Sample mesh

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

Compatibility matrix between blocks

	1	2	3	4
1	1	1	0	0
2	1	1	1	0
3	0	1	1	1
4	0	0	1	1

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

LP3 Part 1: some results

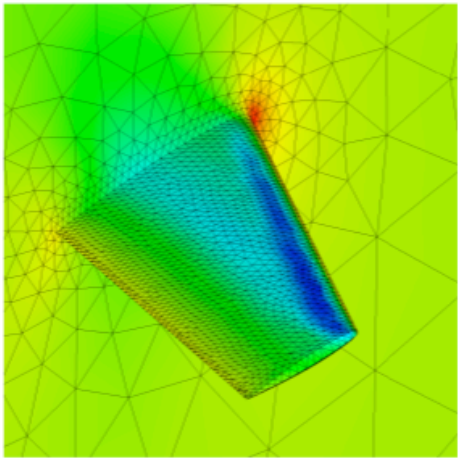
HEXOTIC hex mesher:
Parallel section represents 67% of total serial run time.
Only node smoothing has been parallelized which required two days of work.

cpu	amdahl	speedup
2	1.5	1.5
4	2	1.9
8	2.4	2.2

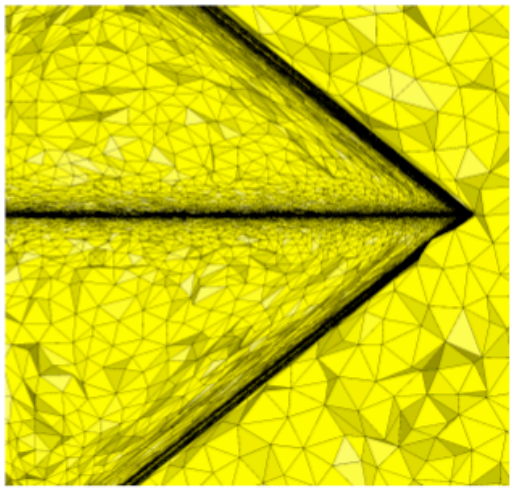
Amdahl's law:
p = parallel section of a program
s = serial section
p + s = 1
speedup with n processors = 1/(s + p / n)
maximum theoretical speedup with an infinite number of processors = 1 / s

WOLF cfd solver:
Parallel section represents 98% of total serial run time.
A dozen of loops have been parallelized in three days.

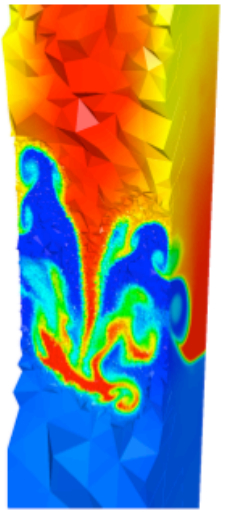
cpu	amdahl	speedup
2	1.9	1.8
4	3.7	3.1
8	7	5.7
16	12.3	10.3
32	19.7	17



