

# Architecture of Modern FE Code

with OOP

---

Theodore Chang, Ph.D.

initial: July 30, 2021

revised: March 30, 2025

# Objective

---

# Objectives

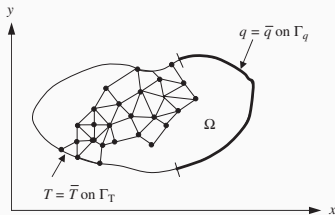
- to report the architecture of **suanPan**
  - a parallel FEA package
  - written in modern C++ (11-20)
  - based on shared memory model
  - with distributed memory model support
- to discuss improvements regarding HPC in simulation
  - potential alternatives
  - possible extensions to other platforms
- spoiler: no 'new' stuff

## Background

---

# Typical FEA Flow

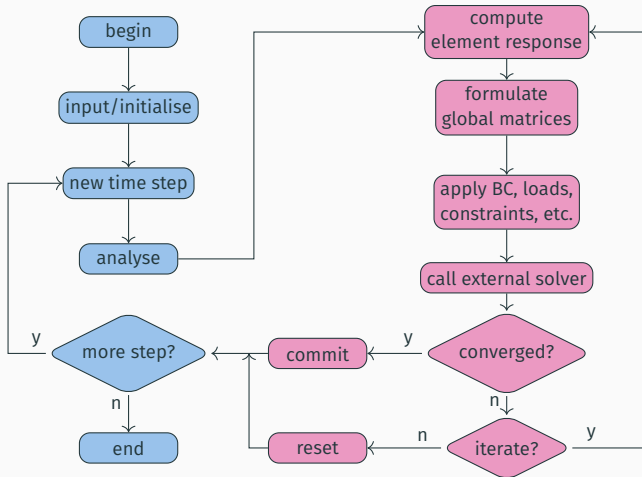
To simulate a continuum problem by FEM, the following tasks are required.



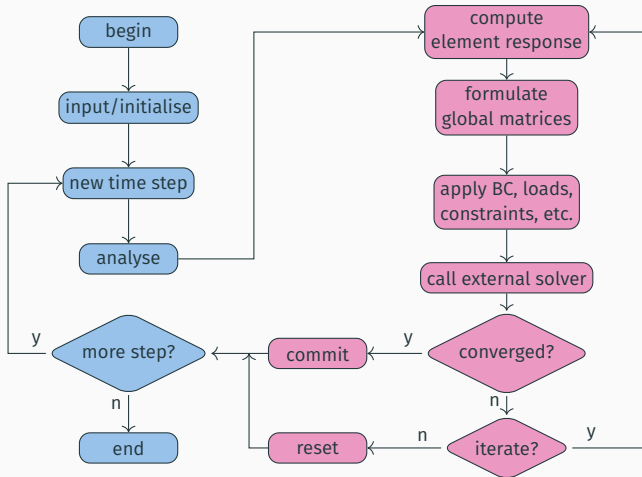
credit: Fish & Belytschko, 2007

- discretise the geometry with nodes and elements
- compute elemental stiffness and resistance (local)
- formulate global stiffness and residual
- apply boundary conditions, loads, constraints
- solve for solution

