TDT4200 Parallel Computing - Fall 2024 Final Exam Solutions

1.1 Performance - T/F

Processor speed is always more important than data locality with respect to performance
Select one alternative (1pt, -0.3 for wrong answer, 0 for no answer)

False – Remember: 1) Location 2) Location and 3) Location (See lecture notes)

1.2 Global Sum T/F

Parallelizing a global sum can be done in a tree structure on hypercubes.
Select one alternative (1pt, -0.3 for wrong answer, 0 for no answer) :

True – this was shown on the blackboard in class.

1.3 Threads prog. can start - T/F

The number of threads a program can start is limited to the number of cores in the processor
Select one alternative (1pt, -0.3 for wrong answer, 0 for no answer):

False - counterexample: Hyperthreading

1.4 Heap T/F

The heap is used to keep track of active function parameters
Select one alternative : **False**

1.5 Pthread Mutex Blocking T/F

A call to pthread_mutex_lock will block the calling thread until its pthread mutex t argument becomes available.
Select one alternative : **True**

1. 6 OpenMP loops - T/F

In an **OpenMP parallel for** loop, a **dynamic** schedule will always assign the same iterations to the same threads if the program is run repeatedly.
Select one alternative (1pt, -0.3 for wrong answer, 0 for no answer): **False**

1.7 MPI Rank - communicator - T/F

An MPI process will retain its original rank as a key in calculating its new rank when doing an MPI_Comm_Split
Select one alternative (1pt, -0.3 for wrong answer, 0 for no answer): **True**

1.8 CUDA thread block T/F

A CUDA **thread block** may be distributed across several SMs.
Select one of the following alternatives (1pt, -0.3 for wrong answer, 0 for no answer):

1.9 Scalability - Two T/F questions

Total weight, 2/80 (-0.3pts for each wrong answers, 0pts for no answers)

a) The Matrix-vector multiplications are strongly scalable.
Select one alternative:

False – Matrix multiplication is known to have weak scalability, i.e. good scalability if increasing problem size (see below). For strong scaling there should be no gathers, reductions or broadcasts.

b) Gustavson´s Law describes how weak scalabilty is maintained by increasing problem size:
Select one alternative:True

2.1 Latency & Bandwidth

What is the difference between **Latency** and **Bandwidth?**
Select one alternative (max 1pt):

a) Latency reflects how long the total communication is, whereas bandwidth the communication rate - No

✅ b) Latency reflects the time elapsing between a source starting to send data until the data starts arriving at the destination, whereas bandwidth is the rate at which a link can transmit data - Yes

c) Latency is how long it takes before the message is ready to be sent, whereas bandwidth is how long it takes to send the message - No
d) a) and b) above - No

2.2 GPU features

Which of the following apply to GPUs -- check all that apply
Select one or more alternatives (max. 2 pts, -0.3 for each wrong answer, 0 for no answer):

✅ Optimization of different level of caches is key
Clock speed is typically higher than CPUs
Fast access to global memory
✅ Memory access optimization though coalescing is key

2.3 SIMT vs SIMD - Multiple choice

Which of the following are true for SIMT -- check all that apply
Select one or more alternatives (max 2 pts, -0.3 for each wrong answer):

Single instruction, multiple operators X
Single instruction, multiple addresses✅
Single instruction, multiple outputs X
Single instruction, multiple flow paths ✅
Single instruction, multiple register sets✅

2.4 CUDA steps - fast or slow

CUDA program typically have the 5 following steps:
1) Setup inputs on the host (CPU-accessible memory)
- Allocate memory for outputs on the host CPU
- Create the kernel that performs the calculations
2) Set up device (GPU) memory incl. Copy inputs from host to GPU
3) Execute kernel on device (GPU)
4) Transfer result from device (GPU) to host
5) Free the device memory

Which are typically the **slow** steps:
Select one alternative (1pt for correct answer, no points if wrong or no answer):

✅ **Steps 2 and 4** (Not **1** & 2, **3** & 4 nor **3 & 5), 1 typically a small part of overall. Doing a lot of 2 & 4 kills performance.**

2.5 Elster´s Algorithm

One or more of the following is true for Elster´s Algorithm
Select one or more alternatives that are TRUE regarding Elster´s algorithm:

✅ Linear with a low scaling factor ✅ May have tail recursion
✅ Loop controls can be done with shift operations X Uses doubly nested loops
X O(N) with a high scaling factor due to complexity X O(N log N)

2.6 ARM GPUs

Based on what you learned at the ARM guest lecture or slides from 2023:
Select one or more alternatives (max. 2pts for correct answers, 0 for no or wrong answers):

✅ ARM Norway was originally established by NTNU students as Falanx Microsystems
X ARM GPUs use branch prediction
X ARM GPUs use speculative execution
✅ ARM GPUs add performance by adding more core
✅ Having the right memory layout makes a huge difference on ARM GPUs

