Numerical Methods for the Solution of PDEs

Laboratory with deal.II — www.dealii.org

Shared memory parallelisation and mesh_loop

Luca Heltai <luca.heltai@unipi.it>

Dipartimento di Matematica
Università di Pisa

Aims for this module

- Identify parts / blocks of code that are (easily) parallelizable
- Learn how to parallelize using
  - Threads
  - Tasks
  - WorkStream::run / MeshWorker::mesh_loop
- Assemble a posteriori error estimators in parallel

Reference material

- Tutorials
  - https://dealii.org/current/doxygen/deal.II/step_9.html
  - https://dealii.org/current/doxygen/deal.II/step_13.html
  - http://www.math.colostate.edu/~bangerth/videos.676.39.html
  - http://www.math.colostate.edu/~bangerth/videos.676.40.html
- Documentation:
  - https://dealii.org/current/doxygen/deal.II/group__threads.html
  - https://www.dealii.org/current/doxygen/deal.II/namespaceWorkStream.html
  - https://dealii.org/current/doxygen/deal.II/namespaceparallel.html

Identifying parallelizable code

- Consider this example:

```
template <int dim>
void MyProblem<dim>::setup_system (){
  dof_handler.distribute_dofs();
  DoFTools::make_hanging_node_constraints (…);    // 1
  DoFTools::make_sparsity_pattern (…);            // 2
  VectorTools::interpolate_boundary_values (…);   // 3
…
}
```

  - Operations (1,2,3) are independent of one another
  - Could be reordered without consequence

