Optimizing Techniques

`Ayush Sharma (2019101004)`

Demonstrating the techniques that can be used to improve the performance of a program using example of Matrix Chain Multiplication and Floyd Warshal Algorithm. Used perf, cache grind, gprof, and clock_gettime for profiling of all techniques tried during optimization.

- C language has been used.
- Code during optimization were run with -O0 flag. i.e. zero compiler optimization
- Timings during optimization were calculated on local pc with conf. [Ubuntu 18.04.5 LTS, Intel® Core™ i5-8265U CPU @ 1.60GHz × 8 ,Intel® UHD Graphics 620 (WHL GT2), 64-bit]. After the best possible optimization was done, I profiled the final code on abacus server provided to us (and will attach the ss in the last of each algo).

Profiling Tools

Perf

`perf` is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple commandline interface. The perf tool offers a rich set of commands to collect and analyze performance and trace data.

USAGE : perf [--version] [--help] COMMAND [ARGS]

Some of the general supported commands:

> - stat : Used to run a command and obtain performance counter statistics on it.
> - record : Used to record the profile of a cmmand in perf.data file.
> - report : It read perf.data and display the corresponding profile
> - list : It will list all the symbolic event types

Some of the general supported options:

> - -a : Used for tracing system wide performance including the CPUs.
> - -e : Selects events from the list which can be obtained by doing `perf list`.
> - -p : It stats and trace the events on the provide Process ID.
> - -t : It stats and trace the events on the provide Thread ID.

Cachegrind

The main objective of using `cachegrind` was to analyse the cache usage of our working code. It is a cache and branch-prediction profiler. Optimizing for cache hits is a critical part of perfomance engineering. Therefore, cachegrind interact with the machine [by running the code in debug mode i.e. with `-g` flag for our `C` code in a Sandboxed environment] and simulates caching i.e. profile total cache reads, cache hits and cache misses, at each level of memory storage.

Therefore, 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 (I1)
> - L1 data cache reads and read misses, writes and write misses (D1)
> - L2 unified cache reads and read misses, writes and writes misses (LL)

USAGE : valgrind --tool=cachegrind

Gprof

`gprof` is a GNU profiler, which collects and arranges statistics on the program. It produces an execution profile of C, Pascal, or Fortran77 programs. It helps in knowing the time duration of each part of the running program which will help us to understand the frequency of calling a function and pointing out some bugs along with slower part of the code whch needs optimization. Gprof calculates the amount of time spent in each routine. 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.

Outputs Available:

> - The flat profile shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which functions burn most of the cycles, it is stated concisely here.
>
> - The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where you might try to eliminate function calls that use a lot of time.

USAGE :

> - Have profiling enabled while compiling the code, made possible by adding the `-pg` option in the compilation step.
> - Execute the program code to produce the profiling data that will be stored as `gmon.out` in the current working directory.
> - Run the gprof tool on the profiling data file, generated in the step above.

Clock_gettime

It is used to time various part of the program. POSIX is a standard for implementing and representing time sources. In contrast to the hardware clock, which is selected by the kernel and implemented across the system; the POSIX clock can be selected by each application, without affecting other applications in the system.

> - clock_gettime() : The clock_gettime() function gets the current time of the clock specified by clock_id, and puts it into the buffer pointed to by tp.
> - Header File : "time.h".
>
> - Prototype / Syntax : int clock_gettime(clockid_t clock_id, struct timespec *tp);

> - Prototype / Syntax : int clock_gettime(clockid_t clock_id, struct timespec *tp);
>
> - Return Value : return 0 for success, or -1 for failure.
>
> - clock_id : clock id = CLOCK_REALTIME,
> - CLOCK_PROCESS_CPUTIME_ID, CLOCK_MONOTONIC … etc.
> - CLOCK_REALTIME : clock that measures real i.e., wall-clock) time.
> - CLOCK_PROCESS_CPUTIME_ID : High-resolution per-process timer from the CPU.
> - CLOCK_MONOTONIC : High resolution timer that is unaffected by system date changes (e.g. NTP daemons).
> - tp parameter points to a structure containing atleast the following members :

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

USAGE :

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

Matrix Chain Multiplication

Matrix chain multiplication (or the matrix chain ordering problem[citation needed]) is an optimization problem concerning the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved and then multiplying matrices obtained in the correct order.

Optimizations

I targeted optimizing multiplication of two matrices O(N^3) instead of the optimizing the matrix chain ultiplication algorithm. We were given 15 test cases out of which the largest was having `4` matrices of size `1000*1000`. Although I profiled all of them but will list timings and cache profile for that largest test case only.