3. MPI programming -- two longer questions

MPI programming -- total score max 16/80

Consider the following MPI program, which allocates and initializes N x N matrices on each rank, and sets a particular value in the diagonal elements only on rank 0:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

int rank, size;

int N = 8;
double *a = NULL;
#define A(i,j) a[(i)*N+(j)]

int
main ( int argc, char **argv )
{
    if ( argc > 1 )
          N = strtol ( argv[1], NULL, 10 );
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );

    a = malloc ( N*N*sizeof(double) );

    for ( int i=0; i<N; i++ )
       for ( int j=0; j<N; j++ )
             A(i,j) = (double) rank;

    if ( rank == 0 )
          for ( int i=0; i<N; i++ )
               A(i,i) = 42.0;

/* QUESTION 3.1 CODE HERE */
/* QUESTION 3.2 CODE HERE */

        free ( a );
       exit ( EXIT_SUCCESS );
}
```

3.1

Extend the program (above/on the last page) **so that rank 0 transmits its diagonal values into the diagonals of the matrices at every other rank without overwriting the rest of their matrices.** Fill in your answer here

```
MPI_Datatype diagonal;
MPI_Type_vector ( N, 1, (N+1), MPI_DOUBLE, &diagonal );
MPI_Type_commit ( &diagonal );
MPI_Bcast ( a, 1, diagonal, 0, MPI_COMM_WORLD );
MPI_Type_free ( &diagonal );
```

3.2 Extend the program to shift the matrices in a circle so that each rank < size−1 replaces its matrix with the one from rank + 1, and rank size − 1 replaces its matrix with the one from rank 0.

Fill in your answer here

```
double *a_temp = malloc ( N*N*sizeof(double) );
 int left  = (rank+size-1)%size;
 int right = (rank+1)%size;
 MPI_Sendrecv (a, N*N, MPI_DOUBLE, left, 0, a_temp, N*N, MPI_DOUBLE, right, 0,
     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
memcpy ( a, a_temp, N*N*sizeof(double) );
free ( a_temp );
```

See next page for **Problem 4 - Pthreads, OpenMP and Barriers**

4.1 Pthreads - barrier

In pseudo-code∗, write a barrier operation for a threaded program using only locking/mutual exclusion operations (equivalent to pthread mutex t or omp lock t).
(∗*It is not necessary to write detailed and correct pthreads code, only to show how your algorithm works*) **Max 10/80 pts given**.
Fill in your answer here:

```
// Minimal busy-waiting variant; versions where threads 0 through n_threads-2
//  go to sleep and thread n_threads-1 signals them to wake up are better

