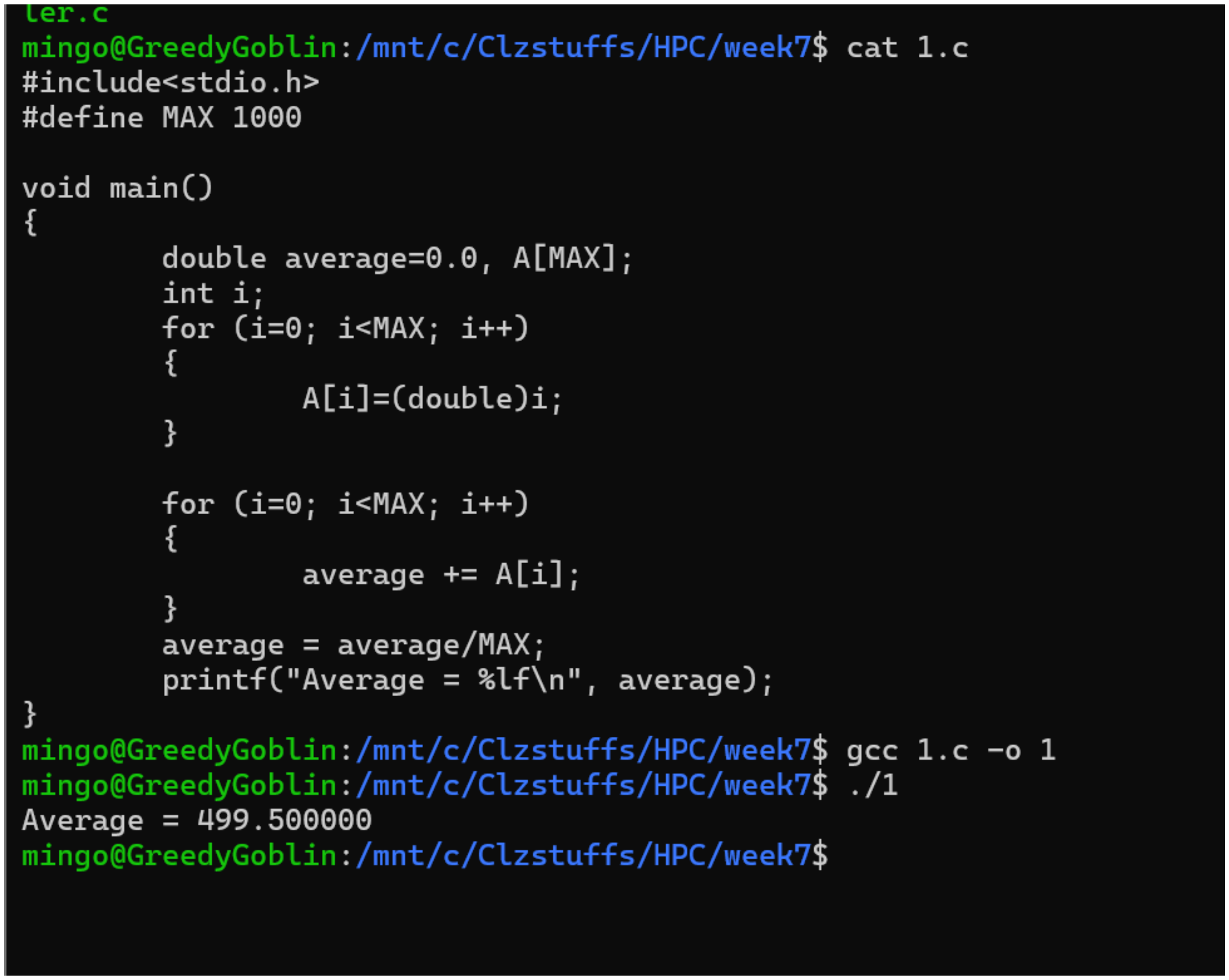
Week 7 zip

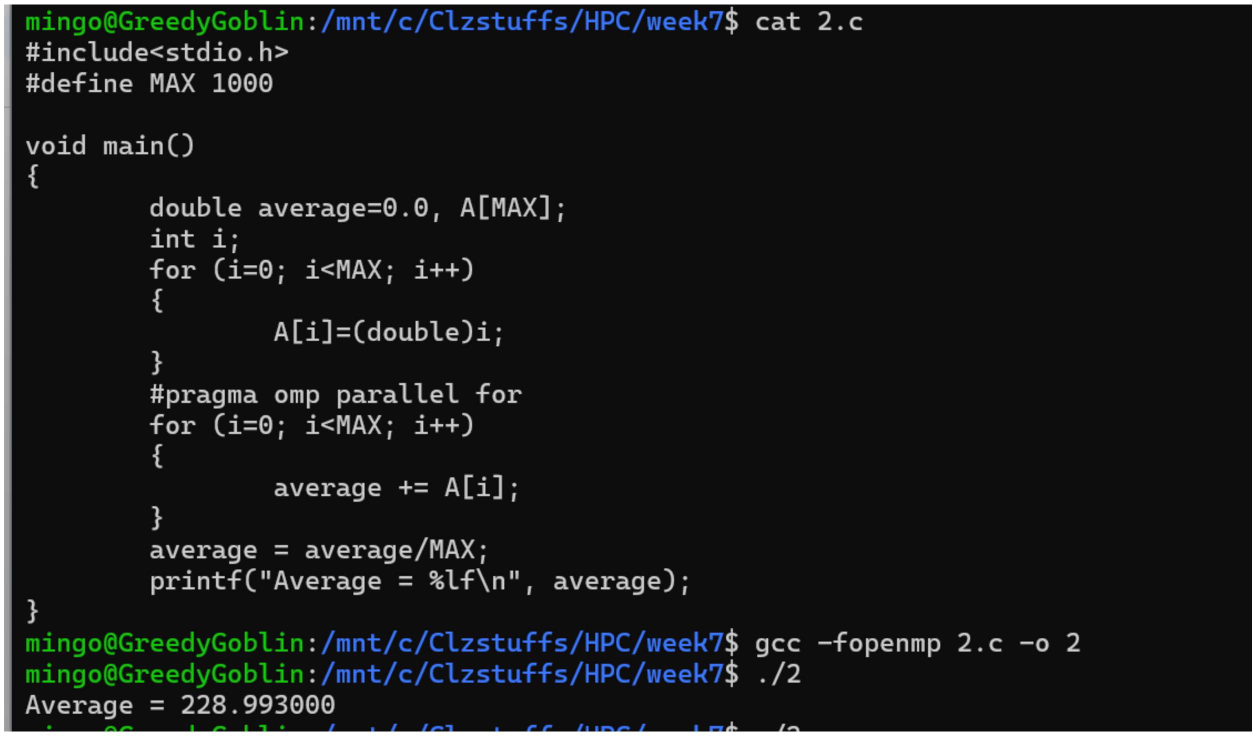
1.



```
ler.c
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 1.c
#include<stdio.h>
#define MAX 1000

void main()
{
        double average=0.0, A[MAX];
        int i;
        for (i=0; i<MAX; i++)
        {
                A[i]=(double)i;
        }

        for (i=0; i<MAX; i++)
        {
                average += A[i];
        }
        average = average/MAX;
        printf("Average = %lf\n", average);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc 1.c -o 1
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./1
Average = 499.500000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

This program fills an array with the numbers 0 to 999 and then adds them together using a single loop. Because the program runs on only one thread, every update happens in order with no interference. After computing the total, it divides by 1000 to obtain the average. The final result of 499.5 confirms that the program works correctly in a purely sequential environment.

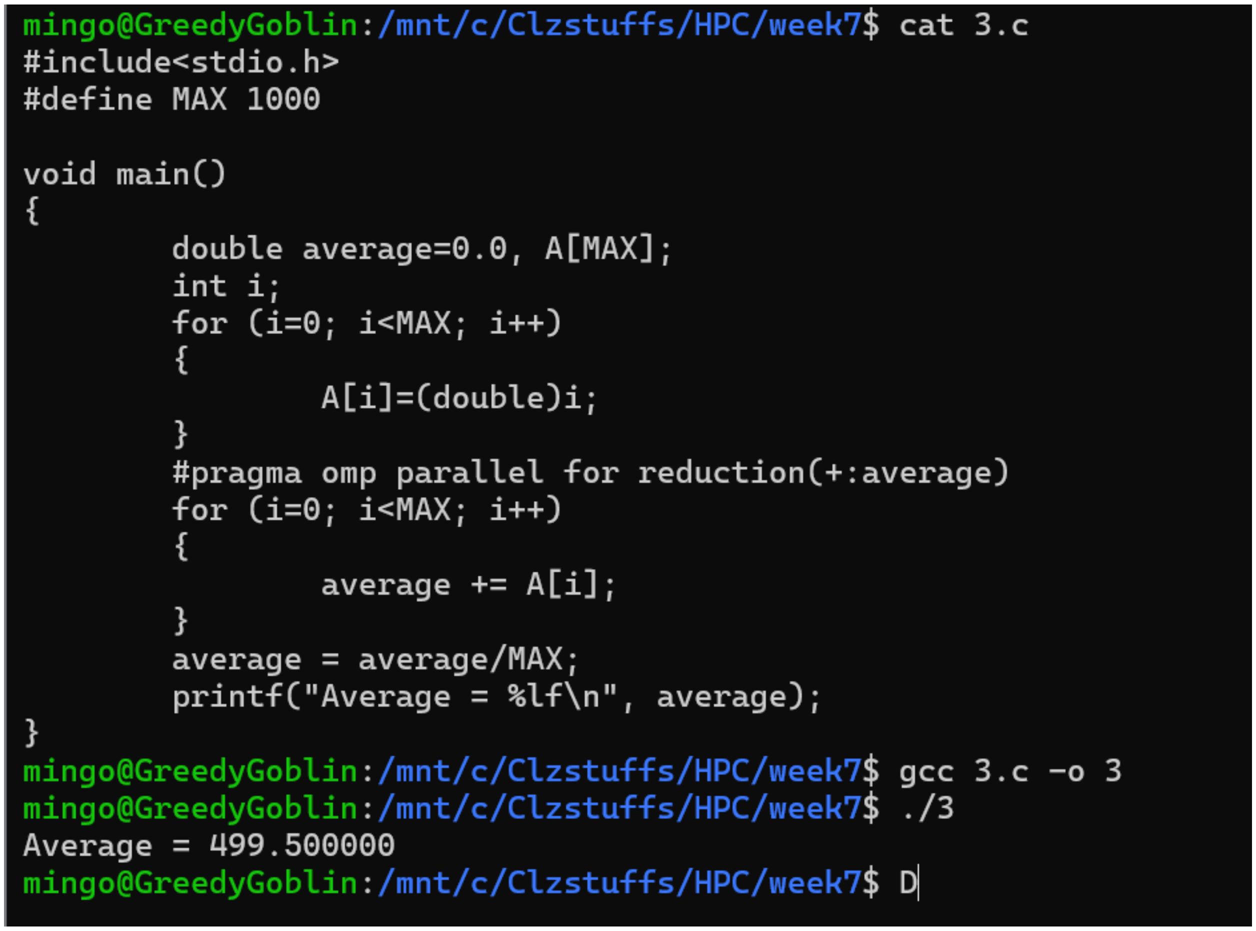
2.



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 2.c
#include<stdio.h>
#define MAX 1000

void main()
{
        double average=0.0, A[MAX];
        int i;
        for (i=0; i<MAX; i++)
        {
                A[i]=(double)i;
        }
        #pragma omp parallel for
        for (i=0; i<MAX; i++)
        {
                average += A[i];
        }
        average = average/MAX;
        printf("Average = %lf\n", average);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp 2.c -o 2
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./2
Average = 228.993000
```