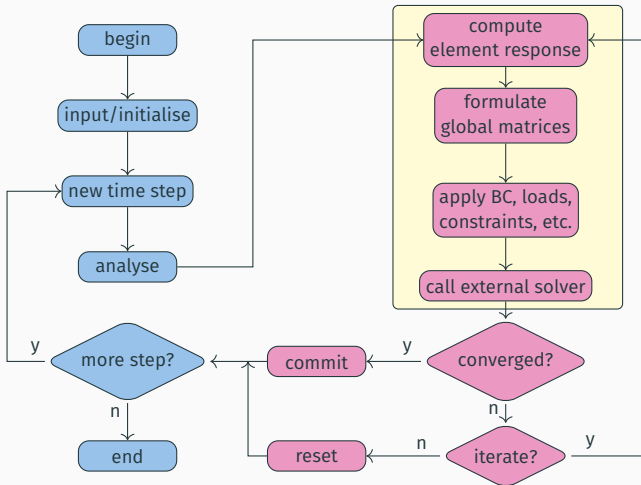
# Typical FEA Flow



# Typical FEA Flow



# Typical FEA Flow



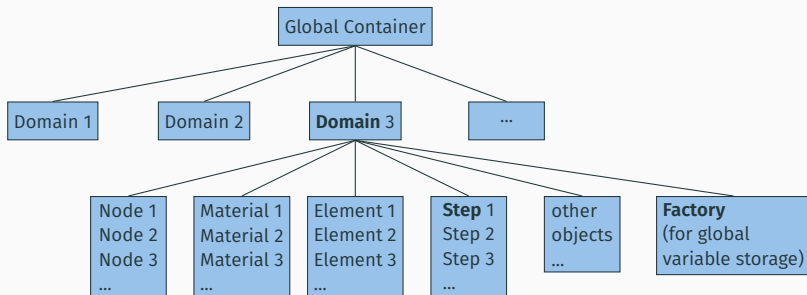


## **Model Establishment and Storage**

---

# Structure

A complete OO style is used, data are stored in objects arranged in a tree structure.



Multiple domains can coexist. Sub-structuring for distributed memory model based parallelisation is possible. Similar to ABAQUS, multiple analysis steps can be defined in each domain.

# Domain

- represents a problem
- the centralised container that provides storage for all nodes, elements, materials, etc.
- two stages: construction and initialisation

```
1 class Domain {
2     shared_ptr<LongFactory> factory; // use double precision
3
4     StepQueue step_pond;
5
6     AmplitudeStorage amplitude_pond;
7     ConstraintStorage constraint_pond;
8     ElementStorage element_pond;
9     MaterialStorage material_pond;
10    NodeStorage node_pond;
11    SectionStorage section_pond;
12    SolverStorage solver_pond;
13 public:
14     // public methods to initialise/update/assemble, etc.
15 };
```

# Storage

`concurrent_unordered_map` from `tbb` or `ppl` is used for concurrent insertion/lookup, etc. Once an object is constructed, no memory reallocation would occur. Minimum memory operation.

```
1  template<typename T> class Storage {
2      vector<shared_ptr<T>> fish; // some funny variable names
3      concurrent_unordered_map<unsigned, shared_ptr<T>,
4          ↪ std::hash<unsigned>> pond;
5      // ...
6  };
7
8  using AmplitudeStorage = Storage<Amplitude>;
9  using ConstraintStorage = Storage<Constraint>;
10 using ElementStorage = Storage<Element>;
11 using IntegratorStorage = Storage<Integrator>;
12 // ...
```

Random access iterator (for easier parallelisation) is provided by `std::vector<shared_ptr<T>>`.

# State Updating

---

compute  
element response

formulate  
global matrices

apply BC, loads,  
constraints, etc.

call external solver

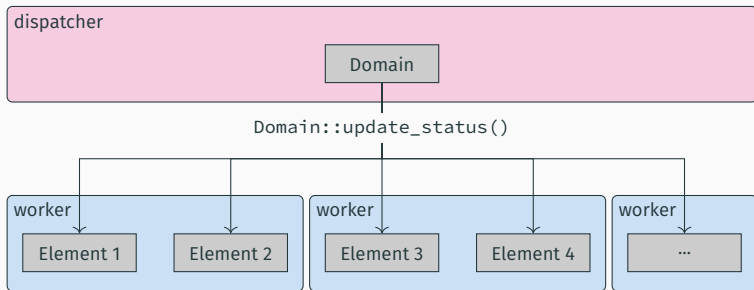


# Element State Updating

Each element updates its state via the attached material object based on trial nodal displacement vector.

$$\mathbf{U} \implies \boldsymbol{\varepsilon} \implies \boldsymbol{\sigma} \implies \mathbf{R}_e \implies \mathbf{K}_e$$

multiple-read-no-write, no racing, lock free, can be safely parallelised via `parallel_for_each`



# Element State Updating

Element can acquire trial displacement from either locally stored Node pointers (easy for shared memory model) or Domain (easy for distributed memory model).

