Source code

```
/*Instruction level Parallelism*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],b[10],c[10],d[10],i;
clrscr();
printf("\n enter array element");
for(i=0;i<5;i++)
{
scanf("%d",&a[i]);
}
printf("\n enter second array element");
for(i=0;i<5;i++)
{
scanf("%d",&b[i]);
}
a[0]=a[0]+b[0];
for(i=0;i<5;i++)
{
a[i+1]=a[i]+b[i];
b[i+1]=a[i+1]+b[i+1];
}
b[i+1]=a[i]+b[i];
for(i=0;i<5;i++)
{
printf("\n matrix a%d",a[i]);
printf("\n matrix b%d",b[i]);
}
getch();
}
```

Output

Result:

Thus the above program has been successfully executed and output are verified.

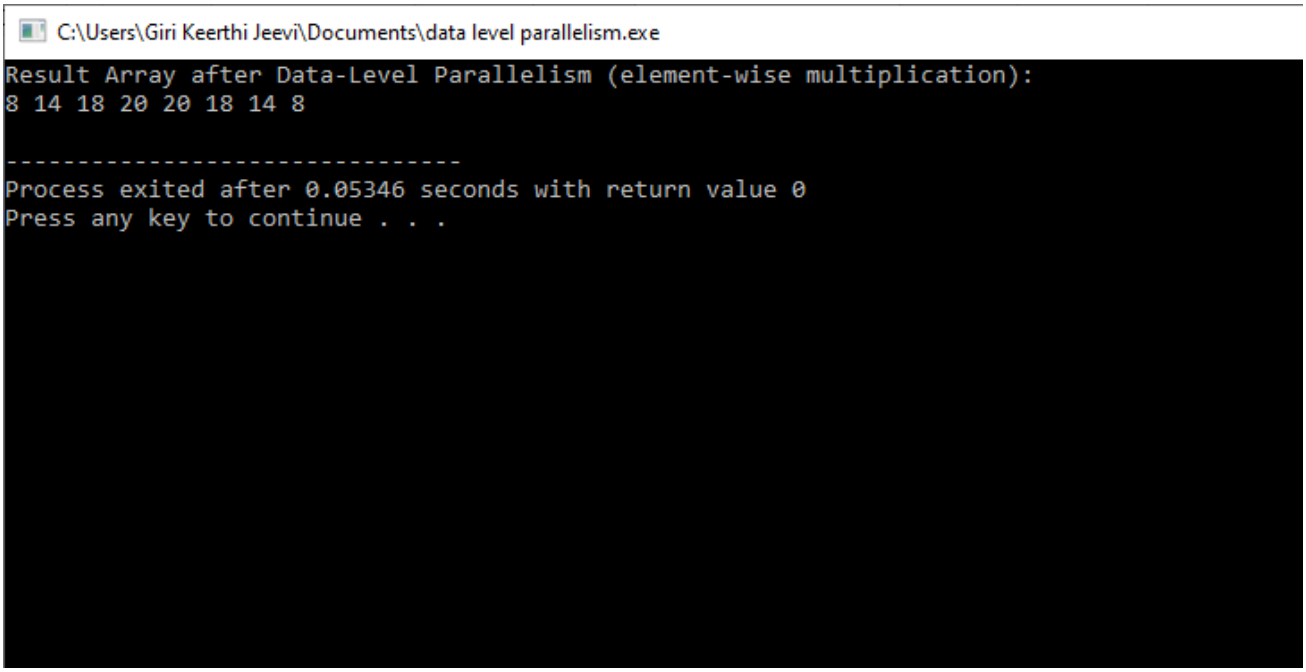
Source code

/*Data level parallelism*/

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define ARRAY_SIZE 8
#define NUM_THREADS 2

typedef struct {
    int* array_a;
    int* array_b;
    int* result;
    int start;
    int end;
} ThreadData;

void* multiply_elements(void* arg) {
    ThreadData* thread_data = (ThreadData*)arg;

    for (int i = thread_data->start; i < thread_data->end; i++) {
        thread_data->result[i] = thread_data->array_a[i] * thread_data->array_b[i];
  }
    pthread_exit(NULL);
}
int main() {
    int array_a[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
    int array_b[ARRAY_SIZE] = {8, 7, 6, 5, 4, 3, 2, 1};
    int result[ARRAY_SIZE];
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];