The code and output show that adding parallelism incorrectly leads to the wrong output Average = 228.993000. This error happens because of a Race Condition. The variable average is shared by all threads. Since there is no protection, multiple threads try to read, add and write back to average at the exact same time, causing data to be lost

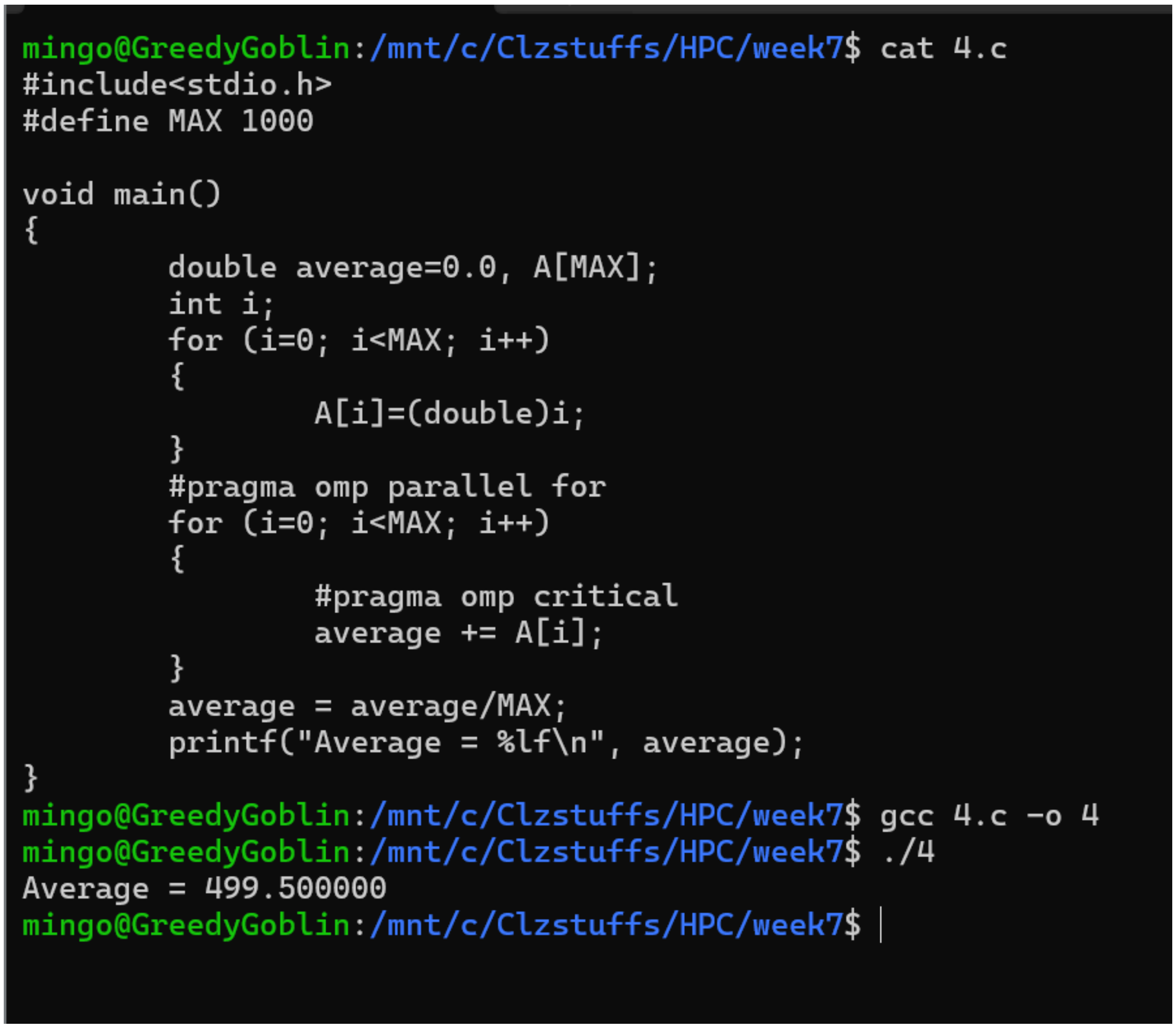
3.



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 3.c
#include<stdio.h>
#define MAX 1000

