MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 41.25:

Parallelization on a cluster of distributed memory machines

Part 2: Debugging with MPI

Debugging with MPI

General observations:

- Debugging single-threaded programs is difficult enough
- Debugging MPI programs sucks (truth!)

- It is essential to know common error sources
- It is essential not to get confused

- There are no free parallel debuggers...
- ...or other free tools that could make your life simpler
- There is *TotalView* (but it is commercial)

Debugging with MPI

Common problems:

- Deadlocks of various kinds
- Erroneously believing that one code block takes very long

- More information on debugging such things: see the deal.II FAQs

Deadlocks

Definition:

- Informally: “The program hangs”
- Formally: “A situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does”

Deadlocks

Example 1: We think of deadlocks as situations like this:

```
void f() {
    do_work (items[0]); // do some work we know has to be done

    // if our time has expired, let someone else do the other item
    if (run time > expected run time)
        MPI_Send (items[1], ...);
    // otherwise complete our work and see if anyone else has more work
    else  {
        do_work (items[1]);
        while (run time < max run time) {
            Item next_item;
            MPI_Recv (&next_item, …);
            do_work (next_item);
        }
}
```

Deadlocks

Example 1: We think of deadlocks as situations like this:
[…]

Analysis:

- All processors may end up waiting for incoming messages in the same place
- Thus, nobody moves – the program hangs!

Approach to debugging:

- Find out *where* each MPI process is
- Understand *why*

Deadlocks

Example 2: Deadlocks more often look like this:

```
void f() {
    int ii = foo();       // compute something locally
    if (need_to_sum(my_rank)) {
        int sum;
        MPI_Reduce (&ii, &sum, 1, MPI_INT, MPI_SUM, ...);
        …;
    }

    int kk = bar();    // compute something else here
    MPI_Reduce (&kk, …);
}
```

Now imagine there is a bug in *need_to_sum()*: it returns *true* only for some processes.

How long does this operation take?

Example 3: Imagine this situation:

```
Timer t;
t.start();
my_function();
t.stop();
if (my_rank == 0)
  std::cout << "Calling my_function() took " << timer() << " seconds.\n";
```

- This is supposed to measure how long *my_function* takes on processor 0
- But in parallel computing, how long a function takes depends on other ranks as well!

How long does this operation take?

Example 3:

```
void my_function () {
   double val = compute_something_locally();
   double global_sum = 0;
   MPI_Reduce (&val, &global_sum, MPI_DOUBLE, 1, …);
   if (my MPI rank == 0)
     std::cout << "Global sum = " << global_sum << std::endl;
 }
```

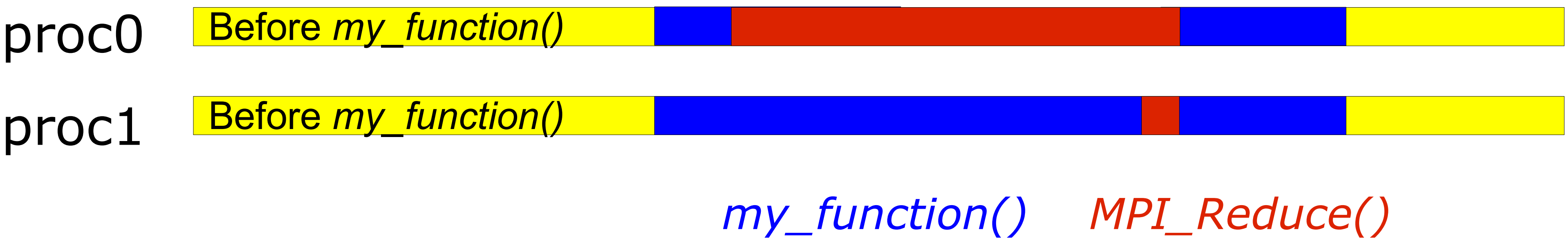
Situation 1:

- *compute_something_locally()* is quick on proc 0, but takes long on proc 1
- But proc 0 will have to wait all of this time in *MPI_Reduce*

Result: Erroneous conclusion that *my_function()* takes long on *processor 0*!

How long does this operation take?

Example 3: Graphical representation:



Situation 1:

- *compute_something_locally()* is quick on proc 0, but takes long on proc 1
- But proc 0 will have to wait all of this time in *MPI_Reduce*

Result: Erroneous conclusion that *my_function()* takes long on *processor 0*!

How long does this operation take?

Example 3:

```
Timer t;
t.start();
my_function();
t.stop();
if (my_rank == 0)
  std::cout << "Calling my_function() took " << timer() << " seconds.\n";
```

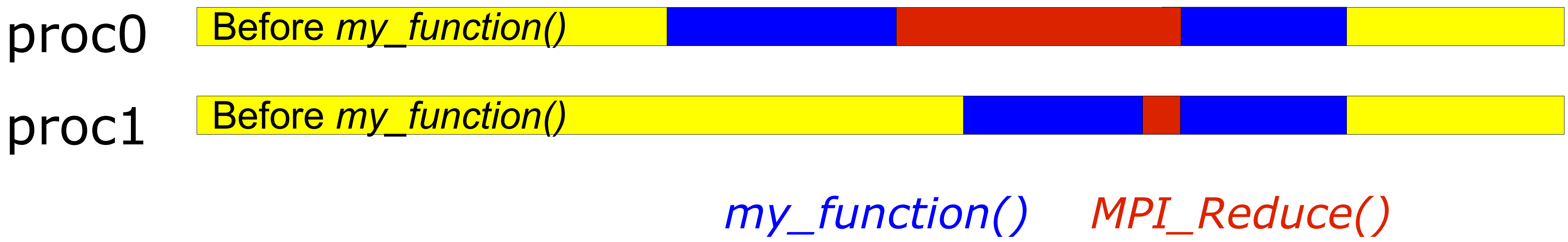
Situation 2:

- The operation *before my_function()* takes long on proc. 1
- But proc. 0 will have to wait *MPI_Reduce* in *my_function*

Result: Erroneous conclusion that *c_s_l()* takes long anywhere!

How long does this operation take?

Example 3: Graphical representation:



Situation 2:

- The operation *before my_function()* takes long on proc. 1
- But proc. 0 will have to wait *MPI_Reduce* in *my_function*

Result: Erroneous conclusion that *c_s_l()* takes long anywhere!

Debugging with MPI

Summary:

- Parallel computations present many riddles during debugging
- One can spend much time looking in the wrong place
- It is important to be familiar with patterns of common mistakes
- Learn how to use debuggers for parallel computations:
  – via *mpirun -np 4 xterm -e gdb ./myprog*
  – by attaching a debugger to a running program

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University