Operating Systems-2: Spring 2024
Programming Assignment 3: Dynamic Matrix Squaring

Waghmare Aditya Abhaykumar
CS22BTECH11061

March 2, 2024

I) Coding Approach

This Program performs parallel matrix multiplication through a Dynamic mechanism in C++. The computations are done by K `threads`. Each thread is dynamically assigned `rowInc` rows to compute, assigned new rows when its previous workload is computed. We use a counter to assign rows to `threads`, which is atomically accessed and incremented by `rowInc` via TAS, CAS, Bounded CAS and atomic increment algorithms(provided by the C++ atomic library). For each of this algorithms, the total time taken and resultant sqaure matrix are written back to their respective output files, named `out_(tas/cas/bcas/atomic_increment).txt`. 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 as well as `bool *waiting` (stores waiting status of threads in Bounded CAS) to array of size K(number of threads).

Once the initialization is done, `Main function` prepares output files and runs the `TAS, CAS, Bounded CAS and Atomic increment` algorithms of Dynamic 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 total time taken and result to respective output file for each method.

For Dynamic Row Assigment to threads for matrix square computation, we use the counter `int rowsAssigned` for TAS, CAS, BCAS and counter `atomic<int> currentRow` for Atomic Increment Algorithm.

Common functions

Listing 1: Single Element Multiplication

```
//computes one element of result square matrix
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

```
//computes one row of result square matrix
void matrix_row_mult(int row){
    for(int col = 0; col < n; col++) {
        matrix_mult(row, col);
    }
}
```

a) `TAS Algorithm`

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void dynamicChunks_TAS(int id);` function. Rows are dynamically assigned by counter variable `int rowsAssigned`. This counter is synchronized through a TAS Algorithm with lock `atomic_flag lock_tas`, where only one thread can enter critical section at a time. TAS algorithm is implemented with C++ Atomic class `test_and_set()` function.

Listing 3: TAS Algorithm Thread Routine

```
//TAS

atomic_flag lock_tas = ATOMIC_FLAG_INIT;//TAS lock, intial value is
   unlocked

void dynamicChunks_TAS(int id){//id is thread id

    while(true){//while not all rows are computed for square matrix

        //aquire TAS lock
        while (lock_tas.test_and_set(memory_order_acquire)) {
            // Spin until lock is acquired
        }

        /*CRITICAL SECTION*/

        //getting rows to work on
        int startRow = rowsAssigned;
        rowsAssigned += rowInc;

        //release lock or unlock
        lock_tas.clear(memory_order_release);

        /*REMAINDER SECTION*/

        //if assigned row is more than matrix rows, then break
        if(startRow >= n) break;

        //compute assigned rows
        for(int i = 0; i < rowInc && i < n; i++) {
            matrix_row_mult(startRow + i);
        }
    }
}
```

Listing 4: Dynamic Parallel Matrix Multiplication with TAS

```
    //TAS

    //prepare output file for TAS
    ofstream out_file("out_tas.txt");
    // Check if the file is open
    if(!out_file.is_open()){
        cerr << "Error opening out_chunk.txt file." << endl;
        return 1;
    }

    reset_result();//sets values of result matrix to 0


    //initially rows assigned to threads are zero
    rowsAssigned = 0;

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

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

    //write the output
    out_file << "Time: "<< time << "\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 << endl;
    }

    out_file.close();
```

b) **CAS Algorithm**

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void dynamicChunks_CAS(int id);` function. Rows are dynamically assigned by counter variable `int rowsAssigned`. This counter is synchronized through a CAS Algorithm with lock `atomic<bool> lock_cas`, where only one thread can enter critical section at a time. CAS algorithm is implemented with C++ Atomic class `compare_exchange_strong(expected, desired)` function. This function works same as `compare_and_exchange(expected, desired)` for boolean values, except the fact that it updates the variable named `expected`. This can be handled in the while loop by reassigning `expected` variable for each iteration.

Listing 5: CAS Algorithm Thread Routine

```
//CAS

atomic<bool> lock_cas(false);//CAS lock, initial value is unlocked

void dynamicChunks_CAS(int id){//id is thread id

    while(true){//while not all rows are computed for square matrix

        //aquire CAS lock
        bool aquired = true;
        bool available = false;

        while (!lock_cas.compare_exchange_strong(available, aquired)){
            available = false;//to keep available variable value false
            // Spin until lock is acquired
        }

