Honesty Guaranty

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

Examination of Northwestern Polytechnical University (A)

2021 — 2022 School Year 1 Semester

School of Computer Science, Course: Parallel Programming , Class Hours: 48
Exam. Date: 2021.12.26 , Exam. Duration: 2 Hours, Written Exam. (Open-book)

Item	I	II	III	IV	V	Total Score
Score						

Class		Student ID.	2018380130	Name	Khan Md Shahedu Islam

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

1. Gustafson's Law

Solution:

Gustafson's objection is that massively parallel machines allow computations previously unfeasible since they enable computations on very large data sets in fixed amount of time. In other words, a parallel platform does more than speeding up the execution of a code: it enables dealing with larger problems.

Suppose I have an application taking a time t_s to be executed on N processing units. Of that computing time, a fraction $(1-f)$ must be run sequentially. Accordingly, this application would run on a fully sequential machine in a time t equal to

$$t = (1-f)t_s + Nft_s$$

If we increase the problem size, we can increase the number of processing units to keep the fraction of time the code is executed in parallel equal to $f \cdot t_s$. In this case, the sequential execution time increases with N which now becomes a measure of the problem size. The speedup then becomes

$$S = \frac{(1-f)t_s + Nft_s}{t_s} = (1-f) + Nf$$

The efficiency would then be

$$e = \frac{S}{N} = \frac{1-f}{N} + f$$

so that the efficiency tends to f for increasing N. The pitfall of these rather optimistic speedup and efficiency evaluations is related to the fact that, as the problem size increases, communication costs will increase, but increases in communication costs are not accounted for by Gustafson's law.

That's the brief definition on Gustafson's law.

2. CCNUMA

Solution:

CCNUMA stands for Cache-coherent non-uniform memory access machines. A CCNUMA machine includes several processing nodes linked through a high-bandwidth low-latency interconnection network. Each processing node includes a high-implementation processor, the related cache, and an allocation of the global shared memory.

Cache coherence is preserved by a directory-based, write-invalidate cache coherence protocol. It can maintain all caches consistent; every processing node has a directory memory corresponding to its allocation of the shared physical memory.

For each memory line, the directory memory saves recognize remote nodes caching that line. Thus, utilizing the directory, it is applicable for a node writing a location to send point-to-point messages to invalidate private copies of the equivalent cache line.

There is another essential attribute of the directory-based protocol is that it does not base on any definite interconnection network topology. Thus, some scalable networks, including a mesh, a hypercube, or a multi-stage network, cart be used to link the processing nodes.

All the CCNUMA machines share the common goal of building a scalable shared-memory multiprocessor. The main difference among them is in the way the memory and cache coherence mechanisms are distributed among the processing nodes.

3. BSP

Solution:

The bulk synchronous parallel (BSP) abstract computer is a bridging model for designing parallel algorithms. It is similar to the parallel random-access machine (PRAM) model, but unlike PRAM, BSP does not take communication and synchronization for granted. In fact, quantifying the requisite synchronization and communication is an important part of analyzing

a BSP algorithm.
In other words, Bulk-synchronous parallelism is a type of course-grain parallelism, where inter-processor communication follows the discipline of strict barrier synchronization. Depending on the context, BSP can be regarded as a computation model for the design and analysis of parallel algorithms, or a programming model for the development of parallel software.

4. CUDA

Solution:

CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing unit (GPU) for general purpose processing – an approach called general-purpose computing on GPUs (GPGPU). CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

A modern CPU consists of several cores that are each capable of executing a single thread. Each of these threads can contain their own sequence of unique instructions. Each core executes a single thread
CUDA Architecture utilizes a different approach where a collection of "streaming multiprocessors" (SM) execute the same set of instructions, including branch conditions on multiple threads on different regions of data. Nvidia has coined the term SIMT, single instruction, multiple threads, to describe this architecture.

5. Cartesian topology

Solution:

MPI topology mechanism is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

A Cartesian grid is a structure, typically in 2~or~3 dimensions, of points that have two neighbors in each of the dimensions. Thus, if a Cartesian grid has sizes K×M×N, its points have coordinates (k,m,n) with $0\leq kwraparound connections is called a periodic grid .

The most common use of Cartesian coordinates is to find the rank of process by referring to it in grid terms. For instance, one could ask `what are my neighbors offset by (1,0,0), (−1,0,0), (0,1,0) et cetera'.

While the Cartesian topology interface is fairly easy to use, as opposed to the more complicated general graph topology below, it is not actually sufficient for all Cartesian graph uses. Notably, in a so-called star stencil , such as the nine-point stencil , there are diagonal connections, which

can not be described in a single step. Instead, it is necessary to take a separate step along each coordinate dimension.