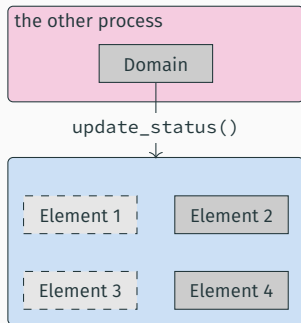
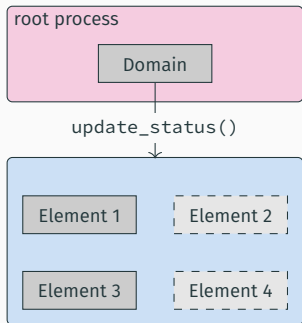
Linear algebra packages are configured so that only one thread is used for small matrices to ensure no nested parallelisation. Default behaviour of MKL.

```
1 int Domain::update_trial_status() const {  
2     // for nodes  
3     parallel_for_each(node_pool.cbegin(), node_pool.cend(), [&](const shared_ptr<Node>& t_node) {  
4         ↪ t_node->update_trial_status(trial_displacement, trial_velocity, trial_acceleration); });  
5     // for elements  
6     parallel_for_each(element_pool.cbegin(), element_pool.cend(), [&](const shared_ptr<Element>&  
7         ↪ t_element) { t_element->update_status(); });  
8     // ...  
9 }
```



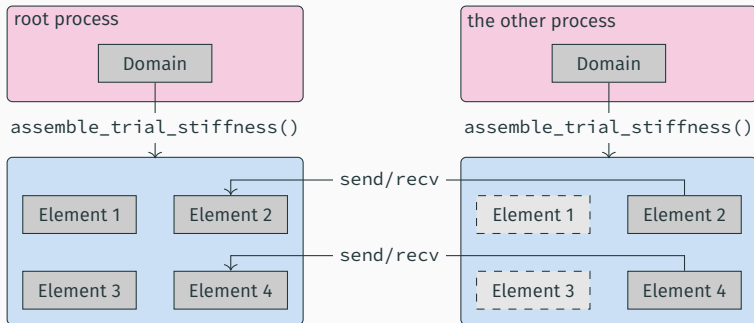
## Distributed Approach A— Scattering

A process ID is assigned to each element, in the following, elements 1 and 3 are assigned to the root process, elements 2 and 4 are assigned to the other process. When `update_status()` is called, each process updates the elements assigned to itself, and skips all other elements.



# Distributed Approach A – Gathering

To collect elemental resistance/stiffness, all other processes send data to the root process. By the end of this communication, the root process contains all elemental quantities stored in local objects. They hold the final results, which may not be computed by themselves (sent over from remote objects).

