Architecture of Modern FE Code

with OOP

Theodore Chang, Ph.D.

initial: July 30, 2021
revised: March 21, 2025

Objective

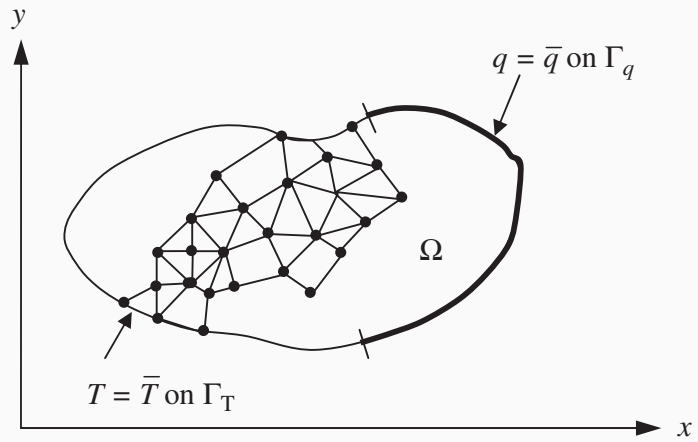
Objectives

- 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

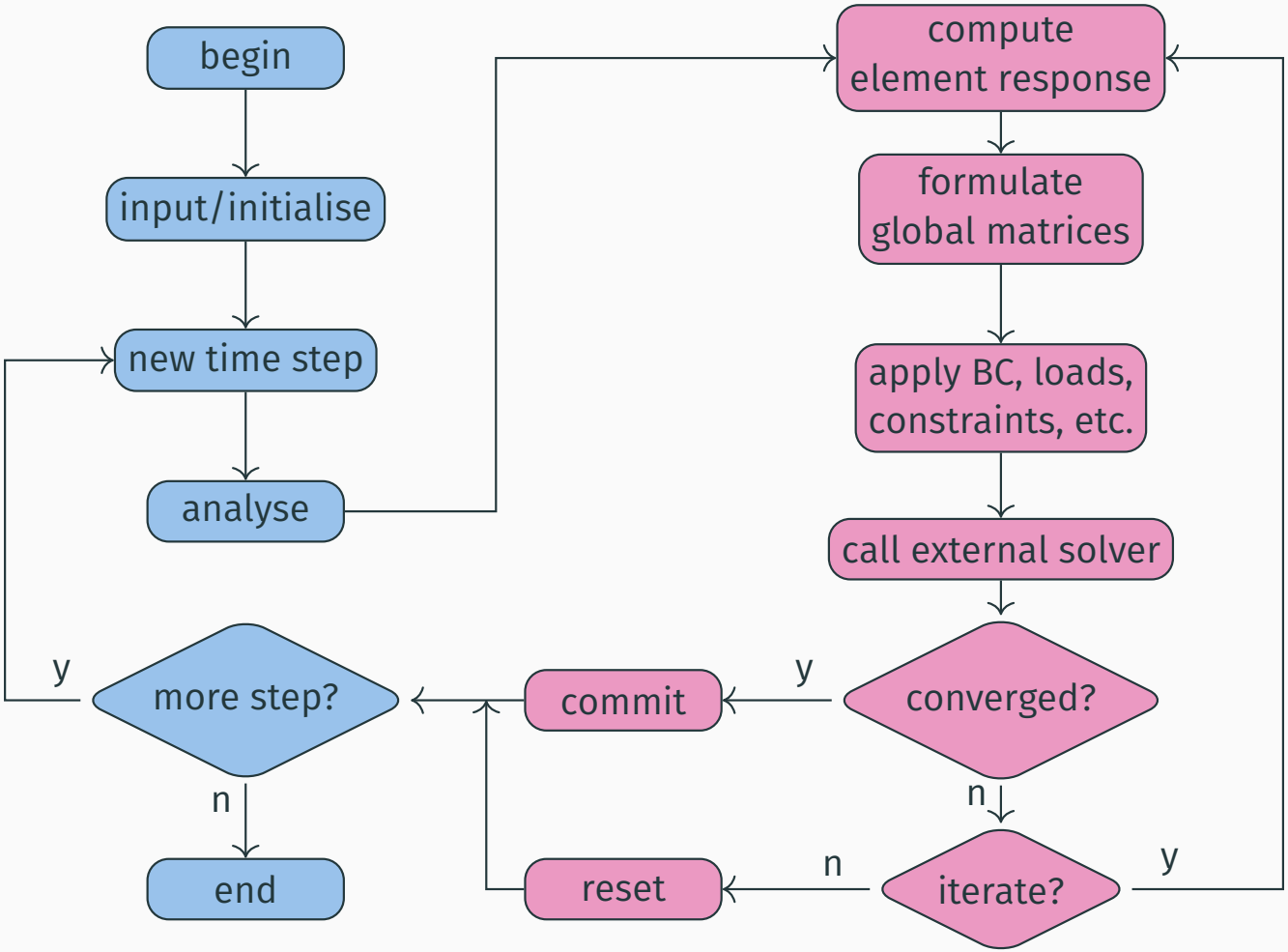
To simulate a continuum problem by FEM, the following tasks are required.



