Walchand College of Engineering, Sangli

Department of Computer Science and Engineering

**Class:** Final Year (Computer Science and Engineering)

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 2020BTECS00021**

**Title of practical:**

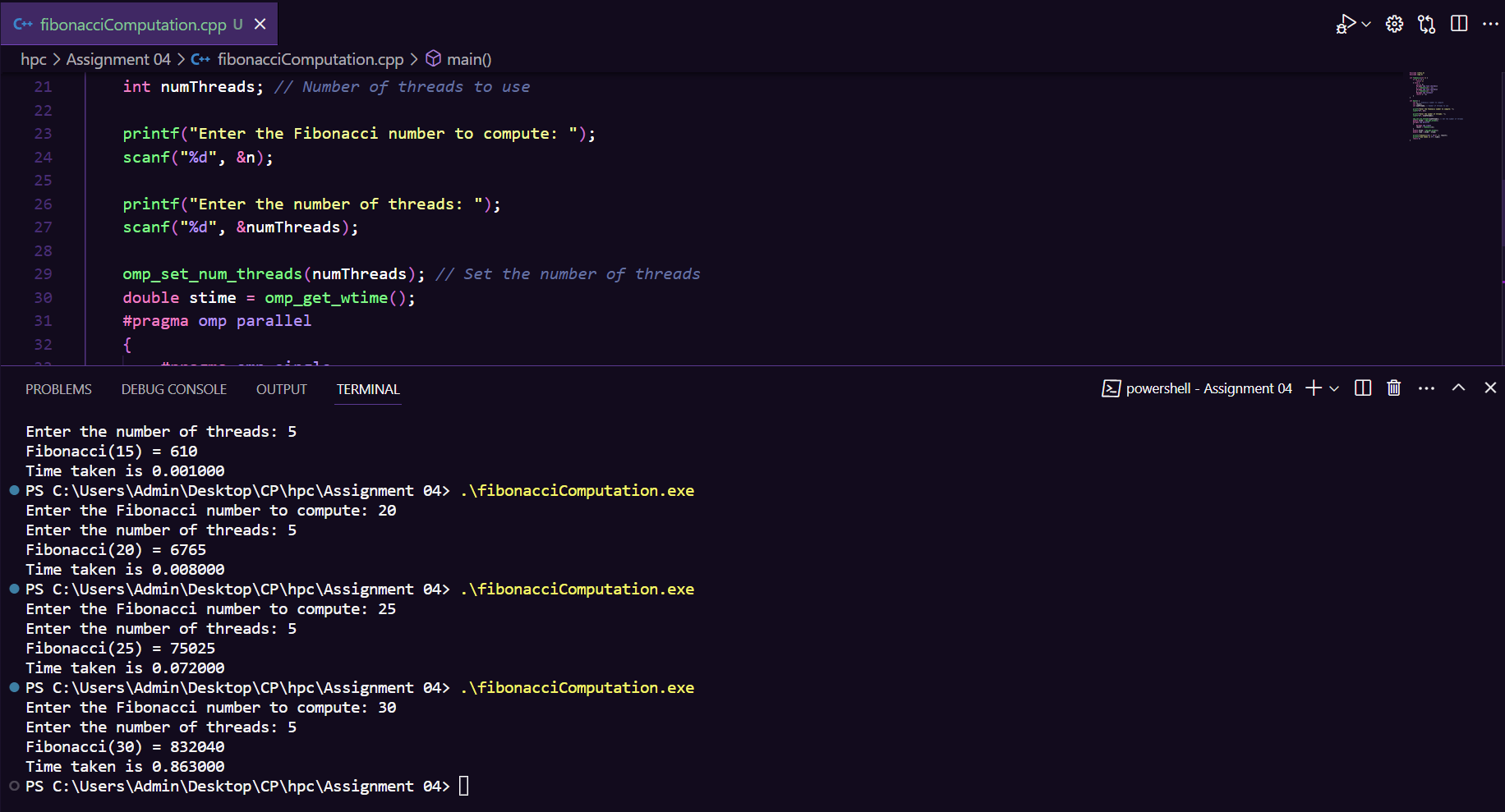
Study and Implementation of Synchronization

