

Final Project - Parallel Dice

Nelson Johansen
Ricardo Matsui
Michael Polyakov

1 Background

The dice package by Dylan Arena [1] provides *getSumProbs* and *getEventProbs* functions which give the probability of a series of n-sided dice events given a number of rolls and dice. Of the two, the *getEventProbs* function is more interesting since it allows for the events to occur in any order and can have ranges for each events. For example, the events may be that the sum of the dice on one roll is between two and four and on another roll the sum is five, but the sums may occur in either order. Because of this, it can be parallelized in the case where the order of the events does not matter because the total probability is comprised of a sum of the probabilities of the unique combinations of the possible sums. This allows for the work to be divided into multiple threads to calculate the probabilities of each case and then combined together.

For example, here is the usage of the *dice* package for calculating the probability that two six-sided dice will have the sums of two on one roll, three to five on another, and three on the remaining roll in three rolls:

```
1 > getEventProb(nr=3, nd=2, ns=6, list(2, 3:5, 3),  
2 + orderMatters=FALSE)  
3 [1] 0.002057613
```

There can also be more rolls than events which means that the outcomes of some rolls do not matter. For example, in the case of eight rolls the probability is much higher that the events will occur:

```
1 > getEventProb(nr=8, nd=2, ns=6, list(2, 3:5, 3),  
2 + orderMatters=FALSE)  
3 [1] 0.05352239
```

2 Original Performance

The performance of the package was poor even for relatively low number of rolls and dice. By adding markers and timing code to the dice package at key locations in the code, we can determine that most of the time is spent calculating the probabilities for each unique case of rolls that match the criteria as seen in Figure 1.

However, if the number of rolls increases but the length of events does not increase, the stage for building the unique matrix of combinations of roll sums becomes a major factor as well because it requires more time determine which permutations of the rolls are

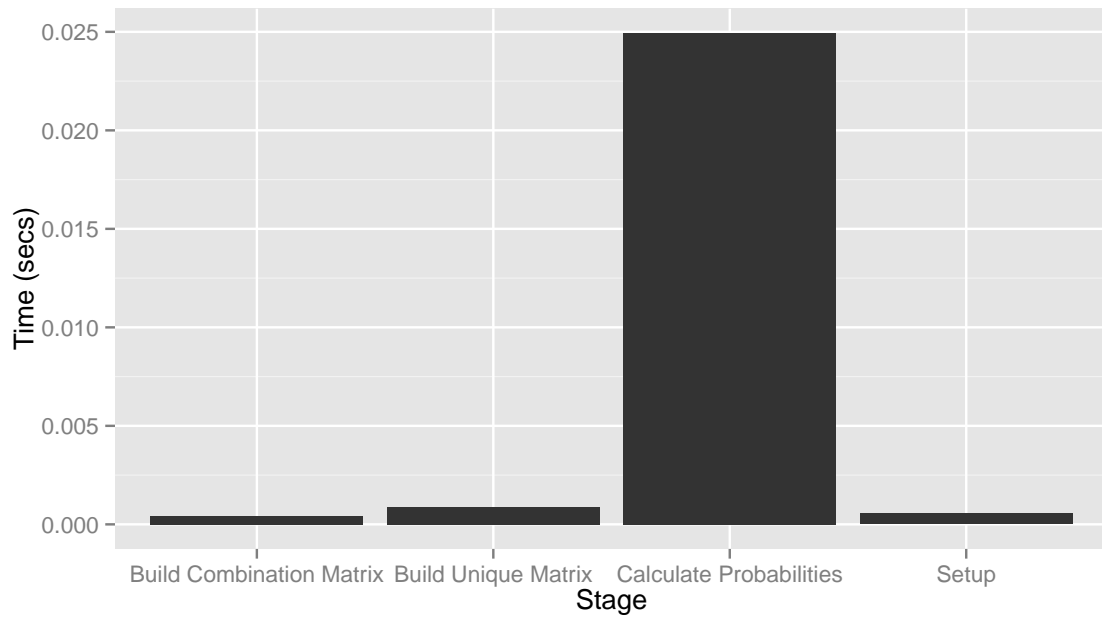


Figure 1: Timing of calling *getEventProb* with $nr = 3$ and three events specified duplicates. Figure 2 shows this timing.

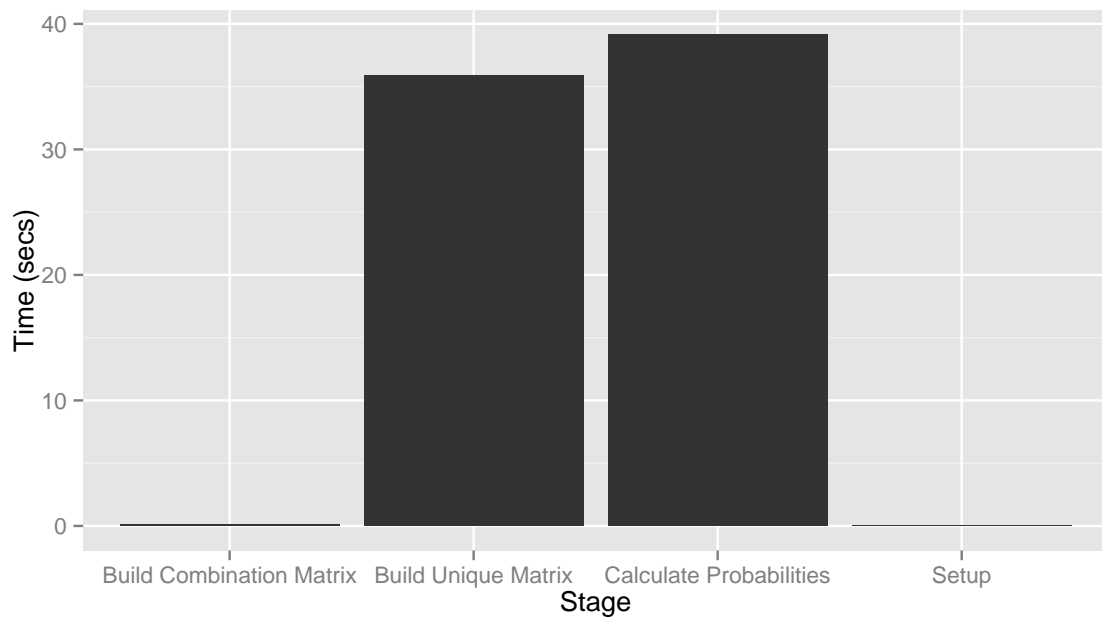


Figure 2: Timing of calling *getEventProb* with $nr = 8$ and three events specified

If the number of events specified is increased to match the roll count, then most of the time is spent again in calculating the probabilities as can be seen in Figure 3. This is because the number of permutations is significantly lower.

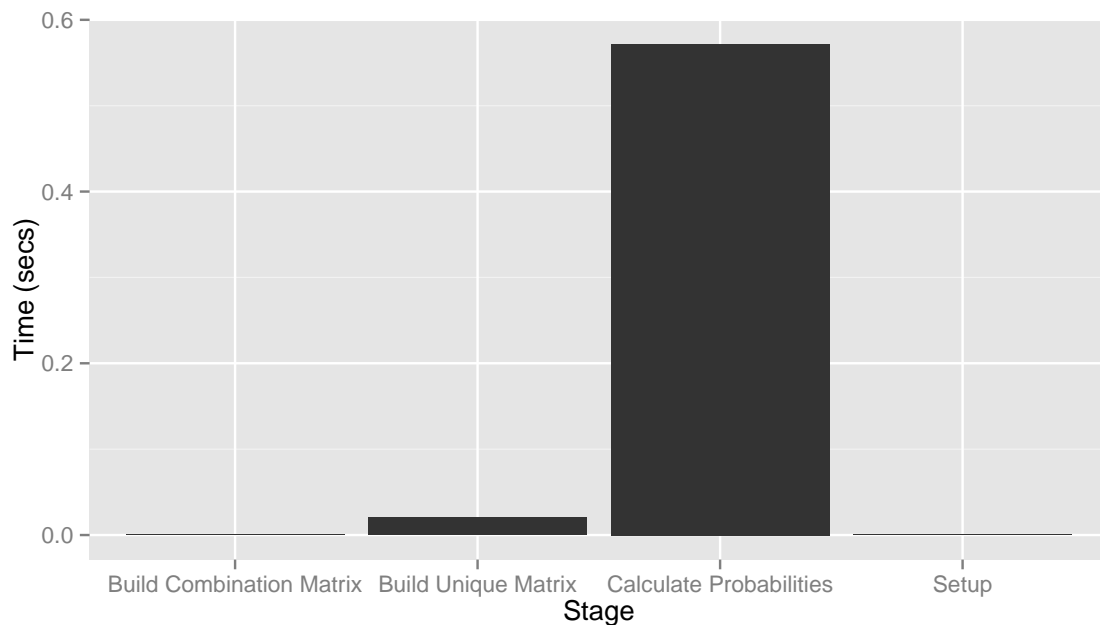


Figure 3: Timing of calling *getEventProb* with $nr = 8$ and eight events specified

Increasing the number of sides by die and the dice counts results in the same effect. By these graphs we can conclude that most of our efforts should be spent optimizing these two portions of the function *getEventProb*. These are namely the portion that computes the probabilities:

```

1 sumOfProbs = sum(apply(combMatrix, 1,
2     function(x) getEventProb(nrolls,
3         ndicePerRoll,
4         nsidesPerDie,
5         as.list(x),
6         orderMatters)))

```

and the portion that creates the unique matrix:

```

1 if (nrolls > 1)
2 {
3     combMatrix = unique(t(apply(combMatrix, 1, sort)))
4 }
5 else
6 {

```

```

7         combMatrix = unique(combMatrix)
8     }

```

3 Code Optimization

By examining the recursive call in calculating the *sumOfProbs* in the previous section, we can see that there are many computations that are duplicated even though their results remain constant throughout the process of computing the total probability. Specifically, there are many repeated calls to *getSumProbs* in *getEventListProbs*. The function *getSumProbs* only depends on *ndicePerRoll* and *nsidesPerDie* which does not change during the calculation, and it calculates the probability of each sum event. As a result, it can be cached and then reused. This significantly decreases the time required for the original code to execute when the number of dice increases as can be seen in Figure 4 for executing with the following arguments:

```

1  getEventProb(nrolls=3, ndicePerRoll=rolls , nsidesPerDie=6,
2              eventList=rep(list(rolls:(5*rolls)), 3), orderMatters=F)

```

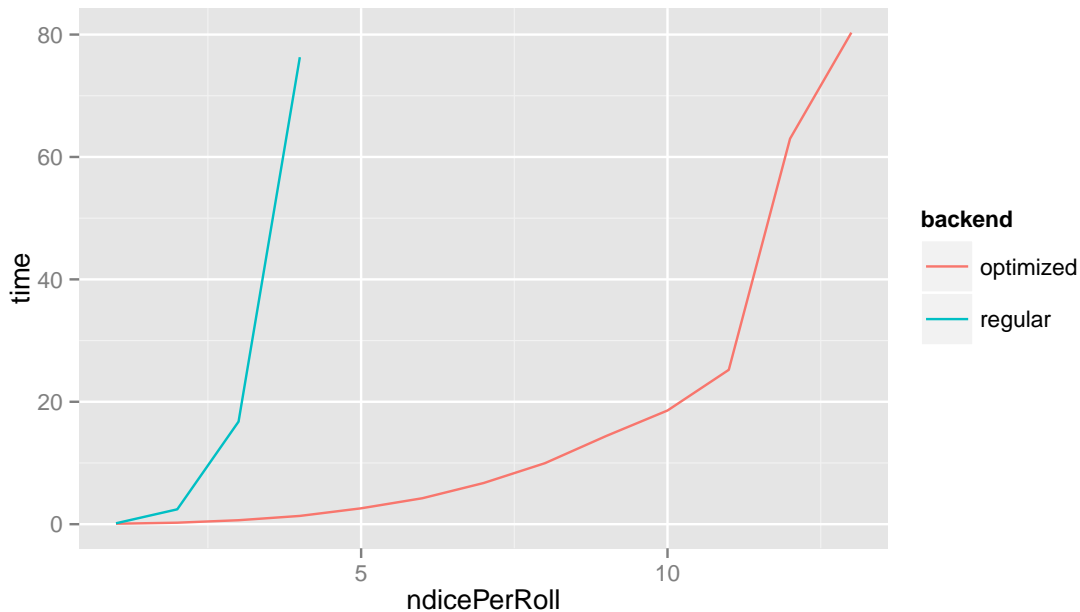


Figure 4: Timing difference between original code and optimized version for number of dice

Note that the event list is chosen to have a large range so that more computation is required. If the number of rolls is increased instead, then the optimizations do not have much effect on the execution time as can be seen in Figure 5.