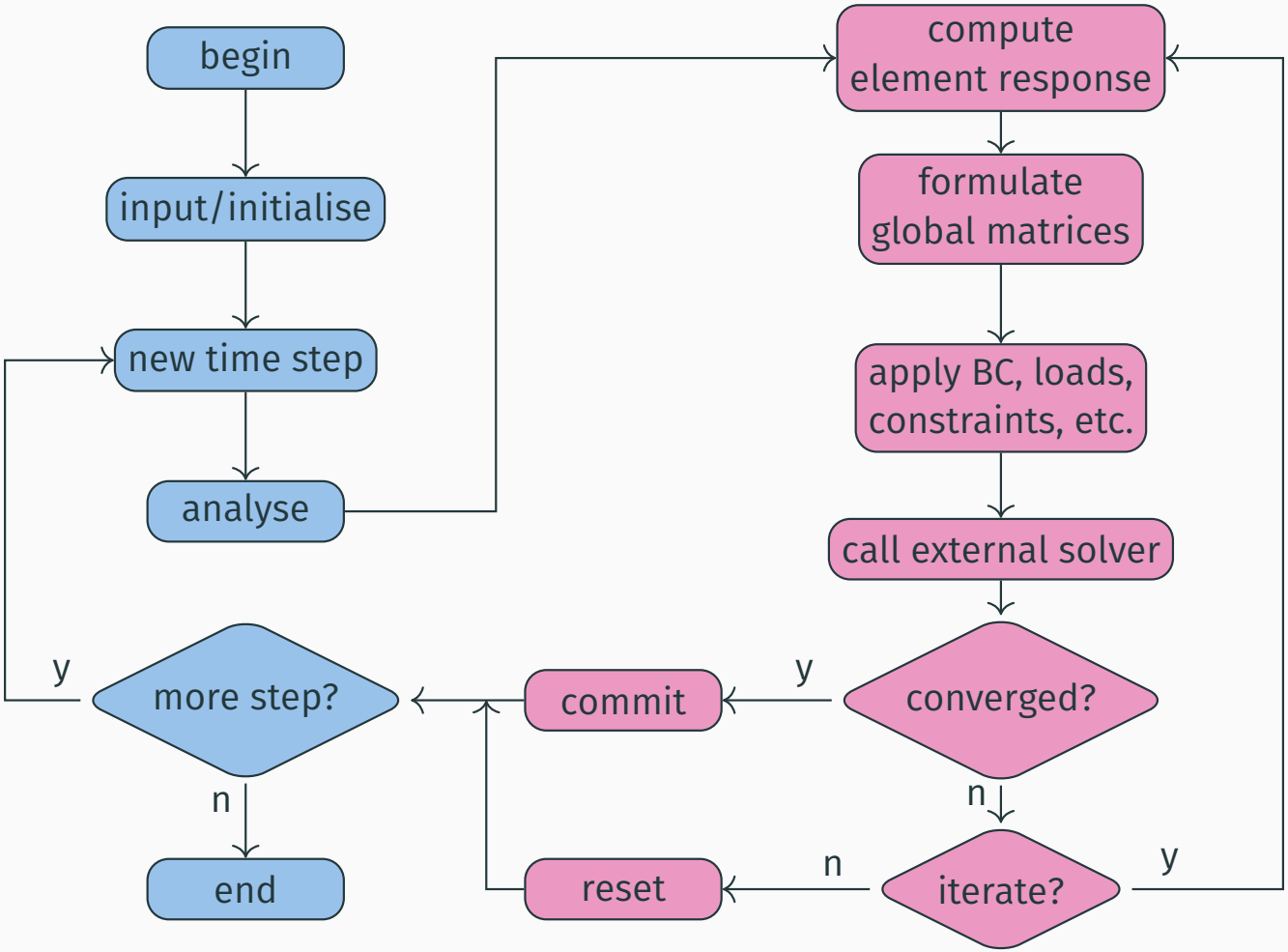
credit: Fish & Belytschko, 2007

- 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

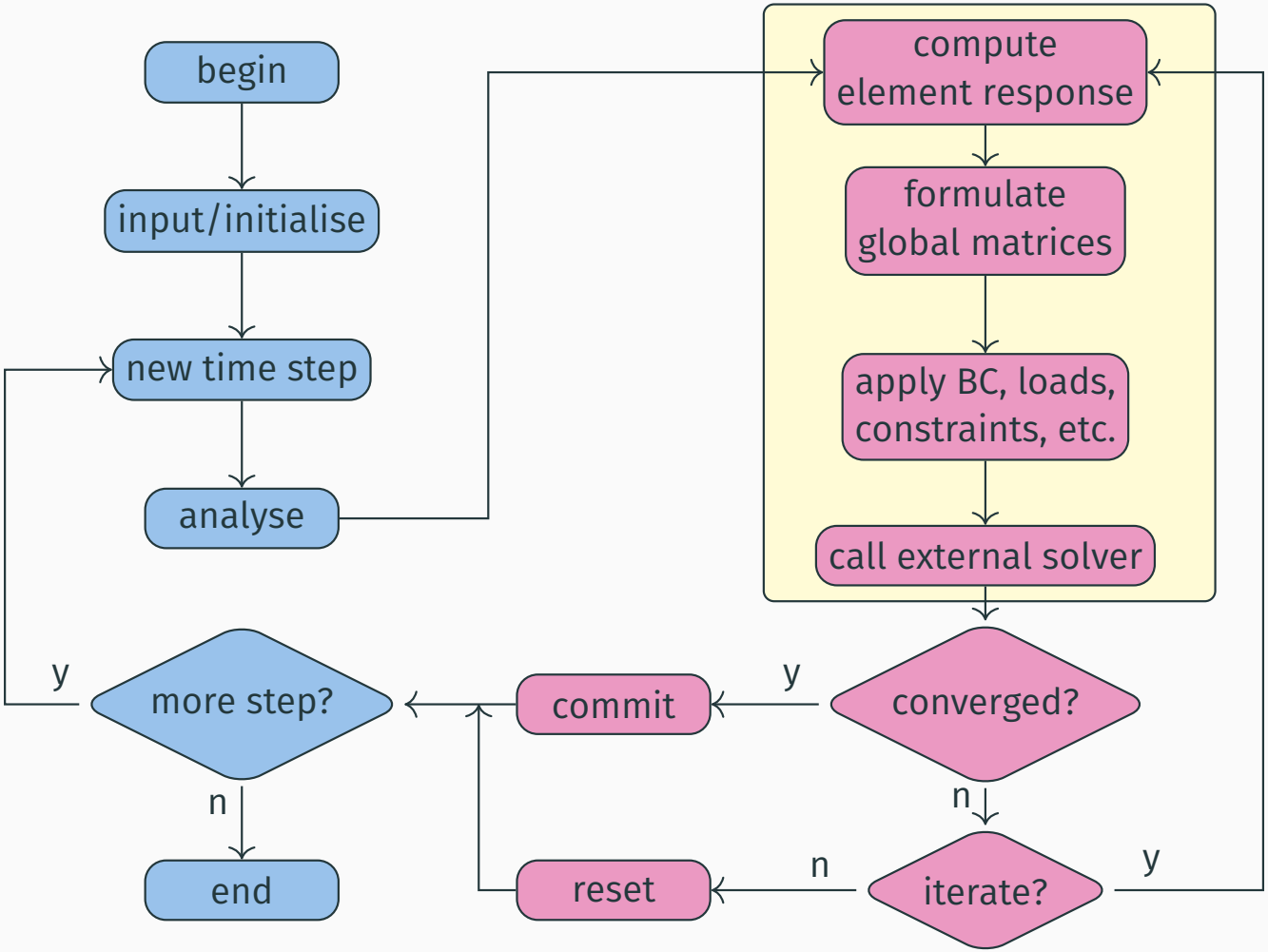
Typical FEA Flow



Typical FEA Flow



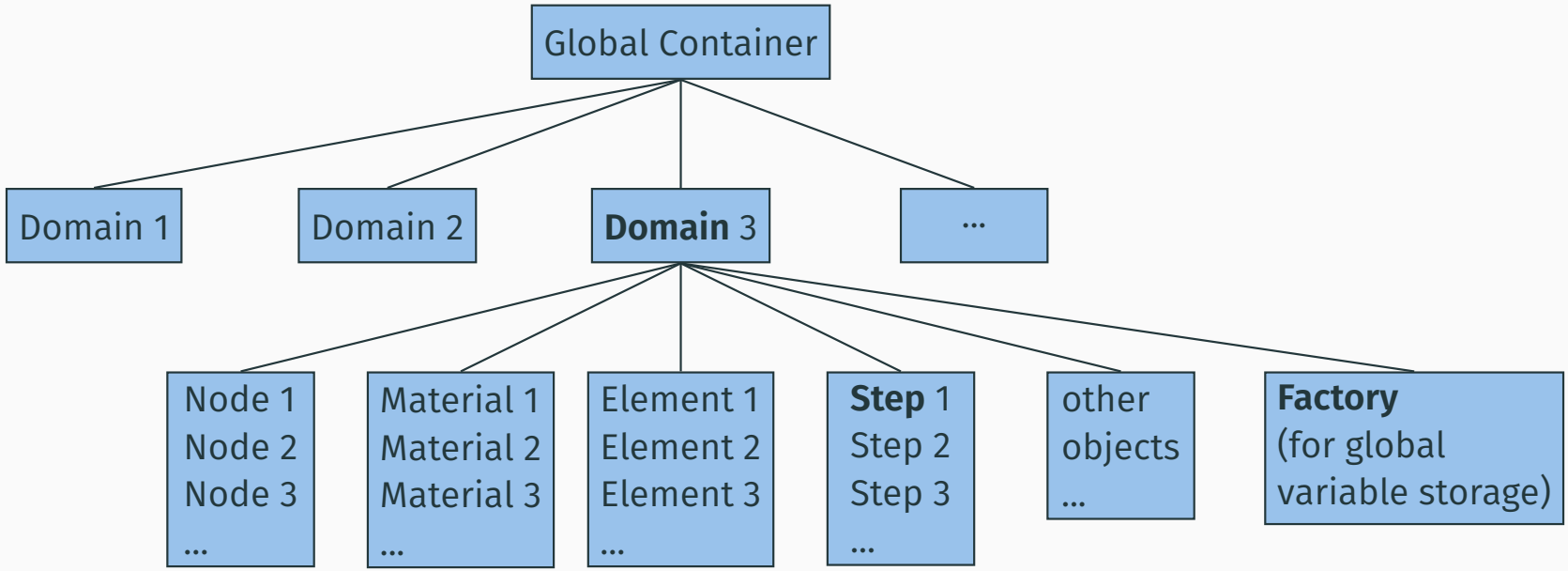
Typical FEA Flow



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 };
```

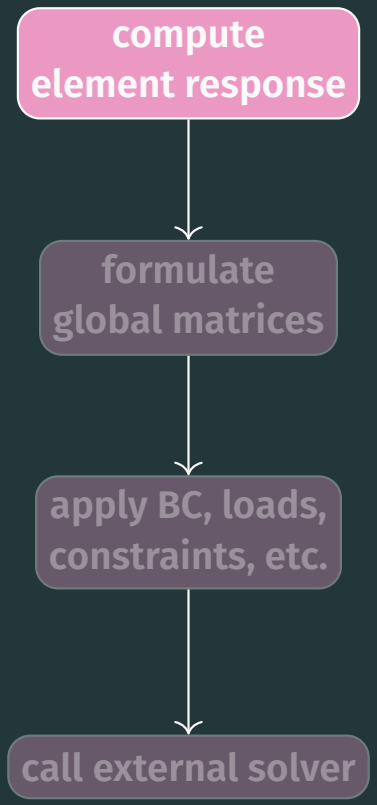
Storage

`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>,
       ↪  std::hash<unsigned>> pond;
4      // ...
5  };
6
7  using AmplitudeStorage = Storage<Amplitude>;
8  using ConstraintStorage = Storage<Constraint>;
9  using ElementStorage = Storage<Element>;
10 using IntegratorStorage = Storage<Integrator>;
11 // ...
```

