**20BCE1550**

**Samridh Anand Paatni**

**CSE4001 Lab 02**

**Open MP Programming**

**Q1. Create a hundred threads using:**

1. **Runtime Library Routines**

**Code:**

// using runtime library routines

#include<stdio.h>

#include<omp.h>

// compile using: `gcc filename -fopenmp`

int main(int argc, char \*argv[]) {

    int tid, numThreads;

    omp\_set\_num\_threads(100);

    # pragma omp parallel private (tid, numThreads)

    {

        tid = omp\_get\_thread\_num();

        printf("welcome to PDC %d\n", tid);

        if (tid == 0) {

            numThreads = omp\_get\_num\_threads();

            printf("%d threads have been created\n", numThreads);

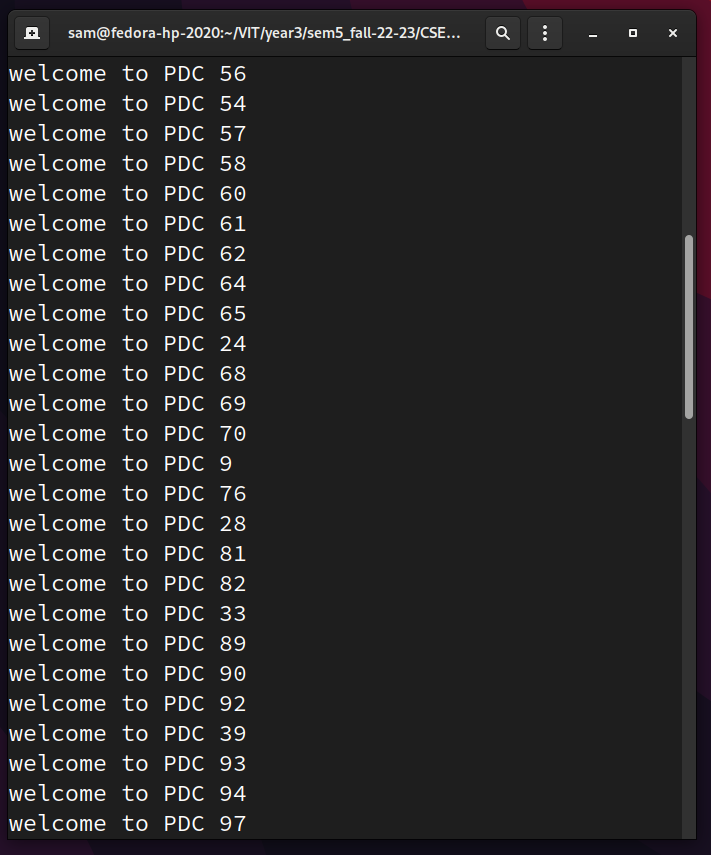
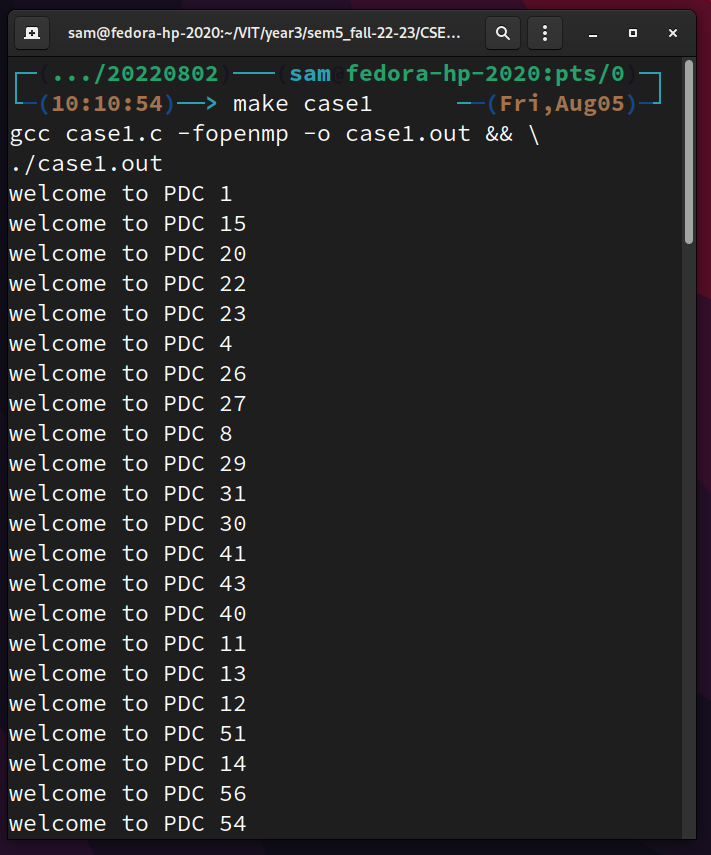
        }

