Report

Objective

Optimising code for matrix multiplication and floyd warshal, as well as gaining understanding of optimisation methods and tools.

Part 0

Gain an understanding of `perf`, `cachegrind`, `gprof` and `clock_gettime`.

1. Perf

`perf` is a performance counter tool for Linux that provides a framework for all things for performance analysis. It covers hardware level (CPU/PMU) features as well as software features such as software counters, tracepoints.

How it works ?

Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. perf provides rich generalized abstractions over hardware specific capabilities. Among others, it provides per task, per CPU and per-workload counters, sampling on top of these and source code event annotation.

Tracepoints are instrumentation points placed at logical locations in code, such as for system calls, TCP/IP events, file system operations, etc. perf dynamically creates tracepoints using the kprobes and uprobes frameworks, for kernel, user

space dynamic tracing. perf can also collect information including timestamps and stack traces

Usage

`perf stat [<OPTIONS>][<COMMAND>][ARGS]`

Example

Running: `perf stat ./a.out < input.txt > output.txt`

```
Performance counter stats for './a.out':

     14847.496986      task-clock (msec)         #    1.000 CPUs utilized
               28      context-switches          #    0.002 K/sec
                0      cpu-migrations            #    0.000 K/sec
            6,901      page-faults               #    0.465 K/sec
  60,48,04,32,898      cycles                    #    4.073 GHz
1,92,20,71,07,093      instructions              #    3.18  insn per cycle
   5,58,31,50,524      branches                  #  376.033 M/sec
       55,06,803      branch-misses             #    0.10% of all branches

     14.848180282 seconds time elapsed

     14.831735000 seconds user
      0.015999000 seconds sys
```

2. Cachegrind

Cachegrind is a cache and branch-prediction profiler. It simulates how a program interacts with the machine's cache by running it in a sandboxed environment which simulates caching. It then measures the cache usage patterns of the program, the number of cache hits and misses and the spatial and temporal locality of the program.

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.

Because of the difference in CPU and memory speeds, cache optimisation is an integral part of any optimisation attempt, and cachegrind helped us analyse our cache usage.

Usage

`valgrind --tool = cachegrind ./a.out`

Example

```
==5618==
==5618== I   refs:      192,169,149,708
==5618== I1  misses:              1,226
==5618== LLi misses:              1,219
==5618== I1  miss rate:            0.00%
==5618== LLi miss rate:            0.00%
==5618==
==5618== D   refs:       91,907,695,023  (91,770,148,151 rd   + 137,546,872 wr)
==5618== D1  misses:        628,936,948  (   627,998,586 rd   +     938,362 wr)
==5618== LLd misses:          1,391,376  (       811,624 rd   +     579,752 wr)
==5618== D1  miss rate:             0.7% (           0.7%     +         0.7%  )
==5618== LLd miss rate:             0.0% (           0.0%     +         0.4%  )
==5618==
==5618== LL refs:           628,938,174  (   627,999,812 rd   +     938,362 wr)
==5618== LL misses:           1,392,595  (       812,843 rd   +     579,752 wr)
==5618== LL miss rate:              0.0% (           0.0%     +         0.4%  )
```

3. clock_gettime()

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()` is a function used to read a given POSIX clock and is defined at <time.h>. The clock_gettime() command takes two parameters: the POSIX clock ID and a timespec structure which will be filled with the duration used to read the clock.

There are multitude POSIX clocks and we can specify which one to use:

- CLOCK_REALTIME
- CLOCK_MONOTONIC
- CLOCK_PROCESS_CPUTIME_ID
- CLOCK_THREAD_CPUTIME_ID

Usage

`int clock_gettime(clockid_t clk_id, struct timespec *tp);`

Example

```c
if (clock_gettime(CLOCK_REALTIME, &begin) == -1)
{
    perror("clock gettime");
    return EXIT_FAILURE;
}
```

4. gprof

`gprof` 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 (`gmon.out` default) which is created by programs that are compiled with the `-pg` option of cc, pc, and f77. The `-pg` option also links in versions of the library routines that are compiled for profiling. Gprof reads the given object file (the default is `a.out`) and establishes the relation between its symbol table and the call graph profile from `gmon.out`.