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

State Updating

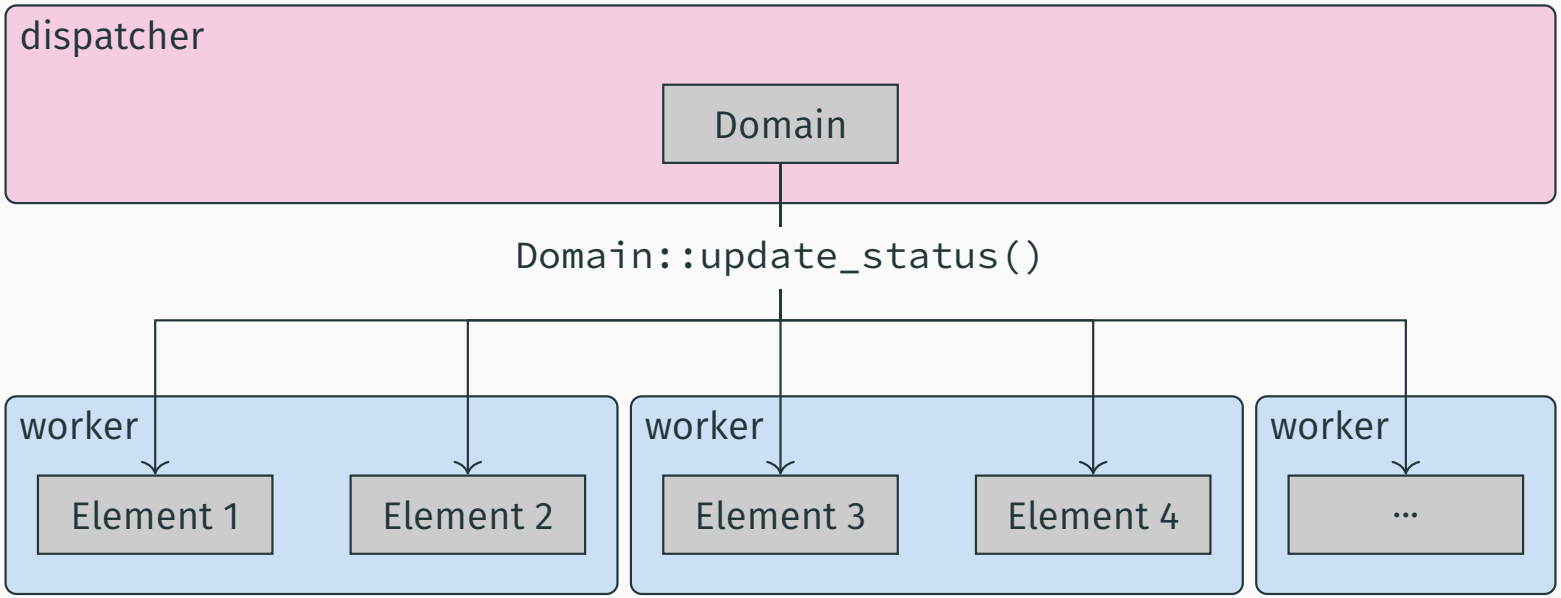

compute
element response
formulate
global matrices
apply BC, loads,
constraints, etc.
call external solver

Element State Updating

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

$$\boldsymbol{U} \Longrightarrow \boldsymbol{\varepsilon} \Longrightarrow \boldsymbol{\sigma} \Longrightarrow \boldsymbol{R}_e \Longrightarrow \boldsymbol{K}_e$$

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



Element State Updating

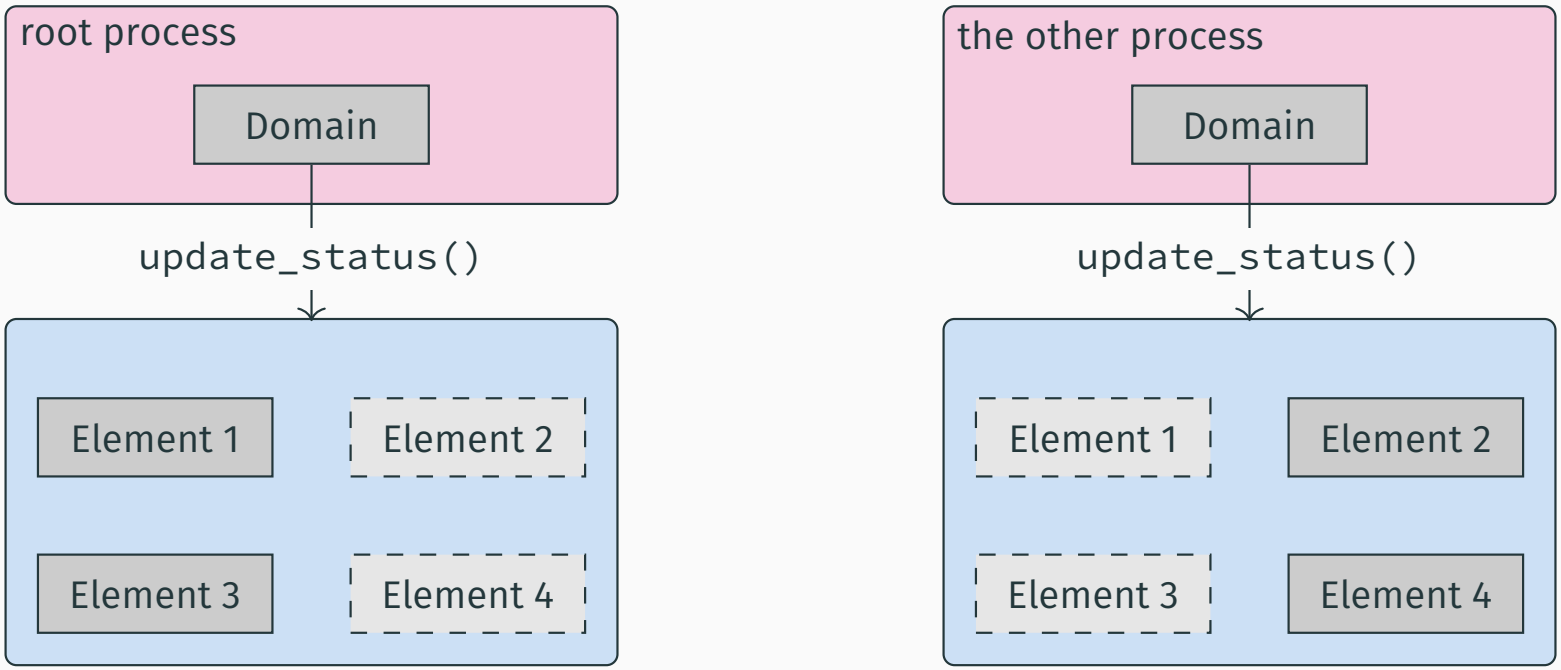
`Element` can acquire trial displacement from either locally stored `Node` pointers (easy for shared memory model) or `Domain` (easy for distributed memory model).

