Front Page / Forside

Front Page -- See attached PDF

1 Performance - T/F

Processor speed is more important than data locality with respect to performance
Select an alternative (-0.5 for wrong answer):

- True ✔
- False

Maximum marks: 1

2 Fastest memory - T/F

L1 cache is the fastest memory available on Intel processors
Select an alternative (-0.5 for wrong answer):

- True
- False ✔

Maximum marks: 1

3 Caches T/F

There are generally **fewer conflict misses** in a **set-associative cache** than a **direct-mapped cache.**
Select one alternative (-0.5 for wrong answer):

- False
- True ✔

Maximum marks: 1

4 Heap T/F

The heap is used to keep track of active function parameters
Select one alternative (-0.5 for wrong answer):

- True
- False ✔

Maximum marks: 1

5 Superscalar - T/F

Superscalar performance is typically due to caching
Select an alternative (-0.5 for wrong answer):

- True ✔
- False

Maximum marks: 1

6 Amdahl´s Law T/F

Amdahl's Law is named after Gene Amdahl in the 1960s who observed that the serial parts quickly dominate the execution times when parallelising codes. For instance, if the serial part of the code takes 10% of the time, Amdahl's Law tells you that

$T_{parallel} = \frac{0.9 \cdot T_{serial}}{p} + 0.1 \cdot T_{serial} = \frac{18}{p} + 2$ for $T_{serial} = 20$ seconds
and $\text{Speedup} = \frac{T_{serial}}{T_{parallel}} = \frac{20}{\frac{18}{p}+2}$.

As **p** gets larger, the Speedup approaches 20/2 = 10, even for many thousands of cores!

In general. if a fraction **r of our serial program remains unparallizable, Amdahl´s Law state that we cannot get a better speedup that 1/r!**

Thus, if the serial part of our program takes only 5 % of the time, the max speed-up is 20.
Select one alternative (2 pts for correct answer, -1 for wrong):

- True ✔
- False

Maximum marks: 2

7 FFT & Bit-rev. - T/F

Both the FFT and Elster's Bit-reversal algorithm are O (N log N)
Select an alternative (-0.5 for wrong answer):

- True
- False ✔

Maximum marks: 1

8 Switch stmt -- T/F

When optimizing a **switch statement** for best performance one should sort the statements in ascending (increasing) order with the amount of time they take to compute.
Select one alternative (-0.5 pts for wrong answer)

- True
- False ✔

Maximum marks: 1

9 Global Sum T/F

Parallelizing a global a sum is often done in a tree structure, especially on hypercubes.
Select one alternative (-0.5 for wrong answer) :

- True ✔
- False

Maximum marks: 1

10 MPI basic -- T/F

MPI programs are Single Program Multiple Data (SPMD) rather than SIMD or MIMD (Flynn's Taxonomy)
Select an alternative (-0.5 for wrong answer) :

- True ✔
- False

Maximum marks: 1

11 MPI derived datatype - T/F

MPI_Type_create_struct is used to build an **MPI derived datatype**
Select one alternative (-0.5 for wrong answer):

- True ✔
- False

Maximum marks: 1

12 MPI Collectives & tags - T/F

MPI collectives may use tags
Select an alternative (-0.5 for wrong answer):

- False ✔
- True

Maximum marks: 1

13 Branching - T/F

Branching may lead to performance issues on both CPUs and GPUs
Select an alternative (-0.5 for wrong answer) :

- True ✔
- False

Maximum marks: 1

14 Pthread mutex - T/F

Variables of type **pthread_mutex_t** need to be **initialized** by the system before they are used
Select one alternative (-0.5 for wrong answer):

- False
- True ✔

Maximum marks: 1

15 Thread safety of strtok T/F

The C string library function **strtok** splits an input string into substrings. It can be called recursively. Implementing it by using a static **char*** variable that refers to the string that was passed on the first call, is **thread safe**.
Select one alternative (-0.5 for wrong answer) :

- True
- False ✔

Maximum marks: 1

16 CUDA Warps - T/F

CUDA Warps use the SIMT/SIMD model
Select an alternative (-0.5 pts for wrong answer)

- True ✔
- False

Maximum marks: 1

17 CUDA thread block T/F

A CUDA **thread block** may be distributed across several SMs.
Select one alternative (-0.5 for wrong answer) :

- False ✔
- True

Maximum marks: 1

18 CUDA - Register spilling - T/F

Since an SM may be able to run more than one warp at the same time, performance overall may improve despite register spilling affecting single thread´s performance.