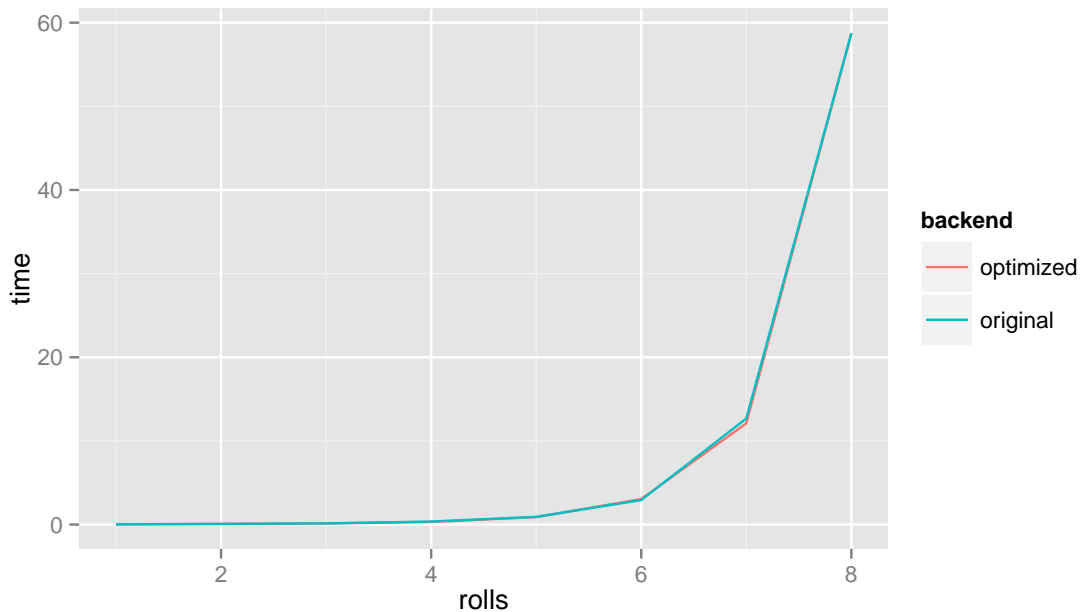


Figure 5: Timing difference between original code and optimized version for number of rolls

4 Parallelizing Using *snow*

As mentioned previously, most of the computation is done in the calculating of the probabilities, and these calculations are clearly independent from each other as the probability of each combination of rolls can be determined without information from the others. Therefore the code can be parallelized relatively easily by computing the probabilities of each combination in parallel using *parRapply* from the *snow* package [3].

```

1 probs = snowGetSumProbs(ndicePerRoll, nsidesPerDie)$probabilities
2 clusterExport(cls, c("snowGetEventProb",
3   ".snowGetEventListProbs", ".checkIntParam",
4   ".checkLogicalParam", "combinations"),
5   envir=environment())
6 sumOfProbs = sum(parRapply(cls, combMatrix,
7   function(x) {
8     snowGetEventProb(nrolls,
9       ndicePerRoll,
10      nsidesPerDie,
11      as.list(x),
12      orderMatters,
13      F,
```

```

14                                     probs)
15     })))
16 outcomeProb = sumOfProbs

```

This allows for the probability calculation stage to be spread over multiple cores. And yields slightly better performance as seen in Figure 6 for approximately $nDicePerRoll > 6$ and four cores. For around $nDicePerRoll \leq 6$, the overhead incurred from synchronizing and transferring data to the cluster outweighs its gains from parallelization. However, even for the last case of thirteen dice, the difference in execution time is $(80.3 \text{ sec} - 59.0 \text{ sec}) / 80.3 \text{ sec} = 26.5\%$, which is significantly faster. This is because the work of calculating the probabilities of each row of *combMatrix* can be divided across all the cores rather than only having one core do the calculations as in the original package code.

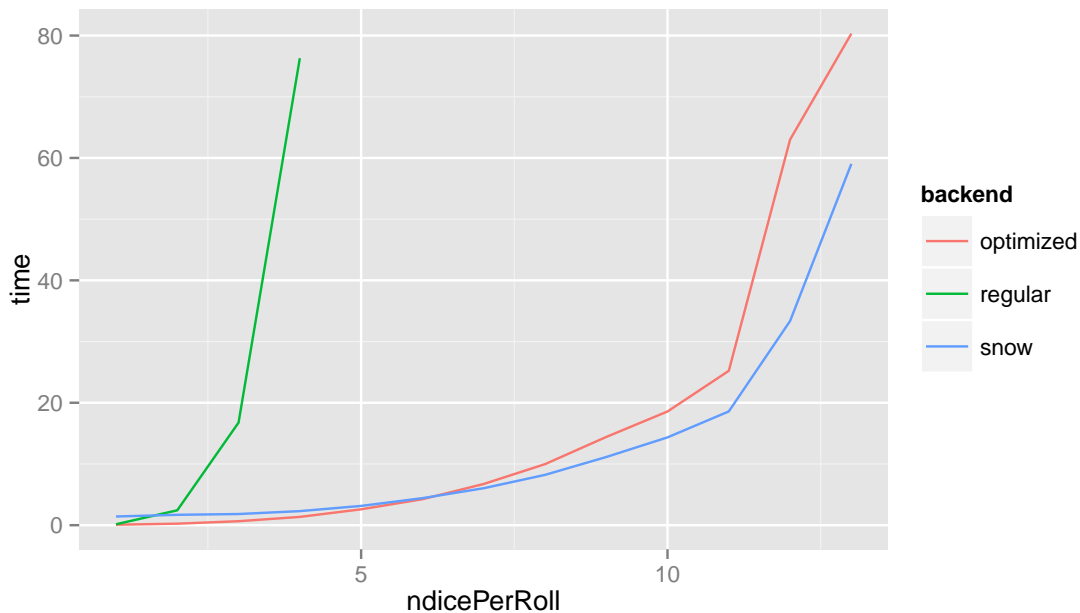


Figure 6: Timing difference for code parallelized using *snow*

Given how relatively trivial the use of *snow* was, the performance improvements are very good. For the following parallelizations, it will require significantly more effort to port the current computations into C++ and parallelize the computation.

5 Parallelizing Using OpenMP

The OpenMP version of the algorithm works in a similar way by dividing the work by row to the different threads. On the main challenges here was reimplementing some of

the package in C++ to be run more efficiently than in R. This re-writing improves the performance significantly as Figure 7 shows. A native implementation of the core part of the package helps speed things up. We had tried implementing all of the package in C++ but some of the algorithms used were difficult and error-prone to implement, so some of the code remained implemented in R. However, the portion still in R is not the bottleneck in the performance of the package.

The key points in the parallelization of the algorithm in OpenMP are the call to the C++ code:

```
1 probs = ompGetSumProbs(ndicePerRoll , nsidesPerDie)$probabilities
2 sumOfProbs <- .Call("ompApplyGetEventProb", combMatrix, nrolls , probs[,2],
3 outcomeProb = sumOfProbs
```

and the C++ implementation of the calculation of each row probability:

```
1 #pragma omp parallel for schedule(static,1)
2 for(int i = 0; i < nrow; i++)
3 {
4     sum[omp_get_thread_num()] += RowGetEventProb(ncol , c_nrolls ,
5         c_ndicePerRoll , c_nsidesPerDie , hv , passprob , i);
6 }
```

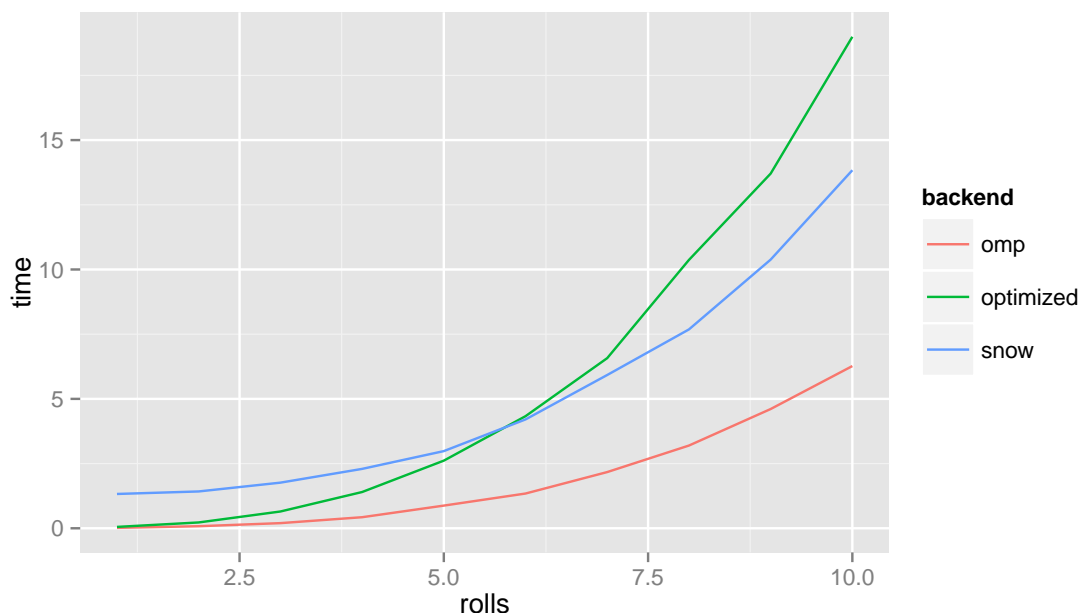


Figure 7: Timing difference for code parallelized using *OpenMP*

6 Parallelizing Using Thrust

The thrust [4] version of the algorithm divides the work by row as in the snow case. It uses a C++ implementation of the probability calculation for each row as in the OpenMP version and performs significantly faster, especially since it's written in a lower level language compared to R. The following is where the transform and reduce occurs:

```
1 thrust::device_vector<int> dm(size);
2 thrust::copy(hv.begin(), hv.end(), dm.begin());
3
4 thrust::host_vector<float> hprobs(probsM.size());
5 thrust::copy(probsM.begin(), probsM.end(), hprobs.begin());
6
7 thrust::device_vector<float> dprobs(size);
8 thrust::copy(hprobs.begin(), hprobs.end(), dprobs.begin());
9
10 thrust::device_vector<int> seq(nrow);
11 thrust::sequence(seq.begin(), seq.end());
12 thrust::device_vector<float> dv(nrow);
13 thrust::transform(seq.begin(), seq.end(),
14                 dv.begin(),
15                 RowGetEventProb(ncol, c_nrolls,
16                               c_ndicePerRoll, c_nsidesPerDie,
17                               dm.begin(), dprobs.begin()));
18 thrust::host_vector<float> host_prob(nrow);
19 thrust::copy(dv.begin(), dv.end(), dprobs.begin());
20
21 return wrap(thrust::reduce(dv.begin(), dv.end()));
```

And the functor has a similar implementation for calculating the probability for each individual row:

```
1 struct RowGetEventProb
2 {
3     int *combMat;
4     float *probs;
5     int ncol;
6     int c_ndicePerRoll;
7     int c_nsidesPerDie;
8     int c_nrolls;
9
10     RowGetEventProb(int ncol, int c_nrolls, int c_ndicePerRoll, int c_n
11 : ncol(ncol), c_nrolls(c_nrolls), c_ndicePerRoll(c_ndicePerRoll),
12 c_nsidesPerDie(c_nsidesPerDie)
```



```

13      {
14          combMat = thrust::raw_pointer_cast(&it_combMat[0]);
15          probs = thrust::raw_pointer_cast(&it_probs[0]);
16      }
17
18      __device__ float operator()(const int& matrix_i)
19      {
20          // See Appendix for code listing
21      }
22  };

```

The performance comparison can be seen in Figure 8 with parallelism of four cores:

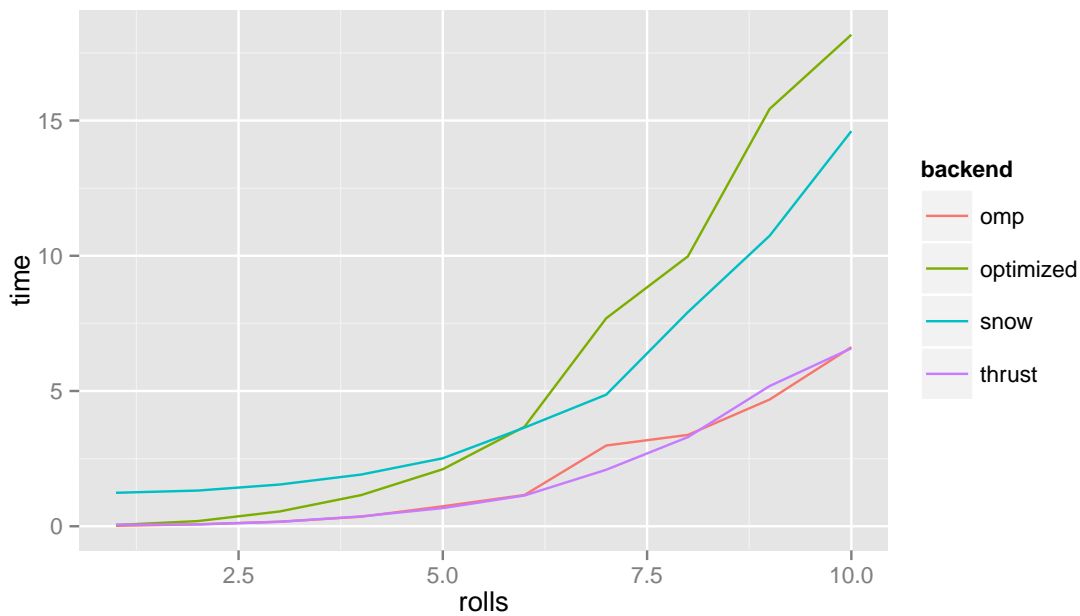


Figure 8: Timing difference for code parallelized using *thrust*

Clearly the *thrust* implementation did not have much of an effect over the direct OpenMP implementation. This shows that the abstraction of *thrust* does have some overhead, but it was not significant in this case. In other words it provided pretty good parallelism despite being at a higher level.

