Name: Shreeyash Shripad Dongarkar

PRN: 22510025

High Performance Computing Lab (B – 1)

Assignment – 3

Title: Study and Implementation of Schedule, No Wait, Reduction, Ordered and Collapse Clause

[GitHub Repository](https://github.com/TerminatorShri/22510025_HPCL)

Problem Statement – 1

Implement and Analyse a Parallel Code for below program using OpenMP

C Program to find minimum scalar product of two different vectors

Code:

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4. #include <time.h>

5.

6. int compare\_asc(const void \*a, const void \*b) {

7.     return (\*(int \*)a - \*(int \*)b);

8. }

9.

10. int compare\_desc(const void \*a, const void \*b) {

11.     return (\*(int \*)b - \*(int \*)a);

12. }

13.

14. void generate\_random\_vector(int \*vec, int n) {

15.     for (int i = 0; i < n; i++) {

16.         vec[i] = rand() % 100;

17.     }

18. }

19.

20. long long min\_scalar\_product(int \*A, int \*B, int n, int num\_threads) {

21.     qsort(A, n, sizeof(int), compare\_asc);

22.     qsort(B, n, sizeof(int), compare\_desc);

23.

24.     long long result = 0;

25.

26.     #pragma omp parallel for reduction(+:result) num\_threads(num\_threads)

27.     for (int i = 0; i < n; i++) {

28.         result += (long long)A[i] \* B[i];

29.     }

30.

31.     return result;

32. }

33.

34. int main() {

35.     int sizes[] = {1000000, 10000000, 50000000};

36.     int threads[] = {1, 2, 4, 8};

37.     FILE \*fp = fopen("PS\_1.csv", "w");

38.

39.     if (!fp) {

40.         perror("Error opening file");

41.         return 1;

42.     }

43.

44.     srand(time(NULL));

45.

46.     fprintf(fp, "Vector\_Size,Threads,Time(s),Min\_Scalar\_Product\n");

47.     printf("%-12s %-8s %-10s %-20s\n", "Size", "Threads", "Time(s)", "Min Scalar Product");

48.

49.     for (int s = 0; s < sizeof(sizes)/sizeof(sizes[0]); s++) {

50.         int n = sizes[s];

51.

52.         for (int t = 0; t < sizeof(threads)/sizeof(threads[0]); t++) {

53.             int thread\_count = threads[t];

54.

55.             int \*A = (int \*)malloc(n \* sizeof(int));

56.             int \*B = (int \*)malloc(n \* sizeof(int));

57.

58.             generate\_random\_vector(A, n);

59.             generate\_random\_vector(B, n);

60.

61.             double start = omp\_get\_wtime();

62.             long long result = min\_scalar\_product(A, B, n, thread\_count);

63.             double end = omp\_get\_wtime();

64.             double elapsed = end - start;

65.

66.             printf("%-12d %-8d %-10.6f %-20lld\n", n, thread\_count, elapsed, result);

67.             fprintf(fp, "%d,%d,%.6f,%lld\n", n, thread\_count, elapsed, result);

68.

69.             free(A);

70.             free(B);

71.         }

72.     }

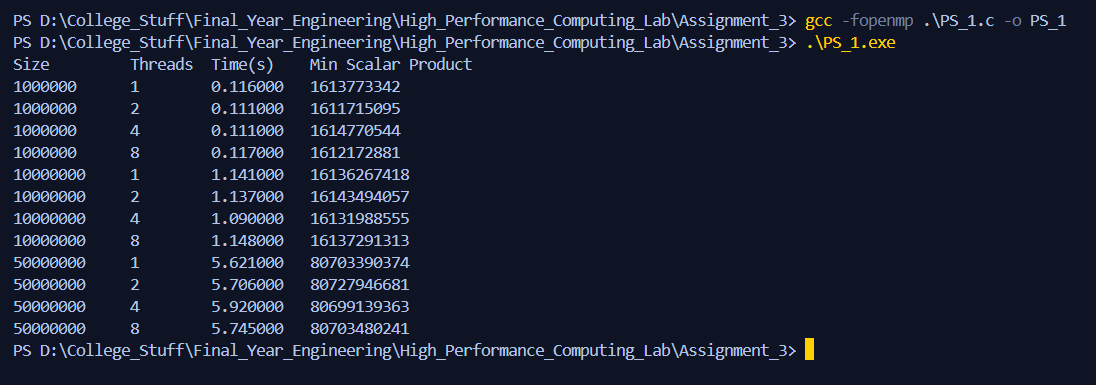
73.

