Operating Systems-2: Spring 2024
Programming Assignment1: Efficient Matrix Squaring

Waghmare Aditya Abhaykumar
CS22BTECH11061

January 25, 2024

I) Coding Approach

Program uses `Chunk, Mixed and Diagonal Chunk(Extra Credit)` methods to perform Parallel Matrix Multiplication with K `threads` for squaring matrix A. The Result for each method is written back to output file `out.txt` with the time taken by each method. The low level design of program is explained below -

Main Function

`Main function` reads the input from the `inp.txt` file and stores the value in global variables. It also initializes the global variables `int **a` (stores the input array) and `int **result` (stores the result of Matrix Multiplication) to arrays of size N*N with `void prepare_2d_arrays();` function.

Once the initialization is done main function prepares output files and runs the `Chunk, Mixed and Diagonal Chunk` methods of Parallel Matrix Multiplication one after the other in `main function` itself. This methods store their result in `int **result` matrix after which `Main function` writes the result and time taken to respective output file for each method.

Common functions

Listing 1: Single Element Multiplication

```
void matrix_mult(int row, int col){
    int r = 0;
    for(int i = 0; i < n; i++) {
        r += a[row][i] * a[i][col];
    }
    result[row][col] = r;
}
```

Listing 2: Matrix Row Multiplication

```
void matrix_row_mult(int row){
    for(int col = 0; col < n; col++) {
        matrix_mult(row, col);
    }
}
```

a) Chunk Method

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void chunk(int id);` function. The argument `int id` defines what rows are assigned to `thread` number id. Thread number id gets rows `id*ceil(N/K) + 0` to `id*ceil(N/K) + ceil(N/K)-1` assigned to it.

Listing 3: Chunk based Parallel Matrix Multiplication

```
//Chunks

//record start time
auto start = chrono::high_resolution_clock::now();

// execute threads
for(int id = 0; id < k; id++){
    threads[id] = thread(chunk, id);
}

// Join threads (wait for them to finish)
for(int id = 0; id < k; id++){
    threads[id].join();
}
//record end time
auto end = chrono::high_resolution_clock::now();
//get execution time
auto time = chrono::duration_cast<chrono::microseconds>(end - start).
   count();

//writing ouput to out.txt
out_file << "Chunks:\n\n";

for(int row = 0; row < n; row++){
    for (int col = 0; col < n; col++) {
        out_file << setw(15) << result[row][col] << ' ';
        // out_file << result[row][col] << ' ';
    }
    out_file << endl;
}
out_file << "\nTime: "<< time << "\n";
```

Listing 4: Matrix Row Multiplication

```
void chunk(int id){
    for(int i = 0; i < p; i++) {
        matrix_row_mult(id * p + i);
    }
}
```

b) **Mixed Method**

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void mixed(int id);` function. The argument `int id` defines what rows are assigned to `thread` number id. Thread number id gets rows `id + 0*K, id + 1*K, ..., id + (ceil(N/K)-1)*K` assigned to it.

Listing 5: Matrix Row Multiplication

```
//Mixed

//record start time
start = chrono::high_resolution_clock::now();

// execute threads
for(int id = 0; id < k; id++){
    threads[id] = thread(mixed, id);
}

// Join threads (wait for them to finish)
for(int id = 0; id < k; id++){
    threads[id].join();
}
//record end time
end = chrono::high_resolution_clock::now();
//get execution time
time = chrono::duration_cast<chrono::microseconds>(end - start).count()
    ;

//writing ouput to out.txt
out_file << "\n\n\n\nMixed:\n\n";

for(int row = 0; row < n; row++){
    for (int col = 0; col < n; col++) {
        // out_file << result[row][col] << ' ';
        out_file << setw(15) << result[row][col] << ' ';
    }
    out_file << endl;
}
out_file << "Time: "<< time << "\n";
```

Listing 6: Matrix Row Multiplication

```
void mixed(int id){
    for(int i = 0; i < p; i++){
        matrix_row_mult(id + i * k);
    }
}
```

c) **Diagonal Chunks Method**

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void diagonal_chunk(int id);` function. The argument `int id` defines what `diagonals`(not rows) are assigned to `thread` number id. A diagonal associated with row number id consists of elements `(row=id, col=0), (row=id+1, col=1), (row=id+2, col=2), ..., (row=(id+n-1-)%n, col=n-1)`. Thread number id gets diagonals associated with rows `id*ceil(N/K) + 0` to `id*ceil(N/K) + ceil(N/K)-1` assigned to it. Function `void matrix_diagonal_mult(int diagonal_row)` is used to get matrix square result for a diagonal associated with row diagonal_row .