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) {
       ↪  t_node->update_trial_status(trial_displacement, trial_velocity, trial_acceleration); });
4      // for elements
5      parallel_for_each(element_pool.cbegin(), element_pool.cend(), [&](const shared_ptr<Element>&
       ↪  t_element) { t_element->update_status(); });
6      // ...
7  }
```

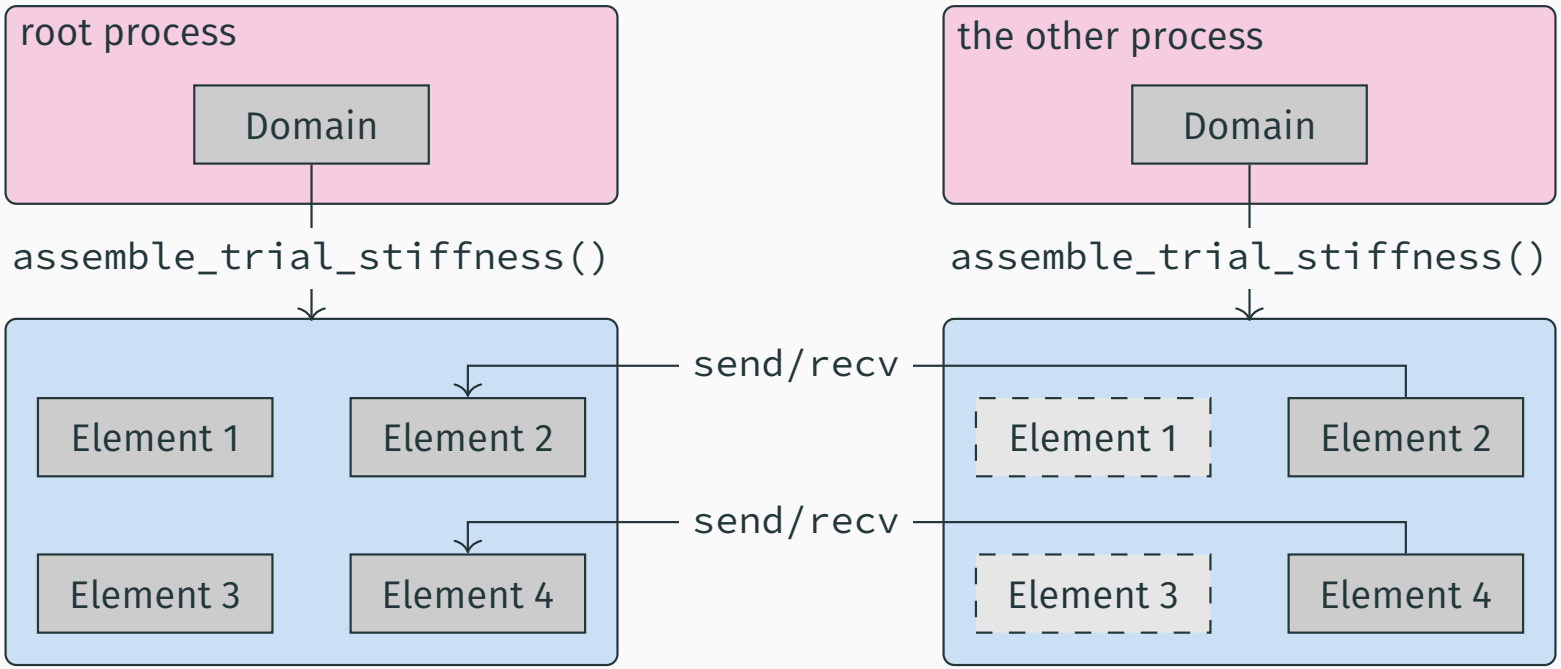
Distributed Approach — Scattering

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 — 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 — Traits

- relatively large memory footprint on each node
- minimal data transfer
- no temporary memory (de)allocation
- no (de)serialisation of objects
- simple communication pattern (only `bcast`, `allreduce`, `send`, `recv`)

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.

```
struct ElementData {
    const uvec node_encoding; // node encoding
    const uvec material_tag;  // material tags

    mat initial_mass;      // mass matrix
    mat initial_damping;   // damping matrix
    mat initial_stiffness; // stiffness matrix
    mat trial_mass;      // mass matrix
    mat trial_damping;   // damping matrix
    mat trial_stiffness; // stiffness matrix
    mat current_mass;      // mass matrix
    mat current_damping;   // damping matrix
    mat current_stiffness; // stiffness matrix
    vec trial_resistance;       // resistance vector
    vec current_resistance;     // resistance vector
    vec trial_damping_force;    // damping force
    vec current_damping_force;  // damping force
    // ...
};
```

Element Class

```cpp
// header
class C3D20 final : public MaterialElement3D {
    struct IntegrationPoint final {
        double weight;
        unique_ptr<Material> c_material;
        mat strain_mat;
    };
    // ...
    vector<IntegrationPoint> int_pt;
};

// implementation
int C3D20::update_status() {
    trial_stiffness.zeros(c_size, c_size);
    trial_resistance.zeros(c_size);

    for(const auto& I : int_pt) {
        I.c_material->update_trial_status(I.strain_mat * get_trial_displacement());
        trial_stiffness += I.weight * I.strain_mat.t() * I.c_material->get_trial_stiffness() *
        ↪  I.strain_mat;
        trial_resistance += I.weight * I.strain_mat.t() * I.c_material->get_trial_stress();
    }

    return SUANPAN_SUCCESS;
}
```

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

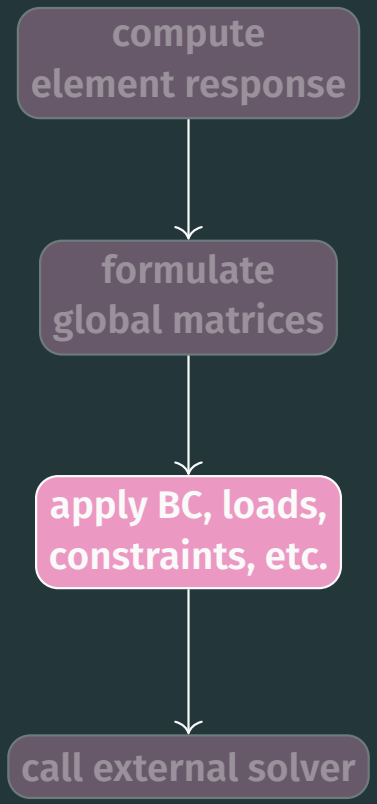
Material Class

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 σ and stiffness E for given strain ε. No external data dependency. Similar to the UMAT subroutine in ABAQUS.

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

```
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:

$$\begin{aligned} &\text{minimize} \quad W(\boldsymbol{u}) \\ &\text{subject to} \quad \boldsymbol{c}(\boldsymbol{u}) = \mathbf{0} \end{aligned}$$

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

The Lagrange function can be constructed as

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

The stationary point can be obtained

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

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

Loads and Constraints

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

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{K} & \cdot \\ \cdot & \cdot \end{bmatrix} + \begin{bmatrix} \nabla^2 \boldsymbol{c} & \nabla \boldsymbol{c}^{\mathrm{T}} \\ \nabla \boldsymbol{c} & \cdot \end{bmatrix} = \begin{bmatrix} \boldsymbol{K} + \boldsymbol{K}_c & \boldsymbol{K}_b^{\mathrm{T}} \\ \boldsymbol{K}_b & \cdot \end{bmatrix}$$

$\boldsymbol{K}_b$ 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.