74.     fclose(fp);

75.     return 0;

76. }

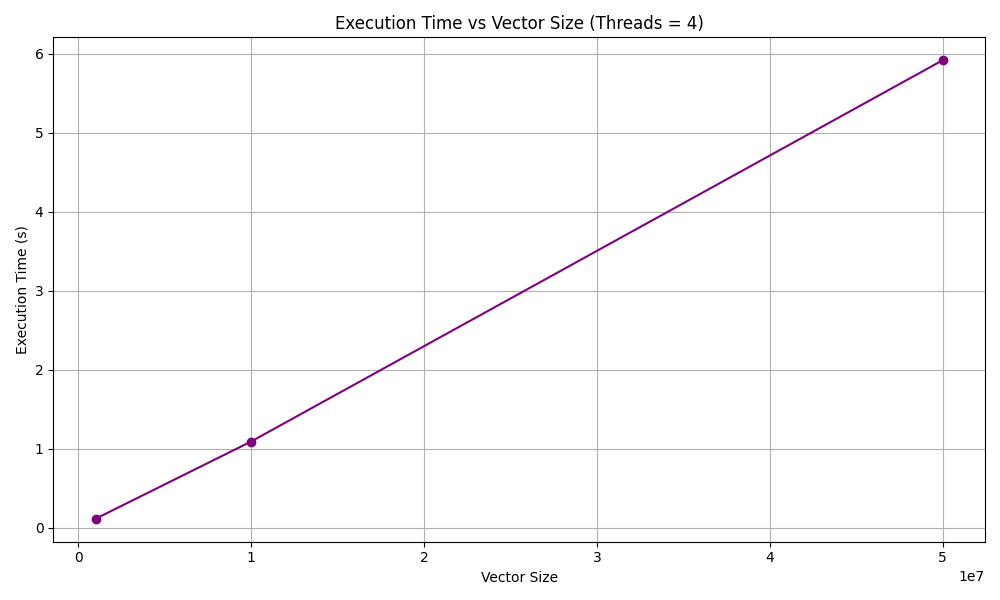
Output:

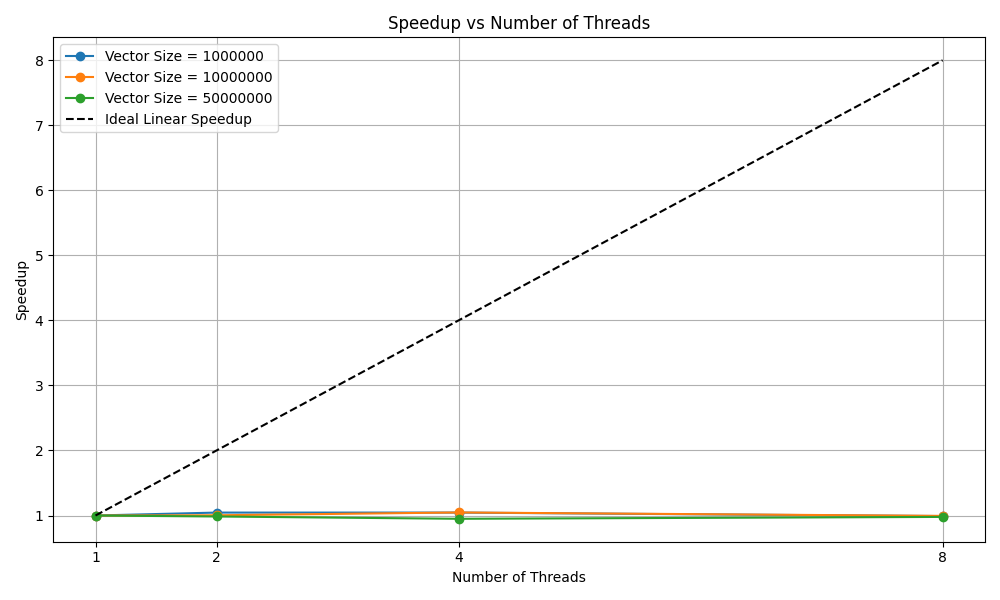


Analysis:



* For smaller vector sizes execution time is already low and on adding threads there’s a slight dip but almost negligible improvement overall this means overhead of thread management likely outweighs any benefit for such small workload.
* Also, for larger vector size increasing thread count may even increase execution time slightly which can be due to memory bandwidth bottlenecks, suboptimal thread scheduling or false sharing.





* For all vector sizes all lines are nearly flat which means increasing number of threads did not reduce execution time meaningfully. This shows no scaling benefit implying poor parallel performance.

Problem Statement – 2

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 1,2,4,8 and plot the speedup versus the number of threads.

ii. Explain whether or not the scaling behaviour is as expected.

Code:

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4. #include <time.h>

5.

6. void generate\_random\_matrix(int \*\*matrix, int size) {

7.     for (int i = 0; i < size; i++)

8.         for (int j = 0; j < size; j++)

9.             matrix[i][j] = rand() % 100;

10. }

11.

12. void matrix\_addition(int \*\*A, int \*\*B, int \*\*C, int size, int threads) {

13.     #pragma omp parallel for collapse(2) num\_threads(threads)

14.     for (int i = 0; i < size; i++)

15.         for (int j = 0; j < size; j++)

16.             C[i][j] = A[i][j] + B[i][j];

17. }

18.

19. int \*\*allocate\_matrix(int size) {

20.     int \*\*matrix = (int \*\*)malloc(size \* sizeof(int \*));

21.     for (int i = 0; i < size; i++)

22.         matrix[i] = (int \*)malloc(size \* sizeof(int));

23.     return matrix;

24. }

25.

26. void free\_matrix(int \*\*matrix, int size) {

27.     for (int i = 0; i < size; i++)

28.         free(matrix[i]);

29.     free(matrix);

30. }

31.

32. int main() {

33.     int sizes[] = {250, 500, 750, 1000, 2000};

34.     int thread\_counts[] = {1, 2, 4, 8};

35.

