Multi-threading support in deal.II

Wolfgang Bangerth
University of Heidelberg

March 2000

Abstract

In this report, we describe the implementational techniques of multi-threading support in deal.II, which we use for the parallelization of independent operations. Writing threaded programs in C++ is obstructed by two problems: operating system dependent interfaces and that these interfaces are created for C programs rather than for C++. We present our solutions to these problems and describe first experiences using multi-threading in deal.II.

1 Background

Realistic finite element simulations tend to use enormous amounts of computing time and memory. Scientists and programmers have therefore long tried to use the combined power of several processors or computers to tackle these problems.

The usual approach is to use physically separated computers (e.g. clusters) or computing units (e.g. processor nodes in a parallel computer), each of which is equipped with its own memory, and split the problem at hand into separate parts which are then solved on these computing units. Unfortunately, this approach tends to pose significant problems, both for the mathematical formulation as well as for the application programmer, which make the development of such programs overly difficult and expensive.

For these reasons, parallelized implementations and their mathematical background are still subject to intense research. In recent years, however, multi-processor machines have been developed, which pose a reasonable alternative to small parallel computers with the advantage of simple programming and the possibility to use the same mathematical formulation that can also be used for single-processor machines. These computers typically have between two and eight processors that can access the global memory at equal cost.

Due to this uniform memory access (UMA) architecture, communication can be performed in the global memory and is no more costly than access to any other memory location. Thus, there is also no more need to change the mathematical formulation to reduce communication, and programs using this architecture look very much like programs written for single processor machines.

The purpose of this report is to explain the techniques used in deal.II (see [1, 2]) by which we try to program these computers. We will first give a brief introduction in what threads are and what the problems are which we have to solve when we want to use multi-threading. The third section takes an in-depth look at the way in which the functionality of the operating system is represented in a C++ program in order to allow simple and robust programming; in particular, we describe the design decisions which led us to implement these parts of the library in the way they are implemented. In the fourth section, we show several examples of parallelization and explain how they work. Readers who are more interested in actually using the framework laid out in this report, rather than the internals, may skip Section 3 and go directly to the applications in Section 4 (page 15).

2 Threads

The basic entity for programming multi-processor machines are *threads*. They represent parts of the program which are executed in parallel. Threads can be considered as separate programs that work on the same main memory. On single-processor machines, they are simulated by letting each

thread run for some time (usually a few milliseconds) before switching to the next thread. On multi-processor machines, threads can truly be executed in parallel. In order to let programs use more than one thread (which would be the regular sequential program), several aspects need to be covered:

- How do we assign operations to different threads? Of course, operations which depend on each other must not be executed in reverse order. This can be achieved by only letting independent operations run on different threads, or by using synchronization methods. this is mostly a question of program design and thus problem dependent, which is why both aspects will only be briefly touched below.
- How does the operating system and the whole programming environment support this?

As mentioned, only the second aspect can be canonicalized, so we will treat it first. Some examples of actual parallelized applications are discussed in Section 4.

3 Creating and managing threads

3.1 Operating system dependence and ACE

While all relevant operating systems now support multi-threaded programs, they all have different notions on what threads actually are on an operating system level, how they shall be managed and created. Even on Unix systems, which are usually well-standardized, there are at least three different and mutually incompatible interfaces to threads: POSIX threads [6], Solaris threads [5], and Linux threads. Some operating systems support more than one interface, but there is no interface that is supported by all operating systems. Furthermore, other systems like Microsoft Windows have interfaces that are incompatible to all Unix systems [3].

Writing multi-threaded programs based on the operating system interfaces is therefore something inherently incompatible unless much effort is spent to port it to a new system. To avoid this, we chose to use the ACE (Adaptive Communication Environment, see [7, 8, 4]) library which encapsulates the operating system dependence and offers a uniform interface to the user.

We chose ACE over other libraries, since it runs on almost all relevant platforms, including most Unix systems and Microsoft Windows, and since it is to our knowledge the only library which is actively developed by a large group. Furthermore, it also is significantly larger than only thread management, offering interprocess communication and communication between different computers, as well as many other services. Contrary to most other libraries, it therefore offers both the ability to support a growing `deal.II` as well as the prospect to support independence also with respect to future platforms.

3.2 `C` interface to threads versus `C++`

While ACE encapsulates almost all of the synchronization and interprocess interface into `C++` classes, it for some reason does not do so for thread creation. Rather it only offers the basic `C` interface: when creating a new thread, a function is called which has the following signature:

Code sample 1

```
void *  f (void * arg);
```

Thus, only functions which take a single parameter of type `void*` and return a `void*` may be called. Further, these functions must be global or static member functions, as opposed to true member functions of classes. This is not in line with the `C++` philosophy and in fact does not fit well into `deal.II` as well: there is not a single function in the library that has this signature.

The task of multi-threading support in `deal.II` is therefore to encapsulate member functions, arbitrary types and numbers of parameters, and return types of functions into mechanisms built atop of ACE. This has been done twice for `deal.II`, and we will explain both approaches. At

present, i.e. with version 3.0, only the first approach is distributed with deal.II, since the second is still experimental and due to the high complexity. The latter approach, however, has clear advantages over the first one, and it is planned to switch to it in the next major version of deal.II.

3.3 First approach

The first idea is the following: assume that we have a class TestClass

Code sample 2

```
class TestClass {
  public:
    void test_function (int i, double d);
};
```

and we would like to call test_object.test_function(1,3.1415926) on a newly created thread, where test_object is an object of type TestClass. We then need an object that encapsulates the address of the member function, a pointer to the object for which we want to call the function, and both parameters. This class would be suitable:

Code sample 3

```
struct MemFunData {
    typedef void (TestClass::*MemFunPtr) (int, double);
    MemFunPtr  mem_fun_ptr;
    TestClass *object;
    int        arg1;
    double     arg2;
};
```

We further need a function that satisfies the signature required by the operating systems (or ACE, respectively), see Code Sample 1, and that can call the member function if we pass it an object of type MemFunData:

Code sample 4

```
void * start_thread (void *arg_ptr) {
                    // first reinterpret the void* as a
                    // pointer to the object which
                    // encapsulates the arguments
                    // and addresses:
  MemFunData *mem_fun_data
        = reinterpret_cast<MemFunData *>(arg_ptr);
                    // then call the member function:
  (mem_fun_data->object)
        ->*(mem_fun_data->mem_fun_ptr) (mem_fun_data->arg1,
                                        mem_fun_data->arg2);
                    // since the function does not return
                    // a value, we do so ourselves:
  return 0;
};
```

Such functions are called *trampoline functions* since they only serve as jump-off point for other functions.

