ECS 158 Final Report:
combn() in Snow, OpenMP, and CUDA

Stefan Peterson
Bijan Agahi
Arjun Bharadwaj

Abstract

Parallel architectures are becoming increasingly prevalent in the modern world and as such have spawned multiple frameworks to assist with parallel programming, but these frameworks vary greatly in their implementation and performance. This paper takes a close look at three frameworks: R-Snow, OpenMP and CUDA. We took on the task of re-writing the combn() function found in R using each of these frameworks. We measure run times of these tests at different sizes and with varying inputs and levels of difficulty. In the end, we point out the strengths and weaknesses of each language as well as note on features which we, as programmers, find particularly useful or troublesome.

Contents

1 Introduction

With the relatively recent introduction of widely available multi-processor CPUs in household computers, parallel programming has become a hot topic in the world of computer science. In particular, the graphics and games industry has driven the multithreaded programming style to the forefront of everyone's attention. Through the process of splitting up tasks and distributing those tasks to individual cores on the CPU or GPU, parallel processing in most cases can provide vast improvements in both performance and efficiency. OpenMP, R-Snow, and CUDA are three such parallel languages, and this paper aims to compare and contrast the strengths, weaknesses, and performance of each language.

1.1 Setup for Benchmarks

All tests for the purposes of this paper were run on Tetraat University of California, Davis. This computer consists of a 64-bit 8-core processor with 16 GB or RAM and an Intel Core i7-2600K CPU with a clock speed of 3.4 GHz and a cache size of 8192 KB. The machine was running release 19 of the linux distribution Fedora.

2 combn() in R-Snow

2.1 combn() in R-Snow: Strategy

2.2 combn() in R-Snow: Implementation

2.3 combn() in R-Snow: Data

2.4 combn() in R-Snow: Analysis

3 combn() in OpenMP

3.1 Strategy

For the OpenMP version of the combinations function, we decided to parallelize by dedicating each thread to a subsection of the problem. Each subsection corresponds to the group of combinations which begin with the same value. In [Appendix Graphic 2] we see all the possible combinations for a $\binom{5}{3}$. Those can be clumped into the three groups each beginning with 1, 2, or 3. `Thread 1` will handle the computation of all combinations beginning with 1, `thread 2` will handle those beginning with 2 and `thread 3` will handle those beginning with 3. Obviously this is not load balanced for small cases since `thread 1` is doing significantly more work than the rest of the threads. When we get to larger problems, we use the built in dynamic scheduling of OpenMP to make an attempt at load balancing. If we had only 2 threads running on the above problem, `thread 1` would still do its chunk of the computation, then `thread 2` would do its and when it finishes (which should be before `thread 1`) it grabs the next process to be executed.