36.     FILE \*fp = fopen("matrix\_addition\_multithread.csv", "w");

37.     if (!fp) {

38.         perror("Error opening file");

39.         return 1;

40.     }

41.

42.     srand(time(NULL));

43.     fprintf(fp, "Matrix\_Size,Threads,Time(s)\n");

44.     printf("%-12s %-8s %-10s\n", "Size", "Threads", "Time(s)");

45.

46.     for (int s = 0; s < sizeof(sizes) / sizeof(sizes[0]); s++) {

47.         int size = sizes[s];

48.

49.         for (int t = 0; t < sizeof(thread\_counts) / sizeof(thread\_counts[0]); t++) {

50.             int threads = thread\_counts[t];

51.

52.             int \*\*A = allocate\_matrix(size);

53.             int \*\*B = allocate\_matrix(size);

54.             int \*\*C = allocate\_matrix(size);

55.

56.             generate\_random\_matrix(A, size);

57.             generate\_random\_matrix(B, size);

58.

59.             double start = omp\_get\_wtime();

60.             matrix\_addition(A, B, C, size, threads);

61.             double end = omp\_get\_wtime();

62.             double elapsed = end - start;

63.

64.             printf("%-12d %-8d %-10.6f\n", size, threads, elapsed);

65.             fprintf(fp, "%d,%d,%.6f\n", size, threads, elapsed);

66.

67.             free\_matrix(A, size);

68.             free\_matrix(B, size);

69.             free\_matrix(C, size);

70.         }

71.     }

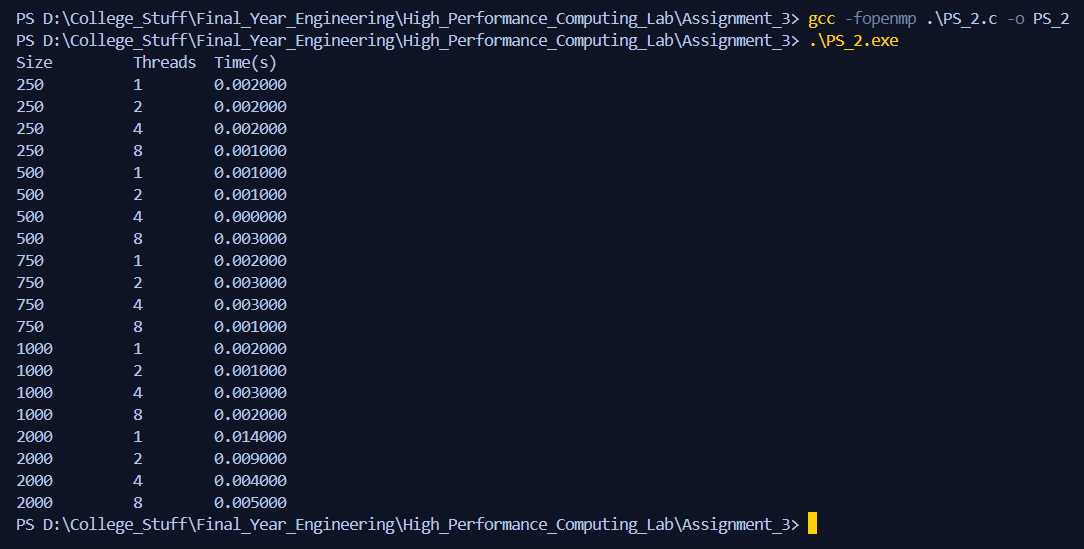
72.

73.     fclose(fp);

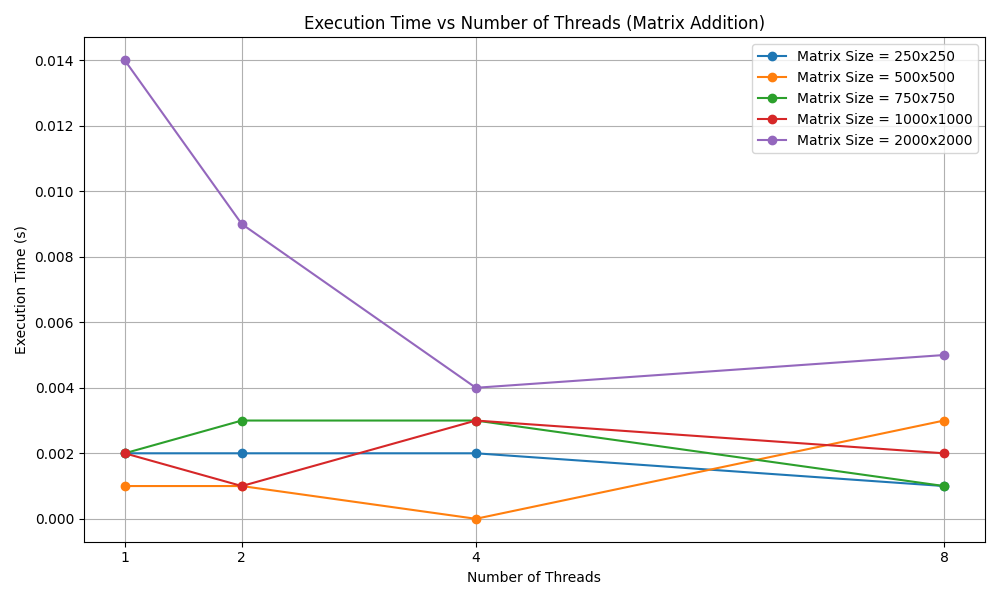
74.     return 0;