void main()
{
        double average=0.0, A[MAX];
        int i;
        for (i=0; i<MAX; i++)
        {
                A[i]=(double)i;
        }
        #pragma omp parallel for reduction(+:average)
        for (i=0; i<MAX; i++)
        {
                average += A[i];
        }
        average = average/MAX;
        printf("Average = %lf\n", average);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc 3.c -o 3
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./3
Average = 499.500000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ D
```

The code and output show the correct way to parallelize the sum calculation using the reduction clause. The line #pragma omp parallel for reduction(+:average) tells the compiler to give each thread its own private copy of the average variable. Each thread adds up a portion of the array into its private variable, avoiding the race condition. At the end, OpenMP safely adds all the private sums together into the main average variable and provides the correct output 499.500000.

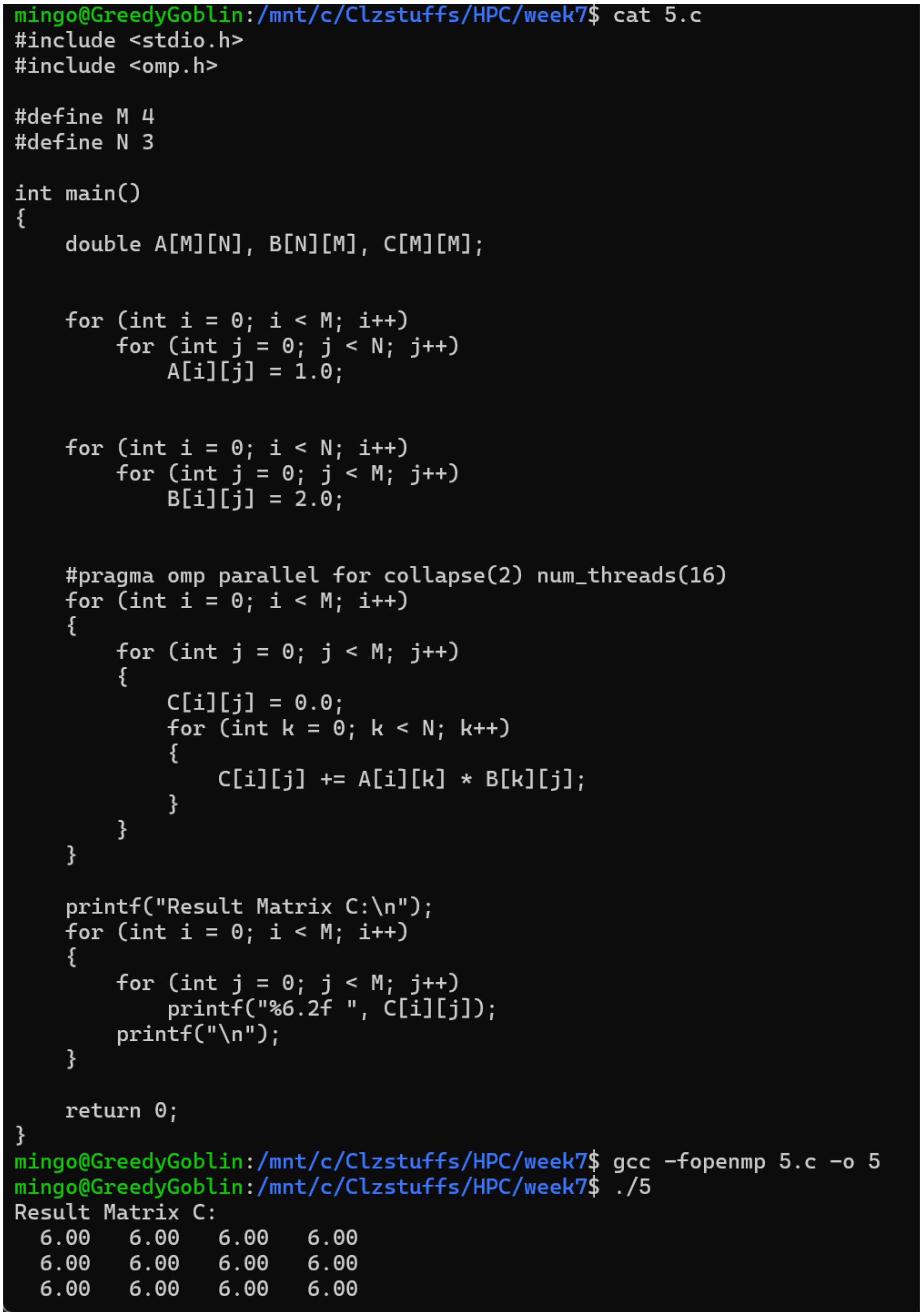
.4.



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 4.c
#include<stdio.h>
#define MAX 1000

void main()
{
        double average=0.0, A[MAX];
        int i;
        for (i=0; i<MAX; i++)
        {
                A[i]=(double)i;
        }
        #pragma omp parallel for
        for (i=0; i<MAX; i++)
        {
                #pragma omp critical
                average += A[i];
        }
        average = average/MAX;
        printf("Average = %lf\n", average);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc 4.c -o 4
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./4
Average = 499.500000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The code and output show another correct way to fix the race condition, using the #pragma omp critical directive. This command tells the compiler that the following line of code average += A[i] is a "Critical Section" and must be executed by only one thread at a time. It provides correct output and also forces the threads to wait in line to update the variable, which turns the parallel program back into a serial one for that specific operation, making it slower than the reduction method used in the previous task.

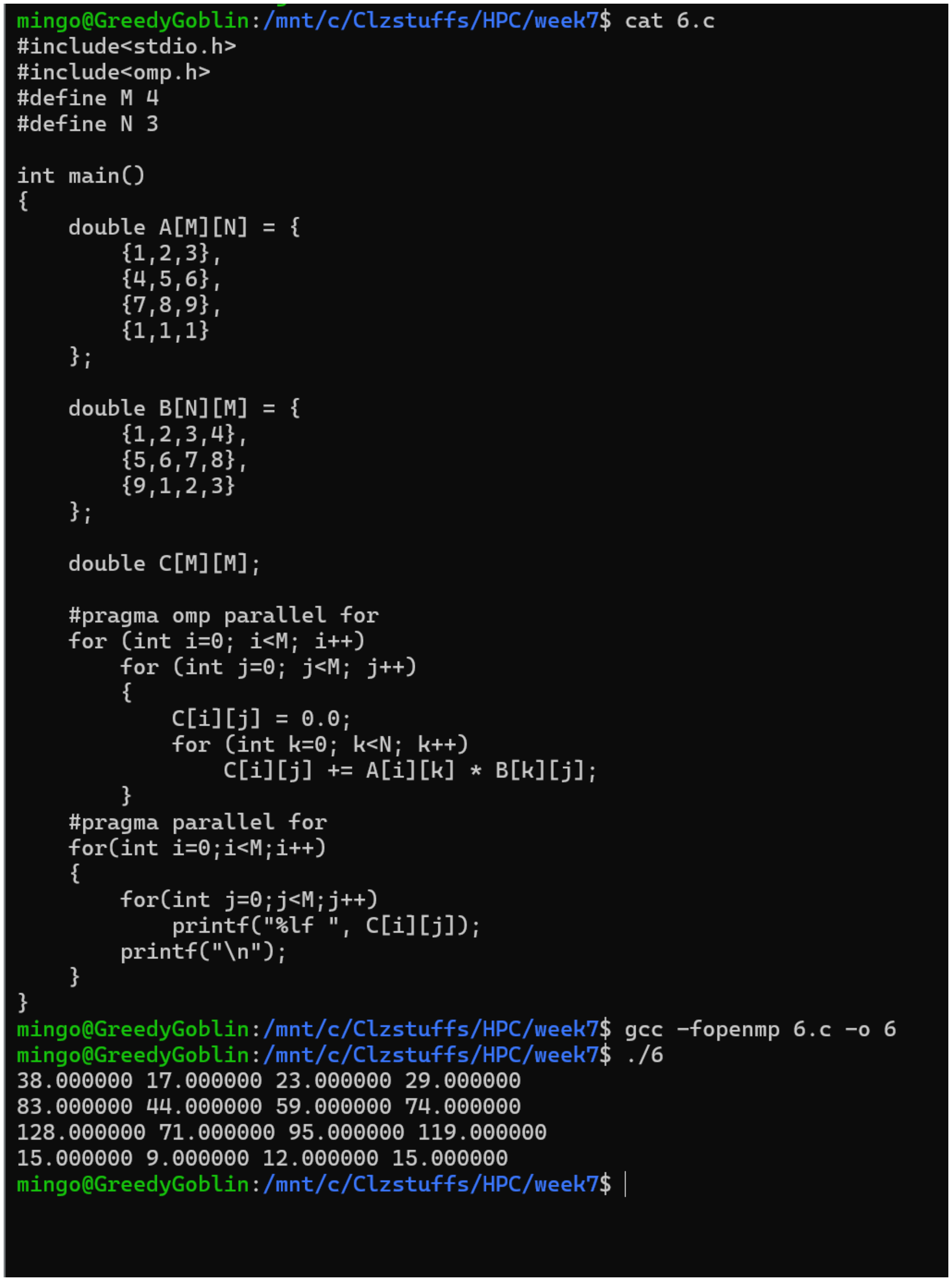
5.



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 5.c
#include <stdio.h>
#include <omp.h>

#define M 4
#define N 3

int main()
{
    double A[M][N], B[N][M], C[M][M];


    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = 1.0;


    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            B[i][j] = 2.0;


    #pragma omp parallel for collapse(2) num_threads(16)
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < M; j++)
        {
            C[i][j] = 0.0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    printf("Result Matrix C:\n");
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < M; j++)
            printf("%6.2f ", C[i][j]);
        printf("\n");
    }

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp 5.c -o 5
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./5
Result Matrix C:
  6.00   6.00   6.00   6.00
  6.00   6.00   6.00   6.00
  6.00   6.00   6.00   6.00
```

The code and output show how to perform matrix multiplication in parallel using OpenMP. The directive #pragma omp parallel for is placed before the outermost loop, which allows the computer to split the calculation of the matrix rows among different threads.The output shows that the threads performed the complex calculation correctly (3 multiplications of 1.0 * 2.0) and concurrently.