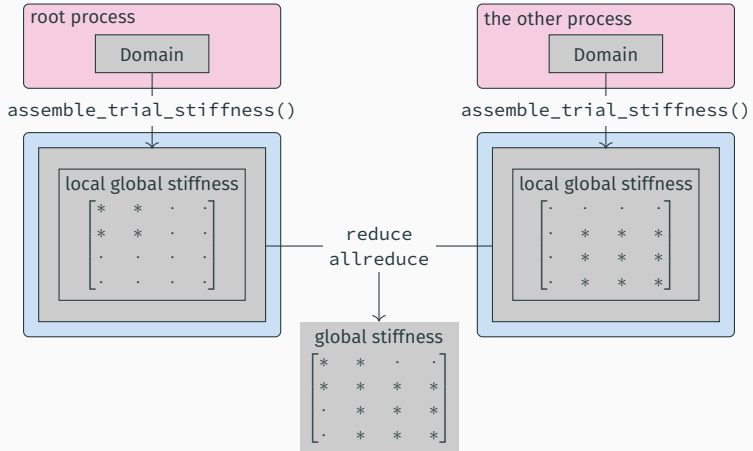


## Distributed Approach A — Traits

- relatively large memory footprint on each node
- minimal data transfer
- no temporary memory (de)allocation
- no (de)serialisation of objects
- simple P2P communication pattern (only bcast, send, recv)
- **a large number (proportional to the number of elements) of small messages**

# Distributed Approach B — Gathering

Elemental resistance/stiffness can be assembled locally. Only local copies of global matrices/vectors will be reduced.



## Distributed Approach B — Traits

- relatively large memory footprint on each node
- no temporary memory (de)allocation
- no (de)serialisation of objects
- simple collective communication pattern (only bcast, reduce/allreduce)
- **minimal number (proportional to the number of processes) of large messages, less concerning because latency matters**

# Element Class

Non-const static variables shall be avoided due to potential racing.

```
1  class Element : protected ElementData {  
2  protected:  
3      vector<weak_ptr<Node>> node_ptr; // node pointers  
4  
5      [[nodiscard]] mat get_coordinate(unsigned) const;  
6  
7      [[nodiscard]] vec get_incre_displacement() const;  
8      [[nodiscard]] vec get_trial_displacement() const;  
9      [[nodiscard]] vec get_current_displacement() const;  
10  
11     [[nodiscard]] vec get_node_incre_resistance() const;  
12     [[nodiscard]] vec get_node_trial_resistance() const;  
13     [[nodiscard]] vec get_node_current_resistance() const;  
14 public:  
15     // ...  
16 };
```

Auxiliary methods can be implemented to hide details.

# Element Class

A general purpose data set is provided.

```
1 struct ElementData {
2     const uvec node_encoding; // node encoding
3     const uvec material_tag; // material tags
4
5     mat initial_mass; // mass matrix
6     mat initial_damping; // damping matrix
7     mat initial_stiffness; // stiffness matrix
8     mat trial_mass; // mass matrix
9     mat trial_damping; // damping matrix
10    mat trial_stiffness; // stiffness matrix
11    mat current_mass; // mass matrix
12    mat current_damping; // damping matrix
13    mat current_stiffness; // stiffness matrix
14    vec trial_resistance; // resistance vector
15    vec current_resistance; // resistance vector
16    vec trial_damping_force; // damping force
17    vec current_damping_force; // damping force
18    // ...
19 };
```

# Element Class

```
1 // header
2 class C3D20 final : public MaterialElement3D {
3     struct IntegrationPoint final {
4         double weight;
5         unique_ptr<Material> c_material;
6         mat strain_mat;
7     };
8     // ...
9     vector<IntegrationPoint> int_pt;
10 };
11
12 // implementation
13 int C3D20::update_status() {
14     trial_stiffness.zeros(c_size, c_size);
15     trial_resistance.zeros(c_size);
16
17     for(const auto& I : int_pt) {
18         I.c_material->update_trial_status(I.strain_mat * get_trial_displacement());
19         trial_stiffness += I.weight * I.strain_mat.t() * I.c_material->get_trial_stiffness() *
20             ↪ I.strain_mat;
21         trial_resistance += I.weight * I.strain_mat.t() * I.c_material->get_trial_stress();
22     }
23
24     return SUANPAN_SUCCESS;
25 }
```

$K = \sum B^T E B$ . Expressive code with high performing lazy evaluation.



# Material Class

Similar to `Element`, if pre-defined data set is used, state is managed internally and automatically.

Only need to provide the method to compute stress  $\sigma$  and stiffness  $E$  for given strain  $\epsilon$ . No external data dependency. Similar to the `UMAT` subroutine in `ABAQUS`.