7 Results

After applying *snow*, OpenMP and *thrust* to the *dice* package we saw strong improvements over the non-parallel dice package. Snow provided the least increase in perfor-

mance with a average close to the optimized version of *dice.R*. Due to snow’s communication between nodes in clusters some performance was lost thus taking up almost as much time as the actual function. Many of the calculations preformed by *getEventList* and *getSumProbs* are short and non-intensive thus providing snow with less time to achieve any meaning full speed up before having to communicate with nodes again and handling data that must be returned. Thrust with an OpenMP backend along with OpenMP provided about a 75% decrease in computation time. Thrust was able to remove the bottle neck within *dice.R* and *dice.cpp* by calculation probabilities based on different rows of the *combMatrix*. This was achieved by creating a functor *RowGetEventProb* which used a device kernel to perform probability calculations on the GPU at a very fast rate since the machines being used to run the code have GPUs with 192 CUDA cores. The OpenMP version also took advantage of the fact that the recursion on the *combMatrix* was the main bottleneck for the original package. Thus by chunking the rows and passing them to multiple cores we were able to achieve a strong performance increase and allowing each CPU to work on a contiguous block of *combMatrix* thus each CPU is working on non-shared data and avoiding false sharing which happens when data shared by multiple CPUs is modified frequently. Also, OpenMP takes a more relaxed consistency approach, forcing updates at all synchronization points [7], which can provide for an increase in performance compared to the other models. After applying snow, OpenMP and thrust to the dice package we saw strong improvements over the non-parallel dice package.

A Code Listings

A.1 *snow*

This contains the *snow* implementation of the dice package. The key point is the location that uses *parRapply* on the cluster to parallelize the computation. Also note the caching the probabilities sums and passing that data rather than recomputing it for each row.

```

1
2 library(gtools)
3 library(parallel)
4 library(dice)
5 library(ggplot2)
6
7 # @@@@
8
9
10 # These helper functions check parameter integrity
11
12 .checkIntParam = function(param, paramName, positive)

```

```

13 {
14   if ((!missing(positive) && param < (if (positive) 1 else 0)) ||
15       (param != floor(param)) ||
16       (length(param) > 1))
17   {
18     if (missing(positive))
19     {
20       paste("\n*", paramName, "must contain a single integer instead of", param)
21     }
22     else if (positive)
23     {
24       paste("\n*", paramName, "must contain a single positive integer instead of", param)
25     }
26     else
27     {
28       paste("\n*", paramName, "must contain a single non-negative integer instead of", param)
29     }
30   }
31 }
32
33
34 .checkLogicalParam = function(param, paramName)
35 {
36   if (length(param) > 1 ||
37       !is.logical(param))
38   {
39     paste("\n*", paramName, "must contain a single logical value (i.e., TRUE or FALSE)")
40   }
41 }
42
43
44 # This helper function returns the probabilities of each element of eventList
45
46 .snowGetEventListProbs = function(ndicePerRoll, nsidesPerDie, eventList, probabilities)
47 {
48
49   # On the assumption that eventList has length nrolls (which is safe since
50   # private helper function), we calculate the probability of getting an ac
51   # outcome (a "success") on each of the rolls by iterating through the vec
52   # successes for that roll and adding the corresponding probability to our
53
54   eventListProbs = c()

```



```

97   errorVector = append(errorVector, .checkIntParam(ndicePerRoll, "ndicePerR
98   errorVector = append(errorVector, .checkIntParam(nsidesPerDie, "nsidesPerL
99
100  if (length(eventList) > nrolls)
101  {
102    errorVector = append(errorVector, "\n* The length of eventList must not
103  }
104  if (orderMatters & length(eventList) != nrolls)
105  {
106    errorVector = append(errorVector, "\n* If orderMatters is passed as TRU
107  }
108  if (!all(sapply(eventList, is.numeric)))
109  {
110    errorVector = append(errorVector, "\n* All elements of eventList must be
111  }
112  if (!all(as.logical(sapply(sapply(eventList, function(x) x == floor(x)), n
113  {
114    errorVector = append(errorVector, "\n* All numbers in each element of e
115  }
116  if (min(sapply(eventList, min)) < ndicePerRoll ||
117      max(sapply(eventList, max)) > (ndicePerRoll * nsidesPerDie))
118  {
119    errorVector = append(errorVector, "\n* All numbers in each element of e
120  }
121  errorVector = append(errorVector, .checkLogicalParam(orderMatters, "order
122
123  if (length(errorVector) > 0)
124  {
125    stop(errorVector)
126  }
127
128  eventList = lapply(eventList, unique)
129
130  # If eventList doesn't have an element for each roll, we add elements unt
131  # after this point, each element of eventList will constrain one roll (bu
132  # those constraints may be simply {min:max} for that roll—i.e., trivial
133
134  if (length(eventList) < nrolls)
135  {
136    eventList = lapply(c(eventList, rep(0, nrolls - length(eventList))), fu
137  }
138

```

```

139   if (orderMatters)
140   {
141     if(is.null(probs)){
142       probs = snowGetSumProbs(ndicePerRoll, nsidesPerDie)$probabilities
143     }
144     outcomeProb = prod(.snowGetEventListProbs(ndicePerRoll, nsidesPerDie, e
145   }
146   else # i.e., if (!orderMatters)
147   {
148     if(top)
149       marker <- checkpoint(marker, "Set_up")
150     # We only calculate probabilities if each element of eventList is a leng
151     # (i.e., a single number), e.g., {2, 3, 2}; if any element is longer tha
152     # {2, {3, 4}, 2}, we call ourselves recursively on each list we can con
153     # length-1 vectors (e.g., in the example above we'd call ourselves on {
154     # {2, 4, 2}); then we sum the resulting probabilities (which, since ord
155     # FALSE, account for all permutations of each of {2, 3, 2} and {2, 4, 2
156     # at our probability for the original list of {2, {3, 4}, 2}
157
158     listElemLengths = sapply(eventList, length)
159     maxListElemLength = max(listElemLengths)
160     if(top)
161       marker <- checkpoint(marker, "MaxList")
162     if (maxListElemLength > 1)
163     {
164       # Here we populate combMatrix with the elements of eventList to produ
165       # matrix each row of which is a selection of one element from each ele
166       # eventList; e.g., given the eventList {{1, 2}, {1, 2, 4}, 2}, we'd p
167       # a 6 x 3 matrix with rows {1, 1, 2}, {1, 2, 2}, {1, 4, 2}, {2, 1, 2}
168       # and {2, 4, 2}
169
170       combMatrix = matrix(nrow = prod(listElemLengths), ncol = nrolls)
171       if (nrolls > 1)
172       {
173         for (i in 1:(nrolls-1))
174         {
175           combMatrix[,i] = rep(eventList[[i]], each = prod(listElemLengths[
176         }
177       }
178       combMatrix[,nrolls] = rep(eventList[[nrolls]])
179
180     if(top)

```

```

181     marker <- checkpoint(marker, "Comb")
182
183     # Next we eliminate all rows that are permutations of other rows (oth
184     # would over-count in the calculations that follow)
185
186     if (nrolls > 1)
187     {
188         combMatrix = unique(t(apply(combMatrix,1,sort)))
189     }
190     else
191     {
192         combMatrix = unique(combMatrix)
193     }
194
195     if(top)
196         marker <- checkpoint(marker, "Unique")
197
198     # Now we make a recursive call for each row of combMatrix and sum the
199     # probabilities to arrive at our probability for the original eventLi
200     # Create cluster
201     cls <- makePSOCKcluster(rep("localhost", parallel))
202     probs = snowGetSumProbs(ndicePerRoll, nsidesPerDie)$probabilities
203     # Export functions
204     clusterExport(cls, c("snowGetEventProb", ".snowGetEventListProbs", ".c
205                     envir=environment())
206     # Apply in parallel
207     result <- parRapply(cls, combMatrix,
208                         function(x) {
209                             snowGetEventProb(nrolls,
210                                                 ndicePerRoll,
211                                                 nsidesPerDie,
212                                                 as.list(x),
213                                                 orderMatters,
214                                                 F,
215                                                 probs)
216                         })
217     # Reduce into sum
218     sumOfProbs = sum(result)
219     outcomeProb = sumOfProbs
220     stopCluster(cls)
221
222     if(top)

```



```

265   errorVector = append(errorVector , .checkLogicalParam(perDieMinOfOne, "perDieMinOfOne")
266
267   if (length(errorVector) > 0)
268   {
269       stop(errorVector)
270   }
271
272   numOutcomes = nsidesPerDie^ndicePerRoll
273   numDiceToDrop = ndicePerRoll - nkept
274   currNumArrangements = 0
275
276   sumModifier = sumModifier + (perDieModifier * nkept)
277
278   currentSum = 0
279
280   vectorOfSums = as.integer((nkept + sumModifier) :
281                               ((nsidesPerDie * nkept) + sumModifier))
282
283   numPossibleSums = length(vectorOfSums)
284
285   # sumTallyMatrix is used to track the number of times we see every possible sum
286   # which we will use to produce the probabilities of every sum (e.g., for
287   # see 10 as a sum 27 times, so the probability of a sum of 10 is 27/216 =
288   # the 5d6 drop 2 case we see 13 as a sum 1055 times, so the probability of
289   # is 1055/7776 = .1356739).
290
291   sumTallyMatrix = matrix(data = c(vectorOfSums,
292                                   as.integer(rep.int(0, numPossibleSums)),
293                                   as.integer(rep.int(0, numPossibleSums))),
294                       nrow = numPossibleSums,
295                       ncol = 3,
296                       dimnames = list(NULL,
297                                       c("Sum",
298                                           "Probability",
299                                           "Ways_to_Roll"))))
300
301   # boundaryVal is the most extreme die-roll value that will be kept (i.e.,
302   # value: e.g., for 5d6 drop lowest two, if our sorted die rolls are {3 4 4
303   # We'll call all dice with this value the "b" dice (because they're on the
304
305   for (boundaryVal in 1 : nsidesPerDie)
306   {

