Abstract
Calculation of the binomial coefficient via the Pascal Triangle offers an interesting playing field for numerical algorithms. Its parallelization in particular is a non-trivial exercise enabling the study of many important aspects of parallel programming. Several algorithmic approaches and their serial and parallel performance are discussed. The OpenMP library was used throughout.

1 Introduction
Calculation of the binomial coefficient
\[
\binom{n}{k} = \frac{n!}{(n-k)!k!}
\]
via the Pascal Triangle (Fig. 1) offers an interesting playing field for numerical algorithms. Each line of Fig. 1 corresponds to one value of \( n \) and contains the values of \( \binom{n}{k} \) for \( k = 0, \ldots, n \).

The software discussed in this text is based on the properties
\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}
\]
indicated by arrows for the non-trivial values, and the symmetry relation
\[
\binom{n}{k} = \binom{n}{n-k}
\]
indicated by the dash-dotted line running down the center of the graph.

2 Serial Execution
2.1 Naive Algorithm
The simplest idea for using the Pascal Triangle to calculate \( \binom{n}{k} \) is based on recursively reducing each value \( \binom{n}{k} \) to its two components \( \binom{n-1}{k-1} \) and \( \binom{n-1}{k} \) in eq. (1). Recursion is aborted when the trivial values \( \binom{n}{0} = \binom{n}{n} = 1 \) are reached. Values from the right hand side of the Pascal Triangle are mapped onto the left hand side by the symmetry relation (2), which reduces the number of if-clauses required for the abortion conditions, as well as coming in handy later.

Implementation
```c
#define ARG_TYPE unsigned int
#define ARG_FMT "%u"
#define RES_TYPE long long unsigned int
#define RES_FMT "%Lu"

RES_TYPE nk_naive_ser(ARG_TYPE nn, ARG_TYPE kk)
{
    if(kk > (nn>>1) )
        return nk_naive_ser(nn, nn-kk);
    if(1 == kk) return nn;
    if(0 == kk) return 1;
    return nk_naive_ser(nn-1, kk-1) + nk_naive_ser(nn-1, kk);
}
```

Properties As can be seen in Fig. 1 this results in many values being calculated repeatedly. A good example of this is \( \binom{5}{2} \), which is broken down into \( \binom{5}{1} \) and \( \binom{5}{3} \). Now \( \binom{5}{3} \) is broken down into \( \binom{5}{2} \) and \( \binom{5}{4} \), while \( \binom{5}{4} \) is broken down into \( \binom{5}{3} \) and \( \binom{5}{5} \), so \( \binom{5}{3} \) is calculated twice. In fact, \( \binom{5}{2} \) is evaluated as \( \binom{5}{3} \) due to symmetry (2), so \( \binom{5}{3} \) is calculated three times during the evaluation of \( \binom{7}{3} \). This problem multiplies exponentially for larger values of \( n \). Clearly, this is a very inefficient algorithm. Fortunately, it can be improved in an obvious way.
2.2 Buffered Algorithm

Instead of calculating the same value several times, all intermediate values, once calculated, are stored in a buffer. Whenever the recursion algorithm revisits this value, the stored value is used. This dramatically reduces the number of recursions necessary.

2.2.1 Buffer Layout

It is not obvious how to map the Pascal Triangle into memory, so this section discusses how this was done in the present exercise. Due to symmetry (2), only the left half of the Pascal Triangle (the "left triangle") is stored. Non-trivial values need not be stored.

A left triangle for \( n \leq n_{\text{max}} \) needs values to be stored at the nodes of a quadratic grid rotated by 45° with respect to the symmetry axis of the Pascal Triangle. The quadratic grid can be indexed tentatively by \((i', j') = (n - k - 2, k - 2)\). The offsets of 2 make sure that the grid begins with \((0,0)\) at the first non-trivial value \( \begin{pmatrix} 4 \\ 2 \end{pmatrix} \). This indexing is exemplified in the left triangle of Fig. 2. For \( n - k - 2 < \ell \) the tentative indices are kept, i.e., \( i = i', j = j' \). In order to obtain the smallest quadratic grid \([0,...,\ell-1]|0,...,\ell-1]\) capable of storing all values needed, the lower left of the left triangle (with light blue background in Fig. 2) needs to be mapped onto the unused index space to the right of the symmetry axis (yellow arrow). This yields \( \ell = (n_{\text{max}} + 1)/2 - 1 \), where \( \lfloor \cdot \rfloor \) denotes rounded-down integer division. For \( n - k - 2 \geq \ell \) the tentative indices are transformed into mapped indices via the relation \( j = 2 * \ell - 1 - i' \) and \( i = j' \).