For linear elastic response,  $\sigma = E\epsilon$ .

```
1  int Elastic1D::update_trial_status(const vec& t_strain) {  
2      trial_strain = t_strain;  
3      trial_stiffness = elastic_modulus;  
4      trial_stress = trial_stiffness * trial_strain;  
5      return SUANPAN_SUCCESS;  
6  }
```

compute  
element response

formulate  
global matrices

apply BC, loads,  
constraints, etc.

call external solver



# Loads and Constraints

Consider the optimisation problem:

$$\text{minimize } W(\mathbf{u})$$

$$\text{subject to } \mathbf{c}(\mathbf{u}) = \mathbf{0}$$

In which  $W$  is the strain energy function,  $\mathbf{c}$  contains  $n$  constraints.

The Lagrange function can be constructed as

$$\mathcal{L}(\mathbf{u}, \boldsymbol{\lambda}) = W(\mathbf{u}) + \boldsymbol{\lambda} \cdot \mathbf{c}(\mathbf{u}).$$

The stationary point can be obtained

$$\nabla \mathcal{L}(\mathbf{u}, \boldsymbol{\lambda}) = \mathbf{0}, \quad \longrightarrow \quad \begin{cases} \mathbf{R} + \boldsymbol{\lambda} \cdot \nabla \mathbf{c} = \mathbf{0}, \\ \mathbf{c} = \mathbf{0}. \end{cases}$$

In which  $\mathbf{R} = \nabla W$  is the resistance.

## Loads and Constraints

Consider a nonlinear context, the system shall be iteratively solved. The effective stiffness is then

$$J = \begin{bmatrix} K & \cdot \\ \cdot & \cdot \end{bmatrix} + \begin{bmatrix} \nabla^2 c & \nabla c^T \\ \nabla c & \cdot \end{bmatrix} = \begin{bmatrix} K + K_c & K_b^T \\ K_b & \cdot \end{bmatrix}$$

$K_b$  is known as border matrix. Often, it is **sparse**. The size of global stiffness changes due to the presence of constraints.

Since the number of constraints is not known in advance, memory reallocation is required, which is not preferred.

Noting  $K_c$  is similar to elemental stiffness  $K_e$  and can be assembled into  $K$ , it is possible to store  $K_b$  separately and static condensation can be used to solve the system. The additional cost is only forward/backward substitution as  $K + K_c$  should have been factorised.

# Loads and Constraints

Constraints can be updated and assembled concurrently. The same constraint can have different number of multipliers during different phases of analysis. Loads can be handled in the same manner.

```
1 // ...
2 auto counter = 0;
3 for(auto& I : constraint_pond.get()) {
4     const auto m_size = I->get_multiplier_size();
5     if(!I->is_initialized() || 0 == m_size) continue;
6     const auto e_size = counter + m_size - 1;
7     t_resistance.subvec(counter, e_size) = I->get_auxiliary_resistance();
8     t_load.subvec(counter, e_size) = I->get_auxiliary_load();
9     t_stiffness.cols(counter, e_size) = I->get_auxiliary_stiffness();
10    counter += m_size;
11 }
12 // ...
```

# Global Assembly

---

compute  
element response

formulate  
global matrices

apply BC, loads,  
constraints, etc.

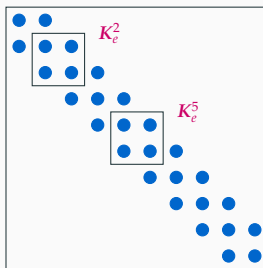
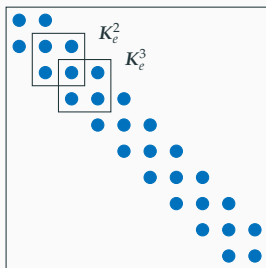
call external solver



# Global Matrix Assembly

Assembling elemental stiffness  $K_e$  into global stiffness  $K$  is often done sequentially due to data overlapping.

