CMPSC 240A HW-3 Report

Tanmoy Sanyal

October 23, 2016

Brief code summary

The code follows the logic outlined in the description of the assignment. `rec_cilkified` works by divide and conquer. The arrays are divided into two (not always equal) parts recursively and their individual dot products are summed up, until a certain threshold (COARSENESS) is reached. After that point, the dot products are done in a simple serial for loop. `cilk_spawn` is used in the splitting so that the recursive function calls are spawned off in different threads. `loop_cilkified` works in a similar way, but instead of recursion, it splits the arrays into chunks of length = COARSENESS and then these chunks are "dot"-ed individually and added up. `hyperobject_cilkified` simply uses the hyperobject construct of cilk++ and it takes care of everything.

For all the scaling experiments, I have assumed that the number of threads is the same as the number of cores (and set using the CILK_NWORKERS environment variable) i.e. hyperthreading has been assumed to be absent.

Scaling with size of array

For this experiment, the COARSENESS was chosen to be 50, and was run on all 24 cores of a single Comet compute node. The array sizes were varied from 10^4 to 10^10. However, the sequential algorithm run-time for sizes below 10^6 is almost negligible, and reported as 0.

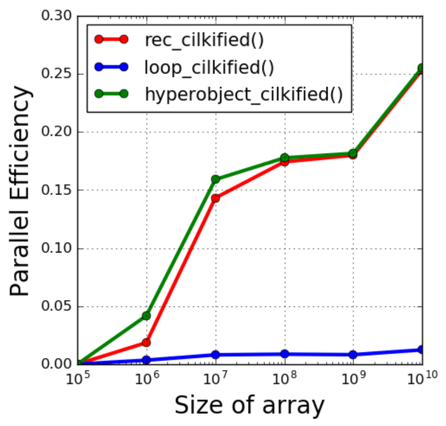

Figure 1: Parallel efficiency vs array size, NCores = 24, COARSENESS = 50

Scaling with number of cores

For this experiment, the array size was fixed at 10^6 and COARSENESS was maintained at 50. The number of cores was varied from 1 to 64 in powers of 2. Note that the work (t_1) for this experiment is not the runtime of the sequential function `std::inner_product`, but the time obtained with 1 core and CILK_NWORKERS =1 .(For 32 and 64 cores, I use 16 cores per compute node and multiple nodes).

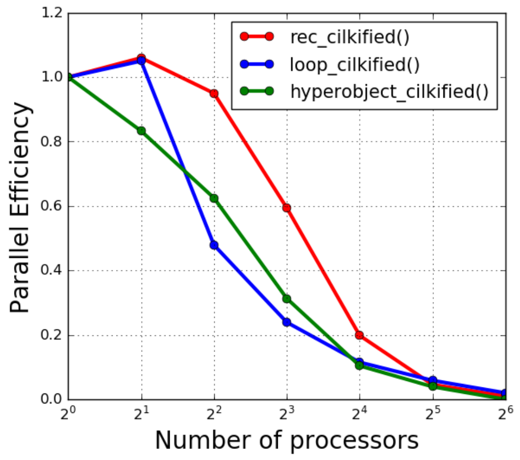

Figure 2: Parallel efficiency vs number of processors, Array size = 10^6, COARSENESS = 50

Sensitivity to COARSENESS

For this experiment, the array size was fixed at 10^8 and was run on all 24 cores of a single Comet compute node. COARSENESS was varied from 1 to 10^8 in powers of 10.

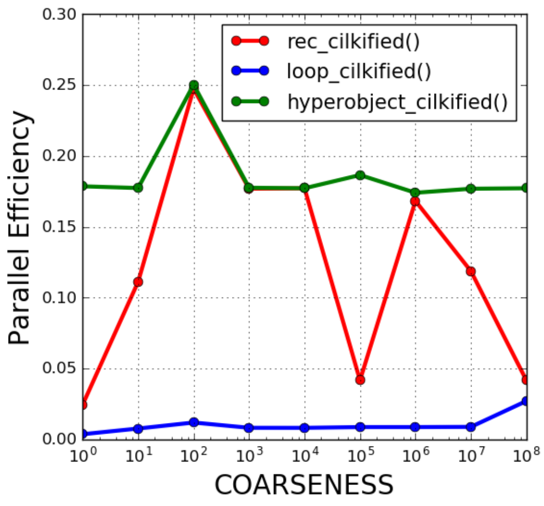

Figure 3: Parallel efficiency vs COARSENESS, Array size = 10^8, NCores = 24

Best performing dot-product code

From the different scaling studies presented above, Fig. 1 shows that with increasing number of cores (and therefore spawned threads), the performance goes down. Thus, the performance is never truly a function of how much computational resources I have at my disposal. On the other hand, increasing array sizes is a very real world phenomenon, and it forms the primary metric on which to judge performance. For large array sizes, `rec_cilkified` and `hyperboject_cilkified` have nearly similar parallel efficiency in Fig. 2. So any of these two might be a good candidate. Now, coming to COARSENESS, we see from Fig. 3, that `rec_cilkified` is much more sensitive than `hyperobject_cilkified`. That means that the former is amenable to easy tuning.

On the basis of the above insights, I would use the following parameters for the best-performing dot product code:

- Function: `rec_cilkified()`
- COARSENESS: 100

The speedup obtained using these settings is: Clearly, this does not enjoy linear speedup. Beyond

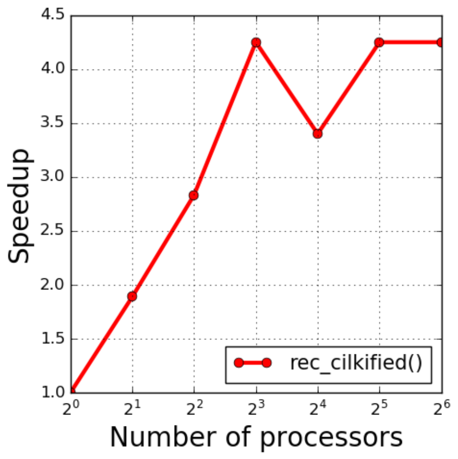

Figure 4: Speedup vs number of cores with `rec_cilkified`, Array size = 10^6, COARSENESS = 100

16 processors, the speedup saturates, because the threads are no longer in truly shared memory and span different Comet nodes. That is a trivial bottleneck. But the speedup also falls (and drastically so) from 8 to 16 processors, which are all within the same node. TODO: Reason

Tracing parallelism in the code