

SPP Assignment 1

REPORT

Submitted by

T.H.Arjun , 2019111012 , CSD

Question 0

1. perf

- **perf** is a profiler tool for Linux 2.6+ based systems. It generates a map where time is spent in our code based on sampling. Sampling is when it does not inject modifications to our code but uses system functions to find bottleneck areas of our program in terms of performance
- **perf record <program_name> <program_arguments>** creates a record
- **perf report** shows the reports
- Run these to produce Flame Graphs
 - **sudo perf script -f |<path_to_Flame_graph>/stackcollapse-perf.pl > out.perf-folded**
 - **<path_to_Flame_graph>/flamegraph.pl out.perf-folded > perf-1.svg**
- **perfstat[<options>] [<command>] [ARGS]** is the general usage for a command.

An illustration of perf for submitted code-

```
> sudo perf stat ./5<sample/Q1/90.txt >90.txt

Performance counter stats for './5':

      2,041.44 msec task-clock           #    0.752 CPUs utilized
         9,108      context-switches    #    0.004 M/sec
           0        cpu-migrations      #    0.000 K/sec
        13,732      page-faults         #    0.007 M/sec
<not supported>    cycles
<not supported>    instructions
<not supported>    branches
<not supported>    branch-misses

      2.715427194 seconds time elapsed

      2.089065000 seconds user
      0.059167000 seconds sys
```

2.cache grind

- Usage : **valgrind --tool = cachegrind <command>**
- Cachegrind is a tool for doing cache simulations and annotating your source line-by-line with the number of cache misses. In particular, it records:
 - L1 instruction cache reads and misses;
 - L1 data cache reads and read misses, writes and write misses;
 - L2 unified cache reads and read misses, writes and writes misses.

- Cache grind for submitted code:

```
> valgrind --tool=cachegrind ./5<sample/Q1/90.txt > 90.txt
==27642== Cachegrind, a cache and branch-prediction profiler
==27642== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==27642== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==27642== Command: ./5
==27642==
--27642-- warning: L3 cache found, using its data for the LL simulation.
--27642-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--27642-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==27642==
==27642== I   refs:      25,171,117,046
==27642== I1 misses:      1,436
==27642== LLi misses:      1,422
==27642== I1 miss rate:      0.00%
==27642== LLi miss rate:      0.00%
==27642==
==27642== D   refs:      7,319,825,372 (6,810,001,499 rd + 509,823,873 wr)
==27642== D1 misses:      377,776,548 ( 376,900,685 rd +   875,863 wr)
==27642== LLd misses:      1,627,949 (   752,167 rd +   875,782 wr)
==27642== D1 miss rate:      5.2% (   5.5% +   0.2% )
==27642== LLd miss rate:      0.0% (   0.0% +   0.2% )
==27642==
==27642== LL refs:      377,777,984 ( 376,902,121 rd +   875,863 wr)
==27642== LL misses:      1,629,371 (   753,589 rd +   875,782 wr)
==27642== LL miss rate:      0.0% (   0.0% +   0.2% )

> /mnt/Desktop/SPP1/q1
```

