Architecture of Modern Finite Element Code

with an object oriented approach and hybrid parallelisation

Theodore Chang, Ph.D.

June 25, 2026

Objective

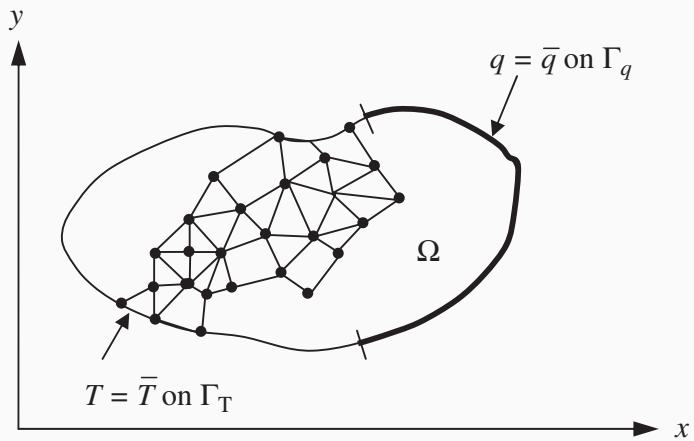
Objectives

- to report the architecture of TLCFEM/suanPan
  - an FEA package written in modern C++ (20)
  - with both shared and distributed memory parallelism
  - covers broad topics in computational solid mechanics
  - contains cutting–edge research outcomes
  - volume: 32k+ tracked lines of code excluding dependencies
- to discuss improvements regarding HPC in simulation
  - parallelisation details

Background

Typical FEA Flow

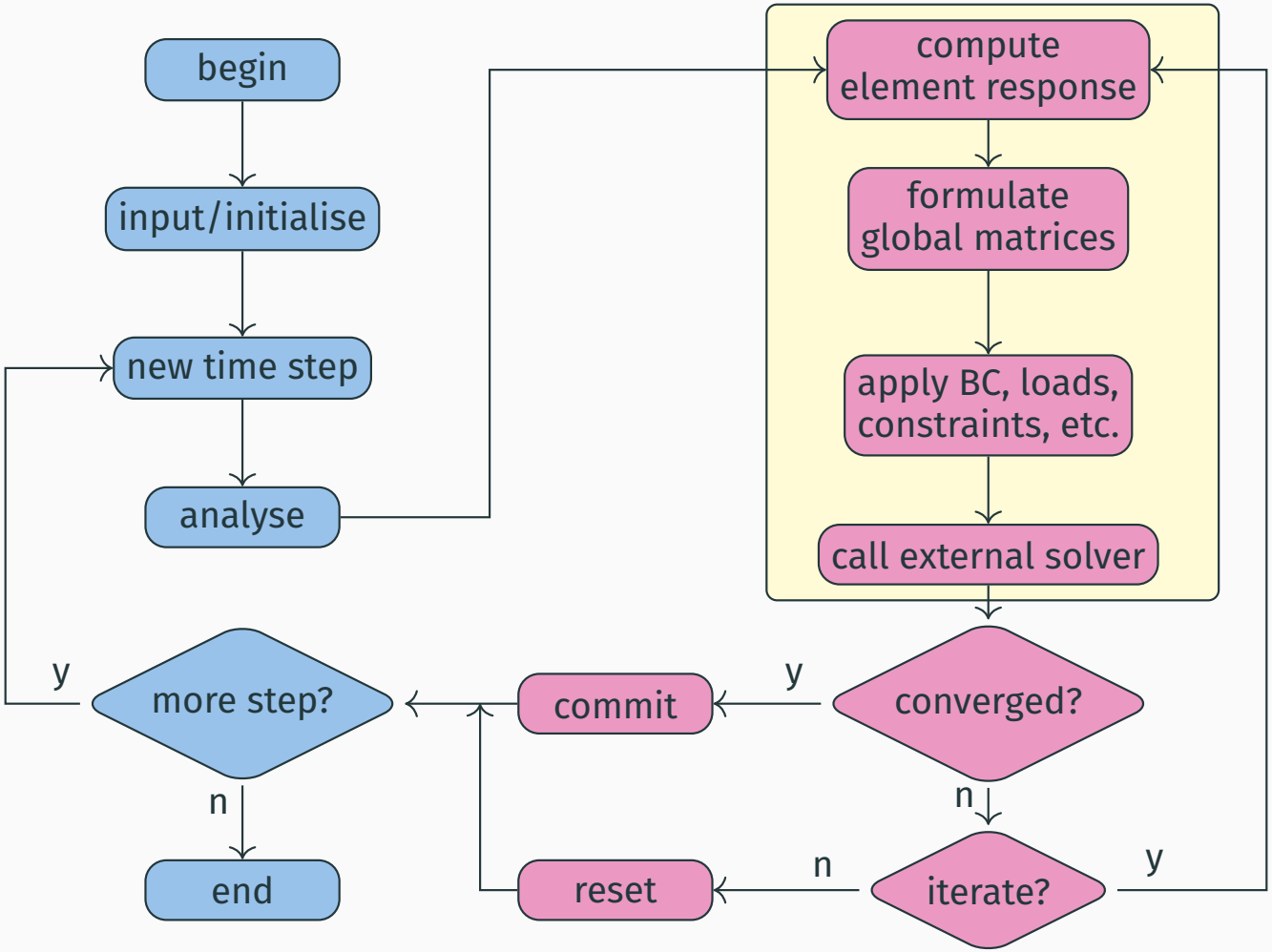
To analyse a continuum problem with FEM, the following tasks are performed.



credit: Fish & Belytschko, 2007

- discretise the geometry with nodes and elements
- compute elemental stiffness K_e and resistance R_e
- assemble global stiffness $K = \sum K_e$ and resistance $R = \sum R_e$
- apply boundary conditions, loads, constraints, etc.
- solve the system $K\Delta U = P - R$

