

COMPARISON BETWEEN PARALLEL AND SERIAL ALGORITHMS USING OpenMP+SIMD

A PROJECT REPORT FOR J COMPONENT

PARALLEL AND DISRIBUTED COMPUTING (CSE 4001)

PROJECT SUPERVISOR- **PROF. SAIRA BANU J**

Submitted by

SHIKHAR SINGH -16BCE2316

ARATRIK PAUL-16BCE0519

School of Computer Science Engineering



Nov. 2017

School of Computer Science and Engineering

CERTIFICATE

This is to certify that the project entitled, "Comparision between Parallel and serial algorithms using OpenMP +SIMD" submitted by the students Shikhar Singh=16BCE2316 and Aratrik Paul =16BCE0519 for the course "Parallel and Distibuted Computing(CSE-4001)" at "VIT university", is an authentic work carried out by the members of the team under my supervision and guidance. To the best of my knowledge, the matter embodied in the project has not been submitted to any other University / Institute for any purpose and is unique.

Date:

Place:

Signature

ACKNOWLEDGEMENTS

We would like to extend our thanks to the management of VIT University, Vellore and our respected chancellor G. Viswanathan for giving us the opportunity to carry out our project in VIT.

We would also like to extend our gratitude to our faculty, Prof. Saira Banu J for her constant and ceaseless guidance and support through the entire course of this project. We would also want to thank ma'am for guiding us all along the way and encouraging us to think out of the box in an attempt to maximize the project's validity for the society.

Shikhar Singh-16BCE2316

Aratrik Paul – 16BCE0519

ABSTRACT

We the students of VIT, Vellore have made a project on “ Comparison of Parallel and serial Algorithms using OpenMP +SIMD ” as our Parallel and Distributed Computing project for the fifth semester. Our goal was to give an overview of the Analysis and speedup of Different Algorithms such as Dijkstra’s Algorithm and find out how effective it would be if it were parallelized.

Table of Contents

- 1. INTRODUCTION**
- 2. LITERATURE SURVEY**
- 3. IMPLEMENTATION DETAILS**
- 4. RESULTS AND DISCUSSIONS**
- 5. PARAMETERS TO BE CONSIDERED**
- 6. CONCLUSION**
- 7. REFERENCES**

1) INTRODUCTION

OpenMP (Open Multi-Processing) is an application programming interface (API) that helps us use a multiple platform shared memory multi - processing programming in C, C++ and/or Fortran, on most platforms, the instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of directives and library routines that influence run-time behavior.

OpenMP is managed by the non for profit technology Company OpenMP Architecture Review Board or OpenMP ARB, jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, and many more.

OpenMP uses a portable and scalable model that gives programmers a simple interface for developing parallel applications for platforms may it be on a computer, Desktop or a Supercomputer.

An app built with the hybrid model of parallel programming can run on a Desktop cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a multiple core node while MPI is used for parallelism amongst nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

OpenMP is the way to implement multithreading, a method of parallelizing where a master thread forks a specified number of slave threads (as it uses the Master Slave Model) and the system divides a task among them. The threads then run at the same time, with the Main Environment allocating threads to different processors.

The Part of the code that is meant to run in the parallel area is marked accordingly, with a compiler directive that forms threads before the section is executed. Each thread has an id attached to it which can be obtained using a function `omp_get_thread_num()`. The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads report back to the master Program and hence we have a completely compiled parallel program.

By default, each thread executes the parallelized section of code independently. Work-sharing can be used to divide a task among the threads so that each thread executes its allocated part of the code. We can achieve both the parallelism methods of task and data parallelism by using OpenMP this way.

The environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on

environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `<omp.h>` in C/C++.

SIMD

Single instruction, multiple data (SIMD) comes under the class of parallel computers in Flynn's taxonomy. It talks about computers with multiple processing elements that perform the same operation on multiple data points at the same point of time. These machines take advantage of data level parallelism, but not concurrency: there are parallel computations, but only a single process at a given moment. SIMD is particularly applicable to normal problems such as adjusting the volume of digital audio. Most modern CPU designs use SIMD instructions to improve the performance of multimedia use.

SIMD was the basis for vector supercomputers of the early 1970s such as the CDC Star-100 and the Texas Instruments ASC, which could operate on a "vector" of data with a single instruction. Vector processing was especially popularized by Cray in the 1970s and 1980s[1]. Vector-processing architectures are now considered separate from SIMD computers, based on the fact that vector computers processed the vectors one word at a time through pipelined processors (though still based on a single instruction), whereas modern SIMD computers process all elements of the vector simultaneously.

All of these developments have been made toward support for real-time graphics, and are therefore brought forward toward processing in two, three, or four dimensions, usually with vector lengths of between two and sixteen words, depending on its data type and architecture. When we want to make the difference between the SIMD architectures of old and the newer architectures, the newer are considered "short-vector" architectures, as earlier SIMD and vector supercomputers had vector lengths from 64 to 64,000. A modern supercomputer is almost always a cluster of MIMD computers, each of which implements (short-vector) SIMD instructions. A modern desktop computer is often a multiprocessor MIMD computer where each processor can execute short-vector SIMD instructions.