V1

- *Time taken* with the trivial algorithm i.e. no optimisation : `~33 seconds`

```
 for (int i = 0; i < p; ++i)
{
    for(int j = 0; j < r; ++j)
    {
        for  (int k = 0; k < q; ++k)
            result->matrix[i][j] += (a->matrix[i][k] * b->matrix[k][j]);
    }
}
```

V2

- Changed the order of loop from `ijk` to `ikj`. This improved speed to 3x. This happens because matrix B has poor spatial locality for cache in the case of i, j, k. The order i, k, j gives good spatial locality for the resultant matrix, good temporal locality for matrix A (accesses same element until j loop is completed), and good spatial locality for matrix B.

```
 for (int i = 0; i < p; ++i)
{
    for (int k = 0; k < q; ++k)
    {
        for (int j = 0; j < r; ++j)
            result->matrix[i][j] += (a->matrix[i][k] * b->matrix[k][j]);
    }
}
```

Run Time was `~9.5s`.

V3

- Instead of chaning the order of loop, I took the transpose of the matrix B. That will still be having good use of spatial locality and same degree of cache misses.
- Using temporary variables instead of writing to memory gave a significant speed up as the runtime became. This occurs because instead of accessing memory every time, we just change a local variable, and change the value in memory at the end.

- Speed incresed due to this by 2s.

```
Matrix * multiply_matrix_v3(Matrix * a, Matrix * b,int p,int q, int r)
{
    Matrix *result = malloc(sizeof(Matrix));

    for (int i = 0; i < p; ++i)
    {
        for (int j = 0; j < r; ++j)
            result->matrix[i][j] =0;
    }

    transpose_matrix(b);



    for (int i = 0; i < p; ++i)
    {
        for(int j = 0; j < r; ++j)
        {
            register long long int temp = 0;
            for  (int k = 0; k < q; ++k)
                temp += (a->matrix[i][k] * b->matrix[j][k]);
            result->matrix[i][j] = temp;
        }
    }
    return result;
}
```

Run time was `~7s`.

V4

- Finally I tried the approach of Blocking/Tilling of a matrix. This uses i,j,k of the outer loop , and multiply the smaller matrices generated by the loops inside.
- Introduced the concept of loop unrolling.
- Used keyword `register` that stores the data in the registers of the processor. And reduce the time in fetching certain values from the main memory.
- Few hit and trial led to discover Block size = 21 giving the best result.

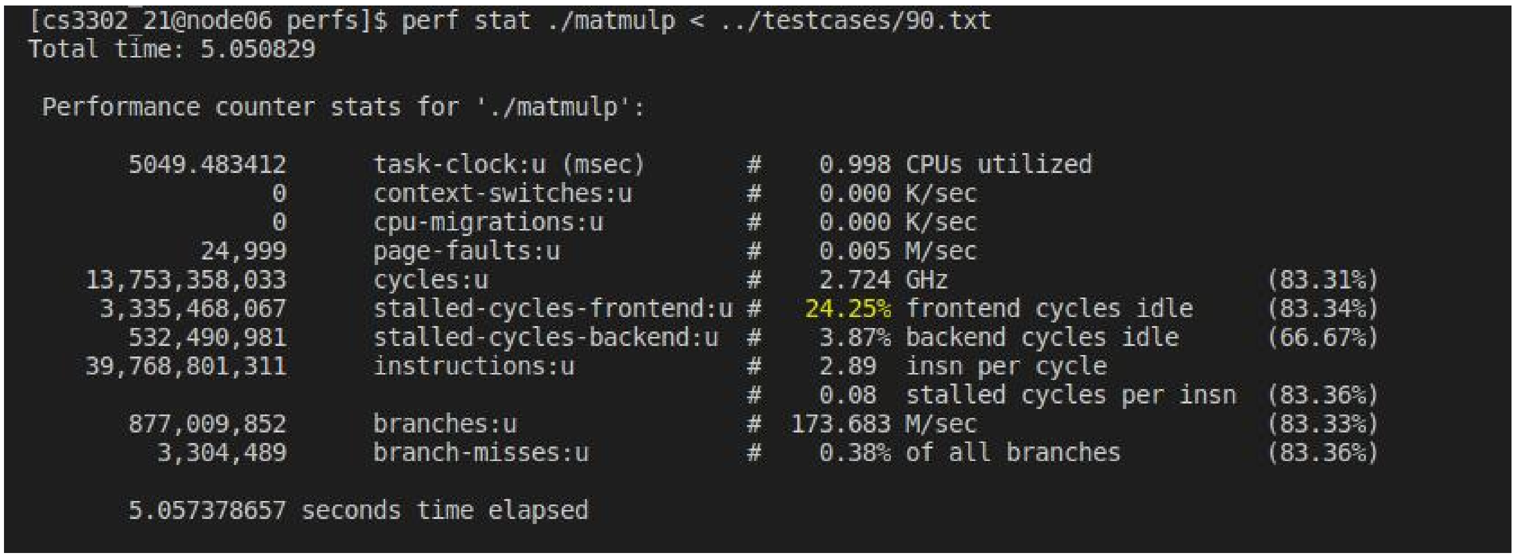
```
 Matrix * multiply_matrix_v4(Matrix * a, Matrix * b,int p,int q, int r)
{
    Matrix *result = malloc(sizeof(Matrix));

    register int ii, jj, kk, BLOCK=21,BLOCK_SUM=512;
    register long long int *aii;
    register long long int *bjj;

    for (ii = 0; ii < p; ++ii)
    {
        for (jj = 0; jj < r; ++jj)
            result->matrix[ii][jj] =0;
    }

    transpose_matrix(b);

    for(int i=0;i<p;i+=BLOCK)
    {
        for(int j=0;j<r;j+=BLOCK)
        {
            for(int k=0;k<q;k+=BLOCK_SUM)
            {
                int min_I = fun_min(p,i+BLOCK);
                for(ii=i;ii<min_I;++ii)
                {
                    aii = a->matrix[ii];
                    register long long int temp = 0;
                    int min_J = fun_min(r,j+BLOCK);
                    for(jj=j;jj<min_J;++jj)
                    {
                        temp = 0;
                        bjj = b->matrix[jj];
                        int min_K = fun_min(q, k+BLOCK_SUM);
                        for(kk=k;kk<min_K-15;kk+=16)
                        {
                            temp += ( (*(aii+kk + 0)) * (*(bjj+kk + 0)) )
                                 ...
                                 +  ( (*(aii+kk + 15)) * (*(bjj+kk + 15)) );
                        }
                        result->matrix[ii][jj]+=temp;
                        register long long int tmp = 0;
                        while(min_K > kk)
                        {
                            tmp += (*(aii + kk)) * (*(bjj + kk));
                            ++kk;
                        }
                        result->matrix[ii][jj]+=tmp;
                    }
                }
            }
        }
    }

    return result;
}
```

