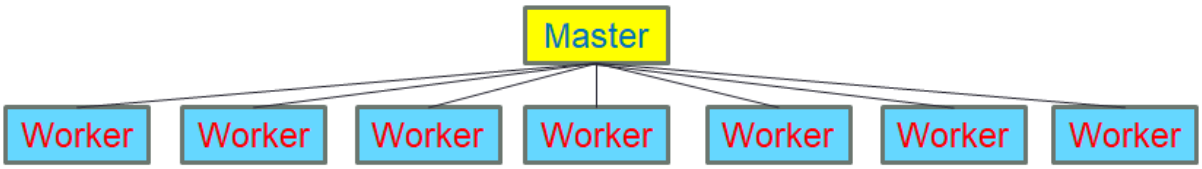
Divide and Conquer with a process pool

1 Introduction

In this practical we are going to look at parallelising a common sorting algorithm, mergesort, with our aim being to sort a list of random numbers in parallel. As we discussed in the divide and conquer lecture, 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 (the serial threshold) then the sequential quicksort algorithm is used to sort a small number of elements. Then the resulting, sorted, values are then sent back 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 from the implementation strategy space. Basically, there is one master UE (MPI process) and many workers. The master maintains a pool of free workers and on start-up will allocate initial (randomly generated) unsorted values 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 process pool code (implemented by the master/worker pattern), 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. Most of the interaction with the process pool is already in the skeleton code, so you shouldn't need to worry about it too much.

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 the default problem size (set by the command line arguments in the submission script) about 20 should be sufficient – so running on a single node of ARCHER with 24 cores is fine.

3 Parallel divide and conquer

We will be using MPI for this practical, if you are not so familiar or a bit rusty with this then the course materials of a recent ARCHER course at http://www.archer.ac.uk/training/course-material/2018/07/mpi-epcc/index.php are a good reference. A good reference for the MPI API can be found at https://www.open-mpi.org/doc/v3.0/

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 these data values should be combined with a call to the ***merge*** routine.
- Compile and run it! As we said before, 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 a full node of ARCHER. The default setting in the submission script (100 data elements, argument one; serial threshold of 10 elements, argument two; and to display both the sorted and unsorted data, argument three) requires 17 UEs (16 workers and a master) so a single node is fine for this problem size

By looking at the output (the sorted list of numbers), it should be fairly clear whether your parallelisation is working or not!

4 Overhead of the process pool and task granularity

The default setting, of 100 data elements and a serial threshold of 10 elements will require 16 workers and is likely to be rather inefficient, as each worker simply hasn't enough data to offset the cost of worker creation and communication. A key question is how does the amount of useful work (computation) compare against the overhead of parallelism.

1. Firstly, let's add in a metric for the cost of a task starting a worker (when it divides the data in half.) The ***start_task_time*** global variable is already declared, so pop some timing (using MPI_Wtime) around the ***startWorkerProcess*** call of the ***sort*** function and update this global variable using that timing. As the task blocks until the process pool responds with the newly activated worker rank, it is possible that this creation of workers might incur significant overhead.

2. Each task does quite a lot of communication, sending values to its children and parent. You will see two additional global variables, ***calc_time*** and ***comm_time***, hook up timings at places in your code that add to the values held in these.
3. For the overall run we want to know the minimum, maximum and average values across UEs for each of the three metrics you have coded. In the ***main*** function, just before the code exits, I suggest using ***MPI_Reduce*** calls with the MIN, MAX and SUM operators. You also need to be a bit careful as some UEs won't calculate any values for these metrics (for instance the process pool master or unused workers.) Therefore, to calculate the average I suggest instead of dividing the sum by the number of UEs, using a further reduction to sum up the processes that contribute a non-zero value to the metric.

Run the code with your new metrics and have a look at the timings produced. How does the amount of computation (the useful work) compare against the overhead (worker creation and communication)?

1. There is also a calculate_num_workers.py Python script supplied, this will report the number of workers required for a specific data size an serial threshold. For instance ***python calculate_num_workers.py 100 10*** will return the number of workers for the default problem size of 100 elements and a serial threshold of 10. Remember for the number of cores needed you will need to add an extra one onto to the number of workers to take account of the master UE.
2. In the submission script there are three arguments provided to the mergesort application, the first is the number of data elements (100 initially), the second is the serial threshold (10 initially) and the third is whether to display the unsorted and sorted data (1 initially.) Experiment with different data size, serial threshold and number of workers to see how these impact the metric values reported. ***IMPORTANT:*** As you increase the size of the data we suggest changing the display argument (the third argument) from 1 to 0. This will save lots of data being displayed, as once you are confident that your parallelisation is working correctly the actual data values are not that interesting!