        /*CRITICAL SECTION*/

        //getting rows to work on
        int startRow = rowsAssigned;
        rowsAssigned += rowInc;

        //release lock
        lock_cas.store(available);

        /*REMAINDER SECTION*/

        //if assigned row is more than matrix rows, then break
        if(startRow >= n) break;

        //compute assigned rows
        for(int i = 0; i < rowInc && i < n; i++) {
            matrix_row_mult(startRow + i);
        }
    }
}
```

Listing 6: Dynamic Parallel Matrix Multiplication with CAS

```
//CAS

//prepare output file for CAS
out_file.open("out_cas.txt");
// Check if the file is open
if(!out_file.is_open()){
    cerr << "Error opening out_mixed.txt file." << endl;
    return 1;
}
reset_result();//sets values of result matrix to 0


//initially rows assigned to threads are zero
rowsAssigned = 0;

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

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

    //write the output
    out_file << "Time: "<< time << "\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 << endl;
    }

    out_file.close();
```

c) Bounded CAS Algorithm

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void dynamicChunks_BCAS(int i);` function. Rows are dynamically assigned by counter variable `int rowsAssigned`. This counter is synchronized through a Bounded CAS Algorithm with lock `atomic_flag lock_bcas`, where only one thread can enter critical section at a time. Here, i is the thread id, which is used in BCAS algorithm. BCAS is implemented with C++ Atomic class `test_and_set()` function as basis for the algorithm.

Listing 7: Bounded CAS Algorithm Thread Routine

```
    //Bounded CAS

    bool *waiting;//waiting list of threads for critical section
    atomic_flag lock_bcas = ATOMIC_FLAG_INIT;//BCAS lock, intial value is unlocked

    void dynamicChunks_BCAS(int i){//i is thread id

        while(true){//while not all rows are computed for square matrix

            //start waiting to aquire lock for critical section
            waiting[i] = true;
            bool key = true;

            //aquire lock
            while (waiting[i] && key){//either aquire lock ourselves or some other process assigns it to us
                key = lock_bcas.test_and_set(memory_order_acquire);
            }
            //stop waiting once lock aquired
```

```
            waiting[i] = false;


            /*CRITICAL SECTION*/

            //getting rows to work on
            int startRow = rowsAssigned;
            rowsAssigned += rowInc;

            //find which other process is waiting for lock
            int j = (i + 1) % k;
            while ((j != i) && !waiting[j]){
                j = (j + 1) % k;
            }
            //if no process waiting for lock release it
            if (j == i){
                //release lock
                lock_bcas.clear(memory_order_release);
            }
            //if some process waiting for lock, pass the lock to it
            else{
                waiting[j] = false;
            }

            /*REMAINDER SECTION*/

            //if assigned row is more than matrix rows, then break
            if(startRow >= n) break;

            //compute assigned rows
            for(int i = 0; i < rowInc && i < n; i++) {
                matrix_row_mult(startRow + i);
            }
        }
    }
```

Listing 8: Dynamic Parallel Matrix Multiplication with Bounded CAS

```
    //Bounded CAS

    //prepare output file for BCAS
    out_file.open("out_bcas.txt");
    // Check if the file is open
    if(!out_file.is_open()){
        cerr << "Error opening out_mixed.txt file." << endl;
        return 1;
    }
    reset_result();//sets values of result matrix to 0


    //initially rows assigned to threads are zero
    rowsAssigned = 0;

    //for bcas
    waiting = new bool[k];
    for(int id=0; id<k; id++) waiting[id] = false;

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

```

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

    //write the output
    out_file << "Time: "<< time << "\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 << endl;
    }

    out_file.close();
```

d) Atomic Increment Algorithm

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void dynamicChunks_atomic_increment(int id);` function. Rows are dynamically assigned by counter variable `atomic<int> currentRow`. This counter is synchronized through atomic classes atomic increment functionality `currentRow.fetch_add(rowInc)`, where only one thread can increment at a time. Atomic Increment is implemented with C++ Atomic class `fetch_add(rowInc)` function.

