MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 40:

Parallelization on a single, shared memory machine

Example 1

Mutually independent parts of a program: Consider this (typical) 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
  …
}
```

Here, operations 1, 2, 3 are all independent of each other:

- Neither needs the results of the other
- We could re-order them as we please

Example 1

Mutually independent parts of a program:
Use explicit threads to parallelize this (pseudo-code):

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

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

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

Example 1

Mutually independent parts of a program:
Use explicit threads to parallelize this (pseudo-code).

Syntactic problems:

- Calling functions via *pthread_create* allows for no easy way to pass arguments
- No easy way to retrieve return values
- Generally rather low-level interface

Example 1

Mutually independent parts of a program:
Improved version using deal.II facilities:

```
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
  …
}
```

Example 1

Mutually independent parts of a program:
Use explicit threads to parallelize.

Other problems:

- Thread creation/termination is expensive
- # of threads does not match # of virtual cores
  – what if there is only one core on my machine?
  – what if there are 64?
- Hard to make scale to large # of threads
- Does not play nice in nested contexts

Example 2

“Embarrassingly parallel” parts of a program:

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

- There are *many* more cells than virtual cores.
- Computations on cells are mutually independent.

Example 2

“Embarrassingly parallel” parts of a program:
Using *many* explicit threads:

```
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 ();
}
```

Note: Not efficient. Too many threads.

Example 2

“Embarrassingly parallel” parts of a program:
Using *some number* of explicit threads:

```
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 ();
}
```

Example 2

“Embarrassingly parallel” parts of a program:
Using *many* explicit threads.

Problems:

- Does not play nice with nested parallelism
- Needs *synchronization*

Synchronization

Think about two threads copying local-to-global:

Thread 1 (on cell *K*)

```
for (i=0..dofs_per_cell)
  for (j=0..dofs_per_cell)
    A(local_dof_indices[i],
      local_dof_indices[j])
    += local_matrix(i,j);
```

Thread 2 (on cell *K*')

```
for (i=0..dofs_per_cell)
  for (j=0..dofs_per_cell)
    A(local_dof_indices[i],
      local_dof_indices[j])
    += local_matrix(i,j);
```

Imagine that by chance

- *local_dof_indices[i]* on *K* == *local_dof_indices[i]* on *K*'
- *local_dof_indices[j]* on *K* == *local_dof_indices[j]* on *K*'

Then: We have a *race condition*!

Race conditions

Assume, for example:

- *local_dof_indices[i]* on *K* and *K'* == 42
- *local_dof_indices[j]* on *K* and *K'* == 108

Thread 1 (on cell *K*)	Thread 2 (on cell *K'*)
A(42,108) += local_matrix(i,j);	A(42,108) += local_matrix(i,j);

Both threads need to:

- Read *A(42,108)* from memory into the processor
- Add their local contribution
- Write the sum from processor into memory at *A(42,108)*

Note: This sequence is not *atomic*.

Race conditions

Definition:

A *race conditions* is a situation where
the result of a program depends on the
relative timing of instructions of multiple threads.

Race conditions happen when multiple threads:

- Execute *read-modify-write* sequences
- In a non-atomic way
- On the same memory location

Race conditions can not happen if either

- All threads only read data, or
- The operation is atomic, or
- Operations happen on different memory locations

Race conditions

Definition:

A *race conditions* is a situation where
the result of a program depends on the
relative timing of instructions of multiple threads.

Race conditions happen when multiple threads:

- Execute *read-modify-write* sequences
- In a non-atomic way
- On the same memory location

Let us simply look at an example...

Race conditions

Race conditions happen when multiple threads:

- Execute *read-modify-write* sequences
- In a non-atomic way
- On the same memory location

Avoiding race conditions:

- Make data access mutually exclusive (“mutex”)
- Each thread
  – copies global data structures
  – works on its own copy
  – mother thread later merges results
- Split work by memory location

Race conditions

Avoiding race conditions:

- Make data access mutually exclusive (“mutex”)
- Each thread
  – copies global data structures
  – works on its own copy
  – mother thread later merges results
- Split work by memory location

Let us just look at an example again...

Race conditions

Avoiding race conditions:

- Make data access mutually exclusive (“mutex”)
- Each thread
  – copies global data structures
  – works on its own copy
  – mother thread later merges results
- Split work by memory location

Note:

- Strategy 1 involves non-parallelizable code parts
- Strategy 2 involves duplicate work

These are included in the **sequential component of Amdahl's law**!

Explicit threads summary

Basic premise:

- Keep *large data structures* only once
- Split *work* onto multiple threads
- Operating system will schedule threads to (virtual) cores

Goals:

- # of threads = # of cores
- All phases of a program use threads
- All threads work for their entire lifetime

Problems:

- # of threads hard to control
- Requires synchronization
- Synchronization triggers Amdahl's law

Explicit threads summary

Experience on using threads explicitly:

- Too low level:
  – hard to match # of threads to # of virtual cores
  – requires explicit synchronization
  – hard to parallelize *all* operations

- Better approach:
  – dissociate what we want to do from actual realization
  – describe *tasks*, not *threads*
  – use higher-level programming interfaces

Tasks

Consider again this code piece:

```
template <int dim>
void MyProblem<dim>::setup_system ()
{
  dof_handler.distribute_dofs();                          // 1
  DoFTools::make_hanging_node_constraints (…);   // 2
  DoFTools::make_sparsity_pattern (…);                 // 3
  VectorTools::interpolate_boundary_values (…);     // 4
  constraints.condense (sparsity_pattern);                // 5
}
```

This describes “tasks” with some dependencies:

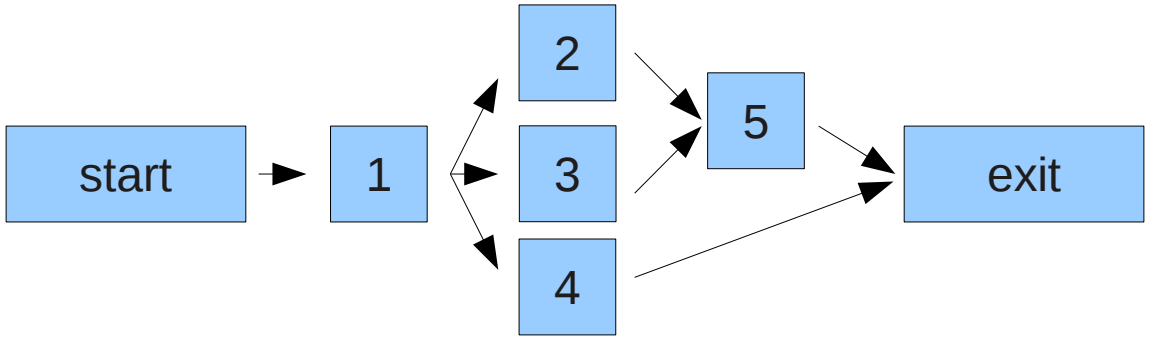
start->1
1->2, 1->3, 1->4
2->5, 3->5
4->exit, 5->exit

Tasks

Consider again this code piece:

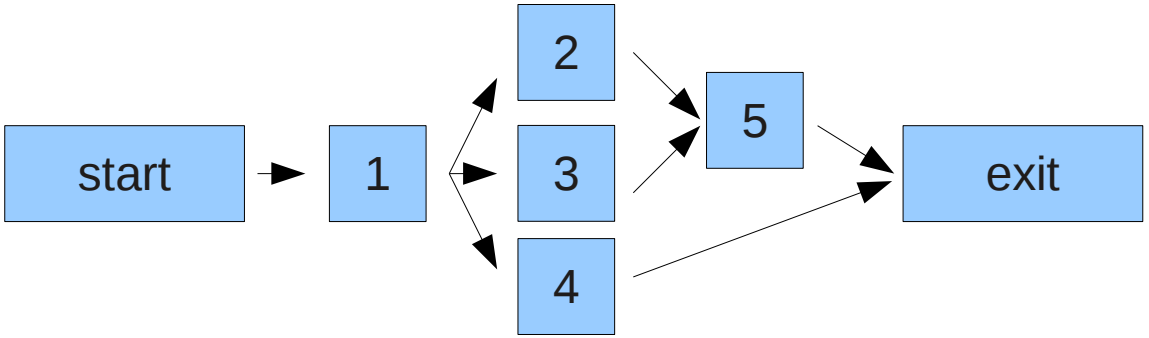
```
template <int dim>
void MyProblem<dim>::setup_system ()
{
  dof_handler.distribute_dofs();                          // 1
  DoFTools::make_hanging_node_constraints (…);   // 2
  DoFTools::make_sparsity_pattern (…);                 // 3
  VectorTools::interpolate_boundary_values (…);     // 4
  constraints.condense (sparsity_pattern);                // 5
}
```

Dependencies:
Representation as a *directed acyclic graph* (*DAG*)



Tasks

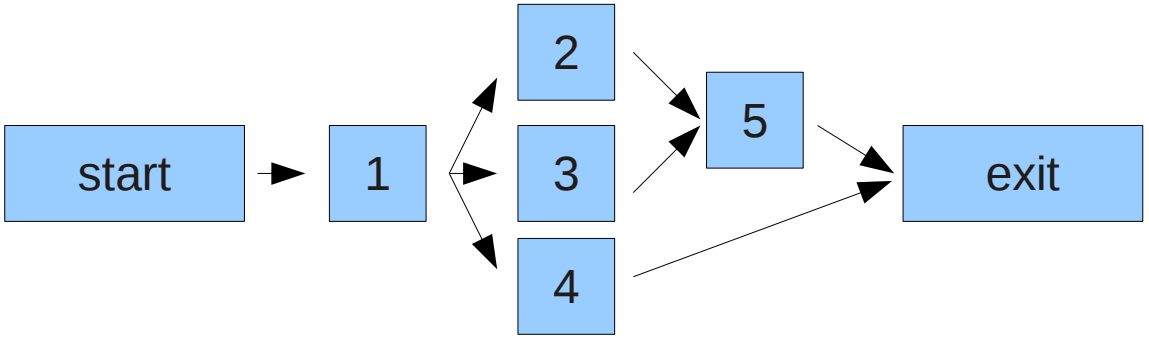
About DAGs:



- Every instruction sequence can be represented as a DAG
- In practice, the DAG is too large:
  – need to look at functions (groups), not instructions
  – need to look at parts of the entire DAG

Tasks

What to do with a DAG?



- Can be represented with two lists:
  – not-ready nodes (with unfinished upstream links)
  – ready nodes (with downstream links)
- When a ready node has been executed, remove upstream link from all downstream not-ready nodes
- When a not-ready node has no more upstream links, move it to the ready list
- Scheduler maps ready nodes to threads/virtual cores

Tasks

Example (semi-pseudo code):

