FEBRUARY 28, 2018

CODE LISTINGS

OMP Dense Matrix

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char* argv[]) {




    //Declarations
    float dampingFactor = 0.15;
    int numPage = atoi(argv[1]);
    int totalSize = numPage * numPage;
    float *sArray, *pageRank, *yArray;
    sArray = (float*)malloc(totalSize*sizeof(float));
    pageRank = (float*)malloc(numPage*sizeof(float));
    yArray = (float*)malloc(numPage*sizeof(float));
    int i, j, K, k;
    K = 1000;
    double startTime, endTime;

    //fill out sArray with 0 and
    //Initial Pagerank vector with 1/Numpage
    for(i = 0; i < numPage; i++) {
        for (j = 0; j < numPage; j++) {
            sArray[i*numPage + j] = 0.0;
        }
        pageRank[i] = 1/(float)numPage;
    }

    //setup sArray with page navigation probabilities
    sArray[1] = 0.5;
    sArray[numPage - 1] = 0.5;
    for (i = 1; i < numPage - 1; i++) {
        for(j = 0; j < numPage; j++) {
            if (i == 1) {
                sArray[i*numPage] = 1.0;
                sArray[i*numPage+2] = 0.5;
                j = numPage;
            }
            else {
                if (j == i) {
                    sArray[(i*numPage) + (j - 1)] = 0.5;
                    sArray[(i*numPage) + (j + 1)] = 0.5;
                    j = numPage;
                }
            }
        }
    }
    sArray[totalSize - 2] = 0.5;

    //Apply damping factor to the sArray
```

```
    for (i = 0; i < numPage; i++) {
        for (j = 0; j < numPage; j++) {
            sArray[i*numPage+j] = ((1-
dampingFactor)*sArray[i*numPage+j])+(dampingFactor/numPage);
        }
    }

    //start timer and perform MatVec K-times in parallel
    startTime = omp_get_wtime();
    for (k = 0; k < K; k++) {
        #pragma omp parallel for private(j)
        for (i = 0; i < numPage; i++) {
            yArray[i] = 0.0;
            for (j = 0; j < numPage; j++) {
                yArray[i] += sArray[i*numPage+j] * pageRank[j];
            }
        }
        #pragma omp master
        for (i = 0; i < numPage; i++) {
            pageRank[i] = yArray[i];
        }
        #pragma end master
    }
    endTime = omp_get_wtime();

    //Print the Pageranks or max and min values
    if (numPage < 20) {
        for (i = 0; i < numPage; i++) {
            printf("%f \n", pageRank[i]);
        }
    }
    else {
       float max, min;
       max = pageRank[0];
       min = pageRank[0];
       for (i = 0; i < numPage; i++) {
            if (max < pageRank[i])
                max = pageRank[i];
            if (min > pageRank[i])
                min = pageRank[i];
        }
        printf("Min Pagerank = %f \n", min);
        printf("Max Pagerank = %f \n", max);
    }