Run time was `(3+-0.2) s`

Final Result

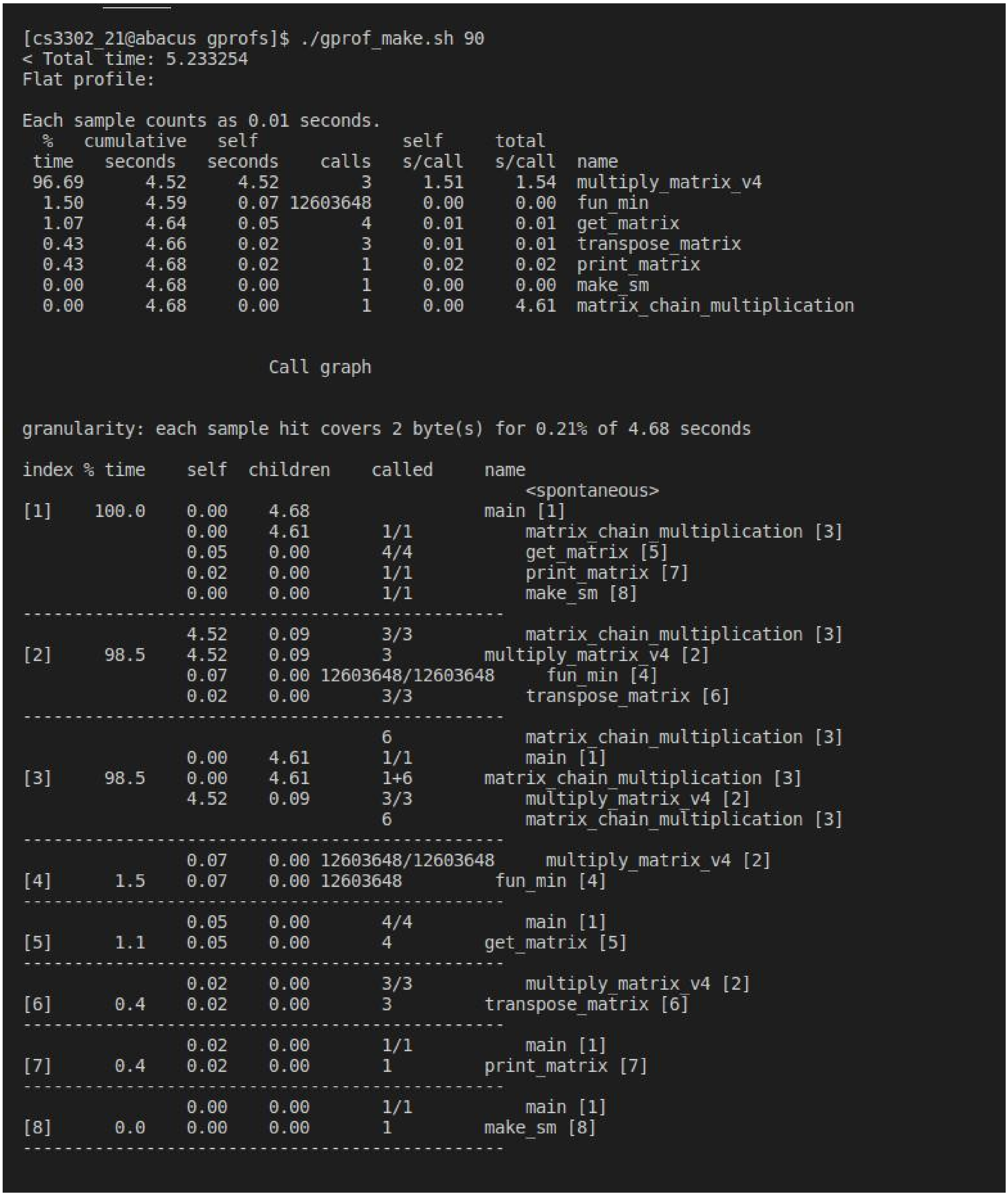
On Abacus we get best running time to be `(5+-0.2)s` with our final code. Profiling on abacus:

Perf Stat on Matrix Chain Multiplication Final Code :



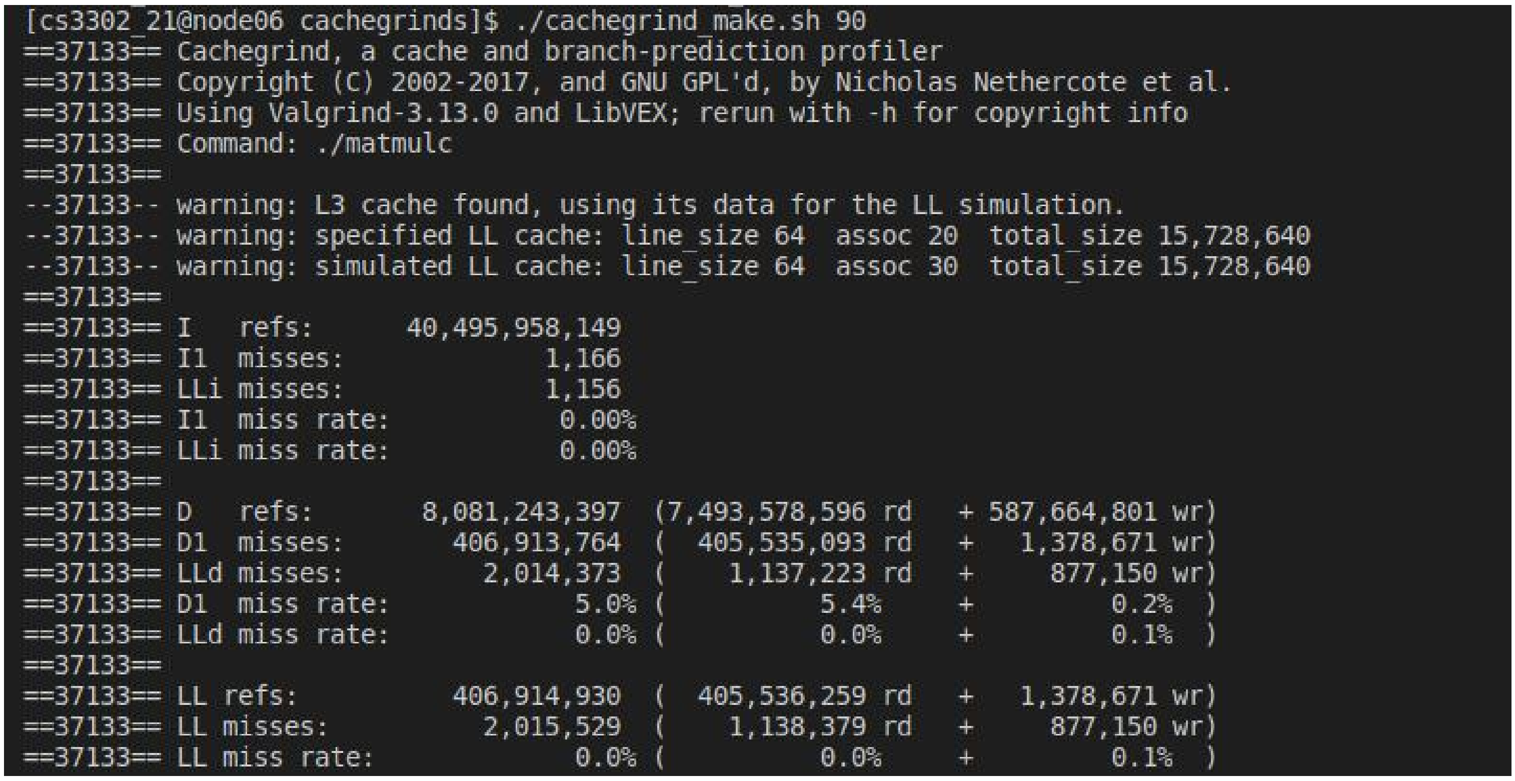
```
[cs3302_21@node06 perfs]$ perf stat ./matmulp < ../testcases/90.txt
Total time: 5.050829

 Performance counter stats for './matmulp':

       5049.483412      task-clock:u (msec)       #    0.998 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
            24,999      page-faults:u             #    0.005 M/sec
    13,753,358,033      cycles:u                  #    2.724 GHz                      (83.31%)
     3,335,468,067      stalled-cycles-frontend:u #   24.25% frontend cycles idle     (83.34%)
       532,490,981      stalled-cycles-backend:u  #    3.87% backend cycles idle      (66.67%)
    39,768,801,311      instructions:u            #    2.89  insn per cycle
                                                  #    0.08  stalled cycles per insn  (83.36%)
       877,009,852      branches:u                #  173.683 M/sec                    (83.33%)
         3,304,489      branch-misses:u           #    0.38% of all branches          (83.36%)

       5.057378657 seconds time elapsed
```