We can then perform the desired call using the following sequence of commands:

```
MemFunData mem_fun_data;
mem_fun_data.mem_fun_ptr = &TestClass::test_function;
```

```
mem_fun_data.object      = &test_object;
mem_fun_data.arg1        = 1;
mem_fun_data.arg2        = 3.1415926;

ACE_Thread_Manager::spawn (&start_thread,
                           (void*)&mem_fun_data);
```

`ACE_Thread_Manager::spawn` is the function from ACE that actually calls the operating system and tells it to create a new thread and call the function which it is given as first parameter (here: `start_thread`) with the parameter which is given as second parameter. `start_thread`, when called, will then get the address of the function which we wanted to call from its parameter, and call it with the values we wanted as arguments.

In practice, this would mean that we needed a structure like `MemFunData` and a function like `start_thread` for each class `TestClass` and all functions `test_function` with different signatures. This is clearly not feasible in practice and places an inappropriate burden on the programmer who wants to use multiple threads in his program. Fortunately, `C++` offers an elegant way for this problem, in the form of templates: we first define a data type which encapsulates address and arguments for all binary functions:

Code sample 5

```
template <typename Class, typename Arg1, typename Arg2>
struct MemFunData {
    typedef void (Class::*MemFunPtr) (Arg1, Arg2);
    MemFunPtr  mem_fun_ptr;
    Class     *object;
    Arg1       arg1;
    Arg2       arg2;
};
```

Next, we need a function that can process these arguments:

Code sample 6

```
template <typename Class, typename Arg1, typename Arg2>
void * start_thread (void *arg_ptr) {
  MemFunData<Class,Arg1,Arg2> *mem_fun_data
        = reinterpret_cast<MemFunData<Class,Arg1,Arg2>*>(arg_ptr);
  (mem_fun_data->object)
        ->*(mem_fun_data->mem_fun_ptr) (mem_fun_data->arg1,
                                        mem_fun_data->arg2);
  return 0;
};
```

Then we can start the thread as follows:

```
MemFunData<TestClass,int,double>  mem_fun_data;
mem_fun_data.mem_fun_ptr = &TestClass::test_function;
mem_fun_data.object      = &test_object;
mem_fun_data.arg1        = 1;
mem_fun_data.arg2        = 3.1415926;

ACE_Thread_Manager::spawn (&start_thread<TestClass,int,double>,
                           (void*)&mem_fun_data);
```

Here we first create an object which is suitable to encapsulate the parameters of a binary function that is a member function of the `TestClass` class and takes an integer and a double. Then we start the thread using the correct trampoline function. It is the user's responsibility to choose the

correct trampoline function (i.e. to specify the correct template parameters) since the compiler only sees a `void*` and cannot do any type checking.

We can further simplify the process and remove the user responsibility by defining the following class and function:

Code sample 7

```
class ThreadManager : public ACE_Thread_Manager {
  public:
    template <typename Class, typename Arg1, typename Arg2>
    static void
    spawn (MemFunData<Class,Arg1,Arg2> &MemFunData) {
      ACE_Thread_Manager::spawn (&start_thread<Class,Arg1,Arg2>,
                                 (void*)&MemFunData);
    };
};
```

This way, we can call

```
ThreadManager::spawn (mem_fun_data);
```

and the compiler will figure out which the right trampoline function is, since it knows the data type of `mem_fun_data` and therefore the values of the template parameters in the `ThreadManager::spawn` function.

The way described above is basically the way which is used in `deal.II` version 3.0. Some care has to be paid to details, however. In particular, `C++` functions often pass references as arguments, which however are not assignable after initialization. Therefore, the `MemFunData` class needs to have a constructor, and arguments must be set through it. Assume, for example, `TestClass` had a second member function

```
void f (int &i, double &d);
```

Then, we would have to use `MemFunData<TestClass,int&,double&>`, which in a form without templates would look like this:

```
struct MemFunData {
    typedef void (TestClass::*MemFunPtr) (int &, double &);
    MemFunPtr  mem_fun_ptr;
    TestClass *object;
    int       &arg1;
    double    &arg2;
};
```

The compiler would require us to initialize the references to the two parameters at construction time of the `MemFunData` object, since it is not possible in `C++` to change to which object a reference points to after initialization. Adding a constructor to the `MemFunData` class would then enable us to write

```
int    i = 1;
double d = 3.1415926;
MemFunData<TestClass,int&,double&>
       mem_fun_data (&test_object, i, d, &TestClass::f);
```

Non-reference arguments could then still be changed after construction. For historical reasons, the pointer to the member function is passed as last parameter here.

The last point is that this interface is only usable for functions with two parameters. Basically, the whole process has to be reiterated for any number of parameters which we want to support. In `deal.II`, we therefore have classes `MemFunData0` through `MemFunData10`, corresponding to member

function that do not take parameters through functions that take ten parameters. Equivalently, we need the respective number of trampoline functions.

Additional thoughts need to be taken on virtual member functions and constant functions. While the first are handled by the compiler (member function pointers can also be to virtual functions, without explicitly stating so), the latter can be achieved by writing `MemFunData<const TestClass,int,double>`, which would be the correct object if we had declared `test_function` constant.

Finally we note that it is often the case that one member function starts a new thread by calling another member function of the same object. Thus, the declaration most often used is the following:

```
MemFunData<TestClass,int&,double&>
       mem_fun_data (this, 1, 3.1415926, &TestClass::f);
```

Here, instead of an arbitrary `test_object`, the present object is used, which is represented by the `this` pointer.

3.4 Second approach

While the approach outlined above works satisfactorily, it has one serious drawback: the programmer has to provide the data types of the arguments of the member function himself. While this seems to be a simple task, in practice it is often not, as will be explained in the sequel.