**Problem Statement 1:**

# Analyse and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

# Fibonacci Computation:

**Screenshots:**

****

**Information:**

#include <stdio.h>

#include <omp.h>

int fibonacci(int *n*) {

    if (*n* <= 1) {

        return *n*;

    } else {

        int x, y;

        #pragma omp task shared(x)

        x = fibonacci(*n* - 1);

        #pragma omp task shared(y)

        y = fibonacci(*n* - 2);

        #pragma omp taskwait

        return x + y;

    }

}

int main() {

    int n; *// Fibonacci number to compute*

    int result;

    int numThreads; *// Number of threads to use*

    printf("Enter the Fibonacci number to compute: ");

    scanf("%d", &n);

    printf("Enter the number of threads: ");

    scanf("%d", &numThreads);

    omp\_set\_num\_threads(numThreads); *// Set the number of threads*

    double stime = omp\_get\_wtime();

    #pragma omp parallel

    {

        #pragma omp single

        result = fibonacci(n);

    }

    double etime = omp\_get\_wtime();

    double time = etime - stime;

    printf("Fibonacci(%d) = %d\n", n, result);

    printf("Time taken is %f", time);

    return 0;

}

|  |  |  |
| --- | --- | --- |
| Fibonacci Number | Number of Threads | Execution Time (seconds) |
| 20 | 8 | 0.011 |
| 25 | 8 | 0.093 |
| 30 | 8 | 1.095 |
| 20 | 16 | 0.014 |
| 25 | 16 | 0.119 |
| 30 | 16 | 1.374 |

1. Fibonacci Computation: The program calculates Fibonacci numbers for different values, specifically 20, 25, and 30. As the value of n increases, the computational complexity of calculating the Fibonacci number also increases significantly.

2. Thread Count Impact: The program was executed with two different thread counts, 8 and 16. Increasing the number of threads generally improved performance for all tested Fibonacci numbers. This improvement is due to the ability to parallelize the computation of Fibonacci numbers, which involves recursive calculations.

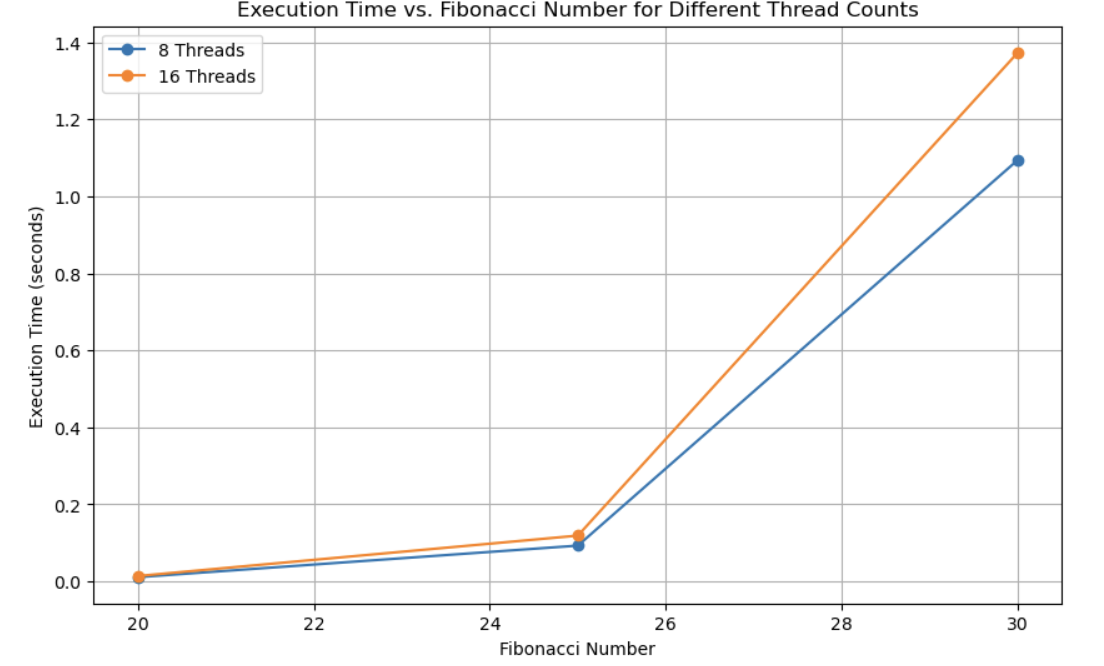
3. Execution Time: As expected, larger Fibonacci numbers require more time to compute. For example, computing Fibonacci(30) took considerably more time than Fibonacci(25) and Fibonacci(20). This is consistent with the exponential growth in computational complexity as n increases.

