

New Technical Features

openCFS User Meeting 2026

Fabian Wein¹

Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Mathematics

April 9, 2026

- in the .xml we can define additional named nodes and elements
- for a virtual coordinate the closest node (element barycenter) is searched
- for `list` the `inc` attribute needs to be small enough to capture all entities
- due to inefficient original implementation, the runtime could have exploded (but is fast now)

```
<domain geometryType="plane">
  ...
  <nodes name="coord">
    <coord x=".15" y=".1" />
  </nodes>

  <nodes name="list">
    <list>
      <freeCoord comp="y" start=".3" stop=".4" inc="0.025"/>
      <fixedCoord comp="x" value=".3"/>
    </list>
  </nodes>
</domain>
```

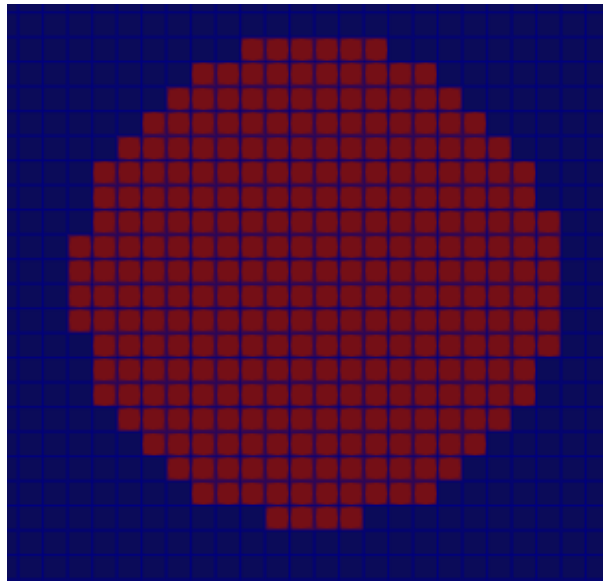
- bounds is a clearer expression for lines/rectangles/volumes
- the attribute eps is optional and has a default of $1 \cdot 10^{-9}$
- negligible runtime cost as the mesh is traversed only once and each entity is tested

```
<nodes name="line">  
  <bounds eps="1e-9">  
    <fixed comp="x" value=".3" />  
    <range comp="y" min=".3" max=".4" />  
  </bounds>  
</nodes>
```

```
<nodes name="box">  
  <bounds>  
    <range comp="x" min=".4" max=".8" />  
    <range comp="y" min=".3" max=".8" />  
  </bounds>  
</nodes>
```

- when we test each entity, we can also make it generic
- a circle of radius $\sqrt{.015} = 0.122$

```
<elems name="condition">  
  <test condition="(x-.2)^2 + (y-.8)^2 lt .015" />  
</elems>
```