To expose the problem, we take an example from one of our application programs where we would like to call the function

```
template <int dim>
void DoFHandler<dim>::distribute_dofs (const FiniteElement<dim> &,
                                       const unsigned int);
```

on a new thread. Correspondingly, we would need to use

```
MemFunData2<DoFHandler<dim>, const FiniteElement<dim> &, unsigned int>
    mem_fun_data (dof_handler, fe, 0,
                  &DoFHandler<dim>::distribute_dofs);
```

to encapsulate the parameters. However, if one forgets the `const` specifier on the second template parameter, one receives the following error message (using gcc 2.95.2):

```
test.cc: In method `void InterstepData<2>::wake_up(unsigned int, Interst
epData<2>::PresentAction)':
test.cc:683:   instantiated from here
test.cc:186: no matching function for call to `ThreadManager::Mem_Fun_Da
ta2<DoFHandler<2>,FiniteElement<2> &,unsigned int>::MemFunData2 (DoFHa
ndler<2> *, const FiniteElement<2> &, int, void (DoFHandler<2>::*)(const
 FiniteElement<2> &, unsigned int))'
/home/atlas1/wolf/program/newdeal/deal.II/base/include/base/thread_manag
er.h:470: candidates are: ThreadManager::MemFunData2<DoFHandler<2>,Fin
iteElement<2> &,unsigned int>::MemFunData2(DoFHandler<2> *, FiniteElem
ent<2> &, unsigned int, void * (DoFHandler<2>::*)(FiniteElement<2> &, un
signed int))
/home/atlas1/wolf/program/newdeal/deal.II/base/include/base/thread_manag
er.h:480:                 ThreadManager::MemFunData2<DoFHandler<2>,Fin
iteElement<2> &,unsigned int>::MemFunData2(DoFHandler<2> *, FiniteElem
ent<2> &, unsigned int, void (DoFHandler<2>::*)(FiniteElement<2> &, unsi
gned int))
/home/atlas1/wolf/program/newdeal/deal.II/base/include/base/thread_manag
```

```
er.h:486:                  ThreadManager::MemFunData2<DoFHandler<2>,Fin
iteElement<2> &,unsigned int>::MemFunData2(const ThreadManager::Mem_Fu
n_Data2<DoFHandler<2>,FiniteElement<2> &,unsigned int> &)
```

While the compiler is certainly right to complain, the message is not very helpful. Furthermore, since interfaces to functions sometimes change, for example by adding additional default parameters that do not show up in usual code, programs that used to compile do no more so with messages as shown above.

Due to the lengthy and complex error messages, even very experienced programmers usually need between five and ten minutes until they get an expression like this correct. In most cases, they don't get it right in the first attempt, so the time used for the right declaration dominates the whole setup of starting a new thread. To circumvent this bottleneck at least in most cases, we chose to implement a second strategy at encapsulating the parameters of member functions. This is done in several steps: first let the compiler find out about the right template parameters, then encapsulate the parameters, use the objects, and finally solve some technical problems with virtual constructors and locking of destruction. We will treat these steps sequentially in the following.

3.4.1 Finding the correct template parameters.

`C++` offers the possibility of templatized functions that deduce their template arguments themselves. In fact, we have used them in the `ThreadManager::spawn` function in Code Sample 7 already. Here, this can be used as follows: assume we have a function encapsulation class

```
template <typename Class, typename Arg1, typename Arg2>
class MemFunData { ... };
```

as above, and a function

```
template <typename Class, typename Arg1, typename Arg2>
MemFunData<Class,Arg1,Arg2>
encapsulate (void (Class::*mem_fun_ptr)(Arg1, Arg2)) {
  return MemFunData<Class,Arg1,Arg2> (mem_fun_ptr);
};
```

Then, if we call this function with the test class of Code Sample 2 like this:

```
encapsulate (&TestClass::test_function);
```

it can unambiguously determine the template parameters to be `Class=TestClass`, `Arg1=int`, `Arg2=double`.

3.4.2 Encapsulating the parameters.

We should not try to include the argument values for the new thread right away, for example by declaring `encapsulate` like this:

```
template <typename Class, typename Arg1, typename Arg2>
MemFunData<Class,Arg1,Arg2>
encapsulate (void (Class::*mem_fun_ptr)(Arg1, Arg2),
             Arg1  arg1,
             Arg2  arg2,
             Class object) {
  return MemFunData<Class,Arg1,Arg2> (mem_fun_ptr, object, arg1, arg2);
};
```

The reason is that for template functions, no parameter promotion is performed. Thus, if we called this function as in

```
encapsulate (&TestClass::test_function,
             1, 3,
             test_object);
```

then the compiler would refuse this since from the function pointer it must deduce that `Arg2 = double`, but from the parameter "3" it must assume that `Arg2 = int`. The resulting error message would be similarly lengthy as the one shown above.

One could instead write `MemFunData` like this:

```
template <typename Class, typename Arg1, typename Arg2>
class MemFunData {
  public:
    typedef void (Class::*MemFunPtr)(Arg1, Arg2);

    MemFunData (MemFunPtr mem_fun_ptr_) {
      mem_fun_ptr = mem_fun_ptr_;
    };

    void collect_args (Class *object_,
                       Arg1   arg1_,
                       Arg2   arg2_) {
      object = object_;
      arg1   = arg1_;
      arg2   = arg2_;
    };

    MemFunPtr  mem_fun_ptr;
    Class     *object;
    Arg1       arg1;
    Arg2       arg2;
};
```

One would then create an object of this type including the parameters to be passed as follows:

```
encapsulate(&TestClass::test_function).collect_args(test_object, 1, 3);
```

Here, the first function call creates an object with the right template parameters and storing the member function pointer, and the second one, calling a member function, fills in the function arguments.

Unfortunately, this way does not work: if one or more of the parameter types is a reference, then the respective reference variable needs to be initialized by the constructor, not by `collect_args`. It needs to be known which object the reference references at construction time, since later on only the referenced object can be assigned, not the reference itself anymore.