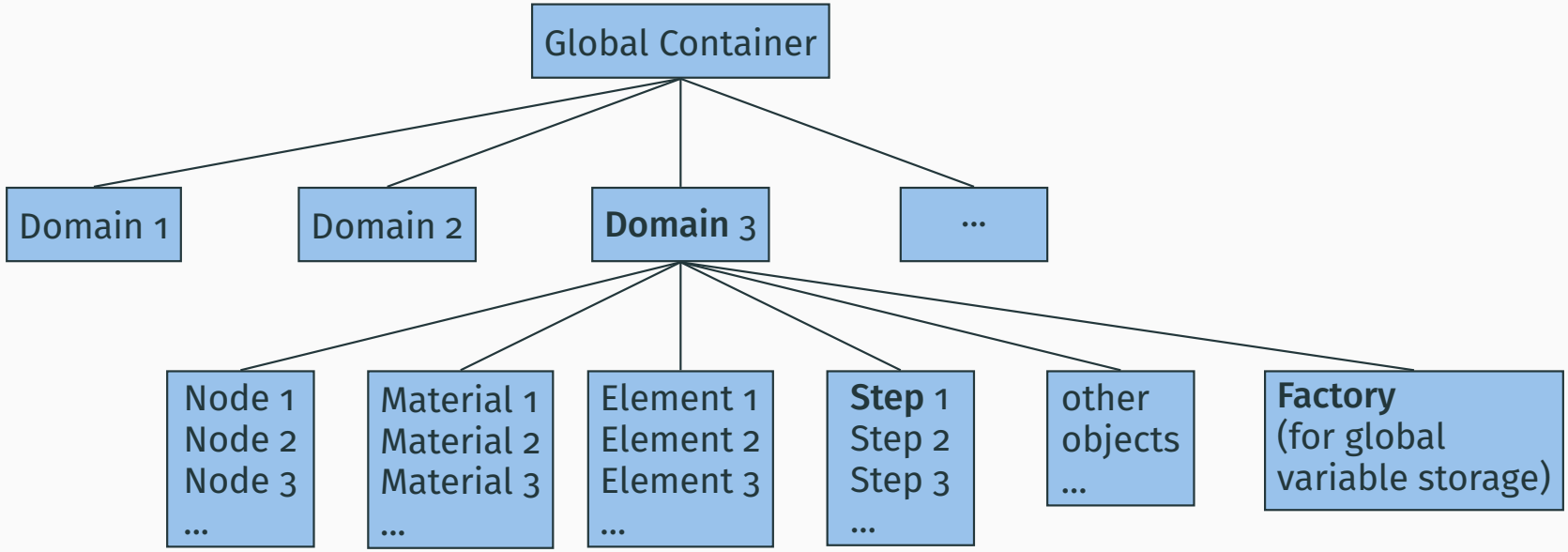
Typical FEA Flow



Storage

Structure

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



Multiple domains can coexist. Similar to ABAQUS, multiple analysis steps can be defined in each domain. Substructuring for distributed memory parallelisation is theoretically possible.

Domain

- represents a problem
- provides storage for all nodes, elements, materials, integrators, etc.

```
class Domain {
    shared_ptr<LongFactory> factory; // use double precision

    StepQueue step_pond;

    AmplitudeStorage amplitude_pond;
    ConstraintStorage constraint_pond;
    ElementStorage element_pond;
    MaterialStorage material_pond;
    NodeStorage node_pond;
    SectionStorage section_pond;
    SolverStorage solver_pond;
public:
    // public methods to initialise/update/assemble, etc.
};
```

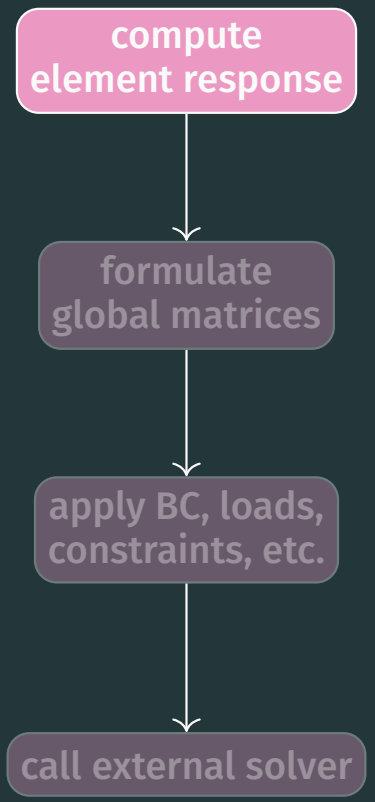
Storage

Concurrent containers are 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 Class

Non-const static variables shall be avoided due to potential racing. Auxiliary methods can be implemented to hide details.

```
1  class Element : protected DataElement {
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 };
```

Element Class

A general purpose dataset is provided to manage state internally. There is no interaction with the outside.

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

    mat initial_mass{}; // mass matrix
    mat trial_mass{};   // mass matrix
    mat current_mass{}; // mass matrix

    mat initial_stiffness{}; // stiffness matrix
    mat trial_stiffness{};   // stiffness matrix
    mat current_stiffness{}; // stiffness matrix

    vec trial_resistance{};            // resistance vector
    vec current_resistance{};          // resistance vector
    // ...
};
```

Element Class

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

