7/30/2021

170024R – A M S D Alahakoon

170031K – M V R Amaratunga

Concurrent Programming

Take-Home Lab I

Contents

[1. Design 2](#_Toc78545233)

[1.1 Pre-processing Unit 2](#_Toc78545234)

[1.2 Concurrent Unit 3](#_Toc78545235)

[1.3 Post Processing 3](#_Toc78545236)

[2. Results 4](#_Toc78545237)

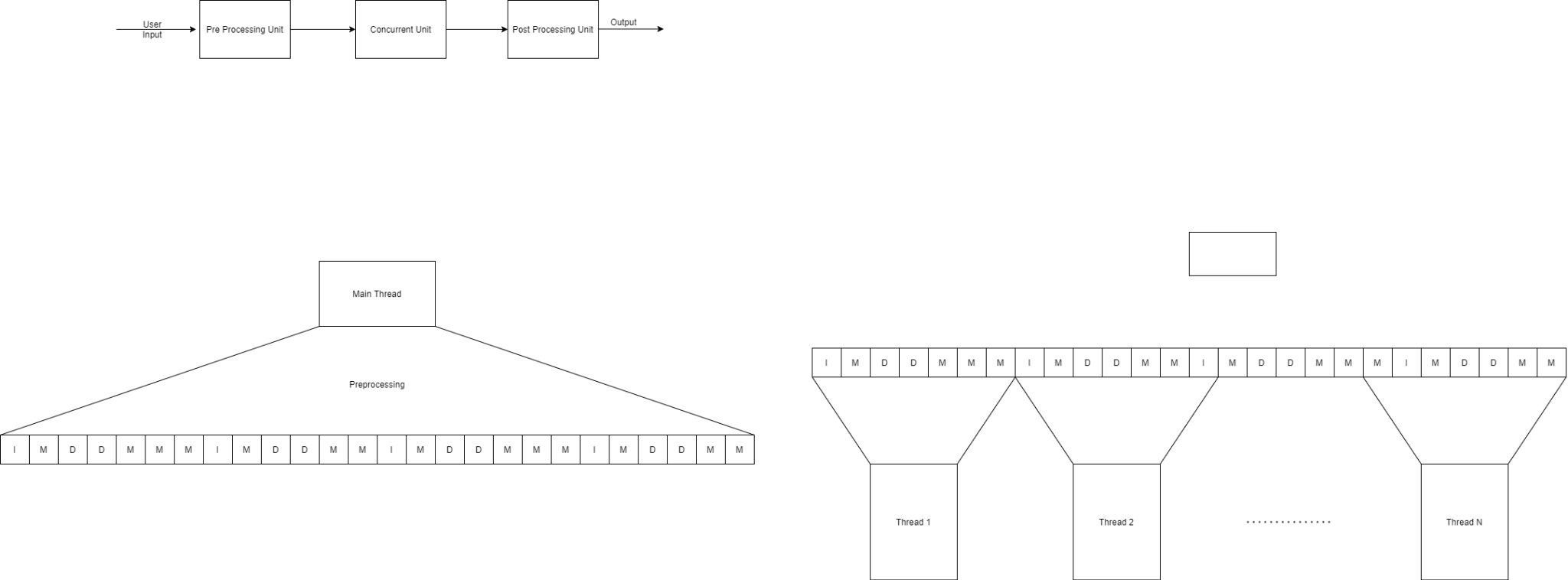
[3. Implementation 7](#_Toc78545238)

[3.1 Pre-processing Unit 7](#_Toc78545239)

[3.2 Concurrent Unit 8](#_Toc78545240)

## Design

Following diagram is a high-level design diagram for our implementation



There are three Main Sections

1. Pre-Processing
2. Concurrent
3. Post Processing

### 1.1 Pre-processing Unit

This unit is responsible for initializing the resources and variables for the concurrent execution unit. There are two main data structures in the proposed method.

* **Linked list**

This linked list is used to do the operations. There are three main operations. They are Insert, Member and Delete. Initially this list is defined as an empty node and then we insert n random numbers into the linked list.