Listing 7: Matrix Row Multiplication

```
//Diagonal Chunks

//record start time
start = chrono::high_resolution_clock::now();
// execute threads
for(int id = 0; id < k; id++){
    threads[id] = thread(diagonal_chunk, id);
}
// Join threads (wait for them to finish)
for(int id = 0; id < k; id++){
    threads[id].join();
}
//record end time
end = chrono::high_resolution_clock::now();
//get execution time
time = chrono::duration_cast<chrono::microseconds>(end - start).count()
   ;

//writing ouput to out.txt
out_file << "\n\n\n\nDiagonal Chunks:\n\n";
for(int row = 0; row < n; row++){
    for (int col = 0; col < n; col++) {
        // out_file << result[row][col] << ' ';
        out_file << setw(15) << result[row][col] << ' ';
    }
    out_file << endl;
}
out_file << "Time: "<< time << "\n";
```

Listing 8: Matrix Row Multiplication

```
void diagonal_chunk(int id){
    for(int i = 0; i < p; i++){
        matrix_diagonal_mult(id * p + i);
    }
}
```

Listing 9: Matrix Row Multiplication

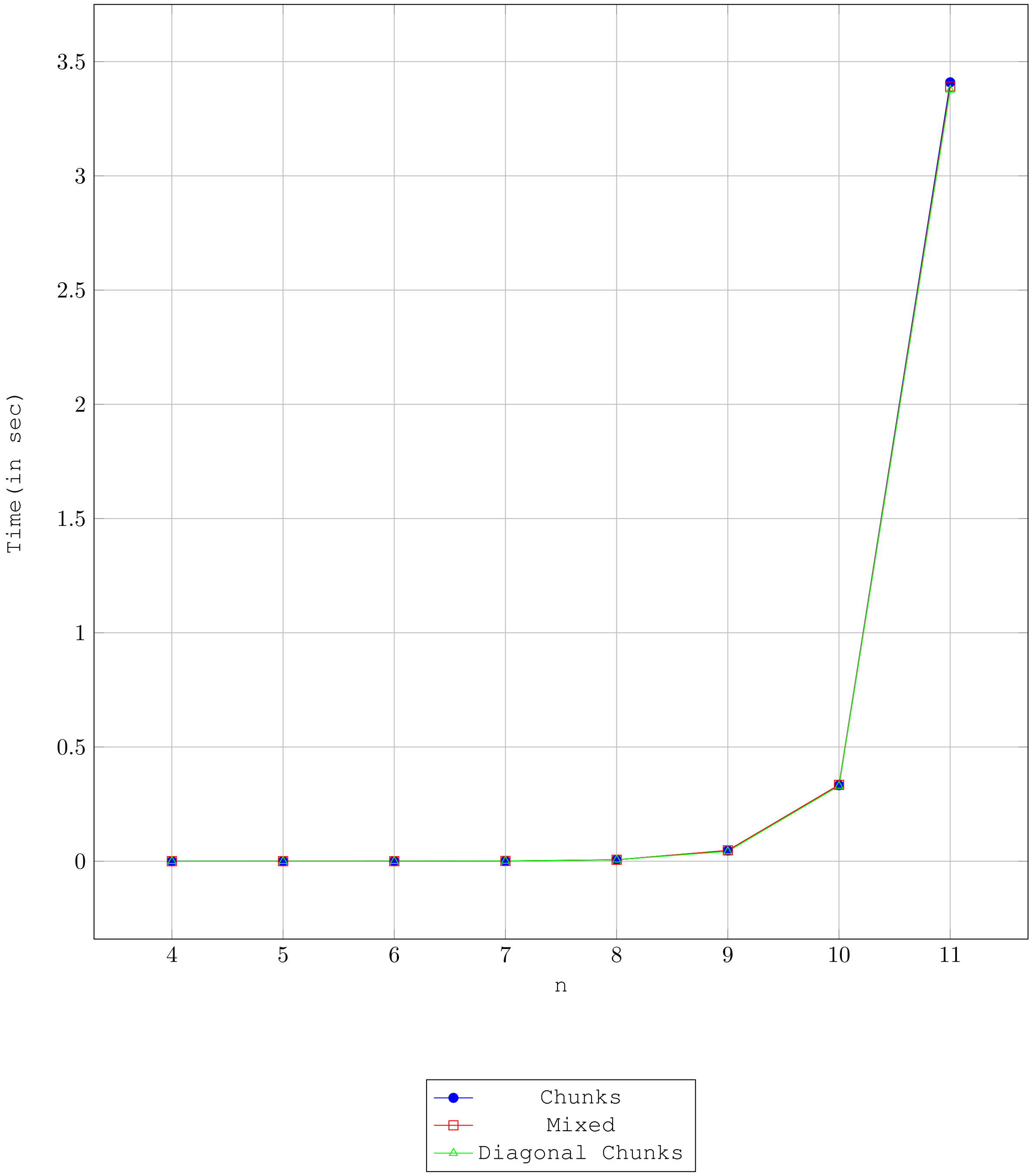
```
void matrix_diagonal_mult(int diagonal_row){
    int row = diagonal_row;;
    for(int col=0; col<n; col++){
        matrix_mult(row, col);
        row++;
        row %= n;
    }
}
```