Since we feel that we are close to a solution, we introduce one more indirection, which indeed will be the last one:

Code sample 8

```
template <typename Class, typename Arg1, typename Arg2>
class MemFunData {
  public:
    typedef void (Class::*MemFunPtr)(Arg1, Arg2);

    MemFunData (MemFunPtr mem_fun_ptr_,
                Class *object_,
                Arg1   arg1_,
                Arg2   arg2_) :
```

```
          mem_fun_ptr (mem_fun_ptr_),
          object      (object_),
          arg1        (arg1_),
          arg2        (arg2_)            {};

    MemFunPtr  mem_fun_ptr;
    Class     *object;
    Arg1       arg1;
    Arg2       arg2;
};


template <typename Class, typename Arg1, typename Arg2>
struct ArgCollector {
    typedef void (Class::*MemFunPtr)(Arg1, Arg2);

    ArgCollector (MemFunPtr mem_fun_ptr_) {
      mem_fun_ptr = mem_fun_ptr_;
    };


    MemFunData<Class,Arg1,Arg2>
    collect_args (Class *object_,
                  Arg1   arg1_,
                  Arg2   arg2_) {
      return MemFunData<Class,Arg1,Arg2> (mem_fun_ptr, object,
                                          arg1, arg2);
    };

    MemFunPtr  mem_fun_ptr;
};


template <typename Class, typename Arg1, typename Arg2>
ArgCollector<Class,Arg1,Arg2>
encapsulate (void (Class::*mem_fun_ptr)(Arg1, Arg2)) {
  return ArgCollector<Class,Arg1,Arg2> (mem_fun_ptr);
};
```

Now we can indeed write for the test class of Code Sample 2:

```
encapsulate(&TestClass::test_function).collect_args(test_object, 1, 3);
```

The first call creates an object of type `ArgCollector<...>` with the right parameters and storing the member function pointer, while the second call, a call to a member function of that intermediate class, generates the final object we are interested in, including the member function pointer and all necessary parameters. Since `collect_args` already has its template parameters fixed from `encapsulate`, it can convert between data types.

3.4.3 Using these objects.

Now we have an object of the correct type automatically generated, without the need to type in any template parameters by hand. What can we do with that? First, we can't assign it to a variable of that type, e.g. for use in several `spawn` commands:

```
MemFunData mem_fun_data = encapsulate(...).collect_args(...);
```

Why? Since we would then have to write the data type of that variable by hand: the correct data type is not `MemFunData` as written above, but `MemFunData<TestClass,int,double>`. Specifying all these template arguments was exactly what we wanted to avoid. However, we can do some such thing if the variable to which we assign the result is of a type which is a base class of `MemFunData<...>`. Unfortunately, the data values that `MemFunData<...>` encapsulates depend on the template parameters, so the respective variables in which we store the values can only be placed in the derived class and could not be copied when we assign the variable to a base class object, since that does not have these variables.

What can we do here? Assume we have the following class structure:

Code sample 9

```
class FunDataBase {};

template <...> class MemFunData : public FunDataBase
{  /* as above */ };

class FunEncapsulation {
  public:
    FunEncapsulation (FunDataBase *f)
                : fun_data_base (f) {};
    FunDataBase *fun_data_base;
};


template <typename Class, typename Arg1, typename Arg2>
FunEncapsulation
ArgCollector<Class,Arg1,Arg2>::collect_args (Class *object_,
                                             Arg1   arg1_,
                                             Arg2   arg2_) {
  return new MemFunData<Class,Arg1,Arg2> (mem_fun_ptr, object,
                                          arg1, arg2);
};
```

Note that in the return statement of the `collect_args` function, first a cast from `MemFunData*` to `FunDataBase*`, and then a constructor call to `FunEncapsulation :: FunEncapsulation (FunDataBase*)` was performed.

In the example above, the call to `encapsulate(...).collect_args(...)` generates an object of type `FunEncapsulation`, which in turn stores a pointer to an object of type `FunDataBase`, here to `MemFunData<...>` with the correct template parameters. We can assign the result to a variable the type of which does not contain any template parameters any more, as desired:

```
FunEncapsulation
    fun_encapsulation = encapsulate (&TestClass::test_function)
                                       .collect_args(test_object, 1, 3);
```

But how can we start a thread with this object if we have lost the full information about the data types? This can be done as follows: add a variable to `FunDataBase` which contains the address of a function that knows what to do. This function is usually implemented in the derived classes, and its address is passed to the constructor:

Code sample 10

```
class FunDataBase {
  public:
    typedef void * (*ThreadEntryPoint) (void *);

    FunDataBase (ThreadEntryPoint t) :
```

```
            thread_entry_point (t) {};

    ThreadEntryPoint thread_entry_point;
};

template <...>
class MemFunData : public FunDataBase {
  public:
            // among other things, the constructor now does this:
    MemFunData () :
            FunDataBase (&start_thread) {};

    static void * start_thread (void *args) {
      // do the same as in Code Sample 4 above
    }
};


void spawn (ACE_Thread_Manager &thread_manager,
            FunEncapsulation   &fun_encapsulation) {
  thread_manager.spawn (*fun_encapsulation.fun_data_base
                                  ->thread_entry_point,
                        &fun_data_base);
};
```

`fun_encapsulation.fun_data_base->thread_entry_point` is given by the derived class as that function that knows how to handle objects of the type which we are presently using. Thus, we can now write the whole sequence of function calls (assuming we have an object `thread_manager` of type `ACE_Thread_Manager`):

```
FunEncapsulation
    fun_encapsulation = encapsulate (&TestClass::test_function)
                                      .collect_args(test_object, 1, 3);
spawn (thread_manager, fun_encapsulation);
```

This solves our problem in that no template parameters need to be specified by hand any more. The only source for lengthy compiler error messages is if the parameters to `collect_args` are in the wrong order or can not be casted to the parameters of the member function which we want to call. These problems, however, are much more unlikely in our experience, and are also much quicker sorted out.