Select one alternative (-0.5 pts for wrong answer) :

- False
- True ✔

Maximum marks: 1

19 HIP and OpenCL

HIP and OpenCL can both be used on systems with AMD GPUs. CUDA is closer to HIP than OpenCL.

Select one alternative (-0.5 for wrong answer):

- True ✔
- False

Maximum marks: 1

20 Comments to T/F questions

If you find one or more of the questions from this section vagues, you may explain one or more of your T/F answers below. Make sure to refer to the T/F problem´s title. We may or may not consider these explainations.

Reminder: Use of Google, ChatGTP or similar tools are not allowed during this exam.

Format

Words: 0

Maximum marks: 10

21 MPI - Make efficient

```
### How would you make the code below more efficient?
```c
int rank, comm_size;
MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
MPI_Comm_size ( MPI_COMM_WORLD, &comm_size );
double data;
if ( rank == 0 ) {
        data = 3.14;
        for ( int i = 1; i < comm_size; ) {
                MPI_Send ( &data, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD );
        }
} else {
        MPI_Recv ( &data, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

Fill in your answer here by giving showing a more efficient code + explaination.

Format

Words: 0

Maximum marks: 6

22 MPI - What is wrong?

```
### What is wrong in this code snippet from an MPI program that tries
### to broadcast from rank 0 to all other processes?
int rank;
MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
int data = 0;
if ( rank == 0 ) {
        data = 3;
        MPI_Bcast ( &data, 1, MPI_INT, 0, MPI_COMM_WORLD );
}
```

Fill in your answer here

Words: 0/50

Maximum marks: 5
```

23 Describe MPI error

```
### What is the error in this code snippet from an MPI program?
int rank;
MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
float data;
if ( rank == 0 ) {
        data = 50.5;
        MPI_Send ( &data, 1, MPI_FLOAT, 1, MPI_ANY_TAG, MPI_COMM_WORLD );
} else if ( rank == 1 ) {
        MPI_Recv ( &data, 1, MPI_FLOAT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

Fill in your answer here

Format

Words: 0

Maximum marks: 4

24 MPI_Cart_shift

The following unfinished MPI code should initialise a cartesian communicator and retrieve information about the position and neighbors of each rank.

Fill in the blanks so that:

- The cartesian communicator *cart* is **periodic in the x axis.**
- The variables *north, south, west,* and *east* are set to the rank of the north, south, west, and east neighbours respectively.

The signature for MPI_Cart_shift:
MPI_Cart_shift(MPI_Comm communicator, int direction, int displacement, int *source, int *destination);

```
#include <mpi.h>
int main ( int argc, char** argv )
{
        MPI_Comm cart;
    int rank, comm_size;
        int dims[2] = {0, 0}
        int periodic[2], coords[2];
        int north, south, west, east;
        MPI_Init ( &argc, &argv );
        MPI_Comm_size ( MPI_COMM_WORLD, &comm_size );
        MPI_Dims_create ( comm_size, 2, dims );
        periodic[0] = _; // Fill in the blank
        periodic[1] = _; // Fill in the blank
        MPI_Cart_create ( MPI_COMM_WORLD, 2, dims, periodic, 0, &cart );
        MPI_Cart_shift ( cart, _, _, _, _ ); // Fill in the blank
        MPI_Cart_shift ( cart, _, _, _, _ ); // Fill in the blank
    MPI_Comm_rank ( cart, &rank );
    MPI_Cart_coords ( cart, rank, 2, coords );
        ...

        MPI_Finalize();
        return 0
}
```

periodic[0] = [ ] (0)

periodic[1] = [ ] (1)

MPI_Cart_shift(cart, [ ] (0) , [ ] (1) ,

[ ] (&north) , [ ] (&south));

MPI_Cart_shift(cart, [ ] (1) , [ ] (1) ,

[ ] (&west) , [ ] (&east));