The Algorithms we have chosen to do the project are Floyd Warshall's Algorithm and Bellman Ford's Algorithm.

A) Bellman Ford's Algorithm

The Bellman Ford algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights with no negative cycles. A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation R , or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

2) Literature Review

1) Performance of Shortest Path Algorithm Based on Parallel Vertex Traversal

Written by- Mihailo Vesović, Aleksandra Smiljanić, Dušan Kostić

Date Published: February 2016

Abstract: Shortest path algorithms for different applications, such as Internet routing, VLSI design and so on are used. Floyd Warshall and Bellman-Ford are commonly used shortest path algorithms which are typically implemented in networks with hundreds of nodes. However, scale of shortest path problems is increasing, and more efficient algorithms are needed. With the development of multicore processors, one natural way to speedup shortest path algorithms is through parallelization. In this paper, we propose a novel shortest path algorithm with parallel vertex transversal, and compare its speed with standard solutions in datacenter topologies.

2) Performance Anaysis of Parallel Algorithms on Multi-Core Systems

Written by: Sanjay Kumar Sharma and Dr Kusum Bhansali

Date Published: October 2012

Abstarct: The current multi-core architectures have become popular due to performance, and efficient processing of multiple tasks simultaneously. Today's the parallel algorithms are focusing on multi-core systems. The design of parallel algorithm and performance measurement is the major issue on multi-core environment. If one wishes to execute a

single application faster, then the application must be divided into subtask or threads to deliver desired result. Numerical problems, especially the solution of linear system of equation have many applications in science and engineering. This paper describes and analyzes the parallel algorithms for computing the solution of dense system of linear equations, and to approximately compute the value of using OpenMP interface. The performances (speedup) of parallel algorithms on multi-core system have been presented. The experimental results on a multi-core processor show that the proposed parallel algorithms achieves good performance (speedup) compared to the sequential.

3) Parallel Computing using OpenMP

Written By: Ms. Ashwini M. Bhugul

Date Published: February ,2017

Abstract: Parallel Computing is most widely used paradigm today to improve the system performance. System performance can be increase by executing application on multiple processors. Today's Computers are complex they are capable of executing application program on multiple processors. OpenMP is API for running application parallelly to improve the speedup. This paper reviews about OpenMP API and focuses on improving the system performance.

4) OpenMP: an industry standard API for shared-memory programming

Written by : L Dagum, R Menon

Date Published : Jan,2010

Abstract: At its most elemental level, OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++ to express shared memory parallelism. It leaves the base language unspecified, and vendors can implement OpenMP in any Fortran compiler. Naturally, to support pointers and allocatables, Fortran 90 and Fortran 95 require the OpenMP implementation to include additional semantics over Fortran 77. OpenMP leverages many of the shared mamory concepts while extending them to support coarse grain parallelism. The standard also includes a callable runtime library with accompanying environment variables.

5) High performance computing using MPI and OpenMP on multi-core parallel systems

Written by: Haoqiang Jin, Piyush Mehrotra

Date Published:March,2012

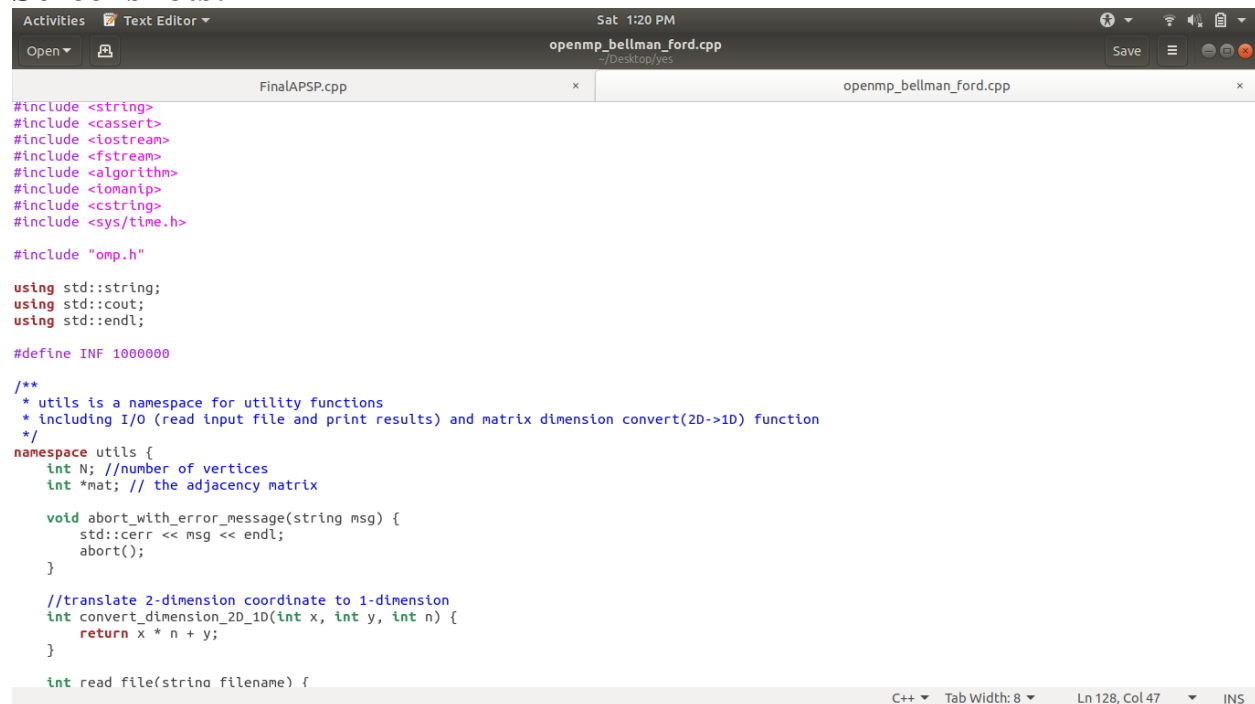
Abstract: The rapidly increasing number of cores in modern microprocessors is pushing the current high performance computing (HPC) systems into the petascale and exascale era. The hybrid nature of these systems – distributed memory across nodes and shared