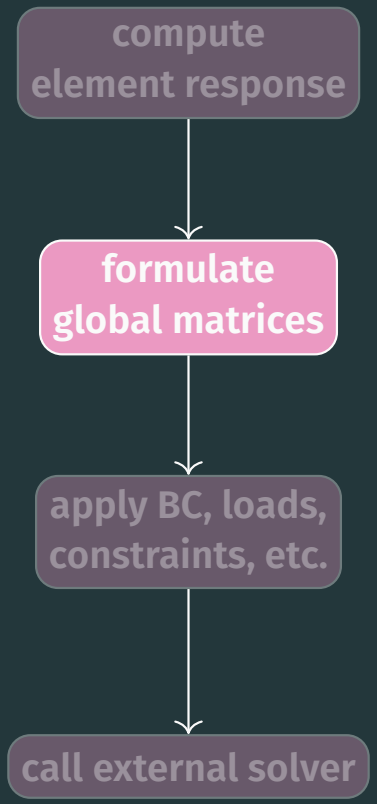
Noting $\boldsymbol{K}_c$ is similar to elemental stiffness $\boldsymbol{K}_e$ and can be assembled into $\boldsymbol{K}$, it is possible to store $\boldsymbol{K}_b$ separately and static condensation can be used to solve the system. The additional cost is only forward/backward substitution as $\boldsymbol{K} + \boldsymbol{K}_c$ 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.

```
// ...
auto counter = 0;
for(auto& I : constraint_pond.get()) {
    const auto m_size = I->get_multiplier_size();
    if(!I->is_initialized() || 0 == m_size) continue;
    const auto e_size = counter + m_size - 1;
    t_resistance.subvec(counter, e_size) = I->get_auxiliary_resistance();
    t_load.subvec(counter, e_size) = I->get_auxiliary_load();
    t_stiffness.cols(counter, e_size) = I->get_auxiliary_stiffness();
    counter += m_size;
}
// ...
```

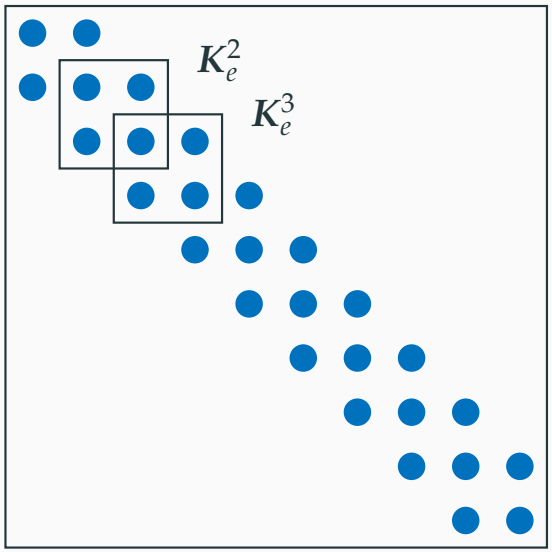
Global Assembly

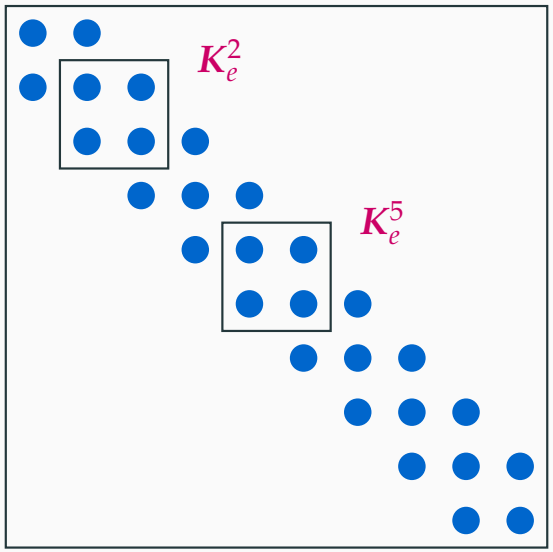

compute
element response
formulate
global matrices
apply BC, loads,
constraints, etc.
call external solver

Global Matrix Assembly

Assembling elemental stiffness $\boldsymbol{K}_e$ into global stiffness $\boldsymbol{K}$ is often done sequentially due to data overlapping.

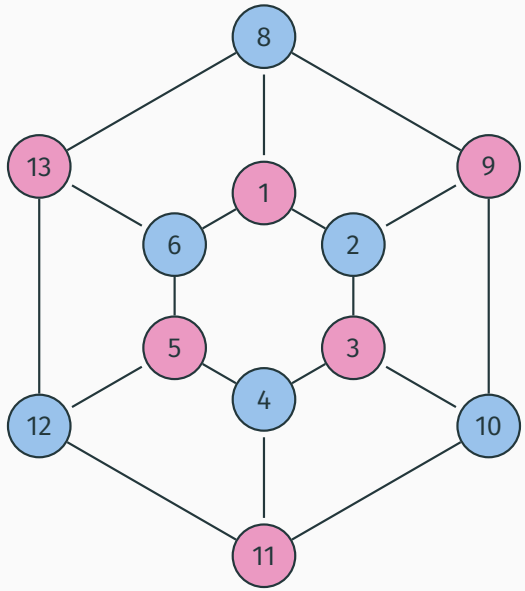
Mutex may be expensive. Instead, the reliable k-coloring algorithm can be used. The earliest reference dates back to 1982.