Consequently, the quadratic grid is mapped onto a linear grid by the standard relation \( t = \ell \cdot j + i \).

Implementation

```c
#include <assert.h>

void get_nmax(ARG_TYPE limit)
{
    max_threads = omp_get_max_threads();
    printf("Enter max. n (2-%i): ", limit);
```
Buffer dimensions: $\ell \times \ell$.

$\ell = (n_{\text{max}} + 1)/2 - 1$ (here, $\ell = 3$).

Tentative indices $(i', j') = (n-k-2, k-2)$.

$\ell = (n_{\text{max}}+1)/2 - 1$ (here, $\ell = 3$).

Tentative indices $(i', j') = (n-k-2, k-2)$.

Buffer layout for storing calculated binomial coefficients. Since the Pascal Triangle is symmetric, calculation of grayed values on the right half can be referred to the left half. The black circles on the left show the non-trivial values starting with $n=4$, $k=2$, $\binom{n}{k} = 6$ at the top and ending with the line $n=8$, $k=0,\ldots,2$ at the bottom, i.e. $n_{\text{max}} = 8$.

```c
scanf("%d", &nmax);
assert( nmax >= 4 && nmax <= limit );
// buffer dimension = [ (nmax+1)/2 -2 ]^2
buflen = ( (nmax+1)>>1 ) - 1;
}
RES_TYPE *noverk;

void init_buffer_serial()
{
    int ii;
    int size = buflen*buflen;

    noverk = malloc( 
        size*sizeof(RES_TYPE) );
    for(ii=0; ii<size; ++ii)
        noverk[ii] = 0;
}
RES_TYPE nk_buf_ser(ARG_TYPE nn, ARG_TYPE kk)
{
    RES_TYPE res;
    if(kk > (nn>>1) )
        return nk_buf_ser(nn, nn-kk);
    if(res = noverk[tri2buf(nn, kk)])
        return res;
    return noverk[tri2buf(nn, kk)] = 
        nk_buf_ser(nn-1, kk-1) + 
        nk_buf_ser(nn-1, kk);
}

Properties
This makes the algorithm extremely efficient, so that we can issue calculations for as much as $n = 9000$ (notwithstanding the fact that overflow errors will make these results inaccurate even with 64bit integers, as the largest central binomial coefficient representable with 64bit unsigned integers is $\binom{68}{34} = 10006297401531025124$).

3 Per-Task Parallelization

3.1 Naive Algorithm

The calculation of several binomial coefficients can be parallelized trivially as a series of independent calculations. Such problems are often called "embarrassingly parallelizable". Parallel computation of
a number of tasks is performed by the function

typedef RES_TYPE (*nk_func_t)(ARG_TYPE nn, ARG_TYPE kk);

void crunch_par(nk_func_t func)
{
    int ii;
    #pragma omp parallel for private(ii)
    for(ii=0; ii<taskmax; ++ii)
    {
        result[ii].par = func(
            task[ii].nn, task[ii].kk);
    }
}

which assigns a private ii from the for loop to each thread, which then performs the corresponding task. These are not necessarily called in order, as completion of the tasks may take wildly different amounts of time. For example, one thread may complete task (ii+1) and be assigned task (ii+2) while another thread is still busy performing task ii.

A comparison with the serial method is facilitated by omission of the #pragma line in the corresponding serial crunching function void crunch_ser(nk_func_t func).

### 3.2 Buffered Algorithm

#### 3.2.1 Distributed First Touch

The initialization of the buffer to zero is also parallelized - this results in mostly local memory access in NUMA (Non-Uniform Memory Access) architectures.

#### 3.2.2 Race Condition

As before, several independent calculation tasks can be parallelized by assigning them to different threads, so all that needs to be done is insertion of the line