3.4.4 Virtual constructors.

While the basic techniques have been fully developed now, there are some aspects which we still have to take care of. The basic problem here is that the `FunEncapsulation` objects store a pointer to an object that was created using the `new` operator. To prevent a memory leak, we need to destroy this object at some time, preferably in the destructor of `FunEncapsulation`:

```
FunEncapsulation::~FunEncapsulation () {
  delete fun_data_base;
};
```

However, what happens if we have copied the object before? In particular, this is always the case using the functions above: `collect_args` generates a temporary object of type `FunEncapsulation`, but there could be other sources of copies as well. If we do not take special precautions, only the pointer to the object is copied around, and we end up with stale pointers pointing to invalid

locations in memory once the first object has been destroyed. What we obviously need to do when copying objects of type `FunEncapsulation` is to not copy the pointer but to copy the object which it points to. Unfortunately, the following copy constructor is not possible:

```
FunEncapsulation::FunEncapsulation (const FunEncapsulation &m) {
  fun_data_base = new FunDataBase (*m.fun_data_base);
};
```

The reason, of course, is that we do not want to copy that part of the object belonging to the abstract base class. But we can emulate something like this in the following way (this programming idiom is called "virtual constructors"):

Code sample 11

```
class FunDataBase {
  public:
    // as above

    virtual FunDataBase * clone () const = 0;
};

template <...>
class MemFunData : public FunDataBase {
  public:
    // as above

                      // copy constructor:
    MemFunData (const MemFunData<...> &mem_fun_data) {...};

                      // clone the present object, i.e.
                      // create an exact copy:
    virtual FunDataBase * clone () const {
      return new MemFunData<...>(*this);
    };
};


FunEncapsulation::FunEncapsulation (const FunEncapsulation &m) {
  fun_data_base = m.fun_data_base->clone ();
};
```

Thus, whenever the `FunEncapsulation` object is copied, it creates a copy of the object it harbors (the `MemFunData<...>` object), and therefore always owns its copy. When the destructor is called, it is free to delete its copy without affecting other objects (from which it may have been copied, or to which it was copied). Similar to the copy constructor, we have to modify the copy operator, as well.

3.4.5 Spawning independent threads.

Often, one wants to spawn a thread which will have its own existence until it finishes, but is in no way linked to the creating thread any more. An example would be the following, assuming a function `TestClass::compress_file(const string file_name)` exists and that there is an object `thread_manager` not local to this function:

```
...
string file_name;
...    // write some output to a file
```

```
// now create a thread which runs 'gzip' on that output file to reduce
// disk space requirements. don't care about that thread any more
// after creation, i.e. don't wait for its return
FunEncapsulation
    fun_encapsulation = encapsulate (&TestClass::compress_file)
                              .collect_args(test_object, file_name);
spawn (thread_manager, fun_encapsulation);

// quit the present function
return;
```

The problem here is that the object `fun_encapsulation` goes out of scope when we quit the present function, and therefore also deletes its pointer to the data which we need to start the new thread. If in this case the operating system was a bit lazy in creating the new thread, the function `start_thread` would at best find a pointer pointing to an object which is already deleted. Further, but this is obvious, if the function is taking references or pointers to other objects, it is to be made sure that these objects persist at least as long as the spawned thread runs.

What one would need to do here at least, is wait until the thread is started for sure, before deletion of the `FunEncapsulation` is allowed. To this end, we need to use a "Mutex", to allow for exclusive operations. A Mutex (short for *mut*ually *ex*clusive) is an object managed by the operating system and which can only be "owned" by one thread at a time. You can try to "acquire" a Mutex, and you can later "release" it. If you try to acquire it, but the Mutex is owned by another thread, then your thread is blocked until the present owner releases it. Mutices (plural of "Mutex") are therefore most often used to guarantee that only one thread is presently accessing some object: a thread that wants to access that object acquires a Mutex related to that object and only releases it once the access if finished; if in the meantime another thread wants to access that object as well, it has to acquire the Mutex, but since the Mutex is presently owned already, the second thread is blocked until the first one has finished its access.

Alternatively, one can use Mutices to synchronize things. We will use it for the following purpose: the Mutex is acquired by the starting thread; when later the destructor of the `FunEncapsulation` class (running on the same thread) is called, it tries to acquire the lock again; it will thus only continue its operations once the Mutex has been released by someone, which we do on the spawned thread once we don't need the data of the `FunEncapsulation` object any more and destruction is safe.

All this can then be done in the following way:

Code sample 12

```
class FunEncapsulation {
  public:
    ...       // as before
    ~FunEncapsulation ();
};


class FunDataBase {
  public:
    ...       // as before
    Mutex       lock;
};

template <typename Class, typename Arg1, typename Arg2>
void * start_thread (void *arg_ptr) {
  MemFunData<Class,Arg1,Arg2> *mem_fun_data
        = reinterpret_cast<MemFunData *>(arg_ptr);
```

```
  // copy the data arguments:
  MemFunData<Class,Arg1,Arg2>::MemFunPtr
          mem_fun_ptr = mem_fun_data->mem_fun_ptr;
  Class * object      = mem_fun_data->object;
  Arg1    arg1        = mem_fun_data->arg1;
  Arg2    arg2        = mem_fun_data->arg2;

  // data is now copied, so the original object may be deleted:
  mem_fun_data->lock.release ();

  // now call the thread function:
  object->*mem_fun_ptr (arg1, arg2);

  return 0;
};


FunEncapsulation::~FunEncapsulation () {
  // wait until the data is copied by the new thread and
  // 'release' is called by 'start_thread':
  fun_data_base->lock.acquire ();
  // now delete the object which is no more needed
  delete fun_data_base;
};


void spawn (ACE_Thread_Manager  &thread_manager,
            FunEncapsulation &fun_encapsulation) {
  // lock the fun_encapsulation object
  fun_encapsulation.fun_data_base->lock.acquire ();
  thread_manager.spawn (*fun_encapsulation.fun_data_base
                                  ->thread_entry_point,
                        &fun_data_base);
};
```

When we call `spawn`, we set a lock on the destruction of the `FunEncapsulation` object just before we start the new thread. This lock is only released when inside the new thread (i.e. inside the `start_thread` function) all arguments have been copied to a safe place. Now we have local copies and don't need the ones from the `fun_encapsulation` object any more, which we indicate by releasing the lock. Inside the destructor of that object, we wait until we can obtain the lock, which is only after it has been released by the newly started thread; after having waited till this moment, the destruction can go on safely, and we can exit the function from which the thread was started, if we like so.