```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;
}
```

Material Class

Similar to `Element`, if pre-defined dataset 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, $\sigma = E\varepsilon$.

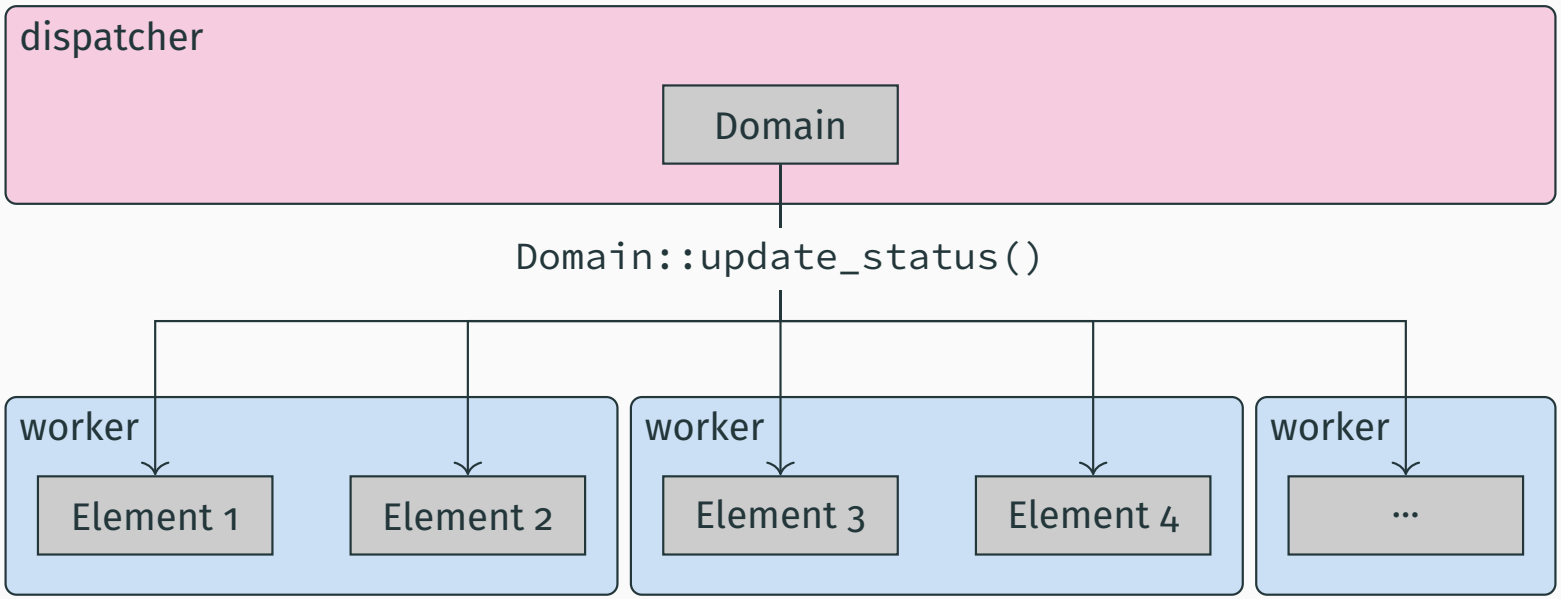
```
1  int Elastic1D::update_trial_status(const vec& t_strain) {
2      trial_strain = t_strain;
3      trial_stiffness = elastic_modulus;
4      trial_stress = trial_stiffness * trial_strain;
5      return SUANPAN_SUCCESS;
6  }
```

State Updating

Each element manages its state independently and updates its state based on global nodal displacement vector.

$$U \Longrightarrow \varepsilon_e \Longrightarrow \sigma_e \Longrightarrow K_e,\ R_e$$

concurrent-read-no-write, no racing, lock free, can be safely parallelised

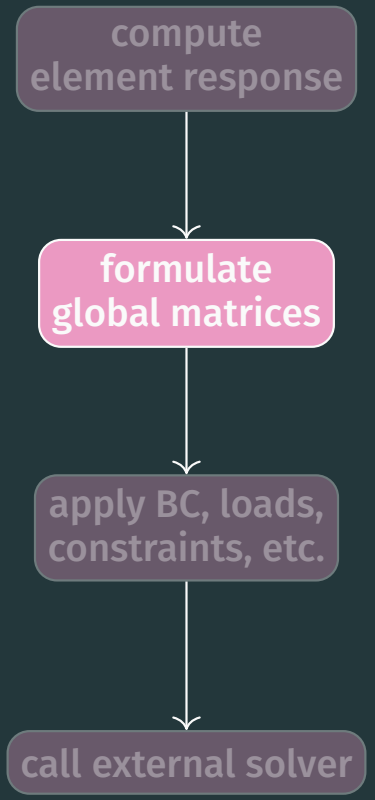


State Updating

Optimal performance is achieved through hybrid parallelism. Each compute node manages a specific subset of elements, updating them concurrently via shared memory without the need for message passing.

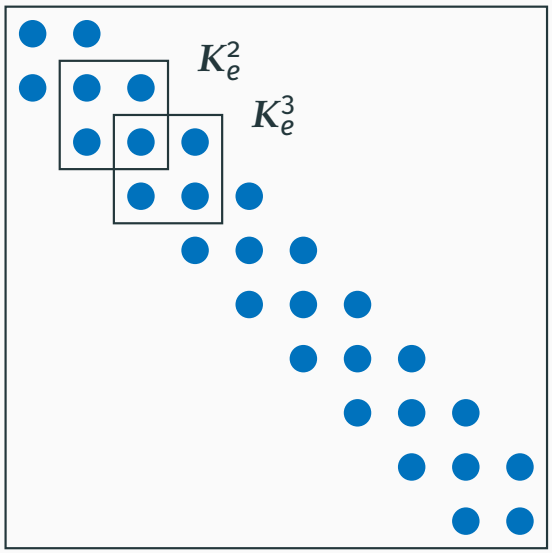
```
1  int Domain::update_trial_status() const {
2      suanpan::for_all(node_pond.get(), [&](const shared_ptr<Node>& t_node)
       ↪  { t_node->update_trial_status(trial_displacement, trial_velocity,
       ↪  trial_acceleration); });
3
4      suanpan::for_all(element_pond.get(), [&](const shared_ptr<Element>&
       ↪  t_element) {
5          if(t_element->is_local) t_element->update_status();
6      });
7      // ...
8  }
```

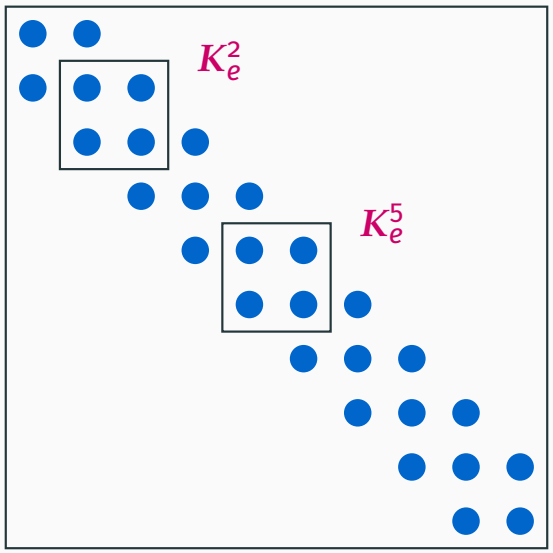
Global Assembly


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

Global Matrix Assembly

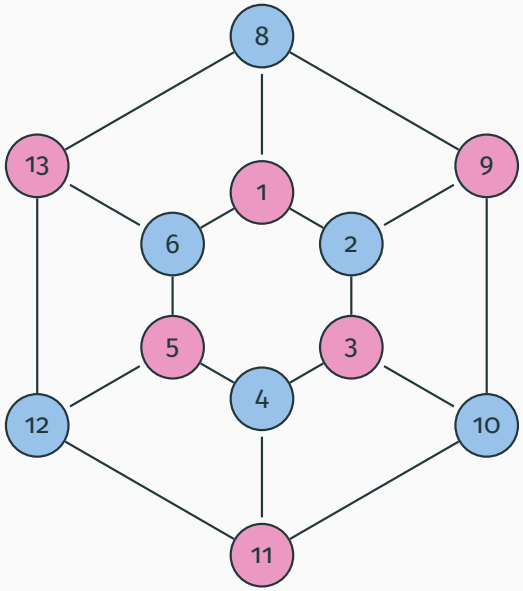
Assembling elemental stiffness K_e into global stiffness K is often done sequentially due to data overlapping.
Mutex may be expensive. Instead, the reliable k-coloring algorithm can be used. The earliest reference dates back to 1982.





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

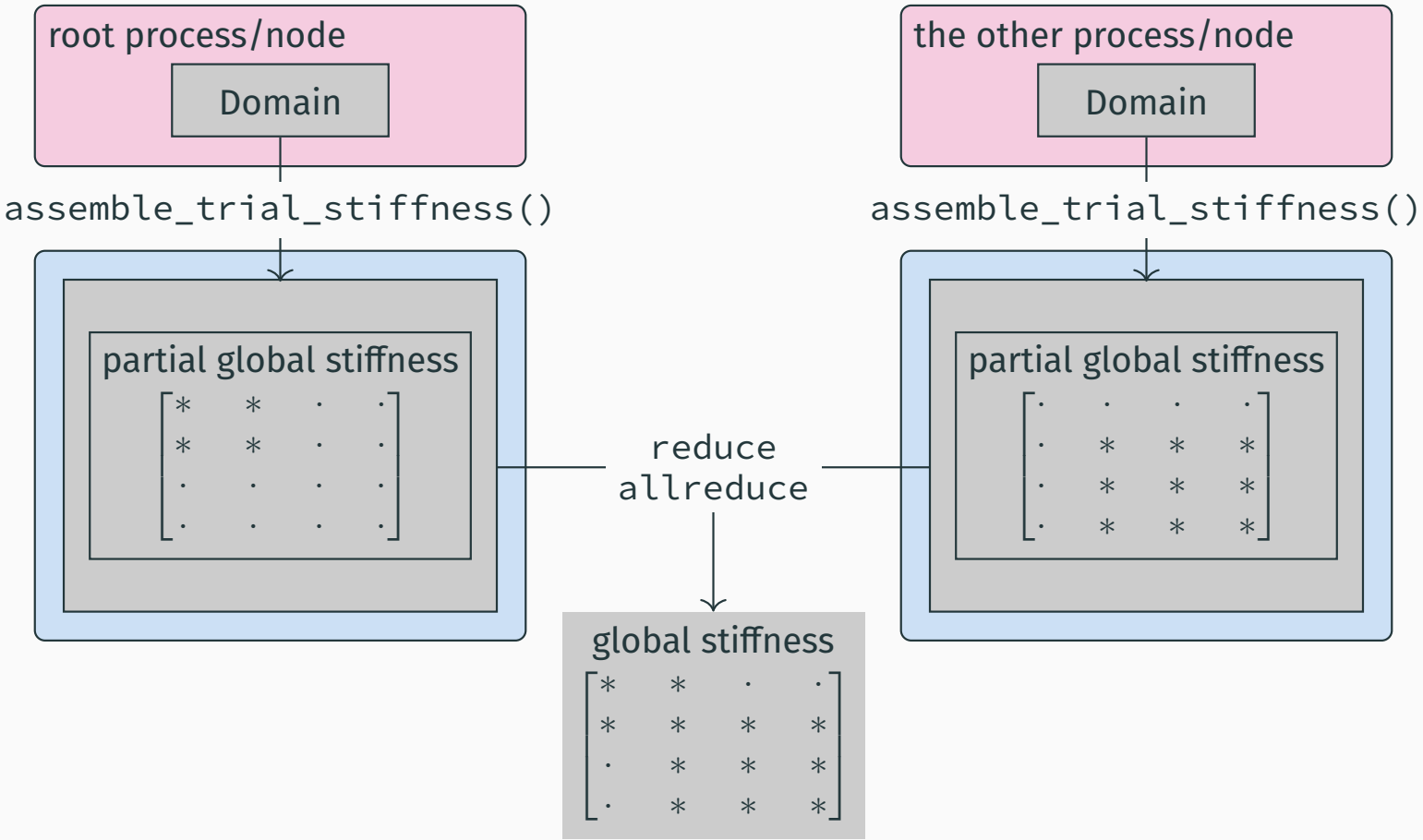