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

1. Functional decomposition is a form of task parallelism ... TRUE
2. MPI programs cannot run on shared-memory systems …. FALSE
3. POSIX threads (Pthreads) programming model is library based …TRUE
4. The current No. 1 Supercomputer in the world is IBM Summit …FALSE
5. Too many barriers may be a major performance challenge … FALSE
6. Performance analysis tool TAU can analyze binary codes ... TRUE

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

1. Suppose the parallel fraction of a serial program is only 0.2, according to Amdahl's law, what is the limit of speedup? (D)
   A. 2.5 B. 4 C. 5 D. 1.25

2. In order to remove the unnecessary synchronization barrier, what statement should we use correctly? (D)
   A. #pragma omp for
   B. #pragma omp single
   C. #pragma omp sections
   D. #pragma omp for nowait

3. Which operation (C) does not belong to point-to-point communication?
   A. MPI_Send
   B. MPI_Bsend
   C. MPI_Barrier
   D. MPI_Irecv

4. Which type of operation (D) does not belong to collective opeartions of MPI?
   A. data movement
   B. synchornization
   C. collective computation
   D. communicator management

5. What are the final results after running? (D)

```
int x=5, i;
omp_set_num_threads(5);
#pragma omp parallel for firstprivate(x)
for (i=1; i<5; i++)
   x=x+2*i;
```

A. x=5 B. x=7 C. x=11 D. x=13

6. There are 12 iterations (index from 0 to 11) of for loop in OpenMP program. Suppose we use 3 threads to share the loop construct, and always assign the iteration index with the following pattern: Thread 0: 0, 1, 6, 7; Thread 1: 2, 3, 8, 9; Thread 2: 4, 5, 10, 11.
   What schedule clause should use? (A)
   A. schedule(static)
   B. schedule(static,2)
   C. schedule(dynamic)
   D. schedule(dynamic,2)

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

1. What are the main considerations for partitioning design when designing parallel algorithms?

Solution:

Considerations for partitioning design when designing parallel algorithms:

- Whether or not my partition defines at least an order of magnitude more tasks than there are processors in your target computer. If not, I will have little flexibility in subsequent design stages.
- Whether or not partition avoids redundant computation and storage requirements. If not, the resulting algorithm may not be scalable to deal with large problems.
- Whether the tasks are of comparable size. If not, it may be hard to allocate each processor equal amounts of work.
- Whether the number of tasks scale with problem size or not. Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, my parallel algorithm may not be able to solve larger problems when more processors are available.
- Whether or not if I have identified several alternative partitions, I can maximize flexibility in subsequent design stages by considering alternatives

now. I will have to remember to investigate both domain and functional decompositions.

2. Please briefly compare OpenMP and Pthreads by describing the advantages and disadvantages of each theads model.

Solution:

OpenMP and Pthreads:

Both Pthreads and OpenMP are multiprocessing concepts that are completely distinct. Pthreads is a threading API with a very low level of abstraction. OpenMP, on the other hand, is considerably higher level, more portable, and does not require you to use C. It can also be scaled much more simply than pthreads.

When compared to a sequential version of matrix multiplication, OpenMP does not surpass it at first. The sequential version is faster than OpenMP on tiny matrices, but as you go to 128 x 128 or larger, OpenMP starts to catch up. After 128 x 128 matrices, the OpenMP advantage continues to improve with each level.

On the majority of the tests, Pthreads outperforms the sequential version. The sequential version outperforms it on the smallest matrix, and it has the same execution time on the 64 × 64 matrix.

advantages and disadvantages

- Pthreads requires a far more customized programme to threading than OpenMP, which was much easier to adapt to sequential code.
- Using OpenMP, it was easier to maintain a programme and represent parallelism in the programme than using Pthreads.

Disadvantages of openMP

- OpenMP codes (with the exception of Intel's OpenMP) cannot be run on distributed memory machines.
- Requires an OpenMP-capable compiler (most do), which is limited by the amount of processors on a single computer.
- It have poorer parallel efficiency on a regular basis.

One of the advantages of OpenMP is that it may coexist with compilers that do not support it. When numerous threads are running at the same time, it is frequently required for one thread to synchronize with another.

- OpenMP provides a number of different forms of data synchronization to assist in a variety of situations circumstances.

The runtime library implements Pthreads as user threads.The async-signal safety of Pthread functions is not guaranteed. Signal handlers must not be summoned.

- Default properties on Pthread synchronization objects may be sensitive to errors.

3. What are the differences between shared memory and message passing programming paradigms? Please describe the pros and cons of them.

Solution:

Shared memory:

It is a memory that any number of process can come and do there specific action. Share means it is share to all the process we do not hide it from any other process so the write of other processes can be read by other process too and the other process also change those write to their own write. To remove the anomaly we use synchronization and concurrency so that the process cannot affect each other work.

Message passing:

It is the technique in which a process can do their work independently and when it got finished it passes the message to other process that i have done my work you can began your work this paradigm is error free but bit slow compare to shared memory paradigms.

Pros of shared memory

- Execution of various process at a same time so more efficient
- Global space so best for user friendly model
- Cpu and resources utilization is less compare to message passing.