The scheme just described also works if we start multiple threads using only one object of type `FunEncapsulation`:

```
FunEncapsulation
    fun_encapsulation = encapsulate (&TestClass::test_function)
                              .collect_args(test_object, arg_value);
spawn (thread_manager, fun_encapsulation);
spawn (thread_manager, fun_encapsulation);

// quit the present function
return;
```

Here, when starting the second thread the spawn function has to wait until the newly started first thread has released its lock on the object; however, this delay is small and should not pose a noticeable problem. Thus, no special treatment of this case is necessary, and we can in a simple way emulate the `spawn_n` function provided by most operating systems, which spawns several new threads at once:

```
void spawn_n (ACE_Thread_Manager &thread_manager,
              FunEncapsulation   &fun_encapsulation,
              const unsigned int  n_threads) {
  for (unsigned int i=0; i<n_threads; ++i)
    spawn (thread_manager, fun_encapsulation);
};
```

A direct support of the `spawn_n` function of the operating system would be difficult, though, since each of the new threads would call `lock.release()`, even though the lock was only acquired once.

Since we have now made sure that objects are not deleted too early, even the following sequence is possible, which does not involve any named variables at all, only a temporary one, which immediately released after the call to `spawn`:

Code sample 13

```
spawn (thread_manager,
       encapsulate (&TestClass::test_function)
          .collect_args(test_object, arg_value));
```

We most often use this very short idiom in the applications in Section 4 and in our own programs.

3.4.6 Number of parameters. Non-member functions.

Above, we have explained how we can define classes for a binary member function. This approach is easily extended to member functions taking any number of parameters. We simply have to write classes `MemFunData0`, `MemFunData1`, and so on, which encapsulate member functions that take zero, one, etc parameters. Likewise, we have to have classes `ArgCollectorN` for each number of parameters, and functions `encapsulate` that return an object of type `ArgCollectorN`. Since functions can be overloaded on their argument types, we need not call the `encapsulate` functions differently.

All of which has been said above can also easily be adopted to global functions or static member functions. Instead of the classes `MemFunDataN` we can then use classes `FunDataN` that are also derived from `FunDataBase`. The respective `ArgCollector` classes then collect only the arguments, not the object on which we will operate. The class, `FunEncapsulation` is not affected by this, nor is `FunDataBase`.

4 Applications

In the next few subsections, we will show usual applications of multi-threading in the `deal.II` library. The programs already use the new scheme discussed in Section 3.4.

4.1 Writing output detached to disk

The output classes, i.e. basically the classes `DataOut` and `DataOutStack` and their base classes, follow a strictly hierarchical model of data flow. The two terminal classes know about such things as triangulations, degrees of freedom, or finite elements, but they translate this structured information into a rather simple intermediate format. This conversion is done in the `build_patches` functions of these classes. The actual output routines only convert this intermediate format into one of the supported graphics formats, which is then a relatively simple task.

This separation of processing of structured data and actual output of the intermediate format was chosen since the actual output routines became rather complex with growing scope of the whole library. For example, we had to update all output functions when vector-valued finite elements were supported, and we had to do so again when discontinuous elements were developed. This became an unmanageable burden with the growing number of output formats, and we decided that an intermediate format would be more appropriate, which is created by only one function, but can be written to output formats by a number of different functions.

In the present context, this has the following implications: once the intermediate data is created by the `build_patches` function, we need no more preserve the data from which it was made (i.e. the grid which it was computed on, or the vector holding the actual solution values) and we can go on with computing on the next finer grid, or the next time step, while the intermediate data is converted to a graphics format file detached from the main process. The only thing which we must make sure is that the program only terminates after all detached output threads are finished. This can be done in the following way:

```
// somewhere define a thread manager that keeps track of all
// detached ('global') threads
ACE_Thread_Manager global_thread_manager;

// This is the class which does the computations:
class MainClass {
    ...

    // now two functions, the first is called from the main program
    // for output, the second will manage detached output
    void write_solution ();
    void write_detached (DataOut<dim> *data_out);
};


void MainClass::write_solution () {
  DataOut<dim> *data_out = new DataOut<dim>();

  // attach DoFHandler, add data vectors, ...

  data_out->build_patches ();

  // now everything is in place, and we can write the data detached
  // Note that we transfer ownership of 'data_out' to the other thread
  Threads::spawn (global_thread_manager,
                  Threads::encapsulate(&MainClass<dim>::write_detached)
                      .collect_args(this, data_out));
};


void MainClass::write_detached (DataOut<dim> *data_out) {
  ofstream output_file ("abc");
  data_out->write_gnuplot (output_file);

  // now delete the object which we got from the starting thread
  delete data_out;
};
```

```
int main () {
  ...  // do all the work

  // now wait for all detached threads to finish
  global_thread_manager.wait ();
};
```

Note that the functions `spawn` and `encapsulate` are prefixed by `Threads::` since in the actual implementation in `deal.II` they are declared within a namespace of that name.

It should be noted that if you want to write output detached from the main thread, and from the main thread at the same time, you need a version of the C++ standard library delivered with your compiler that supports parallel output. For the GCC compiler, this can be obtained by configuring it with the flag `--enable-threads` at build time.

4.2 Assembling the matrix

Setting up the system matrix is usually done by looping over all cells and computing the contributions of each cell separately. While the computations of the local contributions is strictly independent, we need to transfer these contributions to the global matrix afterward. This transfer has to be synchronized, in order to avoid that one thread overwrites values that another thread has just written.