3. gprof

- **gprof** is GNU profiler. It produces an execution profile of C, Pascal, or Fortran77 programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file.
- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This can help in rewriting bottle neck areas
- Usage: **gprof [options] [executable-file [profile-data-files...]] [> outfile]**
- **How it works?**
 - **gprof** calculates the amount of time spent in each function call. Next, these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle.
 - Two modes are available:
 - flat profile : where each function and it's time is shown, this can help to find bottle necks.
 - call graph: which shows function calls and times spend.
 - The code needs to be compiled with **-pg** flag to enable profiling.
 - **-b** flag can be used with **gprof** to supress output.
 - Example:

```

gcc -O0 -pg -o 5 5.c
./5<sample/01/90.txt >90.txt
gprof -b 5 gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
   %   cumulative   self           self      total
time  seconds  seconds   calls   s/call   s/call   name
100.41    1.60    1.60         3     0.53     0.53  matrix_multiply
  0.00    1.60    0.00         1     0.00     1.60  MATRIX_CHAIN_MULTIPLY
  0.00    1.60    0.00         1     0.00     0.00  MATRIX_CHAIN_ORDER

Call graph

granularity: each sample hit covers 2 byte(s) for 0.63% of 1.60 seconds

index % time   self  children   called    name
-----
[1]   100.0     1.60    0.00     3/3      MATRIX_CHAIN_MULTIPLY [2]
      1.60    0.00     3       4      matrix_multiply [1]
-----
      0.00    1.60     1/1      MATRIX_CHAIN_MULTIPLY [2]
[2]   100.0     0.00    1.60     1+4      main [3]
      0.00    1.60     3/3      MATRIX_CHAIN_MULTIPLY [2]
      1.60    0.00     4      matrix_multiply [1]
      0.00    0.00     4      MATRIX_CHAIN_MULTIPLY [2]
-----
[3]   100.0     0.00    1.60     1/1      <spontaneous>
      0.00    1.60     1/1      main [3]
      0.00    0.00     1/1      MATRIX_CHAIN_MULTIPLY [2]
      0.00    0.00     1/1      MATRIX_CHAIN_ORDER [4]
-----
      0.00    0.00     1/1      main [3]
[4]    0.0     0.00    0.00     1       MATRIX_CHAIN_ORDER [4]
-----

Index by function name

[2] MATRIX_CHAIN_MULTIPLY  [4] MATRIX_CHAIN_ORDER  [1] matrix_multiply

```

4.clock_gettime

Need to include it as:

```

#include<time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp);

```

The functions `clock_gettime()` retrieves the time of the specified clock `clk_id`.

`timespec` has following structure...

```

struct timespec {
    time_t    tv_sec;        /* seconds */
    long      tv_nsec;       /* nanoseconds */
};

```

Example code:

```

#include <time.h>

main()
{
    int rc;
    long i;

```

```
struct timespec ts;

for(i=0; i<100000000; i++) {
    rc = clock_gettime(CLOCK_MONOTONIC, &ts);
}
}
```

While compiling it should be linked to lrt library

```
$gcc clock_timing.c -o clock_timing -lrt
```

It supports various clock types like:

- CLOCK_REALTIME
- CLOCK_MONOTONIC
- CLOCK_PROCESS_CPUTIME_ID
- CLOCK_THREAD_CPUTIME_ID

Code execution can be timed like:

```
clock_gettime(CLOCK_MONOTONIC, &start);
//function call here
clock_gettime(CLOCK_MONOTONIC, &end);
```

