


## Honesty Guaranty

I know the examination rules, promise to be honest and abide by the rules. Signature:



### Examination of Northwestern Polytechnical University (A)

2022 — 2023 School Year 1 Semester

School of Computer Science, Course: Parallel Programming, Class Hours: 48  
Exam. Date: 2022.11.28, Exam. Duration: 2 Hours, Written Exam. (Open-book)

Item	I	II	III	IV	V	Total Score
Score						

Class		Student ID.	2019380141	Name	ABID ALI
-------	--	-------------	------------	------	----------

#### I. Provide a very brief definition of the following terms (20 points, 4 points per item)

1. Flynn's taxonomy of computers

**Solution:**

M.J. Flynn created a classification for computer system structure based on the number of instructions and data items that are altered concurrently.

An instruction stream is a collection of instructions read from memory. The activities performed on the data in the processor result in the creation of a data stream. Parallel processing can take place in the instruction or data streams, or both.

According to Flynn's, computer architecture is classified into four types.

1) SISD (Single Instruction Single Data Stream):

A SISD computer system is a single-processor machine that can execute a single instruction on a single data stream. Most conventional computers use SISD architecture, which requires all instructions and data to be stored in main memory.

2) SIMD (Single Instruction Multiple Data Stream):

A SIMD system is a multiprocessor machine that can execute the same instruction on all of its CPUs while working on distinct data streams.

3) MISD (Multiple Instruction Single Data stream):

MISD computing is a multiprocessor system that can execute different instructions on various processing components while still working on the same data set.

#### 4) MIMD (Multiple Instruction Multiple Data Stream)

A MIMD system is a multiprocessor computer that can execute multiple instructions simultaneously across various data streams. Each processing element has its own command and data streams.

#### 2. Cache coherence

##### **Solution:**

For enhanced performance, each processor in a multiprocessor system will frequently have its own cache. Cache coherence refers to the problem of keeping the data in these caches coherent. The main problem is dealing with processor writes.

In a multiprocessor system, data inconsistency can occur between neighboring levels or within the same level of the memory hierarchy. Conflicting copies of the same item, for example, may exist in the cache and main memory.

Cache coherence is an issue since many processors run in parallel and many caches may have different copies of the same memory block. Cache coherence methods help to avoid this issue by making each cached block of data consistent.

#### 3. thread safety

##### **Solution:**

Although multithreading is an advantageous feature, it comes at a cost. Thread-safe implementations are required in multithreaded situations. This means that many threads can use the same resources without disclosing incorrect behavior or causing unexpected outcomes. Thus, giving rise to a programming approach which is known as "thread-safety."

There is just one control flow in single-threaded processes. As a result, the code that these processes run is not needed to be thread-safe.

As a result, the code that these processes run is not needed to be thread-safe. However, several control flows in multithreaded systems may access the same functions and resources at the same time. Multithreaded application code must be reentrant and thread-safe to maintain resource integrity. A thread-safe function prevents lock contention on shared resources. Thread safety only pertains to a function's implementation and has no influence on its outer interface.

#### 4. One sided communication

##### **Solution:**

One-sided communication:

Historically, shared-memory systems allowed one process to access the main memory of another process (Direct Memory Access - DMA) - Today, some networking devices and graphics cards have direct access to the main memory, bypassing the CPU.

#### 5. PGAS

Partitioned Global Address Space (PGAS) is a parallel programming paradigm that aims for high performance while enhancing programmer efficiency. While a globally shared address space improves efficiency, a distinction between local and remote data accesses is required to provide performance gains and ensure scalability on large-scale parallel systems. PGAS preserves global address space while recognizing non-uniform communication costs.

**II. Circle TRUE or FALSE based on the statements. (12 points, 2 points per item)**

1. Currently, the main architecture of parallel computers is cluster ... **FALSE**
2. MPI programs can only be SPMD ... **FALSE**
3. POSIX threads (Pthreads) are using compiler directives ... **TRUE**
4. Hybrid model of MPI and CUDA can be used for GPU clusters ... **TRUE**
5. Communication overhead may be a major performance challenge ... **TRUE**
6. Amdahl's law is optimistic for the scalability of parallel programs ... **FALSE**

