Using multicore machines with deal.II

Wolfgang Bangerth
Department of Mathematics
Texas A&M University

Background

My laptop has 4 (hyperthreaded) cores.
Your workstation probably has many more.
If it hasn't, it definitely will in a couple years

Let's try to use them all.

Background

Finite element codes almost always have several things to do at once:

- Assemble the linear system on one cell while doing the same on a different cell
- Do the matrix-vector product with one matrix row while doing the same on a different row
- Generate graphical output while estimating the error
- Estimate the error on one cell while also estimating it on a different cell.

How can we do these kinds of things in parallel?

Overview

There are two basic approaches:

- Use threads with explicit creation/destruction
- Use tasks and a scheduler

Approach 1: Threads

Definition:

- A program can be split into several different threads
- Threads run in parallel or, if there are too many, are scheduled onto available processors
- All threads that belong to a single program have access to the same memory locations
- Threads need to be explicitly created and we can wait for their demise; for example using
  - Threads::new_thread
  - Threads::Thread::join

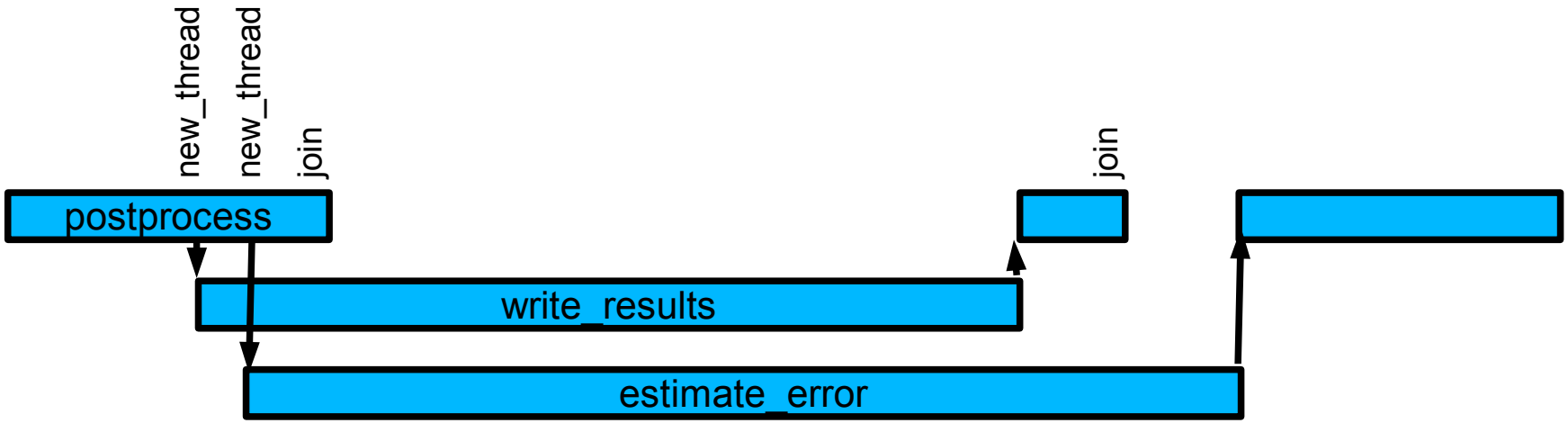
Approach 1: Threads

Example:

```
void MyApp::postprocess ()
{
  Threads::Thread<void>
   thread_1 = Threads::new_thread (&MyApp::write_results,
                                   *this),

   thread_2 = Threads::new_thread (&MyApp::estimate_error,
                                   *this);

  Thread_1.join ();
  thread_2.join ();
}
```