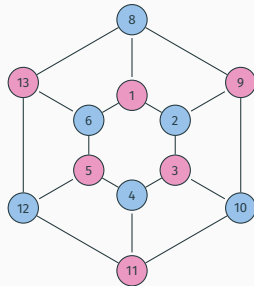
Mutex may be expensive. Instead, the reliable  $k$ -coloring algorithm can be used. The earliest reference dates back to 1982.





# Global Matrix Assembly

- initialisation
  1. construct the graph
  2. color the graph
  3. store the coloring scheme
- state updating
  1. loop over all color groups
  2. update all elements in the same group



```
1 void Domain::assemble_trial_stiffness() const {
2     std::for_each(color_map.begin(), color_map.end(), [&](const vector<unsigned>& color) {
3         tbb::parallel_for_each(color.begin(), color.end(), [&](const unsigned tag) {
4             factory->assemble_stiffness(get_element(tag)->get_trial_stiffness(),
5                 ↪ get_element(tag)->get_dof_encoding());
6         });
7     });
8 }
```

The coloring algorithm can be slow. Maybe try parallel coloring?

Pre-coloring the model leads to a lock-free algorithm for matrix assembly with the dense storage.

What about the sparse storage?

- Use COO format and pre-allocate memory blocks,  $K_e$  can be copied into  $K$  in parallel. Since  $K_e$  spans contiguous memory, the copy operation may be automatically vectorised by compiler.
- Use out-of-core solver (e.g., MUMPS) so no need to assemble global stiffness matrix  $K$ .

For distributed memory model, can sub-structuring be done based on coloring?

# Solving

---

compute  
element response

formulate  
global matrices

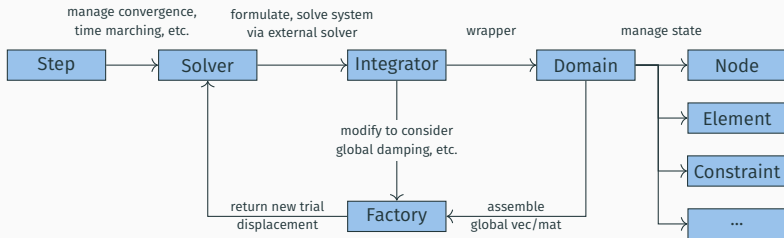
apply BC, loads,  
constraints, etc.

call external solver

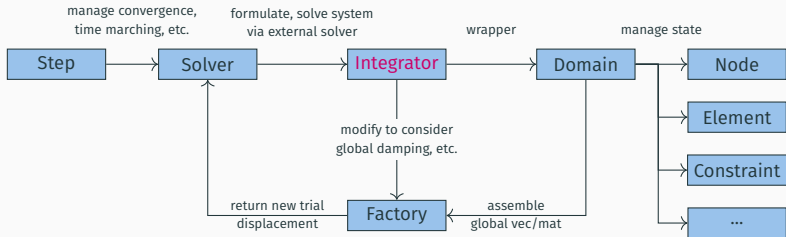


Multiple steps can be defined, they'll be analysed sequentially.

```
1 for(const auto& t_domain : domain_pool)
2     if(t_domain->is_active())
3         for(const auto& t_step : t_domain->get_step_pool()) {
4             // ...
5             if(SUANPAN_FAIL == t_step->Step::initialize()) return SUANPAN_FAIL;
6             if(SUANPAN_FAIL == t_step->initialize()) return SUANPAN_FAIL;
7             if(SUANPAN_FAIL == t_step->analyze()) return SUANPAN_FAIL;
8         }
```



# Analysis Logic



- Solver does not directly communicate with data storage Domain and Factory
- Factory does not directly communicate with local data storage Element, etc.
- Different operations can be injected via overloading.