    int chunk_size = ARRAY_SIZE / NUM_THREADS;
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_data[i].array_a = array_a;
        thread_data[i].array_b = array_b;
        thread_data[i].result = result;
        thread_data[i].start = i * chunk_size;
        thread_data[i].end = (i == NUM_THREADS - 1) ? ARRAY_SIZE : (i + 1) * chunk_size;

        pthread_create(&threads[i], NULL, multiply_elements, (void*)&thread_data[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Result Array after Data-Level Parallelism (element-wise multiplication):\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");
    return 0;
}
```

Output



Result

Thus the above program has been successfully executed and output are verified.

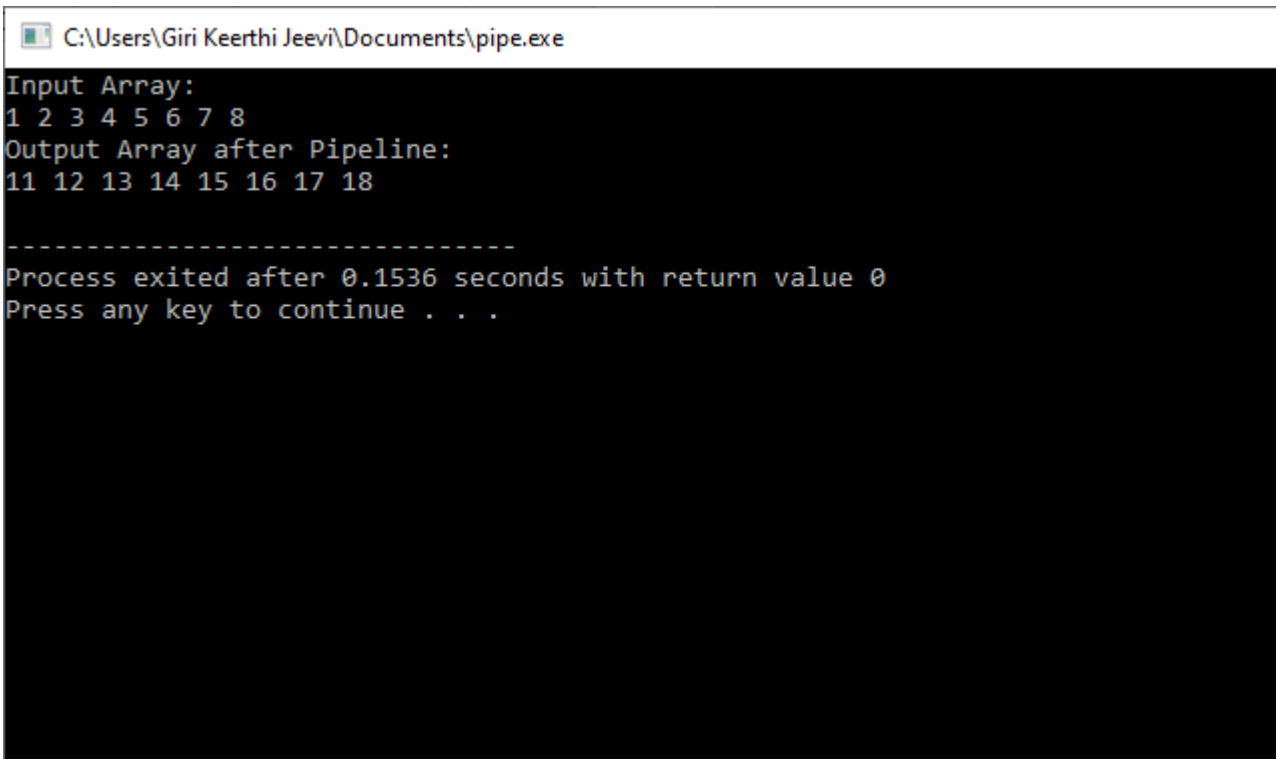
Source code

/*Pipeline*/

```
#include <stdio.h>
#define ARRAY_SIZE 8
// Stage 1: Load data
void load_data(int* input, int* buffer) {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        buffer[i] = input[i];
    }
}
// Stage 2: Process data
void process_data(int* buffer) {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        buffer[i] += 10;  // Simulating some processing
    }
}
// Stage 3: Store data
void store_data(int* buffer, int* output) {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        output[i] = buffer[i];
    }
}
int main() {
    int input_array[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8};
    int buffer[ARRAY_SIZE];
    int output_array[ARRAY_SIZE];

