NTNU

KANDIDAT

10005

PRØVE

TDT4200 1 Parallelle beregninger

Emnekode	TDT4200
Vurderingsform	Skriftlig eksamen
Starttid	06.12.2023 14:00
Sluttid	06.12.2023 18:00
Sensurfrist	06.01.2024 22:59
PDF opprettet	24.10.2024 08:50

Cover Page

Oppgave	Tittel	Oppgavetype
ℹ	Cover page	Informasjon eller ressurser

1. True/False questions

Oppgave	Tittel	Oppgavetype
1	1.1-1.10 True/False	Langsvar

2. OpenMP

Oppgave	Tittel	Oppgavetype
2	2.1 OpenMP	Langsvar
3	2.2 OpenMP	Langsvar

3. MPI and performance models

Oppgave	Tittel	Oppgavetype
4	3.1 MPI and performance models	Langsvar
5	3.2 MPI and performance models	Langsvar

4. CUDA

Oppgave	Tittel	Oppgavetype
6	4.1 CUDA	Langsvar
7	4.2 CUDA	Langsvar
8	4.3 CUDA	Langsvar

5. Programming: OpenMP

Oppgave	Tittel	Oppgavetype
9	5.1 Programming: OpenMP	Langsvar

6. Programming: MPI

Oppgave	Tittel	Oppgavetype
10	6.1 Programming: MPI	Langsvar
ℹ	6.1 Code	Informasjon eller ressurser

Additional pages

Oppgave	Tittel	Oppgavetype
11	Additional text notes	Langsvar
12	Additional handwritten notes	Muntlig

1 1.1-1.10 True/False

Correct answers are scored 1 point. Incorrect answers without any justification are scored -1 point. Incorrect answers with a brief note/explanation are scored 0 points.

1. The number of threads started by a #pragma omp parallel for must be the same for every such directive throughout each run of the program.

2. Atomic operations do not require explicit locks to prevent race conditions.

3. In MPI, a send operation in synchronized mode cannot be non-blocking.

4. A failed branch prediction causes delay in the execution pipeline.

5. The number of pthreads spawned by a program can be higher than the number of physical CPUs in the system.

6. Completion of an MPI Barrier requires all other MPI operations to be completed.

7. An MPI Datatype variable can be committed having different parameters on different ranks.

8. The number of lines in a cache set must be a power of 2.

9. Gustafson’s law does not predict any upper limit on scaled speedup.

10. CUDA kernels can run simultaneously with unrelated function calls on the host processor.

Fill in your answer here

1. False, there is a "num_threads" clause that can be added to the end of the pragma to set a number of threads for that specific parallel block

2. True, atomic operations are atomic at the processor level, so they don't require explicit locks

3. False, all send-operation modes can be non-blocking with the use of MPI_Isend and variants (MPI_Issend for synchronized). They will instead return a MPI_Request that can be waited for later

4. True, as opposed to a successful branch prediction or no branch at all it will cause a delay, as the pipeline must be flushed. It will, however, not cause any additional delay over not attempting branch prediction at all.

5. True, but oversubscription is rarely beneficial in HPC applications as the threads will be "fighting" for execution time on the CPU cores.

6. False, there may still be messages in transit. It only requires all ranks to have reached the barrier

7. True, MPI_Datatypes are simply variables, and each rank in a separate process with separate memory, so this should be no problem. It might get confusing though.

8. False. Some cache-types have requirements for the number of cache-lines but not all. A fully associative cache f.ex. can have any number of cache lines.

9. True. Gustafson's law assumes constant parallel time, meaning any additional parallel processor simply adds more parallel work. There is no limit to the amount of parallel work in relation to the sequential work, thus unlimited speedup.

10. True. CUDA kernels run on the GPU, and the CPU can do other stuff in the meantime.

Ord: 264

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

6 6 9 8 8 3

2 2.1 OpenMP

Briefly describe the difference between dynamic and guided scheduling policies in OpenMP.

Fill in your answer here

Dynamic scheduling policy means OpenMP will divide the work into approximately equally sized chunks and add them all to a work-queue. Whenever one of the threads is free, it will take the first work-package from the queue and execute it. This avoids having multiple finished threads without any work waiting for some slow threads. Especially useful in uneven and unpredictable workloads.

Guided scheduling works similarly, but here OpenMP attempts to optimize package size in relation to the likelyhood of a single thread slowing down the entire program. The first work-packages assigned can be fairly large without any significant risk that one processor will take longer than the rest use on all other work. Later, when we are closer to being finished, there is less leeway for some processors to use more time than others, so it assigns smaller work-packages to reduce risk. OpenMP will therefore split up the work into uneven packages, and the threads will do the largest work-packages first, and the smallest in the end.

Ord: 167

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

6 1 5 2 7 1 5

3 2.2 OpenMP

What is false sharing?

Fill in your answer here

False sharing occurs when two threads running in parallel read/write to different pieces of memory within the same cache-line. Say address A and B are close enough in memory to be cached together. If thread 1 reads address A and thread 2 reads address B, they will cache the memory separately. If thread 1 then writes to address A, the thread 2 will mark the cache as invalid even though it was only using address B. This can cause significant slowdown to parallel programs that can be avoided by more carefully managing the memory layout.

