Divide and Conquer with a process pool

Nick Brown

22 February 2017

1 Introduction

In this practical we are going to look at a parallel mergesort, which will sort a list of random numbers in parallel. As we discussed in the divide and conquer lecture (lecture 4 on Learn), the sort routine will split the data in two, allocate half to another process and the rest to itself and both will recursively call sort again. Once the data size reaches a specific threshold then the sequential quicksort algorithm is used to sort a small number of elements. Then the data is returned to the caller and both streams are merged before being returned back to the process or iteration that called this one. Merge sort is a prime example of divide and conquer from the algorithm strategy space.

We will be using a process pool, which is an implementation of the master/worker pattern (lecture 9 on Learn) from the implementation strategy space. Basically there is one master process and many workers, the master maintains a pool of free workers and on start-up will allocate the initial (randomly generated) data to one of these. Each worker, as it splits its data will signal the master to activate a new worker process and the master will return the process id of this worker. When a worker starts it can gain access to the id of the process that signalled to create it (its parent, which will be the master or another worker.) These IDs can be used as the basis for point to point MPI communication and the sharing of data.

2 Provided code

You are provided with a skeleton implementation that you will need to flesh out to parallelise, a full process pool code, random number generator and quick sort implementation (for Fortran programmers, in C we use the standard library quicksort routine.) You can treat the process pool as a black box and provided here is a summary of its C API, with the Fortran API being similar.

Function	Description
int processPoolInit()	Initialises the process pool (1=worker, 2=master)
void processPoolFinalise()	Finalises and process pool (called from all)
int masterPoll()	Master polls to determine whether to continue or not
int workerSleep()	Worker waits for new task (1=new task, 0=stop)
int startWorkerProcess()	Starts a new worker task and returns the rank of this
int getCommandData()	Retrieves the rank of the task created this one
void shutdownPool()	Called by anyone to shut down the pool

Typically, when you run this code you want to ensure that there are more processes (in the pool) than you will need. For this problem about 20 should be sufficient.

3 Parallel divide and conquer

You are concentrating on the ***mergesort.c*** (or ***mergesort.F90***) file, which contain a skeleton implementation and it is your task to complete it. The code has been commented to give you an idea of aspects to consider and where to place these in the code. You will need to complete the following functionality:

- The sending of the entire unsorted data, from the master to the first worker it creates. This has been started for you in the ***startMergeSort*** routine which starts a worker from the process pool and obtains its id. It is up to you to consider the communication to use.
- The final receiving of data to the master after the workers have completed sorting the data. This is in the ***main*** routine (***mergesort_master*** in the Fortran code.)
- In the ***workerCode*** routine a newly activated worker will need to receive the data (and amount of data) to sort from its parent and then send back the sorted data (of the same size) after the ***sort*** routine has completed.
- In the ***sort*** function you will need to determine the pivot, split the data, send half to the newly created worker and recursively call ***sort*** with the remaining half. Once the ***sort*** routine has completed and the worker has returned its sorted data then this should be combined together with a call to the ***merge*** routine.
- Run it - you will need to ensure that there are more processes available (in the pool) than will be needed, I would suggest running with at least 20 processes on the front end and one node (36 physical cores) on the backend.