Assignment – 2

Question – 4

Read the paper provided on Sparse Matrix-Vector (SpMV) reordering optimizations performed on multicore processors. (only parts a and b are required)

a. Select one of the reordering schemes described and provide pseudocode (i.e., detailed steps and describe the transformations) for 2 of the schemes described in the paper.
b. For 1 of the hardware platforms discussed in the paper, provide the details of the associated memory hierarchy on the system
c. Develop your own SpMV (sparse matrix-vector multiplication) implementation that uses one of the memory reordering algorithms described in the paper (6 are provided). You will need to generate your own sparse matrices or use those available from MatrixMarket (https://math.nist.gov/MatrixMarket/). Report on the speedup achieved as compared to using the standard dense SpMV kernel approach

   Students can earn an extra 10 points of extra credit for implementing and evaluating each memory reordering algorithm (60 maximum points).

Answer:

Here I would choose Nested Dissection (ND) ordering and Graph Partitioning (GP) ordering.

4 (a):

1). Nested Dissection Ordering Pseudocode

```
function ND_Reordering(matrix A):
   if size(A) is small:
      return BaseCaseOrdering(A)

   // Step 1: Build the graph representation from the symmetric matrix A.
   G = BuildGraphFromMatrix(A)

   // Step 2: Compute a vertex separator S that splits G into two roughly equal subgraphs.
   (G1, G2, S) = ComputeVertexSeparator(G)

   // Step 3: Recursively compute the ordering for the two subgraphs.
   order1 = ND_Reordering(Submatrix corresponding to G1)
```

```
    order2 = ND_Reordering(Submatrix corresponding to G2)

    // Step 4: Concatenate the orderings with the separator placed last.
    ND_order = Concatenate(order1, order2, S)

    return ND_order
```

2). Graph Partitioning (GP) based ordering pseudocode:

```
function GP_Reordering(matrix A, num_partitions):
    // Step 1: Construct the graph representation from the matrix A.
    G = BuildGraphFromMatrix(A)

    // Step 2: Partition the graph using a multilevel partitioning algorithm (e.g., METIS).
    // PartitionIDs[v] gives the partition index for vertex v.
    PartitionIDs = METIS_Partition(G, num_partitions)

    // Step 3: Group vertices by their partition IDs.
    partitions = [empty list for each partition in 1..num_partitions]
    for each vertex v in G:
        p = PartitionIDs[v]
        Append v to partitions[p]

    // Step 4 (Optional): Locally reorder vertices within each partition
    // to further enhance data locality (e.g., sort by degree).
    for each partition in partitions:
        partition = LocalReorder(partition)

    // Step 5: Concatenate the partitions to form a global ordering.
    global_order = Concatenate(partitions[1], partitions[2], ..., partitions[num_partitions])

    return global_order
```

4 (b):

Memory Hierarchy Details for the Cavium TX2 (CN9980) Platform
One of the hardware platforms evaluated in the paper is the Cavium TX2 CN9980 processor. Key details from Table 2 in the paper are:

- **Processor:** Cavium TX2 CN9980
- **Instruction Set:** ARMv8.1
- **Microarchitecture:** Vulcan
- **Sockets & Cores:** 2 sockets with 32 cores each (totaling 64 cores)
- **Clock Frequency:** Ranges from 2.0 GHz to 2.5 GHz
- **Cache Hierarchy:**
  - **L1 Cache:** Each core has a 32 KiB instruction cache and a 32 KiB data cache
  - **L2 Cache:** 256 KiB per core.
  - **L3 Cache:** Shared among cores on a socket, with 32 MiB per socket.
- **Memory Bandwidth:** Approximately 342 GB/s

The ARM architecture of the TX2 platform makes it different from regular x86 computers in a number of ways. It has a smaller L2 cache per core than some Intel or AMD chips, and its L3 cache is only 32 MiB per socket, which isn't very much. The machine has 64 cores spread across 2 sockets and a memory bandwidth of 342 GB/s, so it can handle a lot of work at once. Even though sparse matrix-vector multiplication (SpMV) usually uses a lot of memory, reordering can still have big benefits, like making data more local and reducing cache misses. The smaller cache sizes on TX2, on the other hand, may make tuning and software optimizations even more important for getting the most out of the hardware.

4 (c):

Implementing SpMV with memory reordering algorithms involves selecting one of six algorithms, creating sparse matrices, and comparing execution speed to standard dense SpMV. This code uses sparse matrix-vector multiplication using the Reverse Cuthill-Mckee (RCM) algorithm to reorder matrices for optimized performance.

Below is a step-by-step guide on how to make your own sparse matrix-vector multiplication (SpMV) code that uses one of the six reordering algorithms described in the paper, such as Reverse Cuthill–McKee:

- Read a sparse matrix (Matrix Market file) or make one.
- Change the matrix into a graph so that it can be rearranged.
- Use a method for reordering (in this case, Reverse Cuthill–McKee).
- Remap the matrix's rows and columns so that they are in that order.
- Save the matrix that has been rearranged in a sparse format (CSR).
- Do SpMV and see how much faster it is than a simple dense-matrix multiplication method.