    //print runtime
    printf("RUNTIME = %.16f\n", endTime-startTime);
    return 0;
}
```

OMP Sparse Matrix

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char* argv[]) {

    //Declarations
```

```
double dampingFactor = 0.15;
int numPage = atoi(argv[1]);
int totalSize = numPage * numPage;
double *aArray, *pageRank, *yArray;
int *iA, *jA;
aArray = (double*)malloc((2*numPage-1)*sizeof(double));
iA = (int*)malloc((numPage+1)*sizeof(int));
jA = (int*)malloc((2*numPage-1)*sizeof(int));
pageRank = (double*)malloc(numPage*sizeof(double));
yArray = (double*)malloc(numPage*sizeof(double));
int i, j, K, k;
K = 1000;
double startTime, endTime;

//setup the sArray
for (i = 0; i < 2*numPage - 1; i++) {
    if (i == 2 )
        aArray[i] = 1.0;
    else
        aArray[i] = 0.5;
}

//setup the iA and initialize pagerank to 1/Numpage
for (i = 0; i < numPage + 1; i++) {
    if (i == numPage)
        iA[i] = 2*numPage - 1;
    else if (i == 0)
        iA[i] = 0;
    else
        iA[i] = iA[i-1] + 2;
    if (i < numPage)
        pageRank[i] = 1/(double)numPage;
}

//setup jA
jA[0] = 1; jA[1] = numPage -1; jA[2] = 0; jA[3] = 2;
for (i = 4; i < 2*numPage-1; i++) {
    jA[i] = jA[i -2] + 1;
}

//start timer and perform MatVec on the CSR K-times
startTime = omp_get_wtime();
for (k = 0; k < K; k++) {
    #pragma omp parallel for private(j)
    for (i = 0; i < numPage; i++) {
        yArray[i] = 0.0;
        for (j = iA[i]; j < iA[i+1]; j++) {
            yArray[i] += aArray[j] * pageRank[jA[j]];
        }
    }
    #pragma omp master
    for (i = 0; i < numPage; i++) {
        //apply damping
        yArray[i] = ((1 - dampingFactor)*yArray[i]) + (dampingFactor / numPage);
        pageRank[i] = yArray[i];
    }
    #pragma end master
}
endTime = omp_get_wtime();
```

```
    //print pagerank or max and min values
    if (numPage < 20) {
        for (i = 0; i < numPage; i++) {
            printf("PR(%d) = %f \n", i, pageRank[i]);
        }
    }
    else {
        double max, min;
        max = pageRank[0];
        min = pageRank[0];
        for (i = 0; i < numPage; i++) {
            if (max < pageRank[i])
                max = pageRank[i];
            if (min > pageRank[i])
                min = pageRank[i];
        }
        printf("Min Pagerank = %f \n", min);
        printf("Max Pagerank = %f \n", max);
    }

    //print runtime
    printf("RUNTIME = %.16f\n", endTime - startTime);
    return 0;
}
```

MPI Dense Matrix

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    //Declarations
    double dampingFactor = 0.15;
    int numprocs, myid, i, j, k, K;
    int numPage = 1600;
    int totalSize = numPage * numPage;
    int *nAlloc;
    double *sArray, *pageRank, *yArray, *globalPageRank;

    //Initilize MPI variables
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Status status;

    //nAlloc hold the local row size
    double startTime, endTime;
    K = 1000;
    nAlloc = (int*)malloc(numprocs*sizeof(int));

    //find partion positions
    int minPartition = numPage/numprocs;
    for (i = 0; i < numprocs; i++)
        nAlloc[i] = minPartition;

    int remainder = numPage%numprocs;
    for(i = 0; i < remainder; i++)
```

```
        nAlloc[i] += 1;

    //setup array space in each processor
    sArray = (double*)malloc(nAlloc[myid]*numPage*sizeof(double));
    yArray = (double*)malloc(nAlloc[myid]*sizeof(double));
    pageRank = (double*)malloc(nAlloc[myid]*sizeof(double));
    globalPageRank = (double*)malloc(numPage*sizeof(double));

    //initialize Array Spaces
    for(i = 0; i < nAlloc[myid]; i++) {
        for(j = 0; j < numPage; j++) {
            sArray[i*numPage+j] = 0.0;
        }
        pageRank[i] = 1/(double)numPage;
    }

    //Fill sArray with page navigation probabilities
    for (i = 0; i < nAlloc[myid]; i++) {
        if (myid == 0 && i == 0) {
            sArray[1] = 0.5;
            sArray[numPage-1] = 0.5;
            sArray[numPage] = 1.0;
            sArray[numPage+2] = 0.5;
            i++;
        }
        else if (myid == (numprocs-1) && i == (nAlloc[myid]-1)) {
            sArray[i*numPage+numPage-2] = 0.5;
        }
        else {
            int globalRow = myid*nAlloc[myid]+i;
            sArray[i*numPage+globalRow-1] = 0.5;
            sArray[i*numPage+globalRow+1] = 0.5;
        }
    }

    //Apply damping factor on sArray
    for (i = 0; i < nAlloc[myid]; i++) {
        for (j = 0; j < numPage; j++) {
            sArray[i*numPage+j] = ((1-dampingFactor)*sArray[i*numPage+j]) +
(dampingFactor/numPage);
        }
    }

    //Start time and perform MatVec K-times
    startTime = MPI_Wtime();
    for (k = 0; k < K; k++) {
        MPI_Allgather(pageRank, nAlloc[myid], MPI_DOUBLE, globalPageRank, nAlloc[myid],
MPI_DOUBLE, MPI_COMM_WORLD);
        for (i = 0; i < nAlloc[myid]; i++) {
            yArray[i] = 0.0;
            for(j = 0; j < numPage; j++) {
                yArray[i] += sArray[i*numPage+j]*globalPageRank[j];
            }
        }
        for (i = 0; i < nAlloc[myid]; i++) {
            pageRank[i] = yArray[i];
        }
    }
    endTime = MPI_Wtime();
```

