Parallel Programming at the Exascale Era: A Case Study on Parallelizing Matrix Assembly For Unstructured Meshes

Eric Petit, Loïc Thebault , Quang V. Dinh

HPC-CFD Workshop, August 2014

Exascale Challenges for RT and Applications

- Increasing number of nodes, increasing number of cores, accelerators
  - Some ressources do not scale
    - Memory per core, Bandwidth, Coherence protocol, Network interconnect, Fault tolerance
  - Frontier are becoming fuzzier => Heterogeneity
    - Distributed/shared? Software/hardware? "Core" definition? Compute capabilities, imbalance…
  - Multiplication of hierarchical levels => Non uniformity
    - Different scales: BW, memory size, performance
    - Global events: barrier, broadcast, memory coherency

Exascale Challenges for RT and Applications

Evolutions are requested for applications, runtimes and programming models

Exascale Challenges for RT and Applications

- **More concurrency**
  - Enough independant tasks
  - Communication overlap
  - Privatize memory to avoid communication (& sync)
  - Remember Amdhal: The more core, the higher is the proportion of the sequential code
- **More locality**
  - Memory
    - Core level, Socket level (including HWA), Network level
  - But also communications
    - Synchronisation, Data
- **We need both for performance scalability**

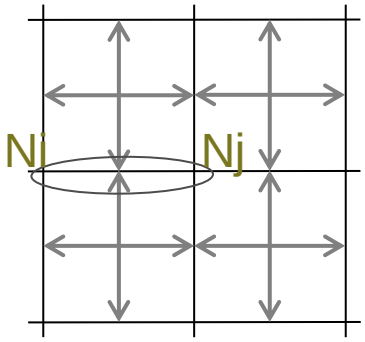
Divide and Conquer Parallelization of Finite Element Method Assembly

- Objectives:
  - Expose parallelism
    - Expose many independent task for many cores
  - Make an efficient use of shared memory
    - Save BW (Locality)
    - Save Memory (Avoid MPI buffer)
  - More efficient synchronisation
    - Prepare scaling on a larger number of cores
    - Keep synchronoisation local
    - Use shared memory instead of MPI
  - Improve the load balancing
    - Fine grain parallelism
    - Work stealing/work sharing

Finite Element Method Assembly

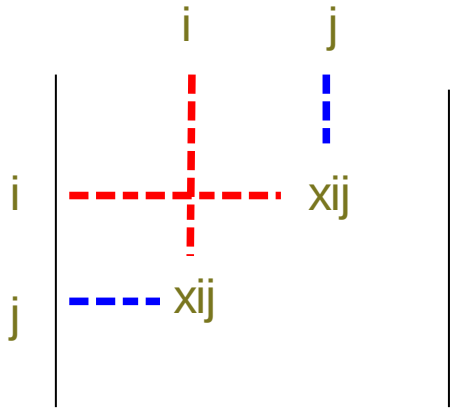
- Basic FEM iteration:
  - Matrix A assembly from the mesh
  - Solve Ax=b
  - Update mesh (values, geometry)
- Building the linear system from the MESH
  - Elt-Node => Sparse matrix storage
  - Reduction of elements values on the edges and / or the nodes
  - WE CONSIDER UNSTRUCTURED MESHES

Finite Element Method Assembly 2D Illustration



Reduction done on each edge from all neighbor elements

Edges update (+= reduction) must be sequential



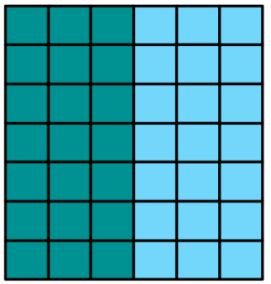
Xij ≠ 0 if there is an edge between i and j

(Very) Sparse and symetric matrix

MPI Domain Decomposition

Distributed memory parallelism

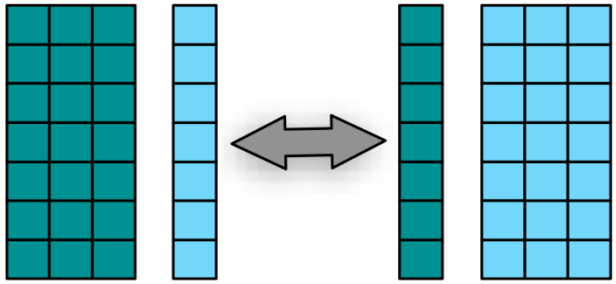
Communicate interface elements



Efficient on current architectures

Sub-optimal on future architectures

- Data duplications
- Synchronizations/Communication



Coloring Approach

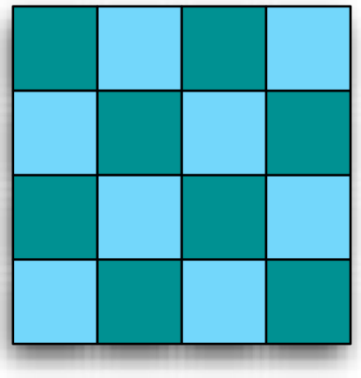
Elements sharing an edge have different colors