```

307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346

```
# numOs is the number of dice whose values are outside of boundaryVal (
# two, if we roll {3 4 4 5 6}, boundaryVal is 4, so numOs is 1).
We'll call these dice the "o"
# dice (because they're [o]utside our boundary).
# NOTE: We have an embedded if clause in our for-loop declaration becau
# and boundaryVal is 1 or we're dropping highest and boundaryVal is nsid
# any dice whose values are outside of boundaryVal, and hence numOs can
The following
# loop syntax might look suspicious, but we *do* want to iterate once in
# as in the case where numDiceToDrop is 0 (and in all three such cases,
for (numOs in 0 : (if ((dropLowest && boundaryVal == 1) ||
                      (!dropLowest && boundaryVal == nsidesPerDie))
{
  # numBsKept is the number of b's that will be kept (e.g., for 5d6 drop
  # {3 4 4 5 6}, numBsKept is 1, because one of the two 4's will be kep
  # Now, since we're discarding numDiceToDrop dice (including the numOs
  # (numDiceToDrop - numOs) of the b's and keep numBsKept of them, and
  # b's is (numBsKept + numDiceToDrop - numOs). NOTE: Hence, the numbe
  # exceed boundaryVal is (nkept - numBsKept). We will call these high
  # they're "inside" our boundary).

  for (numBsKept in 1 : nkept)
  {
    # By this part of the function, we've specified a class of outcomes
    # (boundaryVal, numOs, numBsKept) values—i.e., every outcome in th
    # following properties:
    # 1). the die-roll boundary value boundaryVal;
    # 2). numOs "o" dice, whose values are outside boundaryVal and will
    # 3). numBsKept "b" dice that will be kept. Furthermore, each such
    # 4). (numBsKept + numDiceToDrop - numOs) "b" dice in total, and
    # 5). (nkept - numBsKept) "i" dice, whose values are inside boundary

    numBs = (numBsKept + numDiceToDrop - numOs)
    numIs = (nkept - numBsKept)

    paste("\n\nIn this class, boundaryVal is ", boundaryVal, ", numOs is
    # Now, we're interested in sums for the various outcomes in this cl
```

```

347      # don't depend upon the order in which the various values appear in
348      # rolls; i.e., multiple outcomes in this class will have the same v
349      # the b's, and the i's appear at different places in the sequence o
To account
350      # for this, we need to multiply each distinct outcome by the number
351      # that are identical to it except for the order in which the o's, b
352      # (NOTE: the orders within these groups are accounted for below:
353      # is accounted for immediately below, and we account for the order
354      # section of the code in which we enumerate the i's). For now, we
355      # which to multiply each sum we find; this term is a result of the
356      # combinatoric interpretation as the number of ways to put n distin
357      # case, our die rolls) into 3 bins of size numOs, (numBsKept + numL
358      # and (nkept - numBsKept), corresponding to the number of o's, b's,
359
360      numArrangementsOfDice = (factorial(ndicePerRoll) /
361                               (factorial(numOs) * factorial(numBs) * f
362
363      # [NOTE: The formula above could overflow if ndicePerRoll gets large
364      # consider using lfactorial()—but I think the function would keel
365
366      # Because we support dropping lowest or highest values, we define c
367      # to allow us to operate over appropriate ranges for the rest of th
368
369      innerBoundary = if (dropLowest) nsidesPerDie else 1
370      outerBoundary = if (dropLowest) 1 else nsidesPerDie
371      rangeOfOVals = abs(boundaryVal - outerBoundary)
372      rangeOfIVals = abs(boundaryVal - innerBoundary)
373      possibleIValsVec = if (dropLowest) ((boundaryVal+1) : nsidesPerDie)
374
375      # Next: The value of boundaryVal is fixed for this loop, but there
376      # that outcomes in this class might have; because we don't care abo
377      # to increase our multiplicity term to account for the outcomes tha
378      # to this iteration's distinct outcome but for the values (and order
379
380      numArrangementsOfDice = numArrangementsOfDice * rangeOfOVals^numOs
381
382      # Now that we've accounted for sorting the values into three bins an
383      # "o" values that are immaterial to our calculations, we can treat o
384      # sorted into groups and can focus our attention on the numBsKept b
385      # i's. The numBsKept b's will contribute a known amount to our sum
386      # this class (viz., numBsKept * boundaryVal); but the i's will cont
387      # depending on their values. So now we turn to determining the pos

```

```

388      # for this class by enumerating the possible values for the i's.
We will work as follows:
389      # rangeOfIVals is the distance between the smallest and largest pos
390      # class of outcomes, and we use it to determine the number of distin
391      # class, which is given by rangeOfIVals^numIs. We create an outcome
392      # rows as there are distinct outcomes for this class and nkept colum
393      # in a row corresponds to a die-roll value, and the sum of the row
394      # sum for that distinct outcome. We populate outcomeMatrix with a
395      # value for the i's in this class (and hence all distinct outcomes
We then
396      # calculate the number of permutations of each distinct outcome (e.g.
397      # the outcome {1, 1, 2} has three permutations) and use this inform
398      # probability of every possible outcome in this class.
399
400      if (numBsKept == nkept)
401      {
402          currentSum = (numBsKept * boundaryVal) + sumModifier
403          # We adjust row index by (nkept - 1) so that, e.g., a 3d6 tally s
404          sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = sumTa
405      }
406      else
407      {
408          outcomeMatrix = matrix(nrow = choose((rangeOfIVals + numIs - 1), n
409
410          if (dim(outcomeMatrix)[1] > 0)
411          {
412
413              outcomeMatrix[,1 : numBsKept] = boundaryVal
414
415              hCombs = combinations(n = rangeOfIVals,
416                                  r = numIs,
417                                  v = possibleIValsVec,
418                                  repeats.allowed = TRUE)
419              hPermCounts = apply(hCombs, 1, function(x) factorial(numIs)/prod
420
421              outcomeMatrix[, (numBsKept+1) : nkept] = hCombs
422
423              for (rowNum in 1 : nrow(outcomeMatrix))
424              {
425                  currentSum = sum(outcomeMatrix[rowNum,]) + sumModifier
426                  currNumArrangements = numArrangementsOfDice * hPermCounts[rowN
427                  sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = su

```



```

4
5
6 # These helper functions check parameter integrity
7
8 .checkIntParam = function(param, paramName, positive)
9 {
10   if ((!missing(positive) && param < (if (positive) 1 else 0)) ||
11       (param != floor(param)) ||
12       (length(param) > 1))
13   {
14     if (missing(positive))
15     {
16       paste("\\n*", paramName, "must contain a single integer instead of", p
17     }
18     else if (positive)
19     {
20       paste("\\n*", paramName, "must contain a single positive integer instead
21     }
22     else
23     {
24       paste("\\n*", paramName, "must contain a single non-negative integer in
25     }
26   }
27 }
28
29
30 .checkLogicalParam = function(param, paramName)
31 {
32   if (length(param) > 1 ||
33       !is.logical(param))
34   {
35     paste("\\n*", paramName, " must contain a single logical value (i.e., \\T
36   }
37 }
38
39 # This helper function returns the probabilities of each element of eventLi
40
41 .optGetEventListProbs = function(ndicePerRoll, nsidesPerDie, eventList, prob
42 {
43
44   # On the assumption that eventList has length nrolls (which is safe since
45   # private helper function), we calculate the probability of getting an ac

```

```

46 # outcome (a "success") on each of the rolls by iterating through the vec
47 # successes for that roll and adding the corresponding probability to our
48
49 #cat("E", eventList)
50 #print(probs)
51 eventListProbs = c()
52 for (i in 1:length(eventList))
53 {
54     successesForThisRoll = sort(eventList[[i]])
55     successProbForThisRoll = 0
56     for (j in 1:length(successesForThisRoll))
57     {
58 #         cat("D", (successesForThisRoll[j] - (ndicePerRoll - 1)))
59         #cat("P", probs[(successesForThisRoll[j] - (ndicePerRoll - 1)),2])
60 #         print(ndicePerRoll)
61         #cat("P", ndicePerRoll)
62         #print("P")
63         successProbForThisRoll = successProbForThisRoll + probs[(successesFor
64     }
65     eventListProbs[i] = successProbForThisRoll
66 }
67 #cat("H", eventListProbs[i], "\n")
68 eventListProbs
69 }
70
71 optGetEventProb = function(nrolls ,
72                             ndicePerRoll ,
73                             nsidesPerDie ,
74                             eventList ,
75                             orderMatters=FALSE,
76                             top=F,
77                             probs=NULL)
78 {
79     marker <- Sys.time()
80
81     errorVector = character()
82     errorVector = append(errorVector , .checkIntParam(nrolls , "nrolls", positiv
83     errorVector = append(errorVector , .checkIntParam(ndicePerRoll , "ndicePerR
84     errorVector = append(errorVector , .checkIntParam(nsidesPerDie , "nsidesPerL
85
86     if (length(eventList) > nrolls)
87     {

```

```

88     errorVector = append(errorVector , "\n* The length of eventList must not
89 }
90 if (orderMatters & length(eventList) != nrolls)
91 {
92     errorVector = append(errorVector , "\n* If orderMatters is passed as TRUE
93 }
94 if (!all(sapply(eventList , is.numeric)))
95 {
96     errorVector = append(errorVector , "\n* All elements of eventList must be
97 }
98 if (!all(as.logical(sapply(sapply(eventList , function(x) x == floor(x)) , n
99 {
100     errorVector = append(errorVector , "\n* All numbers in each element of e
101 }
102 if (min(sapply(eventList , min)) < ndicePerRoll ||
103     max(sapply(eventList , max)) > (ndicePerRoll * nsidesPerDie))
104 {
105     errorVector = append(errorVector , "\n* All numbers in each element of e
106 }
107 errorVector = append(errorVector , .checkLogicalParam(orderMatters , "order
108
109 if (length(errorVector) > 0)
110 {
111     stop(errorVector)
112 }
113
114 if(is.null(probs)){
115     probs = optGetSumProbs(ndicePerRoll , nsidesPerDie)$probabilities
116     #print(probs)
117 }
118
119 eventList = lapply(eventList , unique)
120
121 # If eventList doesn't have an element for each roll , we add elements until
122 # after this point , each element of eventList will constrain one roll (but
123 # those constraints may be simply {min:max} for that roll—i.e., trivial
124
125 if (length(eventList) < nrolls)
126 {
127     eventList = lapply(c(eventList , rep(0 , nrolls - length(eventList))), fu
128 }
129

```



```

130  if (orderMatters)
131  {
132      outcomeProb = prod(.optGetEventListProbs(ndicePerRoll, nsidesPerDie, ev
133  }
134  else # i.e., if (!orderMatters)
135  {
136      if(top)
137          marker <- checkpoint(marker, "Set_up")
138      # We only calculate probabilities if each element of eventList is a leng
139      # (i.e., a single number), e.g., {2, 3, 2}; if any element is longer tha
140      # {2, {3, 4}, 2}, we call ourselves recursively on each list we can con
141      # length-1 vectors (e.g., in the example above we'd call ourselves on {
142      # {2, 4, 2}); then we sum the resulting probabilities (which, since ord
143      # FALSE, account for all permutations of each of {2, 3, 2} and {2, 4, 2
144      # at our probability for the original list of {2, {3, 4}, 2}
145
146      listElemLengths = sapply(eventList, length)
147      maxListElemLength = max(listElemLengths)
148      if(top)
149          marker <- checkpoint(marker, "MaxList")
150      if (maxListElemLength > 1)
151      {
152          # Here we populate combMatrix with the elements of eventList to produ
153          # matrix each row of which is a selection of one element from each ele
154          # eventList; e.g., given the eventList {{1, 2}, {1, 2, 4}, 2}, we'd p
155          # a 6 x 3 matrix with rows {1, 1, 2}, {1, 2, 2}, {1, 4, 2}, {2, 1, 2}
156          # and {2, 4, 2}
157
158          combMatrix = matrix(nrow = prod(listElemLengths), ncol = nrolls)
159          if (nrolls > 1)
160          {
161              for (i in 1:(nrolls-1))
162              {
163                  combMatrix[,i] = rep(eventList[[i]], each = prod(listElemLengths[
164              }
165          }
166          combMatrix[,nrolls] = rep(eventList[[nrolls]])
167
168          if(top)
169              marker <- checkpoint(marker, "Comb")
170
171          # Next we eliminate all rows that are permutations of other rows (oth

```

```

172     # would over-count in the calculations that follow)
173
174     if (nrolls > 1)
175     {
176         combMatrix = unique(t(apply(combMatrix,1,sort)))
177     }
178     else
179     {
180         combMatrix = unique(combMatrix)
181     }
182
183     if(top)
184         marker <- checkpoint(marker, "Unique")
185
186     # Now we make a recursive call for each row of combMatrix and sum the
187     # probabilities to arrive at our probability for the original eventLi
188     sumOfProbs = sum(apply(combMatrix, 1,
189                         function(x) optGetEventProb(nrolls,
190                                                         ndicePerRoll,
191                                                         nsidesPerDie,
192                                                         as.list(x),
193                                                         orderMatters,
194                                                         F,
195                                                         probs)))
196     outcomeProb = sumOfProbs
197
198     if(top)
199         marker <- checkpoint(marker, "Sum")
200
201     #print("Recursed")
202 }
203 else
204 {
205     # If each element of eventList is a length-1 vector, we can convert e
206     # itself to a vector; then we calculate the probability of getting th
207     # set of outcomes specified by eventList in any order (reflecting the
208     # orderMatters was passed in as FALSE)
209
210     eventListAsVector = sapply(eventList, max)
211     eventListProb = prod(.optGetEventListProbs(ndicePerRoll, nsidesPerDie
212     outcomeProb = eventListProb * factorial(nrolls) / prod(factorial(table
213 }