// Global variables, initial values
 n_threads = 12
 counter = 0
 lock = open

 // Thread-local function
 set_lock ( lock )
 counter = counter + 1
 release_lock ( lock )
 while ( counter < n_threads )
    wait ( short_interval )
```

4.2 OpenMP Scheduling

What is the difference between the dynamic and guided OpenMP schedules?
Fill in your answer here (max. 3pts)

The dynamic schedule executes threads as soon as possible and uses CPU resources as soon as they're available.
The guided schedule allows for putting restrictions on how to organize the threads

4.3 #pragma omp single

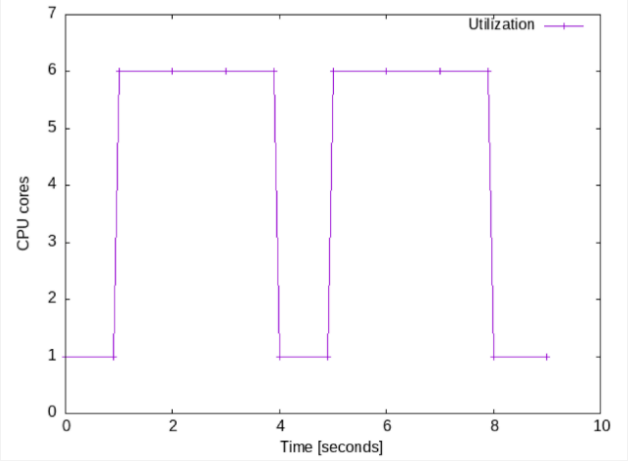
Briefly describe the effect of the **#pragma omp single** directive, as well as a practical use-case for it.
Fill in your answer here (max. 3 pts)

This directive indicates that the annotated block should only be executed on a single thread. This is useful for spawning tasks using #pragma omp, as one usually wants to spawn tasks only once from one thread. Without the single directive, the tasks would be spawned from every thread that is executing the block.

5 General Parallel Computing Questions

5.1 Amdahl - graph

Consider the following graph of parallel utilization over time, recorded from a threaded program on a 6-core processor. Assume that the parallel work speeds up linearly with increasing core counts, and that we compare runs with identical input data.



What is the maximal speedup of the program if we increase the number of cores? (Max 2 pts) Fill in your answer here

From the graph we see 39 seconds of work total, 36 are parallelizable.

f = 3/39 = 1/13, so lim_(p-> ∞) S(p) = 13

Many answered looking at the speedup compared to the measured time with 6 cores, which would be equal to 1/(3/9) = 9/3 = 3x, but this is NOT the max. speedup of the program.

5.2 on next page

5.2: 5 vs 9pt stencils

Let's say we wanted to use a 9-point stencil for approximation on the current time step instead of just 5, as illustrated in Figure 11 from PS 3.

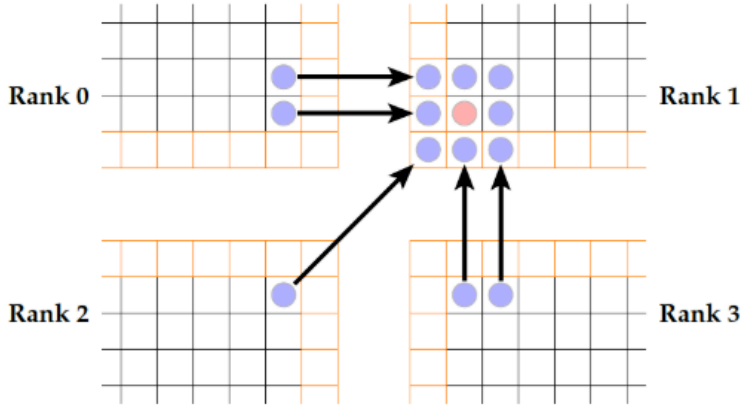


Figure 11: Rank 1 depends on multiple other processes to calculate the red cell for the next time step. Orange represents the ghost cells.

How would you communicate the value needed from Rank 2? (Max. 2 pts)
Hint: There are multiple solutions to this question.

Fill in your answer here

THIS PROBLEM WAS VERBATIM FROM PROBLEM SET GIVEN EARLIER THIS SEMESTER SEE ALSO PS SOLUTIONS!

One solution: We could synchronize the communication in such a way that all east-west exchanges happen first. Then rank 3 would already have the desired value from rank 2. After that, all processes need to be synchronized. Then, all north-south exchanges would happen, leading to the original value from rank 2 being transferred to rank 1 via the communication between rank 3 and 1.

5.3 Pthread borders

Why is there no need for a border exchange when using Pthreads?
Fill in your answer here (max. 2pts)

Pthreads assumes shared memory, so the domain data is shared among the processes. The processes do not maintain their own separate copy of the data, as is the case with distributed memory environments such as MPI. Whenever one thread updates a shared variable, it is immediately available to the other threads, barring issues with caching and memory conflicts

5.4 GPU loops /fast?

Given an input signal x[n], suppose we have two output signals y_1[n] and y_2[n], given by the difference equations:

(a) y_1[n] = x[n - 1] + x[n] + x[n + 1]

(b) y_2[n] = y_2[n - 2] + y_2[n - 1] + x[n]

Which calculation do you expect will have an easier and faster implementation on the GPU, and why?

Fill in your answer here (max 2 pts given)

Equation (a) will be faster, as it clearly distinguishes between read-only and write-only variables. It only reads consecutive values from x, meaning all threads can access it without being interrupted by other threads writing to the array. This is not the case for (b). Here, one thread may read a value from y_2 before updating another value. Consequently, every thread will have issues with caching the values, as they're consistently written by other threads.

5.5 CUDA True/False (4pts total)

Each correct True/False answer is worth 1 pt, -0.3 for wrong answer, 0 for no answer.

5.5 (a) The __global__ function qualifier indicates that the function **can be executed on the host and on the device** (GPU). The function can be called from the host or from the device. **FALSE** _global__ functions are functions called on host (typically from main()) to be executed on the GPU (the device), using the <<< …>>> syntax. In newer versions of CUDA, they may be called from the device within other *global* functions, but the function called is not executed on host.

5.5 (b) The __syncthreads() function synchronizes all threads in the grid.
FALSE The __syncthreads() command is a **block-level** synchronization barrier. A CUDA grid contains several blocks.

5.5 (c) A variable marked with a __shared__ type qualifier is shared among all threads in a thread block, while a variable with a __constant__ type qualifier is shared among all threads in the grid. **TRUE**

5.5 (d) Shared memory has higher throughput and lower latency compared to global memory.
Select an alternative: **TRUE**

Section & - CUDA Questions

Oppg 25/ 6.1 CUDA - short answer

You can get max, 6/80 pts total on the following.

6.1 a) You decide to parallelize a C program that takes 20 seconds to run on a CPU.

You implement a CUDA kernel to replace a C function that consumes 80% of the CPU runtime and find that the total runtime drops to 12 seconds.

i) How much time was spent in the new code? Show your calculation
Fill in your answer here

The new GPU-specific code takes 8 seconds : 8 = 12 - 20(1 - 0.8) seconds.
or
The old C function took 80% of the total runtime of 20 seconds = 16 seconds.
This leaves 4 seconds for the remaining non-parallel C code.
The new total runtime is 12 seconds = 4 + new code = 8 seconds runtime of new code.

ii) You measure the execution time for the GPU kernel and find that it finishes in 2 seconds. What other activity can account for the remaining time spent in the new code?
USE NO MORE THAN 10 WORDS FOR YOUR ANSWER

Fill in your answer here

Allocating device memory and data transfer between host and device.

6.1 b) Name at least 2 advantages of parallelizing with CUDA as opposed to MPI?

Fill in your answer here

- CUDA uses lightweight threads, whereas MPI is process-based.
- CUDA uses shared memory /shared address space during calculations, except from to/from host

See also PS 6 solutions

6.2 CUDA Programming (10/80)

6.2 Part 1. Your internship co-worker notices that the kernel shown below doesn't always work and asks you to help debug it. Each block has a shared flag (cleared in each iteration), and any thread that triggers some condition sets the flag. The kernel should terminate when no thread triggers the condition, but sometimes, some of the threads terminate early.

