



Shared memory parallelization

Denis Davydov (denis.davydov@fau.de)

Luca Heltai (luca.heltai@sissa.it)

25 February - 1 March 2019



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



Aims for this module

- Identify parts / blocks of code that are (easily) parallelizable
- Learn how to parallelize using
 - ThreadGroup (Posix threads)
 - Workstream (Threaded building blocks)

Reference material

- Tutorials
 - https://dealii.org/9.0.0/doxygen/deal.II/step_9.html
 - https://dealii.org/9.0.0/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/9.0.0/doxygen/deal.II/group_threads.html
 - <https://www.dealii.org/9.0.0/doxygen/deal.II/namespaceWorkStream.html>
 - <https://dealii.org/9.0.0/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

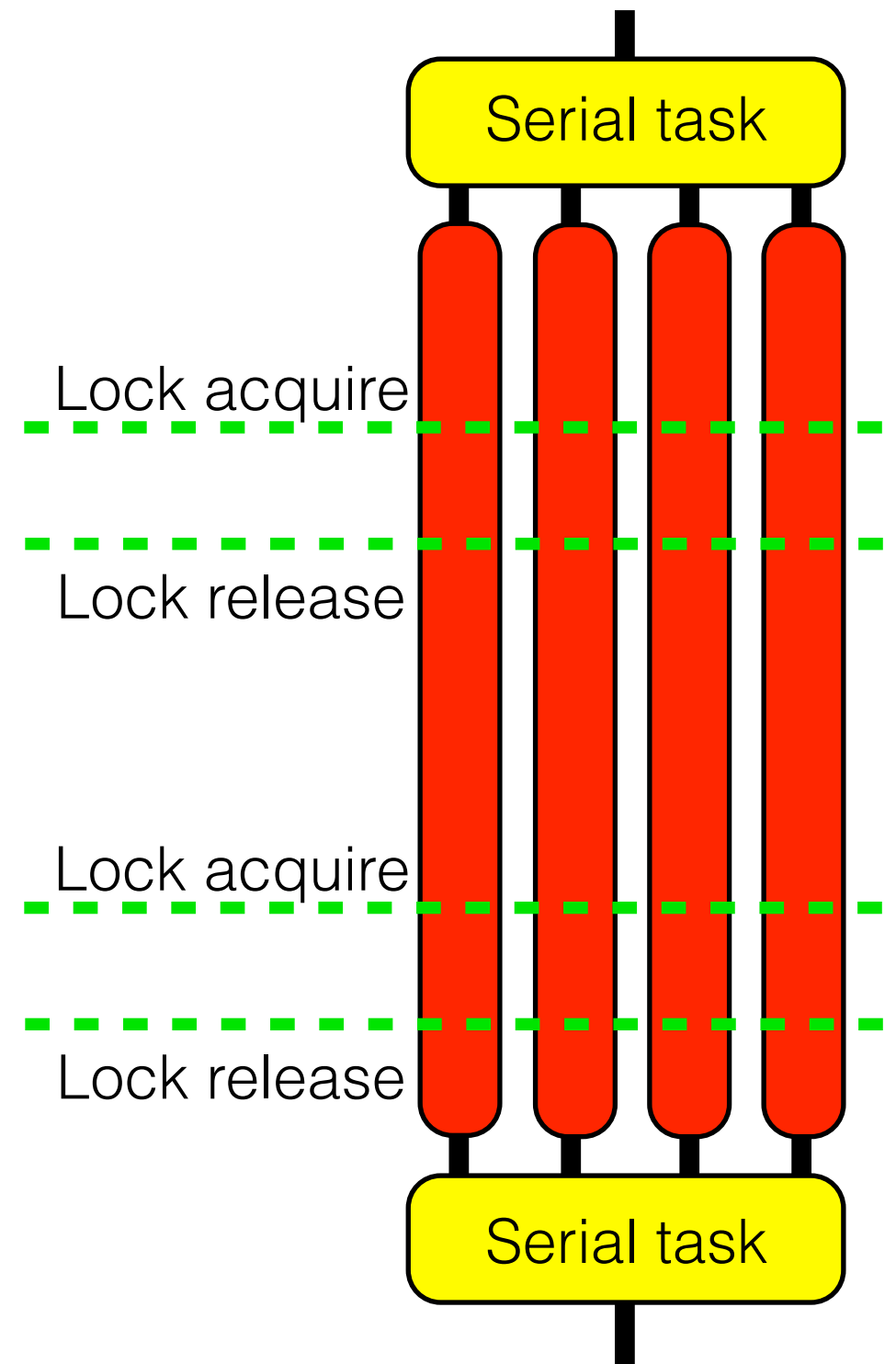
- “Embarrassingly parallelizable tasks”

```
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!**

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



Creating independent threaded tasks: the Thread class

```
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
    ...
}
```

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

Creating independent threaded tasks: the ThreadGroup class

```
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 ();
}
```

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

Creating independent threaded tasks: Ranged based assembly

- Less threads created = more efficient