* **Execution List**

This is an array list consisting of three unique integers zero, one and two with the length of m. Zero for member operation, one for insertion operation and two for deletion operation. First, we generate this list with a ‘m’ number of above operations and shuffle it. This list will divide equally into each thread for doing the operations in future

There are two main operations performed by this unit.

1. Initialization of the Linked-List
   1. This operation used to fill the linked list which is initially empty
2. Initialization of the Execution List
   1. This operation used to create the execution list which consists of identifiers for

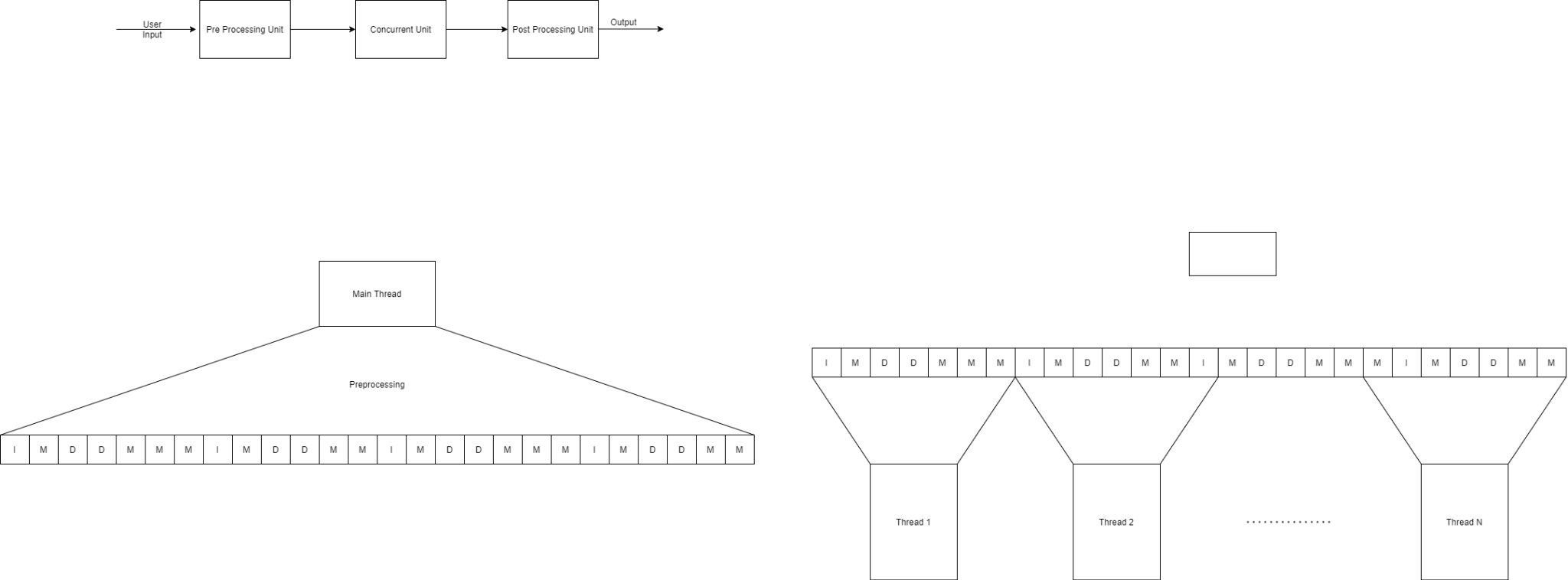
member, insert and delete operations. According to fraction for member, insert and

delete we calculate the number for insert, delete and member operations. Then we create the list and shuffle it randomly

### 1.2 Concurrent Unit

In this section the Insert, Member, Delete functions are executed concurrently in the following manner.

* The Main thread will spawn the number of threads specified by the user and each thread starts the execution method. The rank\_id is passed as the only argument.
* In the execution method the Insert, Member, Delete functions are executed to the given proportions.
* Each thread will calculate the offset that needs to be processed from the Execution list.
* Execute Member function if the list contains 0, Execute Insert function if the list contains 1, Execute Delete function if the list contains a 2.
* When executing each function, a random number between 1-2\*\*16 and passed as an argument.