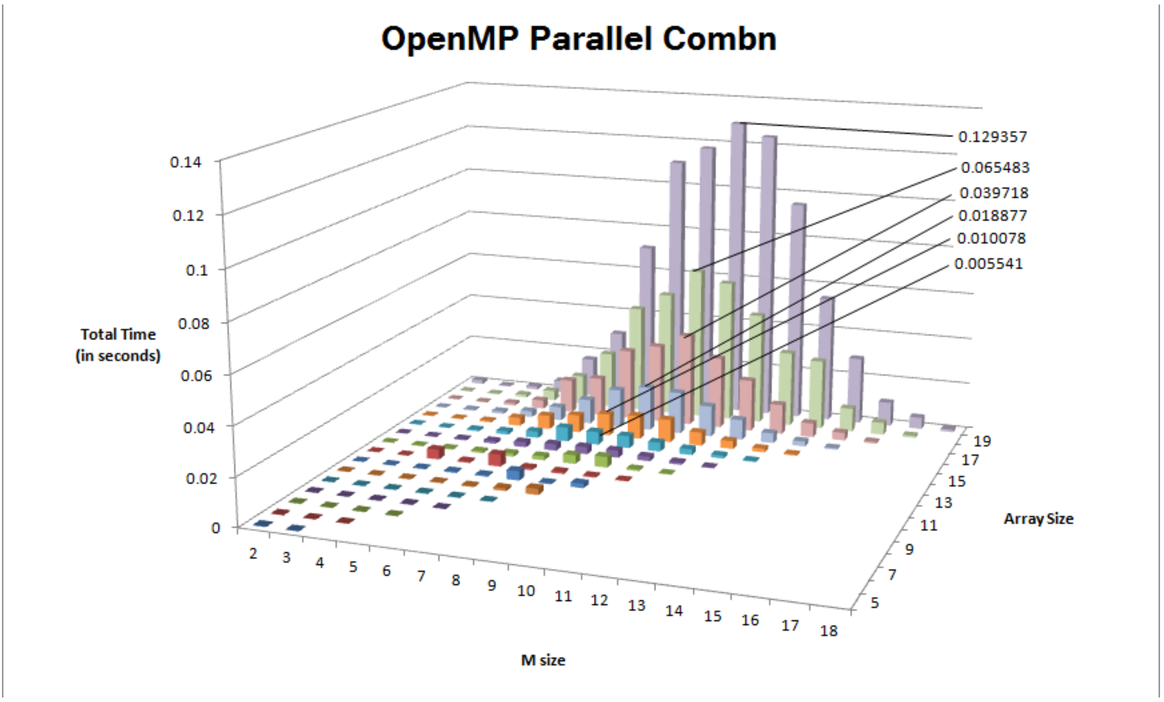
3.2 Implementation

All the references to code will be references to Appendix: A.1

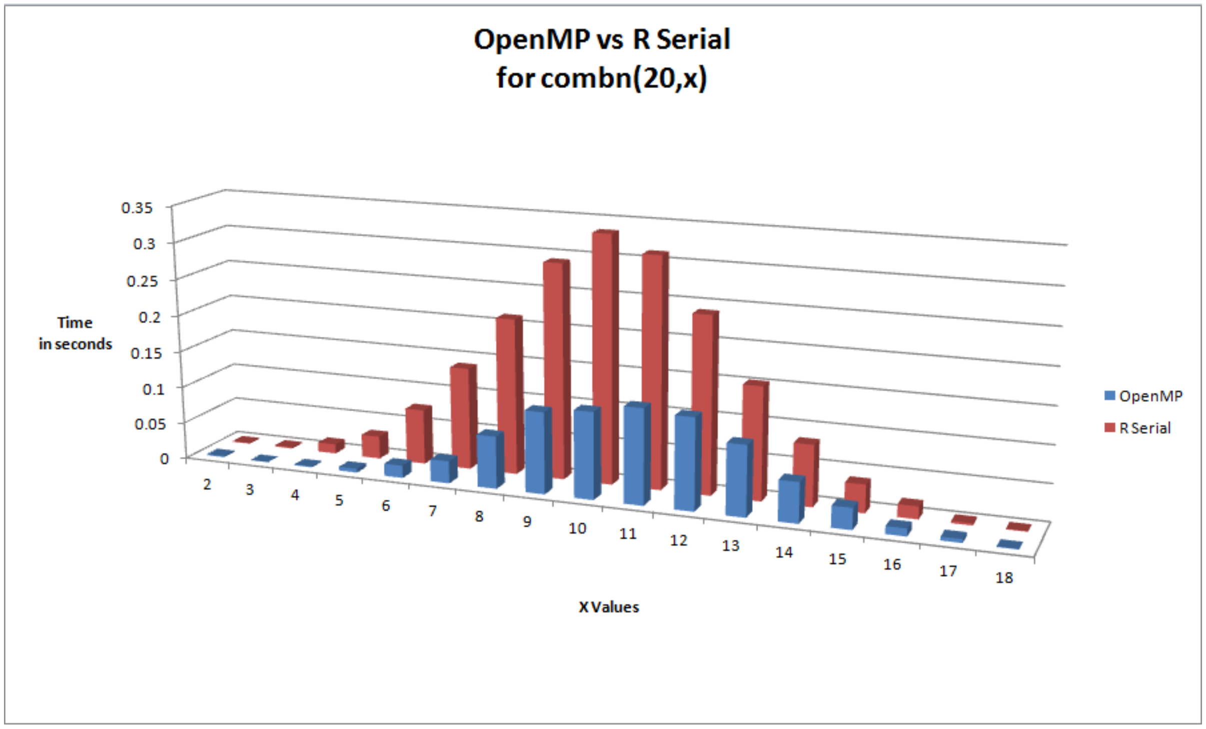
We decided to take a recursive approach to the implementation of the combinations function in C++. The core functionality of the program can be found on lines 23-33 in the method findCombs. This method takes in as its parameters 2 integers, a vector, and a 2d vector. Upon the first call of findCombs() the vector combination will already have been initialized to contain the first value in the combination (i.e. if we assume we are looking for the combination [1,2,3] the vector will be initialized to contain [1] and this is done on line 71 in the combn() function. From there the recursive algorithm will add the next number (i.e. combinations will now contain [1,2]) and then will call findCombs() recursively until the base case is met upon which the vector combination is pushed on to the 2D vector. When the recursion is done and the 2D vector contains all its possible combinations, the function vectorToArray() is called and the 2D vector is parsed and its values are added to the appropriate region of the global array.