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(),
               ↪  get_element(tag)->get_dof_encoding());
5          });
6      });
7  }
```

Global Matrix Assembly

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

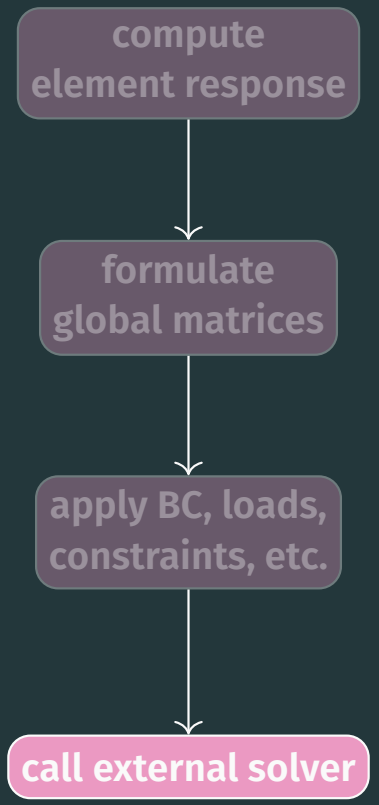
Pre-coloring the model leads to a lock-free algorithm for matrix assembly with the dense storage.

What about the sparse storage?

- Use COO format and pre-allocate memory blocks, $\boldsymbol{K}_e$ can be copied into $\boldsymbol{K}$ in parallel. Since $\boldsymbol{K}_e$ 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 $\boldsymbol{K}$.

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

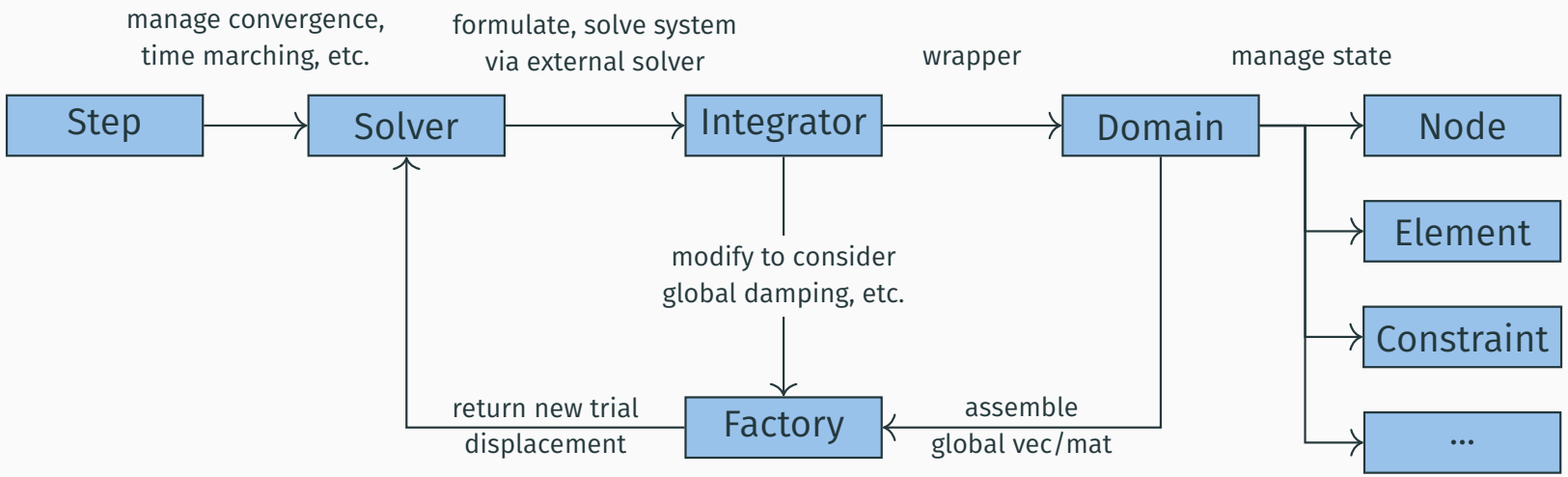
Solving


compute
element response
formulate
global matrices
apply BC, loads,
constraints, etc.
call external solver

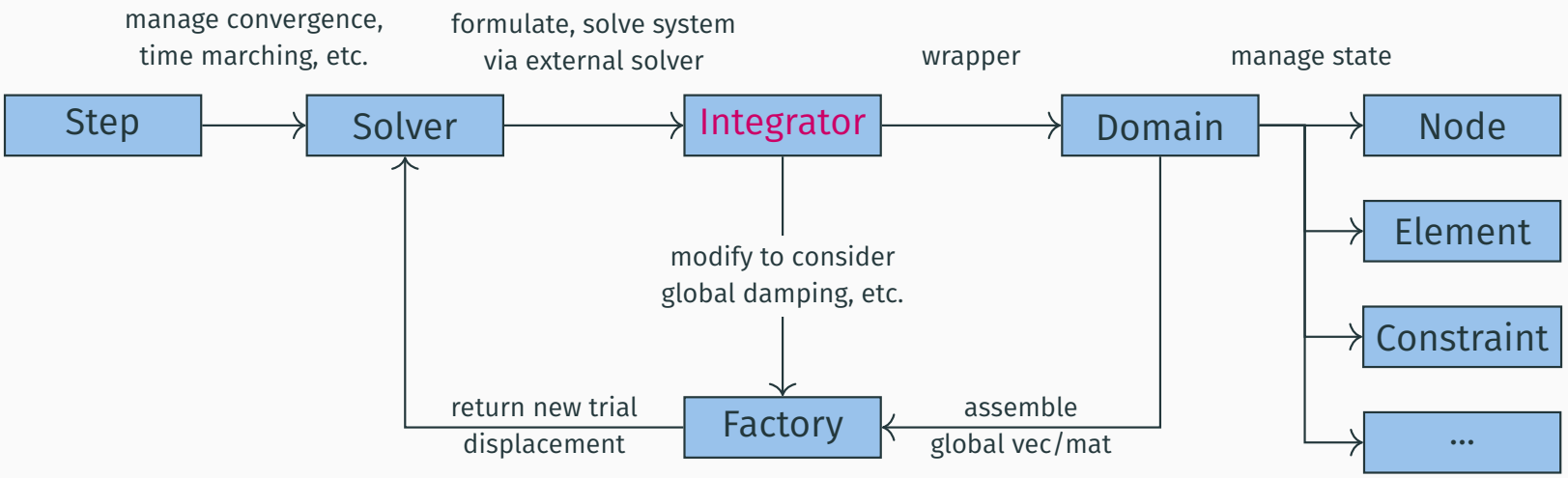
Analysis Logic

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

```cpp
for(const auto& t_domain : domain_pool)
    if(t_domain->is_active())
        for(const auto& t_step : t_domain->get_step_pool()) {
            // ...
            if(SUANPAN_FAIL == t_step->Step::initialize()) return SUANPAN_FAIL;
            if(SUANPAN_FAIL == t_step->initialize()) return SUANPAN_FAIL;
            if(SUANPAN_FAIL == t_step->analyze()) return SUANPAN_FAIL;
        }
```



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.

```
int Newton::analyze() {
    // ...
    // G is an Integrator
    while(true) {
        G->assemble_resistance(); // in parallel
        G->assemble_matrix();     // in parallel
        G->process_load();        // in parallel
        G->process_constraint();  // in parallel

        G->solve(ninja, G->get_force_residual()); // X=inv(A)*B

        G->update_trial_displacement(ninja); // sync global in parallel
        G->update_trial_status();            // sync local in parallel

       if(C->is_converged()) return SUANPAN_SUCCESS; // exit if converged
        if(++counter > max_iteration) return SUANPAN_FAIL;
    }
}
```

Analysis Type

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

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

```
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:

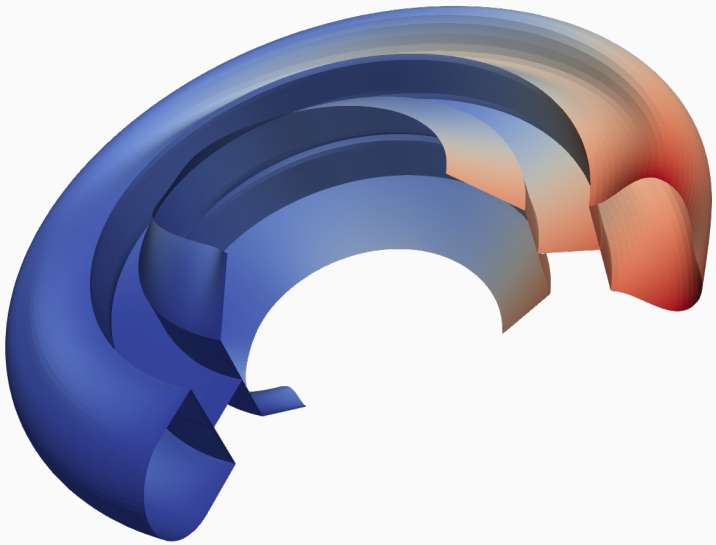
  - OpenBLAS
  - MKL
  - SPIKE
  - ARPACK
  - FEAST
  - SuperLU
  - MUMPS
  - CUDA

