SPP A2 | Floyd Warshal Optimization

- Ayush Sharma
- 2019101004

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.

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`

My machine has total 8 cores

Previous Version (i.e. in Assignment1)

Final Total time:

- On Abacus : ~19.832843 (core independent as no multi-programming was there)
- ~15s on my machine

Process for previous version optimisation

- 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.
- 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);`
- 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+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 `~19.8s` on Abacus.

Later Optimisation for Assignment2

- on Abacus with total core 4 : ~8.310297s (on average)
- on Abacus with total core 5 : ~6.212121s (on average)
- On my machine : ~5.6966s (on average)

Process for multicore programming optimisation

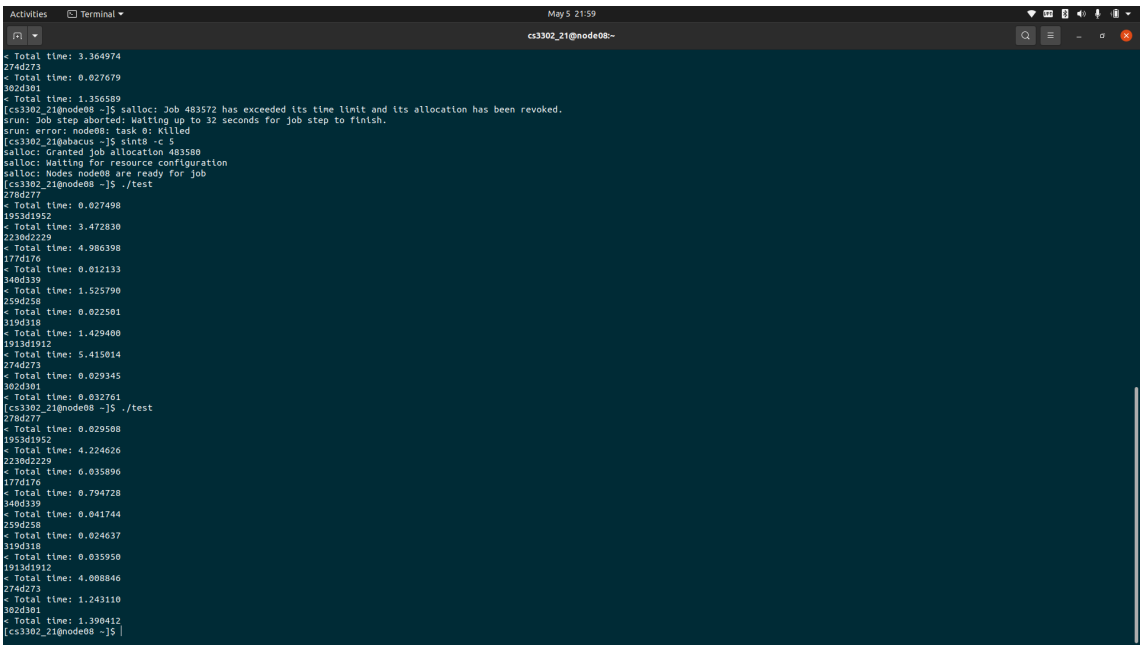
The OpenMP API uses the fork-join model of parallel execution. Multiple threads perform tasks defined implicitly or explicitly by OpenMP directives. All OpenMP applications begin as a single thread of execution, called the initial thread. The initial thread executes sequentially, until it encounters a parallel construct. At that point, this thread creates a group of itself and zero or more additional

threads and becomes the master thread of the new group. Each thread executes the commands included in the parallel region, and their execution may be differentiated, according to additional directives provided by the programmer. At the end of the parallel region, all threads are synchronized

```
void floyd_v1(int * matrix, int V)
{
    register int i,j,k;
    for(k=0;k<V;++k)
    {
        register int* km = (matrix + k*V);

        #pragma omp parallel for private(j) shared(matrix,V)

        for(i=0;i<V;++i)
        {
            register int* im = (matrix + i*V);

            for(j=0;j<V;++j)
            {
                if( (*(im + j)) > ( (*(im + k) ) + ( *(km + j) ) ) )
                    (*(im + j)) = ( (*(im + k) ) + ( *(km + j) ) );
            }
        }
    }
}
```

Final Result

On Abacus we get best running time to be `~5s` with our final code with core 5.



Profiling on abacus:

`gprof`

```
[cs3302_21@node06 ~]$ cat gprof_work.sh
#!/bin/bash
gcc -pg -fopenmp -O2 code.c -o floyd
./floyd < testcases/Q2/t29 > temp
diff temp testcases/Q2out/t29
gprof -b ./floyd gmon.out
[cs3302_21@node06 ~]$ ./gprof_work.sh
2230d2229
< Total time: 5.149734
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.97     26.17    26.17     1919    13.64    13.64  frame_dummy
  0.04     26.18     0.01                             floyd_v1


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.04% of 26.18 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01   26.17                 floyd_v1 [1]
               26.17    0.00    1919/1919        frame_dummy [2]
-----------------------------------------------
               26.17    0.00    1919/1919        floyd_v1 [1]
[2]    100.0   26.17    0.00    1919         frame_dummy [2]
-----------------------------------------------


Index by function name

   [1] floyd_v1                [2] frame_dummy
[cs3302_21@node06 ~]$
```

`Valgrind`

```
Total time: 502.608561
==232890==
==232890== I   refs:      100,931,167,470
==232890== I1  misses:              2,031
==232890== LLi misses:              2,007
==232890== I1  miss rate:            0.00%
==232890== LLi miss rate:            0.00%
==232890==
==232890== D   refs:       35,960,940,773  (35,301,967,179 rd   + 658,973,594 wr)
==232890== D1  misses:        693,564,677  (   693,131,478 rd   +     433,199 wr)
==232890== LLd misses:        660,051,712  (   659,628,559 rd   +     423,153 wr)
==232890== D1  miss rate:             1.9% (           2.0%     +         0.1%  )
==232890== LLd miss rate:             1.8% (           1.9%     +         0.1%  )
==232890==
==232890== LL refs:           693,566,708  (   693,133,509 rd   +     433,199 wr)
==232890== LL misses:         660,053,719  (   659,630,566 rd   +     423,153 wr)
==232890== LL miss rate:              0.5% (           0.5%     +         0.1%  )
```