Divide and Conquer with fork join

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. Previously we considered the distributed memory approach of 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. In short, it added considerable complexity to the code, but this is a price we sometimes have to pay to get things running in parallel! In this practical we are limiting ourselves to a single memory space (a node of ARCHER) – we have less opportunity for large scale parallelism, but the modifications required are much simpler.

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 are not so familiar, or a bit rusty, with OpenMP, then a good online resource, which acts as a good reference manual, can be found at https://computing.llnl.gov/tutorials/openMP/ which should help you with the implementation of this practical. There is also a recent ARCHER course, http://www.archer.ac.uk/training/course-material/2018/03/openmp-soton/index.php, which covers all the concepts needed to complete this practical.

For the Cray compiler (which is the default on ARCHER and used by the provided makefile), no additional arguments are required. The makefile we have supplied should build the code fine. If you play with this after the course on your own machines, then you might need to explicitly enable OpenMP directives in the compiler, e.g. the ***-fopenmp*** argument for GCC (and **-qopenmp** for Intel.)

3 Fork join mergesort

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.

- We will first focus on the ***parallel sections*** directive, with this construct wrapping both calls to ***sort*** so that each of these is 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. Therefore, you want two ***sections*** directives, one for each recursive call into the ***sort*** function and a ***parallel sections*** wrapping these. For the ***parallel sections*** directive, you should also specify what data is shared between the sections.
- At the end of the ***parallel sections*** region of code, your application will block for both sections to complete. This is effectively fork/join, where each section forks to a separate thread and then the main thread waits for these to complete (join back up.)
- Timing is provided, see how your newly parallelised version using OpenMP sections to implement fork/join compares performance wise against the serial code.
- You can add some tracking of the OpenMP thread number, nesting level and pivot inside the sections (see code snippet). Do the reported numbers make sense?

```
printf ("My id %d my depth %d pivot=%d\n",  omp_get_thread_num(), omp_get_level(), pivot)
```

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 24 which will fill the cores of a node of ARCHER. In the submission script we automatically set this at 24 for you.

Once you have got the code working and running. you might see some warnings about oversubscription of threads to cores (and a long runtime!) This is because the ***OMP_NUM_THREADS*** variable corresponds to the number of threads PER PARALLEL REGION. We are using nested parallelism here, where each execution of the ***sort*** function is a new parallel region and as such OpenMP will create this number of threads for every execution of the function – which we don't want! Instead, set this to three (i.e. three threads per region where you have two sections in the ***sort*** function) and two where you are re-using the current thread to do half the sort. What impact does this have on performance?

4 Advanced - Fork/join using OpenMP tasks

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 what is called task scheduling points.) Tasks don't require the use of nested parallel regions, so you can avoid oversubscription problems (this is a major benefit of tasks over sections.)

- Define a ***parallel*** region around the ***sort*** call at the program entry point (***main*** function in C and the ***entryPoint*** procedure in Fortran.) At this point a team of threads will be created and it is important that not all of them call into sort, but instead only one single thread and the other threads available to execute tasks. Hence inside this parallel region, wrap the call to the ***sort*** function with the ***single*** clause too. This last bit is important, one of the gotchas with this task approach is that each thread will try to queue up each task it encounters and-so the ***single*** clause is important here.

- Inside the ***sort*** function create an OpenMP task for both recursive calls to the ***sort*** function. For each of these tasks you should specify what data is shared between this and other tasks.
- After issuing tasks in the ***sort*** function (at the specific level of the sort), you need to explicitly wait for these to complete before merging the results. You can do this via the ***taskwait*** clause.
- Again, add some tracking of the OpenMP thread number, nesting level and pivot in the tasks. Does this make sense?

Now re-run the code, what impact does using tasks have on the overall performance? If you follow the same *extension* that you adopted for OpenMP sections, where only one task is created for the first recursive ***sort*** call and the existing task executes the second recursive ***sort*** call, what impact does this have on performance? (Hint: Here you might want to consider experimenting with the ***taskyield*** construct too, which enables OpenMP to swap out the specific task and replace it with other tasks if deemed advantageous.)

- We only have one parallel region up until this point, with tasks being created recursively by other tasks in this region. It is also possible to create a new parallel region for each recursive level. This is very simple to do – just move your parallel region and single clause into the ***sort*** function, wrapping the task creation.
- How does the performance and reported OpenMP thread number and nesting level differ from the previous task version?

5 Advanced – NUMA region effects

If you have experience with OpenMP, then you will be aware that we need to be very careful with Non-Uniform Memory Access (NUMA) region affects. In ARCHER each processor (12 cores, two processors per node) is a NUMA region and crucially the memory *belonging* to one NUMA region is more expensive to access from the other NUMA region. Codes follow the *first touch* principal, where a page of memory belongs in the NUMA region corresponding to the core that first touched it. We have a potential problem in this example – core 0 touches all the memory when it generates the unsorted values. Hence there is a very high likelihood of experiencing a performance impact in going from 12 to 24 cores.

Do some experimentation here, do you see a *drop off* in performance when you use more than one NUNA region (processor) in ARCHER (e.g. OMP_NUM_THREADS from 12 to 24.) I suggest increasing the number of tasks because tasks will be mapped to available threads, so you might need to do a big run with a small serial threshold to get enough tasks so that they are mapped onto the cores of the other NUMA region.