3.3 Data

3.3.1 Graphic A1



3.3.2 Graphic A2



3.4 Analysis

Looking at `Graphic A1` we can see that our implementation of *combn*() in OpenMP follows the standard bell curve where given a value x, its maximum number of calculations and therefore its longest running time occurs when $m = \frac{x}{2}$. We can see this by looking at Pascal's Triangle [Appendix Graphic 1] which directly correlates to the problem of creating a list of combinations. Those values which lie at the center of each row, vertically down the middle, are those with the greatest value for that given row. These values correspond to $\binom{x}{m}$ where again $m = \frac{x}{2}$. When *combn*() is passed these values this is when we see the greatest running time for the function.

Looking at `Graphic A2` we can see that our implementation of *combn*() is on average 2-3 times faster than the serial built in version provided in the utils package of the R library. This is to be expected since we are able to generate combinations in parallel speed up could be further improved by taking our approach of parallelization further down and assigning tasks recursively.

4 combn() in Thrust

4.1 combn() in Thrust: Strategy

4.2 combn() in Thrust: Implementation

4.3 combn() in Thrust: Data

4.4 combn() in Thrust: Analysis

5 Conclusion

6 Bibliography

A Appendix: Code

A.1 combn() — OpenMP implementation

combnOpenMP.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <algorithm>    // std::find
#include <omp.h>

using namespace std;

vector<int> values;
int *allCombs;
int *numEntriesPerLevel;

double choose(int n, int k) {
    if(k == 0) return 1;
    return (n * choose(n - 1, k - 1)) / k;
}

void addComb(vector<vector<int> > &v, vector<int> &v2) {
  v.push_back(v2);
}

void findCombs(int offset, int k, vector<vector <int> > &v, vector<int>
    combination) {
  if (k == 0) {
    addComb(v, combination);
    return;
  }
  for(int i = offset; i <= values.size() - k; ++i) {
    combination.push_back(values[i]);
    findCombs(i + 1, k - 1, v, combination);
    combination.pop_back();
  }
}

void findEntriesPerLevel(int *array, int x, int m) {
  array[0] = 0;
  for(int i = 1; i < x - m + 1; i++) {
    array[i] = array[i - 1] + (choose(x - i, m - 1) * m);
  }
}

void vectorToArray(vector<vector <int> > &v, int a[], int startIndex) {
  vector< vector<int> >::const_iterator row;
    vector<int>::const_iterator col;
    int index = startIndex;