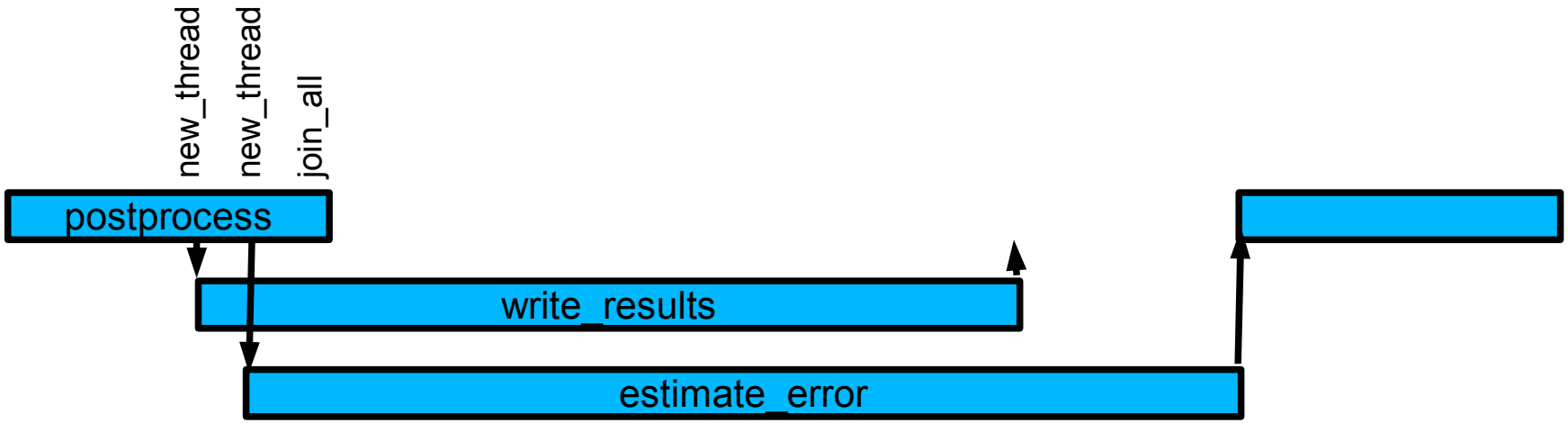
Approach 1: Threads

Example (slightly simpler):

```
void MyApp::postprocess ()
{
  Threads::ThreadGroup<void> tg;
  tg += Threads::new_thread (&MyApp::write_results,
                             *this),

  tg += Threads::new_thread (&MyApp::estimate_error,
                             *this);

  tg.join_all ();
}
```



Approach 1: Threads

Problem 1 with threads:

- Thread creation is expensive: you can do this when calling large functions, but you don't want to do this for every cell

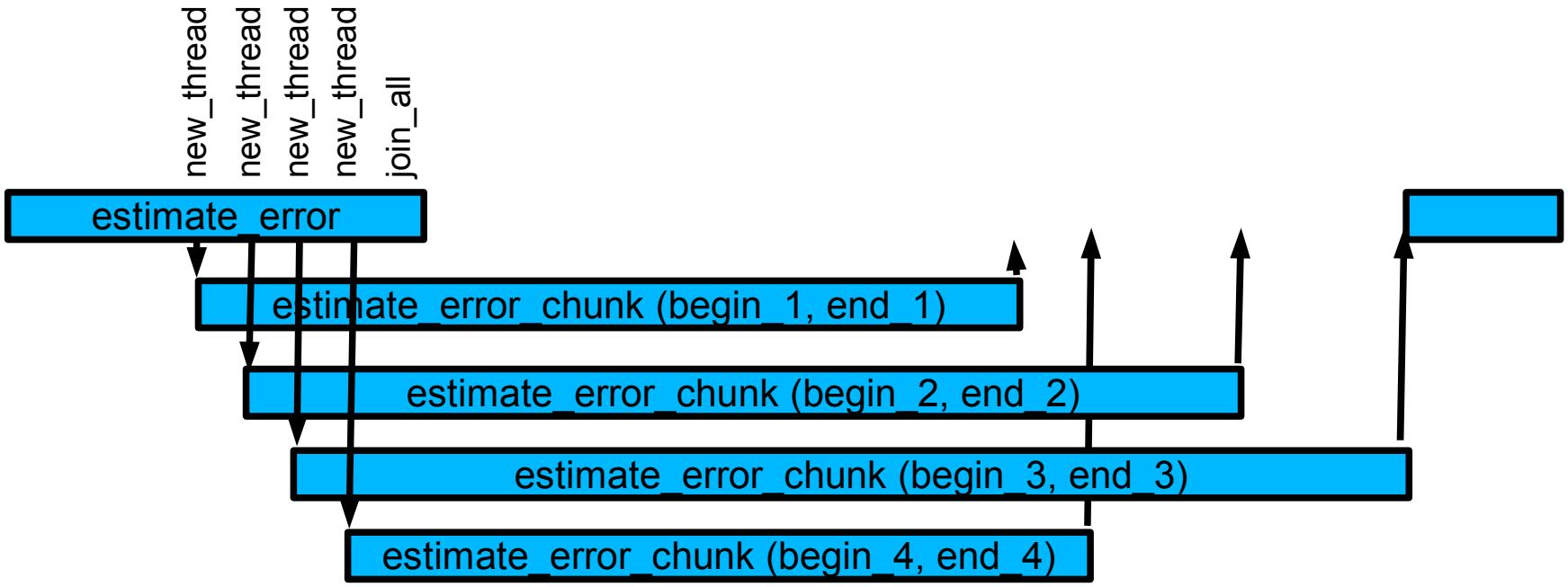
Unfortunately, much of the parallelism in finite element programs comes from many small tasks (integration over cells, matrix-vector multiplication with individual matrix rows).