Listing 9: Atomic Increment Algorithm Thread Routine

```
    //Atomic Increment

    atomic<int> currentRow(0);//number of rows already assigned to threads or counter C for Atomic increment
    void dynamicChunks_atomic_increment(int id){//id is thread id

        while(true){//while not all rows are computed for square matrix

            //atomically getting rows to work on
            int startRow = currentRow.fetch_add(rowInc);

            /*REMAINDER SECTION*/

            //if assigned row is more than matrix rows, then break
            if(startRow >= n) break;

            //compute assigned rows
            for(int i = 0; i < rowInc && i < n; i++) {
                matrix_row_mult(startRow + i);
            }
```

```
        }
    }
```

Listing 10: Dynamic Parallel Matrix Multiplication with Atomic Increment

```
    //atomic increment

    //prepare output file for Atomic increment
    out_file.open("out_atomic_increment.txt");
    // Check if the file is open
    if(!out_file.is_open()){
        cerr << "Error opening out_mixed.txt file." << endl;
        return 1;
    }
    reset_result();//sets values of result matrix to 0


    //initially rows assigned to threads are zero
    currentRow = 0;

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

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

    //write the output
    out_file << "Time: "<< time << "\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 << endl;
    }

    out_file.close();
```

II) Experiments

Test System Specifications:

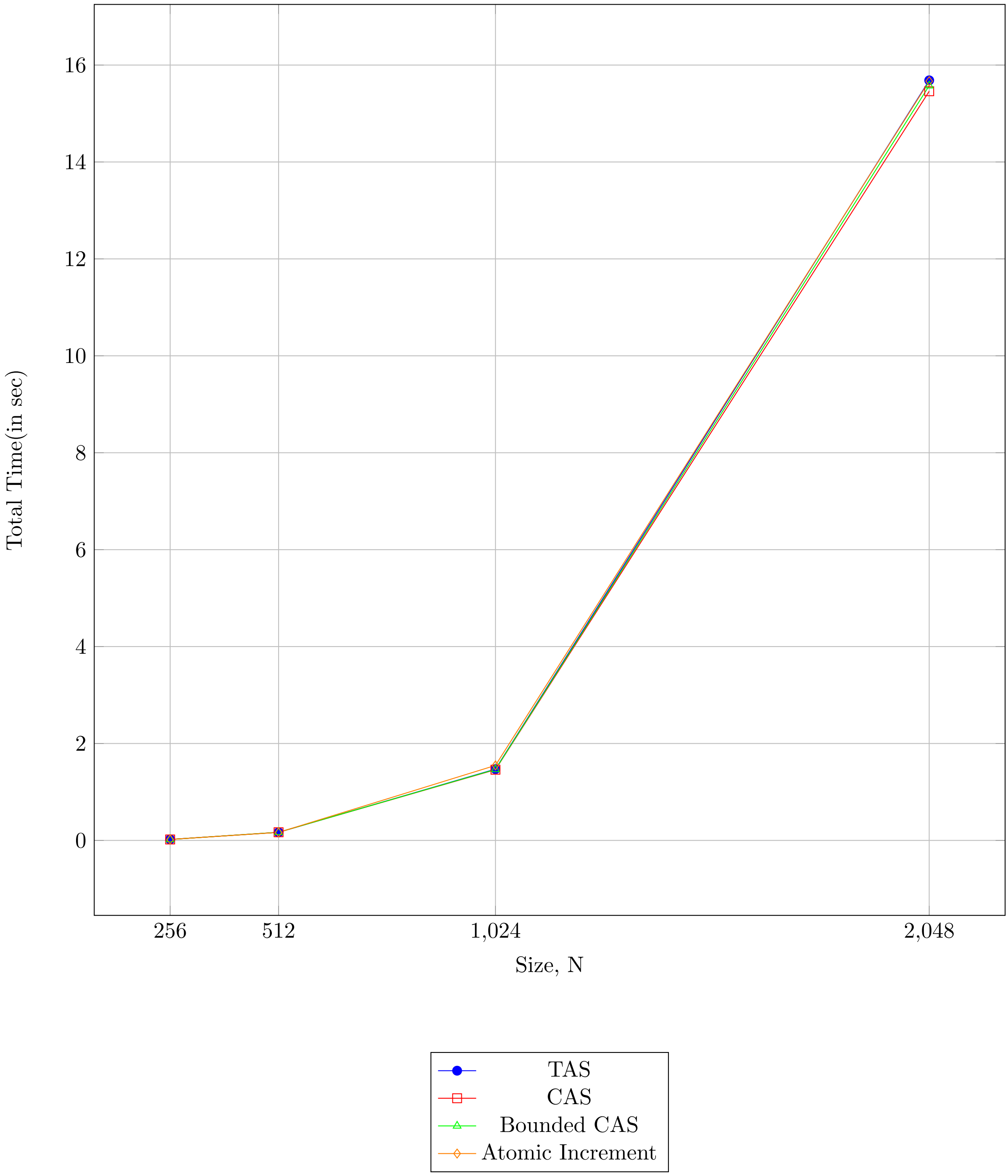
```
aditya@aditya-IdeaPad-3-15ITL6:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
    CPU family:          6
    Model:               140
    Thread(s) per core:  2
    Core(s) per socket:  2
    Socket(s):           1
    Stepping:            1
    CPU max MHz:         4100.0000
    CPU min MHz:         400.0000
    BogoMIPS:            5990.40
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bt
                         s rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
                          tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l2 invpcid_single cdp_l2 ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid e
                         pt_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xg
                         etbv1 xsaves split_lock_detect dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req vnmi avx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx5
                         12_bitalg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx512_vp2intersect md_clear ibt flush_l1d arch_capabilities
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   96 KiB (2 instances)
  L1i:                   64 KiB (2 instances)