```
1  void Domain::assemble_trial_stiffness() const {
2      std::ranges::for_each(color_map, [&](const std::vector<unsigned>& color) {
3          suanpan::for_all(color, [&](const unsigned tag) {
4              if(const auto& I = get_element(tag); I->is_local)
               ↪  factory->assemble_stiffness(I->get_trial_stiffness(), I->get_dof_encoding(),
               ↪  I->get_dof_mapping());
5          });
6      });
7
8      factory->get_stiffness()->allreduce();
9  }
```

Global Matrix Assembly

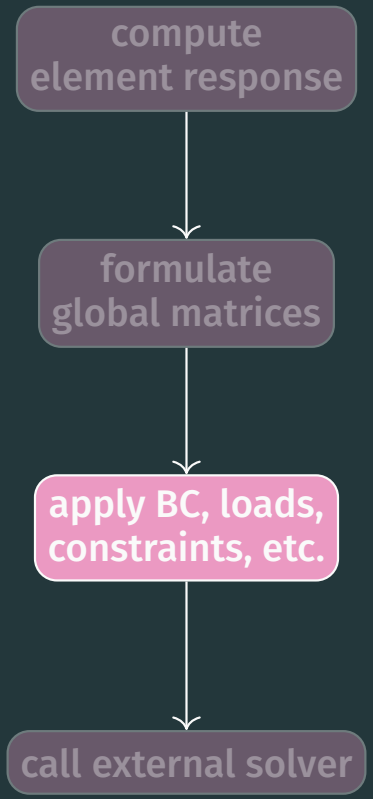
Pre-coloring the model leads to a **lock-free** algorithm for **parallel** matrix assembly with the **dense** storage.
What about the **sparse** storage? Use COO format and pre-allocate memory blocks, K_e can be copied into K in parallel. Since K_e spans contiguous memory, the copy operation may be automatically vectorised by compiler via SIMD.
Elemental quantities are assembled locally on each compute node to formulate the partial global matrices/vectors. Only these partial quantities will be reduced (synchronised) to formulate the final full quantities.

Distributed Approach — Gathering



Distributed Approach — Traits

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


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

Loads and Constraints

Consider the optimisation problem:

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

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

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

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

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

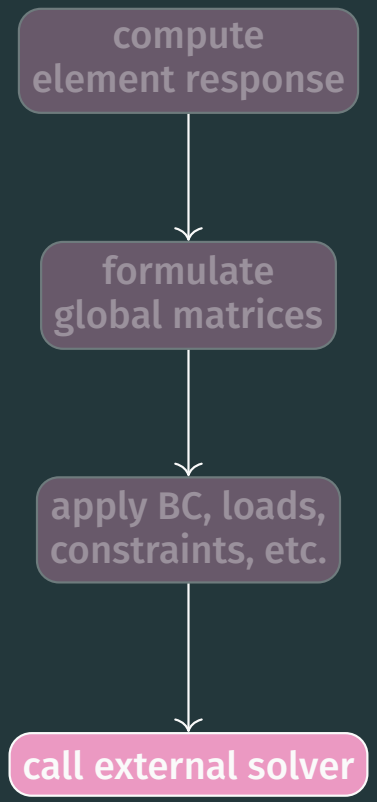
K_b is known as border matrix. Often, it is **sparse**. The size of global stiffness changes due to the presence of constraints.
Since the number of constraints is not known in advance, memory reallocation is required, which is not preferred.
Noting K_c is similar to elemental stiffness K_e and can be 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.