Approach 1: Threads

Problem 2 with threads:

- Breaking many tasks into a small number of “chunks” and putting them onto individual threads does not usually load balance very well

This is because not every integration over a cell costs equally much time.

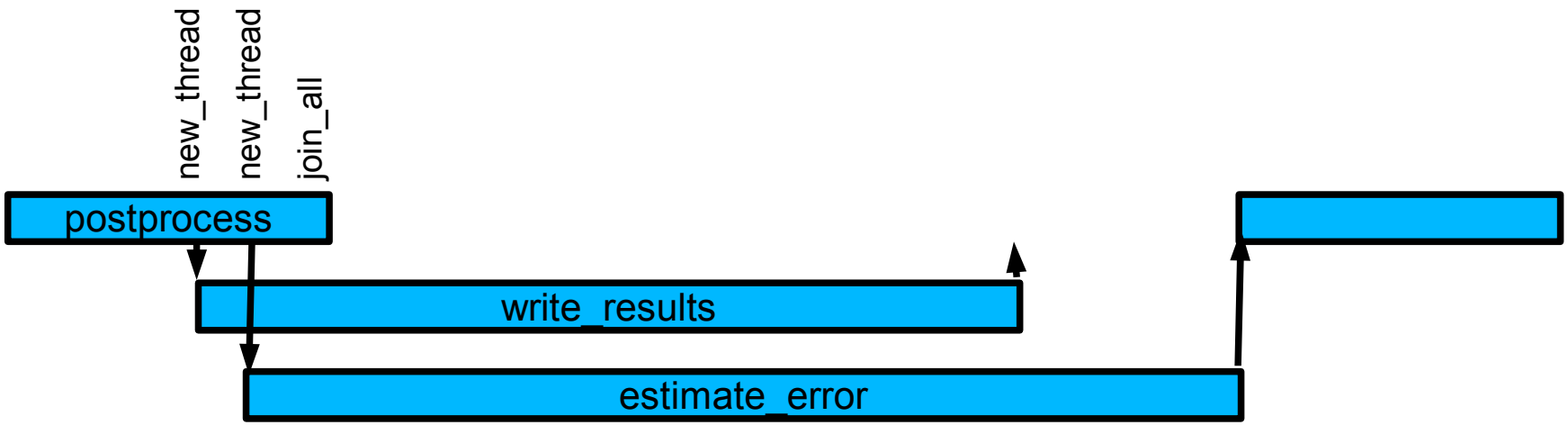


Approach 1: Threads

Problem 3 with threads:

- It doesn't usually take into account how many processors you really have

Example: Assume you have only 1 processor, then starting threads altogether is inefficient.

