COMP3467 Advanced Computer Systems
Parallel programming and system administration
Lecturer:Laura Morgenstern
Durham University

Hien Ha
Student ID: 000931275

1.1

```
// Timer Init
itime = omp_get_wtime();

generateMagicSquare(pattern, modifier, magicSquare, N, M);

gMSe = omp_get_wtime();

bool is_magic_square = isMagicSquare(magicSquare, M);

//------------------------------------//
//BOOL FOR DETERMINING MAGIC SQUARE----//
//------------------------------------//

// Timer end
ftime = omp_get_wtime();

// Timer print out
printf("\n");
printf("generateMagicMatrix computation time: %.15f\n", gMSe - itime);
printf("isMagicSquare computation time: %.15f\n", ftime - gMSe);
printf("Total computation time: %.15f\n", ftime - itime);
```

```
sumRow computation time: 0.000001917481422
sumColumn computation time: 0.000001057386398
isEqual computation time: 0.000000002384186
isPairwiseDistinct computation time: 59.494616985321045

generateMagicMatrix computation time: 0.003712177276611
isMagicSquare computation time: 59.495847940444946
Total computation time: 59.499560117721558
146365 146366 146392 ...
 .
 .
 .
93706 93835 93834
Generated matrix is a magic square.
```

Above is an example of a timer that was used to measure magic_matrix.cpp performance. The function isMagicSquare requires the most computation time, but to be exact, it would be isPairwiseDistinct. The computation time of sumRow and sumColumn is the average of the N times the functions have been called, while isPairwiseDistinct is timed innately from within the function. The computation time in tabular form:

Function Name	Computation Time(s)
sumRow	0.000001917481422
sumColumn	0.000001057386398
isEqual	0.00000002384186
isPairwiseDistinct	59.494616985321045
generateMagicMatrix	0.003712177276611
isMagicSquare	59.495847940444946
Total:	59.499560117721558

1.3

Porting code to GPU using #pragma omp target leverages parallelism to allow GPU computations. #pragma omp parallel for employs parallelising for loops and distributing workload between threads. Mapping is used to copy data to and from GPU.

```
#pragma omp target map(tofrom:magicSquare[:M][:M], modifier[:N][:N], pattern[:N][:N])
{
    if(omp_is_initial_device())
    {
      printf("Running on CPU\n");
    }
    #pragma omp parallel for collapse(2) schedule(static, CHUNK_SIZE)
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            modifier[i][j] *= M;
        }
    }

    #pragma omp barrier
```

For nested loops above, the collapse construct is used to collapse 2 loops to one, reducing overhead. Static scheduling is used to enhance load balancing by distributing loop iterations based on chunk size.

```
#pragma omp parallel for collapse(2) shared(magicSquare, pattern, modifier) private(iOuter, jOuter)
```

Other constructs are: shared, where variables are shared amongst threads to enhance memory management and reduce data transfer overhead; private, which ensures thread safety by creating private copies of variables for each thread, preventing data races, avoiding contention; reduction, which simplifies parallel code, ensuring thread-safe accumulation, enhancing performance in reductions.

```
#pragma omp target teams distribute parallel for reduction(+:main_diag_sum) map(to:matrix[:N][:N]) schedule(static, CHUNK_SIZE)
// compute sum of elements on main diagonal
for (int i = 0; i < N; i++)
{
    main_diag_sum += matrix[i][i];
}
int row_sum = row_sums[0];
if (main_diag_sum != row_sum) return false;
```

Using cache blocking/matrix tiling, matrices are broken down into smaller blocks to improve cache locality and reduce cache misses.

```
// VERSION 3
int iOuter, jOuter;
//------------------------------------------------------------------
// OpenMP here!!!---------------------------------------------------
#pragma omp parallel for collapse(2) shared(magicSquare, pattern, modifier) private(iOuter, jOuter)
for (iOuter = 0; iOuter < M; iOuter += CHUNK_SIZE)
{
    for (jOuter = 0; jOuter < M; jOuter += CHUNK_SIZE)
    {
        for (int i = iOuter; i < iOuter + CHUNK_SIZE && i < M; i++)
        {
            int patternRow = i % N;
            int modifierRow = i / N;
            int* patternRowPtr = pattern[patternRow];
            int* modifierRowPtr = modifier[modifierRow];
            for (int j = jOuter; j < jOuter + CHUNK_SIZE && j < M; j++)
            {
                int patternCol = j % N;
                int modifierCol = j / N;
                magicSquare[i][j] = patternRowPtr[patternCol] + modifierRowPtr[modifierCol];
            }
        }
    }
}
```

Compared to:

```
for (int i = 0; i < M; i++)
{
    for (int j = 0; j < M; j++)
    {
        int patternRow = i % N;
        int patternCol = j % N;
        magicSquare[i][j] = pattern[patternRow][patternCol];
        magicSquare[i][j] += modifier[i/N][j/N];
    }
}
```

Function isPairwiseDistinct utilises nested #pragma omp parallel for directives, with the usage of a reduction clause with logical OR operator to reduce local boolean values in each thread into a global one.

```
bool isPairwiseDistinct( int** matrix, int N) {
    double s,e;
    s = omp_get_wtime();
    bool result = false;
    #pragma omp target teams distribute parallel map(to:matrix[:N][:N])
    {
        #pragma omp parallel for collapse(2) shared(result) schedule(static, CHUNK_SIZE)
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                int currentElement = matrix[i][j];

                #pragma omp parallel for collapse(2) reduction(||:result) schedule(static, CHUNK_SIZE)
                for (int row = 0; row < N; row++) {
                    for (int col = 0; col < N; col++) {
                        if (row != i || col != j) {
                            int otherElement = matrix[row][col];
                            if (currentElement == otherElement) {
                                {
                                    result = true;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    e = omp_get_wtime();
    printf("isPairwiseDistinct: %.15f\n", e - s);
    return result;
```

The best runtime for GPU code for N=20 recorded is: 0.241568088531494s, compared to best case: 52.17411s for sequential runtime.