75. }

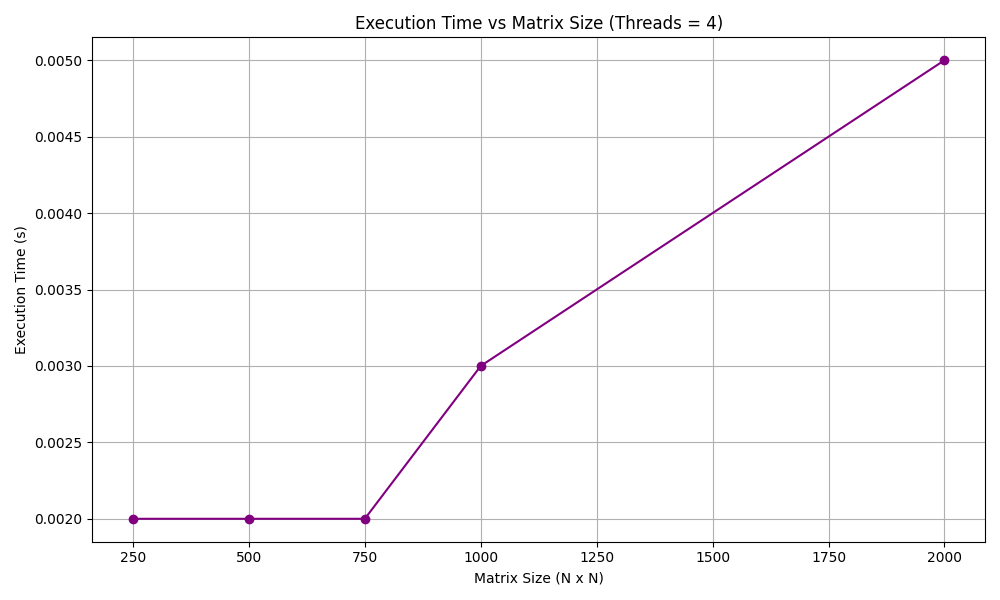
Output:

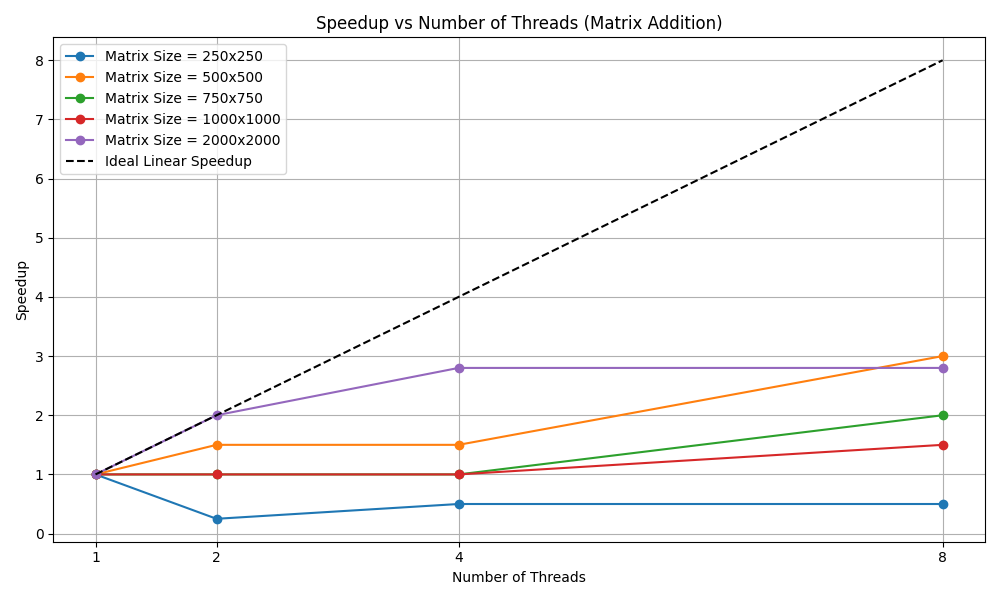


Analysis:



* For smaller matrix size execution times are extremely low, increasing thread count has low effect and may increase execution time due to thread management overhead.
* For medium sized matrices results are somewhat inconsistent as performance can depend on system load or thread scheduling.
* For larger size we can see significant reduction in execution time but after particular thread count performance starts degrading might be due to diminishing returns on parallelism.





* For smaller overhead of threading is more than work done hence parallelism hurts smaller problems.
* For medium size gradual increase in speedup yet far from ideal one due to overhead of thread management.
* For larger size more computation per thread gives better amortization of overhead showing best speedup.

Problem Statement – 3

For 1D Vector (size=200) and scalar addition, write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyse the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyse the speedup. iii. Demonstrate the use of nowait clause.

Code:

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4. #include <time.h>

5.

6. #define VECTOR\_SIZE 200

7. #define SCALAR 10

8.

9. void generate\_vector(int \*vec, int size) {

10.     for (int i = 0; i < size; i++)

11.         vec[i] = rand() % 100;

12. }

13.