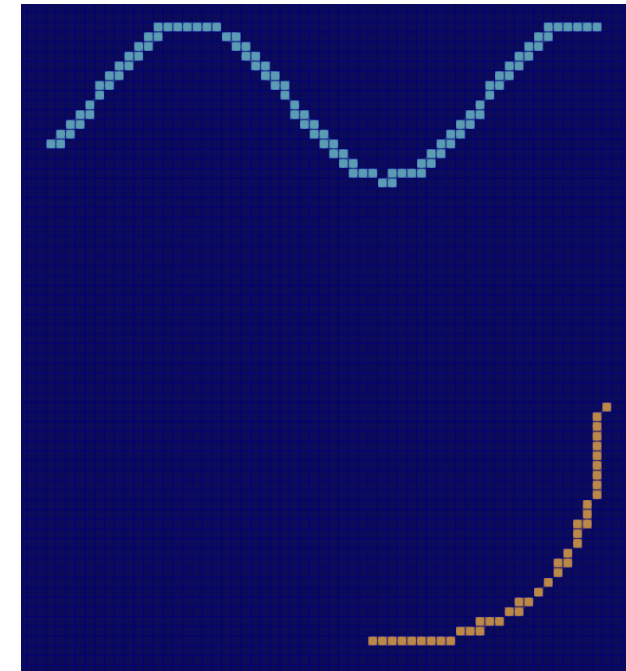


User Defined Nodes/Elements: expression

- generating virtual coordinates and search for each one the closest entity
- fast due to kd-tree implementation (nanoflann replaces FLANN in openCFS)
- the `comp="u"` is an auxiliary variable (here the $t \in [0; 1]$ for a Bézier curve)
- for elements take care that given coordinates are not exactly between barycenters

```
<elems name="expression_func">
  <expression>
    <sample comp="x" start="0.201" stop=".901" inc=".0025" />
    <eval comp="y" value="-.1*sin(4*pi*x) + .8" />
  </expression>
</elems>

<elems name="expression_u">
  <expression>
    <sample comp="u" start="0" stop="1" inc=".01" />
    <eval comp="x" value="(1-u)^2*0.60001 + 2*(1-u)*u*0.9 + u^2*0.90001" />
    <eval comp="y" value="(1-u)^2*0.10001 + 2*(1-u)*u*0.1 + u^2*0.40001" />
  </expression>
</elems>
```



From the godfather of computer science

Premature optimization is the root of all evil

— Donald E. Knuth



In my humble opinion

- the worst we can have in openCFS is hard to read code which is not maintainable
- changes or bug hunting will likely cost much more lifetime than saved runtime
- code shall be optimized only when there are validating tests
- code shall be optimized only when we can measure there is an improvement
- typical example for backfiring optimization: parallelization of short code

Performance benchmarks










- the new gitlab repository <https://gitlab.com/openCFS/performance> collects samples
- just add relevant samples like in the test suite and add your reference .info.xml
- no automatism, no validation, just a collection of samples to share and refer to
- a runtime of few seconds makes it easier to test optimizations and measure improvements
- could be the base for future automated performance regression tests (→ Dominik)

Timers

- timers are written to .info.xml. There are subtimers which can have parents
- `shared_ptr<Timer> timer = info->Get("stuff/timer")->AsTimer();
timer->Start();`
- analyze with `performance.py my.info.xml`

Profiling

- `CFS_PROFILING=ON` adds `-g -fno-omit-frame-pointer`
- use the profiling tool of your platform (gprof, vtune, ...)

3.34 s	8,9 %	0 s		▼ CoupledField::CFS::Run() cfs
2.31 s	6,1 %	0 s		▼ CoupledField::CFS::SolveProblem() cfs
1.87 s	5,0 %	0 s		▼ CoupledField::Domain::PostInit(unsigned int) cfs
1.87 s	5,0 %	0 s		▼ CoupledField::SingleDriver::InitializePDEs() cfs
1.87 s	5,0 %	0 s		▼ CoupledField::Domain::InitPDEs(unsigned int) cfs 
1.34 s	3,6 %	0 s		▼ CoupledField::StdPDE::DefineAlgSys() cfs
744.00 ms	2,0 %	0 s		> CoupledField::Assemble::SetupMatrixGraph(int, int) cfs
285.00 ms	0,8 %	2.00 ms		> CoupledField::AlgebraicSys::GraphSetupDone() cfs

Inefficient algorithms

- SimInputMesh (27 MB): 3.28 sec → 0.09 sec
- user defined nodes/elements: 2.04 sec → 0.01 sec

Too many heap allocations

- heap allocations (new/delete) requires thread safe memory management of blocks by the OS
- the stack space is for free but limited in size and to function call
- `std::map`, `std::set`, `std::unordered_set`, ... have one heap allocation per element
- boost has flat variants which benefit from `.reserve()` but cannot be applied everywhere
- proper `Reserve()` before `Push_back()` can have huge impact
- use `std::array<TYPE, N>` for small fixed-size array on stack (no heap allocation)
- `MapEdges()/MapFaces()`: 1.2 sec → 0.22 sec with the same algorithm

- `boost::container::small_vector<TYPE, N>` starts with stack and dynamically extends to heap
- now `Vector` has initially 3 elements on the stack (for coordinates)
- now `Matrix` has initially space for 6×6 elements ($14e6 \rightarrow 192$ heap allocations)
- run `cfs` with `-d` to see number and size (in elements) of remaining `Matrix` heap allocations
- for critical sections, **if there is a real benefit**, wrap `Vector/StdVector` with external memory:
`std::array<double, 100> a; StdVector<double> v(a);`
- wrapping of external memory is refactored for `Vector` and `StdVector`:
 - `Vector`: no `Resize()` or `Push_back()` allowed
 - `StdVector`: `Resize()` and `Push_back()` allowed as long as we stay in external memory capacity
- BTW: why use `StdVector` and not `std::vector`? `std::vector` has no easy range check