2.2

With MPI to distribute the workload among the MPI processes, magic_matrix.cpp can be parallelised. MPI_Init, MPI_Comm_rank, MPI_Comm_size initialise MPI. For data distribution, the magic matrix can be distributed between processes where each process run on a subset of the matrix with size (N*N) * (N*N): rank 0 process row 0 to (N * N)/(num_ranks - 1), rank 1 work row (N * N)/num_ranks to 2(N * N)/(num_ranks - 1), and so on.

Each process gets the entire pattern and the relevant row of the modifier using:

```
MPI_Bcast(&(pattern[0][0]), N*N, MPI_INT, 0, MPI_COMM_WORLD);
```

Where pattern is broadcasted as a whole to all ranks.

```
MPI_Scatter(&(modifier[0][0], N * M/num_ranks, MPI_INT, local_modifier, N * M/num_ranks, MPI_INT, 0,
MPI_COMM_WORLD);
```

Where modifier is split/scattered between ranks, M/num_ranks is the number of rows per rank, and local_modifier is an array populated with the specific row of modifier relevant to each rank.
MPI_Scatterv can be used if data distribution is not uniform.
Local computations take place in each process, then aggregated using MPI_Gather/MPI_Allgather or MPI_Reduce/MPI_Allreduce depending on the function, ie. gather to aggregate data portions into one:

```
MPI_Gather(&(magicSquare[rank *  M/num_ranks][0]),  M/num_ranks * N, MPI_INT, &(magicSquare[0][0]),
M/num_ranks * N, MPI_INT, 0, MPI_COMM_WORLD);
```

Or reduce for a reduction operation across all values:

```
MPI_Reduce(&local_result, &global_result, 1, MPI_C_BOOL, MPI_LOR, 0, MPI_COMM_WORLD);
```
This is used in isPairwiseDistinct function, where any duplicates found will return true due to OR operation

Where local_result is the bool from each process, reduced to global_result.
Use MPI_Barrier(MPI_COMM_WORLD) for synchronisation where processes depend on data from each other.

For 9x9 output matrix to be distributed among 3 processes, this is a graph for data distribution where each row in a process represents a row of the magic matrix:

-Rank 0:

[0,0]	[0, 1]	[0, 2]	[0, 3]	[0, 4]	[0, 5]	[0, 6]	[0, 7]	[0, 8]
[1, 0]	[1, 1]	[1, 2]	[1, 3]	[1, 4]	[1, 5]	[1, 6]	[1, 7]	[1, 8]
[2, 0]	[2, 1]	[2, 2]	[2, 3]	[2, 4]	[2, 5]	[2, 6]	[2, 7]	[2, 8]

-Rank 1:

[3,0]	[3, 1]	[3, 2]	[3, 3]	[3, 4]	[3, 5]	[3, 6]	[3, 7]	[3, 8]
[4, 0]	[4, 1]	[4, 2]	[4, 3]	[4, 4]	[4, 5]	[4, 6]	[4, 7]	[4, 8]
[5, 0]	[5, 1]	[5, 2]	[5, 3]	[5, 4]	[5, 5]	[5, 6]	[5, 7]	[5, 8]

-Rank 2:

[6,0]	[6, 1]	[6, 2]	[6, 3]	[6, 4]	[6, 5]	[6, 6]	[6, 7]	[6, 8]
[7, 0]	[7, 1]	[7, 2]	[7, 3]	[7, 4]	[7, 5]	[7, 6]	[7, 7]	[7, 8]
[8, 0]	[8, 1]	[8, 2]	[8, 3]	[8, 4]	[8, 5]	[8, 6]	[8, 7]	[8, 8]

2.3
By running command numact1 -H and cat numact1.out, the output obtained is:

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
```

```
node 0 size: 31847 MB
node 0 free: 4080 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 32211 MB
node 1 free: 5110 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

2.4

To make an installation of magic_matrix.cpp available to all users on the PVM, a centralised shared file system could be implemented by creating a shared directory in the head node containing magic_matrix.cpp and any required dependencies. Users on the PVM nodes can access this shared directory. Installed and deployed Ansible only in the head node, an Ansible playbook can define the tasks needed to deploy magic_matrix.cpp, such as copying files to target machines, setting up permissions and authorisation, or updating dependencies and configuration files in the shared directory. An inventory file is useful in listing all nodes and their connection details.

The advantages for this approach are: scalable, flexible, idempotent, automation, reduce storage costs, performant in fast local networks, enables same file access on multiple nodes, reduce system admin overhead, centralised updates and modifications to magic_matrix.cpp or dependencies,.etc.

Its disadvantages are: require fast network connection, overhead for small-scale deployments, potential for conflicts when programs are run simultaneously leading to resource competition, potentially loose version control, security concerns where stored sensitive information is at risk.

By carefully considering security, version control and conflicts, this approach is an excellent way to make magic_matrix.cpp available to users.