### 1.3 Post Processing

In this section the threads are joined and terminate the process.

## Results

**Case 1**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.99, *mIndert* = 0.005, *mDelete* = 0.005

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implementation** |  |  | **No of threads** | |  |  |  |  |
| **1** |  | **2** | | **4** |  | **8** |  |
|  |  |  | |  |  |  |  |
| Serial |  |  |  | |  |  |  |  |
| One mutex for entire list |  |  |  |  |  |  |  |  |
| Read-Write lock |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implementation** |  |  | **No of threads** | |  |  |  |  |
| **1** |  | **2** | | **4** |  | **8** |  |
| **Average** | **Std** | **Average Std** | | **Average** | **Std** | **Average** | **Std** |
| Serial |  |  |  | |  |  |  |  |
| One mutex for entire list |  |  |  |  |  |  |  |  |
| Read-Write lock |  |  |  |  |  |  |  |  |

**Case 2**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.90, *mIndert* = 0.05, *mDelete* = 0.05

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implementation** |  |  | **No of threads** | |  |  |  |  |
| **1** |  | **2** | | **4** |  | **8** |  |
|  |  |  | |  |  |  |  |
| Serial |  |  |  | |  |  |  |  |
| One mutex for entire list |  |  |  |  |  |  |  |  |
| Read-Write lock |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implementation** |  |  | **No of threads** | |  |  |  |  |
| **1** |  | **2** | | **4** |  | **8** |  |
| **Average** | **Std** | **Average** | **Std** | **Average** | **Std** | **Average** | **Std** |
| Serial |  |  |  | |  |  |  |  |
| One mutex for entire list |  |  |  |  |  |  |  |  |
| Read-Write lock |  |  |  |  |  |  |  |  |

**Case 3**

*n* = 1,000 and *m* = 10,000, *mMember* = 0.50, *mIndert* = 0.25, *mDelete* = 0.25

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implementation** |  |  | **No of threads** | |  |  |  |  |
| **1** |  | **2** | | **4** |  | **8** |  |
|  |  |  | |  |  |  |  |
| Serial |  |  |  | |  |  |  |  |
| One mutex for entire list |  |  |  |  |  |  |  |  |
| Read-Write lock |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Implementation** |  |  | **No of threads** | |  |  |  |  |
| **1** |  | **2** | | **4** |  | **8** |  |
| **Average** | **Std** | **Average** | **Std** | **Average** | **Std** | **Average** | **Std** |
| Serial |  |  |  | |  |  |  |  |
| One mutex for entire list |  |  |  |  |  |  |  |  |
| Read-Write lock |  |  |  |  |  |  |  |  |

## Implementation

### Pre-processing Unit

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

printf("Sequential Linked List Testing\n");

number\_of\_thread = strtol(argv[1], NULL, 10);

int n = atoi(argv[3]);

m = atoi(argv[2]);

// int instructions[m];

m\_fraction = atof(argv[4]);

i\_fraction = atof(argv[5]);

d\_fraction = atof(argv[6]);

pthread\_t thread\_pool[number\_of\_thread];

int random = rand() % MAX;

root = (struct node\*) malloc(sizeof(struct node));

root->data = random;

int i = 1;

srand(time(NULL));

while(i < n-1) {

random = rand() % MAX;

if (Insert(random, &root) == 1) {

i++;

}

}

createInstructionList(instructions, &m, &m\_fraction, &i\_fraction, &d\_fraction);

. . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . .

}

### 3.2 Concurrent Unit

long thread;

for (thread = 0; thread < number\_of\_thread; thread++)

{

pthread\_create(&thread\_pool[thread], NULL, execute, (void \*) thread);

}

#### Sequential Execution

void \*execute(void \*args) {

int temp;

int total = m/number\_of\_thread;

long rank = (long) args;

// printf("Thread Rank:- %ld\n", rank);

for (size\_t i = total \* rank; i < total \* rank + total; i++)

{

temp = rand() % MAX;

if (instructions[i] == 0) {

Member(temp, root);

} else if (instructions[i] == 1) {

Insert(temp, &root);

} else {

Delete(temp, &root);

}

}

return NULL;

}

#### 3.2.2 Mutex Execution

void \*execute(void \*args) {

int temp;

int total = m/number\_of\_thread;

long rank = (long) args;

// printf("Thread Rank:- %ld\n", rank);

for (size\_t i = total \* rank; i < total \* rank + total; i++)

{

temp = rand() % MAX;

if (instructions[i] == 0) {

pthread\_mutex\_lock(&mutex);

Member(temp, root);

pthread\_mutex\_unlock(&mutex);

} else if (instructions[i] == 1) {

pthread\_mutex\_lock(&mutex);

Insert(temp, &root);

pthread\_mutex\_unlock(&mutex);

} else {

pthread\_mutex\_lock(&mutex);

Delete(temp, &root);

pthread\_mutex\_unlock(&mutex);

}

}

return NULL;

}

#### 3.2.3 Read-Write-Lock Execution

void \*execute(void \*args) {

int temp;

int total = m/number\_of\_thread;

long rank = (long) args;

// printf("Thread Rank:- %ld\n", rank);

for (size\_t i = total \* rank; i < total \* rank + total; i++)

{

temp = rand() % MAX;

if (instructions[i] == 0) {

pthread\_rwlock\_rdlock(&rwlock);

Member(temp, root);

pthread\_rwlock\_unlock(&rwlock);

} else if (instructions[i] == 1) {

pthread\_rwlock\_wrlock(&rwlock);

Insert(temp, &root);

pthread\_rwlock\_unlock(&rwlock);

} else {

pthread\_rwlock\_wrlock(&rwlock);

Delete(temp, &root);

pthread\_rwlock\_unlock(&rwlock);

}

}

return NULL;

}

#### 3.2.4 Post-Processing

for (thread = 0; thread < number\_of\_thread; thread++) {

pthread\_join(thread\_pool[thread], NULL);

}

1. **Analysis**

**Case 1**

Chart, line chart

Description automatically generated

* In the above graph, although the serial execution has values on multiple threads, we have only considered one thread. we can observe the serial execution time has a flat line which is expected.
* In the mutex execution, we can observe the time increase, even when the number of threads is increase. This behaviour is expected, because although multiple threads are running, since we lock each member, insert, and delete execution by a mutex, technically the execution of those functions calls happens sequentially. Additionally locking and unlocking overheads are added to the execution time.
* In the read-write-lock we can observe, reduction in time when increasing threads. In that case when multiple member calls can be executed parallelly, but not the delete or insert. Since huge proportion of the executions are member functions, those member functions are executed parallelly. Only delete and insert functions are executed sequentially. Therefore, we can see an improvement over the serial execution.

**Case 2**

Chart, line chart

Description automatically generated

* In this case, serial execution and Mutex are as explain above. But we can observe the read-write-lock behaves differently. This behaviour is due to the increase of proportions in delete and insert executions. Therefore, the parallel execution proportions will be reduced. Also, the locking mechanisms has an overhead, which is not present in the serial execution. Hence, we can observe such a behaviour.

**Case 3**

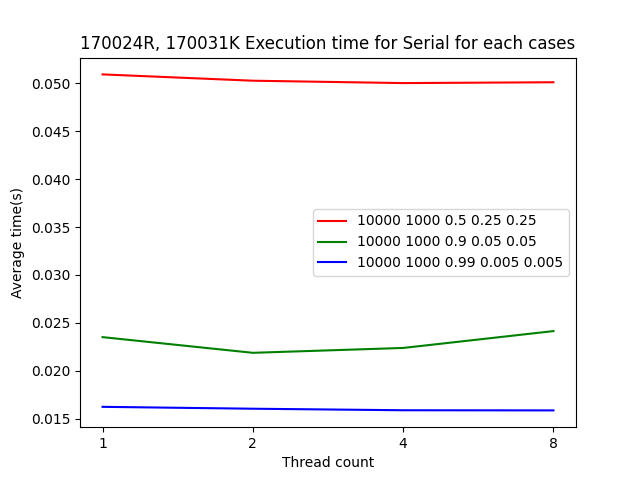
Chart, line chart

Description automatically generated

* Here 50% of the functions call are delete and insert functions. Therefore, most of the executions happens sequentially. Also, the read-write-lock have by default a higher overhead than a simple mutex locking mechanism. Hence, we can see a higher execution time for read-write-lock.