Maximum marks: 10

25 MPI Deadlock?

```
#include <mpi.h>
int main ( int argc, char** argv )
{
        ...

        int recv_buffer[40];
        int send_buffer[10];
        for ( int i = 0; i < 10; i++ ) {
                send_buffer[i] = i;
        }
        if ( coords[0] != 0 ) {
                MPI_Sendrecv ( send_buffer, 10, MPI_INT, north, 0, &recv_buffer[0], 10, MPI_INT, north, 0, cart, MP
        }
                MPI_Sendrecv ( send_buffer, 10, MPI_INT, south, 0, &recv_buffer[10], 10, MPI_INT, south, 0, cart, M
        if ( coords[1] != 0 ) {
                MPI_Sendrecv ( send_buffer, 10, MPI_INT, west, 0, &recv_buffer[20], 10, MPI_INT, west, 0, cart, MPI
    }
        MPI_Sendrecv ( send_buffer, 10, MPI_INT, east, 0, &recv_buffer[30], 10, MPI_INT, east, 0, cart, MPI_STATUS_
        ...
}
```

Assume the cartesian communicator is set up as described in the MPI_Cart-Shift problem.
Above is code for the processes to communicate with their immediate neighbours.

Will this code lead to a deadlock? Why/why not? State your assumptions. Fill in your answers here:

Format

Words: 0

Maximum marks: 5

26 (8%) OpenMP

(2%) In OpenMP there is no guarantee of fairness in mutual exclusion constructs. This means that it's possible that a thread can be blocked forever when using a #pragma omp critical inside a while loop.

Select an alternative

- True ✔
- False

(2%) OpenMP threads share stack and program counter

Select an alternative

- False ✔
- True

(2%) False sharing occurs when threads use data in the same cache line, but do not use the same variable when accessing the data, so that the behaviour of the thread with respect to memory access is the same as if there were sharing a variable.

How can false sharing be avoided during matrix-vector multiplication?

Select one alternative

- It is not possible to avoid false sharing when doing matrix multiplication in OpenMP
- Have each thread use private storage during multiplication loop, then update shared storage when done ✔
- Make sure the resulting y vector is small enough
- Avoid padding the resulting y vector.

(2%) Why does the following code exhibit poor performance on 4 or more cores?

```
#pragma omp parallel for shared(a,b,sum) private(I,tmp)
for (i = 0; i < n; i++) {
  tmp = a[i] * b[i];
  #pragma omp atomic
  sum = sum + tmp;
}
```

Select one alternative

- synchronisation need to be added
- #pragma omp for needs to use global tmp
- #pragma atomic serialises the summation ✔
- 4 cores are too many for parallelising global sums

Maximum marks: 8

27 (3%) Parallel prog.

Writing efficient parallel programs usually involve
Select one or more alternatives:

- [x] load balancing
- [x] synchronisation of the cores
- [x] coordinating the work of the cores
- [x] communications among the cores

Maximum marks: 3

28 (2%) Looping and efficiency

Given an array defined as double m[2048][2048] and the following two snippets of code:
1)
```
for(i=0; i<2048; i++)
    for(j=0; j<2048; j++) {
        double value = m[i][j]; // do some calculations with value
}
```
2)
```
for(j=0; j<2048; j++)
    for(i=0; i<2048; i++) {
        double value = m[i][j]; // do some calculations with value
}
```
That is, the order of the two for loops are reversed.

Weight: 2% (-0.5% for wrong answer)
Which is the most efficient option?

- [ ] 1
- [x] 2
- [ ] They are equally efficient

Maximum marks: 2

29 (2%) Latency & Bandwidth

What is the difference between **Latency** and **Bandwidth?**
Select one alternative:

- [ ] a) Latency reflects how long the total communication is, whereas bandwidth the communication rate
- [x] b) Latency reflects that time elapsing between a source starting to send data until the data starts arriving at destination, whereas bandwidth is the rate which a link can transmit data
- [ ] 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
- [ ] a) and b) above

Maximum marks: 2

30 (2%) 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:

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

Maximum marks: 2

31 PageRank

Larry Page and Sergey Brin developed PageRank at Stanford University in 1996 as part of a research project about a new kind of search engine. They later founded a company.

What was the name of the company?