```
    //Gather the last calculations
    MPI_Allgather(pageRank, nAlloc[myid], MPI_DOUBLE, globalPageRank, nAlloc[myid], MPI_DOUBLE,
MPI_COMM_WORLD);

    //Print the Pageranks using Processor 0
    if (myid == 0) {
        if (numPage < 20) {
            for (i = 0; i < numPage; i++) {
                printf("PR(%d) = %f \n", i, globalPageRank[i]);
            }
        }
        else {
            double max, min;
            max = globalPageRank[0];
            min = globalPageRank[0];
            for (i = 0; i < numPage; i++) {
                if (max < globalPageRank[i])
                    max = globalPageRank[i];
                if (min > globalPageRank[i])
                    min = globalPageRank[i];
            }
            printf("Min Pagerank = %f \n", min);
            printf("Max Pagerank = %f \n", max);
        }
        //print the runtime
        printf("RUNTIME = %.16f\n", endTime - startTime);
    }
    MPI_Finalize();
    return 0;
}
```

PROBLEM 1: SMALL PROBLEM

NumPage = 16

Note: Answer below is same for OMP A, OMPB, and MPI C. Tested on 1, 2, 4, and 8 processes.

```
PR(0) = 0.062500
PR(1) = 0.097289
PR(2) = 0.081856
PR(3) = 0.073255
PR(4) = 0.068449
PR(5) = 0.065744
PR(6) = 0.064183
PR(7) = 0.063215
PR(8) = 0.062500
PR(9) = 0.061785
PR(10) = 0.060818
PR(11) = 0.059257
PR(12) = 0.056551
PR(13) = 0.051746
PR(14) = 0.043144
PR(15) = 0.027711
```

PROBLEM 2: LARGE PROBLEM

OMP Dense Matrix

Problem Size (numPage)	CPU per Task (numThread)	Runtime	Speedup
1600	1	19.8216404459672049	1.0000
1600	2	19.4074936889810488	1.0213
1600	4	9.9788244770024903	1.9864
1600	8	5.2312034509959631	3.7891

OMP Sparse Matrix

Problem Size (numPage)	CPU per Task (numThread)	Runtime	Speedup
1600	1	0.0880342739829794	1.0000
1600	2	0.0864430639776401	1.0184
1600	4	0.0579239950166084	1.5198
1600	8	0.0461550360196270	1.9074

MPI Dense Matrixw

Problem Size (numPage)	Number of Process (numTasks)	Runtime	Speedup
1600	1	15.8162488937377930	1.0000
1600	2	15.5464968681335449	1.0174
1600	4	7.8547549247741699	2.0136
1600	8	4.1564948558807373	3.8052
1600	16	2.2458489385284957	7.0424

Values from Pagerank Vector (numPage = 1600)

```
Minimum Pagerank = 0.000277
Maximum Pagerank = 0.000973
```

DISCUSSIONS

Part 1

There was no perfect speedup in any of the three approaches used. However, the MPI C approach delivered better speedup than the OpenMP methods. For OpenMP approaches, we are unable to see a perfect speedup because of the overhead associated with threads. Basically, the speedup is affected by thread creation and management overhead. The thread creation overhead occurs when the system starts the threads following the directive "`#pragma omp parallel for private(j)`." The management overhead occurs during entire life cycle of the thread as the system manage waits and joins of different threads asynchronously.

To achieve better speedup, we can change the scheduling of the change threads from the default static scheduling to dynamic.

Part 2

The three approaches were pretty much straightforward. The hardest of among then was part C and the easiest was part A. The major issue with part C is the splitting of the array into different processors and translating the values properly to match the original array. In terms of difficulty, part B is closely similar to part A. The major effort went to figuring out the pattern of the sparse matrix.

The best performance was exhibited by part B. The runtime of part B on one thread is relatively lower than the best times of part A in 8 threads and part C in 16 processors. Generally, part C performed faster than part A leaving A with the worst performance. The speed of part B is expected since the use of sparse matrix, we achieve $1/10^{th}$ of the computation required for part A or C.