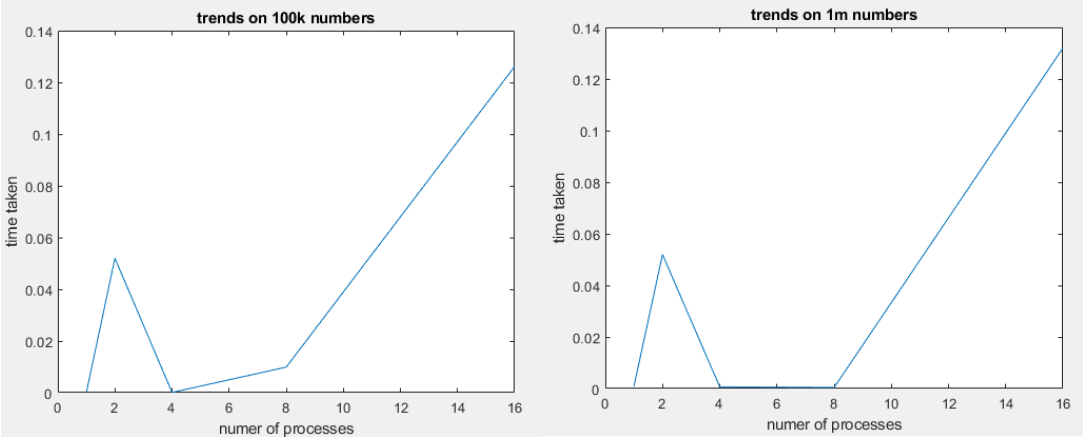
The charmpp program of prefix sum contains two files, prefix.ci and Prefix.C. The prefix.ci file declares the global variables and the methods of Chare classes that will be used in Prefix.C. In the Prefix.C file, firstly a CProxy_Main class is defined with some private variables (output file names, timers) declared. Then in main chare I read the data into a vector and cut it into smaller arrays and distribute to each Chare. Then I let each chare to report it finishes receiving data in case the following steps happen before all chares have been ready.

Then I write in a nonblocking parallelism. I create a private dictionary in each chare to store the messages passed from other chares. This is more efficient because some chares have fewer computation jobs than the other, then if I force all chares to synchronize with each other at each phase, these chares need to wait for a long time with noting to do. With the dictionary, I will infinitely store the messages passed from other chares and set the key as the phase number when the message is sent, and set the value as the summation of the previous chare. Then each chare can take the message out from the dictionary and operate differently with respect to different phases.

It can be calculated that the chare with index i needs to receive floor(log2(i)+1) number of messages, so I can check whether a chare has already received all the messages need. If one chare finishes all its receiving job, then only sending jobs are remaining. Finally, to guarantee a chare finishes all its jobs, 1. It receives all messages needed. 2. Its current phase is beyond the max phase, or the current phase is the max phase but the chare it want to send to is out of index. If the conditions are satisfied, then a global variable numDone will increment by 1. If numDone == numChares, it means all chares finish their jobs and I'll call the done() method in the main proxy.

In the done() method, a write(output_file_name) function in chare 0 will be invoked. The function will be called recursively, where chare 0 will firstly write output to a file, then invoke the write() in chare 2 and so on. Thus, the writing job is actually sequential because the order is greatly important.



The performance on 100 thousand and 1million number of integers are shown above. The trends are consistent, but pretty random. All the time taken is measured using number of chares equal to 4*num of processes. The time taken is almost the same while using 1, 4 and 8 processes, and becomes worse after 16 processes. This is because the overhead of communication overweighs the benefits of computation. Using 1 processes but 4 chares perform surprisingly the best. Maybe it is because it's difficult to synchronize the jobs when there are too many chares.