```
template <int dim>
void MyProblem<dim>::setup_system ()
{
  Threads::Task<void> t1, t2, t3, t4, t5;

  t1 = Threads::new_task (dof_handler, distribute_dofs, ...);
  t2 = t1.enqueue (DoFTools::make_hanging_node_constraints, …);
  t3 = t1.enqueue (DoFTools::make_sparsity_pattern, …);
  t4 = t1.enqueue (VectorTools::interpolate_boundary_values (…);
  t5 = (t2,t3).enqueue (constraints, condense, sparsity_pattern);

  t4.join ();
  t5.join ();
}
```

Tasks

Why tasks instead of threads:

- Scheduler maps “ready” tasks to available worker threads
- No need to explicitly manage # of threads
- # of currently executing tasks is never greater than # of threads (== # of virtual cores)
- Easier to describe dependencies
- No overhead for creating/destroying threads

Embarrassingly parallel tasks

Consider again example 2:

```
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_task (&MyProblem<dim>::assemble_on_one_cell,
                       this,
                       cell);
  threads.join_all ();
}
```

This is now possible (if inefficient with 1000s of cells).
Synchronization is still necessary in copy-local-to-global.

Embarrassingly parallel tasks

Higher-level strategies:

- Split *assemble_on_one_cell* into two parts:
  – *local_assemble*
  – *copy_local_to_global*
- One task of each kind for each cell
- Tell scheduler that only one *copy_l_to_g* task can run at any given time (no synchronization necessary!)
- Create tasks for new cells on the fly as ready-list runs low on tasks

Implementation: deal.II's *WorkStream* class, based on the pipeline model.

Usage: Assembly, *DataOut*, *KellyErrorEstimator*, ...

Shared memory parallelization

Summary:

- Explicit threads are too low level:
  – hard to match # of threads to # of virtual cores
  – requires explicit synchronization
- Task-based description
  – better at matching # of threads to # of virtual cores
  – can avoid some explicit synchronization
  – provides higher-level interfaces
- Either way, it is hard to parallelize *all* operations
- Some kind of synchronization is necessary for shared memory operations
- Copying data structures may be necessary but unpopular

Shared memory parallelization

More information:

- deal.II documentation module “*Parallel computing with multiple processors accessing shared memory*”
- Documentation of the *WorkStream* class
- Documentation of the *parallel::transform(), parallel::apply_to_subranges()*, … functions
- step-9, step-32, step-35, step-44

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University