  // Pipeline Stage 1: Load data
    load_data(input_array, buffer);

    // Pipeline Stage 2: Process data
    process_data(buffer);
```

```
    // Pipeline Stage 3: Store data
    store_data(buffer, output_array);

    // Display results
    printf("Input Array:\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", input_array[i]);
    }
    printf("\n");

    printf("Output Array after Pipeline:\n");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", output_array[i]);
    }
    printf("\n");

    return 0;
}
```

Output



Result

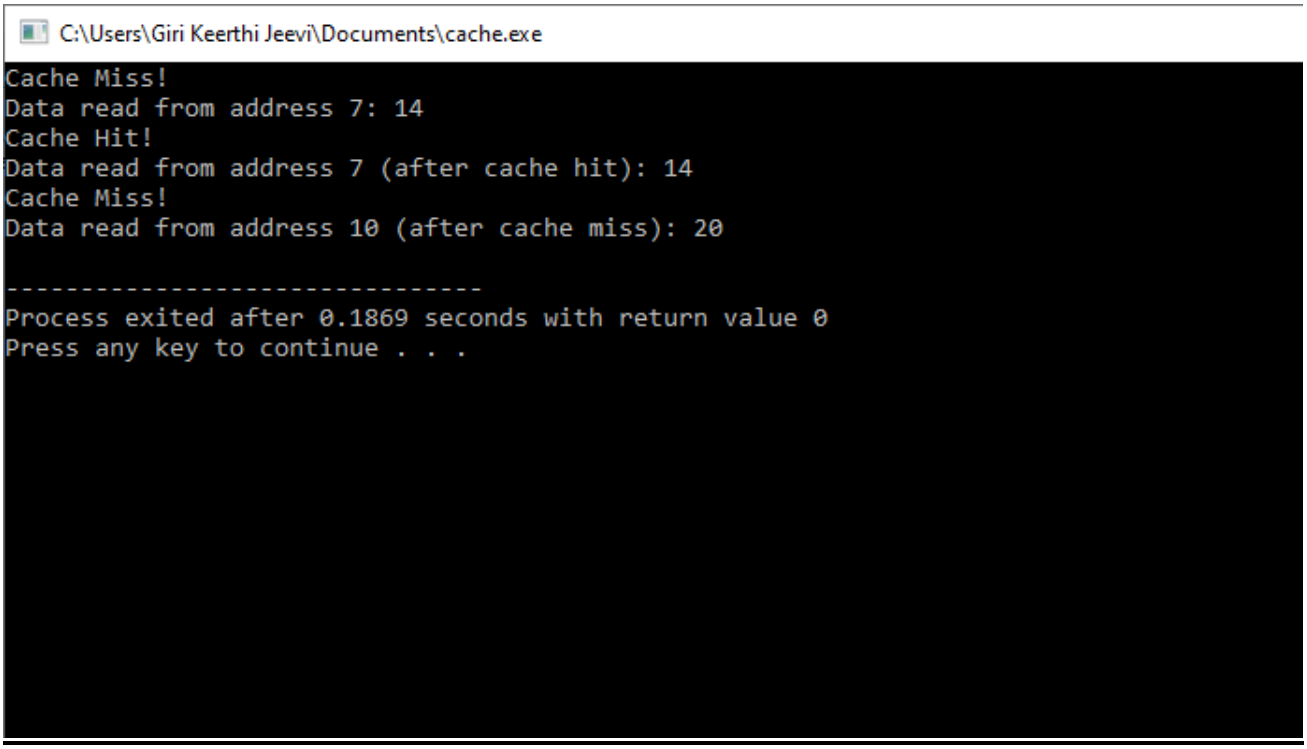
Thus the above program has been successfully executed and output are verified.

Source code

```
/*Cache*/
#include <stdio.h>
#include <stdlib.h>
#define CACHE_SIZE 4
typedef struct {
   int valid;
   int tag;
   int data;
} CacheLine;

void initializeCache(CacheLine* cache, int cacheSize) {
   for (int i = 0; i < cacheSize; i++) {
      cache[i].valid = 0;
      cache[i].tag = -1;
      cache[i].data = 0;
   }
}
int readFromMemory(int address) {
   // Simulating reading data from memory
   return address * 2;  // Just a simple example, you can replace this with real data retrieval logic
}
int readFromCache(CacheLine* cache, int address) {
   int index = address % CACHE_SIZE;
   int tag = address / CACHE_SIZE;
   if (cache[index].valid && cache[index].tag == tag) {
      // Cache hit
      printf("Cache Hit!\n");
      return cache[index].data;
   } else {
```

