Parallel Design Patterns: Exercise 4

Lawrence Mitchell

February 2, 2012

Using a task farm to distribute work

Often it is not possible to arrive at a static decomposition of a problem that balances work well between processes. Computing the mandelbrot set in parallel is an example of this. Points that are in the set take much longer to compute than points that are out of the set, so in our line-wise decomposition, the "middle" lines take more time to compute.

One method to deal with this is to divide the total work into much smaller parts, and use a task farm approach. A single master process keeps track of the tasks and doles them out to workers. When a worker completes its task, it returns the result to the master and asks for a new task. This process repeats until all tasks have been processed, at which point workers are signalled to shut down.

In this exercise, you will be implementing a simple task farm to compute fractal sets, building on the code from exercise 3. We will be using MPI for parallelisation. Unlike the previous practicals, in this one much more of the design decisions are left to you.

Template code

The template code contains the main program, which initialises and finalises the MPI processes, reads command line options, calls out to compute the fractal set, and then writes the resulting image to a file. The `compute_set` function spins out a master process (using the function `master_loop` and a number of slave processes (`worker_loop`). Everything else is left up to you.

Things to think about

Once the workers have started up, you need a way of sending the problem description to them. Some of the data is constant for the whole problem: the minimum and maximum pixel values, the grid size and the number of iterations. It probably doesn't make sense to send this data along with every task, so you could broadcast it from the master process to the works at the beginning of the master/worker loop. The following MPI function may be useful

```
MPI_Bcast(void *data, int count, MPI_Datatype type,
          int root, MPI_Comm comm)
```

```
call MPI_Bcast(data, count, type, root, comm, ierr)
   <type> :: data(*)
   integer :: count, type, root, comm, ierr
```

The datatype argument should be a defined MPI constant. Note that there are differences in the constants between Fortran and C versions. For the purposes of this practical, you'll need:

`MPI_INT` a C `int`

`MPI_FLOAT` a C `float`

`MPI_COMPLEX` a Fortran `complex`

`MPI_INTEGER` a Fortran `integer`

You'll need to be able to send a message from the master to a worker, we'll use blocking communications in this practical, because they're easiest to reason about. That means using `MPI_Send` to send messages, and `MPI_Recv` to receive them.

```
MPI_Send(void *data, int count, MPI_Datatype type,
         int dest, int tag, MPI_Comm comm)
```

```
call MPI_Send(data, count, type, dest, tag, comm, ierr)
   <type> :: data(*)
   integer :: count, type, dest, tag, comm, ierr
```

```
MPI_Recv(void *data, int count, MPI_Datatype type,
         int source, int tag, MPI_Comm comm,
         MPI_Status *status)
```

```
call MPI_recv(data, count, type, source, tag, comm, &
     status, ierr)
   <type> :: data(*)
   integer :: count, type, source, tag, comm, ierr
   integer, dimension(MPI_STATUS_SIZE) :: status
```

Note the master does not know which worker will be the first to complete its task, so it doesn't know what to give as the source argument to `MPI_Recv`. There are a few ways of dealing with this, but the simplest is to post a receive matching any possible sender, to do this pass the special `MPI_ANY_SOURCE` value as the source argument. You will then need to figure out which worker the data came from, this can be done by accessing values in the `status` variable. The sender of the message is `status.MPI_SOURCE` in C and `status(MPI_SOURCE)` in Fortran.

Debugging problems

The easiest thing to get wrong in this practical is a deadlock in the processes due to unmatched sends or receives. Think about how to ensure that workers don't post another "wait for more work" receive when there are no more tasks to send. Also think about how the master process can keep track of how many outstanding tasks there are (for which it will need to post a receive for the data).

Debugging MPI programs is even more difficult than debugging serial programs. You can either resort to printf-style debugging (make sure each line tags itself with the rank of the process writing it!), or (for small numbers of processes) you can use a parallel symbolic debugger. On HECToR you could use totalview, on ness you can do it with multiple serial debuggers, although this quickly gets painful.

```
 mpiexec -n 4 xterm -e gdb ./fractal-set
```

starts for xterm processes running `gdb` on the `fractal-set` executable. Type `run` in each resulting window and the processes start up. Now you can interrupt each process and inspect it like a normal serial debugger. Hopefully this will allow you to figure out where a deadlock is occurring.

An alternative implementation approach

If it turns out you implement the task farm with blocking communications relatively quickly, have a go at doing so with non-blocking communication. That is, using `MPI_Irecv` and `MPI_Isend`. In this case, the logic for waiting on tasks may be different. After the master sends a task, it can immediately post a receive for the results. Equally, the worker could post a receive for its next task before starting on the current one. With this approach, you may be able to overlap computation and communication to obtain slightly better performance, albeit at the expense of more complicated code.