Gprof on Matrix Chain Multiplication Final Code :



```
[cs3302_21@abacus gprofs]$ ./gprof_make.sh 90
< Total time: 5.233254
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 96.69      4.52     4.52        3     1.51     1.54  multiply_matrix_v4
  1.50      4.59     0.07 12603648     0.00     0.00  fun_min
  1.07      4.64     0.05        4     0.01     0.01  get_matrix
  0.43      4.66     0.02        3     0.01     0.01  transpose_matrix
  0.43      4.68     0.02        1     0.02     0.02  print_matrix
  0.00      4.68     0.00        1     0.00     0.00  make_sm
  0.00      4.68     0.00        1     0.00     4.61  matrix_chain_multiplication


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.21% of 4.68 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    4.68                 main [1]
                0.00    4.61       1/1           matrix_chain_multiplication [3]
                0.05    0.00       4/4           get_matrix [5]
                0.02    0.00       1/1           print_matrix [7]
                0.00    0.00       1/1           make_sm [8]
-----------------------------------------------
                4.52    0.09       3/3           matrix_chain_multiplication [3]
[2]     98.5    4.52    0.09       3         multiply_matrix_v4 [2]
                0.07    0.00 12603648/12603648     fun_min [4]
                0.02    0.00       3/3           transpose_matrix [6]
-----------------------------------------------
                                   6             matrix_chain_multiplication [3]
                0.00    4.61       1/1           main [1]
[3]     98.5    0.00    4.61       1+6       matrix_chain_multiplication [3]
                4.52    0.09       3/3           multiply_matrix_v4 [2]
                                   6             matrix_chain_multiplication [3]
-----------------------------------------------
                0.07    0.00 12603648/12603648     multiply_matrix_v4 [2]
[4]      1.5    0.07    0.00 12603648         fun_min [4]
-----------------------------------------------
                0.05    0.00       4/4           main [1]
[5]      1.1    0.05    0.00       4         get_matrix [5]
-----------------------------------------------
                0.02    0.00       3/3           multiply_matrix_v4 [2]
[6]      0.4    0.02    0.00       3         transpose_matrix [6]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[7]      0.4    0.02    0.00       1         print_matrix [7]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[8]      0.0    0.00    0.00       1         make_sm [8]
-----------------------------------------------
```

Cachegrind on Matrix Chain Multiplication Final Code :



```
[cs3302_21@node06 cachegrinds]$ ./cachegrind_make.sh 90
==37133== Cachegrind, a cache and branch-prediction profiler
==37133== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==37133== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==37133== Command: ./matmulc
==37133==
--37133-- warning: L3 cache found, using its data for the LL simulation.
--37133-- warning: specified LL cache: line_size 64  assoc 20  total_size 15,728,640
--37133-- warning: simulated LL cache: line_size 64  assoc 30  total_size 15,728,640
==37133==
==37133== I   refs:      40,495,958,149
==37133== I1  misses:             1,166
==37133== LLi misses:             1,156
==37133== I1  miss rate:           0.00%
==37133== LLi miss rate:           0.00%
==37133==
==37133== D   refs:       8,081,243,397  (7,493,578,596 rd   + 587,664,801 wr)
==37133== D1  misses:       406,913,764  (  405,535,093 rd   +   1,378,671 wr)
==37133== LLd misses:         2,014,373  (    1,137,223 rd   +     877,150 wr)
==37133== D1  miss rate:            5.0% (          5.4%     +         0.2%  )
==37133== LLd miss rate:            0.0% (          0.0%     +         0.1%  )
==37133==
==37133== LL refs:          406,914,930  (  405,536,259 rd   +   1,378,671 wr)
==37133== LL misses:          2,015,529  (    1,138,379 rd   +     877,150 wr)
==37133== LL miss rate:             0.0% (          0.0%     +         0.1%  )
```

Floyd Warshall Algorithm

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights. A single execution of the algorithm will find the lengths of shortest paths between all pairs of vertices.

Optimizations

We were given 10 test cases out of which the largest was having `2229` number of nodes and `155317` number of edges. There can be self loop and multiple edges between same two nodes. Although I profiled all of them but will list timings and cache profile for that largest test case only.