- here compile time is relevant for life time of developers :)
- ClangBuildAnalyzer creates build statistics. Add `-ftime-trace` to `CMAKE_CXX_FLAGS`
- compilation of 370 .cc files: parsing (frontend): 538.1 sec, codegen & opts (backend): 275.1 sec
- removing includes (especially boost) from headers saves $\approx 20\%$ compile time (in progress)

Hint: Reuse Precompiled cfsdepscache

- have a local `.cfs_platform_defaults.cmake` in your home directory with

```
set(CFS_DEPS_CACHE_DIR_DEFAULT "$ENV{HOME}/code/cfsdepscache")
```

- CFS_FSANITIZE=ON adds `-fsanitize=address,undefined`
- configure with advanced (hidden) CMake option `CFS_FSANITIZE_OPTIONS`
- other options: `thread`, `leak`, ...

Story of a bug

- with new Vector based on three elements on stack, `VolumeElementConversion_CGNS` failed
- but only with `gcc14-musl` in the CI pipeline: "file content not as expected"
- running locally the test with `cfs` compiled with `fsanitize` revealed:

```
... ERROR: AddressSanitizer: heap-buffer-overflow ... READ of size 4 ... thread T0
#0 ... CoupledField::SimInputCGNS::ReadUnstructuredZone(int, int) :1048
#1 ... CoupledField::SimInputCGNS::ReadMesh(CoupledField::Grid*) :655
... allocated by thread T0 here: ...
#2 ... std::vector<unsigned int, .... vector.h:957
#3 ... CoupledField::SimInputCGNS.cc:92
```

- points to issue: `numElemsOfDim_.resize(3)` but the readers uses 1-based dimension

- w.r.t. AI coding, we are all beginners. Let's share our experience and learn together.
- I use the Visual Code extension GitHub Copilot Chat with Github Pro:
 - Github Pro \$10/month + max \$20/month Copilot premium requests
 - favorite AI: Claude Sonnet 4.6, the free GPT-4.1 is almost useless
- copilot code completion assistance is almost as annoying as helpful
- sometimes the chat can find bugs by providing two h5dumps of .cfs results
- perfect for pasting C++ compile issues
- I almost always start the chat with *don't change code*. Examples:
 - *check the current function for potential issues*
 - *change the loop to C++11 range based loop*
 - *could I use a const ref of the shared pointer here?*
 - *analyze the clang build analyzer output and make suggestions*
- half of the time, the suggestions are simply wrong and would produce bad code
- note, that the AI reproduces legacy style in the code (Double, iterators, ...)

- PlainMechanic3D (60x60x60 mesh), MacBook Pro M1 (2021), 8 threads, Feb. 2026

TIMER (sec)	:	cnt	:	WALL	~	CPU	:	
total	:		:	14.30 [100.0%]	~	44.28 [100.0%]	:	3.1
init_static	:	1	:	5.08 [35.6%]	~	6.27 [14.2%]	:	1.2
solve_ginkgo	:	1	:	4.15 [29.0%]	~	28.86 [65.2%]	:	7.0
grids	:	1	:	4.09 [28.6%]	~	4.08 [9.2%]	:	
* graph_setup	:	1	:	3.49 [24.4%]	~	4.70 [10.6%]	:	1.3
assemble	:	2	:	0.61 [4.3%]	~	4.67 [10.5%]	:	

- current state of runtime optimization:

total	:		:	7.55 [100.0%]	~	36.74 [100.0%]	:	4.9
solve_ginkgo	:	1	:	4.00 [53.0%]	~	28.55 [77.7%]	:	7.1
init_static	:	1	:	1.75 [23.2%]	~	2.49 [6.8%]	:	1.4
* -graph	:	1	:	1.17 [15.5%]	~	1.91 [5.2%]	:	1.6
hdf5	:	1	:	0.59 [7.8%]	~	0.59 [1.6%]	:	
assemble	:	2	:	0.59 [7.8%]	~	4.44 [12.1%]	:	
grid	:	2	:	0.24 [3.2%]	~	0.24 [0.7%]	:	
* -readMesh	:	1	:	0.09 [1.2%]	~	0.09 [0.2%]	:	
* mapFacesEdges	:	2	:	0.21 [2.8%]	~	0.21 [0.6%]	:	