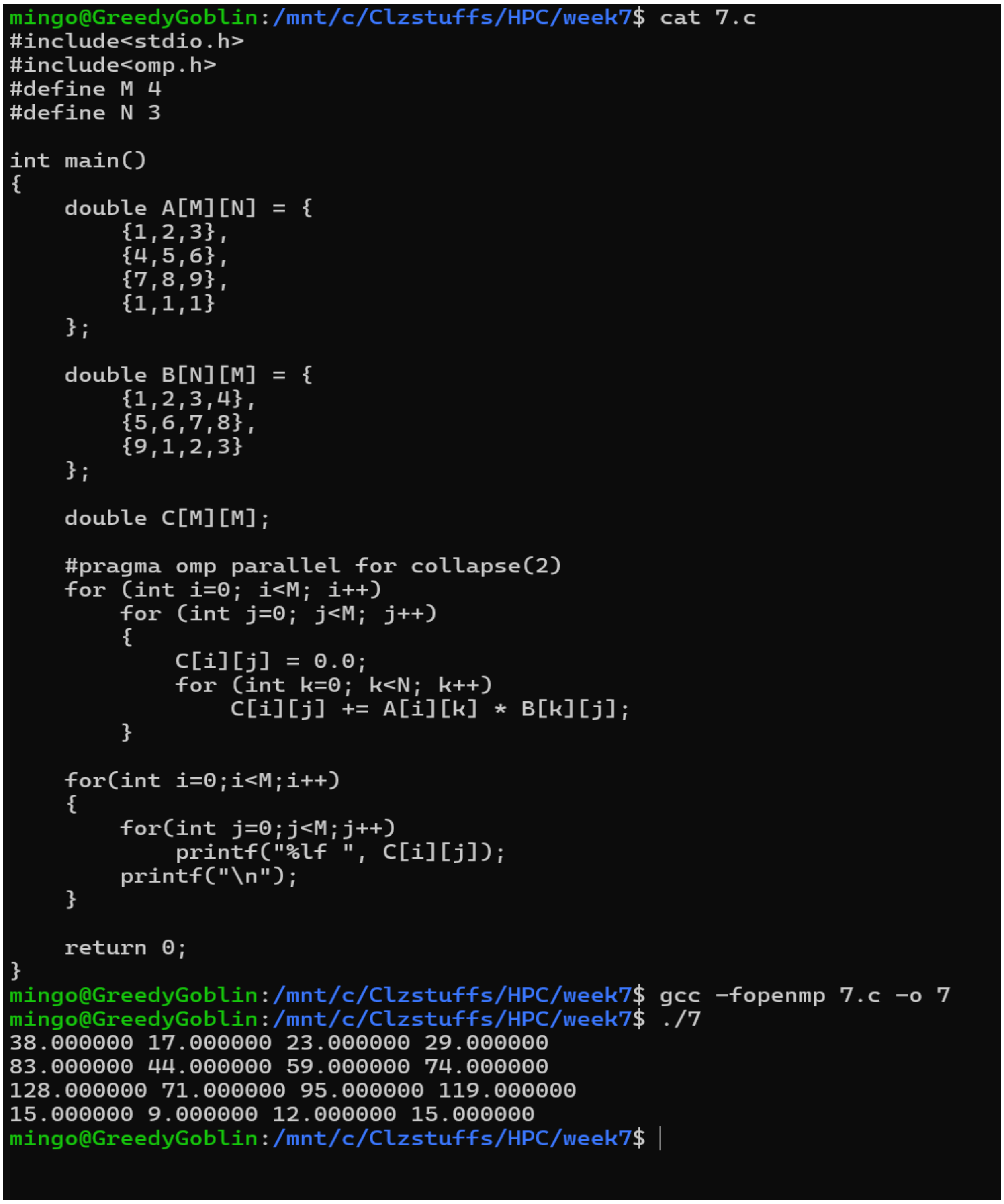
6.



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 6.c
#include<stdio.h>
#include<omp.h>
#define M 4
#define N 3

int main()
{
    double A[M][N] = {
        {1,2,3},
        {4,5,6},
        {7,8,9},
        {1,1,1}
    };

    double B[N][M] = {
        {1,2,3,4},
        {5,6,7,8},
        {9,1,2,3}
    };

    double C[M][M];

    #pragma omp parallel for
    for (int i=0; i<M; i++)
        for (int j=0; j<M; j++)
        {
            C[i][j] = 0.0;
            for (int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    #pragma parallel for
    for(int i=0;i<M;i++)
    {
        for(int j=0;j<M;j++)
            printf("%lf ", C[i][j]);
        printf("\n");
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp 6.c -o 6
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./6
38.000000 17.000000 23.000000 29.000000
83.000000 44.000000 59.000000 74.000000
128.000000 71.000000 95.000000 119.000000
15.000000 9.000000 12.000000 15.000000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The code and output show parallel matrix multiplication. The program initializes two matrices, A and B with values and uses #pragma omp parallel for to distribute the work of calculating the rows of the result matrix C. The directive splits the outermost loop iterations among threads, allowing different rows of C to be executed simultaneously. The output displays the correct dot products of the rows of A and columns of B.

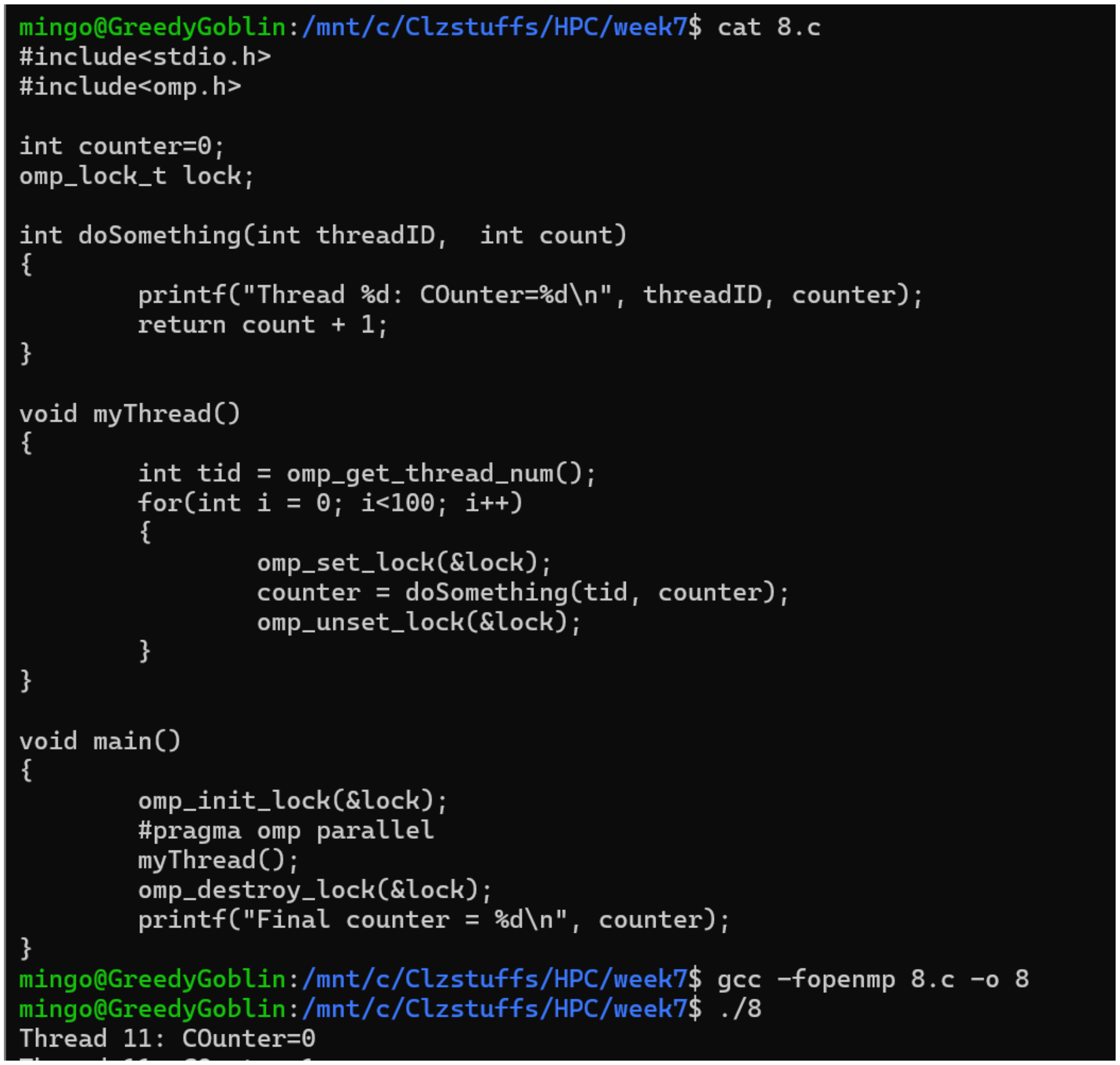
7.