```
// ...
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;
}
// ...
```

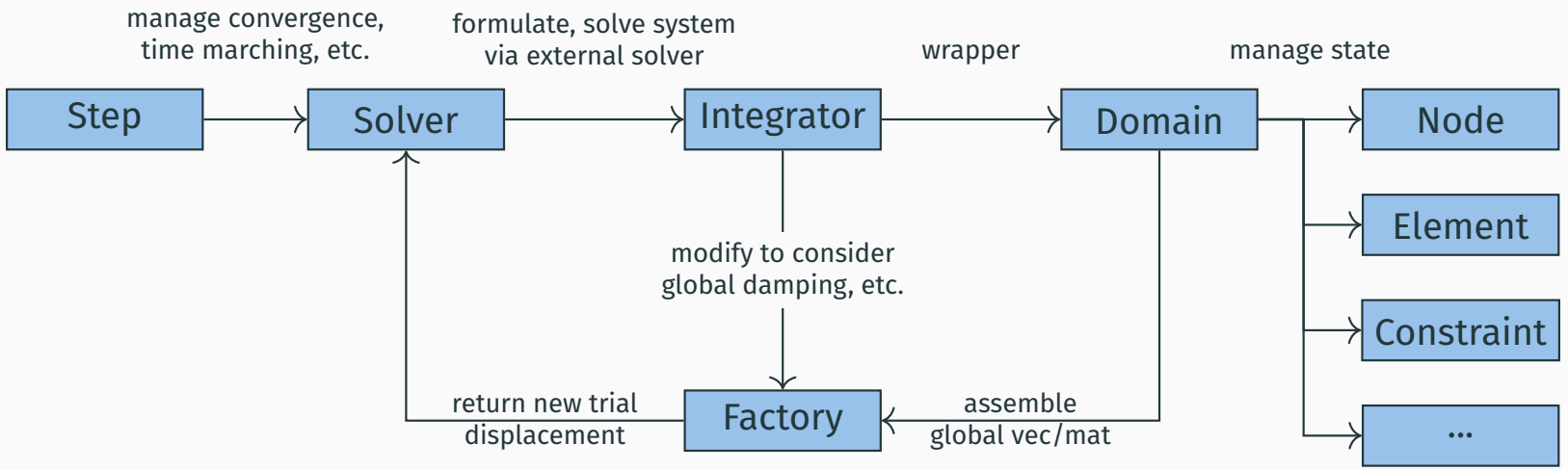
Solving


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

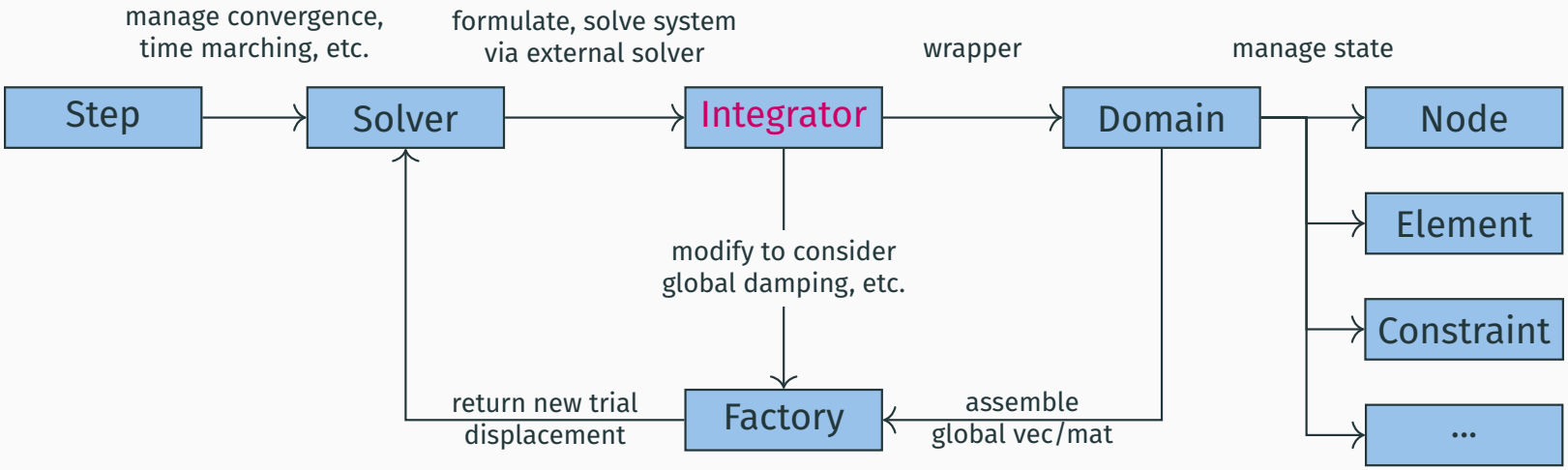
Analysis Logic

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.

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

