University of California, Davis

ECS 158

Programming on Parallel Architectures

Parallelization of R's combn() Function

Authors:
Trisha Funtanilla
Syeda Inamdar
Eva Li
Jennifer Wong

Professor:
Norm Matloff

March 16, 2015

1 Objectve

2 Function

2.1 Description

2.2 Reasons for Selection

3 Parallel Implementations

3.1 OpenMP

3.1.1 Load Balancing

Initially, the program was parallelized by directingly using OpenMP's loop directive `for`. By using the `for` directive, we are allowing OpenMP to handle the distribution of the tasks to the threads. This makes it easier for the programmer; however, it isn't necessarily the best way to distribute the tasks. For our program, it caused an extreme load imbalance. The distribution of the task was heavily skewed such that the thread with lower id's generated far more combinations than the thread with higher id's. This is because for each element i in the input vector x, the number of combinations that can be generated with x_i is larger than that with x_{i+1}.

In order to compensate for the load imbalance, the assignment of tasks to the threads was done in a wrap around manner. Using this method, for the first distribution, each thread will work on the element indexed in the same value as their id (e.g. thread 0 on $x[0]$, thread 1 on $x[1]$, and so on). Then, the following distribution depends on the total number of threads, the id of each thread, and the index of its previous assignment. With `nth` corresponding to the total number of threads and `me` corresponding to the thread id, the assignment following the initial distribution is determined using the equation:

$$next\_task = prev\_task + 2 * (nth - me - 1) + 1 \tag{1}$$

The distribution that will then follow the above depends on the id of each thread the index of the previous task. This next distribution is determined by the equation:

$$next\_task = prev\_task + 2 * me + 1 \tag{2}$$

The succeeding distributions alternates in using the two equations until the last distribution, which is for $n - m + 1$ (handled by thread 0) or the last x_i that can form a combination of size m with the elements succeeding it.

In this algorithm, each thread knows which indexes in x to generate combinations for. The threads do not need to communicate with each other, avoiding huge overhead especially for large values of n.

3.1.2 Chunk Size

The equations from the section above implicitly defines the chunk size so that each thread gets assigned roughly the same number $x_i's$ to work on. It is essentially the number of times task distributions occur in the load balance algorithm, which is roughly $(n - m + 1)/nth$.

3.1.3 Scheduling Policies

Additional optional arguments `schedule` and `chunksize` were added to the function, allowing the user to select between static, dynamic, and guided policies and any integer for the chunksize. If neither is provided, the default scheduling policy of our program is static, since we are both manually determining the chunk size and assigning the tasks to each thread. Dynamic and guided selections for scheduling make use of OpenMP's scheduling clauses.

3.1.4 Cache Considerations

4 Timing Comparisons

5 Conclusions

1 Appendix