Time taken with the trivial algorithm i.e. no optimisation : `~63 seconds`

V1

- Converted 2-D array for storing adjacency matrix into 1-D. Then for accessing `A[i][j]` we used pointer notation `(*(A + i*V + j))` , with keeping in mind the fact that generally 2-D matrices are stored in Row Major format in main memory.
- Pointer accesing to memory Restricted pointer access instead of array look-ups.
- Pre increment over post increment Pre-increment is faster than post-increment because post increment keeps a copy of previous (existing) value and adds 1 in the existing value while pre-increment is simply adds 1 without keeping the existing value.

```
 void floyd_v1(int * matrix, int V)
{
    for(int k=0;k<V;++k)
    {
        for(int i=0;i<V;++i)
        {
            for(int j=0;j<V;++j)
            {
                if(k==j) continue;
                if(k==i) continue;
                if( (*(matrix + k*V + j)) == INFINITY ) continue;
                if( (*(matrix + i*V + k)) == INFINITY ) continue;
                if( (*(matrix + i*V + j)) > ( (*(matrix + i*V + k) ) + ( *(matrix + k*V + j) ) ) )
                    (*(matrix + i*V + j)) = ( (*(matrix + i*V + k) ) + ( *(matrix + k*V + j) ) );
            }
        }
    }
}
```

Run time was still `~58s` .

V2

- Used `register` keyword to reduce fetch time for `i` , `j` , `k` iterators which were being used more often.
- Used a temporary variable for storing and reducing memory lookup using pointers i.e. `register int * kmj = (matrix + k*V + j);` and `register int * imk = (matrix + i*V + k);`

```
void floyd_v2(int * matrix, int V)
{
    register int i,j,k,kmj,imk, v;
    register int * km;
    register int * im;
    v = V;
    for(k=0;k<v;++k)
    {
        for(i=0;i<v;++i)
        {
            for(j=0;j<v;j++)
            {
                if(k==j) continue;
                if(k==i) continue;
                register int * kmj = (matrix + k*V + j);
                if( (*kmj) == INFINITY ) continue;
                register int * imk = (matrix + i*V + k);
                if( (*imk) == INFINITY ) continue;
                if( (*(matrix + i*V + j)) > ( kmj + imk ) ) (*(matrix + i*V + j)) = ( kmj + imk );
            }
        }
    }
}
```

Run time was `~37.56s`.

V3

- Tweaked the value of `INFINITY` to 1000000000.
- Removed spurious `if` conditions.
- Used more general memory lookup pointer variable i.e. replace `imk` & `kmj` with `im` and `km`. These new variable will be have more reads / writes ratio as compared to `imk` & `kmj` .

```

void floyd_v2(int * matrix, int V)
{
    register int i,j,k,kmj,imk, v;
    register int * km;
    register int * im;
    v = V;
    for(k=0;k<v;++k)
    {
        km = (matrix + k*v);
        for(i=0;i<v;++i)
        {
            im = (matrix + i*v);
            for(j=0;j<v;++j)
            {
                ...
            }
        }
    }
}
```

Run time was `~25s`

V4

- Saw chance of loop unrolling, hence grabbed the opportunity. Experimenting with values we get 16 as optimised unrolling parameter.

```
void floyd_v2(int * matrix, int V)
{
    ...
            for(j=0;j+15<v;j+=16)
            {
                imk = (*(im + k));

                kmj = (*(km + j+0));
                if( (*(im + j+0)) > ( kmj + imk ) ) (*(im + j+0)) = ( kmj + imk );
                ...

                kmj = (*(km + j+15));
                if( (*(im + j+15)) > ( kmj + imk ) ) (*(im + j+15)) = ( kmj + imk );
            }
            while(j<v)
            {
                kmj = (*(km + j));
                imk = (*(im + k));
                if( (*(im + j)) > ( kmj + imk ) ) (*(im + j)) = ( kmj + imk );
                j++;
            }
        }
    }
}
```