```



```

256 currentSum = 0
257
258 vectorOfSums = as.integer((nkept + sumModifier) :
259                          ((nsidesPerDie * nkept) + sumModifier))
260
261 numPossibleSums = length(vectorOfSums)
262
263 # sumTallyMatrix is used to track the number of times we see every possible sum
264 # which we will use to produce the probabilities of every sum (e.g., for
265 # see 10 as a sum 27 times, so the probability of a sum of 10 is 27/216 =
266 # the 5d6 drop 2 case we see 13 as a sum 1055 times, so the probability of
267 # is 1055/7776 = .1356739).
268
269 sumTallyMatrix = matrix(data = c(vectorOfSums,
270                                as.integer(rep.int(0, numPossibleSums)),
271                                as.integer(rep.int(0, numPossibleSums))),
272                    nrow = numPossibleSums,
273                    ncol = 3,
274                    dimnames = list(NULL,
275                                    c("Sum",
276                                        "Probability",
277                                        "Ways_to_Roll")))
278
279 # boundaryVal is the most extreme die-roll value that will be kept (i.e.,
280 # value: e.g., for 5d6 drop lowest two, if our sorted die rolls are {3 4 4 5 6},
281 # We'll call all dice with this value the "b" dice (because they're on the boundary).
282
283 for (boundaryVal in 1 : nsidesPerDie)
284 {
285
286     # numOs is the number of dice whose values are outside of boundaryVal (
287     # two, if we roll {3 4 4 5 6}, boundaryVal is 4, so numOs is 1).
    We'll call these dice the "o"
288     # dice (because they're [o]utside our boundary).
289     # NOTE: We have an embedded if clause in our for-loop declaration because
290     # and boundaryVal is 1 or we're dropping highest and boundaryVal is nsidesPerDie
291     # any dice whose values are outside of boundaryVal, and hence numOs can be 0.
    The following
292     # loop syntax might look suspicious, but we *do* want to iterate once in
293     # as in the case where numDiceToDrop is 0 (and in all three such cases,
294
295     for (numOs in 0 : (if ((dropLowest && boundaryVal == 1) ||

```

```

296                                     (!dropLowest && boundaryVal == nsidesPerDie)) (
297     {
298
299         # numBsKept is the number of b's that will be kept (e.g., for 5d6 drop
300         # {3 4 4 5 6}, numBsKept is 1, because one of the two 4's will be kept
301         # Now, since we're discarding numDiceToDrop dice (including the numOs
302         # (numDiceToDrop - numOs) of the b's and keep numBsKept of them, and
303         # b's is (numBsKept + numDiceToDrop - numOs). NOTE: Hence, the number
304         # exceed boundaryVal is (nkept - numBsKept). We will call these high
305         # they're "inside" our boundary).
306
307         for (numBsKept in 1 : nkept)
308         {
309
310             # By this part of the function, we've specified a class of outcomes
311             # (boundaryVal, numOs, numBsKept) values—i.e., every outcome in the
312             # following properties:
313             # 1). the die-roll boundary value boundaryVal;
314             # 2). numOs "o" dice, whose values are outside boundaryVal and will
315             # 3). numBsKept "b" dice that will be kept. Furthermore, each such
316             # 4). (numBsKept + numDiceToDrop - numOs) "b" dice in total, and
317             # 5). (nkept - numBsKept) "i" dice, whose values are inside boundaryVal.
318
319             numBs = (numBsKept + numDiceToDrop - numOs)
320             numIs = (nkept - numBsKept)
321
322             paste("\n\nIn this class, boundaryVal is ", boundaryVal, ", numOs is ",
323
324             # Now, we're interested in sums for the various outcomes in this class
325             # don't depend upon the order in which the various values appear in
326             # rolls; i.e., multiple outcomes in this class will have the same v
327             # the b's, and the i's appear at different places in the sequence of
328
329             To account
330             # for this, we need to multiply each distinct outcome by the number
331             # that are identical to it except for the order in which the o's, b
332             # (NOTE: the orders *within* these groups are accounted for below:
333             # is accounted for immediately below, and we account for the order
334             # section of the code in which we enumerate the i's). For now, we
335             # which to multiply each sum we find; this term is a result of the
336             # combinatoric interpretation as the number of ways to put n distin
337             # case, our die rolls) into 3 bins of size numOs, (numBsKept + numD
338             # and (nkept - numBsKept), corresponding to the number of o's, b's,

```

337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376

```
numArrangementsOfDice = (factorial(ndicePerRoll) /  
                          (factorial(numOs) * factorial(numBs) * f  
  
# [NOTE: The formula above could overflow if ndicePerRoll gets large  
# consider using lfactorial()—but I think the function would keel  
  
# Because we support dropping lowest or highest values, we define c  
# to allow us to operate over appropriate ranges for the rest of th  
  
innerBoundary = if (dropLowest) nsidesPerDie else 1  
outerBoundary = if (dropLowest) 1 else nsidesPerDie  
rangeOfOVals = abs(boundaryVal - outerBoundary)  
rangeOfIVals = abs(boundaryVal - innerBoundary)  
possibleIValsVec = if (dropLowest) ((boundaryVal+1) : nsidesPerDie)  
  
# Next: The value of boundaryVal is fixed for this loop, but there  
# that outcomes in this class might have; because we don't care abo  
# to increase our multiplicity term to account for the outcomes tha  
# to this iteration's distinct outcome but for the values (and orde  
  
numArrangementsOfDice = numArrangementsOfDice * rangeOfOVals^numOs  
  
# Now that we've accounted for sorting the values into three bins an  
# "o" values that are immaterial to our calculations, we can treat  
# sorted into groups and can focus our attention on the numBsKept b  
# i's. The numBsKept b's will contribute a known amount to our sum  
# this class (viz., numBsKept * boundaryVal); but the i's will cont  
# depending on their values. So now we turn to determining the pos  
# for this class by enumerating the possible values for the i's.  
  
We will work as follows:  
# rangeOfIVals is the distance between the smallest and largest pos  
# class of outcomes, and we use it to determine the number of distin  
# class, which is given by rangeOfIVals^numIs. We create an outcom  
# rows as there are distinct outcomes for this class and nkept column  
# in a row corresponds to a die-roll value, and the sum of the row  
# sum for that distinct outcome. We populate outcomeMatrix with a  
# value for the i's in this class (and hence all distinct outcomes  
  
We then  
# calculate the number of permutations of each distinct outcome (e.g.  
# the outcome {1, 1, 2} has three permutations) and use this inform  
# probability of every possible outcome in this class.
```

```

377
378   if (numBsKept == nkept)
379   {
380       currentSum = (numBsKept * boundaryVal) + sumModifier
381       # We adjust row index by (nkept - 1) so that, e.g., a 3d6 tally s
382       sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = sumTa
383   }
384   else
385   {
386       outcomeMatrix = matrix(nrow = choose((rangeOfIVals + numIs - 1), n
387
388       if (dim(outcomeMatrix)[1] > 0)
389       {
390
391           outcomeMatrix[,1 : numBsKept] = boundaryVal
392
393           hCombs = combinations(n = rangeOfIVals,
394                                r = numIs,
395                                v = possibleIValsVec,
396                                repeats.allowed = TRUE)
397           hPermCounts = apply(hCombs, 1, function(x) factorial(numIs)/prod
398
399           outcomeMatrix[, (numBsKept+1) : nkept] = hCombs
400
401           for (rowNum in 1 : nrow(outcomeMatrix))
402           {
403               currentSum = sum(outcomeMatrix[rowNum,]) + sumModifier
404               currNumArrangements = numArrangementsOfDice * hPermCounts[rowN
405               sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = st
406           }
407       }
408   }
409 }
410 }
411 }
412
413 if (perDieMinOfOne)
414 {
415     if (sumTallyMatrix[numPossibleSums,1] <= nkept)
416     {
417         sumTallyMatrix = matrix(data = c(nkept, numOutcomes, numOutcomes),
418                                nrow = 1,

```



```

59
60   if (length(eventList) > nrolls)
61   {
62     errorVector = append(errorVector , "\n* The length of eventList must not
63   }
64   if (orderMatters & length(eventList) != nrolls)
65   {
66     errorVector = append(errorVector , "\n* If orderMatters is passed as TRUE
67   }
68   if (!all(sapply(eventList , is.numeric)))
69   {
70     errorVector = append(errorVector , "\n* All elements of eventList must be
71   }
72   if (!all(as.logical(sapply(sapply(eventList , function(x) x == floor(x)) , TRUE)))
73   {
74     errorVector = append(errorVector , "\n* All numbers in each element of eventList
75   }
76   if (min(sapply(eventList , min)) < ndicePerRoll ||
77       max(sapply(eventList , max)) > (ndicePerRoll * nsidesPerDie))
78   {
79     errorVector = append(errorVector , "\n* All numbers in each element of eventList
80   }
81   errorVector = append(errorVector , .checkLogicalParam(orderMatters , "orderMatters"))
82
83   if (length(errorVector) > 0)
84   {
85     stop(errorVector)
86   }
87
88   eventList = lapply(eventList , unique)
89
90   # If eventList doesn't have an element for each roll , we add elements until
91   # after this point , each element of eventList will constrain one roll (but
92   # those constraints may be simply {min:max} for that roll—i.e., trivial
93
94   if (length(eventList) < nrolls)
95   {
96     eventList = lapply(c(eventList , rep(0 , nrolls - length(eventList))) , function(x) {
97   }
98
99   if (orderMatters)
100  {

```

```

101     if(is.null(probs)){
102         probs = snowGetSumProbs(ndicePerRoll, nsidesPerDie)$probabilities
103     }
104     outcomeProb = prod(.snowGetEventListProbs(ndicePerRoll, nsidesPerDie, e
105 }
106 else # i.e., if (!orderMatters)
107 {
108     # We only calculate probabilities if each element of eventList is a leng
109     # (i.e., a single number), e.g., {2, 3, 2}; if any element is longer tha
110     # {2, {3, 4}, 2}, we call ourselves recursively on each list we can con
111     # length-1 vectors (e.g., in the example above we'd call ourselves on {
112     # {2, 4, 2}); then we sum the resulting probabilities (which, since ord
113     # FALSE, account for all permutations of each of {2, 3, 2} and {2, 4, 2
114     # at our probability for the original list of {2, {3, 4}, 2}
115
116     listElemLengths = sapply(eventList, length)
117     maxListElemLength = max(listElemLengths)
118     if (maxListElemLength > 1)
119     {
120         # Here we populate combMatrix with the elements of eventList to produ
121         # matrix each row of which is a selection of one element from each ele
122         # eventList; e.g., given the eventList {{1, 2}, {1, 2, 4}, 2}, we'd p
123         # a 6 x 3 matrix with rows {1, 1, 2}, {1, 2, 2}, {1, 4, 2}, {2, 1, 2}
124         # and {2, 4, 2}
125
126         combMatrix = matrix(nrow = prod(listElemLengths), ncol = nrolls)
127         if (nrolls > 1)
128         {
129             for (i in 1:(nrolls-1))
130             {
131                 combMatrix[,i] = rep(eventList[[i]], each = prod(listElemLengths[
132             }
133         }
134         combMatrix[,nrolls] = rep(eventList[[nrolls]])
135
136         # Next we eliminate all rows that are permutations of other rows (oth
137         # would over-count in the calculations that follow)
138
139         if (nrolls > 1)
140         {
141             combMatrix = unique(t(apply(combMatrix,1,sort)))
142         }

```



```

185 # We begin with preliminary error-checking
186
187 errorVector = vector(mode = "character", length = 0)
188 errorVector = append(errorVector, .checkIntParam(ndicePerRoll, "ndicePerRoll"))
189 errorVector = append(errorVector, .checkIntParam(nsidesPerDie, "nsidesPerDie"))
190 errorVector = append(errorVector, .checkIntParam(nkept, "nkept", positive = TRUE))
191 if (nkept > ndicePerRoll)
192 {
193   errorVector = append(errorVector, "\n* nkept must not be greater than ndicePerRoll")
194 }
195 errorVector = append(errorVector, .checkIntParam(sumModifier, "sumModifier"))
196 errorVector = append(errorVector, .checkIntParam(perDieModifier, "perDieModifier"))
197 errorVector = append(errorVector, .checkLogicalParam(perDieMinOfOne, "perDieMinOfOne"))
198
199 if (length(errorVector) > 0)
200 {
201   stop(errorVector)
202 }
203
204 numOutcomes = nsidesPerDie^ndicePerRoll
205 numDiceToDrop = ndicePerRoll - nkept
206 currNumArrangements = 0
207
208 sumModifier = sumModifier + (perDieModifier * nkept)
209
210 currentSum = 0
211
212 vectorOfSums = as.integer((nkept + sumModifier) :
213                           ((nsidesPerDie * nkept) + sumModifier))
214
215 numPossibleSums = length(vectorOfSums)
216
217 # sumTallyMatrix is used to track the number of times we see every possible sum
218 # which we will use to produce the probabilities of every sum (e.g., for
219 # see 10 as a sum 27 times, so the probability of a sum of 10 is 27/216 =
220 # the 5d6 drop 2 case we see 13 as a sum 1055 times, so the probability of
221 # is 1055/7776 = .1356739).
222
223 sumTallyMatrix = matrix(data = c(vectorOfSums,
224                                   as.integer(rep.int(0, numPossibleSums))),
225                          as.integer(rep.int(0, numPossibleSums))),
226                          nrow = numPossibleSums,