14. void vector\_scalar\_add\_static(int \*vec, int \*result, int size, int chunk, int threads) {

15.     #pragma omp parallel for schedule(static, chunk) num\_threads(threads)

16.     for (int i = 0; i < size; i++) {

17.         result[i] = vec[i] + SCALAR;

18.     }

19. }

20.

21. void vector\_scalar\_add\_dynamic(int \*vec, int \*result, int size, int chunk, int threads) {

22.     #pragma omp parallel for schedule(dynamic, chunk) num\_threads(threads)

23.     for (int i = 0; i < size; i++) {

24.         result[i] = vec[i] + SCALAR;

25.     }

26. }

27.

28. void nowait\_demo(int \*vec, int \*res1, int \*res2, int size, int threads) {

29.     #pragma omp parallel num\_threads(threads)

30.     {

31.         #pragma omp for nowait

32.         for (int i = 0; i < size / 2; i++)

33.             res1[i] = vec[i] + 1;

34.

35.         #pragma omp for nowait

36.         for (int i = size / 2; i < size; i++)

37.             res2[i] = vec[i] + 2;

38.     }

39. }

40.

41. int main() {

42.     int chunk\_sizes[] = {1, 5, 10, 20, 50};

43.     int thread\_counts[] = {1, 2, 4, 8};

44.

45.     int \*vec = (int \*)malloc(VECTOR\_SIZE \* sizeof(int));

46.     int \*res\_static = (int \*)malloc(VECTOR\_SIZE \* sizeof(int));

47.     int \*res\_dynamic = (int \*)malloc(VECTOR\_SIZE \* sizeof(int));

48.     int \*res1 = (int \*)calloc(VECTOR\_SIZE, sizeof(int));

49.     int \*res2 = (int \*)calloc(VECTOR\_SIZE, sizeof(int));

50.

51.     srand(time(NULL));

52.     generate\_vector(vec, VECTOR\_SIZE);

53.

54.     FILE \*fs = fopen("PS\_2\_Static.csv", "w");

55.     FILE \*fd = fopen("PS\_2\_Dynamic.csv", "w");

56.     fprintf(fs, "Chunk\_Size,Threads,Time(s)\n");

57.     fprintf(fd, "Chunk\_Size,Threads,Time(s)\n");

58.

59.     printf("STATIC SCHEDULING:\n");

60.     printf("%-12s %-8s %-10s\n", "Chunk", "Threads", "Time(s)");

61.     for (int c = 0; c < sizeof(chunk\_sizes)/sizeof(chunk\_sizes[0]); c++) {

62.         for (int t = 0; t < sizeof(thread\_counts)/sizeof(thread\_counts[0]); t++) {

63.             double start = omp\_get\_wtime();

64.             vector\_scalar\_add\_static(vec, res\_static, VECTOR\_SIZE, chunk\_sizes[c], thread\_counts[t]);

65.             double end = omp\_get\_wtime();

66.             double elapsed = end - start;

67.

68.             printf("%-12d %-8d %-10.6f\n", chunk\_sizes[c], thread\_counts[t], elapsed);

69.             fprintf(fs, "%d,%d,%.6f\n", chunk\_sizes[c], thread\_counts[t], elapsed);

70.         }

71.     }

72.

73.     printf("\nDYNAMIC SCHEDULING:\n");

74.     printf("%-12s %-8s %-10s\n", "Chunk", "Threads", "Time(s)");

75.     for (int c = 0; c < sizeof(chunk\_sizes)/sizeof(chunk\_sizes[0]); c++) {

76.         for (int t = 0; t < sizeof(thread\_counts)/sizeof(thread\_counts[0]); t++) {

77.             double start = omp\_get\_wtime();

78.             vector\_scalar\_add\_dynamic(vec, res\_dynamic, VECTOR\_SIZE, chunk\_sizes[c], thread\_counts[t]);

79.             double end = omp\_get\_wtime();

80.             double elapsed = end - start;

81.

82.             printf("%-12d %-8d %-10.6f\n", chunk\_sizes[c], thread\_counts[t], elapsed);

83.             fprintf(fd, "%d,%d,%.6f\n", chunk\_sizes[c], thread\_counts[t], elapsed);

84.         }

85.     }

86.

87.     fclose(fs);

88.     fclose(fd);

89.

90.     printf("\nNOWAIT CLAUSE DEMO (partial result shown):\n");

91.     nowait\_demo(vec, res1, res2, VECTOR\_SIZE, 4);

92.     for (int i = 0; i < 10; i++)

93.         printf("res1[%d] = %d, res2[%d] = %d\n", i, res1[i], i, res2[i]);

94.