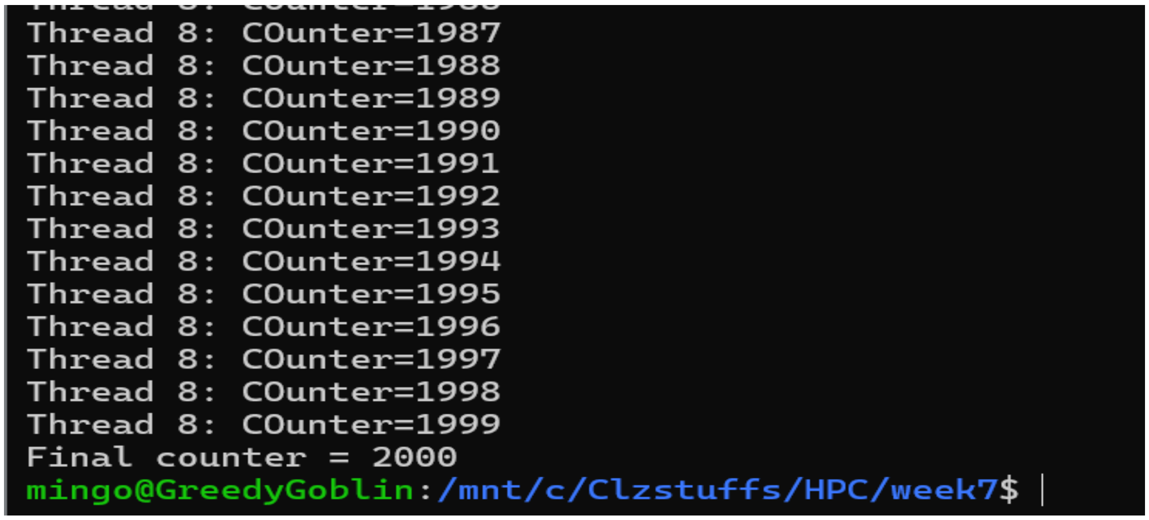
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 7.c
#include<stdio.h>
#include<omp.h>
#define M 4
#define N 3

int main()
{
    double A[M][N] = {
        {1,2,3},
        {4,5,6},
        {7,8,9},
        {1,1,1}
    };

    double B[N][M] = {
        {1,2,3,4},
        {5,6,7,8},
        {9,1,2,3}
    };

    double C[M][M];

    #pragma omp parallel for collapse(2)
    for (int i=0; i<M; i++)
        for (int j=0; j<M; j++)
        {
            C[i][j] = 0.0;
            for (int k=0; k<N; k++)
                C[i][j] += A[i][k] * B[k][j];
        }

    for(int i=0;i<M;i++)
    {
        for(int j=0;j<M;j++)
            printf("%lf ", C[i][j]);
        printf("\n");
    }