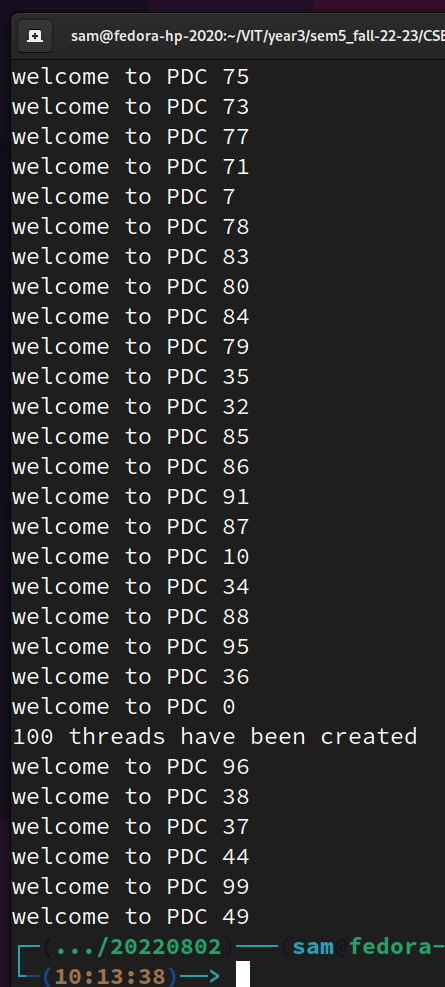
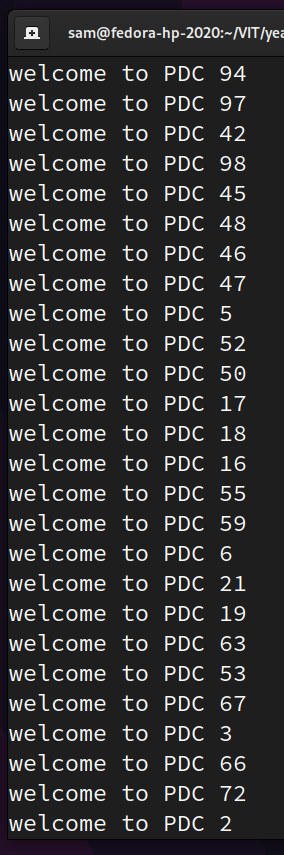
    }

    return 0;

}

**Output:**

****

****

1. **Compiler Directives**

**Code:**

// using compiler directives

#include<stdio.h>

#include<omp.h>

// compile using: `gcc filename -fopenmp`

int main(int argc, char \*argv[]) {

    int tid, numThreads;

    # pragma omp parallel private (tid, numThreads) num\_threads(100)

    {

        tid = omp\_get\_thread\_num();

        printf("welcome to PDC %d\n", tid);

        if (tid == 0) {

            numThreads = omp\_get\_num\_threads();

            printf("%d threads have been created\n", numThreads);

