High Performance Computing
CS5013
Lab 02

R Abinav
ME23B1004

January 21, 2026

1 Implementation

1.1 Parallel Code Using Reduction Construct

```c
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include<time.h>

#define n 100000000

int main(){
    srand(time(NULL));
    double sum = 0;

    int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
    int num_tests = sizeof(threads)/sizeof(threads[0]);

    double t_1 = -1;

    double *numbers = (double*)malloc(n * sizeof(double));
    for(int i=0; i<n; i++){
        numbers[i] = (double)rand() * 1000000;
    }

    //warm up
    printf("Warm up\n");
    omp_set_num_threads(threads[0]);
    #pragma omp parallel reduction(+:sum)
    {
        #pragma omp for
        for(int i=0; i<n; i++){
            sum += numbers[i];
        }
    }
    printf("Warm up complete\n");

    for(int i=0; i<num_tests; i++){
        sum = 0;

        omp_set_num_threads(threads[i]);
        double s_time = omp_get_wtime();

        #pragma omp parallel reduction(+:sum)
        {
            #pragma omp for
            for(int j=0; j<n; j++){
                sum += numbers[j];
            }
        }

        printf("Sum: %lf\n", sum);

        double e_time = omp_get_wtime();
        double exec_time = e_time - s_time;
        printf("%d thread(s) exec time: %lf\n", threads[i],
           exec_time);

        double speedup = -1.00;
        if(i == 0){
            t_1 = exec_time;
            speedup = 1.00;
            printf("The speedup (for %d thread(s)) is: %lf\n",
                   threads[i], speedup);
        }else{
            speedup = (double)t_1/exec_time;
            printf("The speedup (for %d thread(s)) is: %lf\n",
                   threads[i], speedup);
        }

        double f = (threads[i] * (speedup - 1)) /
                   ((threads[i] - 1) * speedup);
        printf("Parallelisation factor (thread(s) = %d): %lf\n",
               threads[i], f);

        printf("\n\n");
    }

    free(numbers);
    return 0;
}
```

Listing 1: OpenMP Parallel Sum with Reduction

1.2 Parallel Code Using Critical Section

```c
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include<time.h>

#define n 100000000

int main(){
    srand(time(NULL));

    int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
    int num_tests = sizeof(threads)/sizeof(threads[0]);

    double t_1 = -1;

    double *numbers = (double*)malloc(n * sizeof(double));
    for(int i=0; i<n; i++){
        numbers[i] = (double)rand() * 1000000;
    }

    //warm up
    printf("Warm up\n");
    double sum_temp = 0;
    omp_set_num_threads(threads[0]);
    #pragma omp parallel
    {
        #pragma omp for
        for(int i=0; i<n; i++){
            sum_temp += numbers[i];
        }
    }
    printf("Warm up complete\n");

    for(int i=0; i<num_tests; i++){
        double sum = 0;
        omp_set_num_threads(threads[i]);
        double s_time = omp_get_wtime();

        #pragma omp parallel
        {
            #pragma omp for
            for(int j=0; j<n; j++){
                #pragma omp critical
                {
                    sum += numbers[j];
                }
            }
        }

        printf("Sum: %lf\n", sum);

        double e_time = omp_get_wtime();
        double exec_time = e_time - s_time;
        printf("%d thread(s) exec time: %lf\n", threads[i],
           exec_time);

        double speedup = -1.00;
        if(i == 0){
            t_1 = exec_time;
            speedup = 1.00;
            printf("The speedup (for %d thread(s)) is: %lf\n",
                   threads[i], speedup);
        }else{
            speedup = (double)t_1/exec_time;
            printf("The speedup (for %d thread(s)) is: %lf\n",
                   threads[i], speedup);
        }

        double f = (threads[i] * (speedup - 1)) /
                   ((threads[i] - 1) * speedup);
        printf("Parallelisation factor (thread(s) = %d): %lf\n",
               threads[i], f);

        printf("\n\n");
    }

    free(numbers);
    return 0;
}
```