The mathematics of PageRank are entirely general and apply to any graph or network in any domain. The eigenvalue problem behind PageRank's algorithm was independently rediscovered and reused in many scoring problems.

Mention at least two other use of PageRank (other than for search engines) and why parallelzing PageRank may or may not be important for that application.

Fill in your answer here

Format

Words: 0

Maximum marks: 4

32 PageRank and Cilk

Cilk, like OpenMP is a general purpuse paralle programming language that is based on C/C++ that allows for multithreading. OpenCilk 2.0 was released in July 2022.

In the following PageRank code, the variables contribution and rank are arrays of doubles, and in_degree and out_degree are arrays of integers. neighbor is a two dimensional array that stores the edges. neighbor[i][j] is the jth neighbor of the ith node.

```
1  void pagerank(double * rank, double * contribution,
2                int ** neighbor, int * in_degree,
3                int * out_degree, int num_vertices) {
4      for (size_t iter = 0; iter < 10; iter++) {
5          cilk_for (size_t i = 0; i < num_vertices; i++){
6              for (int j = 0; j < in_degree[i]; j++){
7                  rank[i] += contribution[neighbor[i][j]];
8              }
9          }
10         for (size_t i = 0; i < num_vertices; i++){
11             contribution[i] = rank[i]/out_degree[i];
12             rank[i] = 0.0;
13         }
14     }
15 }
```

For each of the following code modifications designed to improve performance, select the appropriate option to specify whether it is safe to make the indicated change, whether it is safe if a reducer is used, or whether it is unsafe. **Note: "safe" means that the output must be exactly the same as for the original code. You also get negative scores (-1.5/2) for each wrong answer.**

A document describing cilk_for is provided under PDF-dokument, but should not be need to answer correctly.

Replacing the **for** in line 4 with **cilk_for** is
Select one alternative

- Safe
- Unsafe ✔

Replacing the **for** in line 6 with **cilk_for** is
Select one alternative

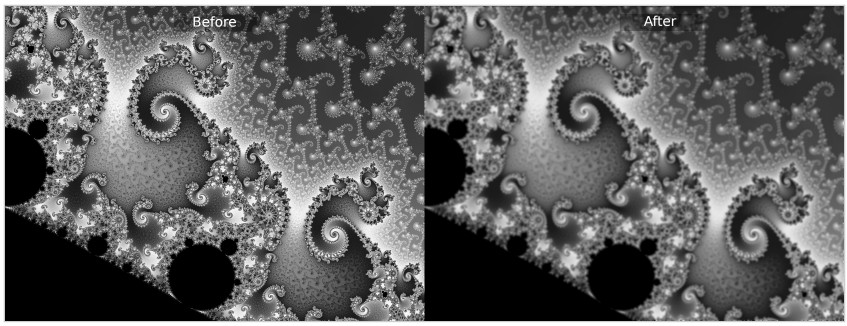
- Safe
- Unsafe ✔

Replacing the **for** in line 10 with **cilk_for** is
Select one alternative:

- Safe ✔
- Unsafe
- Only safe with reducer

Maximum marks: 6

33 CUDA Programming: Grayscale Image Convolution



The *blur* function given in the attached PDF iterates over a *GrayscaleImage* and updates the intensity values of each pixel with the average of itself and its eight neighbors.
$l(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} l(x+i, y+i)$

A picture before and after blurring the image over 20 iterations can be seen in the illustration:

Each pixel is an *unsigned char* with values ranging from 0 (black) to 255 (white).
Rewrite the **blur** function into a CUDA kernel named **blur_gpu**.

It should:
- Be callable from host
- Process **exactly one** pixel per thread.

```
1
```

Maximum marks: 10

34 Grayscale Blur: CUDA Grid Calculation

Assume thread block dimensions of $(32, 32, 1)$. Which of the following alternatives will calculate a grid size that is guaranteed to process the entire image of size *width* x *height*? Select **at most** three alternatives. You will also be given a negative point for each wrong answer.