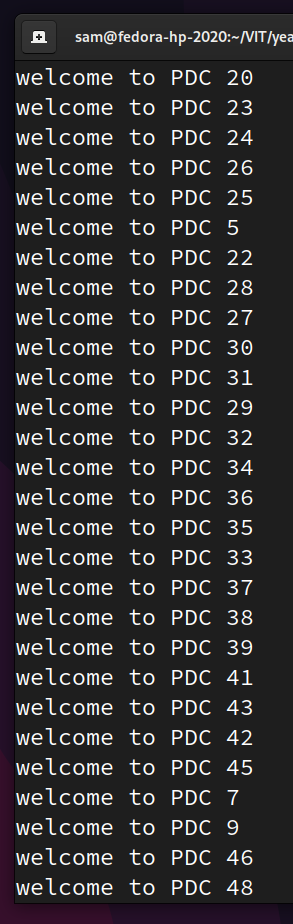
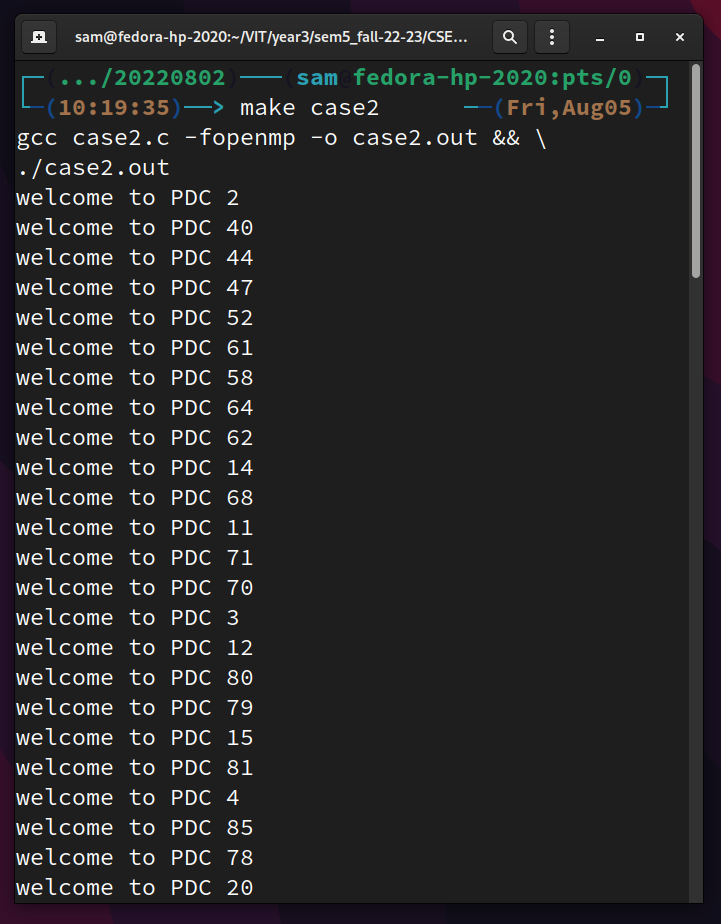
        }

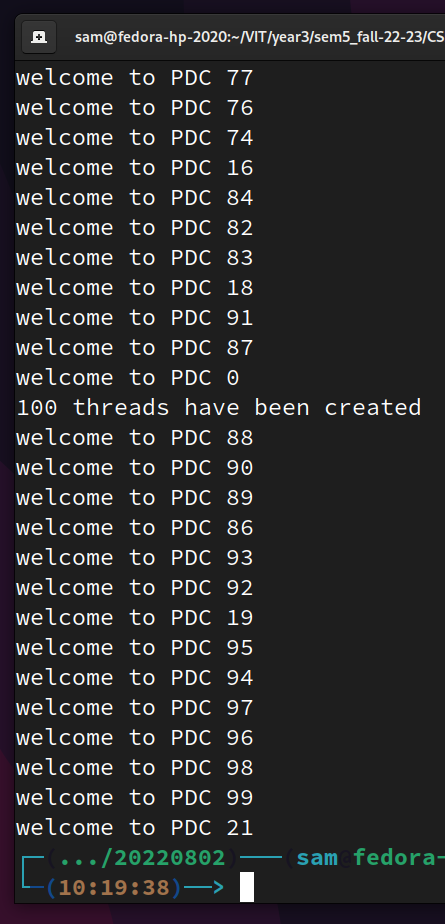
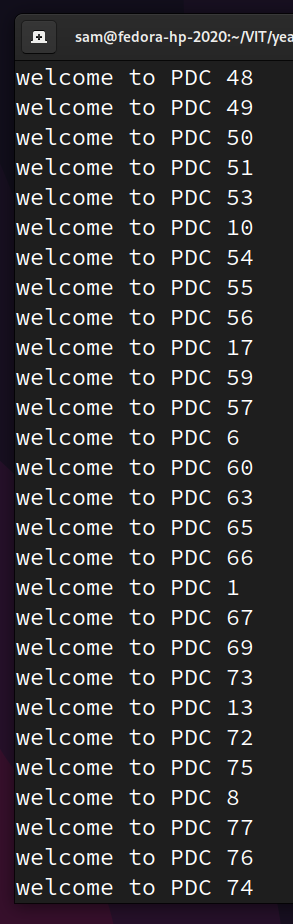
    }

    return 0;