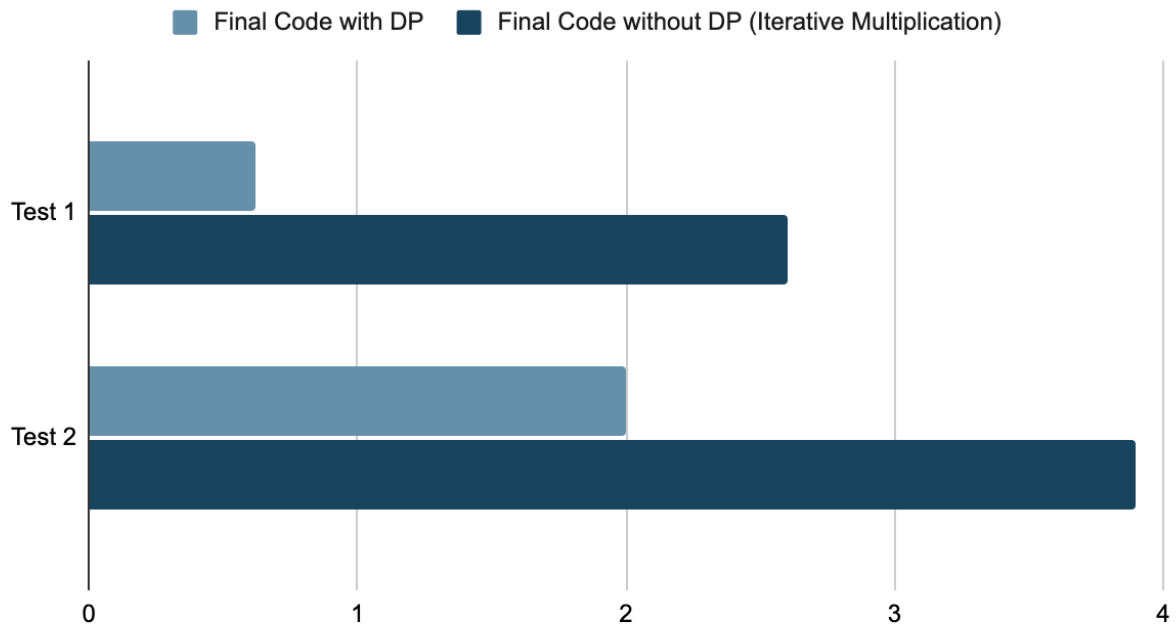
Question 1: Matrix Chain Multiplication

1.1 Algorithmic Optimisation

One of the optimisations in matrix chain multiplication is reducing number of operations by exploiting the associativity of matrices. This can be done as an $O(n^3)$ problem where n is number of matrices. It uses dynamic programming to do the same. A full detail of the algorithm is available at [GFG](#)

The number of operations can reduce by a lot such that the DP overhead is very negligible compared to the performance gain for rectangle matrices. The given testcases doesnt give justice as they are all squares of same size. But I tested on my own test cases. A graph of which is shown below.

Timings



It is very clear that the DP is worth it case of such matrices... I hope the final test cases have such xD.

1.2 Transpose the Second Matrix

Prove the chances of getting cache hit, as the neighbouring elements will now be accessed versus accessing different columns of the different matrices. This is the best Performance I recieved. The transpose is done by `Matrix *transpose(a)` This function is highly optimised with blocking to get efficiency in cache while doing transpose.

```
Matrix *transpose(Matrix *a)
{
    register int N = a->n;
    register int M = a->m;
    register int n, m;
    register int row, col;
    int d_val = 0;
    int diag = 0;
    Matrix *result = (Matrix *)malloc(sizeof(Matrix));
    result->matrix = (int *)malloc(N * M * sizeof(int));
    result->n = M;
    result->m = N;
    register int *A = a->matrix;
    register int *B = result->matrix;
    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;
}

```

1.3 Loop Unrolling

This is an optimisation that compilers do. The Condition check and increments of loop are costly. So Compilers unroll the loop to improve performance. Since the assignment does not allow compiler optimisation I did the unrolling by hand. The best performance I got is for factor 16.

```

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;
```

1.4 Loop Hoisting

This is another optimisation that the compiler does that I did by hand here. Some things remain invariant throughout a loop. These calculations can be reduced by storing them in variables. eg: `a1` and `b1` remain constant throughout the loop in above code.

1.5 Space Optimisation

Space is optimised by using dynamic memory allocation for both the problems.

1.6 Further Small Optimisations

- Use of `register` keyword
- Use of `++i`, `--i` instead of `i++`, `i--` since they store their previous values to return hence the later is slow. -Pointer accessing to memory

```
*(A + i); // Preferred  
A[i];     // Slower on some compilers
```

- Storing in 1D array: The best optimisation in terms of time, as now only 1D arrays are used instead of 2D referencing which is most time consuming.

Some Failed Attempts/ Worse Performances of Optimisation

- Tiling: Didn't reduce the time by much, the parameter is very tough to tune.

```
void matrix_mult_wiki_block(int *A, int *B, int *C,  
                             const int N, const int M, const int K)  
{  
    const int block_size = 64 / sizeof(int);  
    for (int i = 0; i < N; i++)  
    {  
        for (int j = 0; j < K; j++)  
        {  
            C[K * i + j] = 0;  
        }  
    }  
    for (int i0 = 0; i0 < N; i0 += block_size)  
    {  
        int imax = i0 + block_size > N ? N : i0 + block_size;  
  
        for (int j0 = 0; j0 < M; j0 += block_size)  
        {
```

```

    int jmax = j0 + block_size > M ? M : j0 + block_size;

    for (int k0 = 0; k0 < K; k0 += block_size)
    {
        int kmax = k0 + block_size > K ? K : k0 + block_size;

        for (int j1 = j0; j1 < jmax; ++j1)
        {
            int sj = M * j1;

            for (int i1 = i0; i1 < imax; ++i1)
            {
                int mi = M * i1;
                int ki = K * i1;
                int kij = ki + j1;

                for (int k1 = k0; k1 < kmax; ++k1)
                {
                    C[kij] += A[mi + k1] * B[sj + k1];
                }
            }
        }
    }
}