`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.

Several forms of output are available from the analysis such as:

- 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.
- The *annotated source* listing is a copy of the program's source code, labeled with the number of times each line of the program was execute

Usage

`gprof options [executable-file [profile-data-files…]] [> outfile]`

Example: Suppose your code is in TestGprof.c

```
gcc -pg -o TestGprof TestGprof.c
./TestGprof
gprof -b TestGprof gmon.out > analysis.out
```

This will give an human readable file. This file contains two tables:

1. **flat profile**: overview of the timing information of the functions
2. **call graph**: focuses on each function

`b` option will suppress lot of verbose information which would be otherwise included in analysis file.

Example

Running gprof on question 1.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 98.07     10.08    10.08        4     2.52     2.52  matrix_multiply
  0.68     10.15     0.07  5000011     0.00     0.00  scan_d
  0.10     10.16     0.01        1     0.01     0.01  print_matrix
  0.10     10.17     0.01                             main
  0.00     10.17     0.00        9     0.00     0.00  get_matrix
  0.00     10.17     0.00        1     0.00    10.08  chain_multiply
<0x0c>
```

```
Call graph

granularity: each sample hit covers 2 byte(s) for 0.10% of 10.17 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01   10.16                 main [1]
                0.00   10.08       1/1           chain_multiply [3]
                0.07    0.00 5000011/5000011     scan_d [4]
                0.01    0.00       1/1           print_matrix [5]
                0.00    0.00       5/9           get_matrix [6]
-----------------------------------------------
               10.08    0.00       4/4           chain_multiply [3]
[2]     99.1   10.08    0.00       4         matrix_multiply [2]
                0.00    0.00       4/9           get_matrix [6]
-----------------------------------------------
                                   8             chain_multiply [3]
                0.00   10.08       1/1           main [1]
[3]     99.1    0.00   10.08       1+8       chain_multiply [3]
               10.08    0.00       4/4           matrix_multiply [2]
                                   8             chain_multiply [3]
-----------------------------------------------
                0.07    0.00 5000011/5000011     main [1]
[4]      0.7    0.07    0.00 5000011         scan_d [4]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[5]      0.1    0.01    0.00       1         print_matrix [5]
-----------------------------------------------
                0.00    0.00       4/9           matrix_multiply [2]
                0.00    0.00       5/9           main [1]
[6]      0.0    0.00    0.00       9         get_matrix [6]
-----------------------------------------------
<0x0c>
Index by function name

   [3] chain_multiply          [1] main                    [5] print_matrix
   [6] get_matrix              [2] matrix_multiply         [4] scan_d
```

Part 1: Matrix Multiplication

- **Naive Algorithm: Basic O(N^3) algorithm**
    1. Code:

```
Matrix *matrix_multiply(Matrix *A, Matrix *B, int r1, int r2, int c2)
{
    Matrix *result = malloc(sizeof(Matrix));
    int i, k, j;
    for (i = 0; i < r1; ++i)
    {
        for (j = 0; j < c2; j++)
        {
            for (k = 0; k < r2; ++k)
            {
                result->matrix[i][j] += (A->matrix[i][k] * B->matrix[k][j]);
            }
        }
    }
    return result;
}
```

2. Runtime:

```
[cs3302_11@node06 Q1]$ ./run.sh codes/base.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.010597
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.026240
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.033867
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.766627
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.201112
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 4.326435
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.055206
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 2.869244
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.034430
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.142200
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 20.731755
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 13.977061
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 7.281327
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 7.219891
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 40.873230
Time: 98.5492
```

- **Optimisation 1:**

  changing the loop order as to improve the chances of getting cache hit, as the neighbouring elements will now be accesed versus accesing different columns of the different matrices.

  1. Code:

```
Matrix *matrix_multiply(Matrix *A, Matrix *B, int r1, int r2, int c2)
{
    Matrix *result = malloc(sizeof(Matrix));
    int i, k, j;
    for (i = 0; i < r1; ++i)
    {
        for (k = 0; k < r2; ++k)
        {
            for (j = 0; j < c2; j++)
            {
                result->matrix[i][j] += (A->matrix[i][k] * B->matrix[k][j]);
            }
        }
    }

    return result;
}
```

2. Runtime:

```
[cs3302_11@node06 Q1]$ ./run.sh codes/o1.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.010347
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.024933
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.031898
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.663546
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.175023
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 2.181563
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.054458
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 1.493476
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.031842
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.136220
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 8.902141
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 5.959539
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 3.091565
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 3.176497
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 17.072398
Time: 43.0055
```

- **Optimisation 2:**

  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.

  1. Code:

     `++i` faster than `i++`

  2. Runtime:

```
[cs3302_11@node06 Q1]$ ./run.sh codes/o2.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.009254
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.037021
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.046354
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.566503
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.154628
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 1.881069
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.055768
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 1.280332
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.028110
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.137635
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 7.777489
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 5.129214
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 2.622118
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 2.611470
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 14.385458
Time: 36.7225
```

- **Optimisation 3:**

  Pointer accesing to memory Restricted pointer access instead of array look-ups.

  1. Code:

     `*(*(A + i) + j)` faster than `A[i][j]`

  2. Runtime:

```
[cs3302_11@node06 Q1]$ ./run.sh codes/o3.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.007976
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.018347
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.024858
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.452876
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.128944
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 1.462480
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.054878
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 1.017505
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.024097
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.140275
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 6.412716
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 4.102926
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 2.126639
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 2.114344
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 11.812698
Time: 29.9015
```

- **Optimisation 4:**

  Unrolling loops in order to reduce number loop break checks.

  1. Code:

```
// slower
for(int i = 2; i < 6; i++) {
   fib[i] = fib[i-1] + fib[i-2];
}

// faster
for(int i = 2; i < 6; i+=4) {
  fib[i] = fib[i-1] + fib[i-2];
  fib[i+1] = fib[i] + fib[i-1];
  fib[i+2] = fib[i+1] + fib[i];
  fib[i+3] = fib[i+2] + fib[i+1];
}
```

2. Runtime:

```
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.007188
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.015734
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.019954
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.339136
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.099607
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 1.078173
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.054734
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 0.740417
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.019530
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.136436
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 4.524478
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 2.991268
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 1.585096
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 1.665043
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 8.531682
Time: 21.8085
```

- **Optimisation 5:**

  Use Register variables as counters of inner loops: Variables stored in registers can be accessed much faster than variables stored in memory.

  1. Code:

```
register int i = 0;
register int j = 0;
int n = 5;

// using register variables
// as counters make the loop faster
for (i = 0; i < n; ++i) {
    for (j = 0; j <= i; ++j) {
         printf("* ");
    }
    printf("\n");
 }
```

2. Runtime

```
[cs3302_11@node06 Q1]$ ./run.sh codes/o4.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.006775
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.015064
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.019540
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.333575
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.097264
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 1.051678
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.053241
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 0.719663
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.019539
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.134168
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 4.120251
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 2.805966
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 1.463814
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 1.462327
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 8.328603
Time: 20.6314
```

- **Optimisation 6:**

  Using Restrict keyword. When we use restrict with a pointer ptr, it tells the compiler that ptr is the only way to access the object pointed by it and compiler doesn't need to add any additional checks.

  1. Code

```
use(int* a, int* b, int* restrict c)
{
    *a += *c;

    // Since c is restrict, compiler will
    // not reload value at address c in
    // its assembly code. Therefore generated
    // assembly code is optimized
    *b += *c;
}
```

  2. Runtime:

```
[cs3302_11@node06 Q1]$ ./run.sh code1.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q1/0.txt: 0.004264
Time for /scratch/amul-agrawal/A1/Q1/10.txt: 0.010157
Time for /scratch/amul-agrawal/A1/Q1/15.txt: 0.013471
Time for /scratch/amul-agrawal/A1/Q1/20.txt: 0.289469
Time for /scratch/amul-agrawal/A1/Q1/30.txt: 0.080797
Time for /scratch/amul-agrawal/A1/Q1/40.txt: 0.973582
Time for /scratch/amul-agrawal/A1/Q1/50.txt: 0.034026
Time for /scratch/amul-agrawal/A1/Q1/55.txt: 0.659444
Time for /scratch/amul-agrawal/A1/Q1/5.txt: 0.012932
Time for /scratch/amul-agrawal/A1/Q1/60.txt: 0.086989
Time for /scratch/amul-agrawal/A1/Q1/70.txt: 4.077017
Time for /scratch/amul-agrawal/A1/Q1/75.txt: 2.810711
Time for /scratch/amul-agrawal/A1/Q1/80.txt: 1.439120
Time for /scratch/amul-agrawal/A1/Q1/85.txt: 1.435843
Time for /scratch/amul-agrawal/A1/Q1/90.txt: 8.306398
Time: 20.2342
```

Perf