- distributed memory model based parallelisation is managed by external solvers

Benchmark

Benchmark

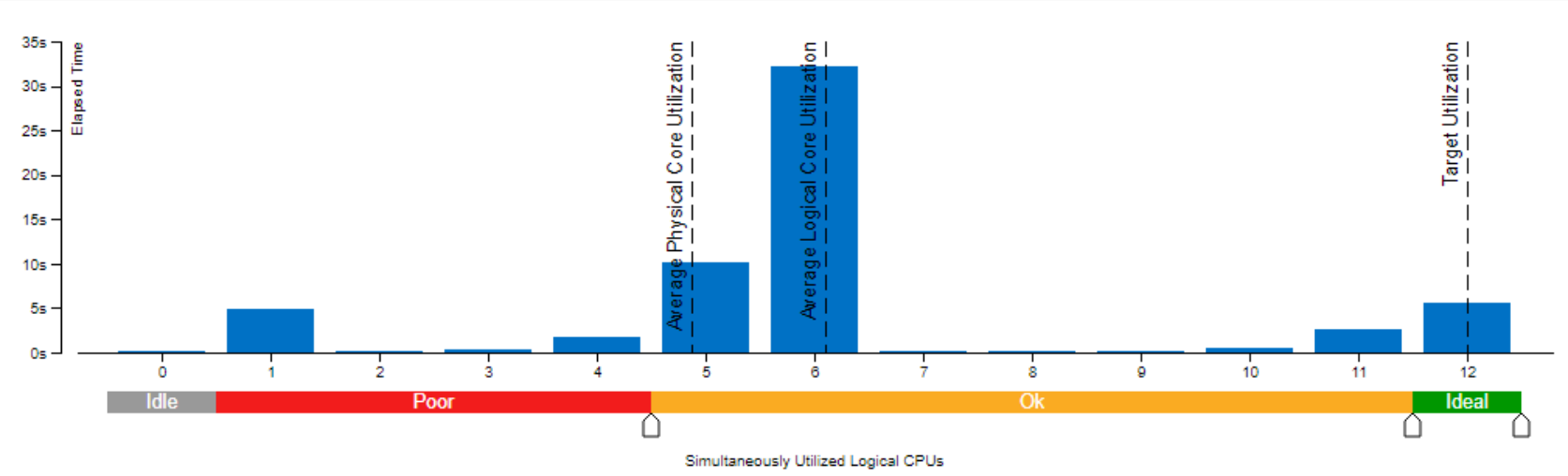
- i7-8700 (6C12T) with DDR4-2666
- about 120 GFLOPS (pure DGEMM)
- 120744 DoFs
- 39990 three-node shell elements
- linear elastic material
- 40 complete analysis cycles
- solved by DPBSV



Benchmark

with default MKL configurations

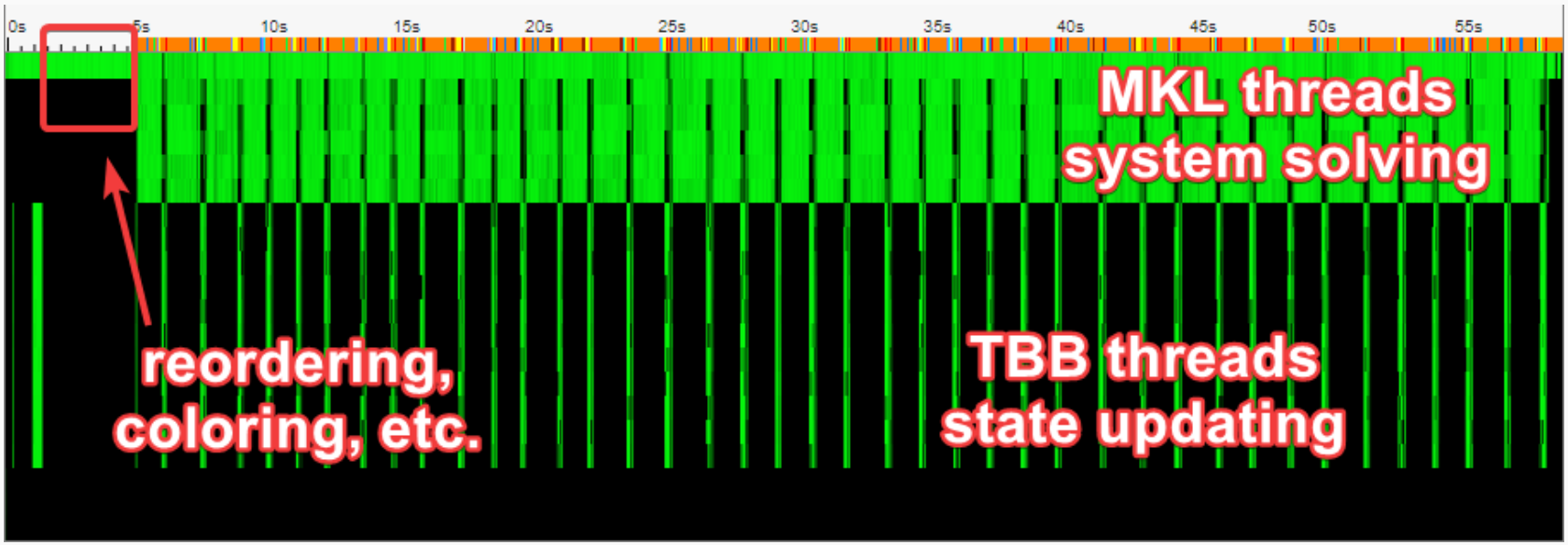
- 105 to 107 GFLOPS overall
- additional 10 GFLOPS excluding initialisation
- 80 % CPU utilisation



Benchmark

with default MKL configurations

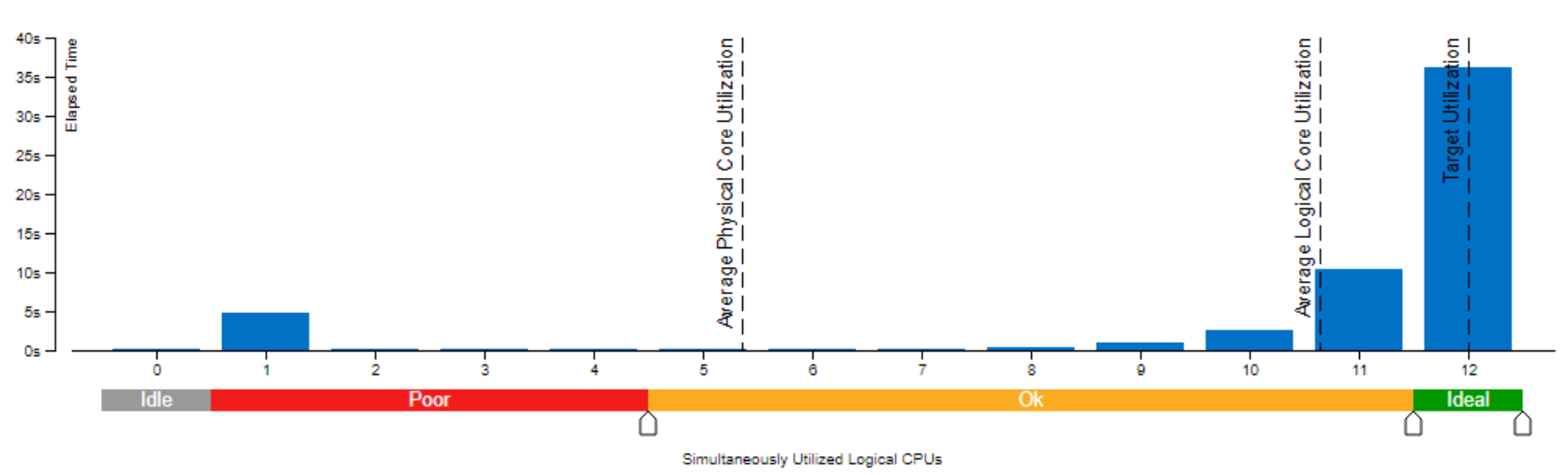
- 80 % CPU utilisation
- 105 to 107 GFLOPS overall
- additional 10 GFLOPS excluding initialisation