```

- Recurse to Cache Size Sub Matrices and then do the transpose technique. That is when the matrices start fitting in cache, change to current technique.

```

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;
        multiply(A, B, split, n, p, dest, doffset, aoffset, boffset);
        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;
    multiply(A, B, m, split, p, dest, doffset, aoffset, boffset);
    multiply(A, B, m, rest, p, dest, doffset, aoffset + split, boffset
+ split);
}
else
{

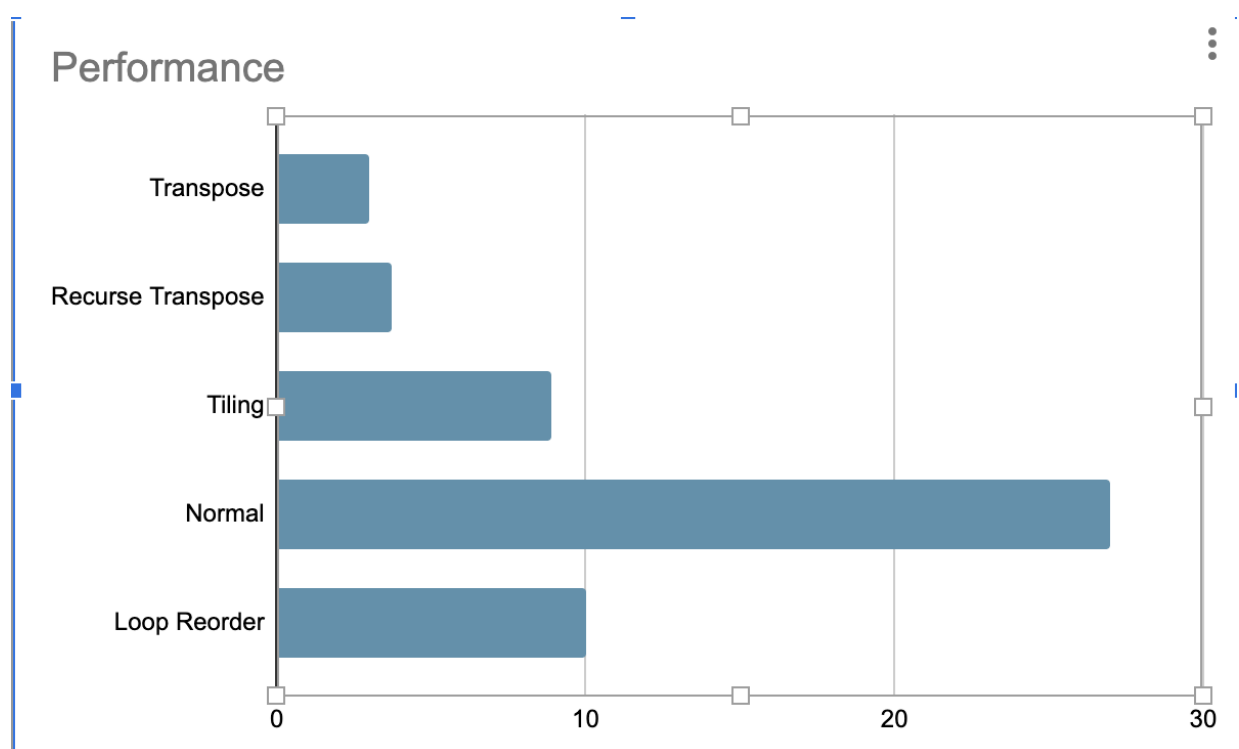
    int split = p >> 1;
    int rest = p - split;
    multiply(A, B, m, n, split, dest, doffset, aoffset, boffset);
    multiply(A, B + split, m, n, rest, dest, doffset + split, aoffset,
boffset);
}
}

```

This actually slowed the program due to recursive calls but cache misses reduced.