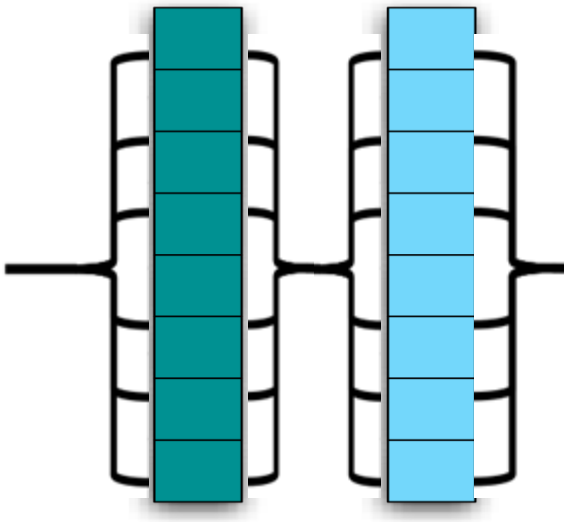
Colors are computed sequentially

Elements of a same color can be computed in parallel

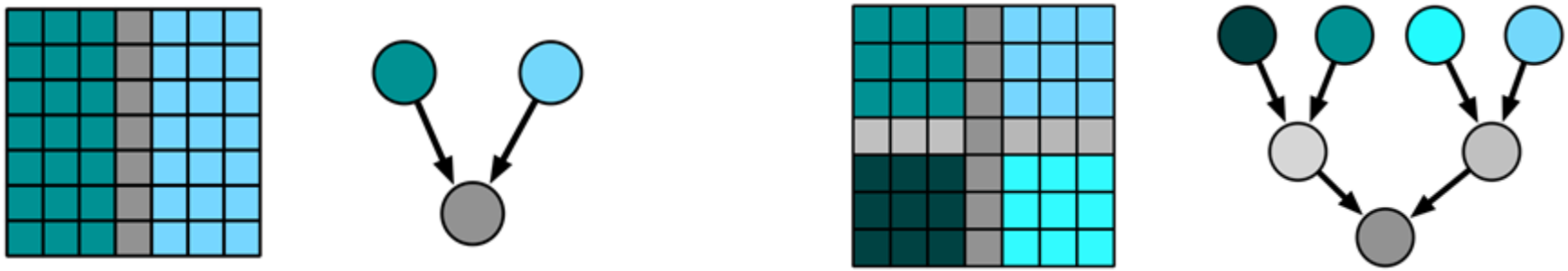
Simple to implement

Bad temporal locality
High memory bandwith requirements
Global synchronizations





The D&C approach

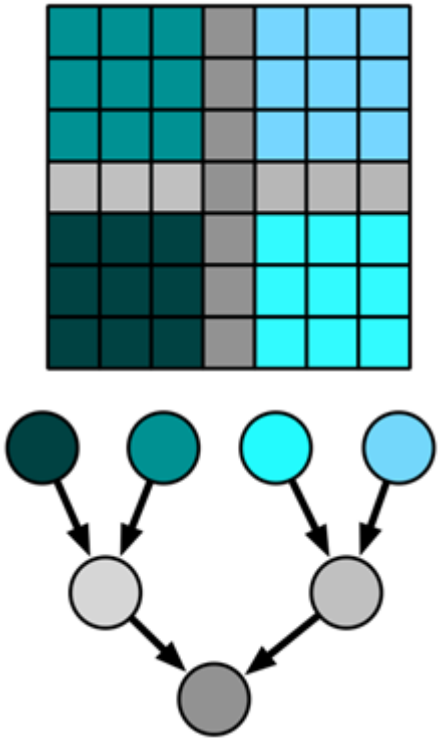


Left and right sub-domains are independant

Elements at the frontier form a separator
Separators must be treated after left and right sub-domains

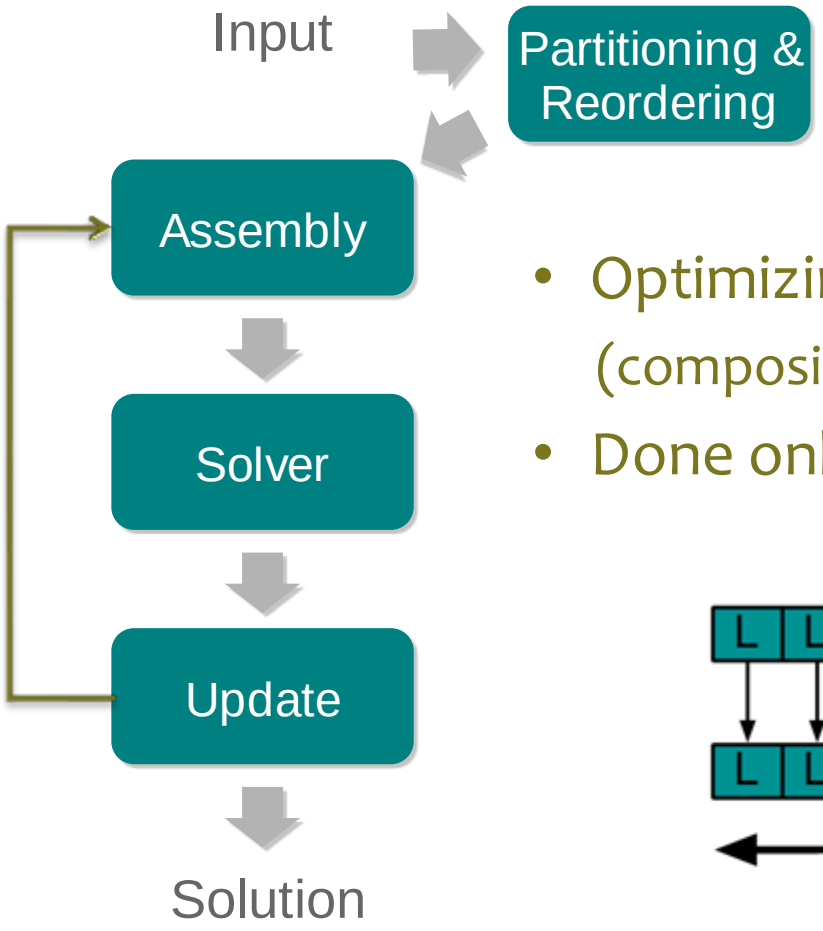
Applied recursively to all sub-domains → Recursive Tree

The D&C approach

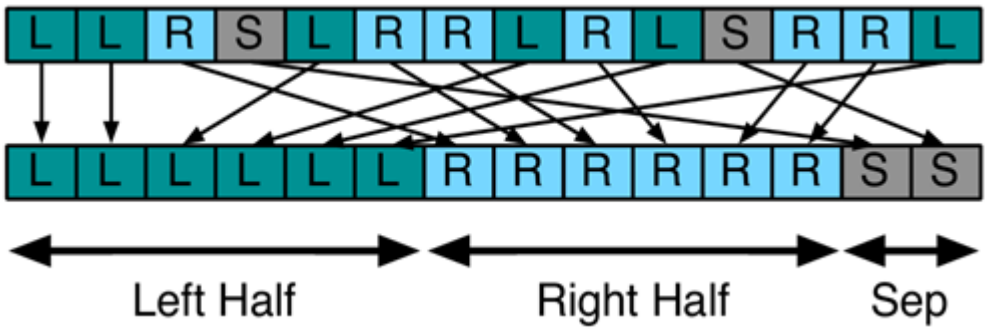


```
function compute (subdomain)
  if Node is not a leaf
    spawn compute (subdomain.left)
    compute (subdomain.right)
    sync
    compute (subdomain.sep)
  else
    FEM_assembly (subdomain)
end
```