```c
#pragma omp parallel for private(ii)
```

in front of the loop that works through the list of tasks.

Now there are several threads walking the Pascal Triangle, possibly solving sub-tasks for each other. Potentially, they may be reading from and writing to the same buffer element. Is that a problem? After all, if two threads read a zero and consequently decide to compute the binomial coefficient corresponding to that spot in the array, they will end up overwriting each other's values - both of which are correct, so no harm is done, right?

Unfortunately, this is true if the value we are reading at the beginning of this can only be either zero or the correct value. What if we are reading a non-zero value produced by another thread while this thread is writing it, and different from the result? Then this wrong value is used in all consequent calculations, resulting in a wrong result stored in the buffer, corrupting several end results.

When does this happen? In practice, I have observed errors in the parallel calculation on a 32 bit machine from $n = 43$. The results involved were larger than $10^{11}$. These results involve intermediate results larger than MAXINT (about $4 \cdot 10^9$). The largest central binomial coefficient below this limit is $\binom{34}{17} = 2333606220$. This is already painfully slow to calculate with the naive algorithm (at over 20s wall-time on an Intel 1.8 GHz Core Duo), so that the buffered algorithm is needed, especially for values of $n \geq 35$.

Smaller results are always correct with the buffered algorithm, as reading and writing of machine words is atomic, i.e. data manipulation of 32 bit values are not broken down into sub-tasks on 32 bit machines. However, a 64 bit value is written by a 32 bit CPU in two 32 bit steps. An error can then occur if one thread has written one non-zero half of a result but not yet the other, while the other thread reads the entire 64 bit number, receiving a wrong, but non-zero, result and using it in its calculation.

This is an example of a race condition, as the writing thread is racing all other threads to complete its writing operation before they begin their reading operations, as illustrated in Fig. 3.

This explanation implies that the problem should go away in 64 bit machines, and in fact it does.

#### 3.2.3 Fixing the Race - Data Synchronicity

Race conditions can be avoided in two ways.

**OpenMP Locks** An array of OpenMP locks is used in parallel to the buffer, with one element for each buffer element. Each thread has to obtain the lock of the corresponding buffer element before reading from or writing to it, so that the race condition is averted.

```c
omp_lock_t *noverk_lock;

void init_lock_parallel()
{
    int ii;
    int size = buflen*buflen;
    noverk_lock = malloc(size*sizeof(omp_lock_t));
```
Machine Word Flags  Instead of obtaining locks, machine word flags are used to claim buffer elements for a thread. Note that it is not possible to use any smaller memory units as flags, such as bits or bytes, as with those, several flags are addressed by the same machine word, which also leads to data synchronicity problems.

```c
int *noverk_flag;

void init_flag_serial()
{
    int ii;
    int size = buflen*buflen;
    noverk_flag = malloc( size*sizeof(int) );
    for(ii=0; ii<size; ++ii) {
        noverk_flag[ii] = 0;
    }
}

void reinit_flag_parallel()
{
    int ii;
    int size = buflen*buflen;

    #pragma omp parallel for private(ii)
    for(ii=0; ii<size; ++ii) {
        noverk_flag[ii] = 0;
    }
}

RES_TYPE nk_buf_flag(ARG_TYPE nn, ARG_TYPE kk)
{
    RES_TYPE res;
    long unsigned int ofs;
    if(kk > (nn>>1) )
        return nk_buf_flag(nn, nn-kk);
    if(1 == kk) return nn;
    if(0 == kk) return 1;
    ofs = tri2buf(nn, kk);
    omp_set_lock(noverk_lock+ofs);
    res = noverk[ofs];
    omp_unset_lock(noverk_lock+ofs);
    if(res)
        return res;
    res = nk_buf_flag(nn-1, kk-1) +
         nk_buf_flag(nn-1, kk);
    omp_set_lock(noverk_lock+ofs);
    noverk[ofs] = res;
    omp_unset_lock(noverk_lock+ofs);
    return res;
}
```

A. Markmann (2009) Parallel Programming Exercise
if( 2 == noverk_flag[ofs] )
{
    return noverk[ofs];
}
else
{
    noverk_flag[ofs] = 1;
    noverk[ofs] = res;
    noverk_flag[ofs] = 2;
    return res;
}

4 Download


References