```cpp
// Newmark is derived from Integrator
void Newmark::assemble_matrix() {
    const auto& D = get_domain().lock(); // weak_ptr
    auto& W = D->get_factory();

    D->assemble_trial_stiffness();
    D->assemble_trial_mass();
    D->assemble_trial_damping();

    // this addition is also parallelised
    W->get_stiffness() += C0 * W->get_mass() + C1 * W->get_damping();
}
```

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 solvers are managed independently

Distributed Solvers

TLCFEM/ezp: lightweight C++ wrapper for distributed solvers
With roughly 2k lines of code, it manages to:

- hides all MPI details
- hides all solver details
- exposes an interface similar to the sequential counterpart
- requires zero extra setup — the global matrix collected from the previous step can be directly passed in and solved

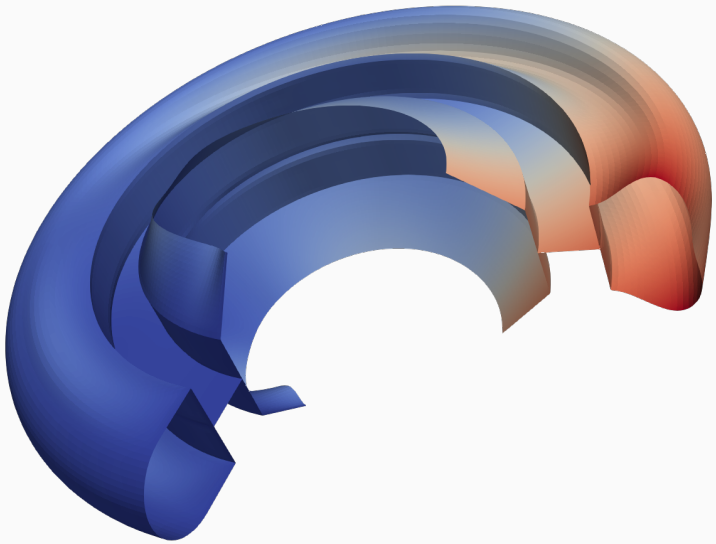
Distributed Solvers — Example

```cpp
#include <ezp/pgesv.hpp>
#include <iomanip>
#include <iostream>

using namespace ezp;

int main() {
    const auto rank = get_env<int>().rank();
    constexpr auto N = 6, NRHS = 2;
    std::vector<double> A, B;

    if(0 == rank) {
        A.resize(N * N, 0.);
        B.resize(N * NRHS, 1.);

        const auto IDX = par_dgesv<int>::indexer{N};
        for(auto I = 0; I < N; ++I) A[IDX(I, I)] = static_cast<double>(I);
    }

    par_dgesv<int>().solve({N, N, A.data()}, {N, NRHS, B.data()});

    if(0 == rank) {
        std::cout << std::setprecision(10) << "Solution:\n";
        for(auto i = 0; i < B.size(); ++i) std::cout << B[i] << '\n';
    }