memory with non-uniform memory access within each node – poses a challenge to application developers. In this paper, we study a hybrid approach to programming such systems – a combination of two traditional programming models, MPI and OpenMP

3) Implementation

We have gone ahead to implement this project on the Ubuntu linux 18.04 LTS system. Where we have used c++ codes and intrinsically installed the gcc compiler and omp libraries used to run an OpenMP Program. The two codes are for Floyd Warshall's and Bellman Ford's Parallel codes using OpenMP + SIMD.

Screenshots:



```
#include <string>
#include <cassert>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <omp.h>
#include <string>
#include <sys/time.h>

using std::string;
using std::cout;
using std::endl;

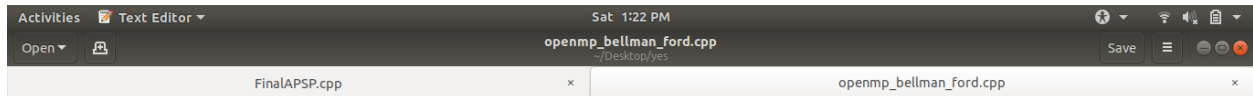
#define INF 1000000

/**
 * utils is a namespace for utility functions
 * including I/O (read input file and print results) and matrix dimension convert(2D->1D) function
 */
namespace utils {
    int N; //number of vertices
    int *mat; // the adjacency matrix

    void abort_with_error_message(string msg) {
        std::cerr << msg << endl;
        abort();
    }

    //translate 2-dimension coordinate to 1-dimension
    int convert_dimension_2D_1D(int x, int y, int n) {
        return x * n + y;
    }

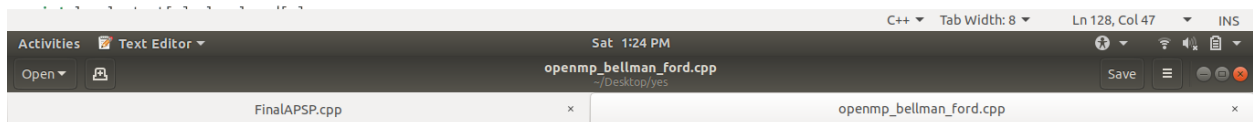
    int read_file(string filename) {
```



```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
        inputf >> mat[convert_dimension_2D_1D(i, j, N)];
    }
return 0;
}

int print_result(bool has_negative_cycle, int *dist) {
    std::ofstream outputf("output.txt", std::ofstream::out);
    if (!has_negative_cycle) {
        for (int i = 0; i < N; i++) {
            if (dist[i] > INF)
                dist[i] = INF;
            outputf << dist[i] << '\n';
        }
        outputf.flush();
    } else {
        outputf << "FOUND NEGATIVE CYCLE!" << endl;
    }
    outputf.close();
    return 0;
}
} // namespace utils

/**
 * Bellman-Ford algorithm. Find the shortest path from vertex 0 to other vertices.
 * @param p number of processes
 * @param n input size
 * @param *mat input adjacency matrix
 * @param *dist distance array
 * @param *has_negative_cycle a bool variable to recode if there are negative cycles
 */
void bellman_ford(int p, int n, int *mat, int *dist, bool *has_negative_cycle) {
```



```
*has_negative_cycle = false;

//step 1: set openmp thread number
omp_set_num_threads(p);

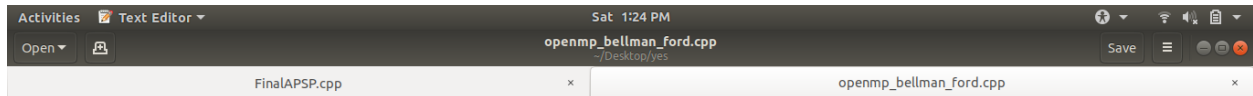
//step 2: find local task range
int ave = n / p;
#pragma omp parallel for
for (int i = 0; i < p; i++) {
    local_start[i] = ave * i;
    local_end[i] = ave * (i + 1);
    if (i == p - 1) {
        local_end[i] = n;
    }
}

//step 3: bellman-ford algorithm
//initialize distances
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    dist[i] = INF;
}
//root vertex always has distance 0
dist[0] = 0;

int iter_num = 0;
bool has_change;
bool local_has_change[p];
#pragma omp parallel
{
    int my_rank = omp_get_thread_num();
    //bellman-ford algorithm
    for (int iter = 0; iter < n - 1; iter++) {
        local_has_change[my_rank] = false;
        for (int u = 0; u < n; u++) {
            for (int v = local_start[my_rank]; v < local_end[my_rank]; v++) {

