1. OpenMP program to calculate pow(i, x) for all threads

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

int main() {
   int i;
   printf("Enter the value of i: ");
   scanf("%d", &i);
   #pragma omp parallel
   {
      int x = omp_get_thread_num();
      double result = pow(i, x);
      printf("Thread %d: %d^%d = %f\n", x, i, x, result);
   }
   return 0;
}
```

2. OpenMP program to sum even and odd numbers in an array

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

int main() {
   int n;
   printf("Enter the number of elements in the array: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter the elements of the array: ");
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
```

```
    }

    int sum_even = 0, sum_odd = 0;

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for (int i = 0; i < n; i++) {
                if (arr[i] % 2 == 0) {
                    sum_even += arr[i];
                }
            }
            printf("Sum of even numbers: %d\n", sum_even);
        }

        #pragma omp section
        {
            for (int i = 0; i < n; i++) {
                if (arr[i] % 2 != 0) {
                    sum_odd += arr[i];
                }
            }
            printf("Sum of odd numbers: %d\n", sum_odd);
        }
    }
    return 0;
}
```

3. OpenMP program to implement basic calculator operations

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

int main() {
  int a, b;
  printf("Enter two numbers: ");
  scanf("%d %d", &a, &b);
  int sum, diff, prod;
  float quot;

  #pragma omp parallel sections
  {
    #pragma omp section
    {
      sum = a + b;
      printf("Sum: %d\n", sum);
    }

    #pragma omp section
    {
      diff = a - b;
      printf("Difference: %d\n", diff);
    }

    #pragma omp section
    {
      prod = a * b;
      printf("Product: %d\n", prod);
    }
```

```
    #pragma omp section
    {
      if (b != 0) {
        quot = (float)a / b;
        printf("Quotient: %.2f\n", quot);
      } else {
        printf("Quotient: undefined (division by zero)\n");
      }
    }
  }
  return 0;
}
```

4. OpenMP program to perform arithmetic operations on two vectors

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

int main() {
  int n = 4; // Vector size
  int A[n], B[n], C[n];
  printf("Enter elements of vector A: ");
  for (int i = 0; i < n; i++) {
    scanf("%d", &A[i]);
  }
  printf("Enter elements of vector B: ");
  for (int i = 0; i < n; i++) {
    scanf("%d", &B[i]);
  }

  #pragma omp parallel sections
  {
```

```
#pragma omp section
{
   for (int i = 0; i < n; i++) {
      C[i] = A[i] + B[i];
      printf("Add: C[%d] = %d\n", i, C[i]);
   }
}

#pragma omp section
{
   for (int i = 0; i < n; i++) {
      C[i] = A[i] - B[i];
      printf("Sub: C[%d] = %d\n", i, C[i]);
   }
}

#pragma omp section
{
   for (int i = 0; i < n; i++) {
      C[i] = A[i] * B[i];
      printf("Mul: C[%d] = %d\n", i, C[i]);
   }
}

#pragma omp section
{
   for (int i = 0; i < n; i++) {
      if (B[i] != 0) {
         C[i] = A[i] / B[i];
         printf("Div: C[%d] = %d\n", i, C[i]);
      } else {
```

```
            printf("Div: C[%d] = undefined (division by zero)\n", i);
          }
        }
      }
    }
    return 0;
}
```

5. OpenMP program to generate prime numbers in a range

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

int is_prime(int num) {
    if (num <= 1) return 0;
    for (int i = 2; i*i <= num; i++) {
        if (num % i == 0) return 0;
    }
    return 1;
}

int main() {
    int start, end;
    printf("Enter the start and end of the range: ");
    scanf("%d %d", &start, &end);

    #pragma omp parallel for
    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            printf("Prime number: %d\n", i);
        }
    }
```

```
    return 0;
}
```

6. OpenMP program to toggle characters in a string based on thread ID

```
#include <stdio.h>
#include <omp.h>
#include <ctype.h>
#include <string.h>

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);
    int n = strlen(str);