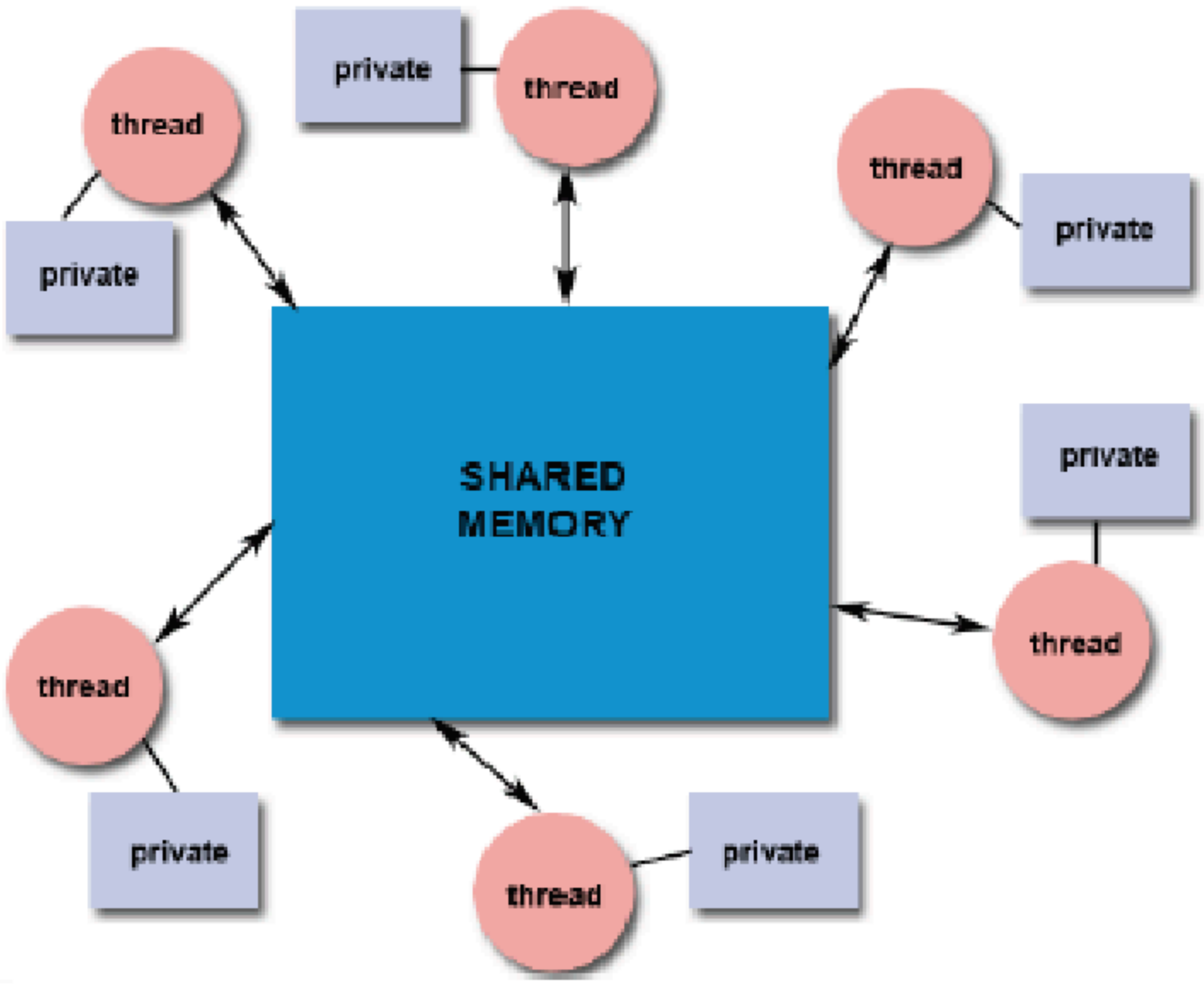
Identifying parallelizable code

- "Embarrassingly parallelizable tasks"

```
template <int dim>
void MyProblem<dim>::assemble_system () {
…
for (auto cell : dof_handler.active_cells()) {
  fe_values.reinit (cell);
  ...assemble local contribution...
  ...copy local contribution into global matrix/rhs vector...
}
}
```

  - Many more cells than machine cores
  - Computations of local matrices/vectors are mutually independent
  - **Accumulation into global system matrix/vector is not!**