```
Performance counter stats for './a.out':

      5103.861450      task-clock:u (msec)       #    0.999 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
           32,676      page-faults:u             #    0.006 M/sec
  14,31,03,82,526      cycles:u                  #    2.804 GHz                      (83.31%)
   4,11,56,49,447      stalled-cycles-frontend:u #   28.76% frontend cycles idle     (83.34%)
   1,37,26,56,474      stalled-cycles-backend:u  #    9.59% backend cycles idle      (66.66%)
  39,62,46,64,137      instructions:u            #    2.77  insn per cycle
                                                 #    0.10  stalled cycles per insn  (83.35%)
     44,32,81,246      branches:u                #   86.852 M/sec                    (83.33%)
        35,08,128      branch-misses:u           #    0.79% of all branches          (83.35%)

      5.110884866 seconds time elapsed
```

Gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 98.40      4.93     4.93        3     1.64     1.64  matrix_multiply
  1.20      4.99     0.06  4000009     0.00     0.00  scan_d
  0.40      5.01     0.02                             main
  0.20      5.02     0.01        1     0.01     0.01  print_matrix
  0.00      5.02     0.00       10     0.00     0.00  get_matrix
  0.00      5.02     0.00        1     0.00     4.93  chain_multiply
^L
```

Cachegrind

```
==53667== I   refs:      18,722,541,523
==53667== I1  misses:               856
==53667== LLi misses:               849
==53667== I1  miss rate:           0.00%
==53667== LLi miss rate:           0.00%
==53667==
==53667== D   refs:       3,474,843,930  (3,272,457,037 rd   + 202,386,893 wr)
==53667== D1  misses:       180,566,030  (  179,804,571 rd   +     761,459 wr)
==53667== LLd misses:         1,271,978  (      516,550 rd   +     755,428 wr)
==53667== D1  miss rate:            5.2% (          5.5%     +         0.4%  )
==53667== LLd miss rate:            0.0% (          0.0%     +         0.4%  )
==53667==
==53667== LL refs:          180,566,886  (  179,805,427 rd   +     761,459 wr)
==53667== LL misses:          1,272,827  (      517,399 rd   +     755,428 wr)
==53667== LL miss rate:             0.0% (          0.0%     +         0.4%  )
```

Part 2: Floyd Warshall Algorithm

The above mentioned generic compiler optimisations were done in this question also. Apart from it the following optimisations were done.

- **Naive Algorithm: Basic O(N^3) algorithm**

  1. Code:

```
int i, j, k;
for ( k = 0; k < n; k++)
{
    for (i = 0; i < n; i++)
    {
        for (j =  0; j < n; j++)
        {
            if (dist[i][k] != INF && dist[k][j] != INF)
            {
               int temp = dist[i][k] + dist[k][j];
                   if (dist[i][j] == INF || temp < dist[i][j])
               {
                dist[i][j] = temp;
               }
            }
        }
    }
}
```

  2. Runtime:

```
[cs3302_11@node06 Q2]$ ./run.sh codes/naive.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q2/t19: 74.696945
Time for /scratch/amul-agrawal/A1/Q2/t29: 109.601279
Time for /scratch/amul-agrawal/A1/Q2/t31: 0.076481
Time for /scratch/amul-agrawal/A1/Q2/t43: 0.447280
Time for /scratch/amul-agrawal/A1/Q2/t49: 0.208492
Time for /scratch/amul-agrawal/A1/Q2/t6: 0.260486
Time for /scratch/amul-agrawal/A1/Q2/t75: 0.379461
Time for /scratch/amul-agrawal/A1/Q2/t77: 70.628294
Time for /scratch/amul-agrawal/A1/Q2/t78: 2.892620
Time for /scratch/amul-agrawal/A1/Q2/t91: 2.982157
Time: 262.171
```

- **Optimisation 1: Moving the dist[i][k] above 1 level**

  By shifting the expression above the jth loop, we also come to know that the jth loop executes only when the condition dist[i][k] != INF is true, implying further possibility of efficiency.