```
__global__ void iterate(void* data)
{
   __shared__ int blockContinueFlag;
   do {
      if (threadIdx.x == 0) {
         blockContinueFlag = 0;
      }
      __syncthreads();
      //do some work...
      if (some condition) {
         // If >= 1 thread(s) execute this, flag will be written to 1
         blockContinueFlag = 1;
      }
      __syncthreads();
   } while (blockContinueFlag);
}
```

6.2 Part 1, (a) Explain what is going wrong (in given code)? USE NO MORE THAN 30 WORDS!
Fill in your answer here

Thread 0 is able to reset the flag to 0 before other threads (other warps) evaluate the flag for the loop test. This is a serious synchronization issue, even if the intent is to synchronize per block only. 0.5 pts if mention only synch of blocks.

(b) Explain how to fix this bug using NO MORE THAN 15 WORDS:

Fill in your answer here:

Add _synchthreads at the beginning of the loop.
0-5- 1pt if give a general solution, not a per block one, as intended. For synchronization across blocks only, you could set up a flag in global memory.

6.2 Part 2: Coalesced Memory Access (6 points)

Let the block shape be (32, 32, 1). Let data be a (float *) pointing to global memory and let data be 128 byte aligned (so data % 128 == 0).

Consider each of the following access patterns.

a) data[threadIdx.x + blockSize.x * threadIdx.y] = 1.0;
Is this write coalesced? How many 128 byte cache lines does this write to?

Fill in your answer here

Yes, this write is coalesced. In a given warp, threadIDx.x will vary from 0 to 31, so the write will access 32 consecutive floats offset starting from 0. A float is 4 bytes , 32 floats * 4 = 128 bytes, so each warp writes to one 128-byte cache line.

(b) data[threadIdx.y + blockSize.y * threadIdx.x] = 1.0;
Is this write coalesced? How many 128 byte cache lines does this write to?

Fill in your answer here

This write is not coalesced because each warp is not accessing the minimum possible 128-bye cache lines. The code does not access the array consecutively. In a warp, threadIdx.y will be the same and threadIx.x will vary from 0 to 31. So we can think of a warp as accessing 32 consecutive rows 1 column per instruction of the data array is 2D where each access is 32 floats = 128 bytes away from the previous access.
So each warp will access 32 different 128-byte cache lines since in a warp each thread is accessing a float 128 byte away form each of the next closest access.

t thus will writes to 32*32=1024 cache lines.
The thread 0,0 accesses location 0, thread 1,0 accesses location 32 and so on. This leads to every warp requiring 32 different cache lines. As there are 32 warps in the block configuration, this adds up to 1024 cache lines.

6.2 Part 3. (PS6 question)

What are some pros and limitations when using cooperative groups?
List at least 2 pros and 2 limitations for full score.

Fill in your answer here

Pros:
- More versatile synchronization options that reflect the hardware architecture.
- Additional communication options, useful for e.g. global sum.

Cons:
- The entire grid must fit into the GPU's capacity. Grid can't be larger than available SMs and their warp capacity.
- May lead to more synchronization and thus more idle time.
- Requires more shared memory, increasing memory pressure.

Bonus question on sustainability (+1 pt max)

Comment on how useful TDT4200 was in making you aware that computing is related to sustainability.

+0.5 pts for parallel computing related to improving performance/reduce energy consumption;
+0.5 for mentioning how solving PDEs can model/simulate physical systems to optimize natural resources.