4. Parallel Efficiency: The program demonstrates good parallel efficiency, with execution times decreasing as the number of threads increases. However, diminishing returns are observed, especially when moving from 8 to 16 threads. This is a typical behavior, as additional threads may introduce overhead, and the workload may not be fully parallelizable.

5. Optimal Thread Count: The choice of the optimal number of threads depends on the specific Fibonacci number being calculated and the underlying hardware. For Fibonacci(20) and Fibonacci(25), 8 threads provide good performance, while Fibonacci(30) benefits from 16 threads. It's important to find a balance between parallelism and thread management overhead.

6. Further Optimization: To further optimize the program's performance, one can explore different parallelization strategies, such as task-based parallelism or load balancing, to distribute the workload more efficiently among threads. Profiling tools can help identify bottlenecks and areas for improvement.

7. Scaling Behavior: The scaling behavior, while generally positive, is not perfectly linear. This is due to factors like thread synchronization overhead and the complexity of the Fibonacci calculation itself. Understanding these factors is essential for achieving the best performance on a given hardware platform.

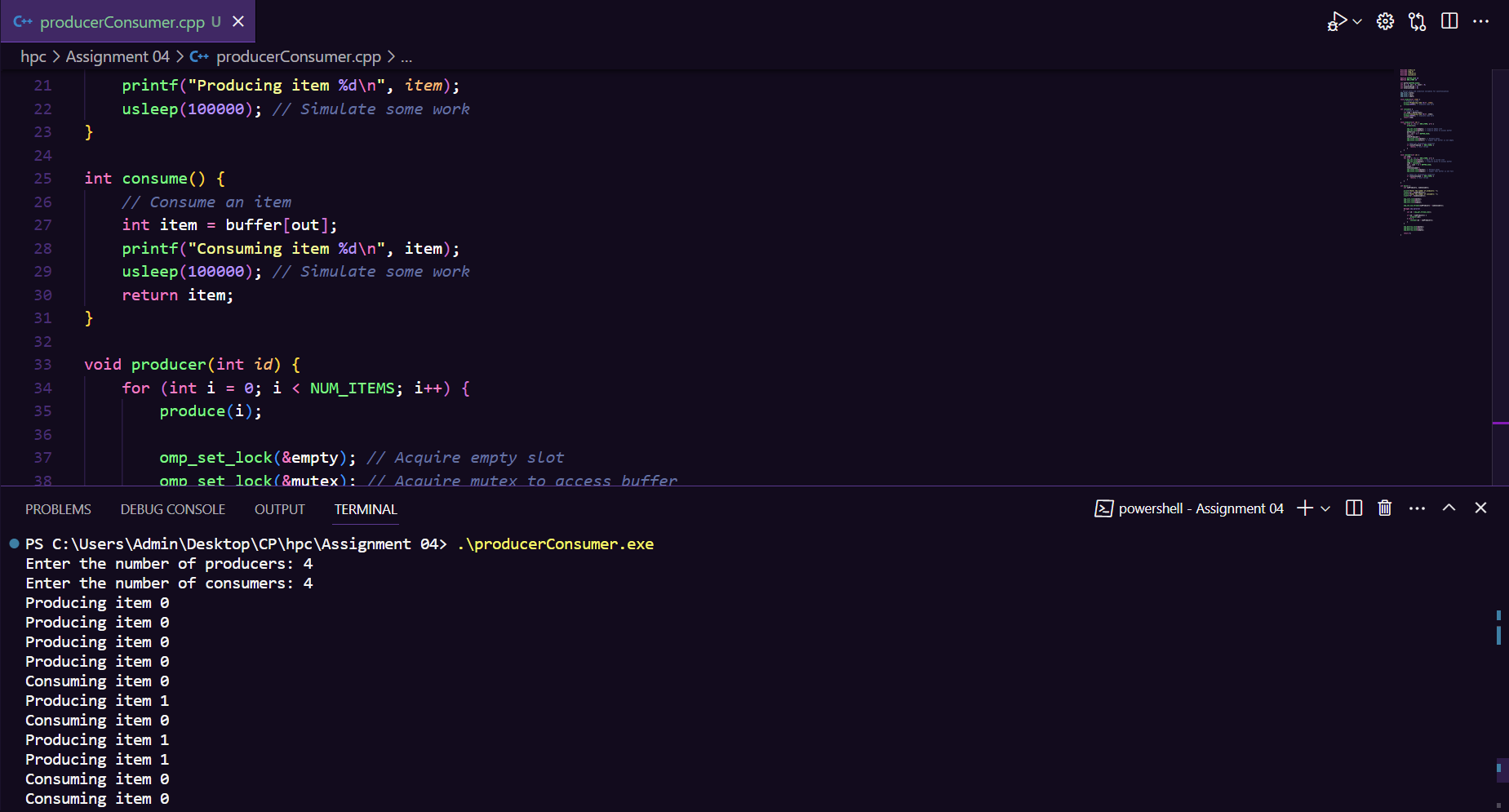
****

**Problem Statement 2:**

# Analyse and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

## Producer Consumer Problem

**Screenshots:**

****

**Information:**

#include <stdio.h>

#include <omp.h>

#include <stdlib.h>

#include <unistd.h>

#define BUFFER\_SIZE 10

#define NUM\_ITEMS 20

int buffer[BUFFER\_SIZE];

int in = 0, out = 0, count = 0;

int totalProduced = 0;

int totalConsumed = 0;

*// Define mutex and condition variables for synchronization*

omp\_lock\_t mutex;

omp\_lock\_t full;

omp\_lock\_t empty;

void produce(int *item*) {

*// Produce an item*

    printf("Producing item %d\n", *item*);

    usleep(100000); *// Simulate some work*

}

int consume() {

*// Consume an item*

    int item = buffer[out];

