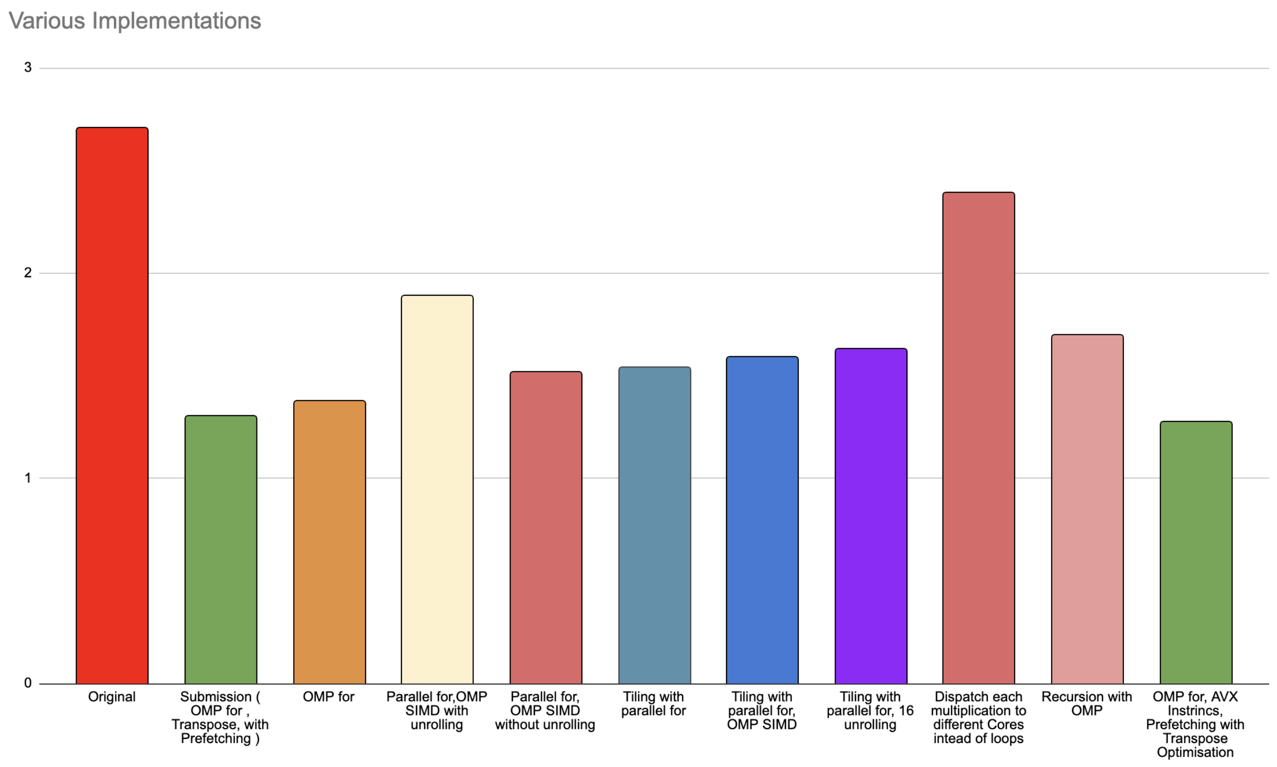
SPP Assignment 2

Submitted by T.H.Arjun, 2019111012, CSD

For this assignment I decided to improve my Q1 code for Chain Matrix Multiplication.

OpenMP Multicore Programming

Visualization of comparison of various implementations that were tried



Version Detail	Time (seconds)
Original	2.709
Submission (OMP for , Transpose, with Prefetching)	1.305
OMP for	1.377
Parallel for,OMP SIMD with unrolling	1.89
Parallel for, OMP SIMD without unrolling	1.518
Tiling with parallel for	1.542

Version Detail	Time (seconds)
Tiling with parallel for, OMP SIMD	1.59
Tiling with parallel for, 16 unrolling	1.627
Dispatch each multiplication to different Cores intead of loops	2.389
Recursion with OMP	1.70
OMP for, AVX Instrincs, Prefetching with Transpose Optimisation	1.273

Note : These values are on my personal device with 90.txt Test Case.

OMP Directives Used

- You create threads in OpenMP with the parallel construct `#pragma omp parallel`. Note that you can specify various constraints to this like `num_threads`. You can use `omp_set_num_threads(omp_get_num_procs())` to set number of threads for parallel regions that do not specify this constraint.
- The loop worksharing construct splits up loop iterations among the threads in a team. We need to use `#pragma omp for` directive for this. You can also add extra constrinats like shared variables, private variables etc.
- OpenMP reduction clause, `reduction (op : list)`. A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+"). Compiler finds standard reduction expressions containing "op" and uses them to update the local copy. Local copies are reduced into a single value and combined with the original global value at the end.
- The sections construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task. Each structured block in the sections construct is preceded by a section directive except possibly the first block, for which a preceding section directive is optional. The method of scheduling the structured blocks among the threads in the team is implementation defined.Each structured block in the sections construct is preceded by a section directive except possibly the first block, for which a preceding section directive is optional. The method of scheduling the structured blocks among the threads in the team is implementation defined. The directive for this is `#pragma omp parallel sections` and `#pragma omp section`
- The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions). The syntax is `#pragma omp simd`. It can be followed by many clauses like:

```
if([simd :] scalar-expression)
safelen(length)
simdlen(length)
linear(list[ : linear-step])
```

```
aligned(list[ : alignment])
nontemporal(list)
private(list)
lastprivate([ lastprivate-modifier:] list)
reduction([ reduction-modifier,]reduction-identifier : list)
collapse(n)
order(concurrent)
```