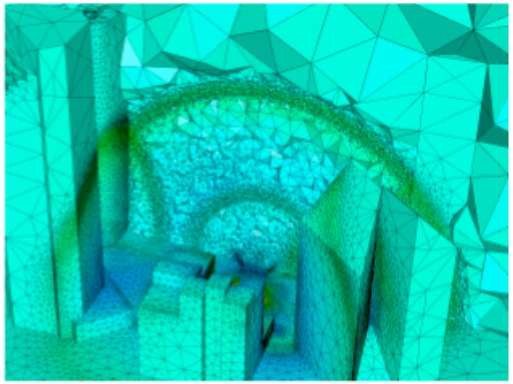
M6



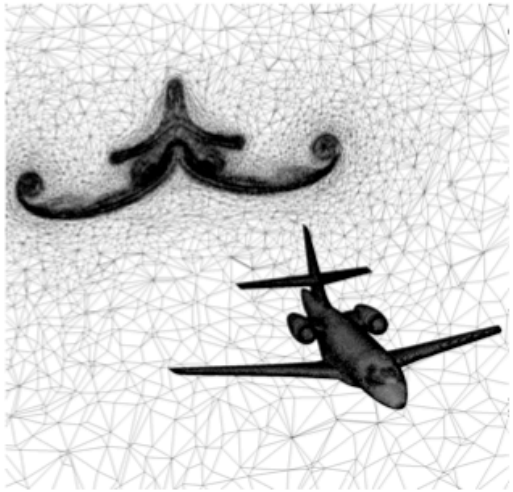
SSBJ



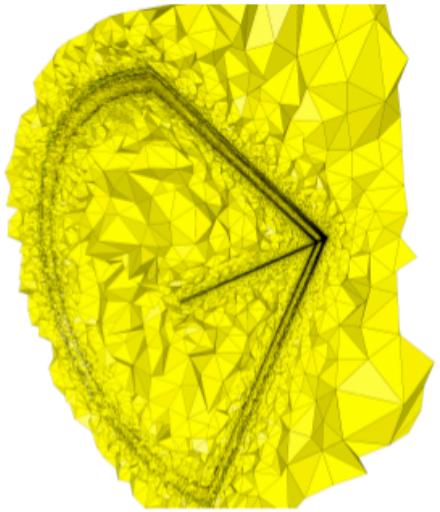
IRT



City



Falcon



Spike

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

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

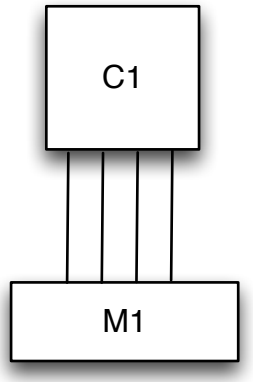
LP3 Part 1 : performance

WOLF CFD solver "spike" test case: 50 iterations, 9 millions vertices:
-running on an xserve with 2 xeon 5570 quad-core at 2.93 ghz
-serial code: 2203s
-parallel code on 8 cores: 289s on wall clock, 2312s cumulative time
-speedup is 7.6
-total parallelism overhead is less than 5%
-memory overhead is negligible: 300 KB

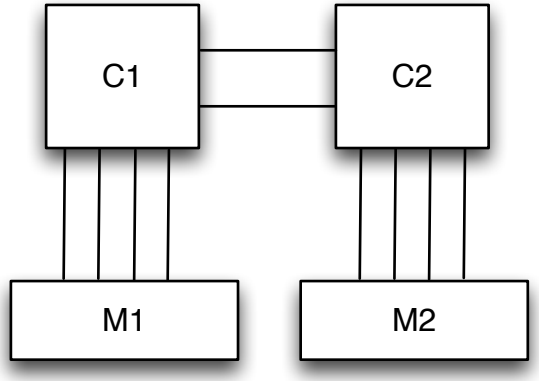
The xserve crossbar architecture is OK, but ccNUMA machines are more challenging:
-altix 4700 with 128 itanium2 at 1.6 ghz
serial code = 9354s
// 8 cores = 1539s (speedup = 6)
//100 cores = 256s (speedup = 36)

Several reasons to these poor speedups:
-Amdahl's law: serial part is 2% in this test case, so theoretical speedup is 50
-High memory latency: 332ns cache miss on Altix vs 112ns on the xserve
-LP3 library overhead