Features:

- **Matrix Reading:** Parses .mtx files in the Matrix Market format.
- **RCM Ordering:** Applies the RCM algorithm to reorder the matrix.
- **SpMV Computation:** Performs sparse matrix-vector multiplication using both the original and reordered matrices.
- **Performance Comparison:** Compares performance between the original and reordered matrices

Output:

```
[prayags@pi ~]$ nano spmv.cpp
[prayags@pi ~]$ g++ -O0 -g -fopenmp spmv.cpp -o spmv
[prayags@pi ~]$ nano toy.mtx
[prayags@pi ~]$ ./spmv toy.mtx
Matrix loaded: 4 x 4, nnz=5
RCM ordering took: 1.95438e-05 seconds.
Applying permutation took: 1.23642e-05 seconds.

CSR original SpMV time (avg): 4.85746e-05 s
CSR RCM SpMV time (avg):      5.32381e-05 s
Naive dense matvec time (avg): 5.50119e-05 s
Speedup vs dense = 1.03332
Speedup vs original CSR = 0.912403
```

Implementing and evaluating each memory reordering algorithm:

1: Reverse Cuthill-McKee (RCM): It is a graph-based reordering method used to reduce the bandwidth of sparse matrices. By reversing the node ordering produced by the Cuthill-McKee algorithm, RCM clusters non-zero elements closer to the diagonal, optimizing data locality and minimizing memory usage. This enhances computational efficiency in numerical operations like solving linear systems, matrix factorization, and finite element analysis, where sparse matrix structures are common. The algorithm is particularly valued for improving performance in large-scale scientific and engineering simulations.

Output:

```
[prayags@pi ~]$ nano rcm.cpp
[prayags@pi ~]$ g++ -O2 -o rcm rcm.cpp
[prayags@pi ~]$ ./rcm
RCM order: 4 2 1 3 0
```

2: Nested Dissection (ND): It is a divide-and-conquer technique that is used to quickly solve big sparse linear systems. It works best with finite element methods and graph-based problems. It works by repeatedly splitting a graph or matrix into smaller problems using separators, which are groups of nodes that, when taken away, make the graph into independent subgraphs of about the same size. This method cuts down on fill-in, which are nonzero elements added during factorization, and makes straight solvers for sparse matrices more efficient.

```
[prayags@pi ~]$ ./nd
6 6
0 1
1 2
2 3
2 4
4 5
1 5
Nested Dissection ordering:
0 2 3 5 1 4
```

3: Graph-Partitioning: It is the process of separating a graph into smaller, about equal-sized subgraphs while reducing the number of edges connecting the subgraphs. Efficient partitioning improves the efficiency of algorithms that work with big graphs.

Here the graph partitioning is done using Kernighan-Lin partitioning algorithm and in the input command the first line contains 2 integers where the number of vertices is n and number of edges is m. The further lines contains 2 integers u and v indicating that there is a edge between vertices u and v.

In this example, the graph has vertices numbered from 0 to 4, and the edges connect the vertices as specified. The graph must be undirected, so each edge implicitly connects both directions (u to v and v to u), and the indices of the vertices should be zero-based (i.e., starting from 0).

```
[prayags@xi ~]$ ./gp
5 6
0 1
0 2
1 2
1 3
2 4
3 4
Initial cut: 2
Final cut cost: 2
Partition (vertex -> A/B):
0 B
1 A
2 A
3 B
4 B
```

4: Hypergraph partitioning (HP): It is the process of dividing a hypergraph into smaller, balanced sub-hypergraphs while minimizing the number of hyperedges that connect different partitions. Unlike traditional graph partitioning, where edges connect only two vertices, a hypergraph allows an edge (called a hyperedge) to connect multiple vertices, making it a more general and flexible model for complex relationships.

In the output below, in row 1 of the input; 6 denotes the vertices and 3 denotes the hyperedges
$HE_0 = \{0, 1, 2\}$
$HE_1 = \{2, 3, 5\}$
$HE_2 = \{1, 4, 5\}$

```
[prayags@rho ~]$ nano hypgraph.cpp
[prayags@rho ~]$ g++ -O2 hypgraph.cpp -o hypgraph
[prayags@rho ~]$ ./hypgraph
6 3
3 0 1 2
3 2 3 5
3 1 4 5
Initial cut: 3
Iteration 0, cut => 0
Final cut cost: 0
Partition:
0 B
1 B
2 B
3 B
4 B
5 B
```

5: **Gray Code Ordering**: It is a binary numbering system where consecutive numbers differ by only one bit.

The below output is obtained when in the input format you put the matrix with n = 5 rows and c = 8 colums

[5 8
2 1 3
3 0 2 4
2 2 7
3 1 2 3
1 5]

First row interprets as 5 rows and 8 colums
Row 0 has 2 nonzeros: col1, col3
Row 1 has 3 nonzeros: col0, col2, col4
Row 2 has 2 nonzeros: col2, col7
Row 3 has 3 nonzeros: col1, col2, col3
Row 4 has 1 nonzero: col5

```
[prayags@rho ~]$ nano gray_code.cpp
[prayags@rho ~]$ g++ -O2 gray_code.cpp -o gray_code
[prayags@rho ~]$ ./gray_code
5 8
2 1 3
3 0 2 4
2 2 7
3 1 2 3
1 5
Gray code ordering (threshold=20):
3 0 1 4 2
```