These are the ones I used during the assignment but there are many more. All these are well documented in OMP Documentation available here

Optimisations in Submitted Code

The following optimisations were done in the submitted code

- Paralising for loops using the `#pragma parallel for` directive in `matrix_multiply` and `transpose` functions. The outer loops here were paralised to take advantage of the multiple cores given in this assignment via this OpenMP Directive. It sends each iteration of the for loop to a different thread.

- In the last assignment, I tried to implement Prefecting to cache using GCC functions but failed to beat the compiler. This time I was able to get it working using Intel Intrinsics. Abacus uses Intel Xeon Processors and I wanted to optimise the code to this setup. So I went ahead with it and implemented the Cache Prefetching with `void _mm_prefetch (char const* p, int i)` function inside `<xmmintrin.h>` headerfile for Intel Processors. It fetches the line of data from memory that contains address p to a location in the cache heirarchy specified by the locality hint i. The thing I understood this time is that the prefetching should be done in advance instead of the time that you need it. That is I am prefetching for the future iterations now. This showed a bit of improvement. This extra distance of Prefetching is defined by `PFDISTHUGE` which I found to be best at 48

- I set the value of number of threads to the number of cores available to the program so as to reduce the number of context switch overheads by doing `omp_set_num_threads(omp_get_num_procs())` at the start of the program.

This lead to the functions to be modified in the following way:

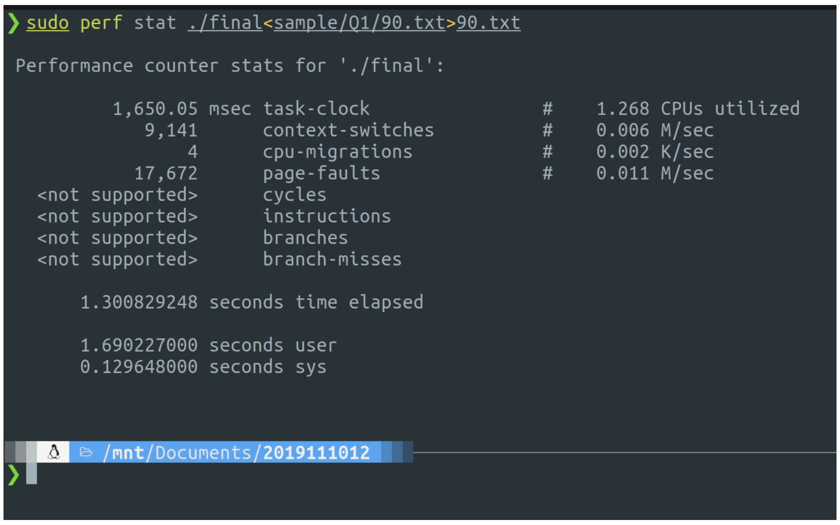
```
Matrix *matrix_multiply(Matrix *a, Matrix *a2)
{
    Matrix *result = (Matrix *)malloc(sizeof(Matrix));
    register int p = a->n, r = a2->m;
    result->matrix = (int *)malloc(p * r * sizeof(int));
    Matrix *b = transpose(a2);
    result->n = p;
    result->m = r;
#pragma omp parallel
    {
        register int q = a->m, i, j, k, temp, *b1, *a1, *c;
#pragma omp for
        for (i = 0; i < p; ++i)
        {
            a1 = a->matrix + i * q;
            c = result->matrix + i * r;

            for (j = 0; j < r; ++j)
            {
                b1 = b->matrix + j * q;
                temp = 0;
                q = q - 16;
                for (k = 0; k < q; k += 16)
                {
#define PFDISTHUGE 48

                    a1 += k;
                    b1 += k;
                    _mm_prefetch(a1 + PFDISTHUGE, _MM_HINT_T0);
                    _mm_prefetch(b1 + PFDISTHUGE, _MM_HINT_T0);
                    temp += (*(a1)) * (*(b1)) + (*(a1 + 1)) * (*(b1 + 1))
+ (*(a1 + 2)) * (*(b1 + 2)) +
                            (*(a1 + 3)) * (*(b1 + 3)) + (*(a1 + 4)) * (*
(b1 + 4)) + (*(a1 + 5)) * (*(b1 + 5)) +
                            (*(a1 + 6)) * (*(b1 + 6)) + (*(a1 + 7)) * (*
(b1 + 7)) + (*(a1 + 8)) * (*(b1 + 8)) +
                            (*(a1 + 9)) * (*(b1 + 9)) + (*(a1 + 10)) * (*
(b1 + 10)) + (*(a1 + 11)) * (*(b1 + 11)) +
                            (*(a1 + 12)) * (*(b1 + 12)) + (*(a1 + 13)) *
(*(b1 + 13)) + (*(a1 + 14)) * (*(b1 + 14)) +
                            (*(a1 + 15)) * (*(b1 + 15));
                    a1 -= k;
                    b1 -= k;
                }

                q = q + 16;
                for (; k < q; k++)
                    temp += (*(a1 + k)) * (*(b1 + k));

                *(c + j) = temp;
            }
        }
    }
    free(a->matrix);
    free(a2->matrix);
    free(b->matrix);
    return result;
}
```

```
Matrix *transpose(Matrix *a)
{
    register int N = a->n;
    register int M = a->m;
    Matrix *result = (Matrix *)malloc(sizeof(Matrix));
```