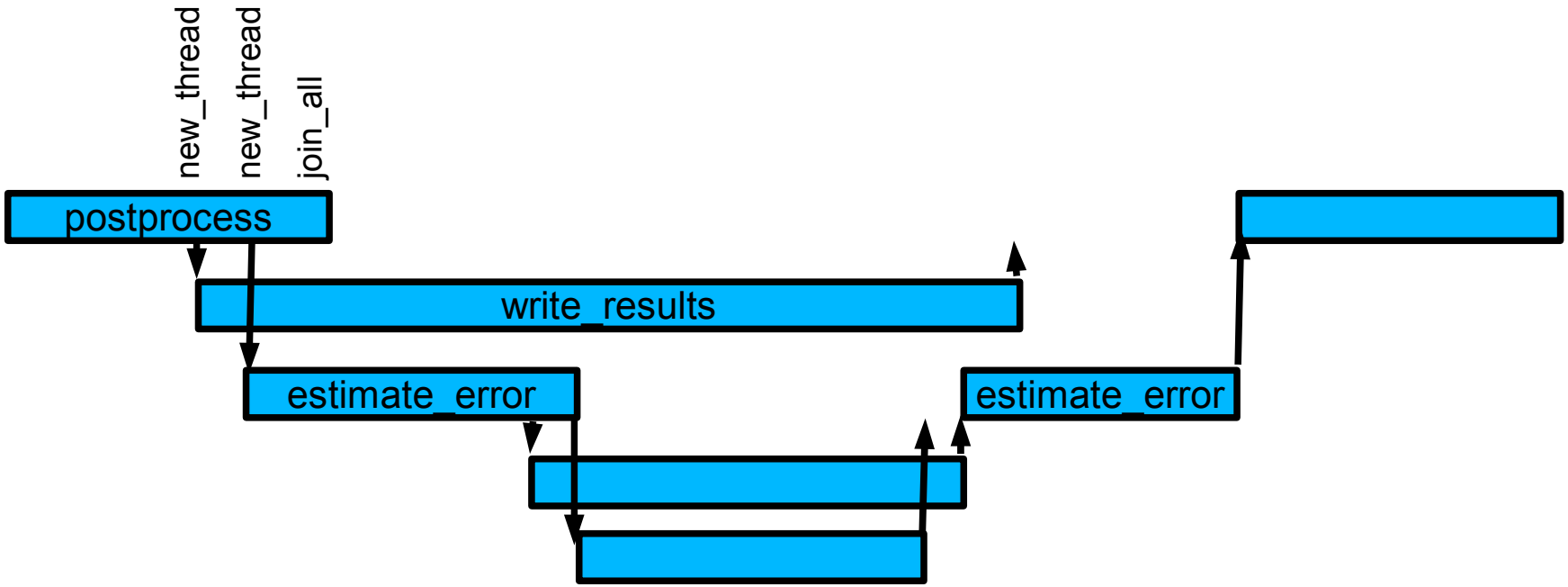


Approach 1: Threads

Problem 3 with threads:

- It doesn't usually take into account how many processors you really have

Example (variant): Assume you have only 2 processors, then starting further threads in estimate_error is inefficient.



The basic problem is that estimate_error is unaware of the number of threads running overall.

