Divide and Conquer with fork join

Nick Brown

2 March 2017

1 Introduction

In this practical we are going to stay with the mergesort, using divide and conquer, example that we considered in the previous practical and implement this parallelism via the Fork/Join implementation strategy. In practical five we considered using a process pool, which itself followed the master worker pattern, to support the parallelisation of the search algorithm. This required explicit worker creation/allocation and messaging between tasks to send unsorted and receive sorted data and added significantly to the complexity of the serial code.

By adopting the fork/join pattern we will see that the code is simpler as OpenMP is well suited to dynamically creating units of execution which are a crucial aspect of fork/join and in turn support the divide and conquer algorithm strategy.

2 OpenMP

If you took our threaded programming module then you will have covered OpenMP already. If not, or if you are a bit rusty on OpenMP, then a good online resource can be found at https://computing.llnl.gov/tutorials/openMP/ which should help you with the implementation of this practical.

Remember, in order to enable OpenMP directives in the compiler then you will need to provide the additional ***-fopenmp*** argument for GCC (and **-openmp** for Intel.)

3 Fork join mergesort

You should do this practical on the CP Lab machine as Cirrus will be unavailable

You have been provided with a serial version of the mergesort, ***mergesort.c*** and ***mergesort.F90***. Similarly to the previous practical, there is also a random number generator provided for generating the initial, unsorted, numbers ***ran2.c*** and ***ran2.F90*** along with a Fortran implementation of quicksort ***qsort.F90*** which is used after the data size reaches a certain minimal threshold. For the C version we instead use the standard ***qsort*** function from ***stdlib***.

Your task is to paralleise this serial code using the non-iterative constructs of OpenMP. You might find it useful to print out the current thread number (***omp_get_thread_num***) and the current level of nested parallelism (***omp_get_level***) to give you an idea of how threads are being created and used as part of this parallel recursion.

- Because the ***sort*** function is recursive, our use of directives represents nested parallelism. Basically we want to allow for a thread to be further split numerous times on each recursive call to the function. To support this you will need to call the ***omp_set_nested(1)*** function (or ***omp_set_nested(.true.)*** in Fortran) at the start of your code.
- First I would suggest using the ***parallel sections*** directive, with this construct wrapping both calls to ***sort*** each with their own ***section***. Effectively this is a non-iterative work sharing construct and specifies that the enclosed ***section(s)*** are to be divide amongst a team of threads. Using this directive, we can utilise nested parallelism, where a thread from the team is used for executing each ***section*** directive recursively.
- As an alternative to sections, one can instead use OpenMP tasks for thread parallelism. Unlike sections, which block at the end of the ***parallel sections*** directive, tasks will queue up and execute whenever possible (at task scheduling points.) Therefore, after issuing tasks, via the ***task*** directive, at a specific level of the sort you will need to wait for them to complete with the ***taskwait*** clause. One of the gotchas with this task approach is that each thread will try to queue up each task it encounters - therefore you either want to specify the ***parallel*** directive outside the ***sort*** function or enclose the task creation directives with a ***single*** clause.
- Remember, you can specify the maximum number of threads by exporting the environment variable ***OMP_NUM_THREADS*** and for this example you can set some arbitrarily large limit such as 32.