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp 7.c -o 7
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./7
38.000000 17.000000 23.000000 29.000000
83.000000 44.000000 59.000000 74.000000
128.000000 71.000000 95.000000 119.000000
15.000000 9.000000 12.000000 15.000000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ |
```

The code and output show the process of implementing the collapse directive in OpenMP. Initially, the code failed to compile because collapse requires loops to be "perfectly nested," meaning no code can exist between the for statements, but the matrix initialization was blocking this structure. I corrected the program by moving the initialization to a separate loop and using collapse(2) to safely distribute the work. This ensures that each thread calculates a specific cell among threads. The computation matches the previous matrix results, but the parallel execution becomes more efficient.

8.



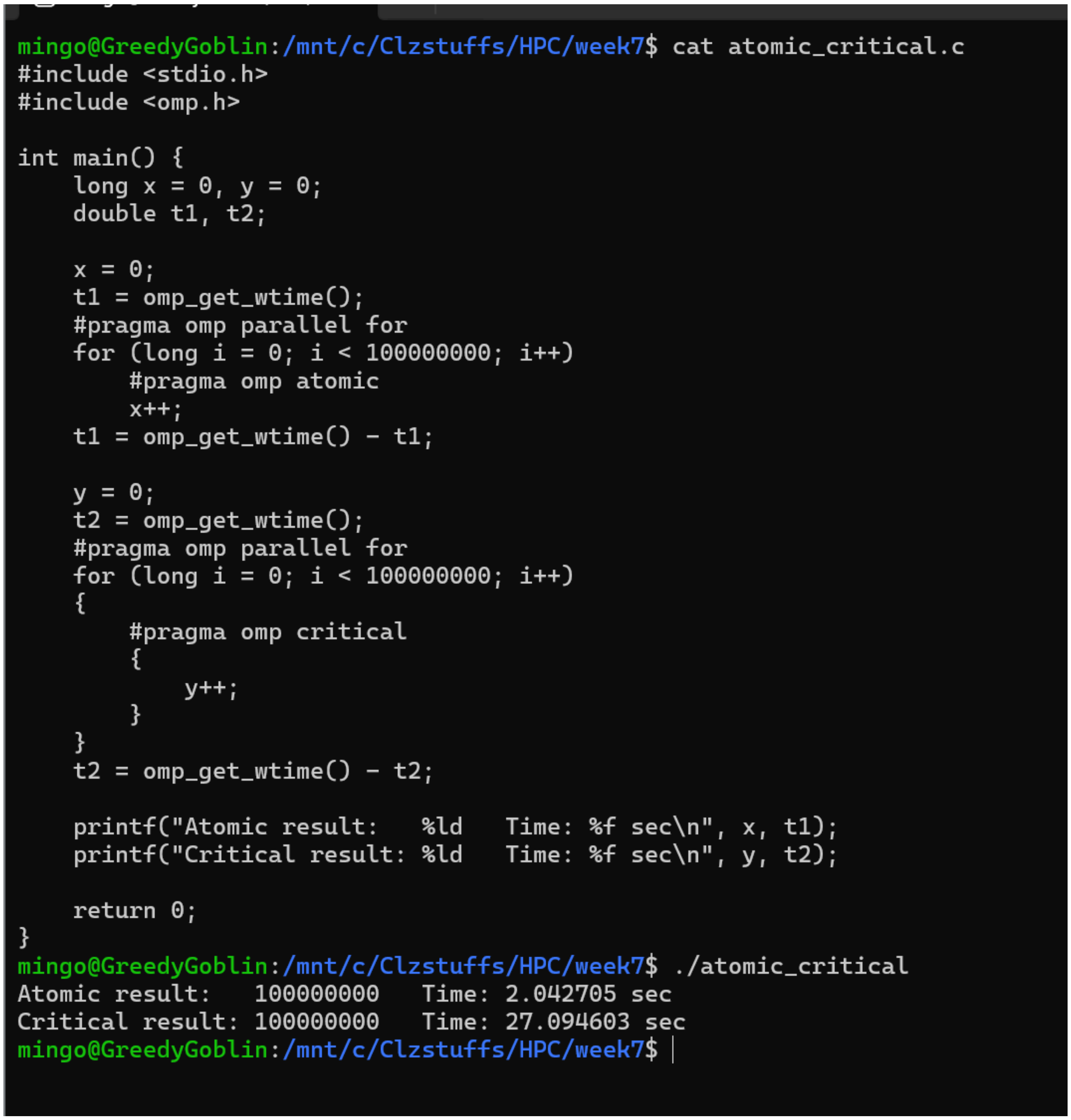
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat 8.c
#include<stdio.h>
#include<omp.h>

int counter=0;
omp_lock_t lock;

int doSomething(int threadID,  int count)
{
        printf("Thread %d: COunter=%d\n", threadID, counter);
        return count + 1;
}

void myThread()
{
        int tid = omp_get_thread_num();
        for(int i = 0; i<100; i++)
        {
                omp_set_lock(&lock);
                counter = doSomething(tid, counter);
                omp_unset_lock(&lock);
        }
}

void main()
{
        omp_init_lock(&lock);
        #pragma omp parallel
        myThread();
        omp_destroy_lock(&lock);
        printf("Final counter = %d\n", counter);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp 8.c -o 8
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./8
Thread 11: COunter=0
```



```
Thread 8: COunter=1987
Thread 8: COunter=1988
Thread 8: COunter=1989
Thread 8: COunter=1990
Thread 8: COunter=1991
Thread 8: COunter=1992
Thread 8: COunter=1993
Thread 8: COunter=1994
Thread 8: COunter=1995
Thread 8: COunter=1996
Thread 8: COunter=1997
Thread 8: COunter=1998
Thread 8: COunter=1999
Final counter = 2000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The code and output show the use of OpenMP Locks to protect a shared variable. The program creates a team of threads (20) and each thread runs the loop 100 times. This means the counter should increment 20 x 100 = 2000 times. By using omp_set_lock before updating the counter and omp_unset_lock immediately after, the code ensures that no two threads write to the counter simultaneously. The output Final counter = 1600 confirms that every single update was received correctly without any race conditions.

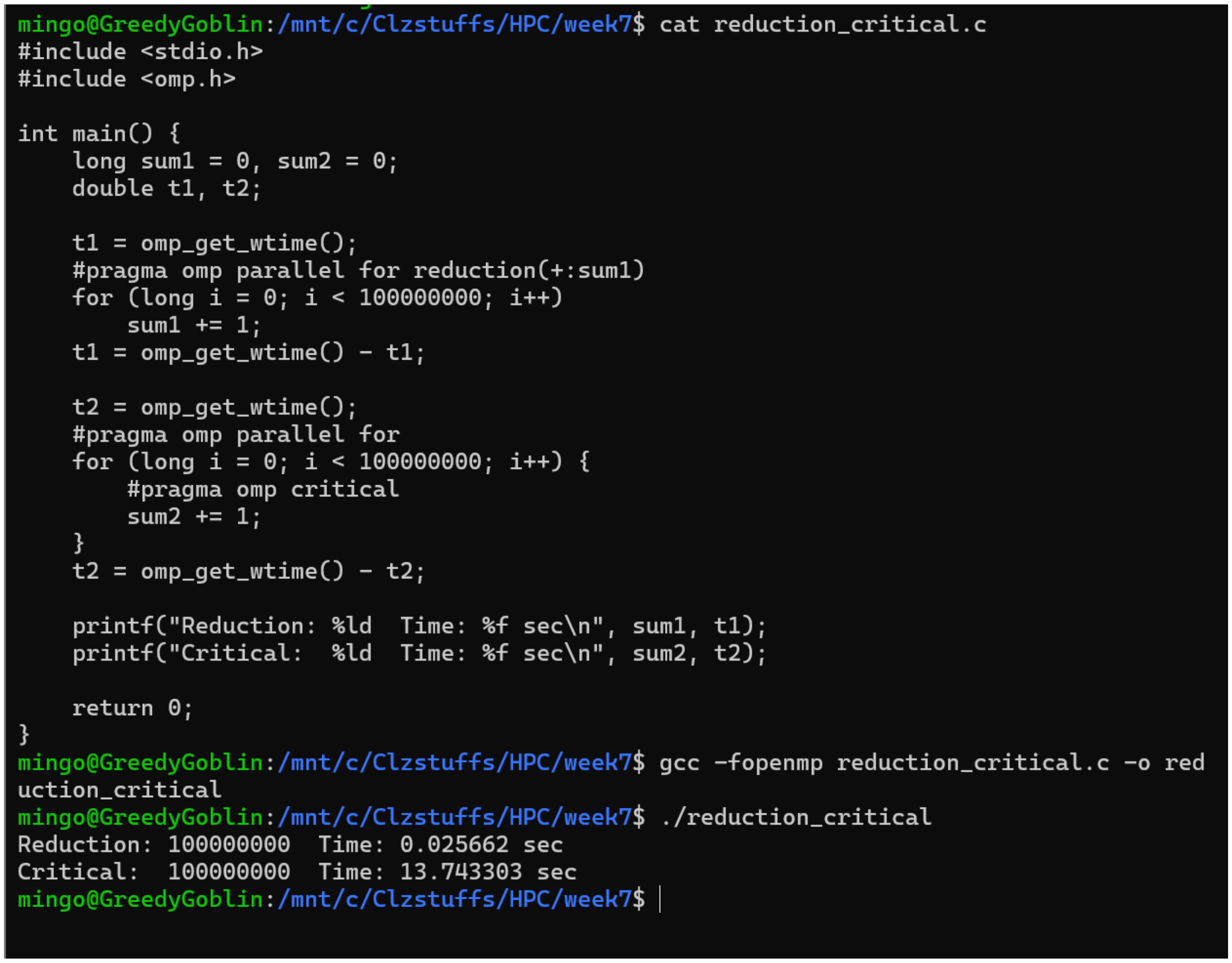
9.atomic critical



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat atomic_critical.c
#include <stdio.h>
#include <omp.h>

int main() {
    long x = 0, y = 0;
    double t1, t2;

    x = 0;
    t1 = omp_get_wtime();
    #pragma omp parallel for
    for (long i = 0; i < 100000000; i++)
        #pragma omp atomic
        x++;
    t1 = omp_get_wtime() - t1;

    y = 0;
    t2 = omp_get_wtime();
    #pragma omp parallel for
    for (long i = 0; i < 100000000; i++)
    {
        #pragma omp critical
        {
            y++;
        }
    }
    t2 = omp_get_wtime() - t2;

    printf("Atomic result:   %ld   Time: %f sec\n", x, t1);
    printf("Critical result: %ld   Time: %f sec\n", y, t2);

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./atomic_critical
Atomic result:   100000000   Time: 2.042705 sec
Critical result: 100000000   Time: 27.094603 sec
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The code and output show the large performance gap between using atomic and critical for updating a shared variable inside a parallel loop. Both versions complete the correct number of increments, but the atomic version finishes in about **2.04 seconds**, while the critical section takes around **27.09 seconds**. This happens because an atomic update protects only the specific memory operation (x++), allowing threads to continue quickly, while a critical section forces threads to enter one at a time, creating long waiting times. The results clearly show that for simple increments, atomic is much faster and more efficient than using a critical section.