```
            // Cache miss
            printf("Cache Miss!\n");
            // Simulating reading data from memory
            int data = readFromMemory(address);

            // Update the cache
            cache[index].valid = 1;
            cache[index].tag = tag;
            cache[index].data = data;
            return data;
        }
    }
    int main() {
        CacheLine cache[CACHE_SIZE];
        initializeCache(cache, CACHE_SIZE);
        int readAddress = 7;

        // Read from cache
        int readData = readFromCache(cache, readAddress);
        printf("Data read from address %d: %d\n", readAddress, readData);

        // Read the same address again (cache hit)
        readData = readFromCache(cache, readAddress);
        printf("Data read from address %d (after cache hit): %d\n", readAddress,
readData);
        int newAddress = 10;

        // Read from a different address (cache miss)
        int newData = readFromCache(cache, newAddress);
printf("Data read from address %d (after cache miss): %d\n", newAddress, newData);
        return 0;
    }
```

Output



Result

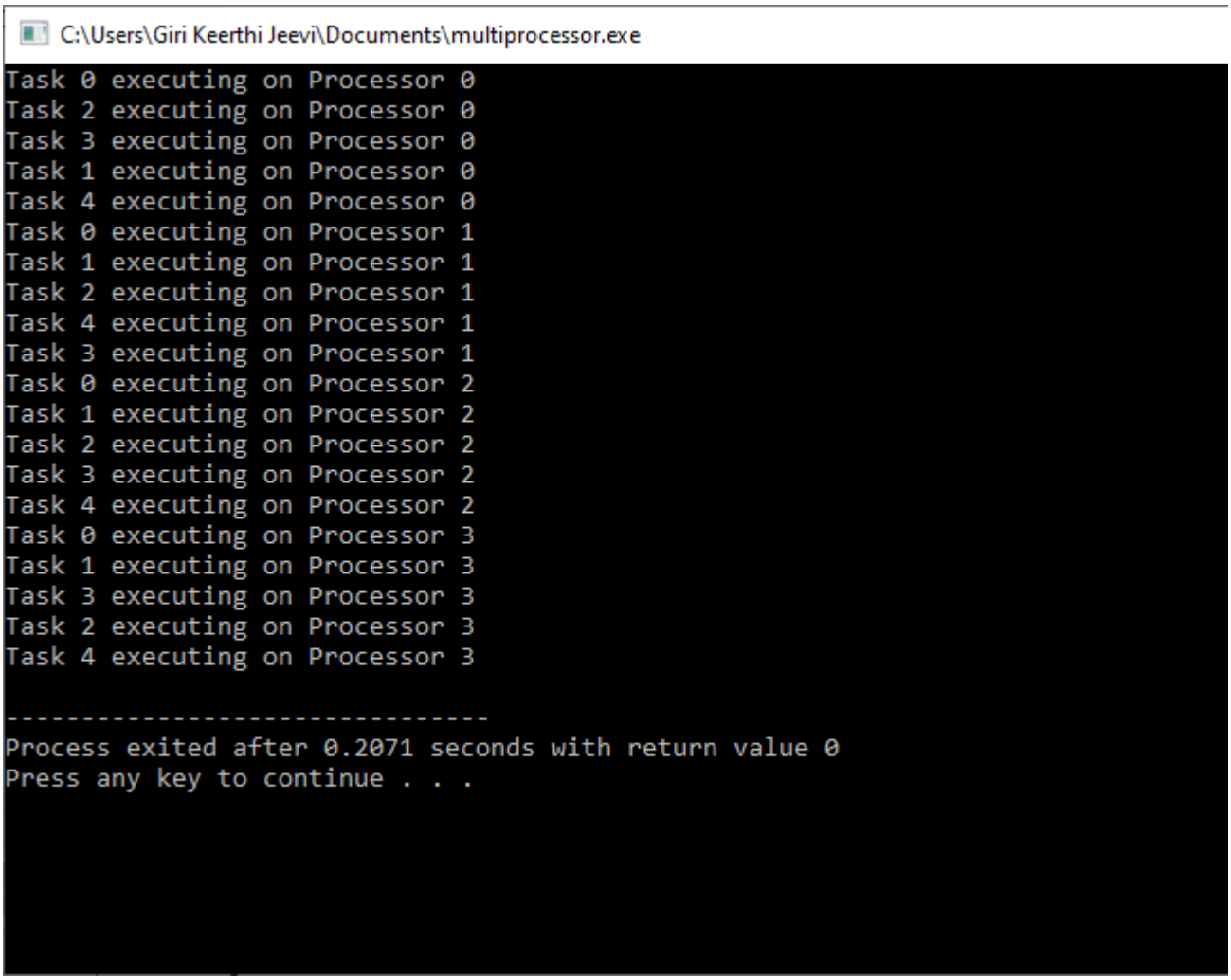
Thus the above program has been successfully executed and output are verified.

Source code