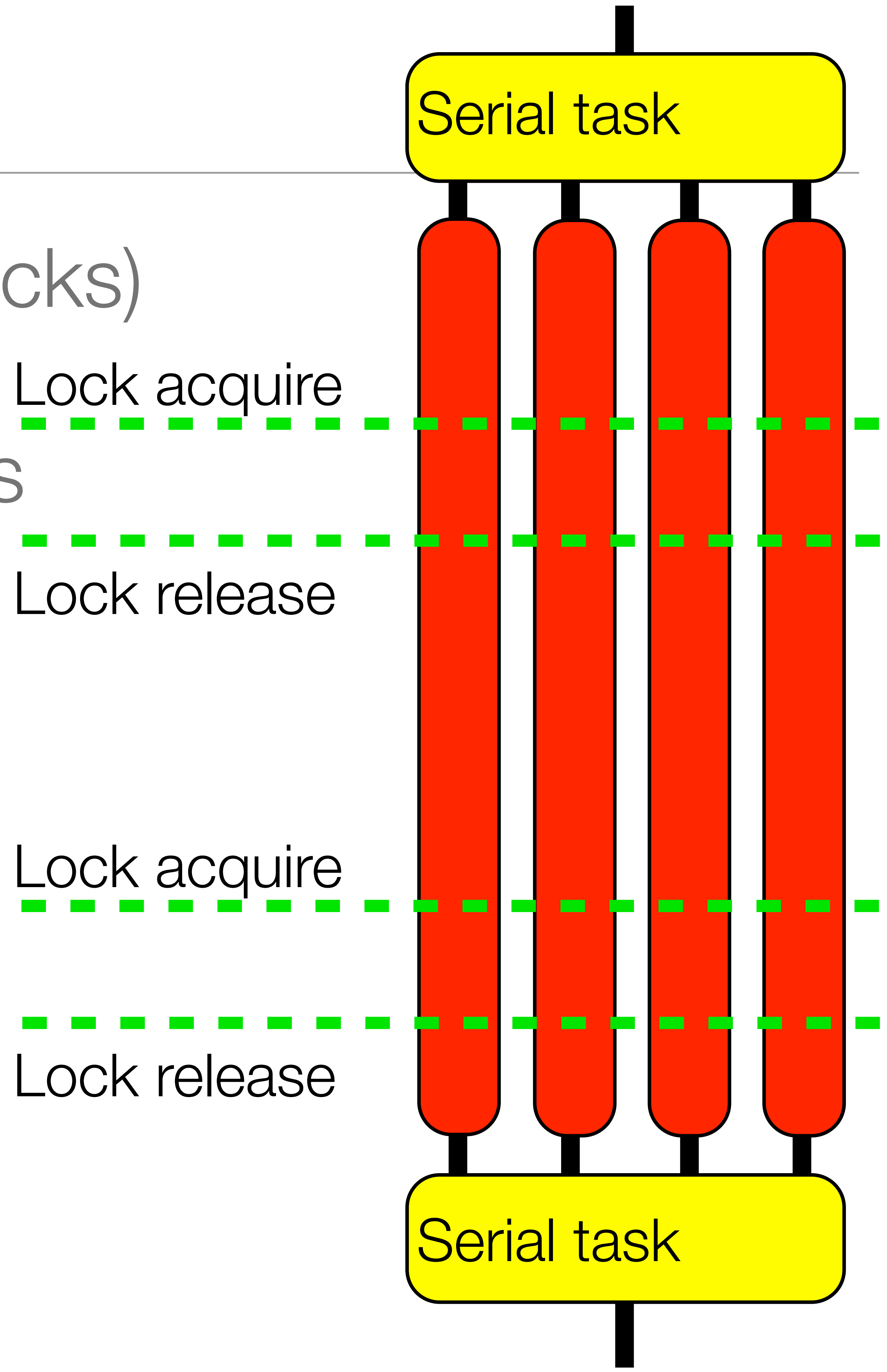
Shared memory model



- All threads have access to the same global shared memory.
- Threads also have their own private memory.
- Shared data is accessible by all threads.
- Private data can be only accessed by the thread that owns it.
- Programmers are responsible for synchronizing access (protecting) globally shared data.

Independent threaded tasks: Option 1

- Code divergence with / without barriers (global / in-thread locks)
- Best used for small number of completely independent tasks
- Inside each thread: Shared data
  - Reading is a safe operation!
  - Use locks to allow data writing
    - Convergence point for threads (bottleneck)
    - Potential for deadlocks



Information about the system

- Query number of cores, and number of enabled threads
- Set maximum number of threads you want to “spawn”

```
MultithreadInfo::n_cores()

MultithreadInfo::n_threads()

MultithreadInfo::set_threads_limit()
```

Creating independent threaded tasks: the **Threads** class

- The call to join() is a blocking call
- Waits for the thread to finish before continuing

```
template <int dim>
void MyProblem<dim>::setup_system (){
  dof_handler.distribute_dofs();

  Threads::Thread<void> thread1, thread2, thread3;

  thread1 = Threads::new_thread (&DoFTools::make_hanging_node_constraints,…);
  thread2 = Threads::new_thread (&DoFTools::make_sparsity_pattern, …);
  thread3 = Threads::new_thread (&VectorTools::interpolate_boundary_values,,…);

  thread1.join();     // and same for thread2, thread3
  …
}
```

Creating independent threaded tasks: the ThreadGroup class

- Why is this inefficient?
- How do we prevent data races?

```
void MyProblem<dim>::assemble_on_one_cell (cell_iterator &cell) {…}

void MyProblem<dim>::assemble_system () {
  Threads::ThreadGroup<void> threads;

  for (cell=dof_handler.begin_active(); …)
    threads += Threads::new_thread (
      &MyProblem<dim>::assemble_on_one_cell,
      this, cell);

  threads.join_all ();
}
```