}

**Output:**

****

****

1. **Environment Variables**

**Code:**

// using environment variables

#include<stdio.h>

#include<omp.h>

// compile using: `gcc filename -fopenmp`

// before running, give the command: `export OMP\_NUM\_THREADS=100` in bash

int main(int argc, char \*argv[]) {

    int tid, numThreads;

    # pragma omp parallel private (tid, numThreads)

    {

        tid = omp\_get\_thread\_num();

        printf("welcome to PDC %d\n", tid);

        if (tid == 0) {

            numThreads = omp\_get\_num\_threads();

            printf("%d threads have been created\n", numThreads);

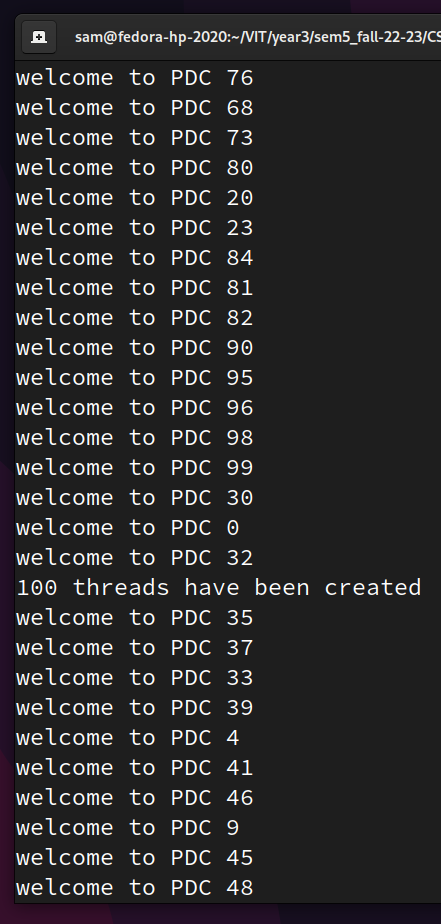
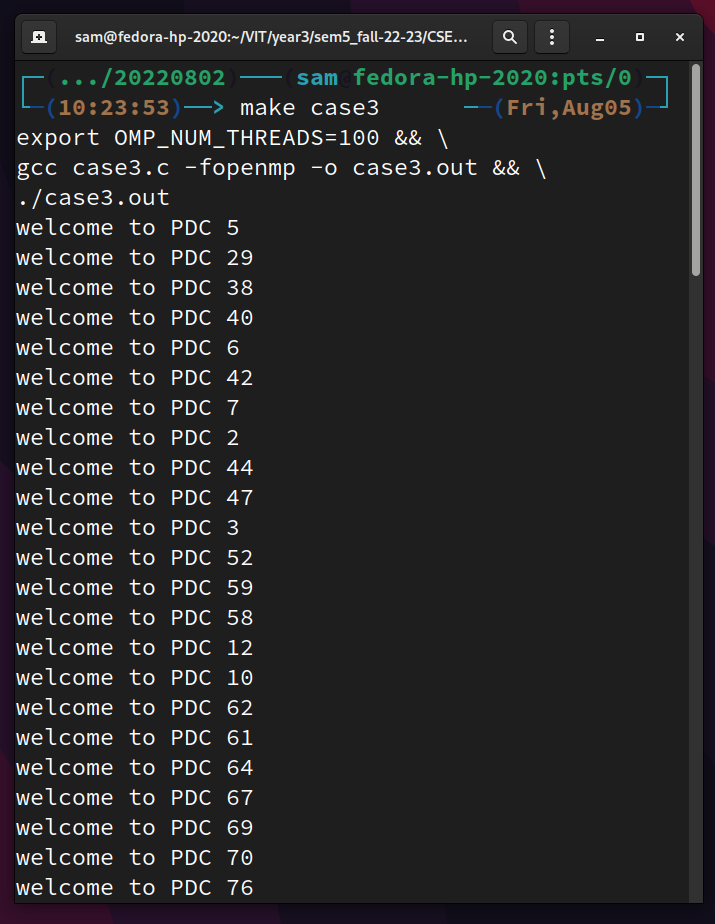
        }