The D&C approach

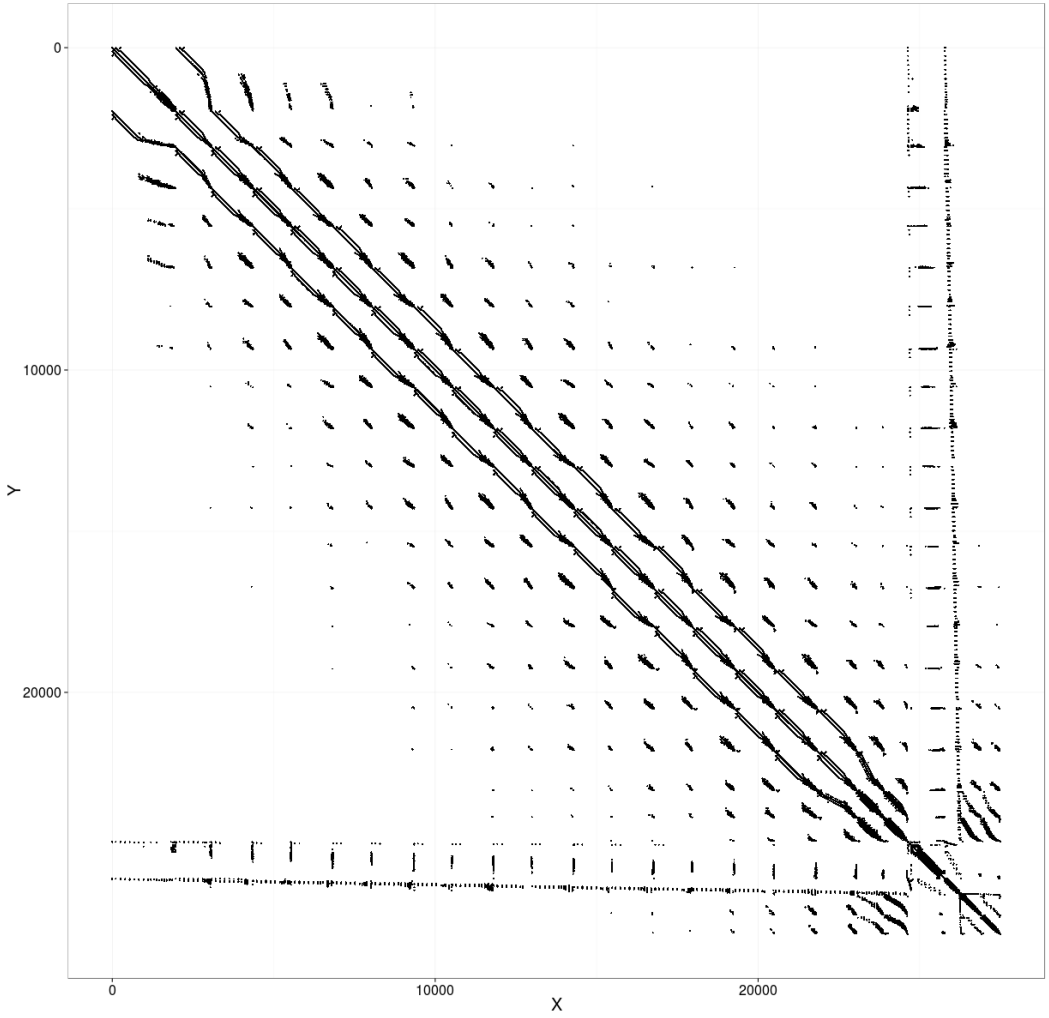


- Optimizing locality : data permutation (composition of all recursive node permutation)
- Done only once (Topological partition)