Creating independent threaded tasks: Ranged based assembly

```
void MyProblem<dim>::assemble_on_cell_range (
  cell_iterator &range_begin,
  cell_iterator &range_end) {…};

void MyProblem<dim>::assemble_system () {
  Threads::ThreadGroup<void> threads;

  std::vector<std::pair<cell_iterator, cell_iterator> >
    sub_ranges = Threads::split_range (
      dof_handler.begin_active(),
      dof_handler.end(),
      n_virtual_cores);

  for (t=0; t<n_virtual_cores; ++t)
    threads += Threads::new_thread (
      &MyProblem<dim>::assemble_on_cell_range,
      this,
      sub_ranges[t].first,
      sub_ranges[t].second);

  threads.join_all ();}
}
```

Independent threaded tasks

- How do we prevent data races?

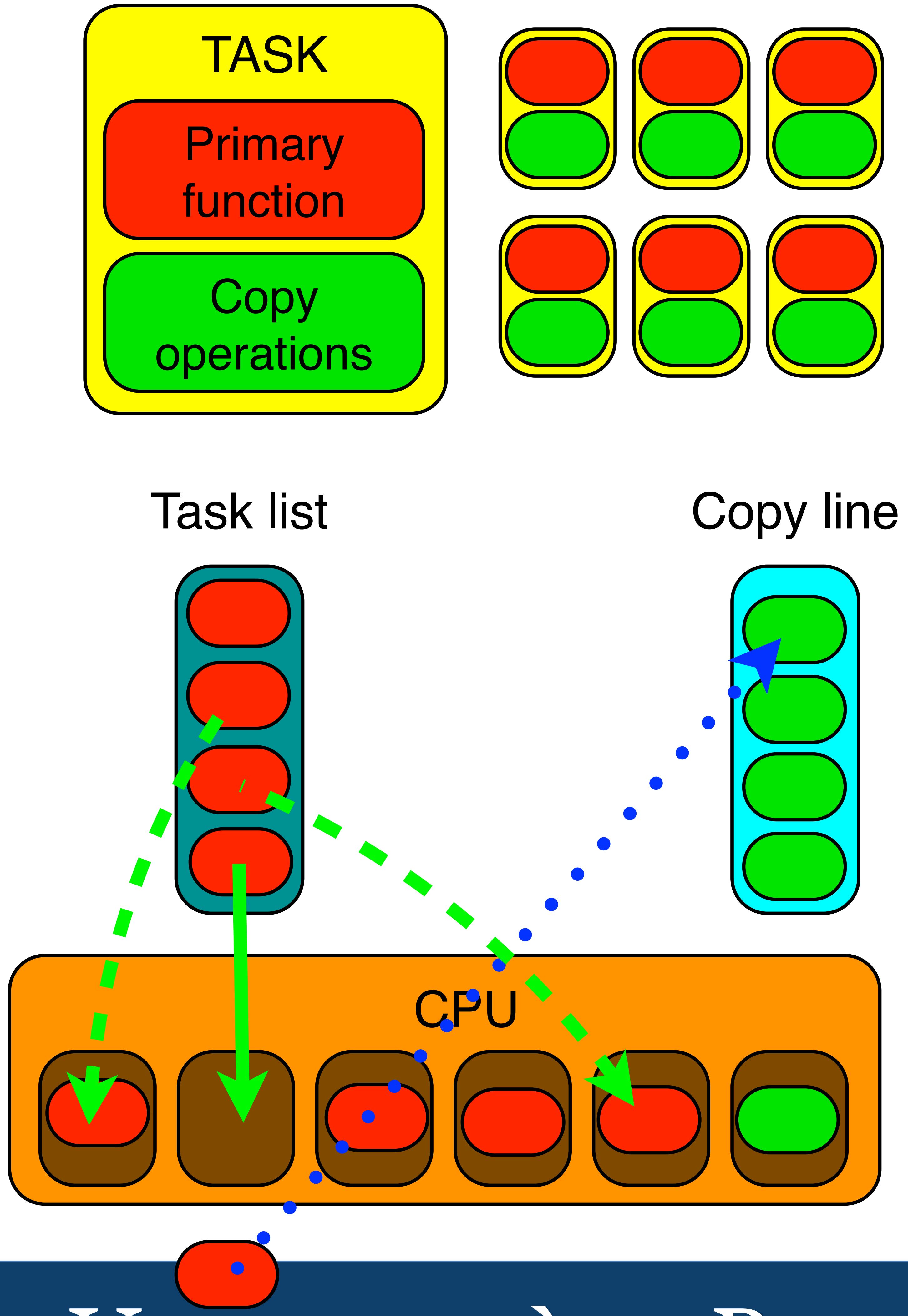
```
void MyProblem<dim>::assemble_on_one_cell (cell_iterator &cell) {

  static Threads::Mutex mutex;

  mutex.acquire ();
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix.add (dof_indices[i], dof_indices[j],
                         cell_matrix(i,j));
  ...same for rhs...
  mutex.release ();
}
```