- Cache Prefetching: GCC provides a cool method to prefetch data to cache by software using `__builtin_prefetch`. You can instruct to load data at an address to reduce cache misses. I tried doing it but it reduced performance due to function calls and it seems the processor is better than me at this job.
- Branch Prediction: Again gcc provides a cool feature `__builtin_expect_with_probability` which can hint the compiler about branching. I tried hinting but guess my predictions were wrong. I also used profiling to predict branches but it varied a lot between test cases. So I decided its better not to use it.
- Changing Loop Order: This was one of the first things done as it is the fastest way to optimise matrix multiplication but transpose outperformed.

Performances of Various Verions:



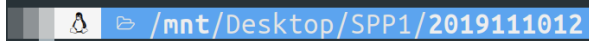
Ouputs of submitted code:

```
> sudo perf stat ./1<../it2/1/sample/Q1/90.txt>1.txt
Performance counter stats for './1':

      2,287.79 msec task-clock              #    0.618 CPUs utilized
        9,109      context-switches        #    0.004 M/sec
           3       cpu-migrations          #    0.001 K/sec
       17,642      page-faults             #    0.008 M/sec
<not supported>    cycles
<not supported>    instructions
<not supported>    branches
<not supported>    branch-misses

      3.700616351 seconds time elapsed

      2.317034000 seconds user
      0.222112000 seconds sys
```

 /mnt/Desktop/SPP1/2019111012

```
> gprof -b 1 gmon.out
Flat profile:
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
97.92	1.61	1.61	3	0.54	0.54	matrix_multiply
1.22	1.63	0.02	3	0.01	0.01	transpose
1.22	1.65	0.02				main
0.00	1.65	0.00	1	0.00	1.63	MATRIX_CHAIN_MULTIPLY
0.00	1.65	0.00	1	0.00	0.00	MATRIX_CHAIN_ORDER

Call graph

granularity: each sample hit covers 2 byte(s) for 0.61% of 1.65 seconds

index	% time	self	children	called	name
<spontaneous>					
[1]	100.0	0.02	1.63		main [1]
		0.00	1.63	1/1	MATRIX_CHAIN_MULTIPLY [3]
		0.00	0.00	1/1	MATRIX_CHAIN_ORDER [5]

[2]	98.8	1.61	0.02	3/3	MATRIX_CHAIN_MULTIPLY [3]
		1.61	0.02	3	matrix_multiply [2]
		0.02	0.00	3/3	transpose [4]

[3]	98.8	0.00	1.63	4	MATRIX_CHAIN_MULTIPLY [3]
		0.00	1.63	1/1	main [1]
		1.61	0.02	1+4	MATRIX_CHAIN_MULTIPLY [3]
				3/3	matrix_multiply [2]
				4	MATRIX_CHAIN_MULTIPLY [3]

[4]	1.2	0.02	0.00	3/3	matrix_multiply [2]
		0.02	0.00	3	transpose [4]

[5]	0.0	0.00	0.00	1/1	main [1]
		0.00	0.00	1	MATRIX_CHAIN_ORDER [5]

Index by function name

[3] MATRIX_CHAIN_MULTIPLY	[1] main	[4] transpose
[5] MATRIX_CHAIN_ORDER	[2] matrix_multiply	

```

> valgrind --tool=cachegrind ./1< /mnt/Desktop/SPP1/q2/sample/Q1/20.txt >3.txt
==56311== Cachegrind, a cache and branch-prediction profiler
==56311== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==56311== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==56311== Command: ./1
==56311==
--56311-- warning: L3 cache found, using its data for the LL simulation.
--56311-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--56311-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==56311==
==56311== I   refs:      1,161,621,805
==56311== I1 misses:      1,476
==56311== LLi misses:      1,442
==56311== I1 miss rate:      0.00%
==56311== LLi miss rate:      0.00%
==56311==
==56311== D   refs:      358,343,105 (303,156,622 rd + 55,186,483 wr)
==56311== D1 misses:      13,796,284 ( 13,648,036 rd + 148,248 wr)
==56311== LLd misses:      126,609 (    2,136 rd + 124,473 wr)
==56311== D1 miss rate:      3.9% (    4.5% + 0.3% )
==56311== LLd miss rate:      0.0% (    0.0% + 0.2% )
==56311==
==56311== LL refs:      13,797,760 ( 13,649,512 rd + 148,248 wr)
==56311== LL misses:      128,051 (    3,578 rd + 124,473 wr)
==56311== LL miss rate:      0.0% (    0.0% + 0.2% )

```

```

> /mnt/Desktop/SPP1/2019111012

```

Question 2: Floyd Warshall

1.1 Algorithmic Optimisation

I implemented recursive Floyd Warshall for final submission. I read a research paper. It recurses down till the CACHE size. And when all matrices fit in cache it turns to normal floyd warshall.