Cons of shared memory

- Scalability - There is always we need to specify the process scalability and their priorities
- Deadlock - due to share memory we face the issue of deadlock and starvation for other process.
- Hardware requirements - more hardware complexity can occur to handle all these drawbacks.

Pros of message passing

- Scalability - we do not need to specify any process scalability

- No need of techniques like synchronization concurrent process
- Naturally good for the process to maintain a local environment

Cons of message passing

- Lot of overhead cpu wastage because of single process working on i/o
- Little complicated to implement compare to serial execution
- Always need a technique or passing messages so cpu utilization are there.

V. Programming (29 points)

1. The following program uses Non-blocking MPI routines. Please complete the missing lines of code. (**9 points**)

```
#include "mpi.h"
#include <stdio.h>
int main (int argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

/* Please complete the missing lines of code using MPI Routines here.*/
MPI_Init(&argc, &argv);
_____
MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
_____
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
_____
Prev = rank-1;
next = rank+1;
if    (rank == 0)    prev= numtask-1;
if    (rank == (numtask -1)) prev= 0;
MPI_Irecv (&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &regs[0]);
```

/* Please complete the missing line of code using MPI Routine here.*/

MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &regs[1]);
MPI_Isend (&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &regs[2]);

/* Please complete the missing line of code using MPI Routine here.*/

MPI_Isend (&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &regs[3]);

printf(" Process %d of %d\n", rank, numtasks);

/* Please complete the missing line of code using MPI Routine here.*/

MPI_Waitall (4, regs, stats);

MPI_Finalize();
}

2. (**6 points**) Consider the problem of counting the number of 5 in ***a[]*** using *t* number of threads. Suppose that length can be evenly divided by *t*. Function *counting* is the thread function, *length* is the size of *a[]*.
   The following solution has been proposed:

```
void* counting(void* thread_id){
  int id = (int)thread_id
  int length_per_thread = length/t;
  int start = id * length_per_thread;
  int i;
  for (i= start; i< start+length_per_thread; i++)
  {
      if (a[i] == 5)
          count +=1;
  }
}
```

Questions:

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

Solution:

There is a problem in using the common counter in all the threads. Instead of this we should use an array of the t parts (it means that size of the array is t). Intially the value of all the elements of this array will be zero and each thread will increase the value of their part only. For example we have 4 threads then:

we will have an array (let's name it countArray) [0,0,0,0]

1st thread having id = 0, will increase the count in of first element of the array, i.e. countArray[0].

Similarly 2nd thread will increase the count of countArray[1].

And at the end when all threads have finished their work we will add all element of the countArray.

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

Solution:

I will have to only replace the count with countArray[id] .

Code:

```
void* counting(void* thread_id){

int id = (int)thread_id

int length_per_thread = length/t;

int start = id * length_per_thread;

int i;

for (i= start; i< start+length_per_thread; i++)

{

if (a[i] == 5)

coutArray[id] +=1;

}

}
```

3. Read the program *main.c* and answer the questions. (**8 points**)

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

void main() {
    int x = 0;
    int y = 0;
    int i;
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        #pragma omp atomic
        x++;
        #pragma omp master
        {
            #pragma omp critical
            {
                y++;
                printf("master:  %d\n",  omp_get_thread_num(  ));
            //Line-1
            }
        }
        #pragma omp single
        {
            #pragma omp critical
            {
                printf("single:  %d\n",  omp_get_thread_num(  ));
            //Line-2
                y++;
                }
            }
    }
    printf("x: %d, y: %d\n", x, y); //Line-3
}
```

Questions:

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

Solution:

To execute the program use the following command:

gcc -o op open.c -fopenmp

where,

gcc - is the compiler

o - is the parameter used to create executable

op - is the executable

open.c is the program name

fopenmp - is the parameter to compile and link openmp library

2) What are the outputs of Line-1, Line-2 and Line-3? (**6 points**)

Solution:

The output is as follows:

Line 1: master = 0

Line 2: single = 3

Line 3: x=4 and y=2

Conclusion:

I compiled an openmp program using the gcc compiler using the flag -for openmp with the gcc command and create executable of the openmp program.

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 tmp;
for (i=1; i< X; i++)
      for (j= 0; j<Y; j++) {
            tmp = i*j;
            a[i,j]=tmp*a[i-1,j];
}
```

Solution:

```
#define X 50

#define Y 50

int i, j;

double a[X,Y];

double tmp;

#pragma omp parallel for private (i, tmp)

for (j= 0; j<Y; j++)

for (i=1; i< X; i++) {

tmp = i*j;

a[i,j]=tmp*a[i-1,j];

}
```

Explanation:

In this program for loop parallelization will be done.

In first subscript parallelize j loop first which is inner loop due to that here we need to swap the for loops (inner loop and outer loop) which solves the overhead problem of forking.

To perform above operation successfully make variables I and tmp as private as given in the code.