CPSC 8810/ECE 89300

High-Performance Computing for Power System Modeling and Simulation

Mid-term Exam

04/02/2020

NAME: _____Biyang Fu_____ SCORE: _____ / 100

I. Single Choice (20 points)

1. (4 points) In **OpenMP**, the **Default clause** allows the user to specify the default scope for all of the variables in the parallel region. If it is not specified at the beginning of a parallel region, the default is automatically set to ____? _C
   (A) NONE
   (B) Private
   (C) Shared
   (D) NULL

2. (4 points) Which **synchronization constructs** in **OpenMP** executes a structed block within a parallel loop in sequential order? _B
   (A) Barrier Construct
   (B) Ordered Construct
   (C) Atomic Construct
   (D) Critical Construct

3. (4 points) In **CUDA** programming, what kind of **device memory type** is best used for data that will not change over the course of kernel execution (read-only) to reduce the required memory bandwidth? _C
   (A) Global memory
   (B) Shared memory
   (C) Constant memory
   (D) Registers

4. (4 points) The atomicAdd() function invocation in **CUDA _A**
   (A) is slow because requires synchronization
   (B) is actually very fast and typically its impact on the performance is negligible
   (C) can only be invoked by threads in the same warp
   (D) can probably be interrupted by another thread

5. (4 points) Which **MPI** routine reduce values from all processes and distribute the result back to all processes? _B
   (A) MPI_Reduce
   (B) MPI_Allreduce
   (C) MPI_Reduce_scatter
   (D) MPI_Allgather

II. Multiple Choices (20 points)

1. (4 points) Which of the following are the benefits of using **high performance computing**? _ABCD
   (A) Save time and/or money
   (B) Solve larger/more complex problems
   (C) Provide concurrency
   (D) Make better use of underlying parallel hardware

2. (4 points) If the total size of a **GPU** block is limited to 512 threads, which of the following **subdivision of a grid** is allowable? _ABD
   (A) (8, 16, 2)
   (B) (16, 16, 2)
   (C) (32, 32, 1)
   (D) (512, 1, 1)

3. (4 points) Which of the following are **shared memory (with threads) based model**? _ABC
   (A) OpenMP
   (B) Pthreads
   (C) CUDA
   (D) MPI

4. (4 points) Which of the following statements regarding the use of **Pthreads** are true? _ABD
   (A) Once created, threads are peers, and may create other threads
   (B) Condition variables must be used together with mutex to avoid race condition
   (C) A thread cannot be canceled by another thread
   (D) Only one argument can be passed to the thread start routine pthread_create()

5. (4 points) Which of the following statements regarding **MPI communicators and groups** are true? _ABC
   (A) An MPI communicator is an object containing a group of processes that may communicate with each other
   (B) We can create a new communicator by splitting the original communicator into n-parts
   (C) Every process in an MPI Group has a unique rank between 0 and MPI_Group_size-1
   (D) A group is a global object, and can be used for any communication

III. Short Answer (20 points)

1. (4 points) State an advantage and a disadvantage of **shared memory parallel architecture**.

Advantages:

Global address space provides a user-friendly programming perspective to memory.

Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

Disadvantages:

Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.

2. (4 points) What happens if two threads assigned to different blocks write to the same memory location in global memory? How to resolve the problem in **CUDA**?

Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is write. Programs with race conditions may produce unexpected, seemingly arbitrary results.

We can resolve this problem by using atomics operation. Atomic operation is an operation that forces otherwise parallel threads into a bottleneck, executing the operation one at a time.

3. (4 points) Does the following code have a **loop-carried dependence**? If so, identify the dependence or dependences and which loop carries it/them.

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      A[i][j+1] = A[i][j] + 1;
```

Yes. The j loop has a loop-carried dependence. Dependence exists across iterations, if the j loop is removed, the dependence will not exist.

4. (4 points) What is **reduction**? Give an example of a realistic use of reduction in **OpenMP** or **MPI**.

In OpenMP, a reduction clause allows you to get a single value from associative operations such as addition and multiplication. Private copies of the reduction variable are sent to each thread and at the end of the parallel region, the copies are summarized (reduced) to give a global shared variable.
For example:

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

#define N 10

int main(void) {
        float a[N], b[N], result;
        int I, TID, nthreads;
```

```
        omp_set_num_threads(4);
#pragma omp parallel default(none), private(i), shared(a,b)
        {
#pragma omp for
                for (i=0; i<N; i++) {
                        a[i] = (i+1)*1.0;
                        b[i]= (i+1)*2.0;
                }
        }
                result = 0.0;
#pragma omp parallel default(none), private(i), shared(a,b) reduction(+:result)
        {
#pragma omp for
                for (i=0; i<N; i++)
                        result = result + (a[i]*b[i]);
        }
        printf("final result = %f \n", result);
}
```

5. (4 points) Explain which sparse linear solver (Ax=b) is more readily parallelizable on GPU, direct methods or iterative methods?

When the matrix A is sparse, the factors obtained from the matrix decomposition when using direct methods are denser than the input matrix. Moreover, direct methods are prohibitive in terms of memory and floating-point operations when solving very large systems. These direct methods are also not easily parallelized on modern architectures. Thus, iterative methods, computing a sequence of approximate solutions for the system Ax = b starting from an initial guess, are a good alternative.

