

Parallel Programming Languages & Systems

Exercise Sheet 2

*This exercise sheet is assessed and must be your own work. Please be sure that you have read, understood and adhered to the School and University guidelines on late submission of coursework and academic misconduct, which can be found via the course webpage. It accounts for 10% of the course final mark. The deadline for electronic submission is **4pm on Thursday 19th March 2015**. You should submit a separate program (including corresponding discussion) for each part of the exercise. **You must use filenames as indicated in the exercise**, since only these will be accepted by the submission program. Thus, anything you add must be included in these files, even if good software engineering practice might suggest that additional files would be cleaner! The two programs (including discussion) are weighted 2:1 (ie the first program is worth twice as much as the second one).*

You will write two C/MPI programs which implement variants on a given parallel algorithm. Good marks will be awarded for following the specification, for appropriate use of MPI operations and for general clarity of code, presentation (for example appropriate commenting) and discussion.

Since it is not feasible for everyone to have sole access to a real parallel computer, you will be running your program on a normal DICE workstation, using Unix processes to emulate the parallelism. This means that you will not get any feel for the real performance of the algorithm. However, all the tricky conceptual issues of MPI programming will be very apparent to you and dealing with these is the real purpose of the exercise.

See the end of this document and the course website for information on how to compile and run MPI programs on DICE.

Part A: Adaptive Quadrature with a Bag of Tasks

Adaptive Quadrature is a recursive algorithm that computes an approximation of the integral of a function $F(x)$, using static quadrature rules on adaptively refined sub-intervals of the integration domain. It was discussed on slides 24-25 of the course overheads. In this exercise, in order to make the marking easier, you will be working with a fixed function F and fixed values for EPSILON , A and B . These have been hard coded into all the provided files. **If you are tempted to change them, for the sake of experimentation, then please be sure to restore the original values before submitting.**

In the file `aquadsequential.c` you will find a straightforward sequential implementation. You can use this to verify the results of your parallel implementations. To compile the sequential program, use the command

```
[mymachine]: gcc -o aquadsequential aquadsequential.c -lm
```

When you run it, you should see something like

```
[mymachine]mic: ./aquadsequential  
Area=7583461.801486
```

In this part of the exercise you are asked to implement a parallelized MPI version of the Adaptive Quadrature algorithm, using the *Bag of Tasks* pattern. Slide 29 already sketches out a shared memory version of such a program. Your job is to build something similar, but with a farmer process storing and controlling access to the bag, and a collection of worker processes interacting with the farmer via MPI communications to get tasks and return results (like the sketch on slide 31).

Writing the Program

In order to avoid wasting time with I/O in C, and in order to make the marking task a little easier, you are provided with a template program which you should expand. This can be copied from the course web page, where it is called `aquad.c`. **You should call your copy of the file `aquadPartA.c`.** As noted above, you are provided with the parameters for the adaptive quadrature algorithm (function, epsilon and range). The `main` function of the program is also provided and is responsible for MPI initialization, some variable initialization at process 0 (which will execute the farmer), and printing out of the results. As you can see, the program proceeds in typical SPMD fashion:

- process 0 initialises some data, then calls the farmer function which eventually returns the required area, then prints out some statistics;
- all other processes act as farm workers.

Your task is to implement the `farmer` and `worker` functions. For the farmer, you will need to

- Implement a data structure to store the bag of tasks. Files `stack.c` and `stack.h` contain a simple implementation of a stack which you could use for this purpose, although you are free to implement this some other way if you prefer. Notice that the stack has been specialised to store pairs of doubles (ie the two floating point numbers which represent the end points of a task). For example, the initial task will be from A to B. Only the farmer will ever access the bag directly. To help you understand the operation of the stack, a simple demo program, `stackdemo.c` has been provided. It contains its own compile and run instructions.
- Use MPI to interact with the workers, distributing tasks and receiving results until the computation is complete. Tasks should be distributed one at a time, as workers become free and/or tasks become available. Upon completing a task, a worker will send back to the farmer either a result (to be added to the overall total), or two

new tasks descriptions to be added to the bag. An obvious optimisation would be to have the worker keep one of the tasks and return the other, but **please do not implement this**. All new tasks should be passed back to the farmer for subsequent redistribution. Once the computation is complete, the workers should be passed a message indicating that they may now terminate.

- Accumulate the main result (ie the value of the integral), for eventual return, and also gather counts of the number of tasks executed by each worker. The array `tasks_per_process`, which is local to process 0, has been created for this purpose. Your farmer should increment the appropriate element of this array each time it sends out a new task. The code for printing results is already present in `main` and should not be changed.