Select one or more alternatives:

- [x] { floor((float) width / 32.0f, floor((float) height / 32.0f, 1 }
- [x] { ceil ((float) width / 32.0f), ceil((float) height / 32.0f, 1 }
- [ ] { width / 32, height / 32, 1 }
- [x] { width / 32 + 1, height / 32 + 1, 1 }
- [ ] { ceil(width / 32), ceil(height / 32), 1 }

Maximum marks: 3

35 Grayscale Blur: Grid Size Efficiency

Consider the previous question (Grayscale Blur: CUDA Grid Calculation).
Of the alternatives you picked, which method is the most efficient?
State the alternative and describe **briefly** why you think this is the case.
State your assumptions.

Fill in your answer here

Format

Words: 0

Maximum marks: 8

Grayscale Blur: Further Optimization (10pts)

To iteratively blur the image, the kernel may be called multiple times from the host, synchronizing and swapping image pointers between each iteration.

Instead of iteratively calling the kernel multiple times, we would like the kernel to be able to perform all iterations before terminating.

We would like to change this:

```
int main() {
    // ... setup ...
    for ( int i = 0; i < NUM_ITERATIONS; i++ ) {
        blur_gpu<<<grid_dim, thread_dim>>>(d_img, d_result);
        // ... error handling ...
        swap((void **) &d_img, (void **) &d_result);
    }
}
```

Into this:

```
int main() {
    // ... setup ...
    blur_gpu<<<grid_dim, thread_dim>>>(d_img, d_result, NUM_ITERATIONS);
}
```

How can the kernel be rewritten to perform all iterations on the GPU? (The kernel should process all iterations before terminating). Your answer should briefly summarize the change(s) you would have to make.

What challenges are you likely to encounter when rewriting the kernel? Shortly summarize the challenges and why they might cause problems if left unhandled. (Hint: There are dependencies between iterations.)

What could be used to remedy the challenge(s)? (Hint: Problem Set 5 - Graded CUDA)

The above will be scored as part of the next problem, where you may add further comment.

The following questions do not need to be answered to get a full score, but are meant for further thought. No extra points will be given!
Fill in your answer here

Does your solution impose constraints on your implementation? If so, shortly describe the constraint(s).

If you answered that there are constraints in your implementation. Shortly describe, in words, how the kernel can be rewritten such that it works for any image size.

What performance impact would your approach have? Shortly summarize both positive

and negative impacts.

36 CUDA Blurr grading

Grading of CUDA Blurr and additional comments you may have re. the CUDA problems in this section.

Fill any additional comments here

Format

Words: 0

Maximum marks: 9

37 Course reflections - multiple choice

In this problem we are curious about how you selected and reflected over the content and problems sets in this course. No points will be deducted, but rather 3 points given if you answer all of these.

We want to improve and really appreciate your honesty and will not count your answers against you. In fact, we will only look at this problem AFTER the rest of your exam is graded.

Which TWO of the followineg statements are the closest to your MAIN reasons for taking this course? Select TWO only.

- [ ] I found it online/in the course catalogue and found it interesting ✔
- [ ] A former student strongly recommended this course to me ✔
- [ ] This course seemed to have the most energetic and motivating instructor ✔
- [ ] It was the most interesting course taught in English ✔
- [ ] I like GPUs and wanted to learn more about CUDA ✔
- [ ] I wanted to take a course related to sustainability ✔
- [ ] I like courses with lots of programming ✔
- [ ] I have to take it as part of my major (e.g. Alg/HPC) ✔

What were the BEST 3 aspects of this course? Pick max. 3.

- [ ] Interesting problem sets that were highly relevant for sustainability. ✔
- [ ] Learning CUDA programming ✔
- [ ] In class instruction, not just videos /remote teaching ✔
- [ ] Learning about parallel computing, performance and energy efficiency ✔
- [ ] Great TA support ✔
- [ ] Graded problem sets that count 50% of final grade ✔
- [ ] Access to real paralle clusters and GPU hardware ✔
- [ ] Interesting problem sets, that were painful but taught me a lot. ✔
- [ ] Enthusiastic instructor ✔
- [ ] Clear syllabus and great on-line resources ✔

Before reading any further, how relevant do you think this course was with respect to sustainability

- [ ] Not relevant
- [ ] Somewhat relevant ✔
- [ ] A must-take course for people in STEM (science, technology, engineering or math) interested in sustainability! ✔
- [ ] Very relevant ✔
- [ ] Relevant ✔