# System Solving

Solver implements different solving algorithms such as (modified) Newton, BFGS, displacement control, arc length control, etc.

```
1  int Newton::analyze() {
2      // ...
3      // G is an Integrator
4      while(true) {
5          G->assemble_resistance(); // in parallel
6          G->assemble_matrix();     // in parallel
7          G->process_load();         // in parallel
8          G->process_constraint();   // in parallel
9
10         G->solve(ninja, G->get_force_residual()); //  $X = \text{inv}(A) * B$ 
11
12         G->update_trial_displacement(ninja); // sync global in parallel
13         G->update_trial_status();           // sync local in parallel
14
15         if(C->is_converged()) return SUANPAN_SUCCESS; // exit if converged
16         if(++counter > max_iteration) return SUANPAN_FAIL;
17     }
18 }
```

# Analysis Type

Basic utility methods in Domain can be invoked in parallel on demand to fulfil the desired task.

$$\tilde{K} = K + c_0 M + c_1 C$$

```
1 // Newmark is derived from Integrator
2 void Newmark::assemble_matrix() {
3     const auto& D = get_domain().lock(); // weak_ptr
4     auto& W = D->get_factory();
5
6     auto fa = std::async([&]() { D->assemble_trial_stiffness(); });
7     auto fb = std::async([&]() { D->assemble_trial_mass(); });
8     auto fc = std::async([&]() { D->assemble_trial_damping(); });
9     fa.get();
10    fb.get();
11    fc.get();
12
13    // this addition is also parallelised
14    W->get_stiffness() += C0 * W->get_mass() + C1 * W->get_damping();
15 }
```



# Matrix Storage Scheme

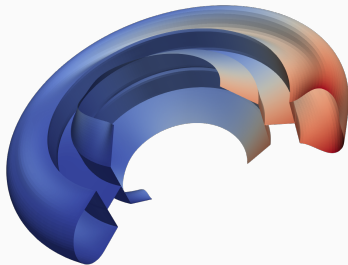
- in-house matrix container that supports:
  - dense (full, banded, symm./asymm.)
  - sparse (COO, CSC, CSR)
- fully decoupled from the FE model
- easy to add new solvers, override `MetaMat::solve()` method
- both double and mixed precision solving strategy
- currently available:
  - OpenBLAS
  - MKL
  - SPIKE
  - ARPACK
  - FEAST
  - SuperLU
  - MUMPS
  - CUDA
- distributed memory model based parallelisation is managed by external solvers

# Benchmark

---

# Benchmark

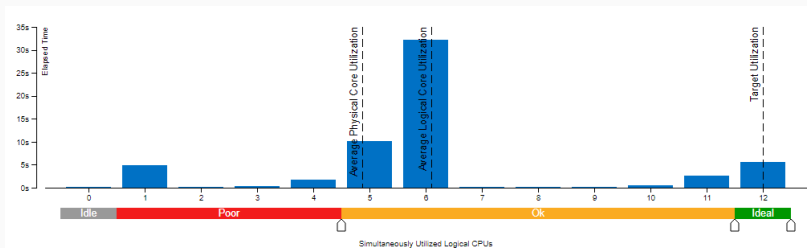
- i7-8700 (6C12T) with DDR4-2666
- about 120 GFLOPS (pure DGEMM)
- 120744 DoFs
- 39990 three-node shell elements
- linear elastic material
- 40 complete analysis cycles
- solved by DPBSV



# Benchmark

with default MKL configurations

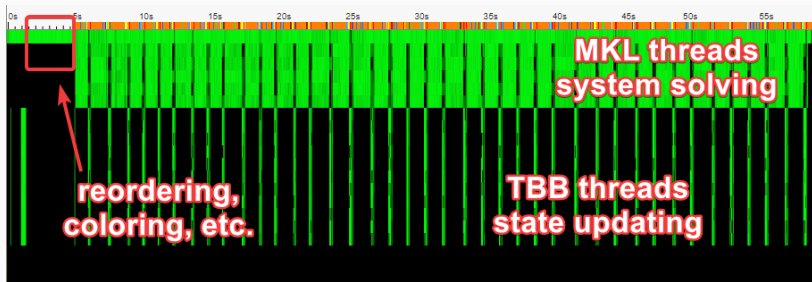
- 105 to 107 GFLOPS overall
- additional 10 GFLOPS excluding initialisation
- 80 % CPU utilisation