Approach 2: Tasks

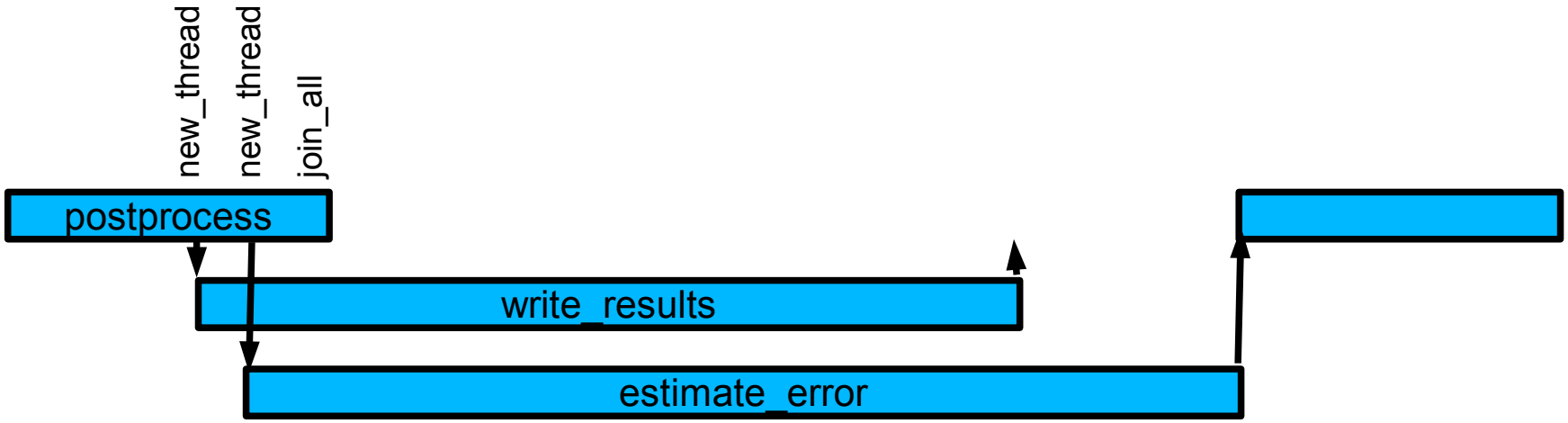
Definition:

- A task is something that needs to be done, now or later
- Tasks are much finer grained than what you would typically create a thread for
- A scheduler creates as many threads as there are cores on your machine and schedules tasks to threads
- Tasks are described to the scheduler using
  - Threads::new_task

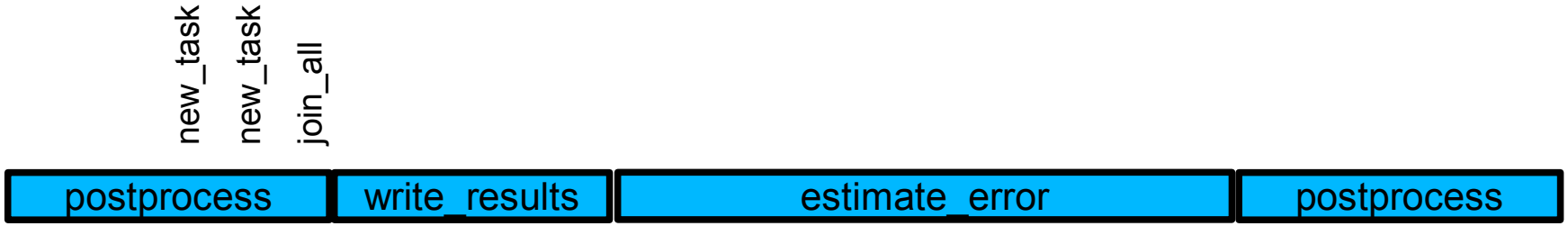
  You can wait for a task to be finished using
  - Threads::Task::join

Approach 2: Tasks

Problem 3 with threads (revisited): Assume you have only 1 processor, then starting threads altogether is inefficient.