In most cases, building the system matrix in parallel will look like the following template:

```
void MainClass::build_matrix () {
  // define how many threads will be used (here: 4)
  const unsigned int n_threads = 4;
  const unsigned int n_cells_per_thread
        = triangulation.n_active_cells () / n_threads;

  // define the Mutex that will be used to synchronise
  // accesses to the matrix
  ACE_Thread_Mutex mutex;

  // define thread manager
  ACE_Thread_Manager thread_manager;

  vector<DoFHandler<dim>::active_cell_iterator>
        first_cells (n_threads),
        end_cells (n_threads);

  DoFHandler<dim>::active_cell_iterator
        present_cell = dof_handler.begin_active ();
  for (unsigned int thread=0; thread<n_threads; ++thread)
    {
      // for each thread: first determine the range of cells on
      // which it shall operate:
      first_cells[thread] = present_cell;

      end_cells[thread] = first_cells[thread];
      if (thread != n_threads-1)
        for (unsigned int i=0; i<n_cells_per_thread; ++i)
          ++end_cells[thread];
      else
        end_cells[thread] = dof_handler.end();
```

```
      // now start a new thread that builds the contributions of
      // the cells in the given range
      Threads::spawn (thread_manager,
               Threads::encapsulate(&MainClass::build_matrix_threaded)
                     .collect_args (this,
                                    first_cells[thread],
                                    end_cells[thread],
                                    mutex));

      // set start iterator for next thread
      present_cell = end_cells[thread];
    };

  // wait for the threads to finish
  thread_manager.wait ();
};


void MainClass::build_matrix_threaded
     (const DoFHandler<dim>::active_cell_iterator &first_cell,
      const DoFHandler<dim>::active_cell_iterator &end_cell,
      ACE_Thread_Mutex                            &mutex)
{
  FullMatrix<double>   cell_matrix;
  vector<unsigned int> local_dof_indices;

  DoFHandler<dim>::active_cell_iterator cell;
  for (cell=first_cell; cell!=end_cell; ++cell)
    {
      // compute the elements of the cell matrix
      ...

      // get the indices of the DoFs of this cell
      cell->get_dof_indices (local_dof_indices);

      // now transfer local matrix into the global one.
      // synchronise this with the other threads
      mutex.acquire ();
      for (unsigned int i=0; i<dofs_per_cell; ++i)
        for (unsigned int j=0; j<dofs_per_cell; ++j)
          global_matrix.add (local_dof_indices[i],
                             local_dof_indices[j],
                             cell_matrix(i,j));
      mutex.release ();
    };
};
```

Note that since the `build_matrix_threaded` function takes its arguments as references, we have to make sure that the variables to which these references point live at least as long as the spawned threads. It is thus not possible to use the same variables for start and end iterator for all threads, as the following example would do:

```
  ....
  DoFHandler<dim>::active_cell_iterator
         first_cell = dof_handler.begin_active ();
```

```
  for (unsigned int thread=0; thread<n_threads; ++thread)
    {
      // for each thread: first determine the range of threads on
      // which it shall operate:
      DoFHandler<dim>::active_cell_iterator end_cell = first_cell;
      if (thread != n_threads-1)
        for (unsigned int i=0; i<n_cells_per_thread; ++i)
          ++end_cell;
      else
         end_cell = dof_handler.end();

      // now start a new thread that builds the contributions of
      // the cells in the given range
      Threads::spawn (thread_manager,
             Threads::encapsulate(&MainClass::build_matrix_threaded)
                   .collect_args (this, first_cell, end_cell, mutex));

      // set start iterator for next thread
      first_cell = end_cell;
    };
  ....
```

Since splitting a range of iterators (for example the range `begin_active()` to `end()`) is a very common task when setting up threads, there is a function

```
template <typename ForwardIterator>
vector<pair<ForwardIterator,ForwardIterator> >
split_range (const ForwardIterator &begin, const ForwardIterator &end,
             const unsigned int n_intervals);
```

in the `Threads` namespace that splits the range [`begin`,`end`) into `n_intervals` subintervals of approximately the same size.

Using this function, the thread creation function can now be written as follows:

```
void MainClass::build_matrix () {
  const unsigned int n_threads = 4;
  ACE_Thread_Mutex   mutex;
  ACE_Thread_Manager thread_manager;

  // define starting and end point for each thread
  typedef DoFHandler<dim>::active_cell_iterator active_cell_iterator;
  vector<pair<active_cell_iterator,active_cell_iterator> >
       thread_ranges
       = split_range<active_cell_iterator> (dof_handler.begin_active (),
                                            dof_handler.end (),
                                            n_threads);

  for (unsigned int thread=0; thread<n_threads; ++thread)
    spawn (thread_manager,
           encapsulate(&MainClass::build_matrix_threaded)
                .collect_args (this,
                               thread_ranges[thread].first,
                               thread_ranges[thread].second,
                               mutex));
```

```
  thread_manager.wait ();
};
```

We have here omitted the `Threads::` prefix to make things more readable. Note that we had to explicitly specify the iterator type `active_cell_iterator` to the `split_range` function, since the two iterators given have different type (`dof_handler.end()` has type `DoFHandler<dim> :: raw_cell_iterator`, which can be converted to `DoFHandler<dim>::active_cell_iterator`) and C++ requires that either the type is explicitly given or the type be unique.

A word of caution is in place here: since usually in finite element computations, the system matrix is ill-conditioned, small changes in a data vector or the matrix can lead to significant changes in the output. Unfortunately, since the order in which contributions to elements of the matrix or vector are computed can not be predicted when using multiple threads, round-off can come into play here. For example, taken from a real-world program, the following contributions for an element of a right hand side vector are computed from four cells: -3.255208333333328815, -3.255208333333333694, -3.255208333333333694, and -3.255208333333331526; however, due to round-off the sum of these numbers depends on the order in which they are summed up, such that the resulting element of the vector differed depending on the number of threads used, the number of other programs on the computer, and other random sources. In subsequent runs of exactly the same programs, the sum was either -13.02083333333332827 or -13.02083333333332610. Although the difference is still only in the range of round-off error, it caused a change in the fourth digit of a derived, very ill-conditioned quantity after the matrix was inverted several times (this accuracy in this quantity was not really needed, but it showed up in the output and also led to different grid refinement due to comparison with other values of almost the same size). Tracking down the source of such problems is extremely difficult and frustrating, since they occur non-deterministically in subsequent runs of the same program, and it can take several days until the actual cause is found.

One possible work-around is to reduce the accuracy of the summands such that the value of the sum becomes irrespective of the order of the summands. One, rather crude method is to use a conversion to data type `float` and back; the update loop from above would then look as follows:

```
for (unsigned int i=0; i<dofs_per_cell; ++i)
  for (unsigned int j=0; j<dofs_per_cell; ++j)
    global_matrix.add (local_dof_indices[i],
                       local_dof_indices[j],
                       static_cast<float>(cell_matrix(i,j)));
```

Note that the cast back to `double` is performed here implicitly. The question whether a reduction in accuracy in the order shown here is tolerable, is problem dependent. There are methods that lose less accuracy than shown above.