# Benchmark

with default MKL configurations

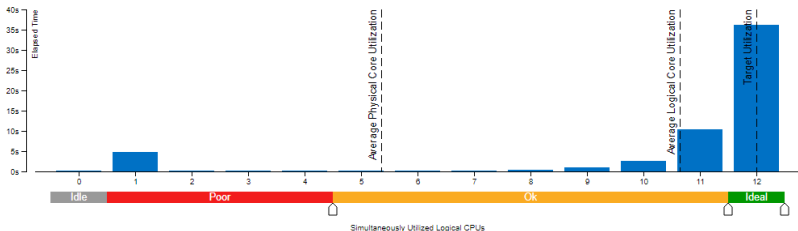
- 80 % CPU utilisation
- 105 to 107 GFLOPS overall
- additional 10 GFLOPS excluding initialisation



# Benchmark

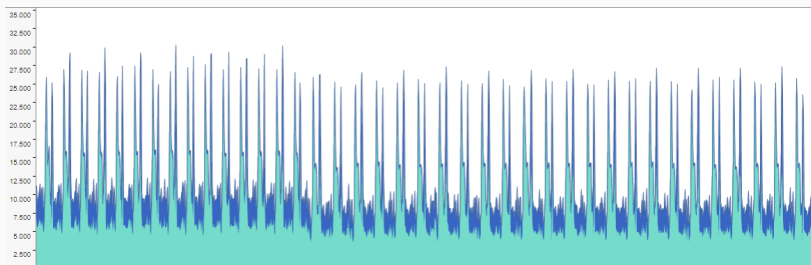
manually override MKL defaults, not optimal but slightly faster

- MKL\_DYNAMIC=FALSE
- MKL\_NUM\_THREADS=12
- 90 % CPU utilisation
- 110 GFLOPS overall
- 120 GFLOPS excluding initialisation

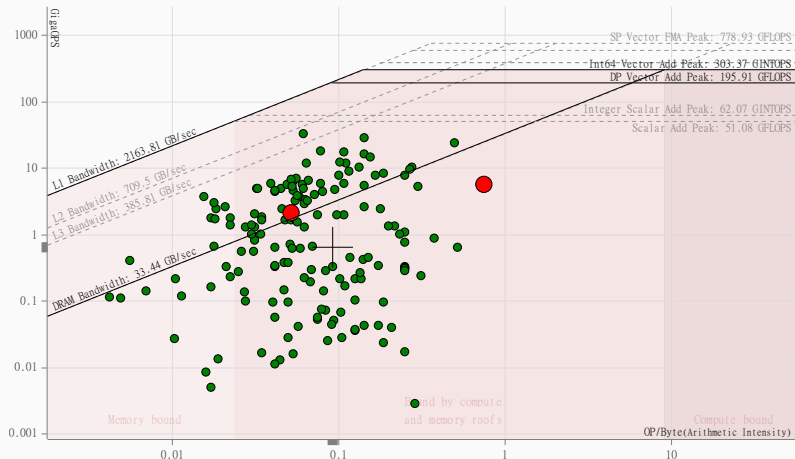


# Benchmark

about memory bounded (benchmark value  $25 \text{ GB s}^{-1}$ )



# Benchmark





- relatively flat analysis logic
- task based parallelisation
- minimum data dependency
- expressive syntax and lazy evaluation (Armadillo)
- highly extensible

## Some Thoughts

- scatter arbitrary objects over arbitrary nodes in a cluster?
- unified shared memory (USM) based OOP?
  - local interface, remote data, remote computation

**Thank you!**