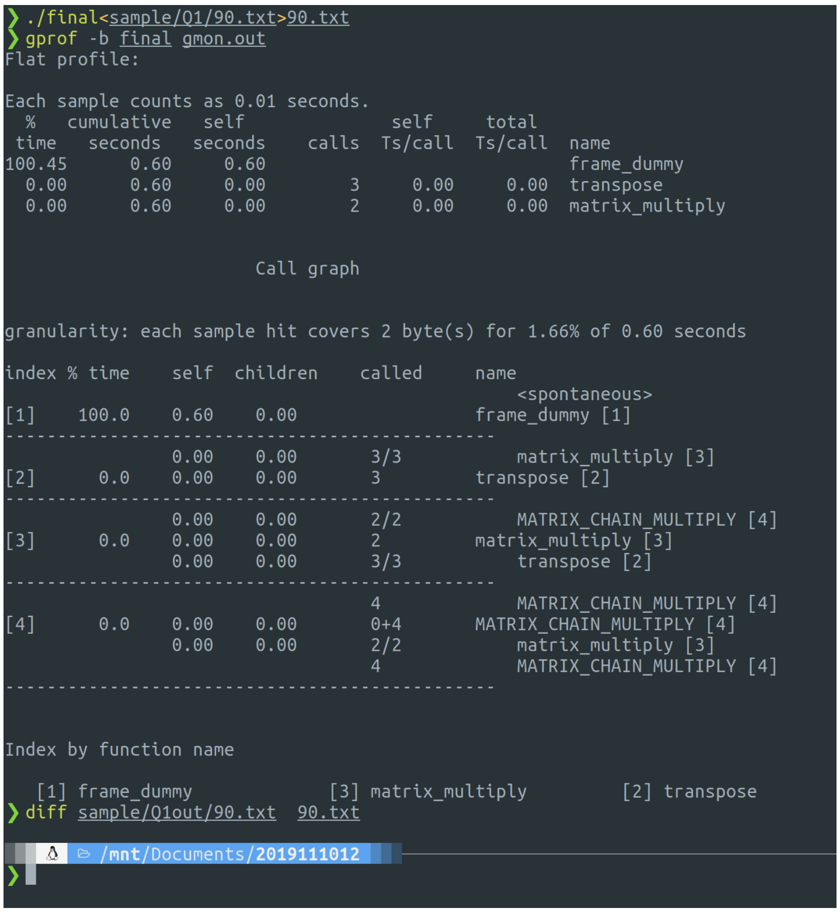
```c
    result->matrix = (int *)malloc(N * M * sizeof(int));
    result->n = M;
    result->m = N;
    register int *A = a->matrix;
    register int *B = result->matrix;
#pragma omp parallel
    {
        register int n, m;
        register int row, col;
        int d_val = 0;
        int diag = 0;
#pragma omp for
        for (col = 0; col < M; col += 8)
        {
            for (row = 0; row < N; row += 8)
            {

                for (n = row; (n < row + 8) && (n < N); ++n)
                {
                    for (m = col; (m < col + 8) && (m < M); ++m)
                    {

                        if (n != m)
                        {

                            *(B + m * N + n) = *(A + n * M + m);
                        }
                        else
                        {
                            diag = n;
                            d_val = *(A + n * M + m);
                        }
                    }
                    if (row == col)
                    {
                        *(B + diag * N + diag) = d_val;
                    }
                }
            }
        }
    }
    return result;
}
```