Listing 2: OpenMP Parallel Sum with Critical Section

2 Results and Analysis

2.1 Performance Data - Reduction Construct

Table 1: Execution Time and Speedup with Reduction

Threads	Execution Time (s)	Speedup
1	0.348574	1.000000
2	0.187231	1.861731
4	0.096168	3.624639
6	0.088245	3.950057
8	0.056314	6.189830
10	0.063332	5.503914
12	0.058904	5.917702
16	0.060231	5.787323
20	0.071491	4.875756
32	0.057659	6.045454
64	0.067183	5.188453

2.2 Performance Data - Critical Section

Table 2: Execution Time and Speedup with Critical Section

Threads	Execution Time (s)	Speedup
1	0.635315	1.000000
2	6.664423	0.095329
4	12.303908	0.051635
6	29.000959	0.021907
8	24.990627	0.025422
10	5.889634	0.107870
12	6.047451	0.105055
16	6.623699	0.095915
20	7.340030	0.086555
32	7.197632	0.088267
64	6.885792	0.092265

2.3 Visualizations

2.3.1 Reduction Construct Performance

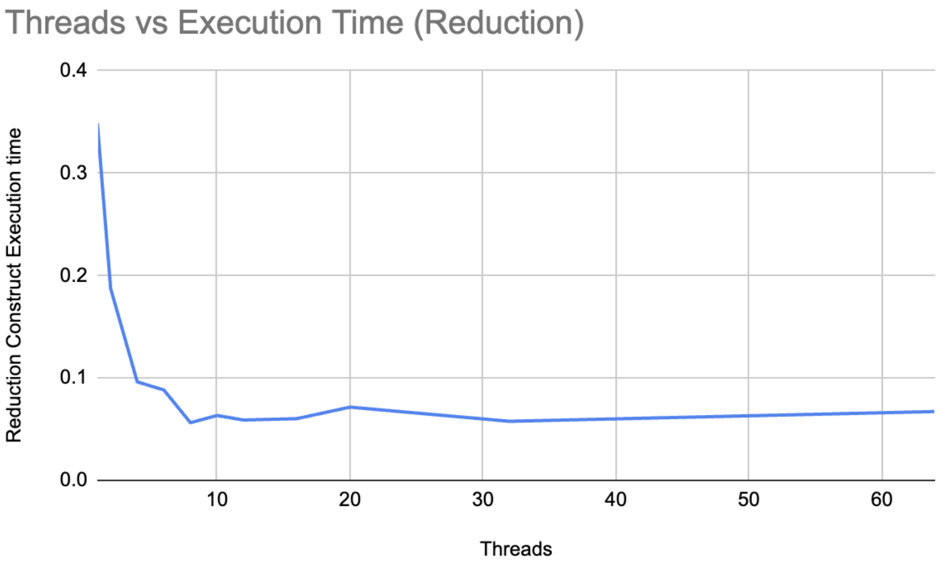

Figure 1: Threads vs Execution Time (Reduction)

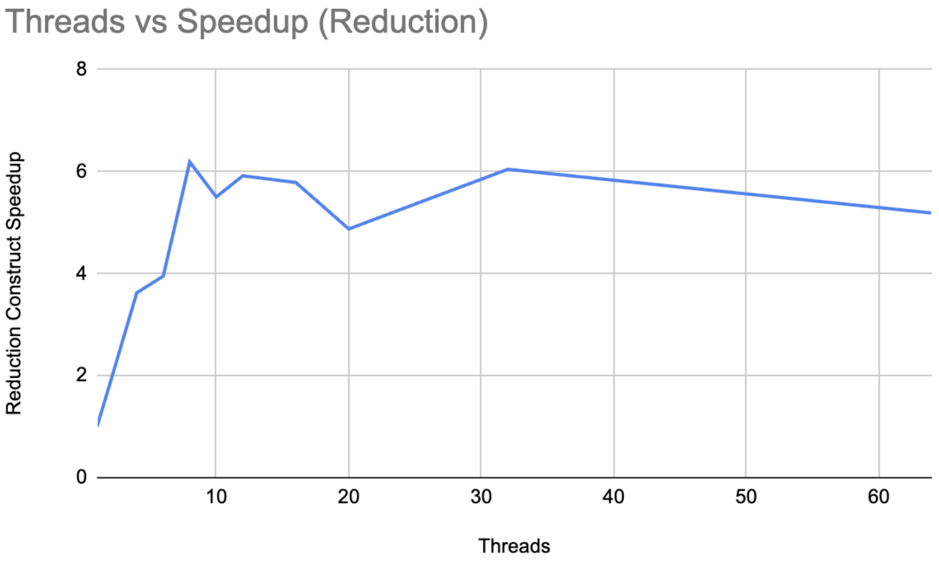

Figure 2: Threads vs Speedup (Reduction)