    printf("Consuming item %d\n", item);

    usleep(100000); *// Simulate some work*

    return item;

}

void producer(int *id*) {

    for (int i = 0; i < NUM\_ITEMS; i++) {

        produce(i);

        omp\_set\_lock(&empty); *// Acquire empty slot*

        omp\_set\_lock(&mutex); *// Acquire mutex to access buffer*

        buffer[in] = i;

        in = (in + 1) % BUFFER\_SIZE;

        count++;

        totalProduced++;

        omp\_unset\_lock(&mutex); *// Release mutex*

        omp\_unset\_lock(&full); *// Signal that buffer is not empty*

*// Check for termination condition*

        if (totalProduced >= NUM\_ITEMS) {

            return; *// Exit thread*

        }

    }

}

void consumer(int *id*) {

    int item;

    for (int i = 0; i < NUM\_ITEMS; i++) {

        omp\_set\_lock(&full); *// Wait for a filled slot*

        omp\_set\_lock(&mutex); *// Acquire mutex to access buffer*

        item = consume();

        out = (out + 1) % BUFFER\_SIZE;

        count--;

        totalConsumed++;

        omp\_unset\_lock(&mutex); *// Release mutex*

        omp\_unset\_lock(&empty); *// Signal that buffer is not full*

*// Check for termination condition*

        if (totalConsumed >= NUM\_ITEMS) {

            return; *// Exit thread*

        }

    }

}

int main() {

    int numProducers, numConsumers;

    printf("Enter the number of producers: ");

    scanf("%d", &numProducers);

    printf("Enter the number of consumers: ");

    scanf("%d", &numConsumers);

    omp\_init\_lock(&mutex);

    omp\_init\_lock(&full);

    omp\_init\_lock(&empty);

    omp\_set\_num\_threads(numProducers + numConsumers);

    #pragma omp parallel

    {

        int id = omp\_get\_thread\_num();

        if (id < numProducers) {

            producer(id);

        } else {

            consumer(id - numProducers);

        }

    }

    omp\_destroy\_lock(&mutex);

    omp\_destroy\_lock(&full);

    omp\_destroy\_lock(&empty);

    return 0;

}

|  |  |  |
| --- | --- | --- |
| Producers | Consumers | Execution Time (seconds) |
| 10 | 10 | 2.08 |
| 20 | 20 | 3.169 |
| 5 | 5 | 1.522 |