\

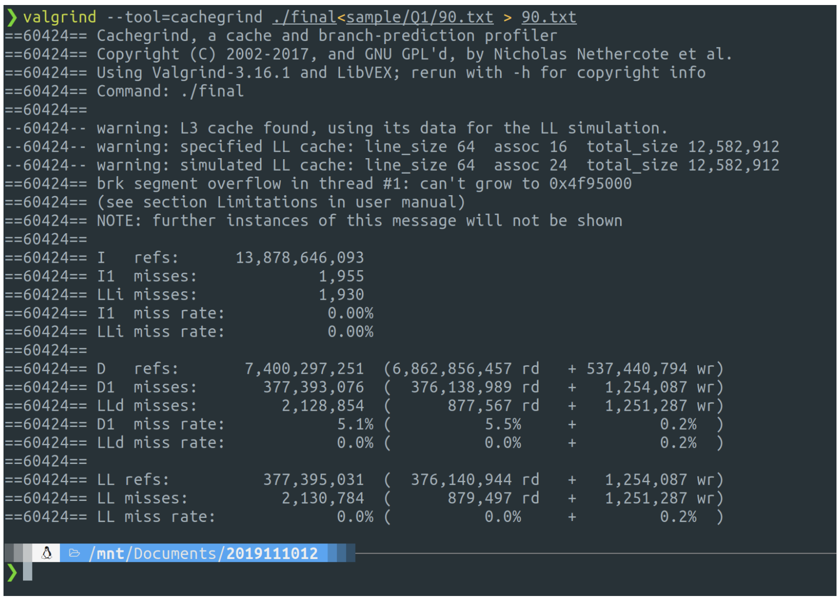
Profiling for submitted Code



```
❯ sudo perf stat ./final<sample/Q1/90.txt>90.txt

 Performance counter stats for './final':

          1,650.05 msec task-clock                #    1.268 CPUs utilized
             9,141      context-switches          #    0.006 M/sec
                 4      cpu-migrations            #    0.002 K/sec
            17,672      page-faults               #    0.011 M/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       1.300829248 seconds time elapsed

       1.690227000 seconds user
       0.129648000 seconds sys


 /mnt/Documents/2019111012
❯
```



```
❯ ./final<sample/Q1/90.txt>90.txt
❯ gprof -b final gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.45      0.60     0.60                             frame_dummy
  0.00      0.60     0.00        3     0.00     0.00  transpose
  0.00      0.60     0.00        2     0.00     0.00  matrix_multiply


                        Call graph


granularity: each sample hit covers 2 byte(s) for 1.66% of 0.60 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.60    0.00                 frame_dummy [1]
-----------------------------------------------
                0.00    0.00       3/3           matrix_multiply [3]
[2]      0.0    0.00    0.00       3         transpose [2]
-----------------------------------------------
                0.00    0.00       2/2           MATRIX_CHAIN_MULTIPLY [4]
[3]      0.0    0.00    0.00       2         matrix_multiply [3]
                0.00    0.00       3/3           transpose [2]
-----------------------------------------------
                                   4             MATRIX_CHAIN_MULTIPLY [4]
[4]      0.0    0.00    0.00       0+4       MATRIX_CHAIN_MULTIPLY [4]
                0.00    0.00       2/2           matrix_multiply [3]
                                   4             MATRIX_CHAIN_MULTIPLY [4]
-----------------------------------------------


Index by function name

   [1] frame_dummy             [3] matrix_multiply         [2] transpose
❯ diff sample/Q1out/90.txt  90.txt

 /mnt/Documents/2019111012
❯
```



```
❯ valgrind --tool=cachegrind ./final<sample/Q1/90.txt > 90.txt
==60424== Cachegrind, a cache and branch-prediction profiler
==60424== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==60424== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==60424== Command: ./final
==60424==
--60424-- warning: L3 cache found, using its data for the LL simulation.
--60424-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--60424-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==60424== brk segment overflow in thread #1: can't grow to 0x4f95000
==60424== (see section Limitations in user manual)
==60424== NOTE: further instances of this message will not be shown
==60424==
==60424== I   refs:      13,878,646,093
==60424== I1  misses:             1,955
==60424== LLi misses:             1,930
==60424== I1  miss rate:           0.00%
==60424== LLi miss rate:           0.00%
==60424==
==60424== D   refs:       7,400,297,251  (6,862,856,457 rd   + 537,440,794 wr)
==60424== D1  misses:       377,393,076  (  376,138,989 rd   +   1,254,087 wr)
==60424== LLd misses:         2,128,854  (      877,567 rd   +   1,251,287 wr)
==60424== D1  miss rate:            5.1% (          5.5%     +         0.2%  )
==60424== LLd miss rate:            0.0% (          0.0%     +         0.2%  )
==60424==
==60424== LL refs:          377,395,031  (  376,140,944 rd   +   1,254,087 wr)
==60424== LL misses:          2,130,784  (      879,497 rd   +   1,251,287 wr)
==60424== LL miss rate:             0.0% (          0.0%     +         0.2%  )

 /mnt/Documents/2019111012
❯
```

Different Versions Mentioned

1. OMP for, AVX Instrincs, Prefetching with Transpose Optimisation (This was not submitted due to restrictions on compiler flags but this performs best)