Benchmark

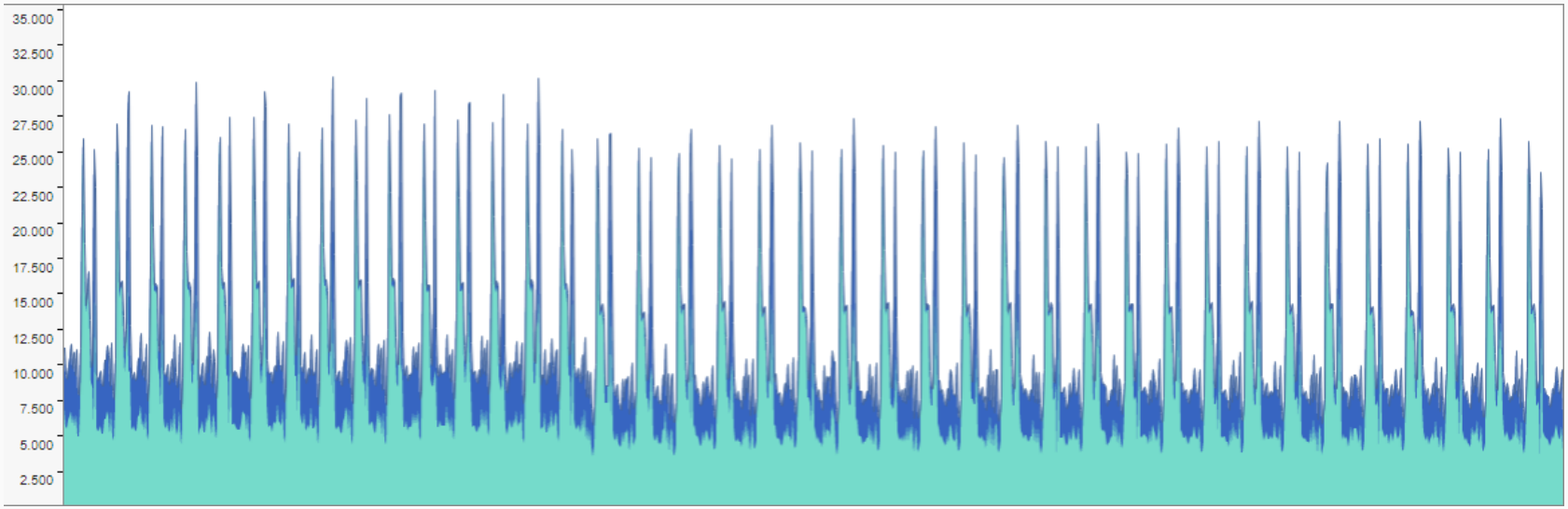
manually override MKL defaults, not optimal but slightly faster

- `MKL_DYNAMIC=FALSE`
- `MKL_NUM_THREADS=12`
- 90 % CPU utilisation
- 110 GFLOPS overall
- 120 GFLOPS excluding initialisation

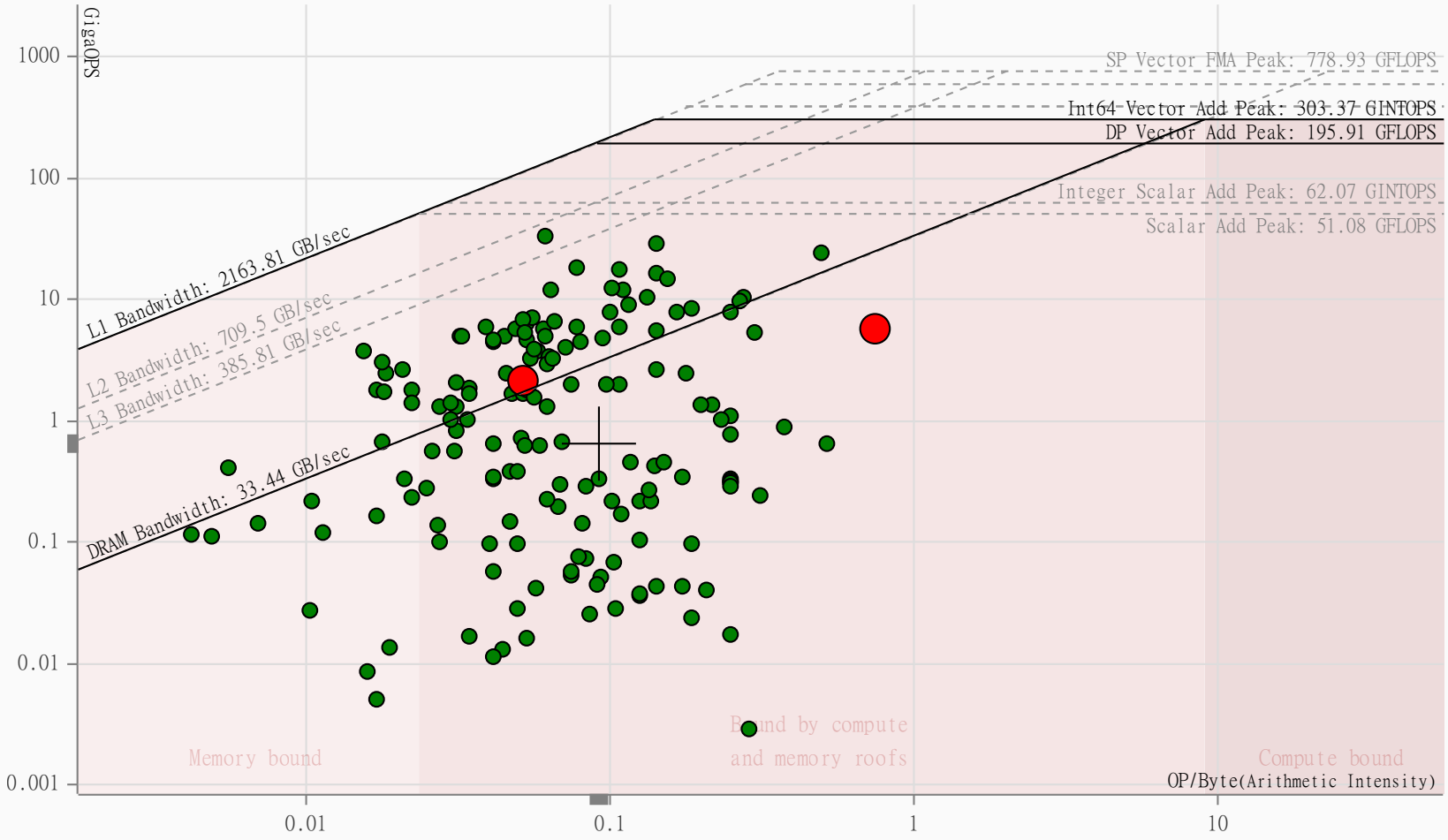


Benchmark

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



Benchmark


GigaOPS
1000
100
10
1
0.1
0.01
0.001
SP Vector FMA Peak: 778.93 GFLOPS
Int64 Vector Add Peak: 303.37 GINTOPS
DP Vector Add Peak: 195.91 GFLOPS
Integer Scalar Add Peak: 62.07 GINTOPS
Scalar Add Peak: 51.08 GFLOPS
L1 Bandwidth: 2163.81 GB/sec
L2 Bandwidth: 709.5 GB/sec
L3 Bandwidth: 385.81 GB/sec
DRAM Bandwidth: 33.44 GB/sec
Memory bound
Bound by compute
and memory roofs
Compute bound
0.01
0.1
1
10
OP/Byte(Arithmetic Intensity)

Summary

- 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!