```
/*Multi processor*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_PROCESSORS 4
#define TASKS_PER_PROCESSOR 5

// Data structure to represent a task
typedef struct {
    int id;
    int processor_id;
} Task;

// Function to simulate task execution
void* executeTask(void* arg) {
    Task* task = (Task*)arg;
    printf("Task %d executing on Processor %d\n", task->id, task->processor_id);
    // Additional task execution logic goes here
    return NULL;
}
int main() {
    // Initialize pthread variables
    pthread_t processors[NUM_PROCESSORS];

    // Simulate tasks
    for (int i = 0; i < NUM_PROCESSORS; ++i) {
        for (int j = 0; j < TASKS_PER_PROCESSOR; ++j) {
            // Create a task
            Task* task = (Task*)malloc(sizeof(Task));
```

```
            task->id = j;
            task->processor_id = i;

            // Simulate task execution on a separate thread
            pthread_create(&processors[i], NULL, executeTask, (void*)task);
        }
    }
    // Wait for all threads to finish
    for (int i = 0; i < NUM_PROCESSORS; ++i) {
        pthread_join(processors[i], NULL);
    }
    return 0;
}
```

Output



Result

Thus the above program has been successfully executed and output are verified.

Source code

```
/*Vector processor*/
#include <stdio.h>
#include <stdlib.h>
#define VECTOR_SIZE 8
typedef struct {
    float* data;
    int size;
} Vector;

Vector vector_add(Vector a, Vector b) {
    Vector result;
    result.size = a.size;
    result.data = (float*)malloc(result.size * sizeof(float));
    for (int i = 0; i < a.size; i++) {
        result.data[i] = a.data[i] + b.data[i];
    }
    return result;
}
void print_vector(Vector v) {
    printf("Vector: ");
    for (int i = 0; i < v.size; i++) {
        printf("%f ", v.data[i]);
    }
    printf("\n");
}
int main() {
    Vector vector_a, vector_b, vector_result;
    vector_a.size = VECTOR_SIZE;
```

```
    vector_b.size = VECTOR_SIZE;
    vector_a.data = (float*)malloc(VECTOR_SIZE * sizeof(float));
    vector_b.data = (float*)malloc(VECTOR_SIZE * sizeof(float));

    // Initialize vectors with some data
    for (int i = 0; i < VECTOR_SIZE; i++) {
        vector_a.data[i] = i + 1.0;
        vector_b.data[i] = VECTOR_SIZE - i;
    }
    printf("Vector A:\n");
    print_vector(vector_a);
    printf("Vector B:\n");
    print_vector(vector_b);
    vector_result = vector_add(vector_a, vector_b);
    printf("Vector Sum:\n");
    print_vector(vector_result);

    // Free allocated memory
    free(vector_a.data);
    free(vector_b.data);
    free(vector_result.data);
    return 0;
}
```

Output

```
Vector A:
Vector: 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
Vector B:
Vector: 8.000000 7.000000 6.000000 5.000000 4.000000 3.000000 2.000000 1.000000
Vector Sum:
Vector: 9.000000 9.000000 9.000000 9.000000 9.000000 9.000000 9.000000 9.000000