```

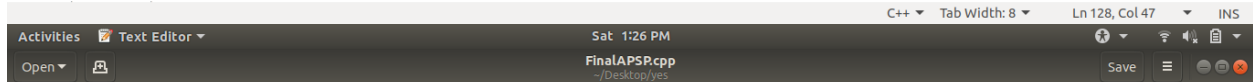
C++ Tab Width: 8 Ln 128, Col 47 INS



```
    has_change = false;
    for (int rank = 0; rank < p; rank++) {
        has_change |= local_has_change[rank];
    }
}
if (!has_change) {
    break;
}
}
}
#pragma omp parallel reduction(|:has_change)
//do one more iteration to check negative cycles
if (iter_num == n - 1) {
    has_change = false;
    for (int u = 0; u < n; u++) {
#pragma omp parallel for simd reduction(|:has_change) // #pragma omp for simd
        for (int v = 0; v < n; v++) {
            int weight = mat[u * n + v];
            if (weight < INF) {
                if (dist[u] + weight < dist[v]) { // if we can relax one more step, then we find a negative cycle
                    has_change = true;
                }
            }
        }
    }
    *has_negative_cycle = has_change;
}

//step 4: free memory (if any)
}

int main(int argc, char **argv) {
    if (argc <= 1) {
        utils::abort_with_error_message("INPUT FILE WAS NOT FOUND!");
    }
}
```



```
#include <iostream>
#include <fstream>
#include <climits>
#include <string>
#include <sstream>
#include <omp.h>
using namespace std;
void updateSubmatrix(int a_row, int a_col, int b_row, int b_col, int c_row, int c_col, int **A, int b)
{
    //This will update the tile which is having dimension bxb
    int i, j, k;
    for (k=0; k<b; k++)
    {
        for (i=0; i<b; i++)
        {
            for (j=0; j<b; j++)
            {
                if (A[b_row+i][b_col+k] < INT_MAX && A[c_row+k][c_col+j] < INT_MAX)
                    A[a_row+i][a_col+j] = min(A[a_row+i][a_col+j], (A[b_row+i][b_col+k] + A[c_row+k][c_col+j]));
            }
        }
    }
}

void tiledFloydWarshall(int **A, int n, int b, int t)
{
    //ntrow is for number of times kth-block will iterate
    //nbrow is actually the dimension of each tile nbrowXnbrow
    int ntrow=n/b;
    int nbrow=n/ntrow;
    int k, i, j;
    omp_set_num_threads(t);

    for (k=0; k<ntrow; k++) //Iterator to position our main tile.
    {
        //Phase 1: updates the k-th diagonal-tile(main tile)
        updateSubmatrix(k*nbrow, k*nbrow, k*nbrow, k*nbrow, k*nbrow, k*nbrow, A, b);

        //Phase 2: updates the tiles which are remained in the main tile row.
        #pragma omp parallel for
        for (j=0; j<ntrow; j++)
        {
            updateSubmatrix(k*nbrow, j*nbrow, k*nbrow, j*nbrow, k*nbrow, j*nbrow, A, b);
        }
    }
}
```

Activities Text Editor Sat 1:24 PM

openmp_bellman_ford.cpp
~/Desktop/yes

Save

FinalAPSP.cpp x openmp_bellman_ford.cpp x

```
if (argc <= 2) {
    utils::abort_with_error_message("NUMBER OF THREADS WAS NOT FOUND!");
}
string filename = argv[1];
int p = atoi(argv[2]);

int *dist;
bool has_negative_cycle = false;

assert(utils::read_file(filename) == 0);
dist = (int *) malloc(sizeof(int) * utils::N);

//time counter
timeval start_wall_time_t, end_wall_time_t;
float ms_wall;

//start timer
gettimeofday(&start_wall_time_t, nullptr);

//bellman-ford algorithm
bellman_ford(p, utils::N, utils::mat, dist, &has_negative_cycle);

//end timer
gettimeofday(&end_wall_time_t, nullptr);
ms_wall = ((end_wall_time_t.tv_sec - start_wall_time_t.tv_sec) * 1000 * 1000
           + end_wall_time_t.tv_usec - start_wall_time_t.tv_usec) / 1000.0;

std::cerr.setf(std::ios::fixed);
std::cerr << std::setprecision(6) << "Time(s): " << (ms_wall / 1000.0) << endl;
utils::print_result(has_negative_cycle, dist);
free(dist);
free(utils::mat);

return 0;
}
```