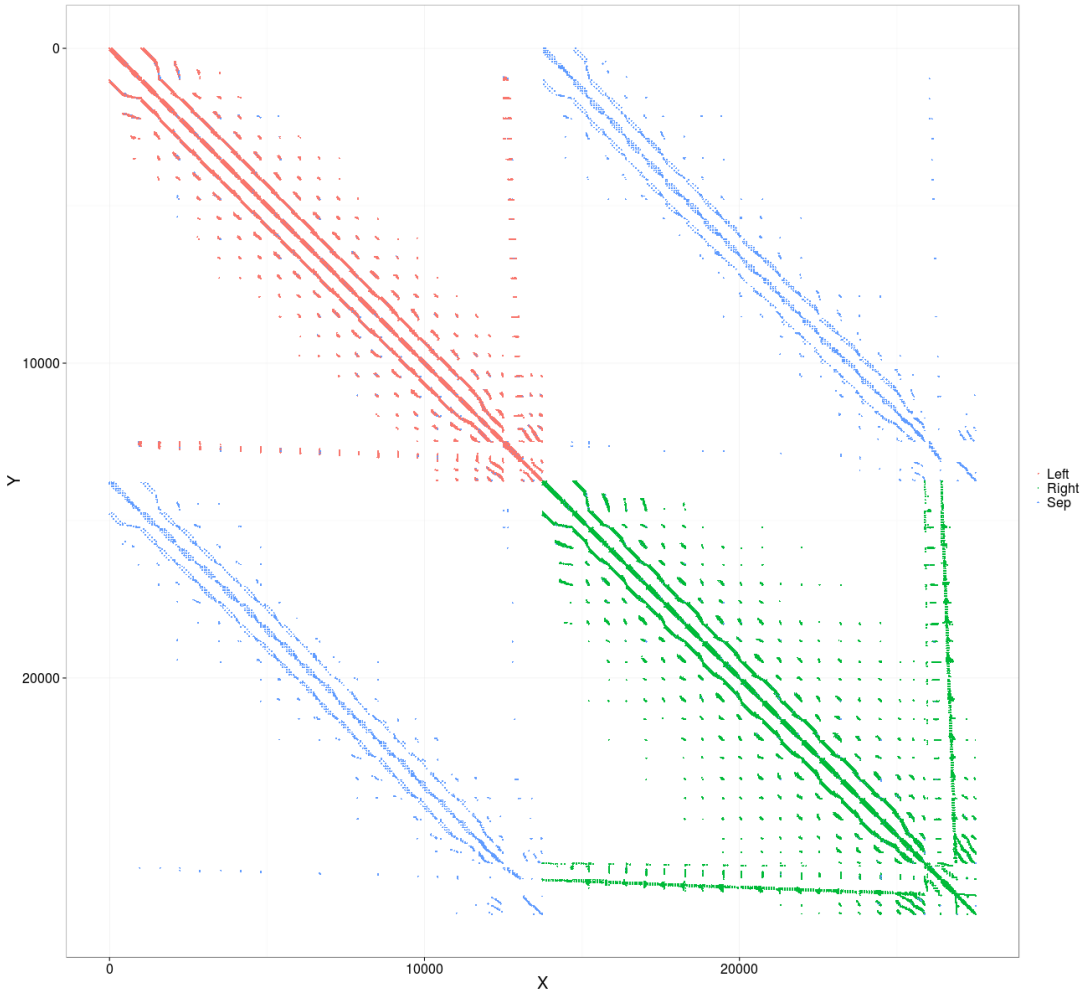
An Illustration of the D&C impact

- Initial



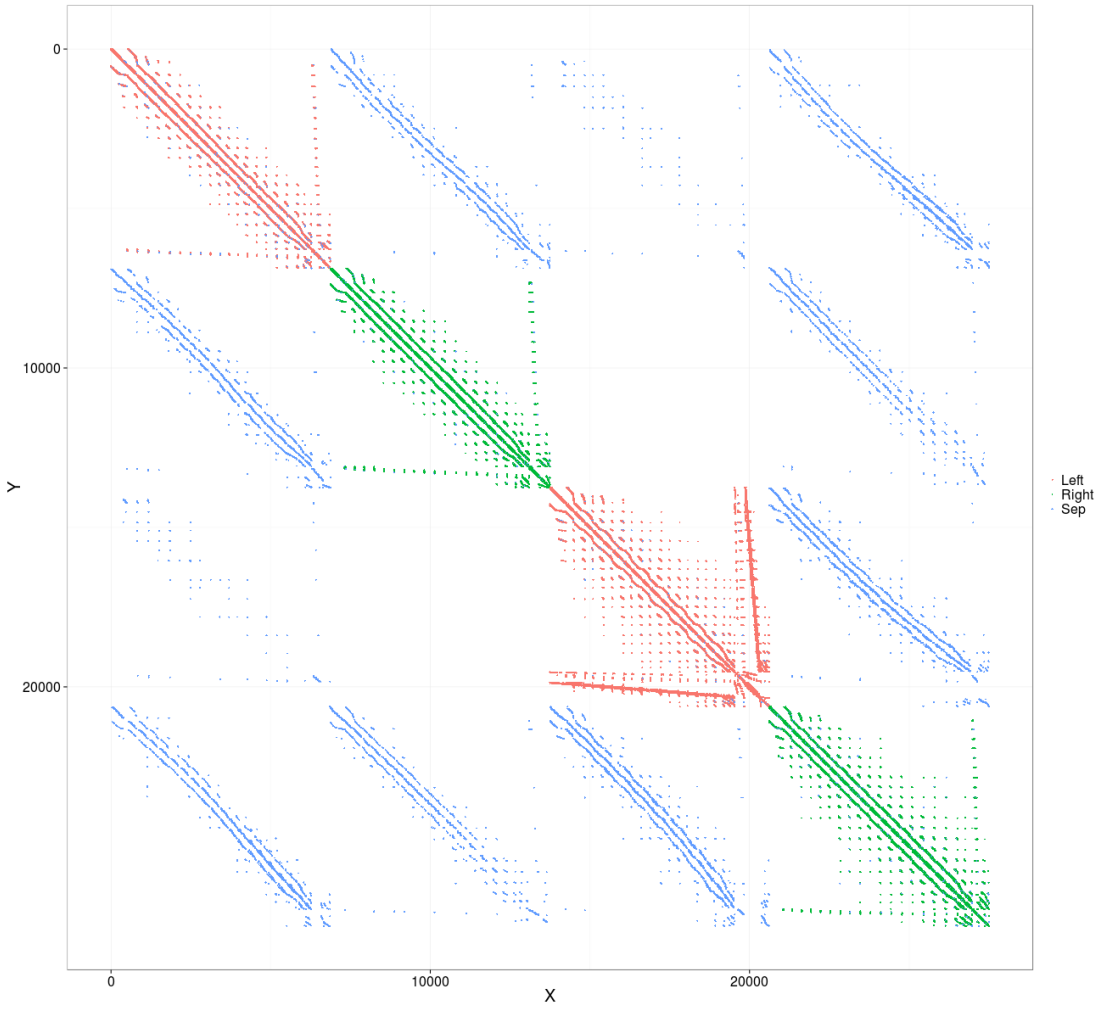
An Illustration of the D&C impact

- D&C 2



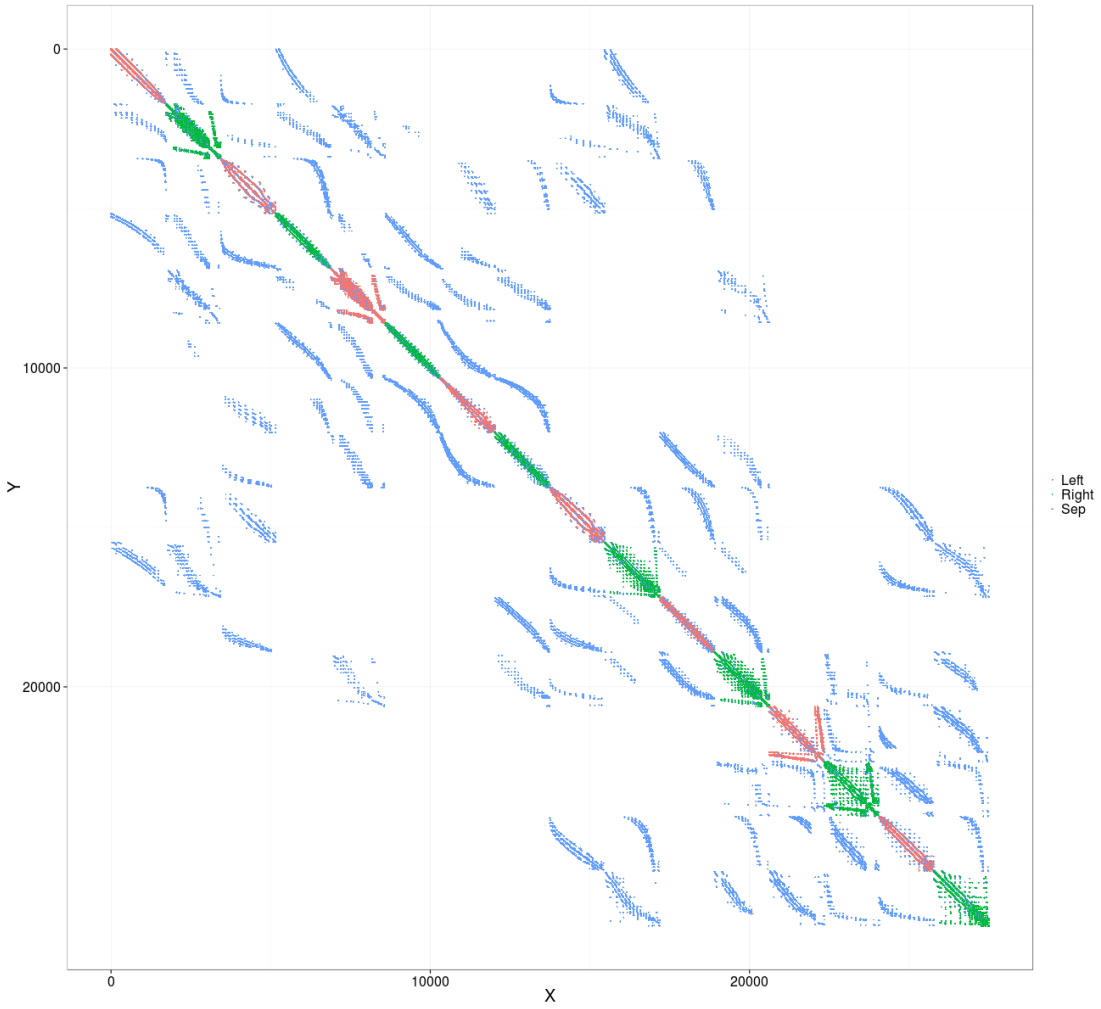
An Illustration of the D&C impact

- D&C 4



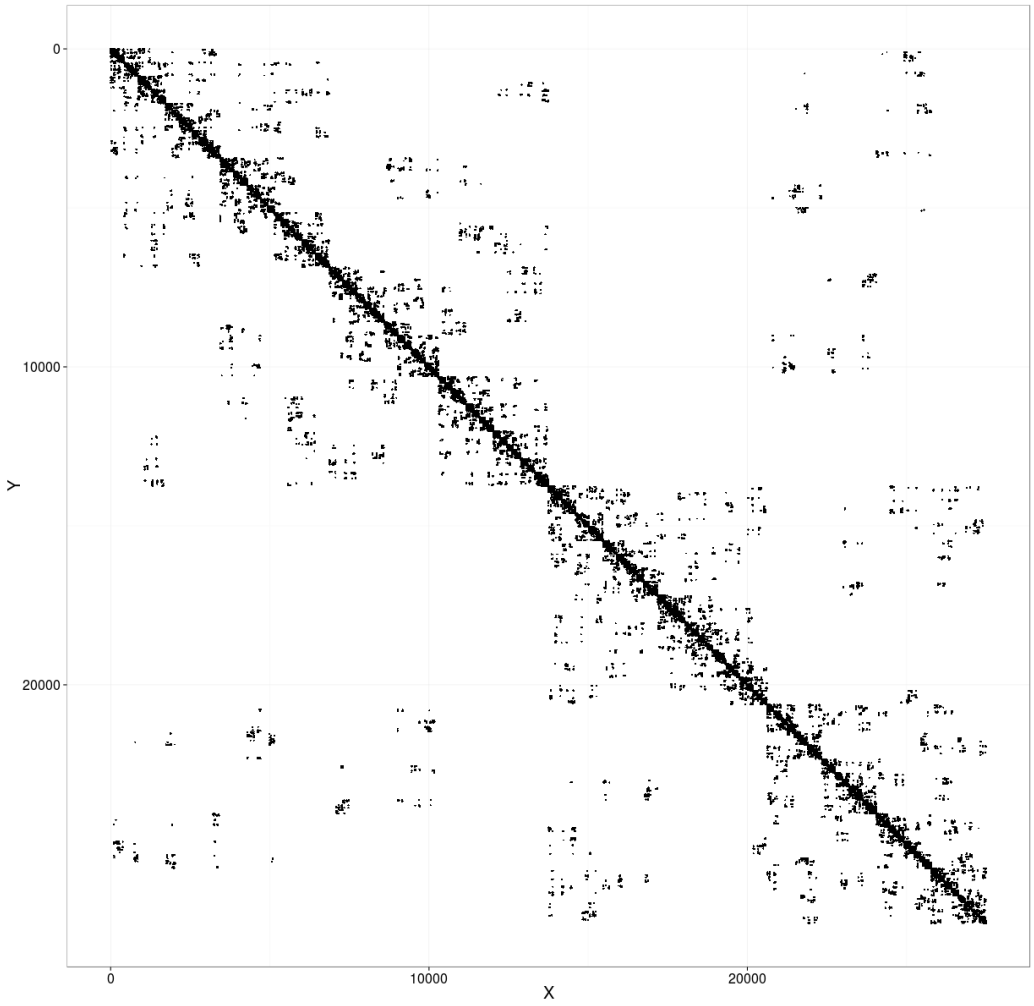
An Illustration of the D&C impact

- D&C 16



An Illustration of the D&C impact

- D&C 3000
- Leaves Size <50



The D&C approach

Can create many independant tasks

=> **Concurrency**

Leaves data set can be downsized at will to fit into caches

=> **Data locality**

Only one synchronization per task between the 2 local children

=> **Sync locality**

Only Log (N) synchronizations on the critical path

=> **Sequential part minimization**

The Dassault test case

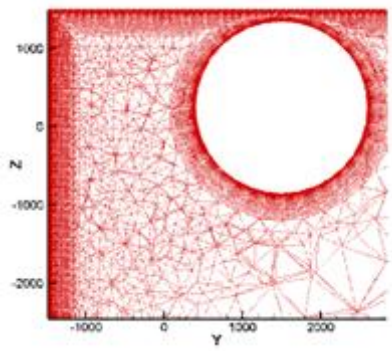
DEFMESH Mesh Deformer from Dassault Aviation