```

```

227         ncol = 3,
228         dimnames = list(NULL,
229                           c("Sum",
230                             "Probability",
231                             "Ways_to_Roll")))
232
233     # boundaryVal is the most extreme die-roll value that will be kept (i.e.,
234     # value: e.g., for 5d6 drop lowest two, if our sorted die rolls are {3 4 4
235     # We'll call all dice with this value the "b" dice (because they're on the
236
237     for (boundaryVal in 1 : nsidesPerDie)
238     {
239
240         # numOs is the number of dice whose values are outside of boundaryVal (
241         # two, if we roll {3 4 4 5 6}, boundaryVal is 4, so numOs is 1).
242         We'll call these dice the "o"
243         # dice (because they're [o]utside our boundary).
244         # NOTE: We have an embedded if clause in our for-loop declaration because
245         # and boundaryVal is 1 or we're dropping highest and boundaryVal is nsidesPerDie
246         # any dice whose values are outside of boundaryVal, and hence numOs can be
247         The following
248         # loop syntax might look suspicious, but we *do* want to iterate once in
249         # as in the case where numDiceToDrop is 0 (and in all three such cases,
250
251         for (numOs in 0 : (if ((dropLowest && boundaryVal == 1) ||
252                               (!dropLowest && boundaryVal == nsidesPerDie)) 0
253                               nsidesPerDie - boundaryVal))
254         {
255
256             # numBsKept is the number of b's that will be kept (e.g., for 5d6 drop
257             # {3 4 4 5 6}, numBsKept is 1, because one of the two 4's will be kept
258             # Now, since we're discarding numDiceToDrop dice (including the numOs
259             # (numDiceToDrop - numOs) of the b's and keep numBsKept of them, and
260             # b's is (numBsKept + numDiceToDrop - numOs). NOTE: Hence, the number
261             # exceed boundaryVal is (nkept - numBsKept). We will call these high
262             # they're "inside" our boundary).
263
264             for (numBsKept in 1 : nkept)
265             {
266
267                 # By this part of the function, we've specified a class of outcomes
268                 # (boundaryVal, numOs, numBsKept) values—i.e., every outcome in the
269                 # following properties:

```

```

267 # 1). the die-roll boundary value boundaryVal;
268 # 2). numOs "o" dice, whose values are outside boundaryVal and will
269 # 3). numBsKept "b" dice that will be kept. Furthermore, each such
270 # 4). (numBsKept + numDiceToDrop - numOs) "b" dice in total, and
271 # 5). (nkept - numBsKept) "i" dice, whose values are inside boundary
272
273 numBs = (numBsKept + numDiceToDrop - numOs)
274 numIs = (nkept - numBsKept)
275
276 paste("\n\nIn this class, boundaryVal is ", boundaryVal, ", numOs is
277
278 # Now, we're interested in sums for the various outcomes in this class
279 # don't depend upon the order in which the various values appear in
280 # rolls; i.e., multiple outcomes in this class will have the same v
281 # the b's, and the i's appear at different places in the sequence of
To account
282 # for this, we need to multiply each distinct outcome by the number
283 # that are identical to it except for the order in which the o's, b
284 # (NOTE: the orders *within* these groups are accounted for below:
285 # is accounted for immediately below, and we account for the order
286 # section of the code in which we enumerate the i's). For now, we
287 # which to multiply each sum we find; this term is a result of the
288 # combinatoric interpretation as the number of ways to put n distin
289 # case, our die rolls) into 3 bins of size numOs, (numBsKept + numD
290 # and (nkept - numBsKept), corresponding to the number of o's, b's,
291
292 numArrangementsOfDice = (factorial(ndicePerRoll) /
293                          (factorial(numOs) * factorial(numBs) * f
294
295 # [NOTE: The formula above could overflow if ndicePerRoll gets large
296 # consider using lfactorial()—but I think the function would keel
297
298 # Because we support dropping lowest or highest values, we define c
299 # to allow us to operate over appropriate ranges for the rest of th
300
301 innerBoundary = if (dropLowest) nsidesPerDie else 1
302 outerBoundary = if (dropLowest) 1 else nsidesPerDie
303 rangeOfOVals = abs(boundaryVal - outerBoundary)
304 rangeOfIVals = abs(boundaryVal - innerBoundary)
305 possibleIValsVec = if (dropLowest) ((boundaryVal+1) : nsidesPerDie)
306
307 # Next: The value of boundaryVal is fixed for this loop, but there

```

```

308 # that outcomes in this class might have; because we don't care about
309 # to increase our multiplicity term to account for the outcomes that
310 # to this iteration's distinct outcome but for the values (and order)
311
312 numArrangementsOfDice = numArrangementsOfDice * rangeOfOVals^numOs
313
314 # Now that we've accounted for sorting the values into three bins and
315 # "o" values that are immaterial to our calculations, we can treat
316 # sorted into groups and can focus our attention on the numBsKept b's
317 # i's. The numBsKept b's will contribute a known amount to our sum
318 # this class (viz., numBsKept * boundaryVal); but the i's will contribute
319 # depending on their values. So now we turn to determining the position
320 # for this class by enumerating the possible values for the i's.
    We will work as follows:
321 # rangeOfIVals is the distance between the smallest and largest possible
322 # class of outcomes, and we use it to determine the number of distinct
323 # class, which is given by rangeOfIVals^numIs. We create an outcomeMatrix
324 # rows as there are distinct outcomes for this class and nkept columns
325 # in a row corresponds to a die-roll value, and the sum of the row
326 # sum for that distinct outcome. We populate outcomeMatrix with a
327 # value for the i's in this class (and hence all distinct outcomes)
    We then
328 # calculate the number of permutations of each distinct outcome (e.g.,
329 # the outcome {1, 1, 2} has three permutations) and use this information
330 # probability of every possible outcome in this class.
331
332 if (numBsKept == nkept)
333 {
334     currentSum = (numBsKept * boundaryVal) + sumModifier
335     # We adjust row index by (nkept - 1) so that, e.g., a 3d6 tally sum
336     sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = sumTallyMatrix[
337 }
338 else
339 {
340     outcomeMatrix = matrix(nrow = choose((rangeOfIVals + numIs - 1), numIs),
341
342     if (dim(outcomeMatrix)[1] > 0)
343     {
344
345         outcomeMatrix[,1 : numBsKept] = boundaryVal
346
347         hCombs = combinations(n = rangeOfIVals,

```



```

348         r = numIs,
349         v = possibleIValsVec,
350         repeats.allowed = TRUE)
351     hPermCounts = apply(hCombs, 1, function(x) factorial(numIs)/prod
352
353     outcomeMatrix[, (numBsKept+1) : nkept] = hCombs
354
355     for (rowNum in 1 : nrow(outcomeMatrix))
356     {
357         currentSum = sum(outcomeMatrix[rowNum,]) + sumModifier
358         currNumArrangements = numArrangementsOfDice * hPermCounts[rowNum]
359         sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] + currNumArrangements
360     }
361 }
362 }
363 }
364 }
365 }
366
367 if (perDieMinOfOne)
368 {
369     if (sumTallyMatrix[numPossibleSums,1] <= nkept)
370     {
371         sumTallyMatrix = matrix(data = c(nkept, numOutcomes, numOutcomes),
372                                nrow = 1,
373                                ncol = 3,
374                                dimnames = list(NULL,
375                                              c("__Sum__", "__Probability__")
376                                )
377     }
378     else
379     {
380         extraWaysToRollMin = sum(sumTallyMatrix[sumTallyMatrix[,1] < nkept, 2])
381         sumTallyMatrix = sumTallyMatrix[sumTallyMatrix[,1] >= nkept, ]
382         sumTallyMatrix[1,2] = sumTallyMatrix[1,2] + extraWaysToRollMin
383     }
384 }
385 sumTallyMatrix[,3] = sumTallyMatrix[,2]
386 sumTallyMatrix[,2] = sumTallyMatrix[,2] / numOutcomes
387
388 overallAverageSum = sum(sumTallyMatrix[,1] * sumTallyMatrix[,3] / numOutcomes)
389

```

```

390     list(probabilities = sumTallyMatrix, average = overallAverageSum)
391 }

1  #include <Rcpp.h>
2  #include <cstdio>
3  #include <omp.h>
4
5  using namespace Rcpp;
6
7  std::vector<std::vector<int > > table(std::vector<std::vector<int > > vec_e
8
9      int prev = vec_eventList[0][0];
10     int numFound = 1;
11
12     for(unsigned int i = 1; i < vec_eventList.size(); i++){
13         if(prev == vec_eventList[i][0]){
14             numFound++;
15         }else{
16             combMatrix[0].push_back(numFound);
17             numFound = 1;
18             prev = vec_eventList[i][0];
19         }
20     }
21     combMatrix[0].push_back(numFound);
22
23     return combMatrix;
24
25 }
26
27 //factorial function
28 int factorial(int n)
29 {
30     int result = 1;
31     while(n > 1) {
32         result *= n--;
33     }
34     return result;
35 }
36
37 // This helper function returns the probabilities of each element of eventL
38 std::vector<float> getEventListProbs(int ndicePerRoll, int nsidesPerDie, std
39

```

```

40 // On the assumption that eventList has length nrolls (which is safe since
41 // private helper function), we calculate the probability of getting an
42 // outcome (a "success") on each of the rolls by iterating through the
43 // successes for that roll and adding the corresponding probability to c
44
45 std::vector<float> eventListProbs;
46
47 #ifdef DEBUG_LIST_PROBS
48     for(unsigned int i = 0; i < eventList.size(); i++){
49         for(unsigned int j = 0; j < eventList[i].size(); j++){
50             std::cout << eventList[i][j] << "_";
51         }
52     }
53     std::cout << std::endl;
54
55 #endif
56
57     for (unsigned int i = 0; i < eventList.size(); i++)
58     {
59         std::sort(eventList[i].begin(), eventList[i].end());
60         float successProbForThisRoll = 0.0;
61         for (unsigned int j = 0; j < eventList[i].size(); j++)
62         {
63             successProbForThisRoll = successProbForThisRoll + probs[(eventList[i][j] - 1) * nrolls + i];
64         }
65         eventListProbs.push_back(successProbForThisRoll);
66     }
67
68     return eventListProbs;
69 }
70
71 float RowGetEventProb(int ncol, int c_nrolls, int c_ndicePerRoll, int c_nside)
72 {
73
74
75     std::vector<std::vector<int>> vec_eventList;
76     for(int* p = combMat+matrix_i*ncol; p < combMat+matrix_i*ncol+ncol; p++)
77         vec_eventList.push_back(std::vector<int>(p, p+1));
78 }
79
80 //If each element of eventList is a length-1 vector, we can convert eventList
81 //itself to a vector; then we calculate the probability of getting the

```

```

82 //set of outcomes specified by eventList in any order (reflecting the f
83 //orderMatters was passed in as FALSE)
84
85 //Replaces line supply(eventList, max) which converts to a vector
86 std::vector<std::vector<int>>>combMatrix(1);
87
88 combMatrix = table(vec_eventList, combMatrix);
89
90 int combMatProduct = 1;
91
92 for(unsigned int i = 0; i < combMatrix[0].size(); i++){
93     combMatProduct *= factorial(combMatrix[0][i]);
94 }
95
96 std::vector<float> eventListProb = getEventListProbs(c_ndicePerRoll, c_
97
98 float productEventListProb = 1.0;
99
100 for(unsigned int i = 0; i < eventListProb.size(); i++){
101     productEventListProb *= eventListProb[i];
102 }
103
104 float result = (productEventListProb * factorial(c_nrolls))/combMatProd
105
106 return result;
107 }
108
109
110 RcppExport SEXP ompApplyGetEventProb(SEXP combMatrix, SEXP nrolls, SEXP probs
111     NumericMatrix m = combMatrix;
112     NumericVector probsM = probs;
113
114     int ncol = m.ncol(),
115         nrow = m.nrow(),
116         size = ncol*nrow;
117     int c_nrolls = as<int>(nrolls);
118     int c_ndicePerRoll = as<int>(ndicePerRoll);
119     int c_nsidesPerDie = as<int>(nsidesPerDie);
120     int* hv = new int[size];
121
122 //#pragma omp parallel for schedule(static,8) collapse(2)
123     for(int i = 0; i < nrow; i++){

```

```

124         for(int j = 0; j < ncol; j++){
125             hv[i*ncol + j] = m(i, j);
126         }
127     }
128
129     std::vector<float> hprobs = as<std::vector<float>>(probs);
130     float * passprob = &hprobs[0];
131
132
133     int nProcessors=omp_get_max_threads();
134     omp_set_num_threads(nProcessors);
135
136
137     float sum[nProcessors];
138     float totalSum = 0;
139
140     for(int i = 0; i < nProcessors; i++)
141     {
142         sum[i] = 0;
143     }
144
145
146     #pragma omp parallel for schedule(static,1)
147     for(int i = 0; i < nrow; i++)
148     {
149         //int me = omp_get_thread_num();
150         sum[omp_get_thread_num()] += RowGetEventProb(ncol, c_nrolls, c_ndic
151         //std::cout << "_" << me << "_";
152     }
153
154
155     for(int i = 0; i < nProcessors; i++)
156     {
157         totalSum += sum[i];
158     }
159
160     return wrap(totalSum);
161 }

```

A.4 Thrust

Similarly, this contains the Thrust implementation of the dice package. The key point is that half the code is in C++ and the other half is in R to get the best performance of both worlds. Also note the location that uses *transform* and *reduce* to parallelize the computation.