Run time was reduced to `~15s`

V5

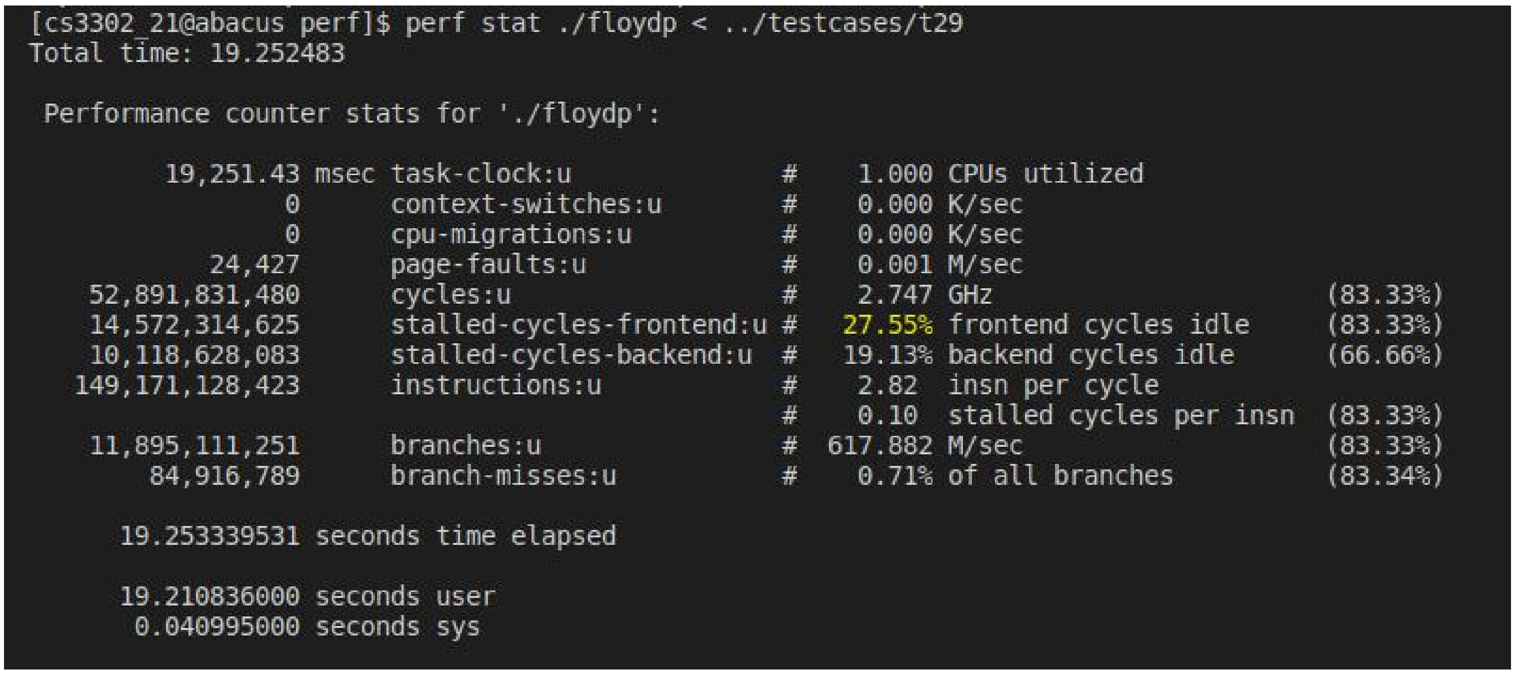
- Did some research and found about Blocked Floyd Warshall algorithm. But it increased the time to `~22.5s`. This may be because blocking works well only for parallelization with cilk_for which we are not allowed to use.

```
void FW_BLOCK(int * matrix,register int I,register int J,register int K,register int BLOCK,register int V)
{
    register int ii, jj, kk;
    register int GOTILL = fun_min(V,K+BLOCK);
    register int FH = fun_min(V, I+BLOCK);
    register int FW = fun_min(V, J+BLOCK);

    for(kk=K;kk<GOTILL;++kk)
    {
        for(ii=I;ii<FH;++ii)
        {
            register int aik = (*(matrix + ( V*(ii) ) + ( kk )));
            for(jj=J;jj+15<FW;jj+=16)
            {
                register int bkj = (*(matrix + ( V*( kk ) ) + ( jj + 0 )));
                register int * mm = (matrix + ( V*( ii ) ) + ( jj + 0 ));
                if( aik+bkj < (*(mm)) ) (*(mm))= aik + bkj;

                ...

                bkj = (*(matrix + ( V*( kk ) ) + ( jj + 15 )));
                mm = (matrix + ( V*( ii ) ) + ( jj + 15 ));
                if( aik+bkj < (*(mm)) ) (*(mm))= aik + bkj;
            }
            while (jj<FW)
            {
                register int bkj = (*(matrix + ( V*( kk ) ) + ( jj )));
                register int * mm = (matrix + ( V*( ii ) ) + ( jj ));
                if( aik+bkj < (*(mm)) )
                    (*(mm))= aik + bkj;
                jj++;
            }

        }
    }
}

void floyd_v3(int * matrix, int V)
{
    register int BLOCK=512,v = V;

    for(register int K=0;K<v;K+=BLOCK)
    {
        // Diagonal update
        FW_BLOCK(matrix,K,K,K,BLOCK,v);


        // Panel Update
        for(register int J=0;J<v;J+=BLOCK)
        {
            if(J!=K)
            {
                // row
                FW_BLOCK(matrix,K,J,K,BLOCK,v);
                // col
                FW_BLOCK(matrix,J,K,K,BLOCK,v);
            }
        }

        // MinPlus Outer Product
        for(register int I=0;I<v;I+=BLOCK)
        {
            if (I==K) continue;
            for(register int J=0;J<v;J+=BLOCK)
            {
                if(J!=K)
                {
                    FW_BLOCK(matrix,I,J,K,BLOCK,v);
                }
            }
        }
    }
}
```