```

void FW(int *A, int *B, int *C, int N, int dm, int dn, int dp)
{
    if ((dm * dp + dm * dn + dn * dp) < 8192)
    {
        register int k, i, j, bik, *aij, temp, *ck, *ai;
        for (k = 0; k < dn; ++k)
        {
            ck = C + k * N;

            for (i = 0; i < dm; ++i)
            {
                bik = *(B + (i * N + k));

                if (bik == 1000000007)
                    continue;
                ai = A + i * N;

                dp = dp - 16;
                for (j = 0; j < dp; j += 16)
                {
                    ck += j;

```

```
temp = bik + *(ck);
aij = ai + j;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 1);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 2);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 3);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 4);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 5);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 6);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 7);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 8);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 9);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 10);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 11);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 12);
++aij;
if (*aij > temp)
    *aij = temp;
temp = bik + *(ck + 13);
++aij;
```

```

        if (*aij > temp)
            *aij = temp;
        temp = bik + *(ck + 14);
        ++aij;
        if (*aij > temp)
            *aij = temp;
        temp = bik + *(ck + 15);
        ++aij;
        if (*aij > temp)
            *aij = temp;

        ck -= j;
    }
    dp = dp + 16;
    for (; j < dp; ++j)
    {
        temp = bik + *(ck + j);
        aij = ai + j;
        if (*aij > temp)
            *aij = temp;
    }
}
}
else
{
    int dm_dash = dm >> 1, dn_dash = dn >> 1, dp_dash = dp >> 1;
    FW(A, B, C, N, dm_dash, dn_dash, dp_dash);

    FW(A + dp_dash, B, C + dp_dash, N, dm_dash, dn_dash, dp -
dp_dash);

    FW(A + dm_dash * N, B + dm_dash * N, C, N, dm - dm_dash, dn_dash,
dp_dash);

    FW(A + (dm_dash)*N + (dp_dash), B + (dm_dash)*N, C + dp_dash, N,
dm - dm_dash, dn_dash, dp - dp_dash);

    FW(A + (dm_dash)*N + (dp_dash), B + (dm_dash)*N + (dn_dash), C +
(dn_dash)*N + (dp_dash), N, dm - dm_dash, dn - dn_dash, dp - dp_dash);

    FW(A + (dm_dash)*N, B + (dm_dash)*N + (dn_dash), C + (dn_dash)*N,
N, dm - dm_dash, dn - dn_dash, dp_dash);

    FW(A + dp_dash, B + dn_dash, C + (dn_dash)*N + (dp_dash), N,
dm_dash, dn - dn_dash, dp - dp_dash);

    FW(A, B + dn_dash, C + (dn_dash)*N, N, dm_dash, dn - dn_dash,
dp_dash);
}
}

```

If `dist[k][j]` is INF we don't need to do the inner loop. The constraints are such that this condition is mostly true. ie. 2500 vertices and at max 1e5 edges. This optimisation increases speed by a lot.

The recursive floyd warshall decreases time by 1s and reduces cache miss from 2.9% to 0.4% which is a lot of saves.

1.2 Loop Unrolling

Done here also

1.3 Loop Hoisting

Done here also

1.4 Space Optimisation

Space is optimised by using dynamic memory allocation for both the problems.

1.5 Futher Small Optimisations

- Use of `register` keyword
- Use of `++i`, `--i` instead of `i++`, `i--` since they store their previous vlues to return hence the later is slow. -Pointer accessing to memory

```
*(A + i); // Preffered
A[i];     // Slower on some compilers
```

- Storing in 1D array: The best optimisation in terms of time, as now only 1D arrays are used instead of 2D referencing which is most time consuming.

Some Failed Attempts/ Worse Performaces of Optimisation

- Normal Floyd Warshall performed worse compared to Recursive.