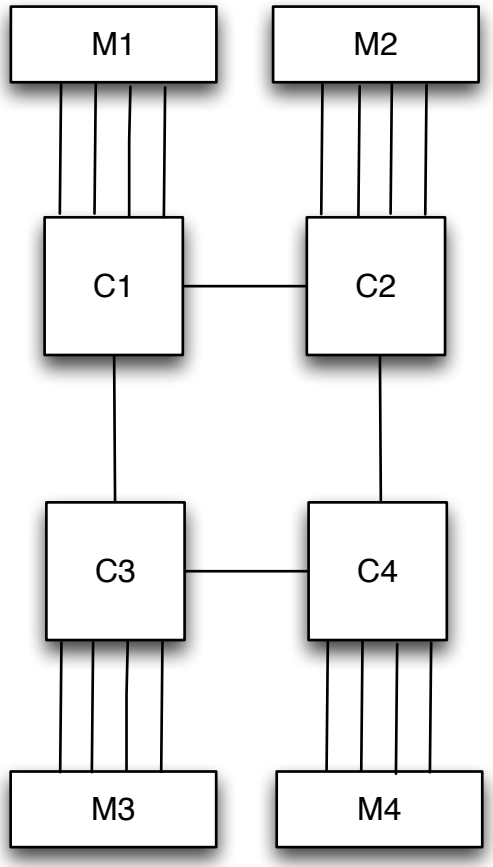
LP3 Part 2: Multiple chips machines



1 CPU:
latency = 100 ns
throughput 60 GB/s



2 CPUs:
latency = (100 + 200) / 2 = 150 ns
throughput (60 + 30) / 2 = 45 GB/s



4 CPUs:
latency = (100 + 200 + 200 + 300) / 4 = 200 ns
throughput (60 + 10 + 10 +10) / 4 = 22.5 GB/s

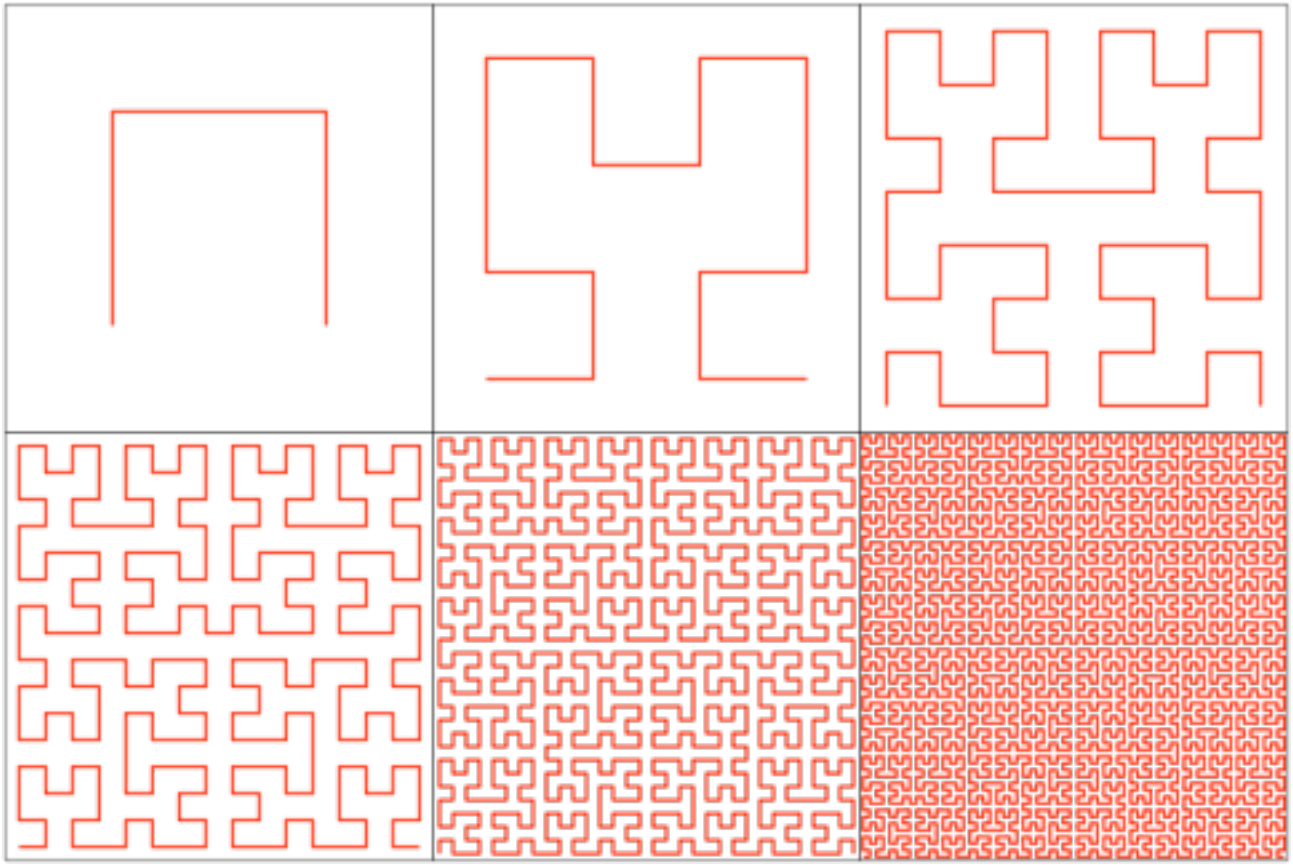
LP3 Part 3: Hilbert renumbering

Level 1

2	3
1	4

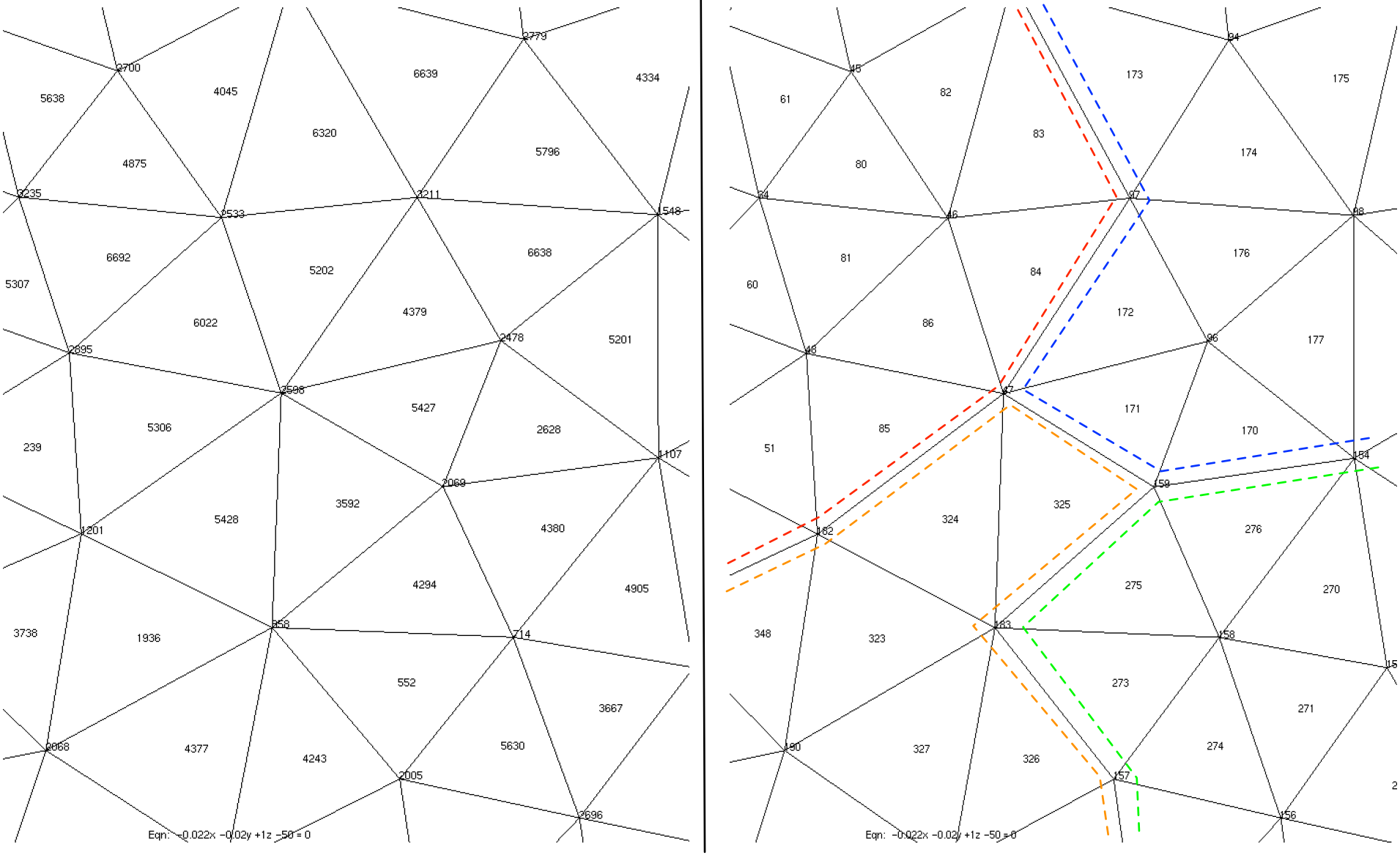
Level 2

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



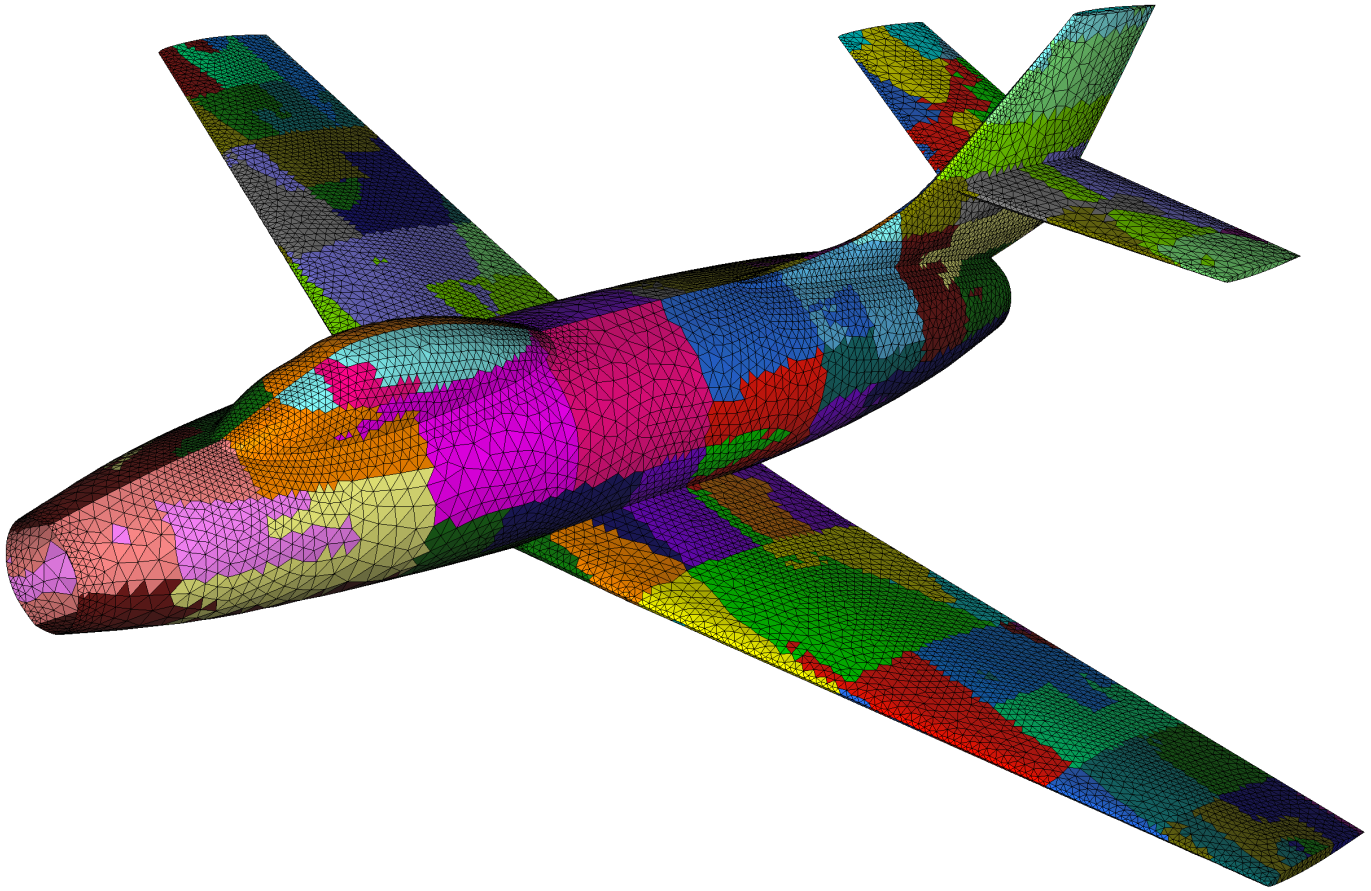
Loïc Maréchal, INRIA, jan 2015, MSU

LP3 Part3: Hilbert renumbering



Loïc Maréchal, INRIA, jan 2015, MSU

LP3 Part 3: Hilbert renumbering