[illegible]


```

79 {
80   errorVector = append(errorVector , "\n* All numbers in each element of e
81 }
82 errorVector = append(errorVector , .checkLogicalParam(orderMatters , "order
83
84 if (length(errorVector) > 0)
85 {
86   stop(errorVector)
87 }
88
89 eventList = lapply(eventList , unique)
90
91 # If eventList doesn't have an element for each roll , we add elements unt
92 # after this point , each element of eventList will constrain one roll (bu
93 # those constraints may be simply {min:max} for that roll—i.e., trivial
94
95 if (length(eventList) < nrolls)
96 {
97   eventList = lapply(c(eventList , rep(0 , nrolls - length(eventList))), fu
98 }
99
100 if (orderMatters)
101 {
102   if(is.null(probs)){
103     probs = snowGetSumProbs(ndicePerRoll , nsidesPerDie)$probabilities
104   }
105   outcomeProb = prod(.snowGetEventListProbs(ndicePerRoll , nsidesPerDie , e
106 }
107 else # i.e., if (!orderMatters)
108 {
109   # We only calculate probabilities if each element of eventList is a leng
110   # (i.e., a single number), e.g., {2, 3, 2}; if any element is longer tha
111   # {2, {3, 4}, 2}, we call ourselves recursively on each list we can con
112   # length-1 vectors (e.g., in the example above we'd call ourselves on {
113   # {2, 4, 2}); then we sum the resulting probabilities (which, since ord
114   # FALSE, account for all permutations of each of {2, 3, 2} and {2, 4, 2
115   # at our probability for the original list of {2, {3, 4}, 2}
116
117   listElemLengths = sapply(eventList , length)
118   maxListElemLength = max(listElemLengths)
119   if (maxListElemLength > 1)
120   {

```



```

121 # Here we populate combMatrix with the elements of eventList to produ
122 # matrix each row of which is a selection of one element from each ele
123 # eventList; e.g., given the eventList {{1, 2}, {1, 2, 4}, 2}, we'd p
124 # a 6 x 3 matrix with rows {1, 1, 2}, {1, 2, 2}, {1, 4, 2}, {2, 1, 2}
125 # and {2, 4, 2}
126
127 combMatrix = matrix(nrow = prod(listElemLengths), ncol = nrolls)
128 if (nrolls > 1)
129 {
130   for (i in 1:(nrolls - 1))
131   {
132     combMatrix[,i] = rep(eventList[[i]], each = prod(listElemLengths[
133   })
134 }
135 combMatrix[,nrolls] = rep(eventList[[nrolls]])
136
137 # Next we eliminate all rows that are permutations of other rows (oth
138 # would over-count in the calculations that follow)
139
140 if (nrolls > 1)
141 {
142   combMatrix = unique(t(apply(combMatrix, 1, sort)))
143 }
144 else
145 {
146   combMatrix = unique(combMatrix)
147 }
148
149 # Now we make a recursive call for each row of combMatrix and sum the
150 # probabilities to arrive at our probability for the original eventLi
151
152 probs = thrustGetSumProbs(ndicePerRoll, nsidesPerDie)$probabilities
153 # Call into C++
154 sumOfProbs <- .Call("thrustApplyGetEventProb", combMatrix, nrolls, pro
155 outcomeProb = sumOfProbs
156 }
157 else
158 {
159   # If each element of eventList is a length-1 vector, we can convert e
160   # itself to a vector; then we calculate the probability of getting th
161   # set of outcomes specified by eventList in any order (reflecting the
162   # orderMatters was passed in as FALSE)

```



```

205
206 numOutcomes = nsidesPerDie^ndicePerRoll
207 numDiceToDrop = ndicePerRoll - nkept
208 currNumArrangements = 0
209
210 sumModifier = sumModifier + (perDieModifier * nkept)
211
212 currentSum = 0
213
214 vectorOfSums = as.integer((nkept + sumModifier) :
215                             ((nsidesPerDie * nkept) + sumModifier))
216
217 numPossibleSums = length(vectorOfSums)
218
219 # sumTallyMatrix is used to track the number of times we see every possible
220 # which we will use to produce the probabilities of every sum (e.g., for
221 # see 10 as a sum 27 times, so the probability of a sum of 10 is 27/216 =
222 # the 5d6 drop 2 case we see 13 as a sum 1055 times, so the probability of
223 # is 1055/7776 = .1356739).
224
225 sumTallyMatrix = matrix(data = c(vectorOfSums,
226                                as.integer(rep.int(0, numPossibleSums)),
227                                as.integer(rep.int(0, numPossibleSums))),
228                      nrow = numPossibleSums,
229                      ncol = 3,
230                      dimnames = list(NULL,
231                                     c("Sum",
232                                       "Probability",
233                                       "Ways_to_Roll"))))
234
235 # boundaryVal is the most extreme die-roll value that will be kept (i.e.,
236 # value: e.g., for 5d6 drop lowest two, if our sorted die rolls are {3 4 4
237 # We'll call all dice with this value the "b" dice (because they're on the
238
239 for (boundaryVal in 1 : nsidesPerDie)
240 {
241
242     # numOs is the number of dice whose values are outside of boundaryVal (
243     # two, if we roll {3 4 4 5 6}, boundaryVal is 4, so numOs is 1).
244     We'll call these dice the "o"
245     # dice (because they're [o]utside our boundary).
246     # NOTE: We have an embedded if clause in our for-loop declaration because

```

```

246 # and boundaryVal is 1 or we're dropping highest and boundaryVal is nsidesPerDie
247 # any dice whose values are outside of boundaryVal, and hence numOs can be
The following
248 # loop syntax might look suspicious, but we *do* want to iterate once in
249 # as in the case where numDiceToDrop is 0 (and in all three such cases,
250
251 for (numOs in 0 : (if ((dropLowest && boundaryVal == 1) ||
252                      (!dropLowest && boundaryVal == nsidesPerDie))) (
253 {
254
255 # numBsKept is the number of b's that will be kept (e.g., for 5d6 drop
256 # {3 4 4 5 6}, numBsKept is 1, because one of the two 4's will be kept
257 # Now, since we're discarding numDiceToDrop dice (including the numOs
258 # (numDiceToDrop - numOs) of the b's and keep numBsKept of them, and
259 # b's is (numBsKept + numDiceToDrop - numOs). NOTE: Hence, the number
260 # exceed boundaryVal is (nkept - numBsKept). We will call these high
261 # they're "inside" our boundary).
262
263 for (numBsKept in 1 : nkept)
264 {
265
266 # By this part of the function, we've specified a class of outcomes
267 # (boundaryVal, numOs, numBsKept) values—i.e., every outcome in the
268 # following properties:
269 # 1). the die-roll boundary value boundaryVal;
270 # 2). numOs "o" dice, whose values are outside boundaryVal and will
271 # 3). numBsKept "b" dice that will be kept. Furthermore, each such
272 # 4). (numBsKept + numDiceToDrop - numOs) "b" dice in total, and
273 # 5). (nkept - numBsKept) "i" dice, whose values are inside boundaryVal
274
275 numBs = (numBsKept + numDiceToDrop - numOs)
276 numIs = (nkept - numBsKept)
277
278 paste("\n\nIn this class, boundaryVal is ", boundaryVal, ", numOs is ",
279
280 # Now, we're interested in sums for the various outcomes in this class
281 # don't depend upon the order in which the various values appear in
282 # rolls; i.e., multiple outcomes in this class will have the same v
283 # the b's, and the i's appear at different places in the sequence of
To account
284 # for this, we need to multiply each distinct outcome by the number
285 # that are identical to it except for the order in which the o's, b

```

```

286 # (NOTE: the orders *within* these groups are accounted for below:
287 # is accounted for immediately below, and we account for the order
288 # section of the code in which we enumerate the i's). For now, we
289 # which to multiply each sum we find; this term is a result of the
290 # combinatoric interpretation as the number of ways to put n distin
291 # case, our die rolls) into 3 bins of size numOs, (numBsKept + numL
292 # and (nkept - numBsKept), corresponding to the number of o's, b's,
293
294 numArrangementsOfDice = (factorial(ndicePerRoll) /
295                          (factorial(numOs) * factorial(numBs) * f
296
297 # [NOTE: The formula above could overflow if ndicePerRoll gets large
298 # consider using lfactorial()—but I think the function would keel
299
300 # Because we support dropping lowest or highest values, we define c
301 # to allow us to operate over appropriate ranges for the rest of th
302
303 innerBoundary = if (dropLowest) nsidesPerDie else 1
304 outerBoundary = if (dropLowest) 1 else nsidesPerDie
305 rangeOfOVals = abs(boundaryVal - outerBoundary)
306 rangeOfIVals = abs(boundaryVal - innerBoundary)
307 possibleIValsVec = if (dropLowest) ((boundaryVal+1) : nsidesPerDie)
308
309 # Next: The value of boundaryVal is fixed for this loop, but there
310 # that outcomes in this class might have; because we don't care abo
311 # to increase our multiplicity term to account for the outcomes tha
312 # to this iteration's distinct outcome but for the values (and order
313
314 numArrangementsOfDice = numArrangementsOfDice * rangeOfOVals^numOs
315
316 # Now that we've accounted for sorting the values into three bins an
317 # "o" values that are immaterial to our calculations, we can treat
318 # sorted into groups and can focus our attention on the numBsKept b
319 # i's. The numBsKept b's will contribute a known amount to our sum
320 # this class (viz., numBsKept * boundaryVal); but the i's will cont
321 # depending on their values. So now we turn to determining the pos
322 # for this class by enumerating the possible values for the i's.
323
We will work as follows:
323 # rangeOfIVals is the distance between the smallest and largest pos
324 # class of outcomes, and we use it to determine the number of distin
325 # class, which is given by rangeOfIVals^numIs. We create an outcom
326 # rows as there are distinct outcomes for this class and nkept colum

```

```

327      # in a row corresponds to a die-roll value, and the sum of the row
328      # sum for that distinct outcome. We populate outcomeMatrix with a
329      # value for the i's in this class (and hence all distinct outcomes
We then
330      # calculate the number of permutations of each distinct outcome (e.g.
331      # the outcome {1, 1, 2} has three permutations) and use this inform
332      # probability of every possible outcome in this class.
333
334      if (numBsKept == nkept)
335      {
336          currentSum = (numBsKept * boundaryVal) + sumModifier
337          # We adjust row index by (nkept - 1) so that, e.g., a 3d6 tally s
338          sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = sumTa
339      }
340      else
341      {
342          outcomeMatrix = matrix(nrow = choose((rangeOfIVals + numIs - 1), n
343
344          if (dim(outcomeMatrix)[1] > 0)
345          {
346
347              outcomeMatrix[,1 : numBsKept] = boundaryVal
348
349              hCombs = combinations(n = rangeOfIVals,
350                                  r = numIs,
351                                  v = possibleIValsVec,
352                                  repeats.allowed = TRUE)
353              hPermCounts = apply(hCombs, 1, function(x) factorial(numIs)/prod
354
355              outcomeMatrix[, (numBsKept+1) : nkept] = hCombs
356
357              for (rowNum in 1 : nrow(outcomeMatrix))
358              {
359                  currentSum = sum(outcomeMatrix[rowNum,]) + sumModifier
360                  currNumArrangements = numArrangementsOfDice * hPermCounts[rowN
361                  sumTallyMatrix[currentSum - sumModifier - (nkept - 1), 2] = s
362              }
363          }
364      }
365  }
366 }
367 }