This was the best performing code out of the bunch but I could not submit it for the assignment. The different versions mentioned below that mention SIMD are not actual SIMD in the sense that they use the SIMD Directive from OMP. They tell the compiler that the section can be vectorised. It is left upto the compiler on how the vectorisation happens. Now this is where Intel Instrincs come in. Intel intrinsic instructions are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code. You code in assembly like statements to excute vectorised code with no dependency on the compiler. The beauty of this is that the code generated is perfectly vectorised and thus performs exceptionally well. I could not submit this code because it requires the `-mavx2` flag passed to GCC to compile this as this is processor specific code. I wrote this code to appreciate the beauty of pure SIMD and its performance compared to leaving it to the compiler. I also added the prefetch speed up in this code. Given below is my implentation of transposed vectorised multiplication routine using AVX Instructions since this question requires long long int. Also the support for long long int (64 bit integers come in AVX, which is available only on some processors)

```
int *transpose_native_parallel_multiple(int *matA, int *matB, int rowA,
int colA, int colB)
{
    __m256i I0, I1, I3, I4, I5, I6, I7, T1, T2, T0, T3, T4, T5, T6, T7,
I2;

    int *matD = (int *)malloc(sizeof(int) * colA * colB);
    for (int x = 0; x < rowA; x += 8)
    {
        for (int y = 0; y < colB; y += 8)
        {
#define PFDISTHUGE 16
```

```
                _mm_prefetch(matB + (y + PFDISTHUGE + 0) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 1) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 2) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 3) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 4) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 5) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 6) ∗ rowA + x,
_MM_HINT_T0);
                _mm_prefetch(matB + (y + PFDISTHUGE + 7) ∗ rowA + x,
_MM_HINT_T0);

                I0 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 0) ∗ rowA +
x));
                I1 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 1) ∗ rowA +
x));
                I2 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 2) ∗ rowA +
x));
                I3 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 3) ∗ rowA +
x));
                I4 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 4) ∗ rowA +
x));
                I5 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 5) ∗ rowA +
x));
                I6 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 6) ∗ rowA +
x));
                I7 = _mm256_loadu_si256((__m256i ∗)(matB + (y + 7) ∗ rowA +
x));

                T0 = _mm256_unpacklo_epi64(I0, I1);
                T1 = _mm256_unpackhi_epi64(I0, I1);
                T2 = _mm256_unpacklo_epi64(I2, I3);
                T3 = _mm256_unpackhi_epi64(I2, I3);
                T4 = _mm256_unpacklo_epi64(I4, I5);
                T5 = _mm256_unpackhi_epi64(I4, I5);
                T6 = _mm256_unpacklo_epi64(I6, I7);
                T7 = _mm256_unpackhi_epi64(I6, I7);

                I0 = _mm256_unpacklo_epi64(T0, T2);
                I1 = _mm256_unpackhi_epi64(T0, T2);
                I2 = _mm256_unpacklo_epi64(T1, T3);
                I3 = _mm256_unpackhi_epi64(T1, T3);
                I4 = _mm256_unpacklo_epi64(T4, T6);
                I5 = _mm256_unpackhi_epi64(T4, T6);
                I6 = _mm256_unpacklo_epi64(T5, T7);
                I7 = _mm256_unpackhi_epi64(T5, T7);

                T0 = _mm256_permute2x128_si256(I0, I4, 0x20);
                T1 = _mm256_permute2x128_si256(I1, I5, 0x20);
```

```
            T2 = _mm256_permute2x128_si256(I2, I6, 0x20);
            T3 = _mm256_permute2x128_si256(I3, I7, 0x20);
            T4 = _mm256_permute2x128_si256(I0, I4, 0x31);
            T5 = _mm256_permute2x128_si256(I1, I5, 0x31);
            T6 = _mm256_permute2x128_si256(I2, I6, 0x31);
            T7 = _mm256_permute2x128_si256(I3, I7, 0x31);

            _mm256_storeu_si256((__m256i *)(matD + ((x + 0) * colB) + y),
T0);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 1) * colB) + y),
T1);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 2) * colB) + y),
T2);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 3) * colB) + y),
T3);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 4) * colB) + y),
T4);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 5) * colB) + y),
T5);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 6) * colB) + y),
T6);
            _mm256_storeu_si256((__m256i *)(matD + ((x + 7) * colB) + y),
T7);
        }
    }

    int *matC = (int *)malloc(sizeof(int) * rowA * colB);
#pragma omp parallel for
    for (int i = 0; i < rowA; ++i)
    {
        for (int j = 0; j < colA; ++j)
        {
            int sum = 0;
#pragma omp simd aligned(matA, matD, matC : 64) reduction(+ \
                                                          : sum)
            for (int k = 0; k < colB; ++k)
            {
                sum += M(i, k, matA, colA) * M(j, k, matD, colB);
            }
            M(i, j, matC, colB) = sum;
        }
    }
    return matC;
}
```

The performace gain because of this was not immense like I wanted it to be. This is because we are using 64 bit integers here and the maximum number of 64 bit integers a vector register on the most advanced Intel CPU can hold is 4 in AVX mode or 2 in other modes. While the code without this is able to do 16 at a time inside the loop. But this provides some speed and I believe in the future the support for 64 bit integers in these vector registers will increase. Also if the problem statement had 32-bit integers then the code would have been much faster in my opinion.

2.Pragma for

This version of the code uses just the `pragma omp for loop` in the submitted code without the prefecting. This showed a little slower performance compared to prefecting.

3.SIMD with unrolling

In this code, I used OMP `simd` directive to try vectorise the code. The simd directive requests the compiler to vectorise the loop. But this actually slowed the code in experience. This according to me is due to the following reasons.