Creating independent threaded tasks: the WorkStream class

- Task-based threading
  - Continuous use of free CPU cores
  - Create a list of tasks
  - When core free, use it to perform next task
    - Expensive operations continually executed
  - Perform blocking tasks independently
    - Data copied to shared objects serially
  - Optimizations:
    - “Automatic” load balancing
    - Overhead reduction: Works on data chunks



Creating independent threaded tasks: parallelization of (per-cell) assembly

```
template <int dim>
void MyClass<dim>::assemble_on_one_cell (
    const typename DoFHandler<dim>::active_cell_iterator &cell)
{
  FEValues<dim> fe_values (...);

  FullMatrix<double> cell_matrix (...);
  Vector<double>     cell_rhs (...);
  std::vector<double> rhs_values (...);

  rhs_function.value_list (...)

  // assemble local contributions
  fe_values.reinit (cell);
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      for (unsigned int q=0; q<n_points; ++q)
        cell_matrix(i,j) += ...;
  ...same for cell_rhs...

  // now copy results into global system
  std::vector<unsigned int> dof_indices (...);
  cell->get_dof_indices (dof_indices);
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix.add (...);
  ...same for rhs...
  // or constraints.distribute_local_to_global (...);
}
```

Expensive constructor call

Repeated memory allocation

Independent tasks

Serial operation

Threading using WorkStream: the ScratchData class

- Assistant struct / class
- Contains reused data structures
  - FEValues objects
  - Helper vectors and storage containers
  - Precomputed data
- Needs a constructor and a copy constructor
  - Some objects must be manually reconstructed
  - We create one initial instance of the class
  - TBB duplicates as required (queue_length)

```
struct ScratchData {
  std::vector<double>        rhs_values;
  FEValues<dim>              fe_values;

  ScratchData (
    const FiniteElement<dim> &fe,
    const Quadrature<dim>    &quadrature,
    const UpdateFlags         update_flags)
    : rhs_values (quadrature.size()),
      fe_values (fe, quadrature, update_flags)
    {}

   ScratchData (const ScratchData &rhs)
   : rhs_values (rhs.rhs_values),
     fe_values (rhs.fe_values.get_fe(),
                rhs.fe_values.get_quadrature(),
                rhs.fe_values.get_update_flags())
    {}
}
```

Threading using WorkStream: the PerTaskData class

- Contains data structures required for serial operations
  - Multiple copies made (queue_length*chunk_size)
  - Must be “self-contained”
- Used in two places
  - Threaded function
    - Bound to an instance of the threaded function
    - Used as a “data-in” object
  - Serial function
    - A used instance is passed to this function
    - Used as a “data-out” object

```
struct PerTaskData {
  FullMatrix<double>        cell_matrix;
  Vector<double>            cell_rhs;
  std::vector<unsigned int> dof_indices;

  PerTaskData (const FiniteElement<dim> &fe)
  : cell_matrix (fe.dofs_per_cell,
                 fe.dofs_per_cell),
    cell_rhs (fe.dofs_per_cell),
    dof_indices (fe.dofs_per_cell)
  {}
}
```