C++ Tab Width: 8 Ln 128, Col 47 INS

```
Activities Text Editor Sat 1:27 PM
FinalAPSP.cpp
~/Desktop/yes

//Phase 3:updates the tiles which are remained in the main tile column.
#pragma omp parallel for
for(i=0;i<ntrow;i++)
{
    if(i==k){}
    else
    {
        updateSubmatrix(i*nbrow, k*nbrow,i*nbrow, k*nbrow,k*nbrow, k*nbrow,A,b);
    }
}

//Phase 4:finally it updates the remaining tiles except that main tile row and column of the matrix
#pragma omp parallel for collapse(2)
for(i=0;i<ntrow;i++)
{
    for(j=0;j<ntrow;j++)
    {
        if(i==k||j==k){}
        else
        {
            updateSubmatrix(i*nbrow, j*nbrow,i*nbrow, k*nbrow,k*nbrow, j*nbrow,A,b);
        }
    }
}

}
}
struct Graph {
    int **A;
    int VertexCount;
    int EdgeCount;
};
int main()
{
    int i,j;
```

```
C++ Tab Width: 8 Ln 119, Col 51 INS
Activities Text Editor Sat 1:27 PM
FinalAPSP.cpp
~/Desktop/yes

int t;
cout<<"Enter Number of Threads :";
cin>>t;
omp_set_num_threads(t);

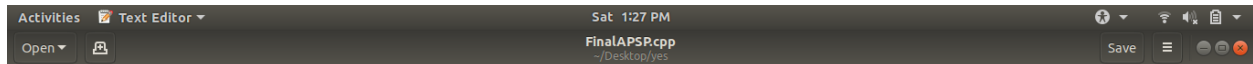
int n,dadd=0,dr;

Graph G;
int **D;
bool declared = false;
string GraphType, token;

cout << endl << "Taking Graph from sample3.txt and "<<INT_MAX<<" means Infinity :\n" << endl;
ifstream fl("./sample3.txt");
for(string line; getline(fl, line);)
{
    if (line[0] == 'c')
        continue;
    else if (line[0] == 'p')
    {
        istringstream iss(line);
        iss >> token >> GraphType >> G.VertexCount >> G.EdgeCount;
        n=G.VertexCount;
        dr=n%b;
        if(dr!=0)
            dadd=b-dr;

        if (G.VertexCount > 0)
        {
            G.A=new int*[G.VertexCount+dadd];
            D= new int*[G.VertexCount+dadd];
            for(int i=0;i<G.VertexCount+dadd;i++)
            {
                G.A[i]=new int[G.VertexCount+dadd];
                D[i]=new int[G.VertexCount+dadd];
            }

            #pragma omp parallel for collapse(2)
```

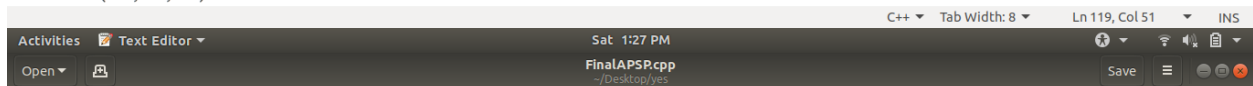


```
    }
    else if(declared && line[0] == 'a')
    {
        int w;
        istringstream iss(line);
        iss >> token >> i >> j >> w;
        G.A[i-1][j-1] = w;
    }
}

//Copying Graph 'G.A' to 'D' to apply standard floyd-warshall algorithm on 'D'.
#pragma omp parallel for collapse(2)
for(i=0;i<n+dadd;i++)
{
    for(j=0;j<n+dadd;j++)
    {
        D[i][j]=G.A[i][j];
    }
}

/* //'D' weight graph, Before Applying Standard Floyd-warshall Algorithm.
cout<<"Parallelized Standard Floyd-Warshall Algorithm :\n\n";
cout<<"Input Graph :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        cout<<D[i][j]<<" ";
    cout<<"\n";
}
*/

//Applying Parallelization to Standard Floyd-Warshall Algorithm and Calculating the Runtime for that Algorithm.
double tm0 = omp_get_wtime();
for(int k=0;k<n;k++)
#pragma omp parallel for collapse(2)
for(i=0;i<n;i++)
```