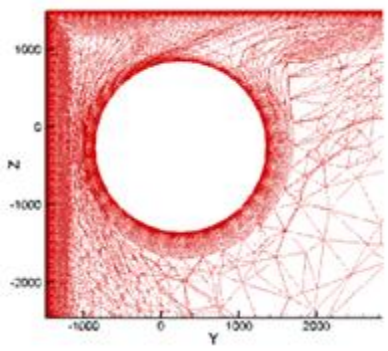
- Fluid Dynamic Code based on Finite Element Method
- Non-Linear: displacements cut in small increments

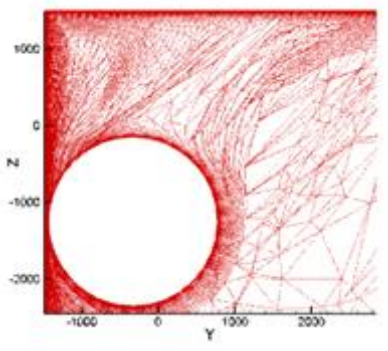
3D Mesh: Airplane Fuel Tank

- Composed of 6 346 108 elements and 1 079 758 nodes

We build an extrapolated proto-app to open-source our solution

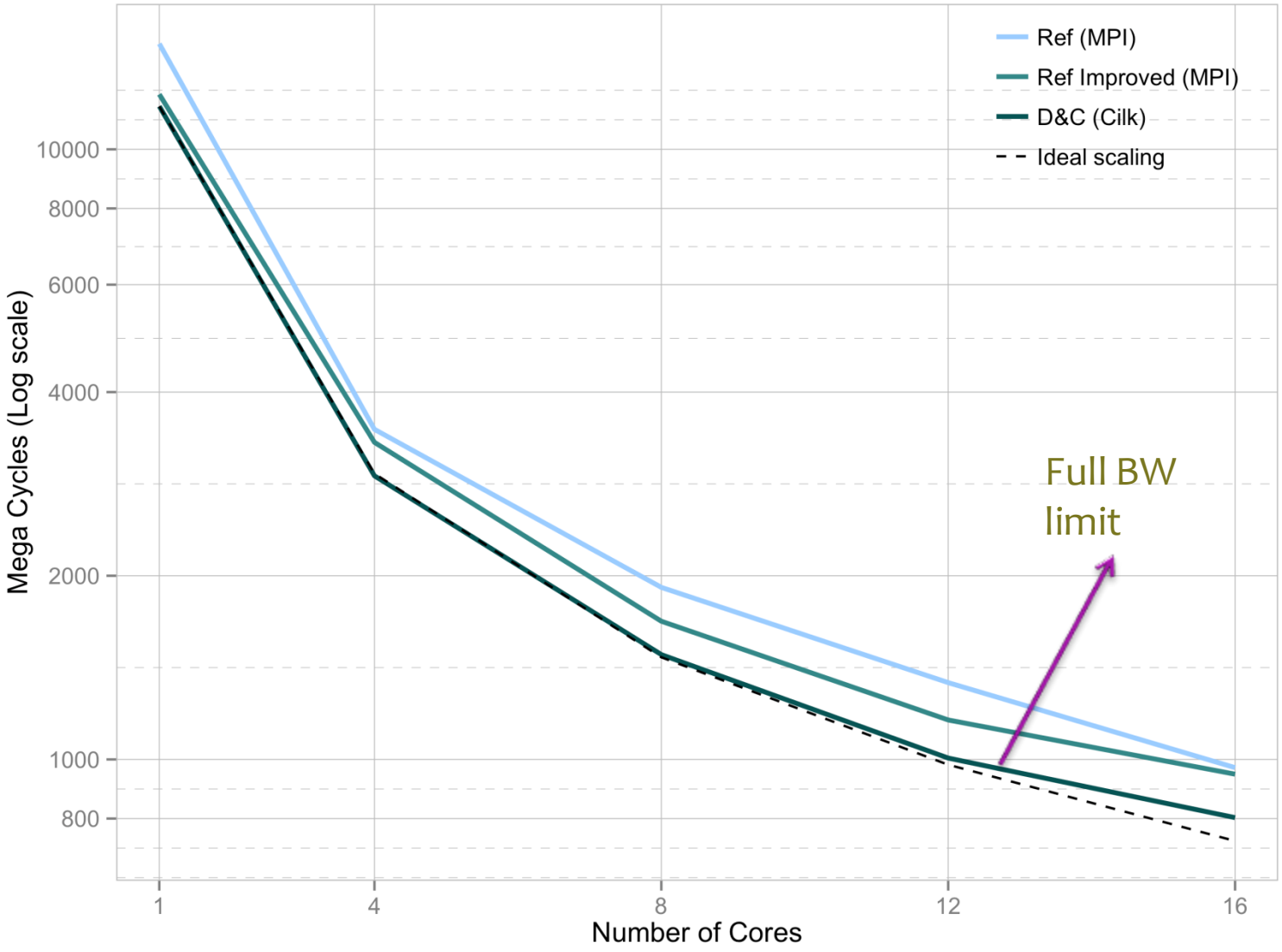






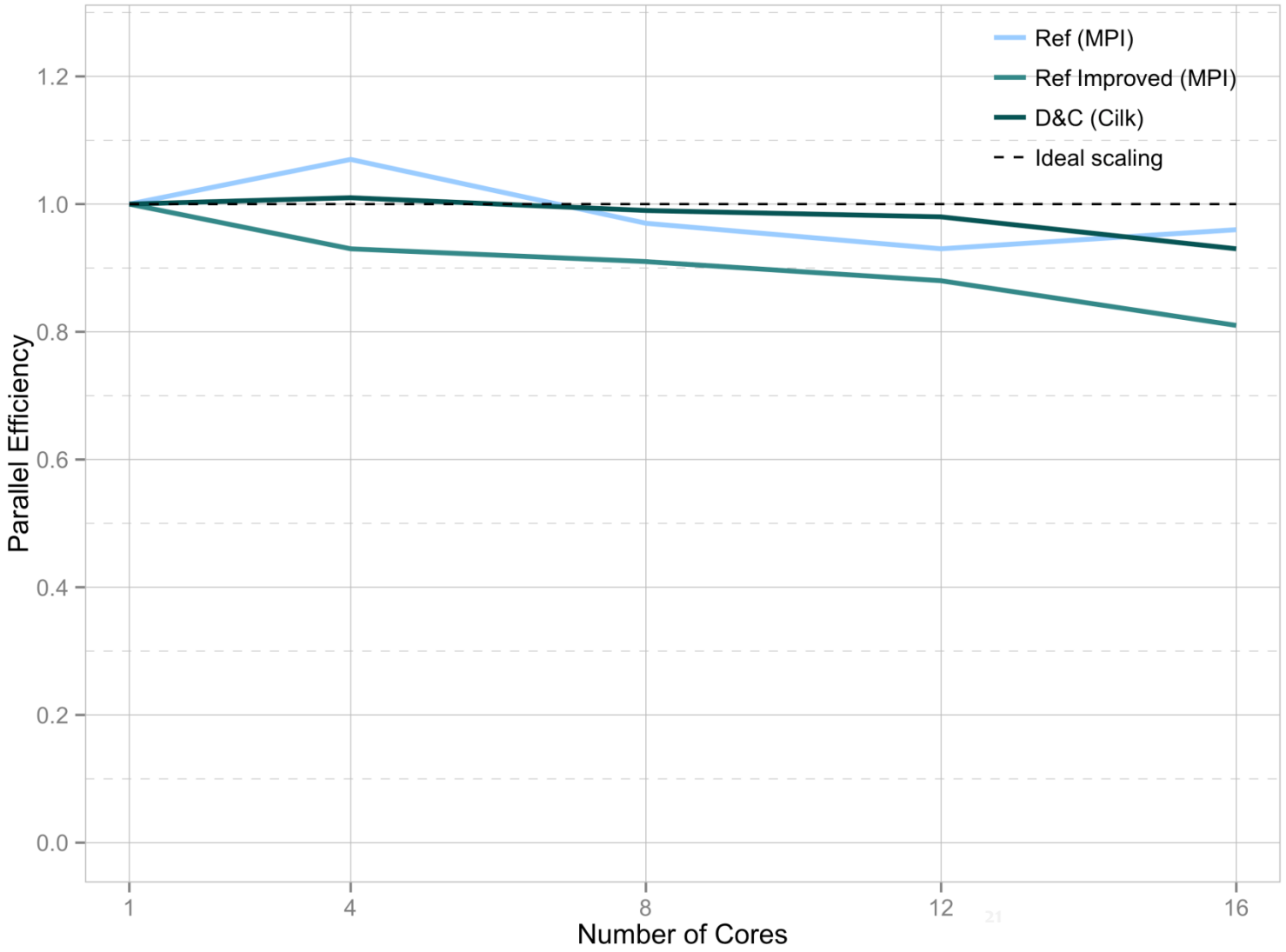
Results: Execution Time

2x8 cores Intel Sandy Bridge E5-2665, 2.4GHz, 64GB RAM memory



Results: Parallel Efficiency

2x8 cores Intel Sandy Bridge E5-2665, 2.4GHz, 64GB RAM memory



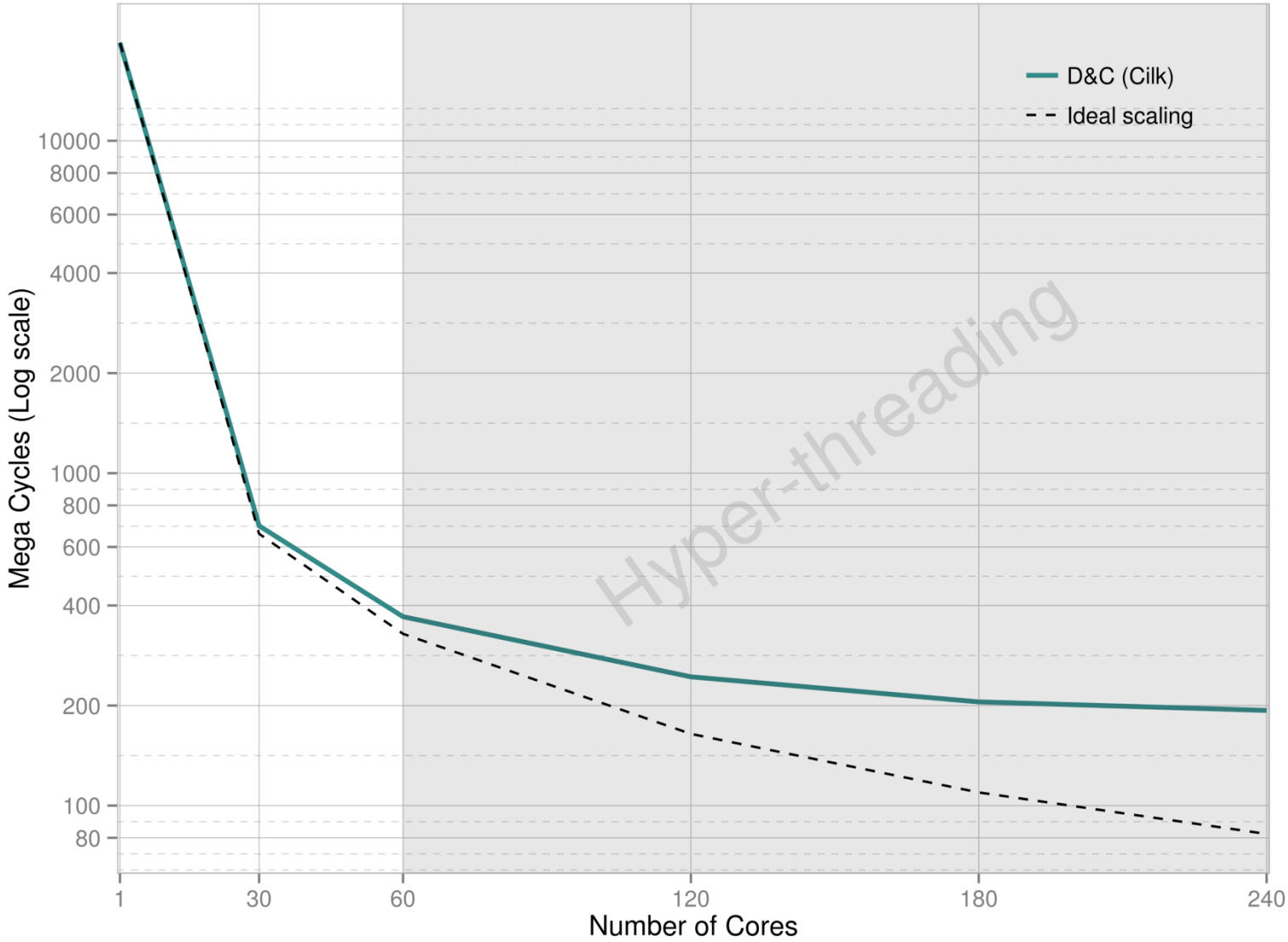
Results: Xeon Phi

Work in progress: preliminary results !