```
for (k = 1; k <= v; ++k)
{
    d = *(dist + k);
    for (i = 1; i <= v; ++i)
    {
        if (i == k)
            continue;

        ik = (*(dist + i) + k);
        if (ik == 1000000007)
            continue;

        c = *(dist + i);
        v = v - 16;
```

```
for (j = 1; j <= v; j += 16)
{
    a = c + j;
    b = d + j;
    temp = ik + *(b);
    if (temp < *(a))
        *(a) = temp;
    temp = ik + *(b + 1);
    if (temp < *(a + 1))
        *(a + 1) = temp;
    temp = ik + *(b + 2);
    if (temp < *(a + 2))
        *(a + 2) = temp;
    temp = ik + *(b + 3);
    if (temp < *(a + 3))
        *(a + 3) = temp;
    temp = ik + *(b + 4);
    if (temp < *(a + 4))
        *(a + 4) = temp;
    temp = ik + *(b + 5);
    if (temp < *(a + 5))
        *(a + 5) = temp;
    temp = ik + *(b + 6);
    if (temp < *(a + 6))
        *(a + 6) = temp;
    temp = ik + *(b + 7);
    if (temp < *(a + 7))
        *(a + 7) = temp;
    temp = ik + *(b + 8);
    if (temp < *(a + 8))
        *(a + 8) = temp;
    temp = ik + *(b + 9);
    if (temp < *(a + 9))
        *(a + 9) = temp;
    temp = ik + *(b + 10);
    if (temp < *(a + 10))
        *(a + 10) = temp;
    temp = ik + *(b + 11);
    if (temp < *(a + 11))
        *(a + 11) = temp;
    temp = ik + *(b + 12);
    if (temp < *(a + 12))
        *(a + 12) = temp;
    temp = ik + *(b + 13);
    if (temp < *(a + 13))
        *(a + 13) = temp;
    temp = ik + *(b + 14);
    if (temp < *(a + 14))
        *(a + 14) = temp;
    temp = ik + *(b + 15);
    if (temp < *(a + 15))
        *(a + 15) = temp;
}
v = v + 16;
```



```

        for (; j <= v; ++j)
        {
            temp = ik + *(d + j);
            if (temp < *(c + j))
                *(c + j) = temp;
        }
    }
}

```

- Branch Prediction and Pre Fetching Failures

Profiling outputs:

```

> valgrind --tool=cachegrind ./2< /mnt/Desktop/SPP1/q2/sample/Q2/t29 >1.txt
==56250== Cachegrind, a cache and branch-prediction profiler
==56250== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==56250== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==56250== Command: ./2
==56250==
--56250-- warning: L3 cache found, using its data for the LL simulation.
--56250-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--56250-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==56250==
==56250== I   refs:      106,208,284,751
==56250== I1  misses:      1,424
==56250== LLi misses:      1,409
==56250== I1  miss rate:      0.00%
==56250== LLi miss rate:      0.00%
==56250==
==56250== D   refs:      28,188,700,538 (27,627,403,280 rd + 561,297,258 wr)
==56250== D1  misses:      53,674,505 ( 53,362,077 rd + 312,428 wr)
==56250== LLd misses:      1,355,149 ( 1,043,850 rd + 311,299 wr)
==56250== D1  miss rate:      0.2% ( 0.2% + 0.1% )
==56250== LLd miss rate:      0.0% ( 0.0% + 0.1% )
==56250==
==56250== LL refs:      53,675,929 ( 53,363,501 rd + 312,428 wr)
==56250== LL misses:      1,356,558 ( 1,045,259 rd + 311,299 wr)
==56250== LL miss rate:      0.0% ( 0.0% + 0.1% )

```

```
> gprof -b 2 gmon.out
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.86	7.39	7.39	1	7.39	7.39	FW
0.27	7.41	0.02				main

Call graph

granularity: each sample hit covers 2 byte(s) for 0.13% of 7.41 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.02	7.39		main [1]
		7.39	0.00	1/1	FW [2]