95.     free(vec);

96.     free(res\_static);

97.     free(res\_dynamic);

98.     free(res1);

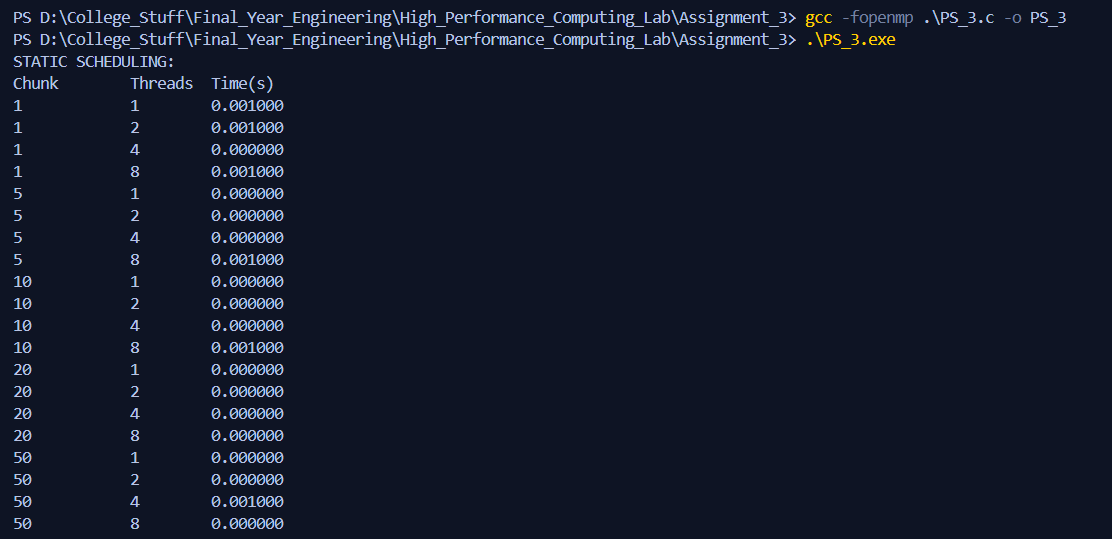
99.     free(res2);

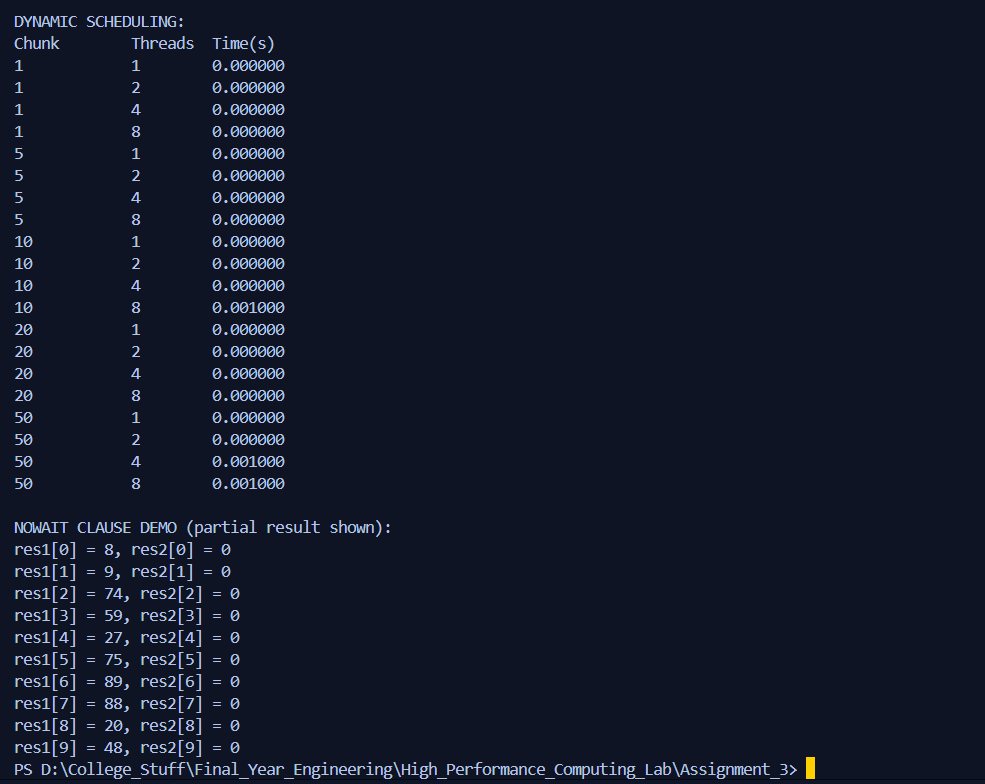
100.

101.     return 0;