IV. Problem Solving (40 points)

1. (8 points) Explain what the following **Pthreads** code is doing:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

pthread_cond_t is_zero ;
pthread_mutex_t mutex;
int shared_data = 5;

void *thread_function(void *arg)
{
  while (shared_data > 0) {
    pthread_mutex_lock(&mutex);
    --shared_data;
    pthread_mutex_unlock(&mutex);
  }

  pthread_cond_signal(&is_zero);
  return NULL;
}

int main(void)
{
  pthread_t thread_ID;
  void *exit_status;
  int i;

  pthread_cond_init(&is_zero, NULL);
  pthread_mutex_init(&mutex , NULL);

  pthread_create(&thread_ID , NULL, thread_function, NULL);

  pthread_mutex_lock(&mutex);
  while (shared_data != 0)
    pthread_cond_wait(&is_zero, &mutex);
  pthread_mutex_unlock(&mutex);

  pthread_join(thread_ID, &exit_status);

  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&is_zero);
  return 0;
}
```

This code is using condition variables waiting and signaling. It uses pthread_cond_t to declare the condition variables at first. And pthread_cond_destroy(condition) can destroy the condition variables. Pthread_cond_signal() signals (or wake up) another thread which is waiting on the condition variable. It is called after mutex is locked, and unlocks mutex in order for pthread_cond_wait() routine to complete. Pthread_cond_wait(condition, mutex) blocks the calling thread until the specified condition is signaled. It Should be called while mutex is locked, and it will automatically release the mutex while it waits. We must call pthread_cond_wait() before calling pthread_cond_signal().

2. (8 points) Explain how you would parallelize the flowing segment of C code. Insert the necessary **OpenMP** pragma(s) into your code to parallelize it.

```
#define M 100
#define N 200

int i, j;
float a[M][N];
flaot sf;

for (i=2; i<M; i++)
  for (j=0; j<N; j++) {
    sf = i*j+(N-j);
    a[i][j] = sf * a[i-2][j];
  }
```

```
#define M 100

#define N 200

int i, j;

float a[M][N];

float sf;

#pragma omp parallel default(none) shared (a) private (i, j, sf)

for (i=2; i<M; i++)

#pragma omp for

    for (j=0; j<N; j++){

            sf = i*j +(N-j);

            a[i][j] = sf  * a[i-2][j];

    }
```

3. (8 points) **GPU** G80 has 16KB of shared memory per SM, and each SM accommodates up to 8 blocks. If each block has to use 3KB of shared memory for an application, no more than how many blocks can be assigned to each SM?

   To reach this maximum, each block must not exceed 16KB/8 = 2KB of shared memory.
   If each block has to use 3KB of shared memory for an application, no more than 16/3 = 5 blocks can be assigned to each SM.

4. (8 points) Suppose I have an integer array A as below:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Write an **MPI** code to create an **indexed datatype** by extracting the highlighted variable potions of array A and distribute to it to all tasks.

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 9

main(int argc, char *argv[])  {
  int numtasks, rank, source=0, dest, tag=1, i;
  int blocklengths[3], displacements[3];
  float a[16] =
    {1, 2, 3, 4, 5, 6, 7, 8,
     9, 10, 11, 12, 13, 14, 15, 16};
  float b[NELEMENTS];

  MPI_Status stat;
  MPI_Datatype indextype;   // required variable

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  blocklengths[0] = 3;
  blocklengths[1] = 4;
  blocklengths[2] = 2;
  displacements[0] = 3;
  displacements[1] = 8;
  displacements[2] = 13;


  // create indexed derived data type
  MPI_Type_indexed(3, blocklengths, displacements, MPI_INT, &indextype);
  MPI_Type_commit(&indextype);

  if (rank == 0) {
    for (i=0; i<numtasks; i++)
```

```
      // task 0 sends one element of indextype to all tasks
        MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
      }

    // all tasks receive indextype data from task 0
    MPI_Recv(b, NELEMENTS, MPI_INT, source, tag, MPI_COMM_WORLD, &stat);
    printf("rank= %d  b= %d %d %d %d %d %d %d %d %d\n",
          rank,b[0],b[1],b[2],b[3],b[4],b[5],b[6],b[7],b[8]);

    // free datatype when done using it
    MPI_Type_free(&indextype);
    MPI_Finalize();
}
```

5. (8 points) Dynamic contingency analysis considers the power system's dynamic behavior over a certain period of time under a contingency, e.g. sudden changes in generator or load, or a network short circuit followed by protective branch switching operations, etc., to determine the transient stability of the system. Suppose we have 1,000 independent **dynamic contingency analysis** cases to run, each running a **dynamic simulation** (comprising of a serial of matrix manipulation and linear system solving) which takes 30 seconds to complete. If we run these 1,000 cases sequentially, it takes over 1,000x30=30,000 seconds to complete. Given 100 available processors, please design a parallel approach to reduce the overall execution time to the largest extent. Show your design below and specify the platform or APIs you'd like to use if your design is implementation specific.

   We can get a Parallel Contingency Analysis Solution using OpenMP standards. Here is a parallel contingency analysis solution with 2 processors.

```
Process inputs
!$ OMP PARALLEL DO
   do i = 1, N of contingency cases

     Solve the ith contingency

!$OMP CRITICAL

     Store the results
     Present the messages

!OMP END CRITICAL

   enddo
!$ OMP END PARALLEL DO
```

These codes are serial and used in parallel programming without modifications. Some serial codes are still executed sequentially for instance processing option inputs. Before any functions can be placed within the parallel do loop, their program codes must comply with OpenMP standards in terms of data dependence and flow controls.