--------------------------------
Process exited after 0.1653 seconds with return value 0
Press any key to continue . . .
```

Result

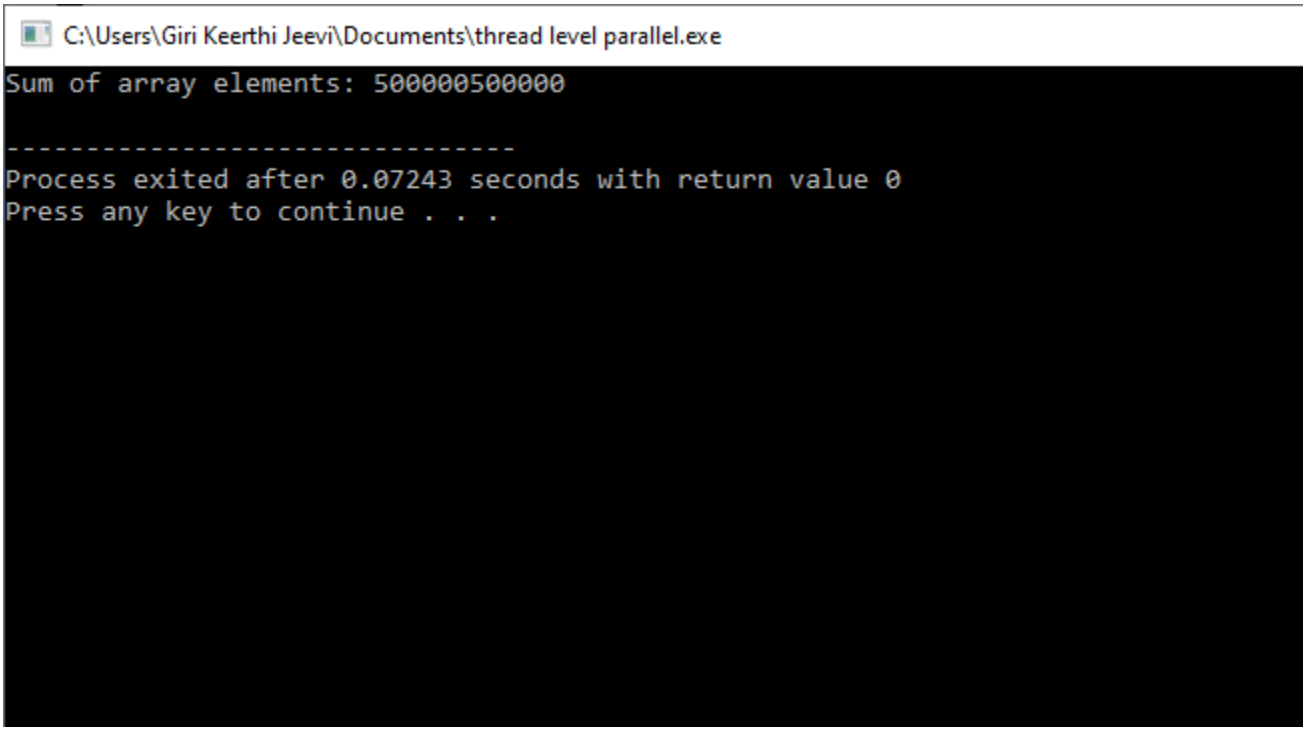
Thus the above program has been successfully executed and output are verified.

Source code

```
/*Thread level parallel*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define ARRAY_SIZE 1000000
#define NUM_THREADS 4
typedef struct {
    int* array;
    int start;
    int end;
    long long sum;
} ThreadData;

void* computeSum(void* arg) {
    ThreadData* threadData = (ThreadData*)arg;
    long long localSum = 0;
    for (int i = threadData->start; i < threadData->end; i++) {
        localSum += threadData->array[i];
    }
    threadData->sum = localSum;
    pthread_exit(NULL);
}
int main() {
    int* array = (int*)malloc(ARRAY_SIZE * sizeof(int));

    // Initialize array with some data
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1;
    }
```

```
    pthread_t threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];

    int chunkSize = ARRAY_SIZE / NUM_THREADS;

    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].array = array;
        threadData[i].start = i * chunkSize;
        threadData[i].end = (i == NUM_THREADS - 1) ? ARRAY_SIZE : (i + 1) *
chunkSize;
        pthread_create(&threads[i], NULL, computeSum, (void*)&threadData[i]);
    }
    long long totalSum = 0;

    // Wait for all threads to finish and accumulate results
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
        totalSum += threadData[i].sum;
    }

    // Display the result
    printf("Sum of array elements: %lld\n", totalSum);

    // Free allocated memory
    free(array);
    return 0;
}
```

Output



Result

Thus the above program has been successfully executed and output are verified.