- I am using 64-bit integers, the most advanced CPUs from intel can hold 4 64-bit integers in vector registers at the moment.
- In my previous code, I am able to access 16 64-bit integers in cache which is much faster.
- The compiler is not doing a good job with SIMD because of my unrolling.

The directive used is `#pragma omp simd aligned(a1, b1 : 64) reduction(+ \ : temp)` The code snippet of this implementation:

```
Matrix *matrix_multiply(Matrix *a, Matrix *a2)
{
    Matrix *result = (Matrix *)malloc(sizeof(Matrix));
    register int p = a->n, r = a2->m;
    result->matrix = (int *)malloc(p * r * sizeof(int));
    Matrix *b = transpose(a2);
    result->n = p;
    result->m = r;
#pragma omp parallel
    {
        register int q = a->m, i, j, k, temp, *b1, *a1, *c;
#pragma omp for
        for (i = 0; i < p; ++i)
        {
            a1 = a->matrix + i * q;
            c = result->matrix + i * r;

            for (j = 0; j < r; ++j)
            {
                b1 = b->matrix + j * q;
                temp = 0;
                q = q - 16;
#pragma omp simd aligned(a1, b1 : 64) reduction(+ \
                                                : temp)
                for (k = 0; k < q; k += 16)
                {
                    a1 += k;
                    b1 += k;
                    temp += (*(a1)) * (*(b1)) + (*(a1 + 1)) * (*(b1 + 1))
+ (*(a1 + 2)) * (*(b1 + 2)) +
                            (*(a1 + 3)) * (*(b1 + 3)) + (*(a1 + 4)) * (*
(b1 + 4)) + (*(a1 + 5)) * (*(b1 + 5)) +
                            (*(a1 + 6)) * (*(b1 + 6)) + (*(a1 + 7)) * (*
(b1 + 7)) + (*(a1 + 8)) * (*(b1 + 8)) +
                            (*(a1 + 9)) * (*(b1 + 9)) + (*(a1 + 10)) * (*
(b1 + 10)) + (*(a1 + 11)) * (*(b1 + 11)) +
                            (*(a1 + 12)) * (*(b1 + 12)) + (*(a1 + 13)) *
(*(b1 + 13)) + (*(a1 + 14)) * (*(b1 + 14)) +
                            (*(a1 + 15)) * (*(b1 + 15));
                    a1 -= k;
                    b1 -= k;
                }

                q = q + 16;
                for (; k < q; k++)
                    temp += (*(a1 + k)) * (*(b1 + k));

                *(c + j) = temp;
            }
        }
    }
    free(a->matrix);
    free(a2->matrix);
    free(b->matrix);
    return result;
}
```

4.SIMD without unrolling

To overcome the challenges mentioned above, I made a version which removes the unrolling and tries to use SIMD. In this version the compiler did a much better work at vectorising the part of the loop but again, it was not able to beat the loop unrolled version due to the fact that I am using long long int and due to the limitations in vector register size.

```
Matrix *matrix_multiply(Matrix *a, Matrix *a2)
{
    Matrix *result = (Matrix *)malloc(sizeof(Matrix));
    register int p = a->n, r = a2->m;
    result->matrix = (int *)malloc(p * r * sizeof(int));
    Matrix *b = transpose(a2);
    result->n = p;
    result->m = r;
#pragma omp parallel
    {
        register int q = a->m, i, j, k, temp, *b1, *a1, *c;
#pragma omp for
        for (i = 0; i < p; ++i)
        {
            a1 = a->matrix + i * q;
            c = result->matrix + i * r;

            for (j = 0; j < r; ++j)
            {
                b1 = b->matrix + j * q;
                temp = 0;
#pragma omp simd aligned(a1, b1 : 64) reduction(+ \
                                                : temp)
                for (k = 0; k < q; k++)
                    temp += (*(a1 + k)) * (*(b1 + k));

                *(c + j) = temp;
            }
        }
    }
    free(a->matrix);
    free(a2->matrix);
    free(b->matrix);
    return result;
}
```