    return 0;
}
```

Benchmark

Benchmark

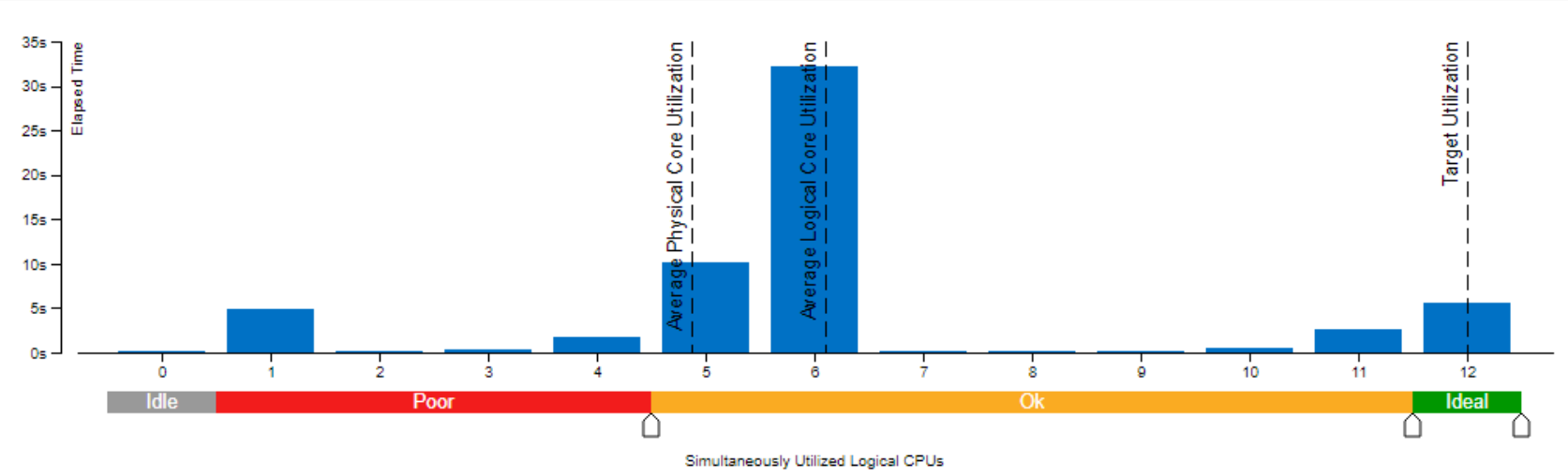
- i7-8700 (6C12T) with DDR4-2666
- about 120 GFLOPS (pure DGEMM)
- 120744 DoFs
- 39990 three-node shell elements
- linear elastic material
- 40 complete analysis cycles
- solved by DPBSV



Benchmark

with default MKL configurations

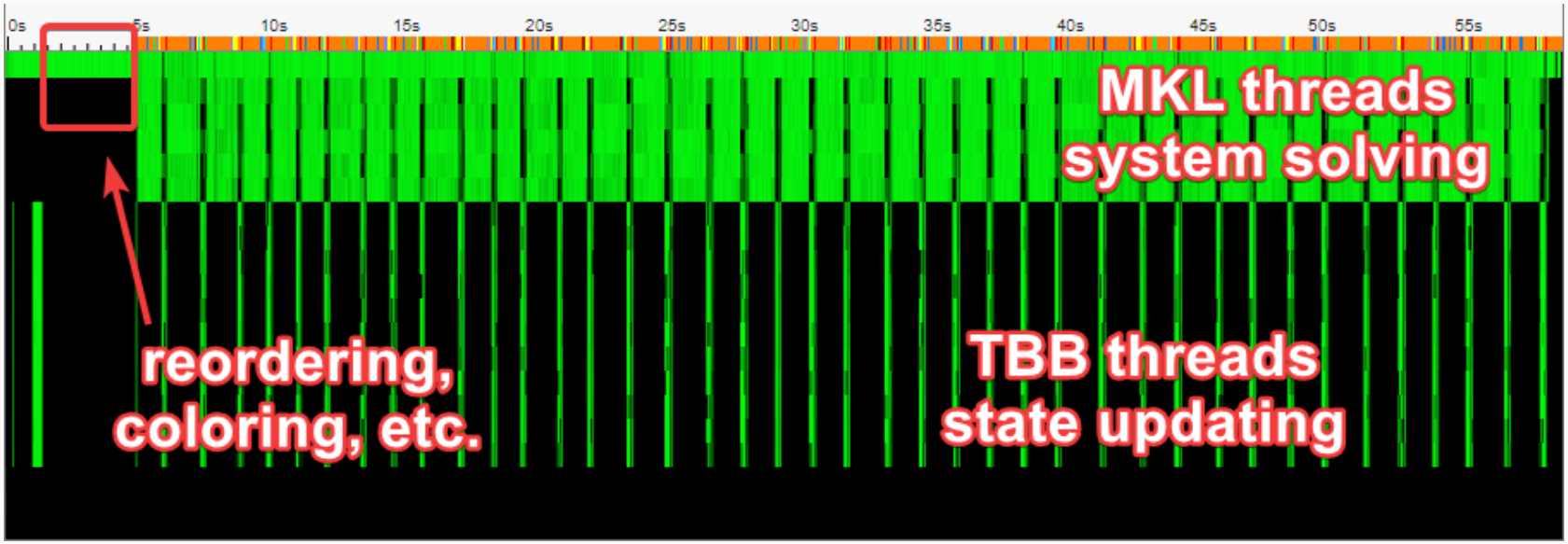
- 105 to 107 GFLOPS overall
- additional 10 GFLOPS excluding initialisation
- 80 % CPU utilisation



Benchmark

with default MKL configurations

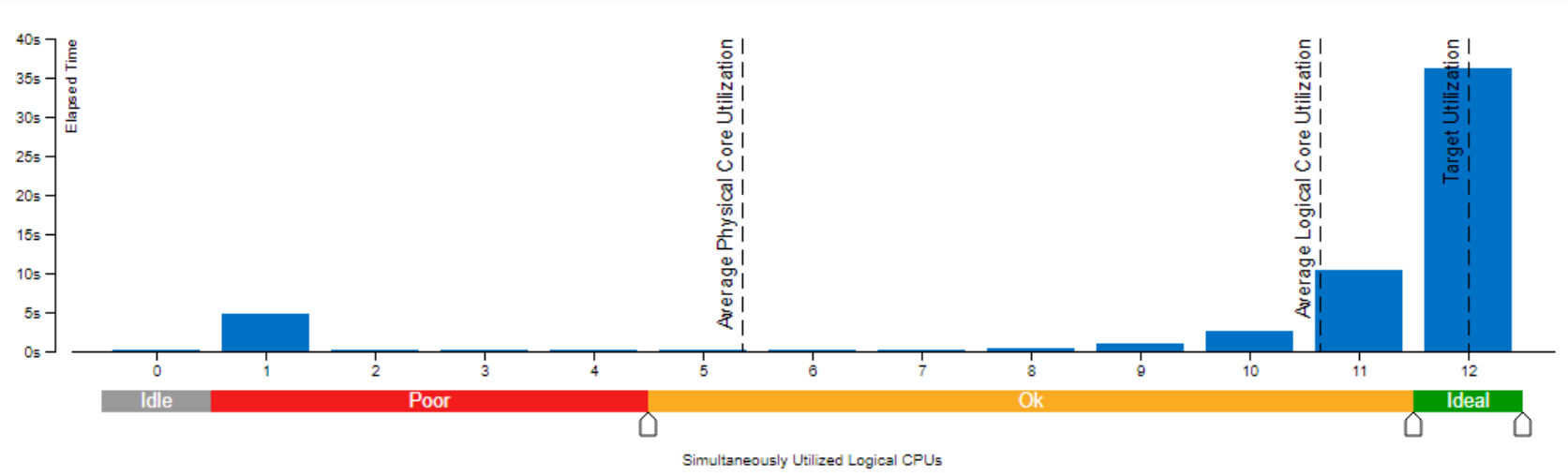
- 80 % CPU utilisation
- 105 to 107 GFLOPS overall
- additional 10 GFLOPS excluding initialisation



Benchmark

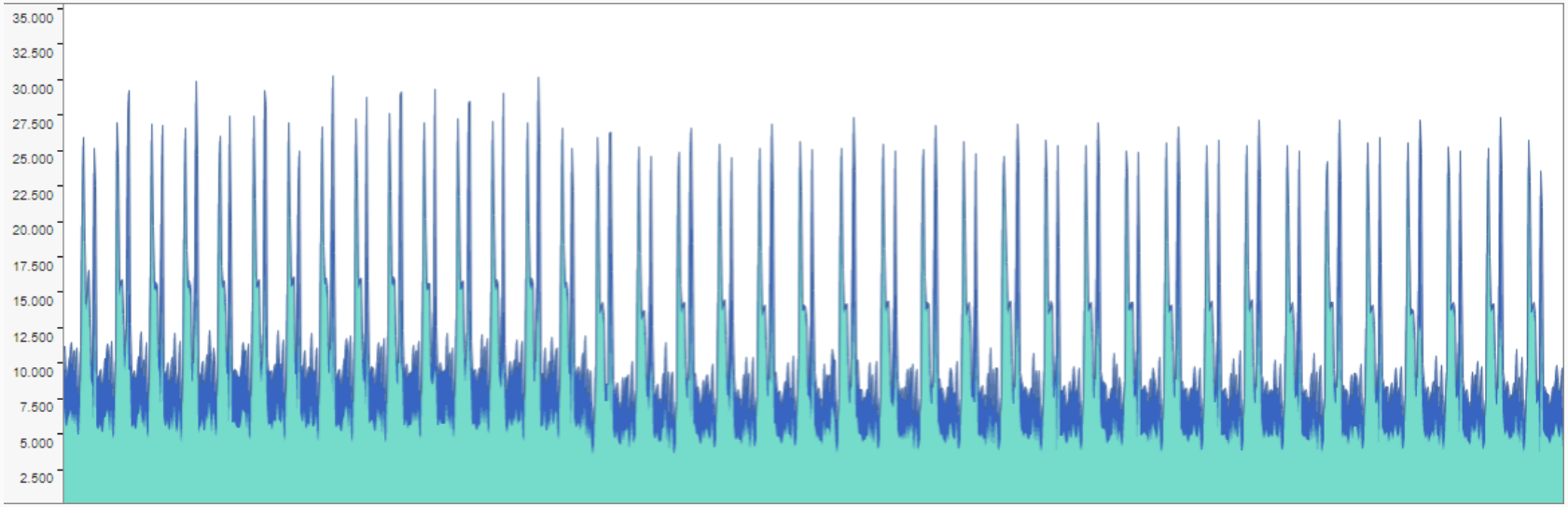