```
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 ();
}
```

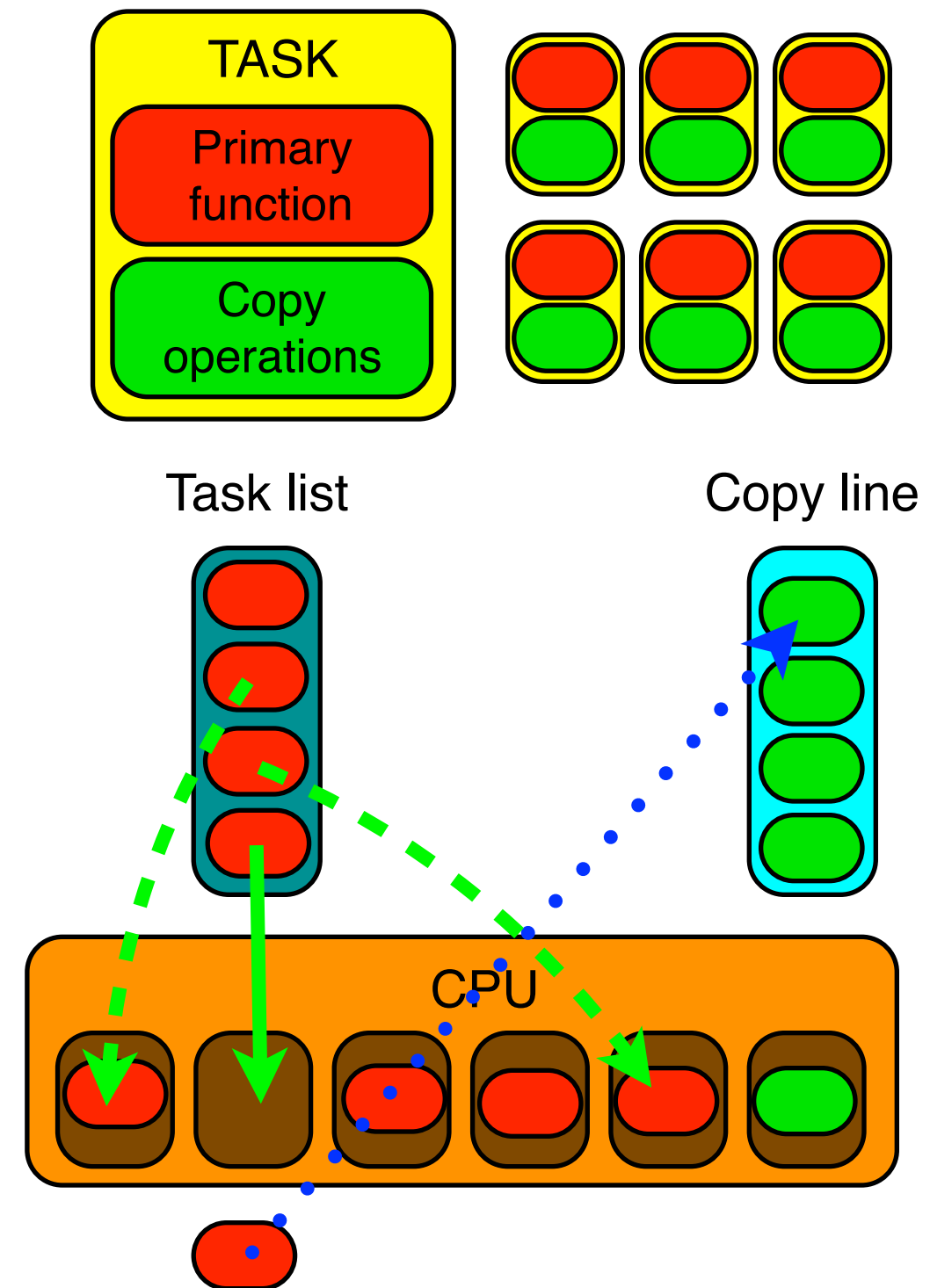
Independent threaded tasks

- 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 (...); ← Expensive constructor call

    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 (...);
}

```

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

```
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) += ...;

    ...
}
```

- Now use objects contained within ScratchData and PerTaskData structs

Threading using WorkStream: Serial copy operation

```
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 (...);
}
```

- Uses writes “fixed” data in PerTaskData to single class object system_matrix (and whatever else)
- Has to be a serially performed operation

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

```
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);
}
```

- This performs the same serial assembly as we had before
 - More efficient though (use of ScratchData)

Threading using WorkStream

```
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 );
```

- 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)

Threading using WorkStream

```
WorkStream::run ( dof_handler.begin_active(),
                  dof_handler.end(),
                  std::bind(&Solid::assemble_residual_one_cell,
                           this,
                           _1, // cell
                           _2, // scratch
                           _3), // data
                  std::bind(&Solid::copy_local_to_global_residual,
                           this,
                           _1, // data
                           &in_vector),
                  scratch_data,
                  per_task_data_residual);
```

- copy_local_to_global_F function prototype:
void function (per_task_data, vector)
- std::bind only binds memory addresses
 - Will make copies of objects not sent in via memory address
 - Need to send in pointers if wish to work on an existing object
 - “std::_1, _2, _3” are placeholders for expected data

[https://www.dealii.org/8.5.1/doxygen/deal.II/
namespaceWorkStream.html](https://www.dealii.org/8.5.1/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.