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        int thread_id = omp_get_thread_num();
        if (isupper(str[i])) {
            str[i] = tolower(str[i]);
        } else {
            str[i] = toupper(str[i]);
        }
        printf("Thread %d: %c\n", thread_id, str[i]);
    }

    printf("Toggled string: %s\n", str);
    return 0;
}
```

7. OpenMP program to compute Fibonacci numbers for an array

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

int fibonacci(int n) {
  if (n <= 1) return n;
  return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
  int A[] = {10, 13, 5, 6};
  int n = sizeof(A) / sizeof(A[0]);
  int result[n];

  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
    result[i] = fibonacci(A[i]);
    printf("Thread %d: Fibonacci of %d is %d\n", omp_get_thread_num(), A[i], result[i]);
  }
  return 0;
}
```

8. OpenMP program to implement Matrix multiplication and analyze speedup and efficiency

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

void matrix_multiply(int **A, int **B, int **C, int size, int num_threads) {
```

```
    omp_set_num_threads(num_threads);
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i][j] = 0;
            for (int k = 0; k < size; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int sizes[] = {200, 400, 600, 800, 1000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    int num_threads_arr[] = {1, 2, 4, 6, 8};
    int num_threads_count = sizeof(num_threads_arr) / sizeof(num_threads_arr[0]);

    for (int s = 0; s < num_sizes; s++) {
        int size = sizes[s];
        int **A = (int **)malloc(size * sizeof(int *));
        int **B = (int **)malloc(size * sizeof(int *));
        int **C = (int **)malloc(size * sizeof(int *));
        for (int i = 0; i < size; i++) {
            A[i] = (int *)malloc(size * sizeof(int));
            B[i] = (int *)malloc(size * sizeof(int));
            C[i] = (int *)malloc(size * sizeof(int));
        }

        // Initialize matrices
        for (int i = 0; i < size; i++) {
```

```
            for (int j = 0; j < size; j++) {
                A[i][j] = rand() % 10;
                B[i][j] = rand() % 10;
            }
        }

        for (int t = 0; t < num_threads_count; t++) {
            int num_threads = num_threads_arr[t];
            double start_time = omp_get_wtime();
            matrix_multiply(A, B, C, size, num_threads);
            double end_time = omp_get_wtime();
            printf("Size: %d, Threads: %d, Time: %f seconds\n", size, num_threads, end_time -
start_time);
        }

        // Free memory
        for (int i = 0; i < size; i++) {
            free(A[i]);
            free(B[i]);
            free(C[i]);
        }
        free(A);
        free(B);
        free(C);
    }

    return 0;
}
```

9. OpenMP program to perform Matrix times vector multiplication and analyze speedup and efficiency

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

void matrix_vector_multiply(int **A, int *B, int *C, int rows, int cols, int num_threads) {
   omp_set_num_threads(num_threads);
   #pragma omp parallel for
   for (int i = 0; i < rows; i++) {
      C[i] = 0;
      for (int j = 0; j < cols; j++) {
         C[i] += A[i][j] * B[j];
      }
   }
}

int main() {
   int sizes[] = {200, 400, 600, 800, 1000};
   int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
   int num_threads_arr[] = {1, 2, 4, 6, 8};
   int num_threads_count = sizeof(num_threads_arr) / sizeof(num_threads_arr[0]);

   for (int s = 0; s < num_sizes; s++) {
      int size = sizes[s];
      int **A = (int **)malloc(size * sizeof(int *));
      int *B = (int *)malloc(size * sizeof(int));
      int *C = (int *)malloc(size * sizeof(int));
      for (int i = 0; i < size; i++) {
         A[i] = (int *)malloc(size * sizeof(int));
      }
```

```
    // Initialize matrix and vector
    for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
        A[i][j] = rand() % 10;
      }
      B[i] = rand() % 10;
    }

    for (int t = 0; t < num_threads_count; t++) {
      int num_threads = num_threads_arr[t];
      double start_time = omp_get_wtime();
      matrix_vector_multiply(A, B, C, size, size, num_threads);
      double end_time = omp_get_wtime();
      printf("Size: %d, Threads: %d, Time: %f seconds\n", size, num_threads, end_time -
start_time);
    }

    // Free memory
    for (int i = 0; i < size; i++) {
      free(A[i]);
    }
    free(A);
    free(B);
    free(C);
  }

  return 0;
}
```