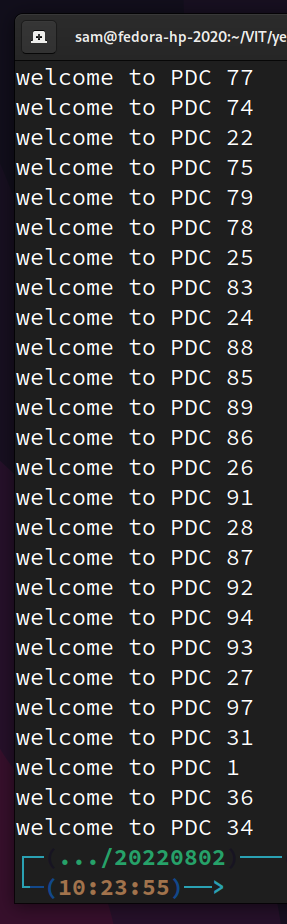
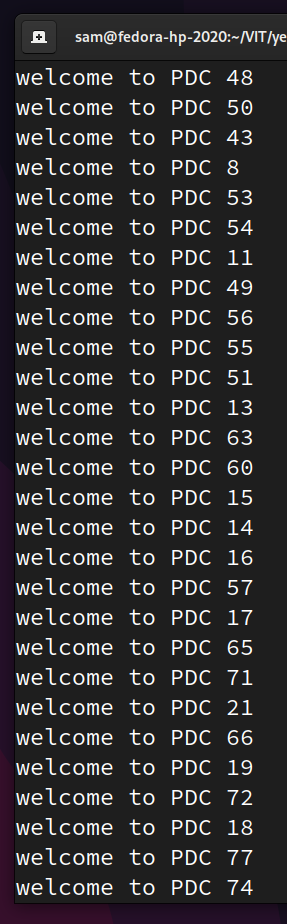
    }

    return 0;

}

**Output:**

****

****

**Q2. Implement vector addition in serial and parallel and compare the results. Do the parallel computation using a 1000 threads.**

**Code:**

**Serial addition:**

#include <stdio.h>

#include <time.h>

#define VECTOR\_SIZE 100000

int main() {

    // make the vectors

    int a[VECTOR\_SIZE], b[VECTOR\_SIZE], c[VECTOR\_SIZE];

    for (int i = 0; i < VECTOR\_SIZE; i++) {

        a[i] = VECTOR\_SIZE - i;

        b[i] = i;

    }

    // serially add the vectors

    clock\_t tSerial = clock();

    for (int i = 0; i < VECTOR\_SIZE; i++) {

        c[i] = a[i] + b[i];

    }

    tSerial = clock() - tSerial;

    // show the output

    printf(

        "Serial addition took %f seconds\n",

        ((double) tSerial)/CLOCKS\_PER\_SEC

    );

    return 0;

}

**Parallel addition:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define VECTOR\_SIZE 100000

int main(int argc, char \* argv[]) {

    // make the vectors

    int a[VECTOR\_SIZE], b[VECTOR\_SIZE], c[VECTOR\_SIZE];

    for (int i = 0; i < VECTOR\_SIZE; i++) {

        a[i] = VECTOR\_SIZE - i;

        b[i] = i;

    }

    int nThreads = atoi(argv[1]);

    // paralelly add the vectors:

    clock\_t tPar = clock();

    // the part of the vector one thread will access

    int slice\_size = VECTOR\_SIZE / nThreads;

    int slice\_start, slice\_end;

    int tid;

    // make threads

    omp\_set\_num\_threads(nThreads);

    #pragma omp parallel private (tid, slice\_start, slice\_end)

    {

        // allot a slice to the particular thread

        tid = omp\_get\_thread\_num();

        slice\_start = tid \* slice\_size;

        slice\_end = slice\_start + slice\_size;

        // perform addition for the elements in the allotted slice\_size

        for (int i = slice\_start; i < slice\_end; i++) {

            c[i] = a[i] + b[i];

        }

    }

    tPar = clock() - tPar;

    // show the output

    printf(

        "Parallel addition took %f seconds with %d threads\n",

        ((double) tPar)/CLOCKS\_PER\_SEC,

        nThreads

    );

    return 0;

}

**Output:**

Running the computation with 4 threads is the fastest (because I have a quad-core laptop). The parallel computation with a 100 more threads is slower than the serial execution.