5.Tiling with Parallised OMP Loops

The next technique I tried was using the Tiling Algorith of Matrix Multiplication with OMP Parallisation. This performed as good as the submitted code. But in some cases was a bit slower due to extra loop iterators and conditional checks. So I went with the submitted code, also I trusted it more because that is what I submitted last time. The tiled Algorithm code is given below with OMP for loops:

```
void matrixMultiplicationOptimized(int A_dim1, int A_dim2, int B_dim1, int
**A, int **B, int **C)
{
#pragma omp parallel
    {
        int i, j, k, l;
        int limit0 = A_dim1; // Index i limit
        int limit1 = B_dim1; // Index j limit
        int limit2 = A_dim2; // Index k limit
        int aux_i, aux_j, aux_k;
        int aux_limit_i; // Block index limit i
        int aux_limit_j; // Block index limit j
        int aux_limit_k; // Block index limit k
        int unroll_factor = 5;
        int unroll_limit;                 // Loop unroll index limit
        int acc0, acc1, acc2, acc3, acc4; // Accumulators, eliminate data
dependencies
#pragma omp for schedule(static)
        for (i = 0; i < limit0; i += g_cacheBlockSize)
        {
            // Blocking index i limit
            aux_limit_i = min((i + g_cacheBlockSize), limit0);

            for (j = 0; j < limit1; j += g_cacheBlockSize)
            {
                // Blocking index j limit
                aux_limit_j = min((j + g_cacheBlockSize), limit1);
                for (k = 0; k < limit2; k += g_cacheBlockSize)
                {
                    // Blocking index k limit
                    aux_limit_k = min((k + g_cacheBlockSize), limit2);

                    unroll_limit = aux_limit_k - (unroll_factor - 1); //
Unrolling by factor of 5

                    for (aux_i = i; aux_i < aux_limit_i; ++aux_i)
                    {
                        for (aux_j = j; aux_j < aux_limit_j; ++aux_j)
                        {

                            acc0 = 0;
                            acc1 = 0;
                            acc2 = 0;
                            acc3 = 0;
                            acc4 = 0;

                            // Unrolling for k loop
                            for (aux_k = k; aux_k < unroll_limit; aux_k +=
unroll_factor)
                            {
                                acc0 += A[aux_i][aux_k] * B[aux_k][aux_j];
                                acc1 += A[aux_i][aux_k + 1] * B[aux_k + 1]
[aux_j];
                                acc2 += A[aux_i][aux_k + 2] * B[aux_k + 2]
[aux_j];
                                acc3 += A[aux_i][aux_k + 3] * B[aux_k + 3]
[aux_j];
                                acc4 += A[aux_i][aux_k + 4] * B[aux_k + 4]
[aux_j];
                            }

                            // Gather possible uncounted elements
                            for (; aux_k < aux_limit_k; ++aux_k)
                                C[aux_i][aux_j] += A[aux_i][aux_k] *
B[aux_k][aux_j];

                            // Sum up everything
                            C[aux_i][aux_j] += acc0 + acc1 + acc2 + acc3 +
acc4;
                        }
                    }
                }
            }
        }
    }
    return;
}
```

6.Tiling with Parallised OMP Loops and SIMD from OMP

The code for this is given below. The compiler again got confused and didn't give me the results I wanted. (I wonder what it could have been with Intel AVX, but due to the fact that I could not submit even if I implemented it, I left it to OMP. I would like to play around with AVX in the future in this code too.)

```
void matrixMultiplicationOptimized(int A_dim1, int A_dim2, int B_dim1, int
**A, int **B, int **C)
{
    int limit0 = A_dim1; // Index i limit
    int limit1 = B_dim1; // Index j limit
    int limit2 = A_dim2; // Index k limit
    int unroll_factor = 5;
#pragma omp parallel
    {
        int i, j, k, l, aux_i, aux_j, aux_k;
        int aux_limit_i;  // Block index limit i
        int aux_limit_j;  // Block index limit j
        int aux_limit_k;  // Block index limit k
        int unroll_limit; // Loop unroll index limit
#pragma omp for schedule(static)
        for (i = 0; i < limit0; i += g_cacheBlockSize)
        {
            // Blocking index i limit
            aux_limit_i = min((i + g_cacheBlockSize), limit0);

            for (j = 0; j < limit1; j += g_cacheBlockSize)
            {
                // Blocking index j limit
                aux_limit_j = min((j + g_cacheBlockSize), limit1);

                for (k = 0; k < limit2; k += g_cacheBlockSize)
                {
                    // Blocking index k limit
                    aux_limit_k = min((k + g_cacheBlockSize), limit2);

                    for (aux_i = i; aux_i < aux_limit_i; ++aux_i)
                    {
                        for (aux_j = j; aux_j < aux_limit_j; ++aux_j)
                        {
                            int temp = 0;
#pragma omp simd aligned(A, B : 64) reduction(+ \
                                              : temp)
                            for (aux_k = k; aux_k < aux_limit_k; ++aux_k)
                                temp += A[aux_i][aux_k] * B[aux_k][aux_j];
                            C[aux_i][aux_j] += temp;
                        }
                    }
                }
            }
        }
    }
    return;
}
```

7.Tiling with Parallised OMP Loops and loop unrolling upto 16

In this version, I unrolled the loop upto a factor of 16. But this slowed it down. This is because of the following reasons>

- There will be an increase in accumulator variables in the above code (variables called `acc`)
- Assignment Operations, creation of these variables, addition operations etc add to the instructions needed to complete the muplication which is already instruction heavy.