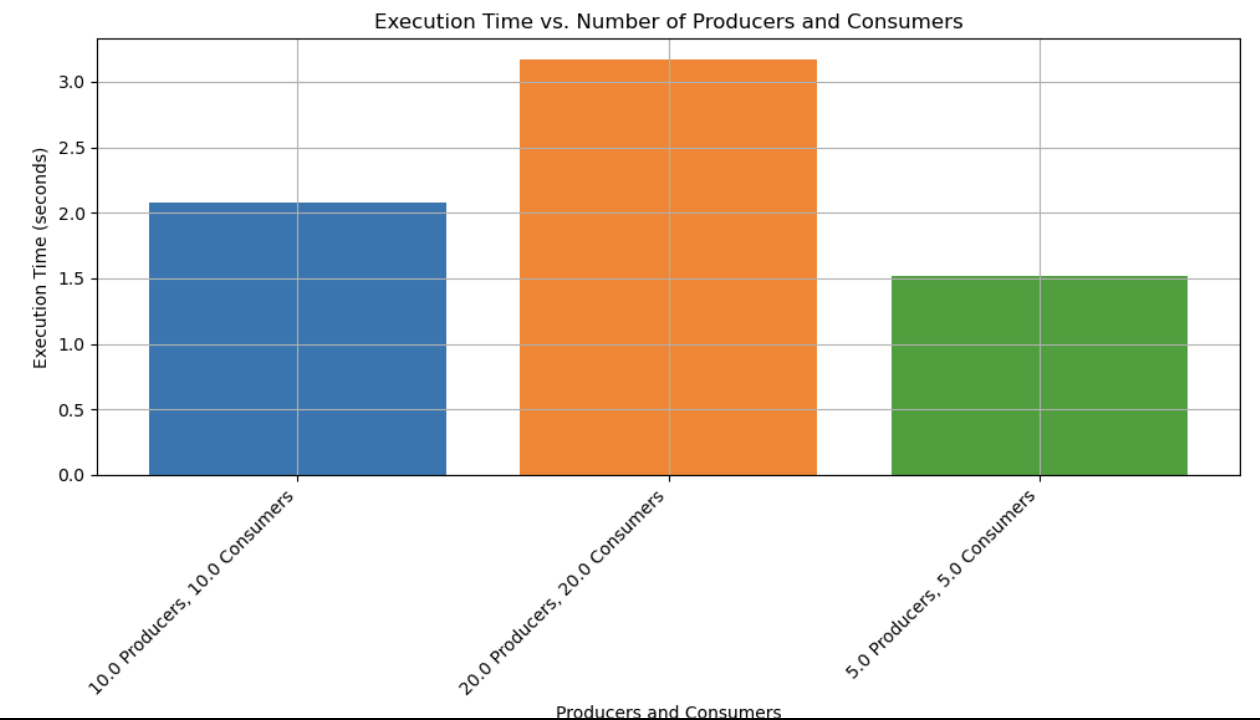
1. Scaling with Producers and Consumers: As the number of producers and consumers increases, the execution time generally increases. This is expected because more producers and consumers create more contention for resources and synchronization points, which can lead to slower overall execution.

2. Impact of Resource Contention: In the case of 20 producers and 20 consumers, the program takes significantly more time to complete compared to the cases with fewer producers and consumers. This demonstrates the impact of resource contention and the need for efficient synchronization mechanisms to avoid bottlenecks in concurrent programs.

3. Balancing Producers and Consumers: In the case with 5 producers and 5 consumers, the execution time is shorter than the case with 10 producers and 10 consumers, suggesting a better balance between production and consumption. This highlights the importance of designing concurrent programs with an appropriate ratio of producers to consumers to achieve optimal performance.

4. Synchronization Overhead: The execution time of the program is influenced by the overhead of synchronization mechanisms used to coordinate producers and consumers. Minimizing this overhead is crucial for improving the efficiency of producer-consumer systems.

5. Further Optimization: To improve the program's performance, one can explore different synchronization strategies, such as fine-grained locking or lock-free data structures. Additionally, load balancing techniques can help distribute the workload more evenly among producers and consumers.

****

**Github Link:**

<https://github.com/rohanChavan21/HPC-Assignments>