  L2:                    2.5 MiB (2 instances)
  L3:                    6 MiB (1 instance)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0-3
Vulnerabilities:
  Gather data sampling:  Mitigation; Microcode
  Itlb multihit:         Not affected
  L1tf:                  Not affected
  Mds:                   Not affected
  Meltdown:              Not affected
  Mmio stale data:       Not affected
  Retbleed:              Not affected
  Spec rstack overflow:  Not affected
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; Enhanced / Automatic IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS SW sequence
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

Figure 1: Test System Specifications

1. Time vs. Size, N:

(Note: From N = 4096, time taken increases to 10s of minutes for each point. To take average over 5 values it will take hours for just N = 4096. Hence, we conduct experiments only till N = 2048.)

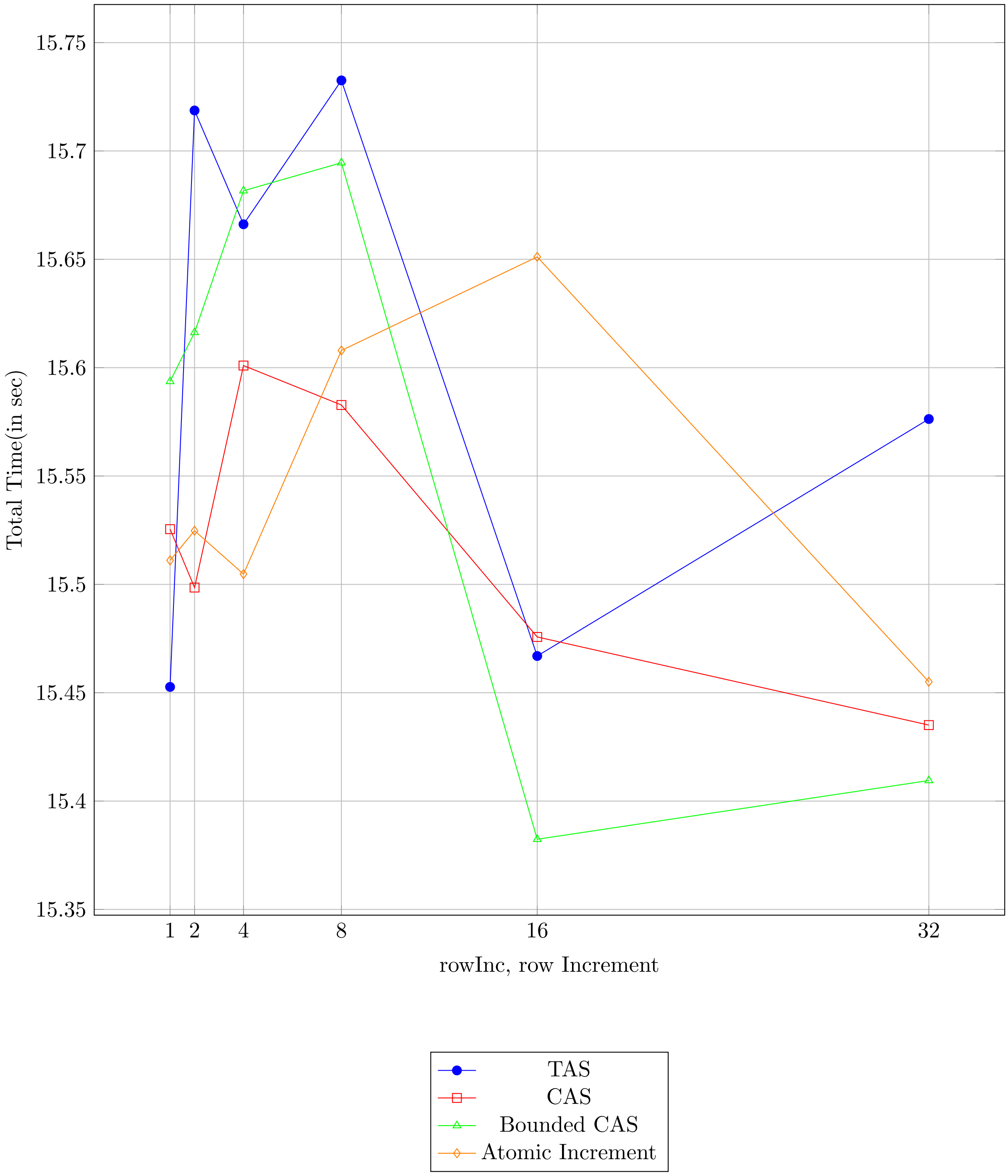
```
rowInc = 16
K = 16
```



Sr	Observation
1	As N increases, the needed computations increases. Hence, time taken increases.
2	TAS, CAS, Bounded CAS and Atomic Increment all take nearly the same time.

2. Time vs. rowInc, row Increment:

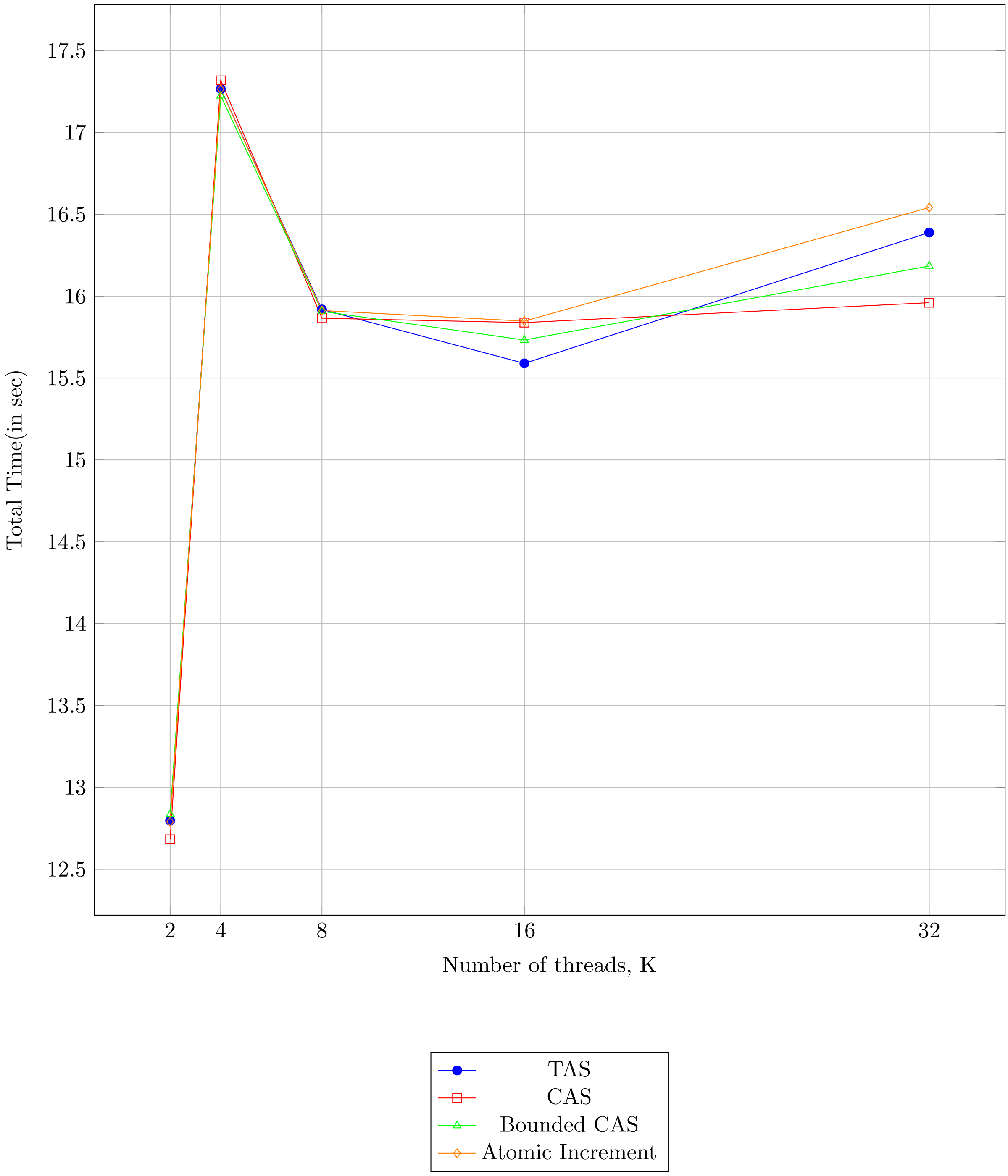
```
N = 2048
K = 16
```



Sr	Observation
1	`rowInc` does not affect Time Taken significantly.
2	TAS, CAS, Bounded CAS and Atomic Increment all take nearly the same time.
3	From rowInc = 1 to rowInc = 8, time taken slightly increases for all algorithms. After rowInc = 8, it starts falling.

3. Time vs. Number of threads, K:

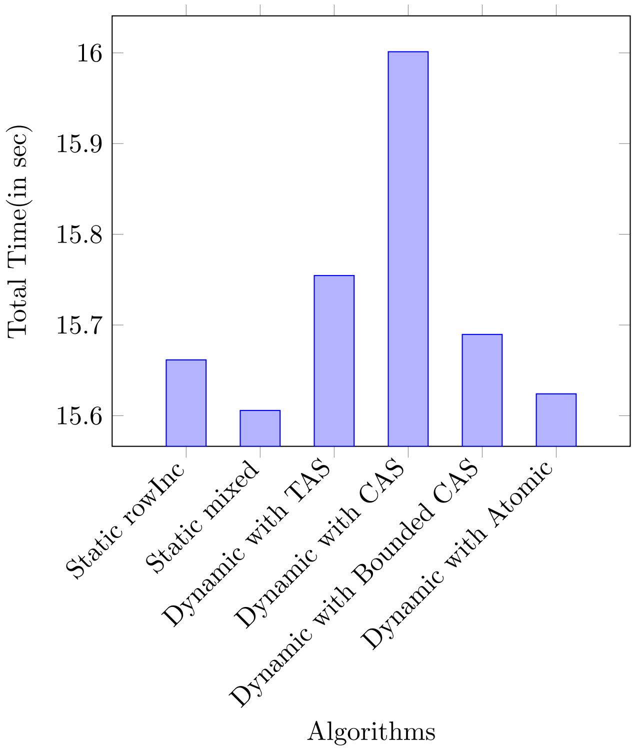
```
N = 2048
rowInc = 16
```



Sr	Observation
1	TAS, CAS, Bounded CAS and Atomic Increment all take nearly the same time.
2	All methods have their least execution time at K = 2 with a sudden spike at K = 4 and fall from K = 4 to 16. We can see a slight increase in time from K = 16 to 32.
5	After K = 8, time taken does not change significantly, for all the methods.

4. Time vs. Algorithms:

```
N = 2048
K = 16
rowInc = 16
```



Sr	Observation
1	All methods take nearly the same time
2	Dynamic with TAS, CAS, BCAS does not improve performance when compared to Static rowInc and Static Mixed. Rather they take slightly more time.
3	Dynamic with CAS takes the most time.
4	Dynamic with Atomic slighty reduces execution time and improves performance when compared to Static rowInc. It also takes less time then TAS, CAS and Bounded CAS.