Loïc Maréchal, INRIA, jan 2015, MSU

LP3: Hilbert renumbering

Test case: a 10 millions tetrahedral mesh is optimized by performing four optimizations steps that smooth nodes and swap edges to improve the quality of elements.
Both smoothing and swapping are multithreaded.

Renumbering time: 2.4 seconds on a quad core laptop.

You can either use an external command line or an internal LP3's procedure.

Core i7	original	Hilbert	speed up
serial	63.1	41.3	1.5
4 cores	36.9	12.7	2.9
speed up	1.7	3.3	

Xeon E5	original	Hilbert	speed up
serial	53.1	41.3	1.2
10 cores	31.8	4.8	6.6
speed up	1.7	8.6	

Conclusion: multithreading is worth considering

Qualities:
-it has a very low memory and time overhead
-threads run asynchronously, avoiding synchronization barriers.
-it is much more nimble since, if a thread gets stuck for any reason,
the others will keep on working without waiting for it.
-memory duplication is avoided with the help of SFC and dependencies tables
-can speed up a code by an order of magnitude with a limited effort

Limitations:
-scaling is harder with ccNUMA multi-chips machines
-lowering Amdahl's factor under 1% is difficult while only parallelizing loops
-64+ cores machines are more expensive than their cluster counterparts

Future work:
-hybrid distributed and shared memory parallelism is probably the best way to go
-parallel stacks based processing because loops are not always the best iterators