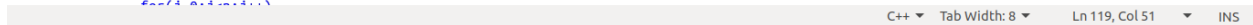
```
    }
}
tm0= omp_get_wtime() - tm0;

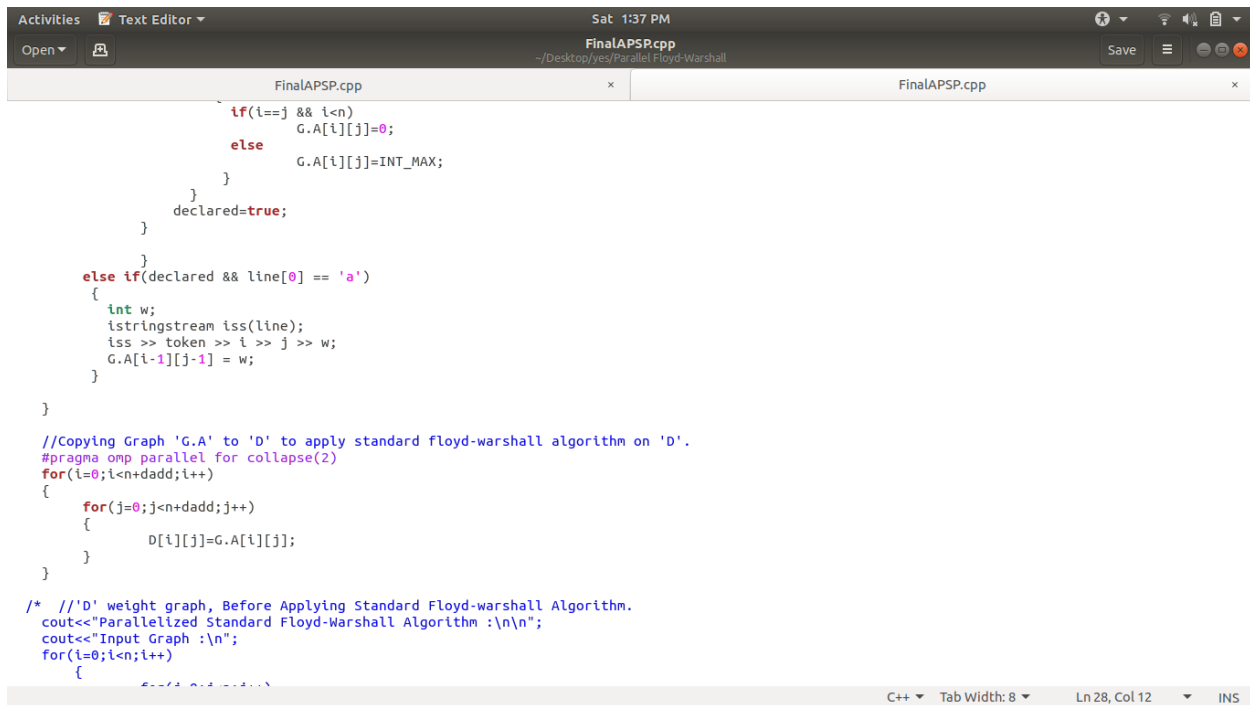
/* //'D' weight graph, After Applying Standard Floyd-warshall Algorithm i.e. Shortest path matrix.
cout<<"All Pairs Shortest Path Matrix :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        cout<<D[i][j]<<" ";
    cout<<"\n";
}
*/
cout << "Parallelized Standard FloydWarshall Algorithm Runtime is: " << tm0 << " seconds.\n\n";

/* //'G.A' weight graph, Before Applying Tiled Floyd-warshall Algorithm.
cout<<"Parallelized Tiled Floyd-Warshall Algorithm :\n\n";
cout<<"Input Graph :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        cout<<G.A[i][j]<<" ";
    cout<<"\n";
}
*/

//Applying Parallelization to Tiled Floyd-Warshall Algorithm and Calculating the Runtime for that Algorithm.
double tm = omp_get_wtime();
tiledFloydWarshall(G.A,n+dadd,b,t);
tm = omp_get_wtime() - tm;

/* //'G.A' weight graph, After Applying Tiled Floyd-warshall Algorithm i.e. Shortest path matrix.
cout<<"All Pairs Shortest Path Matrix :\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
```





```
FinalAPSP.cpp
~ /Desktop/yes/Parallel Floyd-Warshall
Save
FinalAPSP.cpp
FinalAPSP.cpp

    if(i==j && i<n)
        G.A[i][j]=0;
    else
        G.A[i][j]=INT_MAX;
    }
    declared=true;
}

}

else if(declared && line[0] == 'a')
{
    int w;
    istream iss(line);
    iss >> token >> i >> j >> w;
    G.A[i-1][j-1] = w;
}

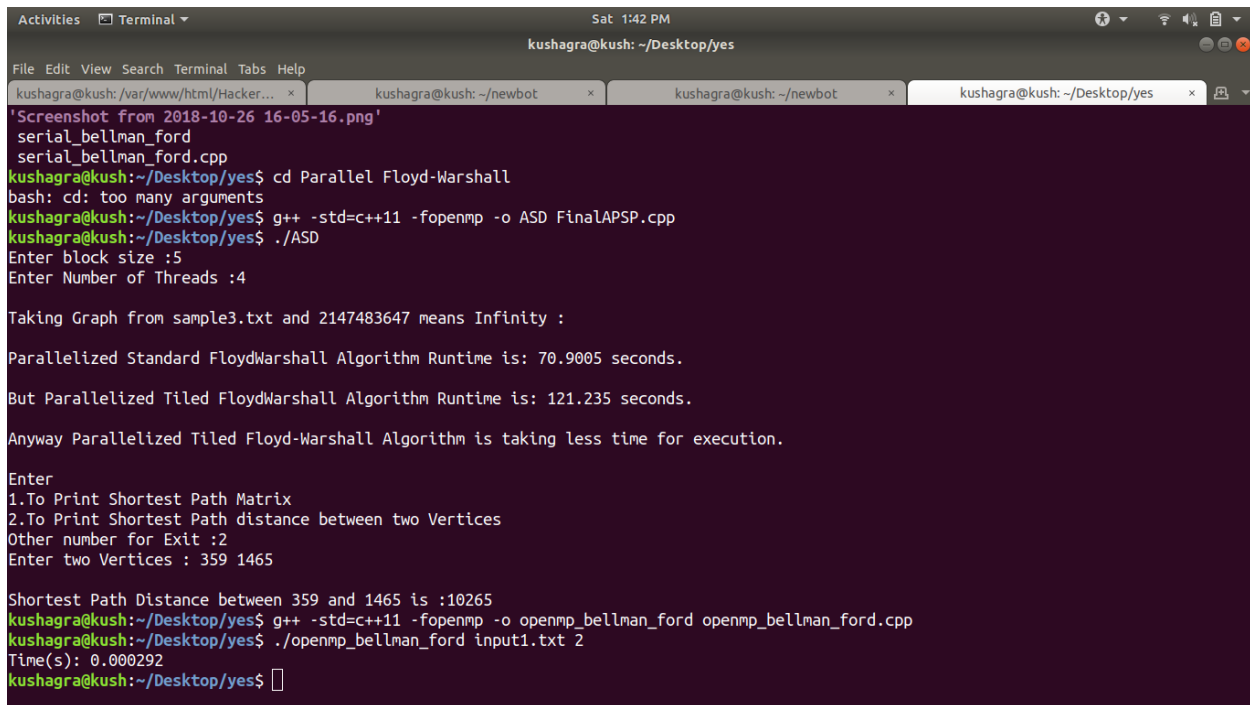
}

//Copying Graph 'G.A' to 'D' to apply standard floyd-warshall algorithm on 'D'.
#pragma omp parallel for collapse(2)
for(i=0;i<n+dadd;i++)
{
    for(j=0;j<n+dadd;j++)
    {
        D[i][j]=G.A[i][j];
    }
}

/* //'D' weight graph, Before Applying Standard Floyd-warshall Algorithm.
cout<<"Parallelized Standard Floyd-Warshall Algorithm : \n\n";
cout<<"Input Graph : \n\n";
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        D[i][j]=G.A[i][j];
    }
}
}

C++ Tab Width: 8 Ln 28, Col 12 INS
```