				299592	FW [2]
		7.39	0.00	1/1	main [1]
[2]	99.7	7.39	0.00	1+299592	FW [2]
				299592	FW [2]

Index by function name

[2] FW [1] main

```
/mnt/Desktop/SPP1/2019111012
```

```
> sudo perf stat ./1< /mnt/Desktop/SPP1/q2/sample/Q1/20.txt >3.txt
```

Performance counter stats for './1':

140.21 msec	task-clock	#	0.565 CPUs utilized
935	context-switches	#	0.007 M/sec
1	cpu-migrations	#	0.007 K/sec
1,991	page-faults	#	0.014 M/sec
<not supported>	cycles		
<not supported>	instructions		
<not supported>	branches		
<not supported>	branch-misses		

0.248280996 seconds time elapsed

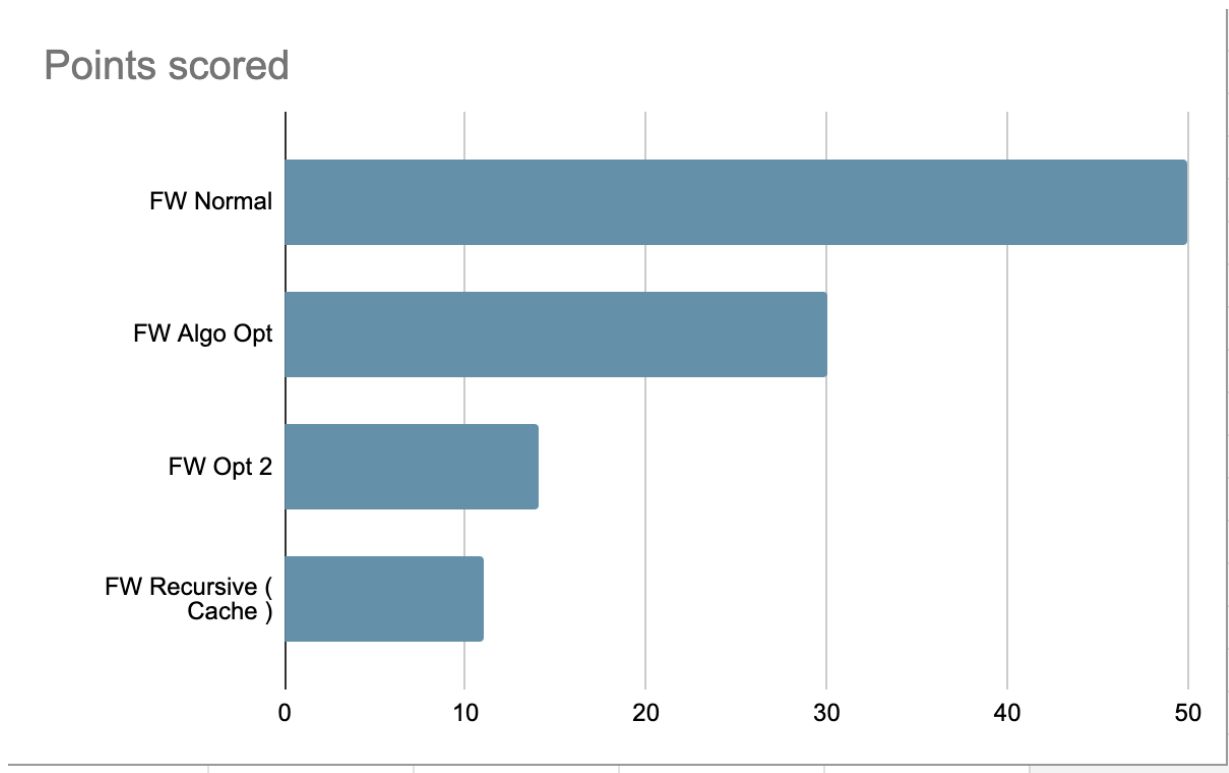
0.139435000 seconds user

0.030312000 seconds sys

```
/mnt/Desktop/SPP1/2019111012
```

Plot:

Points scored



The real advantage of the recursive method comes in dense graphs, where the cache misses can be reduced by a lot using recursive method.