II) Output Time Analysis

a) Time vs Size, N:

K = 8

n is such that N = 2^n

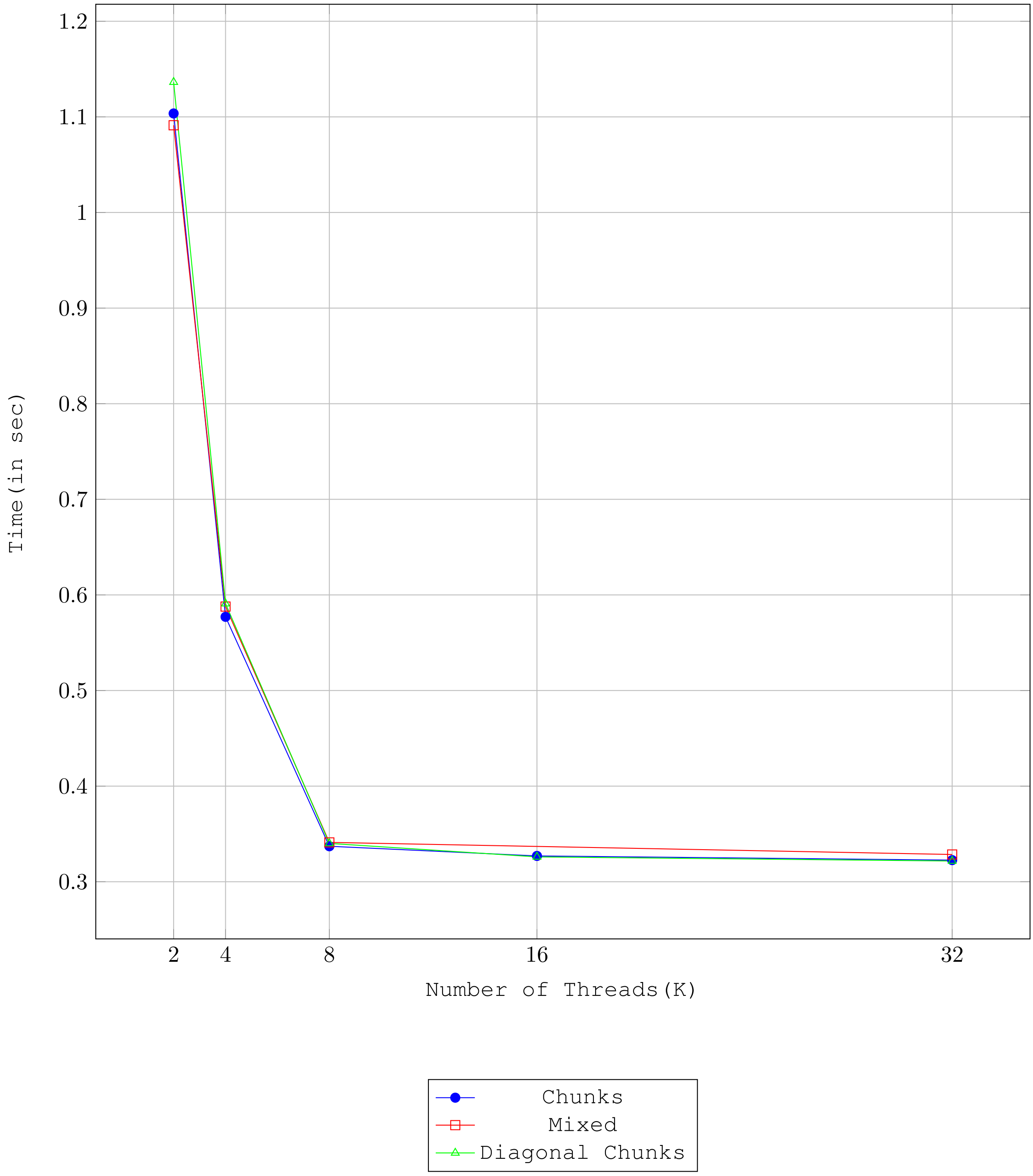


Observations:

- All the methods are taking nearly the same time for larger values of N.
- As N is growing exponentially from 16 to 2048, time taken also seems to be growing exponentially for all the methods.

b) Time vs Number of Threads, K:

N = 1024



Observations:

- All methods nearly take same time. But mixed method is slightly slower.
- As K or number of threads increases, time taken decreases.
- After K = 8, the time boost by threading saturates/stagnates due to limited number of cores.