Final Result

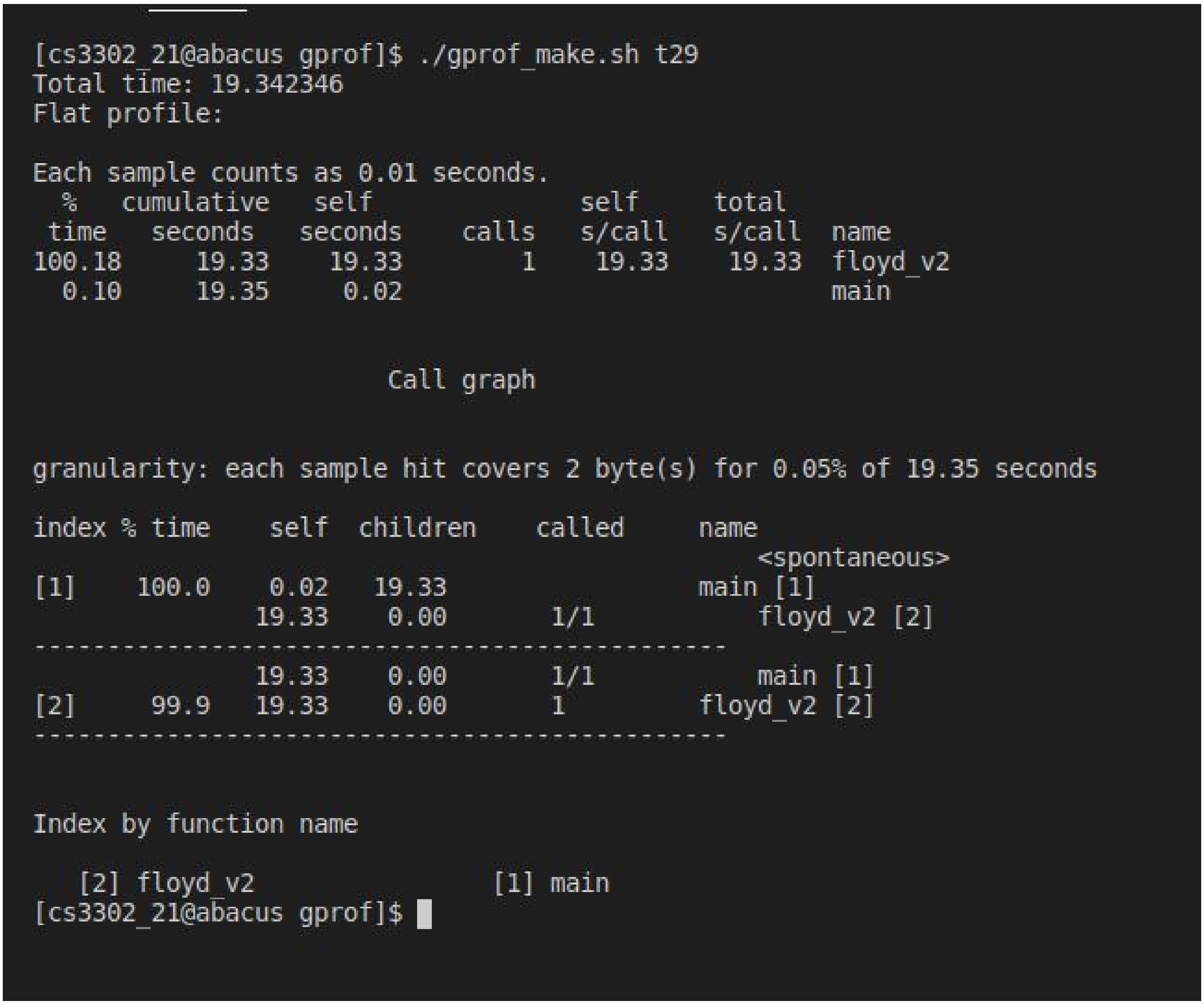
On Abacus we get best running time to be `~19s` with our final code. Profiling on abacus:

Perf stat on Floyd Warshall Final Code :



```
[cs3302_21@abacus perf]$ perf stat ./floydp < ../testcases/t29
Total time: 19.252483

 Performance counter stats for './floydp':

         19,251.43 msec task-clock:u              #    1.000 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
            24,427      page-faults:u             #    0.001 M/sec
    52,891,831,480      cycles:u                  #    2.747 GHz                      (83.33%)
    14,572,314,625      stalled-cycles-frontend:u #   27.55% frontend cycles idle     (83.33%)
    10,118,628,083      stalled-cycles-backend:u  #   19.13% backend cycles idle      (66.66%)
   149,171,128,423      instructions:u            #    2.82  insn per cycle
                                                  #    0.10  stalled cycles per insn  (83.33%)
    11,895,111,251      branches:u                #  617.882 M/sec                    (83.33%)
        84,916,789      branch-misses:u           #    0.71% of all branches          (83.34%)

      19.253339531 seconds time elapsed

      19.210836000 seconds user
       0.040995000 seconds sys
```

Gprof on Floyd Warshall Final Code :



```
[cs3302_21@abacus gprof]$ ./gprof_make.sh t29
Total time: 19.342346
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.18     19.33    19.33        1    19.33    19.33  floyd_v2
  0.10     19.35     0.02                             main


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.05% of 19.35 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02   19.33                 main [1]
               19.33    0.00       1/1           floyd_v2 [2]
-----------------------------------------------
               19.33    0.00       1/1           main [1]
[2]     99.9   19.33    0.00       1         floyd_v2 [2]
-----------------------------------------------


Index by function name

   [2] floyd_v2                [1] main
[cs3302_21@abacus gprof]$ 
```