4)Results and Discussions



```
Activities Terminal
Sat 1:42 PM
kushagra@kush: ~/Desktop/yes
File Edit View Search Terminal Tabs Help
kushagra@kush: /var/www/html/Hacker... x kushagra@kush: ~/newbot x kushagra@kush: ~/newbot x kushagra@kush: ~/Desktop/yes x
'Screenshot from 2018-10-26 16-05-16.png'
serial_bellman_ford
serial_bellman_ford.cpp
kushagra@kush:~/Desktop/yes$ cd Parallel Floyd-Warshall
bash: cd: too many arguments
kushagra@kush:~/Desktop/yes$ g++ -std=c++11 -fopenmp -o ASD FinalAPSP.cpp
kushagra@kush:~/Desktop/yes$ ./ASD
Enter block size :5
Enter Number of Threads :4

Taking Graph from sample3.txt and 2147483647 means Infinity :

Parallelized Standard FloydWarshall Algorithm Runtime is: 70.9005 seconds.

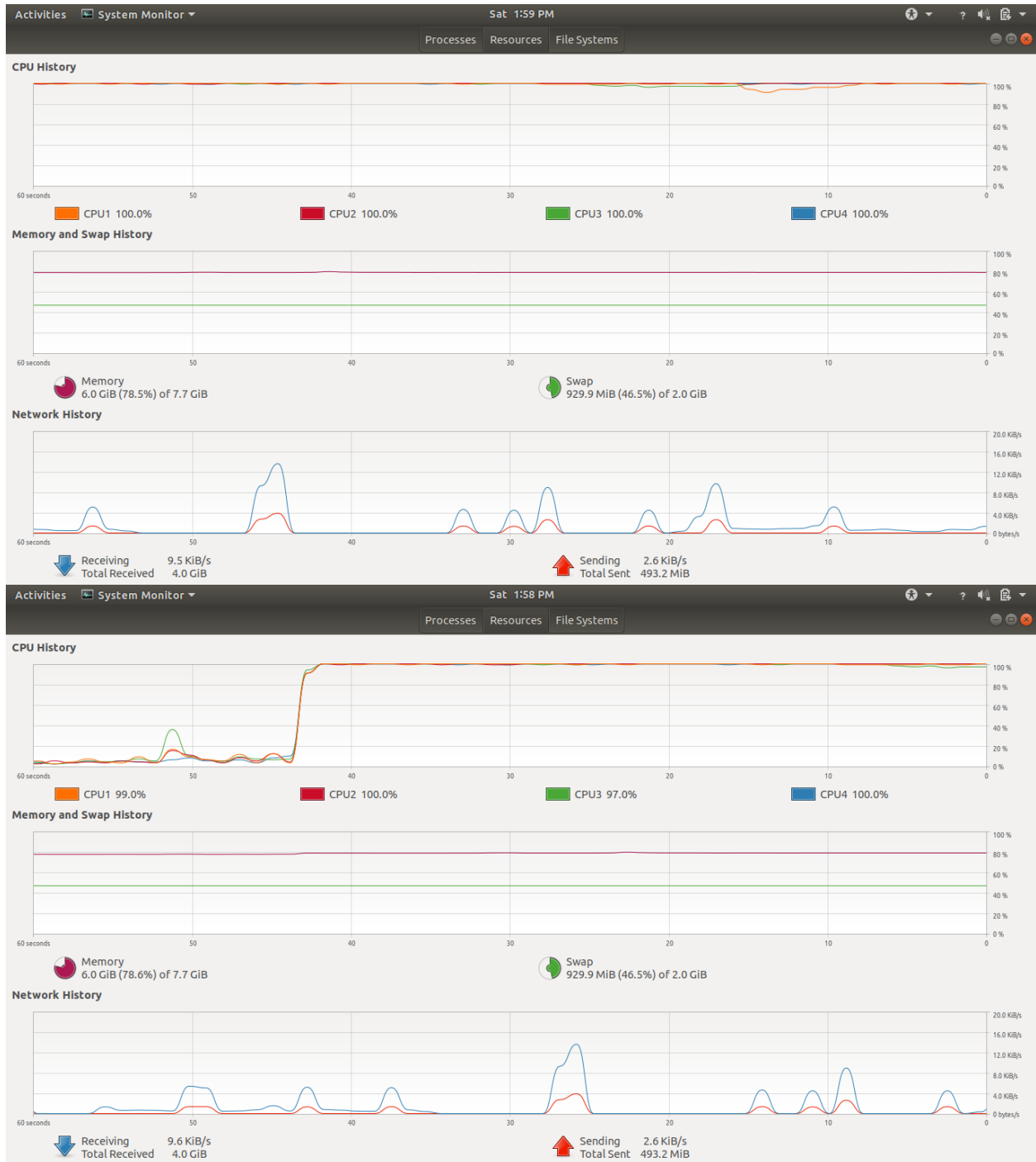
But Parallelized Tiled FloydWarshall Algorithm Runtime is: 121.235 seconds.

Anyway Parallelized Tiled Floyd-Warshall Algorithm is taking less time for execution.

Enter
1.To Print Shortest Path Matrix
2.To Print Shortest Path distance between two Vertices
Other number for Exit :2
Enter two Vertices : 359 1465

Shortest Path Distance between 359 and 1465 is :10265
kushagra@kush:~/Desktop/yes$ g++ -std=c++11 -fopenmp -o openmp_bellman_ford openmp_bellman_ford.cpp
kushagra@kush:~/Desktop/yes$ ./openmp_bellman_ford input1.txt 2
Time(s): 0.000292
kushagra@kush:~/Desktop/yes$
```


