Matrix Multiplication

Implement with the “Divide and Conquer” idea.

A is an a x b matrix, B is an b x c matrix.

- If a<=b: divide A horizontally into “numofthreads” parts and multiply each part with matrix B

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

- Else: divide B vertically into “numofthreads” parts and multiply matrix A with each submatrix of B

$$C = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$$

Detailed Implementation

1, The splitting of matrix -- modifying the traditional matrix multiplication

Traditional matrix multiplication

```
for(int i=0;i<n1;i++){
    for(int k=0;k<n3;k++){
        c[i][k] = 0;
        for (int j=0;j<n2;j++){
            c[i][k] += a[i][j]*b[j][k];
        }
    }
}
```

Modified matrix multiplication with splitted matrices

```
for(int i=nralower;i<nraupper;i++){
    for(int k=ncblower;k<ncbupper;k++){
        out[i][k] = 0;
        for (int j=0;j<nca;j++){
            out[i][k] += a[i][j]*b[j][k];
        }
    }
}
```

Detailed Implementation -- Real Algorithm

Determine which matrix to split

Set the upper limit of division for the current loop and check whether it exceeds the number of rows/columns

Split the selected matrix in each thread according to the lower limit (loop variable i) and predetermined upper limit, and multiply with the other complete matrix

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    if (nra>ncb) {
        int split = nra;
    } else {
        int split = ncb;
    }
    for (int i=id; i<split; i+=nt) {
        if ((i+nt)<=split) {
            int uplimit = i+nt;
        } else {
            int uplimit = split;
        }
        if (split==nra) {
            basic_matrix_multiply(i, uplimit, nca, 0, ncb, a, b, output);
        } else {
            basic_matrix_multiply(0, nra, nca, i, uplimit, a, b, output);
        }
    }
}
```

Other Matrix Functions

1, Right Multiply: Multiply a vector with a matrix (vector a times matrix b)

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    for (int j = id; j < ncb; j += nt) {
        for (int i = 0; i < da; i++) {
            output[j] += a[i] * b[i][j];
        }
    }
}
```

First parse through the columns of matrix b with omp parallel

2, Left Multiply: Multiply a matrix with a vector (matrix a times vector b)

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    for (int i = id; i < nra; i += nt) {
        for (int j = 0; j < db; j++) {
            output[i] += a[i][j] * b[j];
        }
    }
}
```

First parse through the rows of matrix a with omp parallel

Other Matrix Functions

1, Dot Product (vector a times vector b)

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int inc = omp_get_num_threads();
    double temp = 0;
    for (int i = id; i < dim; i += inc) {
        temp += a[i] * b[i];
    }
    #pragma omp critical
    {
        *output += temp;
    }
}
```

Parse through each entry of the vector with multi-threads

A critical session to update the result with the product of each pair of the entries

2, Find Max (find the entry with maximum value in a vector)

```
*max = vector[0];
*max_ind = 0;
#pragma omp parallel for
for (int i = 1; i < dim; i++) {
    #pragma omp critical
    {
        if (vector[i] > *max) {
            *max_ind = i;
            *max = vector[i];
        }
    }
}
```

Parse through each entry of the vector. Achieve the same effect as the method in class with only one layer of loop

A critical session to check whether the current processed entry is larger than the existing max. If true, update the max value and its index with the entry's.

Matrix Inverse -- Gaussian Elimination

```c
int matrix_invert(int dim, double** a, double** output) {
    if (a == NULL || output == NULL) {
        puts("matrix invert NULL ptr");
        return 1;
    }
    matrix_copy(dim, dim, a, output);
    double** i_matrix = create_identity(dim);
    for (int i = 0; i < dim; i++) {
        if (output[i][i] == 0) {
            int j;
            for (j = i; j < dim && output[j][i] == 0; j++);
            if (output[j][i] == 0) {
                return -1;
            }
            swap_row(i, j, output, output);
        }
        double scale = output[i][i];
        #pragma omp parallel for
        for (int j = 0; j < dim; j++) {
            output[i][j] = output[i][j] / scale;
            i_matrix[i][j] = i_matrix[i][j] / scale;
        }
        #pragma omp parallel for
        for (int j = i + 1; j < dim; j++) {
            double factor = output[j][i];
            #pragma omp parallel for
            for (int k = 0; k < dim; k++) {
                output[j][k] = output[j][k] - factor * output[i][k];
                i_matrix[j][k] = i_matrix[j][k] - factor * i_matrix[i][k];
            }
        }
    }
```

Matrix Inverse -- Back Substitution

```
for (int i = dim - 1; i > 0; i--) {
    #pragma parallel for
    for (int j = i - 1; j > -1; j--) {
        double factor = output[j][i];
        #pragma omp parallel for
        for (int k = 0; k < dim; k++) {
            output[j][k] = output[j][k] - factor * output[i][k];
            i_matrix[j][k] = i_matrix[j][k] - factor * i_matrix[i][k];
        }
    }
}
```

Bellman-Ford Shortest Path Algorithm

Algorithm to find shortest path and distance from source node to all other nodes in the graph.

Relaxes edge distances between graphs, if new relaxation provides a shorter path to the node, the distance is updated.

In Serial

```
for (int i = 0; i <= V-1; i++){
    listNode *currentNode = graph->array[i].head;
    for(int j = 0; j<graph->array[i].size; j++){
        int u = currentNode->vertexNumber;
        int weight = currentNode->weight;

            if (StoreDistance[u][1] == -1 || StoreDistance[i][1] + weight < StoreDistance[u][1]){
                StoreDistance[u][1] = StoreDistance[i][1] + weight;
            }
            currentNode = currentNode->nextListNode;

    }
}
```

Parallelizing Bellman-Ford

Parallelize the relaxation process

Compares known distance to neighbor with the distance including the new edge for each of the edges in parallel. Each neighbor receives its own processor

In parallel

```
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if(id==0) nthreads = nthrds;
    for(int j = id; j < V; j+=nthrds){
        for (listNode* current = graph->array[j].head; current != NULL; current = current->nextListNode) {
            int vNum = current->vertexNumber;

            if(StoreDistance[vNum][1] == -1 || StoreDistance[j][1] + current->weight < StoreDistance[vNum][1]){
                #pragma omp critical
                {
                    StoreDistance[vNum][1] = StoreDistance[j][1] + current->weight;
                }
```

```
Vertex  Distance from Source Vertex
0               0
1               420
2               175
3               570
4               670
EROSENSTROM@Erik-PC:/mnt/c/Users/Erik/eclipse-workspace/4010 project$
```

Time Analysis of Bellman-Ford vs Simplex

Ultimately, we will compare the runtime of the Bellman-Ford algorithm and the simplex method

We will also analyze how the runtimes compare to the theoretical times

Question?
Comments?
Concerns?