Cray XE6 Performance Workshop
Cache and Memory Practicals

David Henty

1 Introduction

The basic aim of this practical is to run simple test programs to investigate the various features of caches on modern multicore systems. This includes aspects such as memory bandwidth, cache sizes, line sizes, coherency overheads and NUMA issues.

Today we will be running on the 32-core Interlagos processors used by the Cray XE6. Although quantitative measurements will differ between different processors, all the features we cover should be fairly generic amongst all multicore CPUs. One of the main reasons for using the Cray is that, unlike on most general-purpose Linux systems, the user has extremely fine-grained control over thread placement.

2 Memory Bandwidth

You should work in the `cache` directory for these exercises. Compile the code:

```
user@hector> cc -o cache cache.c location.c
```

and run a single copy of this serial program using:

```
user@hector> qsub submit.sh
```

- Look at the bandwidth figures and see if you can determine the cache boundaries. What are the sizes of the various caches?
- Edit the code so the summation loop runs backwards from n-1 to 0. How does this affect the bandwidth? What is the explanation?
- Use various non-unit strides in the summation loop, e.g. 2, 4, 8, 16 and 32. You should be able to determine the size of the cache lines from the point at which the bandwidth from memory (i.e. the bandwidth for very large n) is constant with respect to the stride. What is the cache line size?

3 Multicore

As an initial test of saturating the memory bandwidth, run 32 simultaneous copies of the code from Section 2. This can be done by setting both `mppwidth` and `mppnppn` to 32 in `submit.sh`. There will be a lot of output (since all 32 processes are writing to the same file), but you should be able to see the various effects.

How does the total memory bandwidth compare to the bandwidth available from a single core? How does this ratio change as the data size increases?

In the `multicore` directory you will see a version of the code, `cacheomp.c`, parallelised using OpenMP. Compile this code and run it using different numbers of threads. What effects do you observe?

You should compare the bandwidth on multiple threads to the bandwidth obtained with the serial code. How does this vary with data size? Can you relate this to the cache sizes on the Interlagos processor?

3.1 False Sharing

Look at the code in `coherency.c` and check you can understand what it does. To illustrate false sharing, this code has two threads continually accessing different elements of an array. If these elements are close together, they may be in the same cache line and cause false sharing. The code varies the separation between the two array elements to look for this effect.

The script `coherency.sh` ensures that the code runs on two cores, with threads placed on cores 0 and 2. On the Interlagos processor, pairs of cores share functional units so you will not see the desired effect if you use the default allocation on cores 0 and 1.

From the results, what is the cache line size?

4 NUMA

Look again at the `cacheomp.c` code. You will see that there is an OpenMP directive around the array initialisation that has been commented out. If you uncomment this line then the array will be initialised in parallel and the first-touch policy should give better memory placement for NUMA codes.

Note that for this exercise you *cannot* run multiple data sizes (i.e. different values of `n`) within the same code. You must set `n = NDBLE` in the loop. To change the data size, change the value of `NDBLE` and recompile.

Run the code on 32 cores and compare the results from the two initialisation policies. What is the difference for very large data sizes, and how do the results compare with those obtained previously by running 32 independent copies of the serial program? Do you see the same NUMA effects when running with data sizes that fit into the L2 cache?