The other, less computationally costly possibility would be to decrease the accuracy of the resulting sum, in the hope that all accumulated round-off error is deleted. However, this is unsafe since the order dependence remains and may even be amplified if the values of the sum lie around a boundary where values are rounded up or down when reducing the accuracy. Furthermore, problems arise if the summands have different signs and the result of summation consists of round-off error only.

4.3 Parallel Jacobi preconditioning

When preconditioning a matrix, for example in a Conjugate Gradients solver, one may choose the Jacobi scheme for preconditioning. The preconditioned vector $\tilde{v}$ is computed from the vector v using the following relationship:

$$\tilde{v}_i = \frac{1}{a_{ii}} v_i,$$

where a_{ii} are the diagonal elements of the matrix which we are presently inverting. As is obvious, the result of preconditioning one element of v is entirely independent of all other elements, so this operation is trivially parallelizable. In practice, this is done by splitting the interval $[0, n)$ into

equal parts $[n_i, n_{i+1}), i = 0, \ldots, p-1$, where n is the size of the matrix, and p is the number of processors. Obviously, $n_0 = 0, n_p = n$, and $n_i < n_{i+1}$.

Just like for splitting a range of iterators using the function `split_range` used above, there is a function

```
vector<pair<unsigned int, unsigned int> >
split_interval (const unsigned int &begin, const unsigned int &end,
                const unsigned int n_intervals);
```

that splits the interval `[begin,end)` into `n_intervals` equal parts. This function will be used to assign each processor its share of elements v_i.

Furthermore, we will use some functionality provided by the `MultithreadInfo` class in `deal.II`. Upon start-up of the library, the static variable `multithread_info.n_cpus` is set to the number of processors in the computer the program is presently running on. `multithread_info` is a global variable of type `MultithreadInfo` available in all parts of the library. Furthermore, there is a variable `multithread_info.n_default_threads`, which by default is set to `n_cpus`, but which can be changed by the user; it denotes the default number of threads which the library shall use whenever multi-threading is implemented for some operation. We will use this variable to decide how many threads shall be used to precondition the vector.

The implementation of the preconditioning function then looks like this:

```
                    // define an abbreviatory data type for an interval
typedef pair<unsigned int, unsigned int> Interval;

void Preconditioner::precondition_jacobi (const Matrix &m,
                                          const Vector &v,
                                          Vector       &v_tilde) {
                        // define an abbreviation to the number
                        // of threads which we will use
  const unsigned int n_threads = multithread_info.n_default_threads;
                        // first split the interval into equal pieces
  vector<Interval> intervals = Threads::split_interval (0, m.rows(),
                                                        n_threads);

                        // then define a thread manager
  ACE_Thread_Manager thread_manager;
                        // and finally start all the threads:
  for (unsigned int i=0; i<n_threads; ++i)
    Threads::spawn (thread_manager,
                    Threads::encapsulate (&Preconditioner::threaded_jacobi)
                           .collect_args (this, m, v, v_tilde, intervals[i]));

                        // wait for all the threads to finish
  thread_manager.wait ();
};


void Preconditioner::threaded_jacobi (const Matrix   &m,
                                      const Vector   &v,
                                      Vector         &v_tilde,
                                      const Interval &interval) {
                       // apply the preconditioner in the given interval
  for (unsigned int i=interval.first; i<interval.second; ++i)
    v_tilde(i) = v(i) / m(i,i);
};
```

It is noted, however, that more practical preconditioners are usually not easily parallelized. However, matrix-vector and vector-vector operations can often be reduced to independent parts and can then be implemented using multiple threads.

5 Conclusions

We have shown how multi-threading is supported in `deal.II` and how it can be used in several examples occuring in common finite element programs. It was demonstrated that implementing a usable C++ interface poses several difficulties, both from the aspect of user friendliness as well as program correctness. In order to overcome these difficulties, first the more simple framework implemented in `deal.II` version 3.0 was discussed, followed by a rather complex scheme which will be the base of implementations in future versions.

The second framework features a more complicated hierarchy of classes as well as intricate use of templates and synchronization mechanisms, which however led to a design in which threads can be created in a user friendly, system independent, C++ like way suitable for common programs. The use of this framework is inherently safe and does not require special knowledge of the internals by the user, and is simple to use. By using it, the overhead required for programming multi-threaded applications is reduced to a minimum and the programmer can concentrate on the task of getting the semantics of multi-threaded programs right, in particular managing concurrent access to data and distributing work to different threads.

The framework has been used in several application programs and has shown that with only marginally increased programming effort, finite element programs can be made significantly faster on multi-processor machines.

Acknowledgments. The author would like to thank Thomas Richter for his work in parallelizing several parts of the `deal.II` library, and Ralf Hartmann for help in the preparation of this report.

References

[1] Wolfgang Bangerth and Guido Kanschat. Concepts for object-oriented finite element software – the `deal.II` library. Preprint 99-43, SFB 359, Universität Heidelberg, October 1999.

[2] Wolfgang Bangerth and Guido Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference.* IWR Heidelberg, October 1999. `http://gaia.iwr.uni-heidelberg.de/~deal/`.

[3] H. Custer. *Inside Windows NT.* Microsoft Press, Redmont, Washington, 1993.

[4] Douglas C. Schmidt et al. WWW homepage of the Adaptive Communications Environment ACE, `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[5] J. Eykholt, S. Kleinman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyon multiprocessing... Multithreading the SunOS kernel. In *Proceedings of the Summer USENIX C++ Technical Conference, San Antonio, Texas*, June 1992.

[6] IEEE. Threads extensions for portable operating systems. Technical report, IEEE, 1995.

[7] Douglas C. Schmidt. ACE: an object-oriented framework for developing distributed applications. In *Proceedings of the Sixth USENIX C++ Technical Conference, Cambridge, Massachusetts.* USENIX Association, April 1994.

[8] Douglas C. Schmidt and Nanbor Wang. An OO encapsulation of lightweight OS concurrency mechanisms in the ACE toolkit. Technical Report WUCS-95-31, Washington University, St. Louis, 1995.