For the worker function you will have to

- Use MPI to interact with the farmer.
- Terminate upon receiving the corresponding message from the farmer.
- During evaluation of a task, please call `usleep(SLEEPTIME)`. This is an artificial delay, required because you are (probably) running the program on only a single real processor. It has the effect of delaying return of a result for long enough to allow the farmer process time to interact with some other worker(s). Without this, depending upon your implementation strategy, it is possible that the farmer could be hijacked by a single worker. This wouldn't be required (and indeed would be crazy!) if you had real parallel processors for the workers.

When you run your program you can expect to see something like the following:

```
[mymachine]mic: /usr/lib64/openmpi/bin/mpirun -c 5 aquadPartA
Area=7583461.801486
```

Tasks Per Process

0	1	2	3	4
0	1643	1642	1641	1641

The area should be the same as reported by the sequential program. The number of tasks executed per worker will vary from run to run, but should appear to be reasonably well balanced, apart from process 0 which completes no tasks because it is the farmer.

Having completed the program, you should write a short description (of around half a page) explaining your implementation strategy and discussing your choice of MPI primitives, as opposed to any obvious alternatives. This discussion should be embedded in your source file as a comment.

Submit your work with the command

```
submit ppls 2 aquadPartA.c
```

Part B: Adaptive Quadrature with Naive Scheduling

The program you wrote for part A reported upon the balance of tasks across workers achieved by the dynamically scheduled bag of tasks approach. In part B you will explore the balance (or imbalance) which would arise if the scheduling was much more naive, by implementing another MPI program. In this naive version, the farmer simply divides the original range into w equal chunks (where w is the number of workers, one less than the total number of processes) and assigns a single chunk to each worker. There is no further allocation of tasks. The farmer finally gathers the local result from each worker and adds them together to produce the overall result. For example, with the range $A=0.0$, $B=5.0$ and $w = 4$, the four chunks would correspond to sub-ranges $(0.0, 1.25)$, $(1.25, 2.5)$, $(2.5, 3.75)$, $(3.75, 5.0)$.

Each worker has to compute the adaptive quadrature integral for its given chunk, using the normal recursive sequential method locally. While doing so, it should keep a count of how many calls are made to the quad function, since this will correspond to the number of tasks (in the sense of version A) which it has dealt with. Once complete, these counts should be gathered in by the farmer, in order to allow it to print out the information on tasks computed per worker.

*As in Part A, you should build upon a new copy of the provided program aquad.c. **You should call your new copy of the file aquadPartB.c.** You will probably also want to copy, and amend the quad function from aquadsequential.c. Since there is no bag of tasks in part B you will not need the stack code.*

When you run the program you should find that the balance of tasks computed by each worker is much worse than for part A. The static partitioning has produced a bad load balance, which would result in a poor run time if executed on a genuinely parallel system. This is because the distribution of work in the problem has not been evenly spread across the range. For example, you might see something like

```
[mymachine]mic: /usr/lib64/openmpi/bin/mpirun -c 5 ./aquadPartB
Area=7583461.801486
```

Tasks Per Process

0	1	2	3	4
0	39	195	1059	5271

For this naive partitioning, you may find that the result and/or the overall total number of tasks executed vary slightly from the bag of tasks version. This is a natural consequence of the initial division into w tasks (unlike the single initial task in the bag of tasks). As before, process 0 completes no tasks because it is the farmer. More interestingly, you should see that the load balance of tasks to workers is very poor.

Having completed the program, you should write a short description (of around half a page) explaining your implementation strategy and discussing your choice of MPI primitives as opposed to any obvious alternatives. This discussion should be embedded in your source file as a comment.

Submit your program for part B with the command

```
submit ppls 2 aquadPartB.c
```

Using MPI

First, see the file

```
http://www.inf.ed.ac.uk/teaching/courses/ppls/mpiHowTo.html
```

for information on how to use MPI on DICE. Having written your source program, in a file `aquadPartA.c`, you should compile it with the command

```
[mymachine]: /usr/lib64/openmpi/bin/mpicc -o aquadPartA aquadPartA.c stack.h stack.c
```

to create the executable `aquadPartA`. To run your successfully compiled program type

```
[mymachine]: /usr/lib64/openmpi/bin/mpirun -c 5 ./aquadPartA
```

which creates (for example) 5 MPI processes each running a copy of `aquadPartA` (in other words, this is SPMD style parallelism with one farmer and four workers).

The course website contains MPI implementations of some demo programs from lectures. You may find it useful to experiment with these while familiarizing yourself with MPI.