    for (row = v.begin(); row != v.end(); ++row)
    {
         for (col = row->begin(); col != row->end(); ++col)
         {
            a[index] = *col;
52             index++;
53          }
54     }
55 }
56
57 int * combn(int x, int m) {
58   int size = choose(x,m);
59   for(int i = 0; i < x; ++i)
60     values.push_back(i+1);
61
62   allCombs = new int[size * m];
63   numEntriesPerLevel = new int[x - m + 1];
64   findEntriesPerLevel(numEntriesPerLevel, x, m);
65
66
67   #pragma omp parallel for schedule(dynamic)
68   for(int i = 1; i <= x - m + 1; i++){
69     vector<vector <int> > levelCombinations;
70     vector<int> combination;
71     combination.push_back(i);
72     findCombs(i, m - 1, levelCombinations, combination);
73     vectorToArray(levelCombinations, allCombs, numEntriesPerLevel[i - 1]);
74     combination.pop_back();
75   }
76   #pragma omp barrier
77
78   return allCombs;
79 }
```

This is a recursive implementation of the combinations function. Here we used `#pragma omp parallel for schec` on line 67 to help load balance the parallelization of the threading.

A.2 combn() — Thrust implementation

combnThrust.cilk

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

using namespace std;




int * combn(int x, int m) {
  int *ret = new int[x * m];

  int loc = 0;
  for(int i = 0; i < x; i++) {
    for(int j = i; j < i + m; j++) {
      ret[loc] = j;
      loc++;
    }
  }
  return ret;
}

```

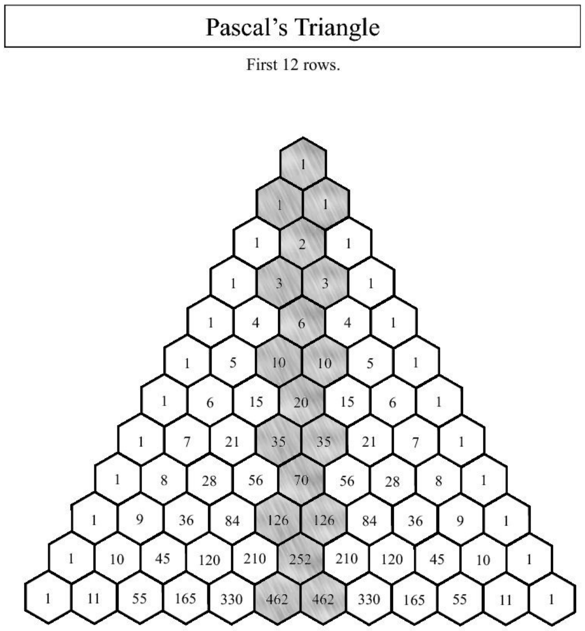
```

template <typename T>
T * combn(T x, int m) {


}


int main (int argc, char** argv) {

  int x = 10;
  int m = 2;
  int * vals[x * m];
    vals = combn(x,m);
    for (int i = 0; i < x * m; i++) {
      cout << vals[x];
    }
    return 0;
}
```

This code is directly translate from the OpenMP version above, using the same algorithm. We have various Cilk++ equivalents of OpenMP, including: the

There are also a variety of files specific for testing purposes included in the submitted .tar file.

B Appendix: Graphics

B.1 Graphic D1



B.2 Graphic D2

$_5C_3$

```
a1  1 1 1 1 1 1
a2  2 2 2 3 3 4
a3  3 4 5 4 5 5

b1  2 2 2
b2  3 3 4
b3  4 5 5

c1  3
c2  4
c3  5
```

C Appendix: Member Contributions

Stefan Peterson

- LaTeX Formatting, editing
- OpenMP implementation
- OpenMP write-up
- Abstract
- Introduction
- Conclusion
- Relevant research

Bijan Agahi

- R-Snow implementation
- Experiment data collection
- Test scripting
- Graphing in R, tabling
- Relevant research

Arjun Bharadwaj

- Thrust implementation
- Relevant research