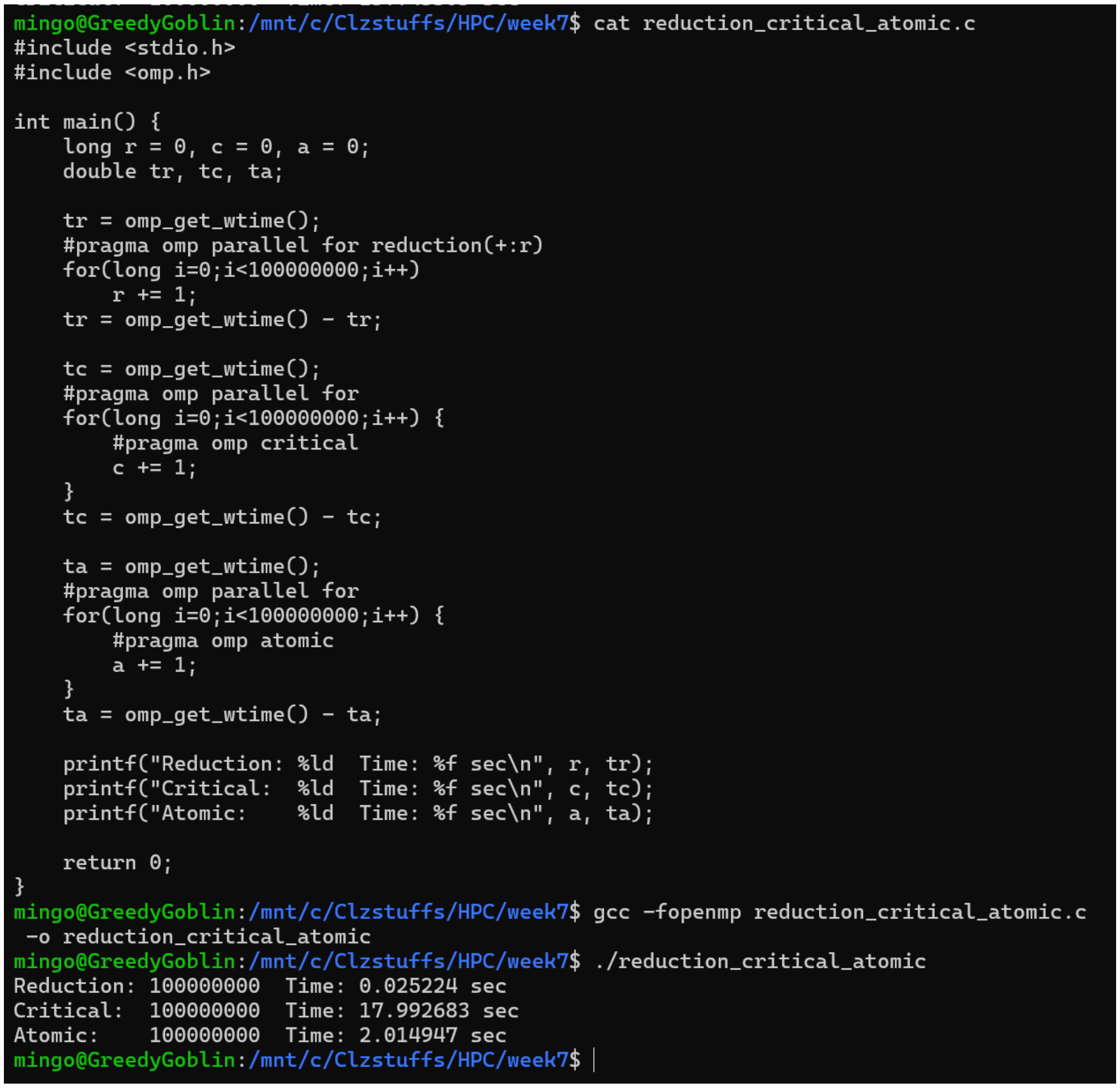
10.reduction_critical



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat reduction_critical.c
#include <stdio.h>
#include <omp.h>

int main() {
    long sum1 = 0, sum2 = 0;
    double t1, t2;

    t1 = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum1)
    for (long i = 0; i < 100000000; i++)
        sum1 += 1;
    t1 = omp_get_wtime() - t1;

    t2 = omp_get_wtime();
    #pragma omp parallel for
    for (long i = 0; i < 100000000; i++) {
        #pragma omp critical
        sum2 += 1;
    }
    t2 = omp_get_wtime() - t2;

    printf("Reduction: %ld  Time: %f sec\n", sum1, t1);
    printf("Critical:  %ld  Time: %f sec\n", sum2, t2);

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp reduction_critical.c -o red
uction_critical
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./reduction_critical
Reduction: 100000000  Time: 0.025662 sec
Critical:  100000000  Time: 13.743303 sec
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The reduction version is extremely fast, taking only about **0.025 seconds**, because each thread keeps its own private copy of the value and OpenMP combines all the partial sums only once at the end. The critical section is much slower, taking around **13.74 seconds**, because every update requires a thread to lock the shared variable, modify it, and unlock it again. This forces all other threads to wait, which greatly increases the execution time. The results clearly show that reduction is the better choice for parallel summation, while critical sections introduce heavy overhead.

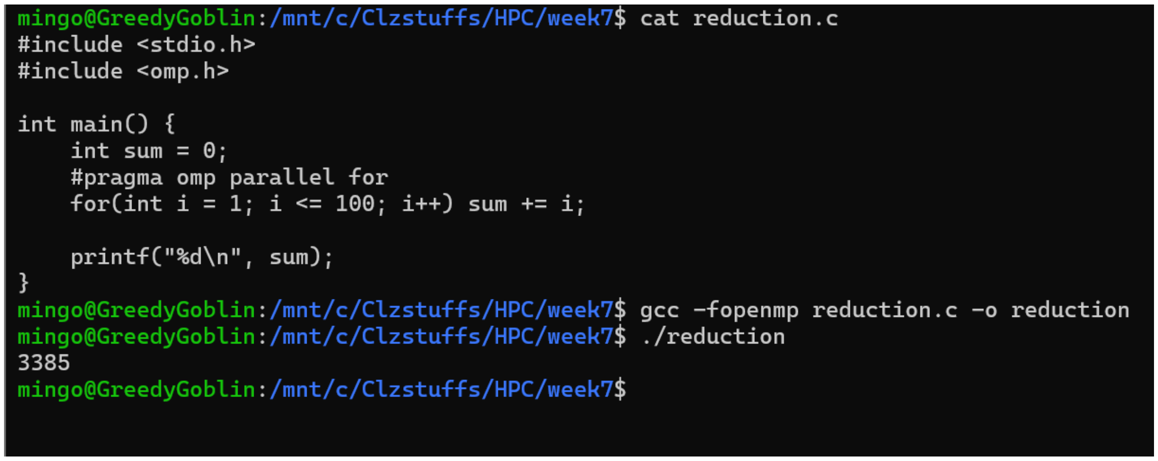
11. Reduction_critical_atomic



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat reduction_critical_atomic.c
#include <stdio.h>
#include <omp.h>

int main() {
    long r = 0, c = 0, a = 0;
    double tr, tc, ta;

    tr = omp_get_wtime();
    #pragma omp parallel for reduction(+:r)
    for(long i=0;i<100000000;i++)
        r += 1;
    tr = omp_get_wtime() - tr;

    tc = omp_get_wtime();
    #pragma omp parallel for
    for(long i=0;i<100000000;i++) {
        #pragma omp critical
        c += 1;
    }
    tc = omp_get_wtime() - tc;

    ta = omp_get_wtime();
    #pragma omp parallel for
    for(long i=0;i<100000000;i++) {
        #pragma omp atomic
        a += 1;
    }
    ta = omp_get_wtime() - ta;

    printf("Reduction: %ld  Time: %f sec\n", r, tr);
    printf("Critical:  %ld  Time: %f sec\n", c, tc);
    printf("Atomic:    %ld  Time: %f sec\n", a, ta);

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp reduction_critical_atomic.c
 -o reduction_critical_atomic
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./reduction_critical_atomic
Reduction: 100000000  Time: 0.025224 sec
Critical:  100000000  Time: 17.992683 sec
Atomic:    100000000  Time: 2.014947 sec
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The reduction clause performs the best (=0.025 sec) since each thread accumulates its own partial sum independently before OpenMP combines the results. The atomic directive is slower (=2.01 sec) because it protects each update but still requires fast hardware-level synchronization. The critical section performs the worst (=17.99 sec), as threads must wait sequentially to enter the protected region for every increment. These results demonstrate that reduction is the most efficient approach for parallel accumulation, whereas critical sections introduce significant overhead and should be avoided for high-frequency updates.

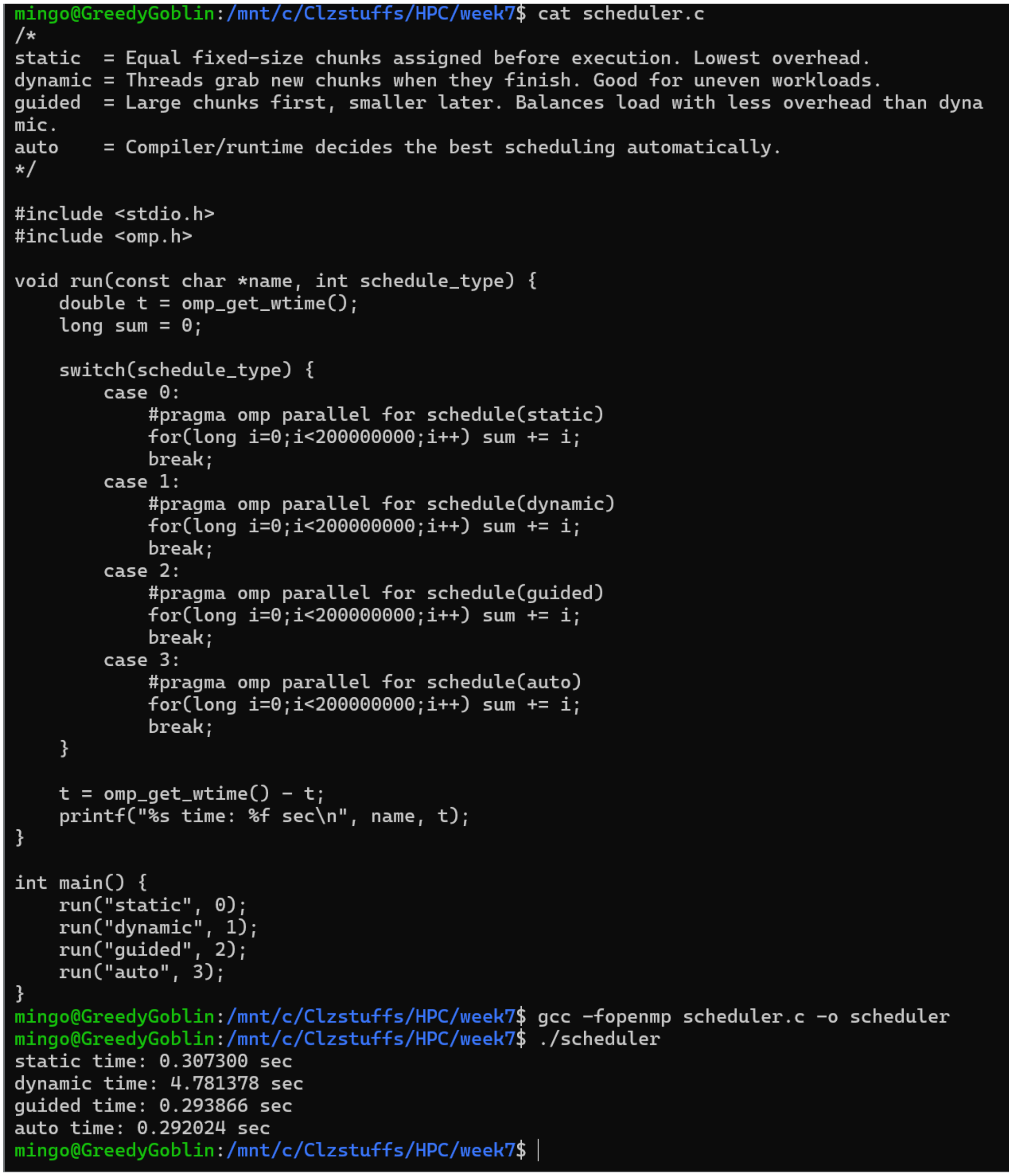
12.Reduction



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat reduction.c
#include <stdio.h>
#include <omp.h>

int main() {
    int sum = 0;
    #pragma omp parallel for
    for(int i = 1; i <= 100; i++) sum += i;

    printf("%d\n", sum);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp reduction.c -o reduction
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./reduction