Threading using WorkStream: Revised assembly

- Now use objects contained within ScratchData and PerTaskData structs

```
template <int dim>
void MyClass<dim>::assemble_on_one_cell (
   const typename DoFHandler<dim>::active_cell_iterator &cell,
   ScratchData &scratch,
   PerTaskData &data)
{
  // reinitialise data
  scratch.fe_values.reinit (cell);
  rhs_function.value_list (scratch.fe_values.get_quadrature_points,
                           scratch.rhs_values);
  ...
  data.cell_matrix = 0;
  data.cell_rhs    = 0;

  // assemble local contributions
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      for (unsigned int q=0; q<fe_values.n_quadrature_points; ++q)
        data.cell_matrix(i,j) += ...;
  ...
}
```

Threading using WorkStream: Serial copy operation

- Uses writes “fixed” data in PerTaskData to single class object system_matrix (and whatever else)

```
template <int dim>
void MyClass<dim>::copy_local_to_global (const PerTaskData &data)
{
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix.add (data.dof_indices[i], data.dof_indices[j],
                         data.cell_matrix(i,j));
  ...same for rhs…
  // or constraints.distribute_local_to_global (...);
}
```

Threading (not) using WorkStream: Manual assembly using these data structures

- This performs the same serial assembly as we had before

- More efficient though (use of ScratchData)

```
ScratchData scratch_data (…);
PerTaskData per_task_data (…);

DoFHandler<deal_II_dimension>::active_cell_iterator
   cell = dof_handler.begin_active(),
   endc = dof_handler.end();
   for (; cell != endc; ++cell)
{
   assemble_system_one_cell(cell,
                            scratch_data,
                            per_task_data);
   copy_local_to_global(per_task_data);
}
```

Threading using WorkStream

- Execute function in threaded manner
- Only operates on functions with a specific prototype
  - Theadable function:
    void function_name(cell, scratch, per_task_data)
  - Serial function:
    void function_name(per_task_data)

```
ScratchData scratch_data (…);
PerTaskData per_task_data (…);

WorkStream::run ( dof_handler.begin_active(),
                  dof_handler.end(),
                  *this,
                  &MyClass::assemble_system_one_cell,
                  &MyClass::copy_local_to_global,
                  scratch_data,
                  per_task_data );
```