8.Recursion with Multiprocessing

Next I tried the recursive Matrix Multiplication Algorithm with OMP Multiprocessing. For this I used `sections` and `section` directive of OMP. So a sections in OMP tells it to execute each section as a thread and then join them back. The main thread waits for both of them to complete in this case. The code for this method is given below. It showed a faster time but not as much as the transpose method, still this is a very viable solution to the matrix multiplication problem.

```
void multiply(int **A, int **B, int m, int n, int p, int **dest, int
doffset, int aoffset, int boffset)
{
    int min = m * n < n * p ? m * n : n * p;
    if (min <= 32768)
    {
        multiply_naive(A, B, m, n, p, dest, doffset, aoffset, boffset);
    }
    else if (m >= (n >= p ? n : p))
    {

        int split = m >> 1;
        int rest = m - split;
#pragma omp parallel sections
        {
#pragma omp section
            multiply(A, B, split, n, p, dest, doffset, aoffset, boffset);
#pragma omp section
            multiply(A + split, B, rest, n, p, dest + split, doffset,
aoffset, boffset);
        }
    }
    else if (n >= (m >= p ? m : p))
    {

        int split = n >> 1;
        int rest = n - split;
#pragma omp parallel sections
        {
#pragma omp section
```

```
            multiply(A, B, m, split, p, dest, doffset, aoffset, boffset);
#pragma omp section
            multiply(A, B, m, rest, p, dest, doffset, aoffset + split,
boffset + split);
        }
    }
    else
    {

        int split = p >> 1;
        int rest = p - split;
#pragma omp parallel sections
        {
#pragma omp section
            multiply(A, B, m, n, split, dest, doffset, aoffset, boffset);
#pragma omp section
            multiply(A, B + split, m, n, rest, dest, doffset + split,
aoffset, boffset);
        }
    }
}
void multiply_naive(int **A, int **B, int m, int n, int p, int **dest, int
doffset, int aoffset, int boffset)
{
    register int i, j, k, temp, *b1, *a1, *c;

    for (i = 0; i < m; ++i)
    {
        a1 = *(A + i) + aoffset;
        c = *(dest + i) + doffset;

        for (j = 0; j < p; ++j)
        {
            b1 = *(B + j) + boffset;
            temp = *(c + j);
            n = n - 16;
            for (k = 0; k < n; k += 16)
            {
                a1 += k;
                b1 += k;
                temp += (*(a1)) * (*(b1)) + (*(a1 + 1)) * (*(b1 + 1)) + (*
(a1 + 2)) * (*(b1 + 2)) +
                        (*(a1 + 3)) * (*(b1 + 3)) + (*(a1 + 4)) * (*(b1 +
4)) + (*(a1 + 5)) * (*(b1 + 5)) +
                        (*(a1 + 6)) * (*(b1 + 6)) + (*(a1 + 7)) * (*(b1 +
7)) + (*(a1 + 8)) * (*(b1 + 8)) +
                        (*(a1 + 9)) * (*(b1 + 9)) + (*(a1 + 10)) * (*(b1 +
10)) + (*(a1 + 11)) * (*(b1 + 11)) +
                        (*(a1 + 12)) * (*(b1 + 12)) + (*(a1 + 13)) * (*(b1
+ 13)) + (*(a1 + 14)) * (*(b1 + 14)) +
                        (*(a1 + 15)) * (*(b1 + 15));
                a1 -= k;
                b1 -= k;
            }
```

```
                n = n + 16;
                for (; k < n; k++)
                    temp += (*(a1 + k)) * (*(b1 + k));
                *(c + j) = temp;
            }
        }
    }
```

9.Split Matrices instead of loops

This was the worst idea I had. I decided to split the matrices of the chain into different threads so that concurrent multiplications occur. But this code was having a lot worse performance. This was because of the following reasons:

- This took away the advantage of the transpose as now multiple matrices were being loaded into cache. Hence we get less cache for each matrix and thus more misses.
- It produces much lesser scope as it requires cache eviction much more and context switches. So to me this was the worst idea. The code snippet for this implementation is as given below:

```
Matrix *MATRIX_CHAIN_MULTIPLY(int i, int j)
{
    if (j == i + 1)
        return matrix_multiply(&matrices[i - 1], &matrices[j - 1]);
    if (i == j)
        return &matrices[i - 1];
    Matrix *B1, *B2;
    omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel sections
    {
#pragma omp section
        B1 = MATRIX_CHAIN_MULTIPLY(i, *(*(s + i) + j));
#pragma omp section
        B2 = MATRIX_CHAIN_MULTIPLY(*(*(s + i) + j) + 1, j);
    }
    return matrix_multiply(B1, B2);
}
```

Again here I am using the `sections` directive mentioned above.

Conclusion

My submission is the fasted code I could achieve under the given constraints and compilation flags and the code is also optimised for Cache and also for memory. The performance gain over the previous assignmnet is 2x. My code was already well optimised and was the best in the last assignment. Thus a 2x improvement is overall good. If the constraint over compilation flags are removed I think there is more scope with Intel Instrinsics. Also in the future as vector registers become more and more common in CPUs with higher capacities problems of such sorts could be solved faster. The original code I started off with was algorithmically solid and hence only needed some small minor changes. Another thing i would like to explore is Strassen's Algorithm but I didnt code it for these two assigns since on my research the algorithm performs poorly on matrices of dimension less than 2kx2k due to overheads in recursion. The assignment specifically specifies size of 1kx1k at max. Hence I didn't try it. In the future for higher constrained problems in matrix multiplication this could be a very good solution along with OMP. The best I was able to produce was Transposing Second Matrix, OMP Loop Paralization with Cache Prefetching along with all optimisations from assignment 1.