It works out of the box!!!
And it works well ! => 1 PHI@1Ghz ~ 16 SNB@2.4Ghz

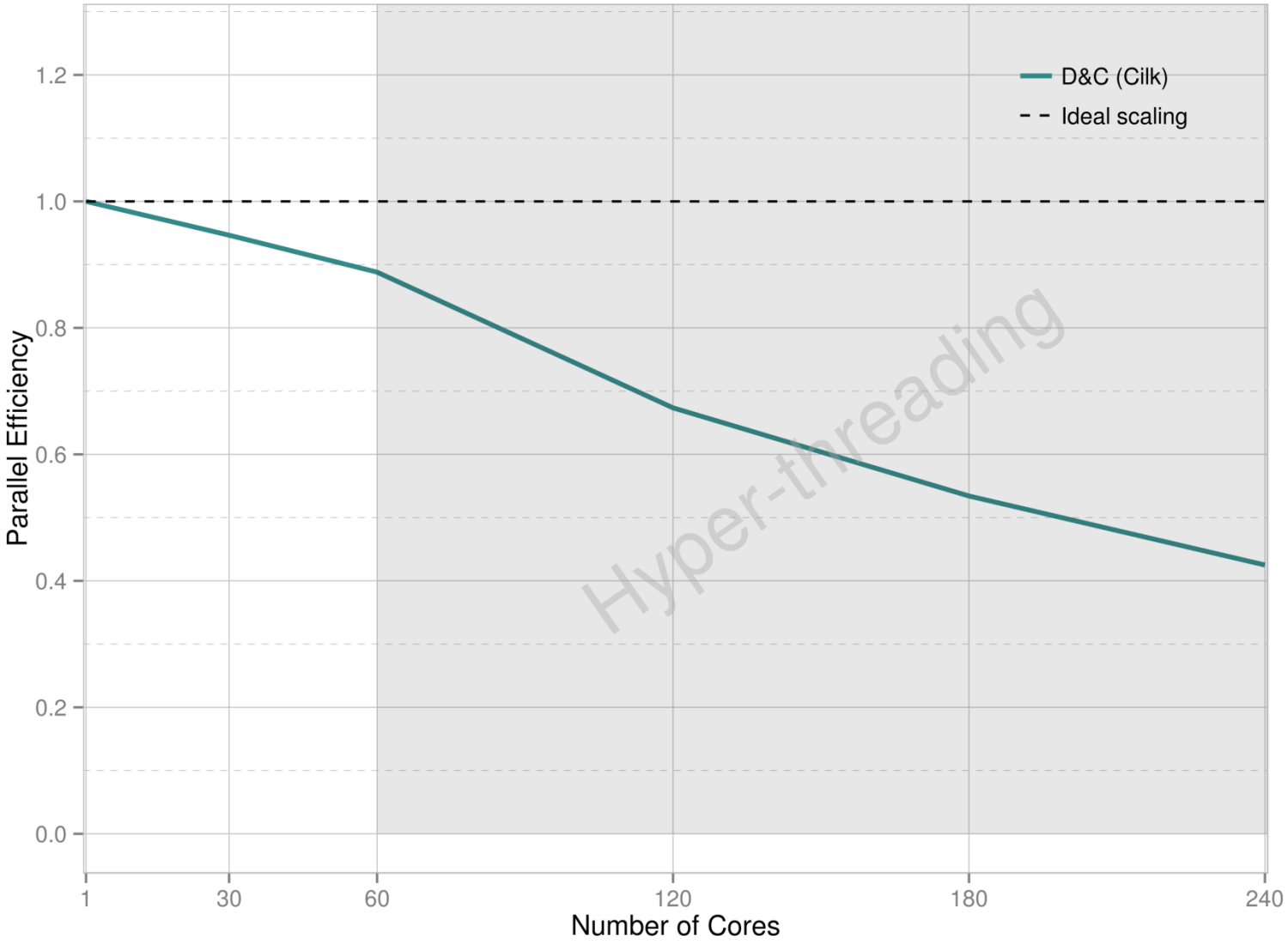
Results: Phi Execution Time

Work in progress: preliminary results ! => not to compare to slide on SNB in this presentation, not the same computation (precomputed connectivity)



Results: Phi Efficiency

Work in progress: preliminary results !



Conclusion 1/2

- The experiment on the DA test-case is successfull and ready to be used in DEFMESH production code.
- Since march the approach start be experimented in their main CFD code.
- Interesting integration process with all the parallelisation in C++ and the physics in FORTRAN
- A proto-app to demonstrate the D&C assembly to the community is ready to be open-sourced.

Conclusion 2/2

- Future works:
  - D&C friendly matrix format (CSB?) and SPMV, SPMM, SPTMM, SPMMM
  - Impact of the partitionner (scotsh, metis)
  - Use D&C at the distributed level
    - Using GASPI/MPI3.0
  - Shared memory parallelized solvers
  - Studying dynamic load balancing
    - Independant work
    - Tree based (D&C)
    - Symmetric execution (MIC/Host, starPU? Xkaapi?)

() The Proto-Application Concept

- Aka mini-app, proxy-app (NERSC trinity, Argonne CESAR…)

- Objectives: Reproduce at scale the behavior of a set of HPC applications and support the development of optimizations that can be translated into the original applications
- Easier to execute, modify and re-implement

- If you cannot make the application open-source, you can at least open-source the problems.
  - Support community engagement
  - Reproducible and comparable results
  - Interface with application developers

Thank you all for your attention !

- Question?