**III. Choose the right answer. (24 points, 4 points per item)**

1. Suppose the parallel fraction of a program is 0.8, according to Gustafson's law, what is the speedup if we use 100 processors to run this parallel program? ( **B** )  
A. 1.25 B. 5 C. 80.2 D. 100
2. Which of the following about OpenMP is incorrect? ( **C** )  
A. OpenMP is an API that enables explicit multi-threaded parallelism.  
B. The primary components of OpenMP are compiler directives, runtime library, and environment variables.  
C. OpenMP is designed for distributed memory parallel systems and guarantees efficient use of memory.  
D. OpenMP supports UMA and NUMA architectures.
3. Which operation ( **B** ) does not belong to collective communication?  
A. MPI\_Scatter  
B. MPI\_Wait  
C. MPI\_Barrier  
D. MPI\_Reduce
4. Which operation ( **D** ) belongs to non-blocking communication of MPI?  
A. MPI\_Ssend  
B. MPI\_Bsend  
C. MPI\_Rsend  
D. MPI\_Isend
5. What is the final output about  $n$  after running? ( **D** )

```

int n=10, i;
omp_set_num_threads(2);
#pragma omp parallel for firstprivate(n)
for (i=1; i<4; i++)          n=n+2*i;
printf("n= %d\n", n);

```

A. n=10 B. n=11 C. n=19 D. n=26

6. We need to use each thread to calculate one output element of a vector addition. Assume that variable *i* should be the index for the element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index? (C)

- A.  $i = \text{threadIdx.x} + \text{threadIdx.y};$
- B.  $i = \text{blockIdx.x} + \text{threadIdx.x};$
- C.  $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$
- D.  $i = \text{blockIdx.x} * \text{threadIdx.x};$

#### IV. Answer the questions briefly (15 points, 5 points per item)

1. What are the main parallel overheads?

**Solution:**

##### Main parallel overheads:

- a) Inter-process Interaction: Any nontrivial parallel system necessitates interaction and data transmission across its processing components (e.g., intermediate results). The most significant source of parallel processing overhead is often the time spent transmitting data between processing components.
- b) Idling: Many variables, including load imbalance, synchronization, and the presence of serial components in a program, can cause processing nodes in a parallel system to become idle.
- c) Excess Calculation: The difference in computation done by the parallel program and the best serial program is the extra computation overhead generated by the parallel program.

2. What is the False sharing? And how do you solve this problem?

**Solution:**

False sharing happens when multiple processors update different parts of the same cache line. As a consequence, even if no modifications are made to shared variables, the system is ignorant.

In an invalidate protocol, when a processor alters its piece of a cache-line, the other copies of that line are invalidated. The line must be obtained from the remote processor when other processors seek to update their cache-line segments. False-sharing can easily result in a cache-line being ping ponged between many CPUs. Because all reads may be done locally, but all writes must be updated, this case is somewhat better with an update mechanism. This avoids the waste of an invalidate operation.

To avoid false sharing, it is to be made sure that data requested by separate threads is given to different cache lines.

Moreover, to reduce erroneous sharing, the following measures can be employed in general: Use as much private or thread private data as possible. Use the compiler's optimization tools to decrease memory loads and stores. Pad data structures such that each thread's data is stored on a separate cache line.

3. What are the advantages and limitations of hybrid MPI and OpenMP programming model?

**Solution:**

Advantages:

- a) A hybrid paradigm is excellent for the most common hardware environment of clustered multi/many-core machines: In actuality, a hybrid programming style is more suited to the architecture of today's supercomputers, which contain networked shared-memory nodes, NUMA (Non-uniform memory access) machines, and other features.
- b) Greater scalability, which may be achieved by lowering the amount of MPI messages as well as the number of processes participating in collective communications. Furthermore, using OpenMP inside poorly balanced (workflow) processes might improve load balancing and hence boost scalability.
- c) Raising the code's granularity to boost scalability: A parallel MPI code is a sequence of computation and communication stages. The granularity of a code is defined as the average of two successive computation and communication stages. The more comprehensive the code, the more scalable it becomes. The hybrid approach significantly improves the granularity of a code, enhancing its scalability.
- d) The total consumed memory can be reduced by employing the OpenMP shared memory approach, which eliminates duplicated data in MPI processes and, as a result, the MPI library itself requires less memory.
- e) Can go beyond the algorithmic constraints of the MPI technique, such as the maximum decomposition in one direction.
- f) Reducing the number of MPI processes can also increase the performance of some algorithms: In linear algebra algorithms that employ iterative resolution methods, a smaller number of domains leads to better preconditioning (smaller condition number)
- g) Fewer concurrent I/O requests and a larger average record size
- h) Fewer and more correctly sized searches lessen burden on meta-data servers, potentially saving significant time on a massively parallel application.

Disadvantages:

- a) Implementing a hybrid MPI/OpenMP model is a difficult effort that requires a high level of expertise.
- b) Total performance improvements over a single MPI or OpenMP solution are not assured, because deploying a hybrid method will almost certainly involve additional costs to the current code.