manually override MKL defaults, not optimal but slightly faster

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

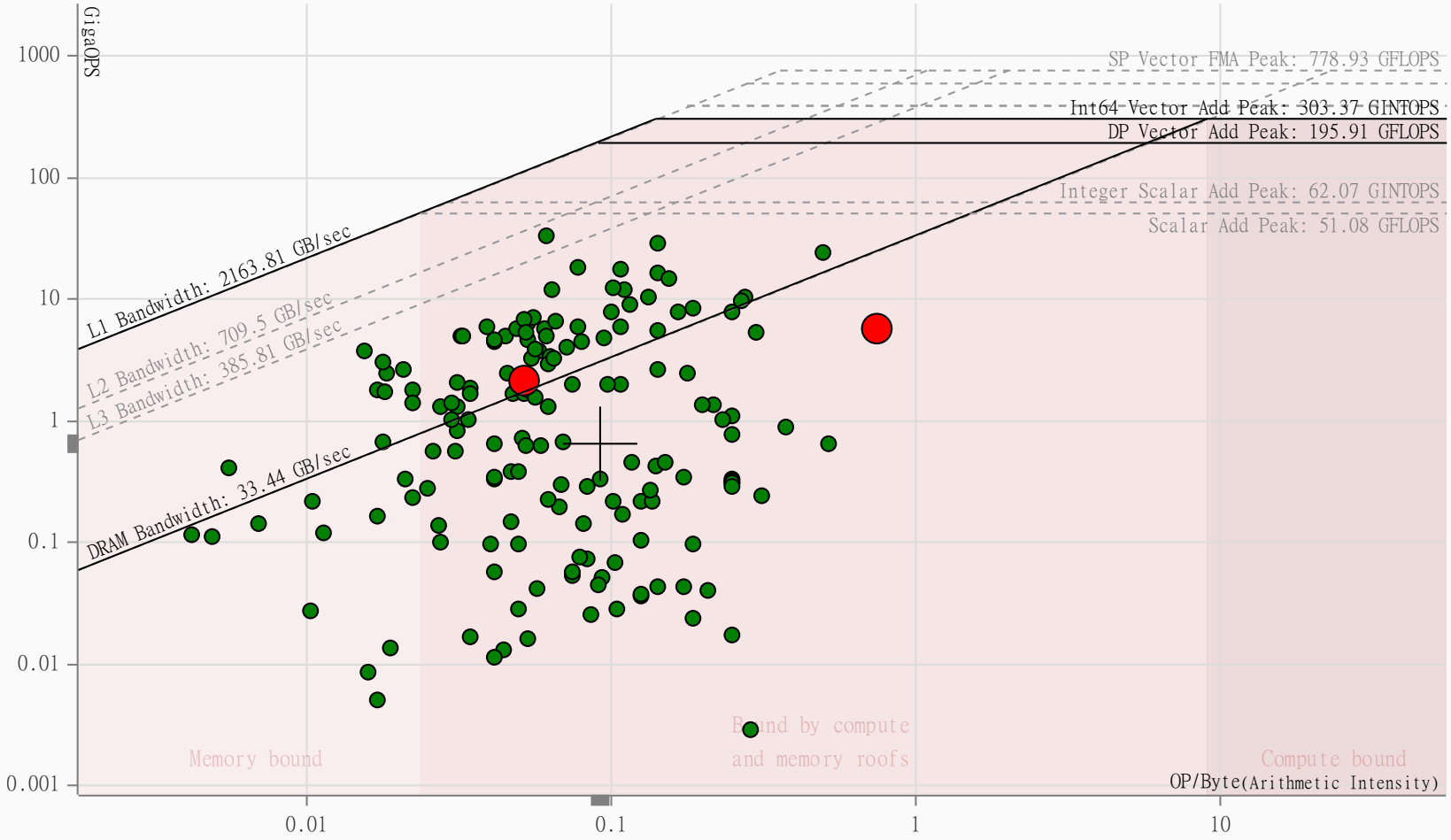


Benchmark

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



Benchmark


GigaOPS
1000
100
10
1
0.1
0.01
0.001
0.01
0.1
1
10
OP/Byte(Arithmetic Intensity)
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

Summary

Base parallelisation drivers: `std::execution`, `tbb`, `blas`, `lapack`, `scalapack`, `mpi`, `cuda`, etc.

- fully parallelised across all stages
- task based parallelisation
- relatively flat analysis logic
- minimum data coupling/dependency
- highly extensible

Future work: DPCPP (SYCL) unified shared memory (USM) based approach?

Thank you!