Workstream

https://www.dealii.org/current/doxygen/deal.II/namespaceWorkStream.html

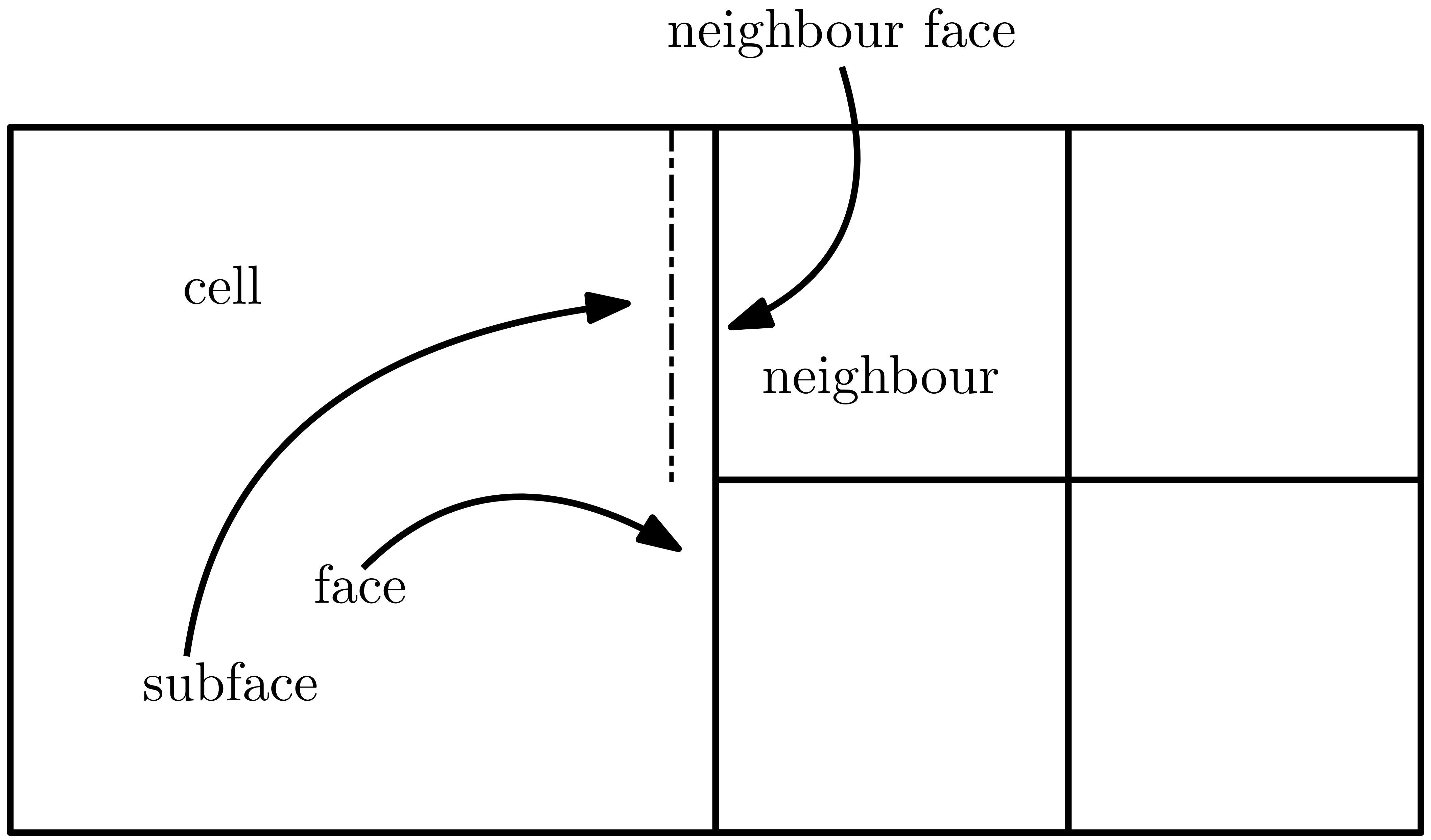
NOTE: If your data objects are large, or their constructors are expensive, it is helpful to keep in mind that queue_length copies of the ScratchData object and queue_length*chunk_size copies of the CopyData object are generated.

Specialisation for Grid-like containers

- If you need to assemble on a mesh like container where:
  - we need to work on *cells*
  - we need to work on *faces* (on the boundary)
  - we need to work on *facets* (faces between cells)
  - we have *hanging-nodes*
- *Then: your data objects will most likely always look the same*