## V. Programming (29 points)

1. The following C++ program calculates the value of Pi. Please complete the missing lines of code. ( 9 points)

```
#include <iostream>
#include "mpi.h"
#include <ctime>
#include <cmath>
using namespace std;

const int N = 1000000;
double start, finish;
int main (int argc, char *argv[]) {
    int numprocs, myid;
    /* Please complete the missing lines of code using MPI Routines here.*/
    MPI_Init(&argc, &argv); // Initialization of the MPI library
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // Get the number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); /*Get the current process
    identification number 0, 1, 2, 3, ..., numprocs – 1 */
    start = MPI_Wtime();
    double partSum=0.0; double Pi = 0.0;
    for (int i = myid; i < N; i += numprocs) { partSum += sqrt (1 –
        (double(i) / N) * (double (i) / N)) / N;
    }
    /* Please complete the missing line of code using a MPI Routine here.*/
    MPI_Reduce(&partSum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    cout << "Myid" << myid << ", partSum:" << partSum << endl;
    if (myid == 0) {
        Pi *= 4.0;
        finish = MPI_Wtime();
        cout << "The value of Pi is : " << Pi << endl;
        cout << "#" << numprocs << "run time : " << finish – start << endl;
    }
    /* Please complete the missing line of code using a MPI Routine here.*/
    MPI_Finalize();
    //system("pause");
    return 0;
}
```

2. (6 points) Consider the problem of counting the array a. the final summation of a is stored in variable sum. Please read the following code and answer questions.

```

#include <stdio.h>
#include <omp.h>
#define N 10
int
main(void) {
    int i, sum=0, a[N];
    for(i=0;i<N;i++)
        a[i] = i;
    #pragma omp parallel for shared(a,sum) private(i)
    for(i=0;i<N;i++) sum += a[i];
    printf("sum = %d\n",sum); return
    0;
}

```

Questions:

- 1) Do you think the program can get the correct result? If not, can you explain it? **(3 points)**

Yes, it's correct we get result 45

- 2) Propose a solution to overcome the problem in the above example. **(3 points)**

I compiled and got 45. So, I think there is no need to change the code

3. Consider the following OpenMP program and answer questions. **(8 points)**

```

#include <stdio.h>
#include <omp.h>
#define COLUMNS 3
#define ROWS 3
int m[ROWS][COLUMNS];
void fillColumn(int j){
    int i;
    #pragma omp for
    for (i = 0; i < ROWS; i++) m[i][j]
        = omp_get_thread_num();
}
int
main(){
    int i, j;
    for (i = 0; i < ROWS; i++) // initialize the array
        for (j= 0; j < COLUMNS; j++) m[i][j] = -1;

    #pragma omp parallel num_threads(COLUMNS-1)
    fillColumn(0);
    #pragma omp parallel for num_threads(COLUMNS-1)
    for (j = 1; j < COLUMNS; j++) fillColumn(j);
    for (i = 0; i < ROWS; i++) // print out the results
    {
        for (j= 0; j < COLUMNS; j++)
            printf(" %2d ", m[i][j]);
        printf("\n");
    }
    return 0;
}

```

Questions:

- 1) Supposing that you use gcc compiler, what the compiler command is to get an executable openmp program? (2 points)

Let our openmp program name is main.c

First I will create a Makefile in the same folder of main.c file. And specify the all and executable file(exec) inside the Makefile like this:

all:

gcc -o hello-world -fopenmp hello-world.c

where,

gcc - is the compiler

o - is the parameter used to create executable

main.c is the program name

fopenmp - is the parameter to compile and link openmp library

exec:

./main

Then I will open the folder using any suitable ide and open the terminal:

In the terminal I will run gradually:



make run

make execute

Now we have got our executable file named main

- 2) What is the output of this program using openmp? (6 points)

Output of this program using openmp is:

0 0 -1

0 0 -1

1 -1 1

4. Consider the following program. Explain how you would parallelize the following segment of C code. Modify the code and insert the necessary OpenMP pragma(s) into your code. (6 points)

```
#define X 50
#define Y 50
int i, j;
double a[X,Y]; double
m;
for (i=1; i< X; i++) for
    (j= 0; j<Y; j++) { m
        = i*j;
        a[i,j]=tmp*a[i-
1,j]; }
```

Solution

```
#include <stdio.h>
#include <omp.h>
#define X 50
#define Y 50
int main()
{
    int i, j, tmp=1;
    double a[X][Y];
    double m;
    #pragma omp parallel for
    for (i = 1; i < X; i++)
```

```

    for (j = 0; j < Y; j++)
    {
        m = i + j;
        a[i] [j] = tmp * a[i - 1][j];
    }
for (i = 1; i < X; i++)
    for (j = 0; j < Y; j++)
    {
        printf(" %2f ", a[i][j]);
    }
printf("\n");
return 0;
}

```