  1. Code:

```
int i, j, k;
for ( k = 0; k < n; k++)
{
    for ( i = 0; i < n; i++ )
    {
       if( dist[i][k] != INF )
       {
            for ( j =  0; j < n; j++ )
            {
                if ( dist[k][j] != INF )
                {
                    int temp = dist[i][k] + dist[k][j];
                    if (dist[i][j] == INF || temp < dist[i][j])
                    {
                       dist[i][j] = temp;
                    }
                }
            }
       }
    }
}
```

  2. Runtime:

```
[cs3302_11@node06 Q2]$ ./run.sh codes/o1.c sa
Running on Sample Test Cases
Time for /scratch/amul-agrawal/A1/Q2/t19: 59.697337
Time for /scratch/amul-agrawal/A1/Q2/t29: 88.548410
Time for /scratch/amul-agrawal/A1/Q2/t31: 0.067049
Time for /scratch/amul-agrawal/A1/Q2/t43: 0.350729
Time for /scratch/amul-agrawal/A1/Q2/t49: 0.157836
Time for /scratch/amul-agrawal/A1/Q2/t6: 0.198583
Time for /scratch/amul-agrawal/A1/Q2/t75: 0.283337
Time for /scratch/amul-agrawal/A1/Q2/t77: 56.696291
Time for /scratch/amul-agrawal/A1/Q2/t78: 0.192948
Time for /scratch/amul-agrawal/A1/Q2/t91: 0.250352
Time: 206.443
```

Perf

```
Performance counter stats for './a.out':

     16012.687039      task-clock:u (msec)       #    0.999 CPUs utilized
               0      context-switches:u        #    0.000 K/sec
               0      cpu-migrations:u          #    0.000 K/sec
          24,467      page-faults:u             #    0.002 M/sec
  45,34,01,84,918      cycles:u                  #    2.832 GHz                      (83.33%)
  18,71,96,27,202      stalled-cycles-frontend:u #   41.29% frontend cycles idle     (83.34%)
  10,41,20,77,754      stalled-cycles-backend:u  #   22.96% backend cycles idle      (66.66%)
  81,75,63,78,770      instructions:u            #    1.80  insn per cycle
                                                 #    0.23  stalled cycles per insn  (83.34%)
  12,13,98,62,487      branches:u                #  758.140 M/sec                    (83.33%)
      8,67,84,293      branch-misses:u           #    0.71% of all branches          (83.34%)

     16.027484438 seconds time elapsed
```

Gprof

```
amul@acer:~/course/sem4/spp/assignments/20191013
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.71     13.33    13.33        1    13.33    13.33  FloydWarshall
  0.22     13.36     0.03        1     0.03     0.03  print_distance
  0.07     13.37     0.01                             main
  0.00     13.37     0.00   465953     0.00     0.00  scan_d
^L
```

Valgrind

```
==54573== I   refs:      5,731,309,367
==54573== I1  misses:              786
==54573== LLi misses:              778
==54573== I1  miss rate:          0.00%
==54573== LLi miss rate:          0.00%
==54573==
==54573== D   refs:      1,632,626,561  (1,592,067,922 rd   + 40,558,639 wr)
==54573== D1  misses:       48,481,327  (   48,167,217 rd   +    314,110 wr)
==54573== LLd misses:       44,264,878  (   43,952,188 rd   +    312,690 wr)
==54573== D1  miss rate:           3.0% (          3.0%     +        0.8%  )
==54573== LLd miss rate:           2.7% (          2.8%     +        0.8%  )
==54573==
==54573== LL refs:          48,482,113  (   48,168,003 rd   +    314,110 wr)
==54573== LL misses:        44,265,656  (   43,952,966 rd   +    312,690 wr)
==54573== LL miss rate:            0.6% (          0.6%     +        0.8%  )
```