2.3.2 Critical Section Performance

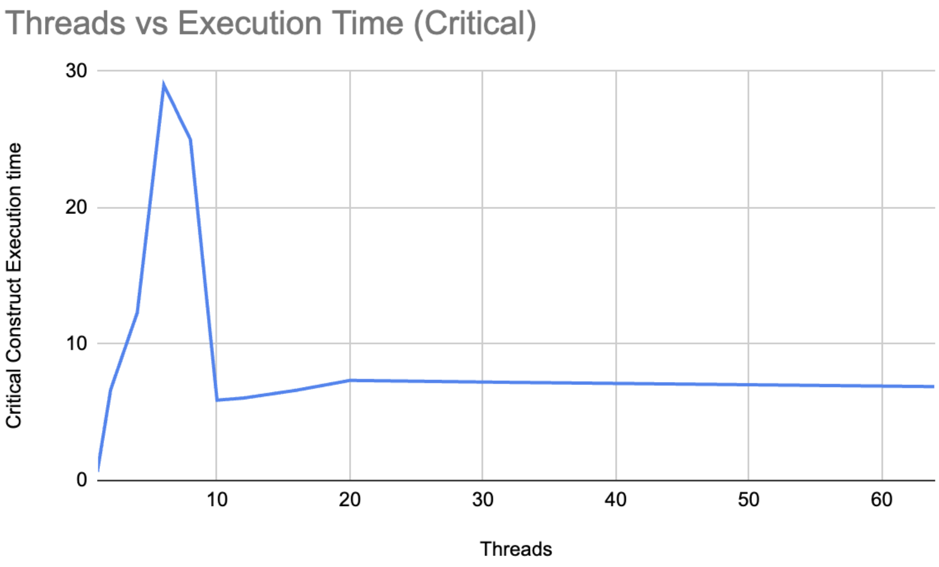

Figure 3: Threads vs Execution Time (Critical)

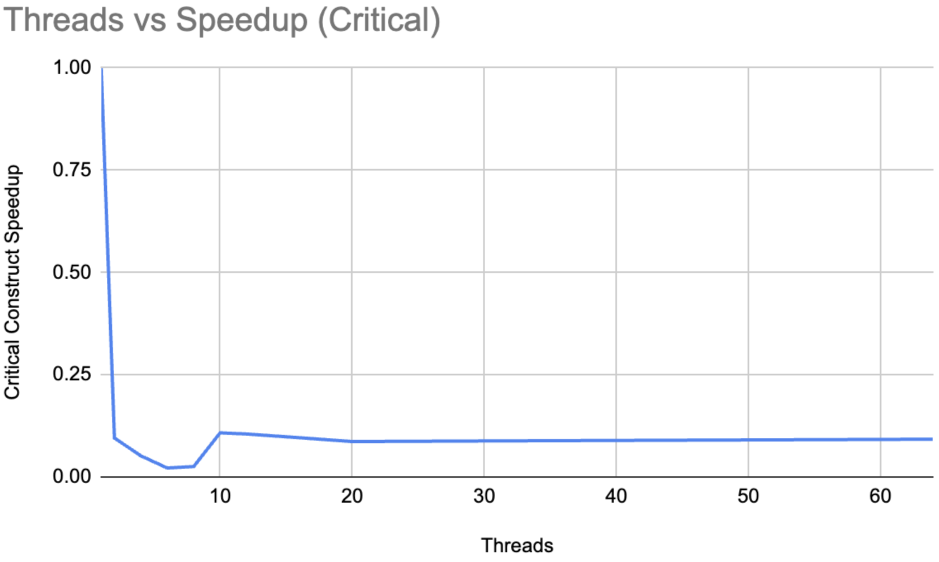

Figure 4: Threads vs Speedup (Critical)