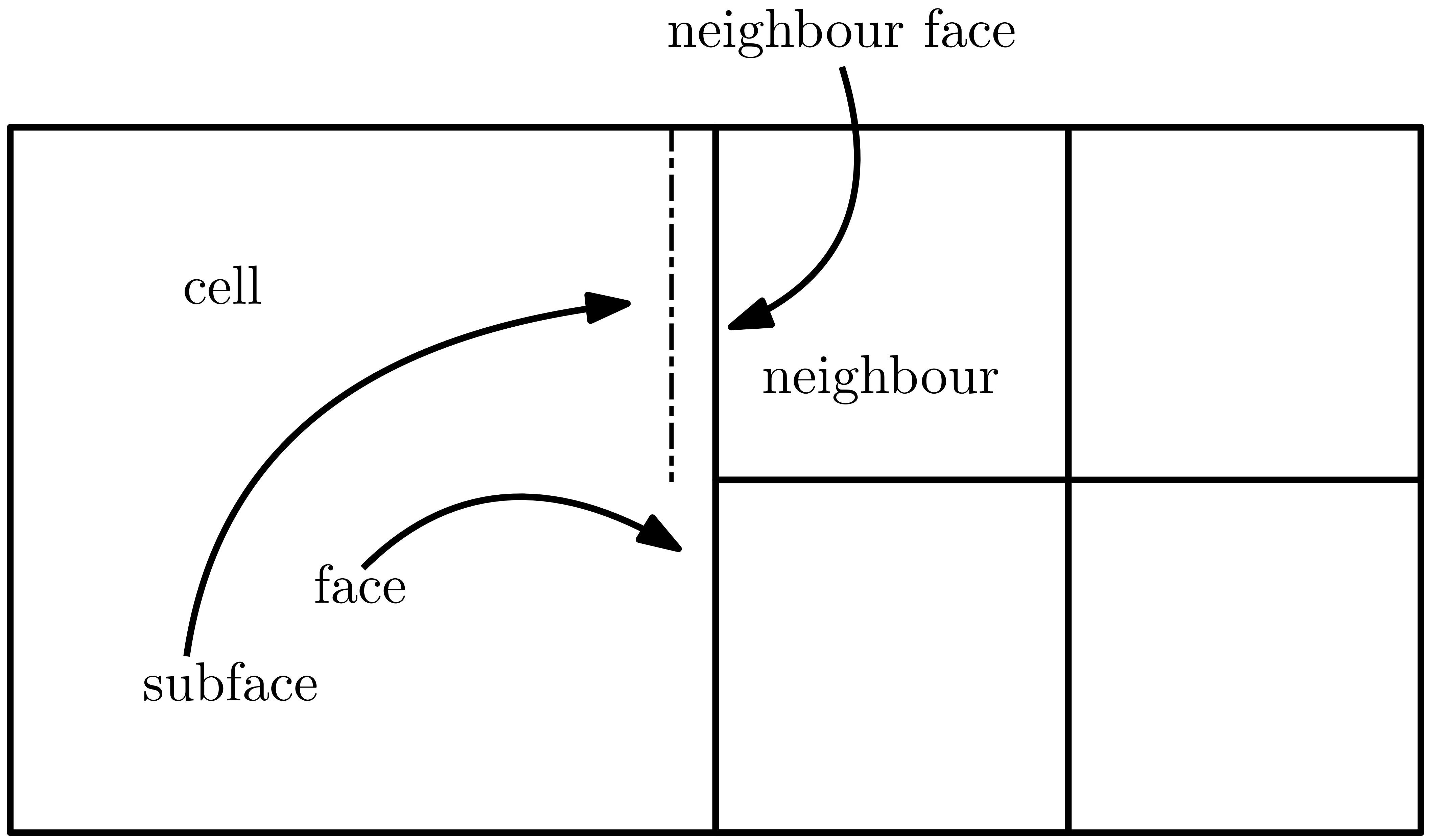
Automate works on mesh-like containers

- MeshWorker::ScratchData
- MeshWorker::CopyData
- MeshWorker::mesh_loop



Difficult part: assemble terms on faces

- To assemble terms between a cell and its neighbour, we need information about:
  - who is our neighbour on a given face?
  - what is the neighbour face index, w.r.t. to the neighbour cell?
  - is the neighbour finer?
    - if yes, what subface do I need to take on my face, to match his face?
  - is it coarser?
    - if yes, what are the face and subface indices we need to use on our neighbour to match our face?



FEValues, **FEFaceValues**,

FESubfaceValues, **FEInterfaceValues**

Automate works on mesh-like containers

- MeshWorker::ScratchData:
  - proxy for
    - **FEValues**
    - **FEFaceValues**
    - **FESubfaceValues**
    - **FEInterfaceValues**