Ord: 95

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 4 2 8 1 8 7

4 3.1 MPI and performance models

What is superlinear speedup, and how can it occur?

Fill in your answer here

Superlinear speedup is when the speedup you gain from adding more processors is larger than the number of processors used. S(p) > p.

This can occur if adding processors splits up the problem into smaller chunks, such that each processor works on a separate chunk. These smaller chunks may fit into a faster cache than the original problem, allowing the processors to calculate their chunk faster than the time it would take to calculate the same part of the problem on a single processor. In some cases this can result in the entire problem being solved faster than 1/p of the sequential time.

Ord: 103

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

4 7 0 1 6 5 4

5 3.2 MPI and performance models

Assume that we have a machine with given Hockney model parameters L (latency/alpha) and B (bandwidth/beta), and a rectangular problem of 128x256 8-byte data points distributed evenly on a Cartesian grid of 2x4 ranks.
What is the estimated cost of a border exchange for a given rank?
Show your reasoning.

Fill in your answer here

As the problem is distributed evenly on a 2x4 grid, each rank will operate with a 64x64 grid of 8-byte data points.

Assuming the problem wraps around on the edges, each rank will need to send and receive 64*4=256 8-byte data points to and from neighbors. Our rank will first need to send all these values and then once they are received, the other ranks can send their values back. This means we will need to wait L+256*B for our thread to send, and then L+256*B for it to receive. In a well optimized program we can send to all neighbors simultaneously, and receive from them simultaneously. Thus the total cost will be 2L+512B.

Ord: 114

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 0 5 5 2 9 9

6 4.1 CUDA

Briefly describe what thread divergence is, and which performance issue it causes.

Fill in your answer here

Thread divergence is when two threads start doing different things (in the code), for example because of a branch. As CUDA warps are SIMT (single instruction multiple thread), threads within the same warp cannot do different instructions at the same time. They will instead run both diverging branches in sequence, with each thread ignoring the instructions that do not apply to them. This obviously causes performance issues when there is a significant amount of divergence, as all the branches will need to be run in sequence on all the threads.

Ord: 90

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

8 1 5 0 5 0 1

7 4.2 CUDA

What is the difference between a thread block and a warp?

Fill in your answer here

A thread block is a collection of threads which is run on a single SM. The threads within a thread block can easily synchronize and communicate with each other, as the SM has a shared register bank for all the threads. A warp is a collection of threads running simultaneously, all with the same instruction. A block might contain multiple warps, as long as they all fit on the same SM.

Ord: 71

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

5 1 5 7 8 0 8

8 4.3 CUDA

Why can it be advantageous to have thread blocks and warps of the same size?

Fill in your answer here

Thread blocks and warps of the same size can give very good utilization of the GPU. This will allow for the thread blocks to fill up the maximum amount of warps in an SM, as long as there is enough space for the required registers. If the thread blocks are multiple warps in size, they might not fit neatly in the SM, and there will be some unutilized recourses that cannot be utilized as they do not fit an entire thread block.

Ord: 82

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

3 4 0 4 6 1 2

9 5.1 Programming: OpenMP

The following sequential program reads a sequence of 2D points from the standard input stream into a list, and calculates the average of distances between all pairs of points. Your task is to parallelize the distance calculation using OpenMP.
It is not necessary to replicate the entire code in your answer, as long as you supply sufficient context to see where your modifications are meant to be placed.

```
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

typedef struct {
   float x, y;
} point_t;

float distance ( point_t *a, point_t *b ) {
   float dx, dy;
   dx = a->x - b->x;
   dy = a->y - b->y;
   return sqrt ( dx*dx + dy*dy );
}

int main ( int argc, char **argv ) {
   int64_t N = (argc>1) ? strtol(argv[1],NULL,10) : (1<<8);
   point_t *list = malloc ( N*sizeof(point_t) );
   for ( int64_t n=0; n<N; n++ )
      scanf ( "%f %f\n", &(list[n].x), &(list[n].y) );

   float sum = 0.0;
   for ( int64_t i=0; i<N; i++ )
      for ( int64_t j=i+1; j<N; j++ )
         sum = sum + distance ( &list[j], &list[i] );

  printf ( "Avg. distance %f\n", sum / (float)N );
  free ( list );
  exit ( EXIT_SUCCESS );
}
```