5) Parameters to be Considered





6) Conclusions

There is certainly a decrease in computation of Bellman Ford's Algorithm and Floyd Warshall's Algorithm when we use OpenMP +SIMD in the Algorithm as it uses all the 4 CPU's of the Computer as thought of. After parallelising the code was seen to be smoother and the output remained to result to us faster.

7) References

- T.H. Cormen, E.C. Leiserson, R.L. Rivest: Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 1990.
- A. Smiljanić, N. Maksić: Improving Utilization of Data Center Networks, IEEE Communication Magazine, Vol. 51, No. 11, Nov. 2013, pp. 32 – 38.
- A. Smiljanić, J. Chao, C. Minkenberg, E. Oki, M. Hamdi: Switching and Routing for Scalable and Energy-Efficient Networking, Vol. 32, No. 1, Jan. 2014, pp. 1 – 3.
- E.F. Moore: The Shortest Path through a Maze, International Symposium on the Theory of Switching, Cambridge, MA, USA, 02-05 April 1957
- E.W. Dijkstra: A Note on Two Problems in Connection with Graphs, Numerische Mathematik, Vol. 1, No. 1, Dec. 1959, pp. 269 – 271. [

- M.L. Fredman R.E. Tarjan: Fibonacci Heaps and their uses in Improved Network Optimization Algorithms, Journal of the ACM, Vol. 34, No. 3, July 1987, pp. 596 – 615. [
- D. Dundjerski, M. Tomasević: Graphical Processing Unit-based Parallelization of the Open Shortest Path First and Border Gateway Protocol Routing Protocols, Concurrency and Computation: Practice and Experience, Vol. 27, No. 1, Jan. 2015, pp. 237 – 251.
- J. Kim, W.J. Dally, S. Scott, D. Abts: Technology-driven, Highly-scalable Dragonfly Topology, International Symposium on Computer Architecture, Beijing, China, 21-25 June 2008, pp. 77 – 88.
- M. Al-Fares, A. Loukissas, A. Vahdat: A Scalable, Commodity Data Center Network Architecture, Conference SIGCOMM 2008, Seattle, WA, USA, 17-22 Aug. 2008, pp. 63 – 74.
- M. Besta, T. Hoefer: Slim Fly: A Cost Effective Low-diameter Network Topology, International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16-21 Nov. 2014, pp. 348 – 359.
- S. Mozes and C. Sommer. Exact distance oracles for planar graphs. SODA, 2012.
- G. Nannicini, P. Baptiste, G. Barbier, D. Krob, and L. Liberti. Fast paths in large-scale dynamic road networks. Computational Optimization and Applications, 2010.