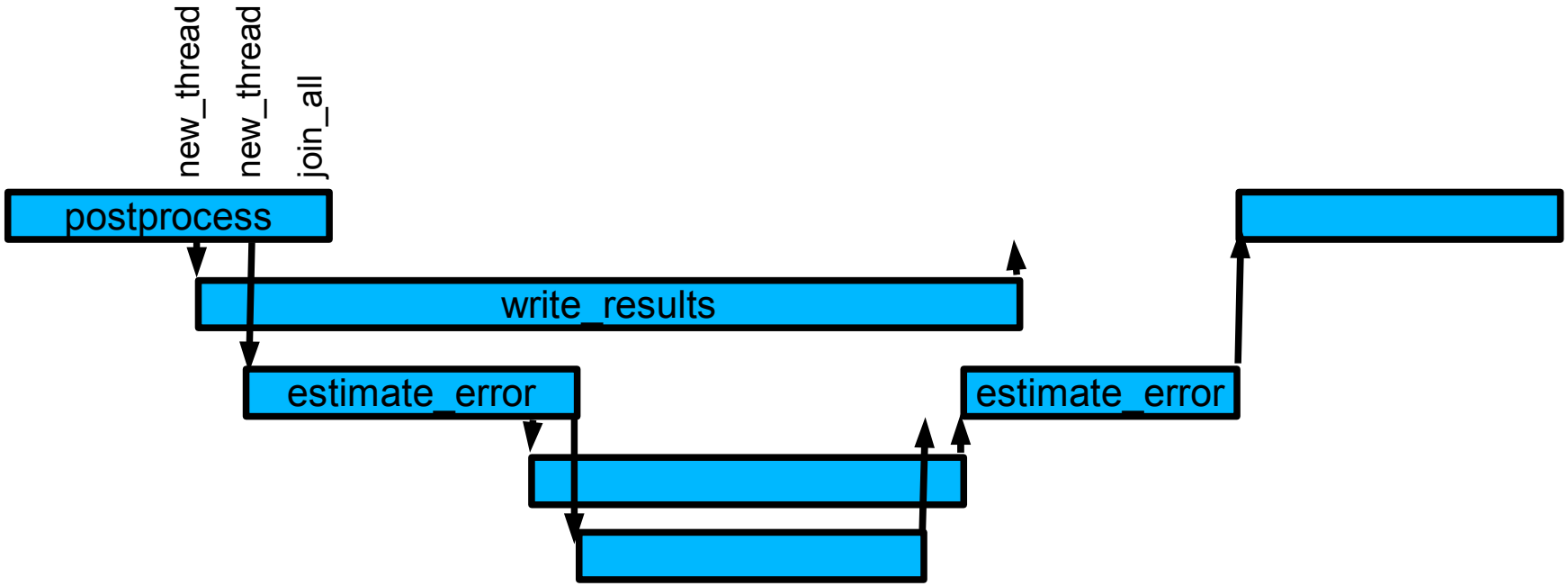


Using new_task instead of new_thread would yield the following execution schedule:

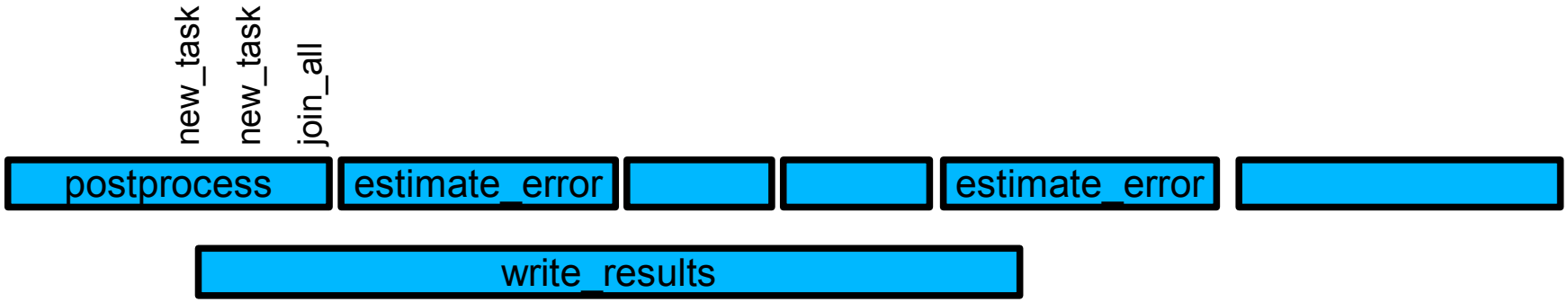


Approach 2: Tasks

Problem 3 (variant) with threads revisited: Assume you have only 2 processors, then starting further threads in estimate_error is inefficient.