Fill in your answer here

```
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

typedef struct {
   float x, y;
} point_t;
```

```
float distance ( point_t *a, point_t *b ) {
    float dx, dy;
    dx = a->x - b->x;
    dy = a->y - b->y;
    return sqrt ( dx*dx + dy*dy );
}

int main ( int argc, char **argv ) {
    int64_t N = (argc>1) ? strtol(argv[1],NULL,10) : (1<<8);
    point_t *list = malloc ( N*sizeof(point_t) );
    for ( int64_t n=0; n<N; n++ )
        scanf ( "%f %f\n", &(list[n].x), &(list[n].y) );

    float sum = 0.0;

    // Initialize parallel threads
    #pragma omp parallel
    {
        // Use a local sum in each thread to avoid unnecessary atomic writes to the global sum
        float local_sum = 0.0;

        // Execute loop iterations in parallel, no need to wait at the end (that would only slow down
the atomic add operations)
        #pragma omp for nowait
        for ( int64_t i=0; i<N; i++ ) {
            for ( int64_t j=i+1; j<N; j++ ) {
                local_sum = local_sum + distance ( &list[j], &list[i] );
            }
        }

       // atomic write to global sum to avoid potential conflicts when reading and writing
       #pragma omp atomic
       sum += local_sum;
    }

    printf ( "Avg. distance %f\n", sum / (float)N );
    free ( list );
    exit ( EXIT_SUCCESS );
}
```

Ord: 205

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 6 2 5 8 2 6

10 6.1 Programming: MPI

The code on the following pages is extracted from an MPI-enabled implementation of an image filter, which partitions a 4096 × 4096 array of 3-byte pixel values onto ranks in a cartesian communicator, repeatedly applies a 3 × 3-point computational stencil to it, and saves the result.

The functions to load, save, and partition the image are omitted for brevity, but you may assume that the height and width of each rank's local domain is stored in img slice[2], that the dimensions of the communicator evenly divide the image size, and that the partitioning is otherwise correctly set up by the mpi configure subdomain function.

The problem requires a border exchange routine, because the formula for each point x, y depends on all of its 8 neighbors from x − 1, y − 1 through x + 1, y + 1, as seen in the filter iteration. Your task is to implement this border exchange by extending the code in three locations, each marked with comments containing numbered TODO remarks:

1. Globally declare the MPI Datatype variables you will need.
2. Configure and commit those MPI Datatype variables.
3. Implement the border exchange using your committed data types.

You may assume that the image data are surrounded by a 1-point boundary of already initialized zero values, and that it is not necessary to add any computation or communication in order to implement a boundary condition.

Fill in your answer here

```
1.
// Datatypes for border exchange
MPI_Datatype border_horizontal, border_vertical;

2.
// contiguous data type corresponding to one row of the local image slice
MPI_Type_contiguous ( img_slice[1], mpi_pixel, &border_horizontal );
MPI_Type_commit ( &border_horizontal );

// vector data type corresponding to one column of the local image slice
MPI_Type_vector ( img_slice[0], 1, img_slice[1], mpi_pixel, &border_vertical );
MPI_Type_commit ( &border_vertical );

3.
// Get neighbors in horizontal direction
MPI_Cart_shift (cart, 1, 1, &left, &right);
// Get neighbors in vertical direction
MPI_Cart_shift (cart, 0, 1, &down, &up);

// Send/receive to the right
MPI_Sendrecv( IMG(0,img_slice[1],r), 1, border_vertical, right, 0, IMG(0,-1,r), 1,
border_vertical, left, 0, cart, MPI_STATUS_IGNORE );
```

```
// Send/receive to the left
MPI_Sendrecv( IMG(0,0,r), 1, border_vertical, left, 1, IMG(0,img_slice[1]+1,r), 1, border_vertical, right, 1, cart, MPI_STATUS_IGNORE );

// Send/receive up
MPI_Sendrecv( IMG(img_slice[0],0,r), 1, border_horizontal, up, 2, IMG(-1,0,r), 1, border_horizontal, down, 2, cart, MPI_STATUS_IGNORE );

// Send/receive down
MPI_Sendrecv( IMG(0,0,r), 1, border_horizontal, down, 3, IMG(img_slice[0]+1,0,r), 1, border_horizontal, up, 3, cart, MPI_STATUS_IGNORE );

// Send/receive corners (I'm assuming that the ranks are numbered from bottom left to top right, as I'm unsure how to properly get diagonal neighbors in cartesian communicator)
MPI_Sendrecv( IMG(0,0,r), 1, mpi_pixel, down-1, 4, IMG(img_slice[0]+1, img_slice[1]+1, r), 1, mpi_pixel, up+1, 4, cart, MPI_STATUS_IGNORE ); // down-left
MPI_Sendrecv( IMG(img_slice[0], img_slice[1], r), 1, mpi_pixel, up+1, 5, IMG(-1, -1, r), 1, mpi_pixel, down-1, 5, cart, MPI_STATUS_IGNORE ); // up-right
MPI_Sendrecv( IMG(0, img_slice[1], r), 1, mpi_pixel, up-1, 6, IMG(img_slice[0]+1, -1,r), 1, mpi_pixel, down+1, 6, cart, MPI_STATUS_IGNORE ); // up-left
MPI_Sendrecv( IMG(img_slice[0], 0, r), 1, mpi_pixel, down+1, 7, IMG(-1, img_slice[1]+1, r), 1, mpi_pixel, up-1, 7, cart, MPI_STATUS_IGNORE ); // down-right
```

Ord: 260

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

5 1 5 0 7 0 4

11 Additional text notes

You may add any additional footnotes you wish to attach here.

Fill in your answer here

Ord: 0

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 0 7 0 1 2 5

12 Additional handwritten notes

You may add any additional handwritten notes here.

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 3 9 7 0 1 9