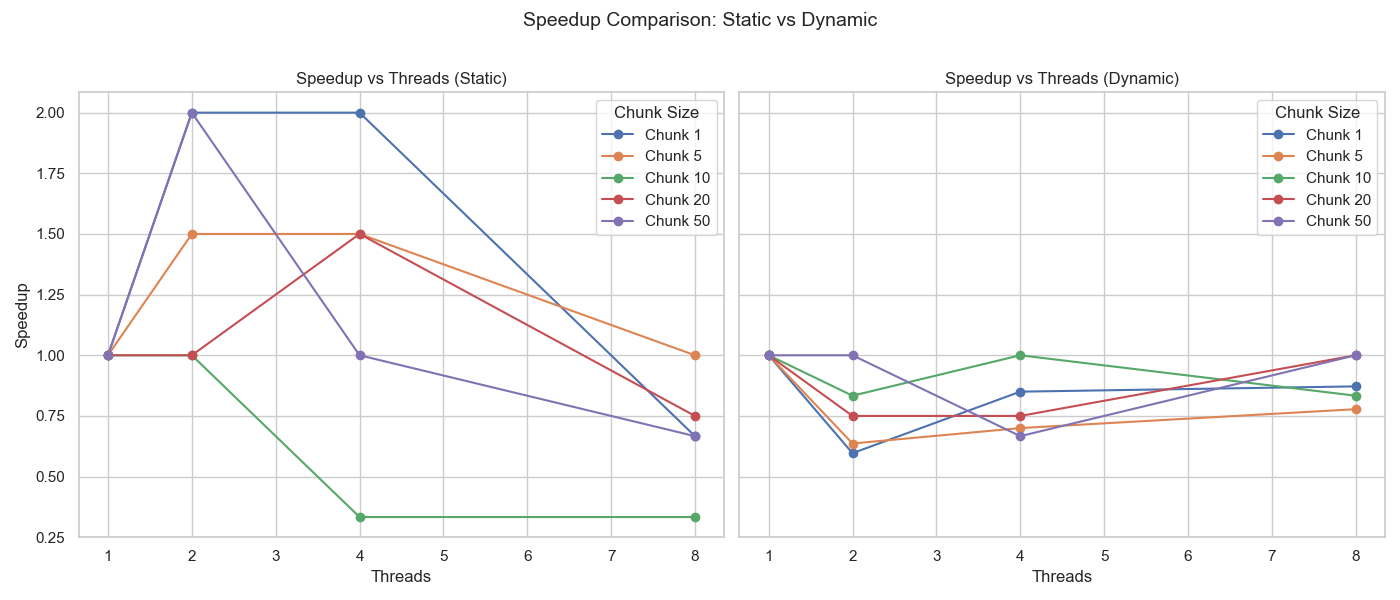
102. }

Output:





Analysis:

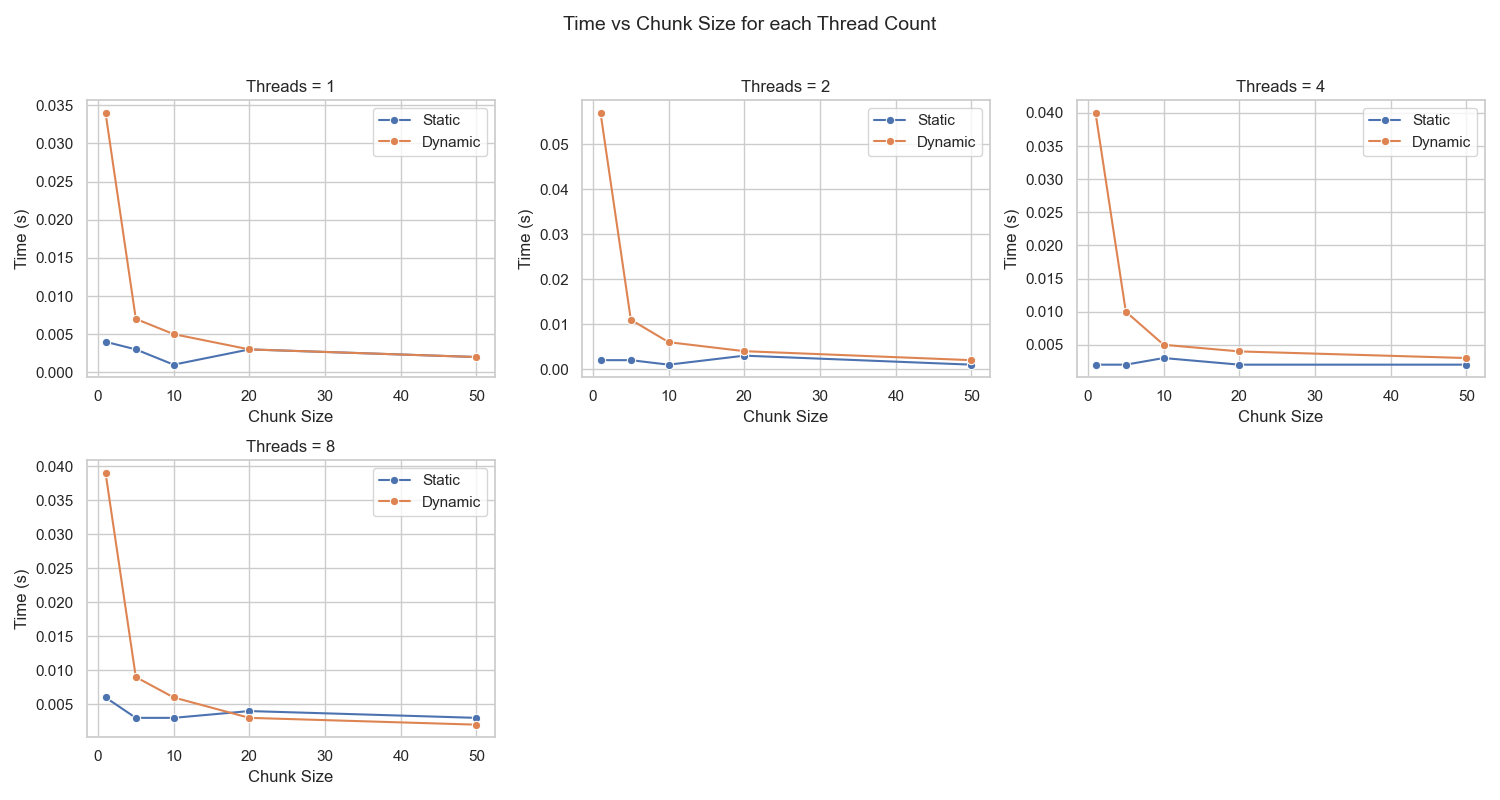


Static Scheduling:

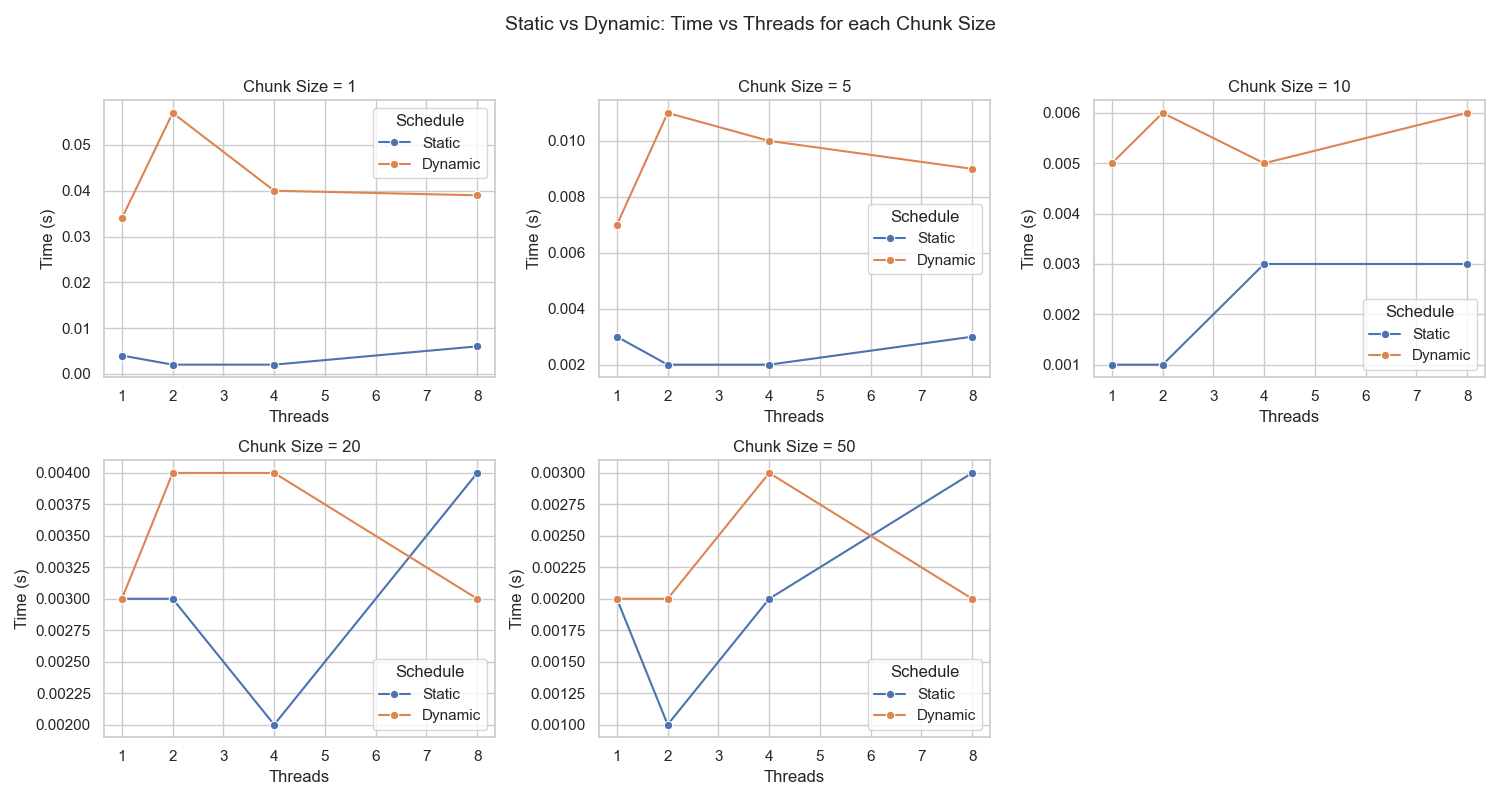
* For smaller chunk size best performance but then degrades might be due to thread management overhead.
* For medium chunk size reasonable performance but in some cases, we can see poor load balancing.
* For larger chunk size we can see for larger chunks this doesn’t scale well.
* Static works best when iteration time is uniform and predictable.

Dynamic Scheduling:

* Much more stable speedup across chunks, no major spikes or drops but performance is not that impressive.
* Useful when workload per iteration varies, but here it's uniform → so overhead dominates.



* For smaller thread count static shows better performance as dynamic has higher overhead due to unnecessary scheduling logic and higher overhead of dynamic allocation and lock contention.
* Static performs overall better providing best trade-offs between load balance and scheduling overhead.



* For smaller chunk size and for smaller threads static is significantly better due to massive scheduling overhead but at higher thread counts dynamic improves and might outperform static at sometimes suggesting dynamic paying off at higher thread counts and reasonable chunk size.