**Analysis Each method Separately**

**Sequential Case**

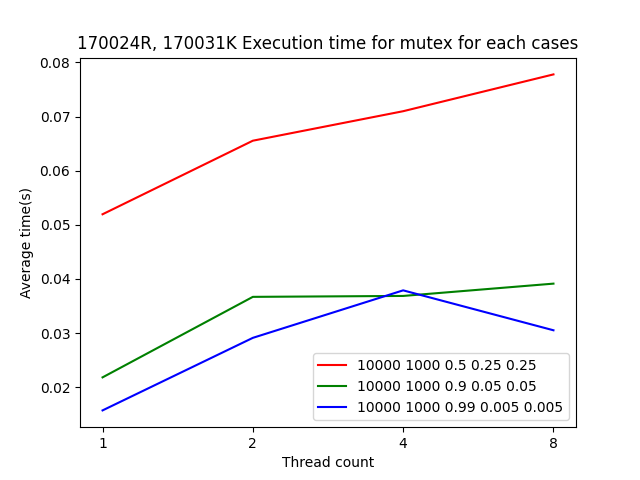
****

The red line represents configuration of Member - 50%, Delete – 25%, Insert – 25%.

The green line represents configuration of Member – 90%, Delete – 5%, Insert – 5%.

The blue line represents configuration of Member – 99%, Delete - .5%, Insert - .5%.

Here the thread count represents single thread separate execution, this is to be consistent with the other graphs. So, the flat line and the small deviation for separate runs is obvious. But in the red line more delete and insert functions are executed, which are more expensive that the Member functions. The same reasons are applicable for the green line.



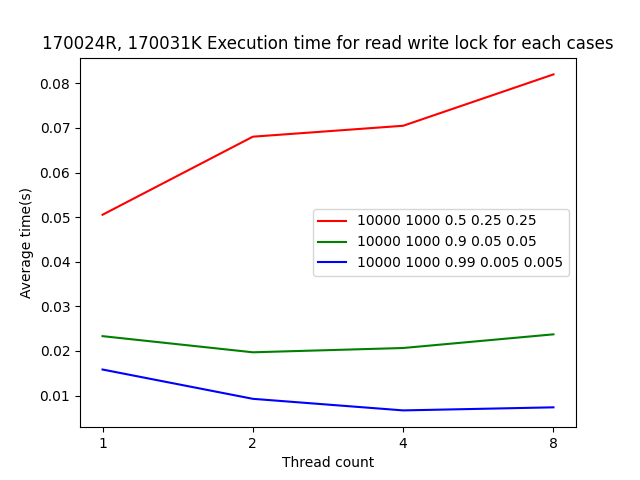
The red line represents configuration of Member - 50%, Delete – 25%, Insert – 25%.

The green line represents configuration of Member – 90%, Delete – 5%, Insert – 5%.

The blue line represents configuration of Member – 99%, Delete - .5%, Insert - .5%.

In the case of mutex, the same explanation of sequential case can be given to this scenario, when considering each configuration. In a particular execution the execution time increases since the switching between a single mutex is increasing and that will increase.

**Read Write Lock**



The red line represents configuration of Member - 50%, Delete – 25%, Insert – 25%.

The green line represents configuration of Member – 90%, Delete – 5%, Insert – 5%.

The blue line represents configuration of Member – 99%, Delete - .5%, Insert - .5%.

Here the Member functions are running parallelly, in the first two case (green and blue) reduces since member functions divided among each thread and run parallelly. But in the red line since 50% of the function’s calls are executed sequentially. The overall time is increasing.