3385
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

The loop runs in parallel using #pragma omp parallel for, but since multiple threads attempt to add to *sum* at the same time, their updates interfere with each other. This leads to lost additions and produces the incorrect final result of **3385** instead of the correct sum of 5050.

13.Scheduler



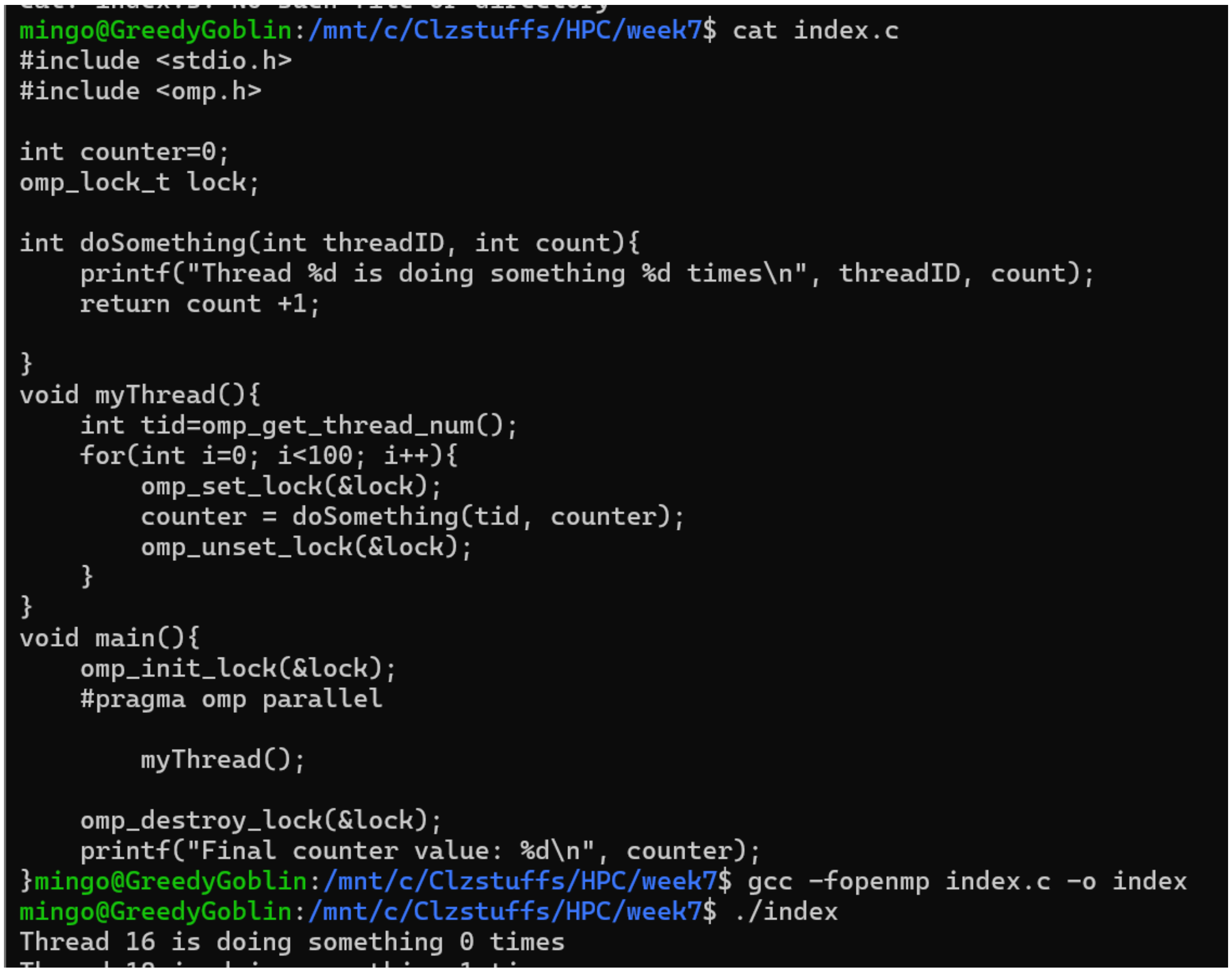
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat scheduler.c
/*
static  = Equal fixed-size chunks assigned before execution. Lowest overhead.
dynamic = Threads grab new chunks when they finish. Good for uneven workloads.
guided  = Large chunks first, smaller later. Balances load with less overhead than dyna
mic.
auto    = Compiler/runtime decides the best scheduling automatically.
*/

#include <stdio.h>
#include <omp.h>

void run(const char *name, int schedule_type) {
    double t = omp_get_wtime();
    long sum = 0;

    switch(schedule_type) {
        case 0:
            #pragma omp parallel for schedule(static)
            for(long i=0;i<200000000;i++) sum += i;
            break;
        case 1:
            #pragma omp parallel for schedule(dynamic)
            for(long i=0;i<200000000;i++) sum += i;
            break;
        case 2:
            #pragma omp parallel for schedule(guided)
            for(long i=0;i<200000000;i++) sum += i;
            break;
        case 3:
            #pragma omp parallel for schedule(auto)
            for(long i=0;i<200000000;i++) sum += i;
            break;
    }

    t = omp_get_wtime() - t;
    printf("%s time: %f sec\n", name, t);
}

int main() {
    run("static", 0);
    run("dynamic", 1);
    run("guided", 2);
    run("auto", 3);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp scheduler.c -o scheduler
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./scheduler
static time: 0.307300 sec
dynamic time: 4.781378 sec
guided time: 0.293866 sec
auto time: 0.292024 sec
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

Static (0.30s): Fast because the work is divided evenly before the loop starts. Each thread already knows exactly which iterations to run, so there is almost no scheduling overhead.

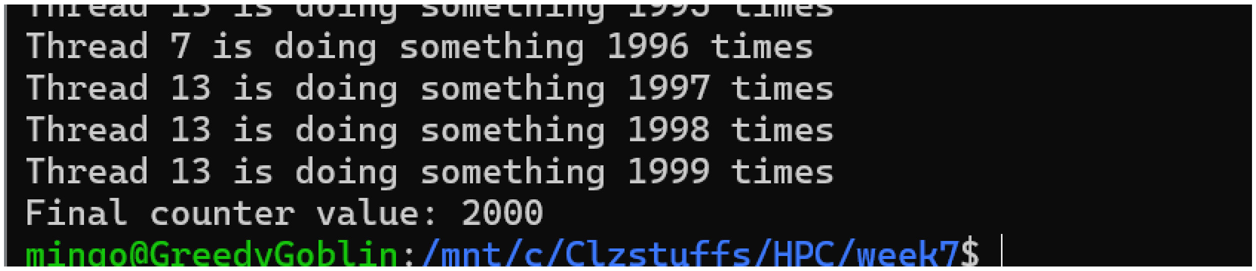
Guided (0.29s) & Auto (0.29s): Slightly faster than static because they begin with larger chunks to reduce overhead, and then switch to smaller chunks to keep the threads balanced. Auto lets OpenMP decide the most efficient strategy, and guided adjusts chunk sizes automatically.

Dynamic (4.78s): Much slower because threads repeatedly request small chunks of work during execution. Since every iteration takes the same amount of time, this constant "asking for new work" adds unnecessary overhead, making dynamic the slowest scheduling option.

14.



```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ cat index.c
#include <stdio.h>
#include <omp.h>

int counter=0;
omp_lock_t lock;

int doSomething(int threadID, int count){
    printf("Thread %d is doing something %d times\n", threadID, count);
    return count +1;

}
void myThread(){
    int tid=omp_get_thread_num();
    for(int i=0; i<100; i++){
        omp_set_lock(&lock);
        counter = doSomething(tid, counter);
        omp_unset_lock(&lock);
    }
}
void main(){
    omp_init_lock(&lock);
    #pragma omp parallel

        myThread();

    omp_destroy_lock(&lock);
    printf("Final counter value: %d\n", counter);
}mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ gcc -fopenmp index.c -o index
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$ ./index
Thread 16 is doing something 0 times
```



```
Thread 7 is doing something 1996 times
Thread 13 is doing something 1997 times
Thread 13 is doing something 1998 times
Thread 13 is doing something 1999 times
Final counter value: 2000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week7$
```

This program creates multiple threads, and each thread runs a loop where it locks a shared counter, prints a message showing its thread ID and how many times it has updated the counter, increments the counter, and then unlocks it