```

```

368
369     if (perDieMinOfOne)
370     {
371         if (sumTallyMatrix[numPossibleSums,1] <= nkept)
372         {
373             sumTallyMatrix = matrix(data = c(nkept, numOutcomes, numOutcomes),
374                                     nrow = 1,
375                                     ncol = 3,
376                                     dimnames = list(NULL,
377                                                    c("__Sum__","__Probability__")
378         )
379         else
380         {
381             extraWaysToRollMin = sum(sumTallyMatrix[sumTallyMatrix[,1] < nkept,2])
382             sumTallyMatrix = sumTallyMatrix[sumTallyMatrix[,1] >= nkept,]
383             sumTallyMatrix[1,2] = sumTallyMatrix[1,2] + extraWaysToRollMin
384         }
385     }
386
387     sumTallyMatrix[,3] = sumTallyMatrix[,2]
388     sumTallyMatrix[,2] = sumTallyMatrix[,2] / numOutcomes
389
390     overallAverageSum = sum(sumTallyMatrix[,1] * sumTallyMatrix[,3] / numOutcomes)
391
392     list(probabilities = sumTallyMatrix, average = overallAverageSum)
393 }

1  #include <Rcpp.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/host_vector.h>
4  #include <thrust/sequence.h>
5  #include <cstdio>
6
7  using namespace Rcpp;
8
9  __device__ std::vector<std::vector<int>>> table(std::vector<std::vector<int>>>
10
11     int prev = vec_eventList[0][0];
12     int numFound = 1;
13
14     for(unsigned int i = 1; i < vec_eventList.size(); i++){
15         if(prev == vec_eventList[i][0]){

```

```

16         numFound++;
17     }else{
18         combMatrix[0].push_back(numFound);
19         numFound = 1;
20         prev = vec_eventList[i][0];
21     }
22 }
23 combMatrix[0].push_back(numFound);
24
25 return combMatrix;
26
27 }
28
29 //factorial function
30 int factorial(int n)
31 {
32     int result = 1;
33     while(n > 1) {
34         result *= n--;
35     }
36     return result;
37 }
38
39 // This helper function returns the probabilities of each element of eventList
40 --device-- std::vector<float> getEventListProbs(int ndicePerRoll, int nsides)
41
42     // On the assumption that eventList has length nrolls (which is safe since
43     // private helper function), we calculate the probability of getting an
44     // outcome (a "success") on each of the rolls by iterating through the
45     // successes for that roll and adding the corresponding probability to
46
47     std::vector<float> eventListProbs;
48
49 #ifdef DEBUG_LIST_PROBS
50     for(unsigned int i = 0; i < eventList.size(); i++){
51         for(unsigned int j = 0; j < eventList[i].size(); j++){
52             std::cout << eventList[i][j] << " ";
53         }
54     }
55     std::cout << std::endl;
56
57 #endif

```



```

58
59     for (unsigned int i = 0; i < eventList.size(); i++)
60     {
61         std::sort(eventList[i].begin(), eventList[i].end());
62         float successProbForThisRoll = 0.0;
63         for (unsigned int j = 0; j < eventList[i].size(); j++)
64         {
65             successProbForThisRoll = successProbForThisRoll + probs[(eventList[i][j] - 1) % probs.size()];
66         }
67         eventListProbs.push_back(successProbForThisRoll);
68     }
69
70     return eventListProbs;
71 }
72
73 struct RowGetEventProb
74 {
75     int *combMat;
76     float *probs;
77     int ncol;
78     int c_ndicePerRoll;
79     int c_nsidesPerDie;
80     int c_nrolls;
81
82     RowGetEventProb(int ncol, int c_nrolls, int c_ndicePerRoll, int c_nsidesPerDie)
83         : ncol(ncol), c_nrolls(c_nrolls), c_ndicePerRoll(c_ndicePerRoll),
84           c_nsidesPerDie(c_nsidesPerDie)
85     {
86         combMat = thrust::raw_pointer_cast(&it_combMat[0]);
87         probs = thrust::raw_pointer_cast(&it_probs[0]);
88     }
89
90     __device__ float operator()(const int& matrix_i)
91     {
92         std::vector<std::vector<int>> vec_eventList;
93         for(int* p = combMat+matrix_i*ncol; p < combMat+matrix_i*ncol+ncol; p++)
94             vec_eventList.push_back(std::vector<int>(p, p+1));
95     }
96
97     //If each element of eventList is a length-1 vector, we can convert
98     //itself to a vector; then we calculate the probability of getting
99     //set of outcomes specified by eventList in any order (reflecting the

```

```

100         //orderMatters was passed in as FALSE)
101
102         //Replaces line supply(eventList, max) which converts to a vector
103         std::vector<std::vector<int>>>combMatrix(1);
104
105         combMatrix = table(vec_eventList, combMatrix);
106
107         int combMatProduct = 1;
108
109         for(unsigned int i = 0; i < combMatrix[0].size(); i++){
110             combMatProduct *= factorial(combMatrix[0][i]);
111         }
112
113         std::vector<float> eventListProb = getEventListProbs(c_ndicePerRoll);
114
115         float productEventListProb = 1.0;
116
117         for(unsigned int i = 0; i < eventListProb.size(); i++){
118             productEventListProb *= eventListProb[i];
119         }
120
121         float result = (productEventListProb * factorial(c_nrolls))/combMatProduct;
122
123         return result;
124     }
125 };
126
127 RcppExport SEXP thrustApplyGetEventProb(SEXP combMatrix, SEXP nrolls, SEXP ndicePerRoll,
128     NumericMatrix m = combMatrix;
129     NumericVector probsM = probs;
130
131     // Import into C++ from SEXP
132     int ncol = m.ncol(),
133         nrow = m.nrow(),
134         size = ncol*nrow;
135     int c_nrolls = as<int>(nrolls);
136     int c_ndicePerRoll = as<int>(ndicePerRoll);
137     int c_nsidesPerDie = as<int>(nsidesPerDie);
138     thrust::host_vector<int> hv(size);
139     for(int i = 0; i < nrow; i++){
140         for(int j = 0; j < ncol; j++){
141             hv[i*ncol + j] = m(i, j);

```

```

142     }
143 }
144
145 // Copy into device vectors
146 thrust::device_vector<int> dm(size);
147 thrust::copy(hv.begin(), hv.end(), dm.begin());
148
149 thrust::host_vector<float> hprobs(probsM.size());
150 thrust::copy(probsM.begin(), probsM.end(), hprobs.begin());
151
152 thrust::device_vector<float> dprobs(size);
153 thrust::copy(hprobs.begin(), hprobs.end(), dprobs.begin());
154
155 thrust::device_vector<int> seq(nrow);
156 // Set up indexing
157 thrust::sequence(seq.begin(), seq.end());
158 thrust::device_vector<float> dv(nrow);
159
160 // Run functor
161 thrust::transform(seq.begin(), seq.end(),
162                  dv.begin(),
163                  RowGetEventProb(ncol, c_nrolls,
164                                 c_ndicePerRoll, c_nsidesPerDie,
165                                 dm.begin(), dprobs.begin()));
166 thrust::host_vector<float> host_prob(nrow);
167 // Copy results back
168 thrust::copy(dv.begin(), dv.end(), dprobs.begin());
169
170 // Return sum (default reduce behavior)
171 return wrap(thrust::reduce(dv.begin(), dv.end()));
172 }

```

A.5 Speed Test

This contains the code used to time the different backends and plot their performance. Note that it allows for the arguments to be passed in and various backends to be turned off and on.

```

1 library(dice)
2 library(ggplot2)
3 # source('snowdice.R')
4 # source('optdice.R')

```

```

5 # source('cppdice.R')
6 # source('thrstdice.R')
7 # source('ompdice.R')
8
9 delta <- 0.00001
10
11 # Helper to record into the dataframe
12 recordTime <- function(df, i, rolls, start, backend){
13   time <- Sys.time() - start
14   units(time) <- "secs"
15   df$time[i] <- time
16   df$rolls[i] <- rolls
17   df$backend[i] <- backend
18   df
19 }
20
21 # Stops execution if the results did not match
22 stopIfDifferent <- function(a, b){
23   print(a)
24   print(b)
25   stopifnot(abs(a - b) < delta)
26 }
27
28 # Benchmarks each backend with different number of dice
29 speedTest <- function(rollsToTest, argfun, backends){
30   backend_count <- length(which(sapply(backends, function(x){x[1]}) == T))
31   # Set up data frame
32   df <- data.frame(time=numeric(length(rollsToTest)*backend_count),
33                     rolls=numeric(length(rollsToTest)*backend_count),
34                     backend=character(length(rollsToTest)*backend_count),
35                     stringsAsFactors = FALSE)
36   i <- 1
37   # Test each problem size
38   for(rolls in rollsToTest){
39     # Generate args from parameter
40     args <- argfun(rolls)
41
42     # Test original version
43     if(backends$original){
44       start <- Sys.time()
45       original <- do.call(getEventProb, args)
46       df <- recordTime(df, i, rolls, start, "original")

```

```

47     i <- i + 1
48 }
49
50 # Test snow version
51 if(backend$snow){
52     start <- Sys.time()
53     result <- do.call(snowGetEventProb, args)
54     if(!backend$original){
55         original <- result
56     }
57     df <- recordTime(df, i, rolls, start, "snow")
58     i <- i + 1
59     stopIfDifferent(original, result)
60 }
61
62 # Test thrust version
63 if(backend$thrust){
64     start <- Sys.time()
65     result <- do.call(thrustGetEventProb, args)
66     df <- recordTime(df, i, rolls, start, "thrust")
67     i <- i + 1
68     stopIfDifferent(original, result)
69 }
70
71 # Test thrust version
72 if(backend$omp){
73     start <- Sys.time()
74     result <- do.call(ompGetEventProb, args)
75     df <- recordTime(df, i, rolls, start, "omp")
76     i <- i + 1
77     stopIfDifferent(original, result)
78 }
79
80 if(backend$cpp){
81     start <- Sys.time()
82     result <- do.call(cppGetEventProb, args)
83     df <- recordTime(df, i, rolls, start, "cpp")
84     i <- i + 1
85     stopIfDifferent(original, result)
86 }
87
88 # Test optimized version

```

```

89     if(backends$optimized){
90         start <- Sys.time()
91         result <- do.call(optGetEventProb, args)
92         df <- recordTime(df, i, rolls, start, "optimized")
93         i <- i + 1
94         stopIfDifferent(original, result)
95     }
96
97     print('——')
98 }
99 # Plot results
100 p <- ggplot(df, aes(y=time, x=rolls, group=backend, colour=backend))
101 p <- p + geom_line()
102 p
103 }
104
105 # Run test
106 # speedTest(rollsToTest=1:10, function(rolls){
107 #   list(nrolls=3, ndicePerRoll=rolls, nsidesPerDie=6, eventList=rep(list(r
108 # }, list(snow=T, original=F, thrust=T, cpp=F, optimized=T, omp=T))

```

B Contributions

- Nelson Johansen - C++ Implementation, Report
- Ricardo Matsui - Snow, Thrust, LaTeX Formatting
- Michael Polyakov - OpenMP, Report

References

- [1] Arena, Dylan. *Package dice*
<http://cran.r-project.org/web/packages/dice/dice.pdf>
- [2] Wickham, Hadley. *Package ggplot2 - bar_geom*
http://docs.ggplot2.org/current/geom_bar.html
- [3] Tierney Luke, Rossini A., Li Na, Svecikova H. *Package snow*. 2013 September
<http://cran.r-project.org/web/packages/snow/snow.pdf>

- [4] Thrust :: CUDA Toolkit Documentation
<http://docs.nvidia.com/cuda/thrust/index.html#axzz3UzCCURk>
- [5] OpenMP
<http://openmp.org/wp/>
- [6] Leisch, Friedrich. *Creating R Packages: A Tutorial* 2009 September
<http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf>
- [7] Matloff, N. (2015). *Programming on Parallel Machines*. University of California, Davis.
<http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBookW2015.pdf>