With tasks this would execute as follows:



Tasks with simple loops

Example: Consider the following vector addition loop

```
void Vector::operator += (const Vector &v)
{
  for (unsigned int i=0; i<size(); ++i)
    elements[i] += v.elements[i];
}
```

This is an embarrassingly parallel problem, and we could create a task for each vector element.

But that would of course be inefficient: Scheduling these tasks would be much more expensive than executing them.

Tasks with simple loops

Example: Consider the following vector addition loop

```
void Vector::operator += (const Vector &v)
{
  for (unsigned int i=0; i<size(); ++i)
    elements[i] += v.elements[i];
}
```

Instead, use something like this:

```
void add_to (double *result, double *rhs)
{
  *result += *rhs;
}

void Vector::operator += (const Vector &v)
{
  parallel::transform (this->begin(), this->end(),
                       v.begin(),
                       &add_to);
}
```

Tasks with simple loops

How *parallel::transform* works:

- Take one half of the entire range *(begin, end)* and divide it by the number of processors available
- Make one task out of each of these blocks
- Take one half of the rest and divide it into tasks
- Take one half of the rest and divide it into tasks, etc
- Until blocks become smaller than a certain threshold

This is divide-and-conquer. It guarantees reasonably good processor utilization.

Tasks with synchronization: WorkStream

Sometimes tasks (and threads) step on each others' feet:

```
void assemble_on_cell (const cell_iterator &cell)
{
  FullMatrix cell_matrix;
  … assemble … assemble … assemble …

  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix(global_i,global_j) += cell_matrix(i,j);
}

void assemble ()
{
  parallel::for_all (dof_handler.begin_active(),
                     dof_handler.end(),
                     &assemble_on_cell);
}
```

Note the load and store into the system matrix. This can't work with multiple threads.

Tasks with synchronization: WorkStream

We would need to write this as follows:

```
void assemble_on_cell (const cell_iterator &cell)
{
  FullMatrix cell_matrix;
  … assemble … assemble … assemble …

  static 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(global_i,global_j) += cell_matrix(i,j);
  mutex.release ();
}

void assemble ()
{
  parallel::for_all (dof_handler.begin_active(),
                     dof_handler.end(),
                     &assemble_on_cell);
}
```

Tasks with synchronization: WorkStream

Problems with this approach:

- Explicit synchronization with mutices is expensive
- It does not scale well to when we will all have 256 cores
- Even if these were non-issues, it still has a problem: The order in which we add to the matrix is undetermined, but in floating point arithmetic

  a+b+c

  is not the same as

  a+c+b

  That means that we won't get the same matrix twice in a row.

Tasks with synchronization: WorkStream

Solution: We have to separate

- the embarrassingly parallel and independent computation of local contributions, and
- the reduction operation of adding to the global matrix

into two parts. Furthermore:

- The reduction operation should only run on a single thread to avoid explicit synchronization
- The reduction operation should always run in exactly the same order
- The local computation can run in any order and in parallel

This is exactly what the *WorkStream* class does.

Tasks with synchronization: WorkStream

Conceptually, code will then look as follows:

```
void assemble_on_cell (const cell_iterator &cell,
                       CopyData &copy_data)
{
  … assemble … fill copy_data.cell_matrix … assemble …
}

void copy_local_to_global (const CopyData &copy_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(global_i,global_j) += cell_matrix(i,j);
}

void assemble ()
{
  WorkStream::run (dof_handler.begin_active(),
                   dof_handler.end(),
                   &assemble_on_cell,
                   &copy_local_to_global);
}
```

Tasks with synchronization: WorkStream

In practice:

- *WorkStream* does not work on individual cells but gives each processor several at a time (e.g. 8)
- Guarantees that the result is the same every time, independently on the order in which cells are assembled
- Is currently used in *DataOut*, *KellyErrorEstimator*, *MatrixCreator*
- *step-32, step-35, step-37*
- Should at one point be used in *MeshWorker*

For now:

Make thinking in parallel your default!

Help parallelize more parts of deal.II!

(And read the documentation in the "Threads/Tasks" module.)