Pick the aspect of the course that seems most relevant for sustainability

- [ ] HPC systems use a lot of electric power, so parallel computing is not really sustain ✔ e
- [ ] Problem set on shallow water equation that is highly relevant for sustainability ✔
- [ ] Optimizing for locality, reduces moving of data which is energy costly ✔
- [ ] Programming in parallel utilizes the computing systems better, so likely more energ ✔ efficient

What were the most interesting aspects and/or truthful statement regarding the Problem sets. Select one or more alternatives

- [ ] That they were graded so the exam only counts 50% which releaves a lot of stress ✔
- [ ] Liked learning MPI the most ✔
- [ ] Liked learning OpenMP/Pthreads the most ✔
- [ ] Working on the same problem, the shallow water equation, in several programming ✔ enviroments
- [ ] Like learning CUDA the most ✔
- [ ] The shallow water equations since they relate so closely to sustainability ✔
- [ ] The problems sets t
- [ ] The shallow water equations since they relate so closely to other numerical simulat ✔ s I expect to work with

Select all the statements that you think mostly aligns with your opinion of TDT4200.

- [ ] Prefer to study on my own, so did not attend to most classes
- [ ] Would probably not recemmend this class. Content was not as expected
- [ ] I may recommend this class --- Will have to see how this exams goes...
- [ ] Clear material available -- never felt like I had to come to class
- [ ] Did not attend many classes due to time clash with another course
- [ ] Yes, love courses with enthusiastic instructors that make me feel like coming to class
- [ ] Instructor talked too fast.
- [ ] The problem sets took too much time compared to what I got out of them
- [ ] Too much work / time-consuming problem sets!
- [ ] Did not attend most classes due to work conflicts
- [ ] Would have been nice if the lectures were at a different time so I could have attended class more often
- [ ] I wish this class had been taught in Norwegian
- [ ] I learned a lot. Love the power of GPUs and would recomment this class ✔

Maximum marks: 3

38 Bonus question and comments

Comment on how useful was TDT4200 regarding making you aware that computing is related to sustainability.
Also add any further comments or quesitions regarding this exam below.
Please add your reflections here. You can get 2 bonus points for this.

Format

Words: 0

Maximum marks: 2

Document 1
Attached

FRONT PAGE – TDT4200 FINAL, FALL 2022

Department of Computer Science

Examination paper for TDT4200, Parallel Computing

Examination date: Dec. 12, 2022

Examination time (from-to): 15:00 – 19:00

Permitted examination support material: None – only aids provided via Inspera

Academic contact during examination: Prof. Anne C. Elster
Phone: 981 02 638

Academic contact present at the exam location: YES, ca 16:00 (4pm) and 18:00

OTHER INFORMATION

Get an overview of the question set before you start answering the questions.

Read the questions carefully and make your own assumptions. If a question is unclear/vague, make your own assumptions and specify them in your answer. Only contact academic contact in case of errors or insufficiencies in the question set. Address an invigilator if you wish to contact the academic contact. Write down the question in advance.

Weighting:
This exam will count 50% towards your final TDT4200 grade (scaled with the problem sets).

The exam problems are weighted and the maximum score possible on this final is 100pts or % assuming we do not need to discount a problem due to late-discovered issues.

ANSWERING TRUE/FALSE QUESTIONS WRONG MAY RESULT IN A NEGATIVE SCORE but will give a total score no lower than 0 on Section 1.

Notifications: If there is a need to send a message to the candidates during the exam (e.g. if there is an error in the question set), this will be done by sending a notification in Inspera. A dialogue box will appear. You can re-read the notification by clicking the bell icon in the top right-hand corner of the screen.

Withdrawing from the exam: If you become ill or wish to submit a blank test/withdraw from the exam for another reason, go to the menu in the top right-hand corner and click “Submit blank”. This cannot be undone, even if the test is still open.

Access to your answers: After the exam, you can find your answers in the archive in Inspera. Be aware that it may take a working day until any hand-written material is available in the archive.

Question 32
Attached

< Table Of Contents

cilk_for

A `cilk_for` loop is a replacement for the normal C/C++ for loop that permits loop iterations to run in parallel.

The general cilk_for syntax is:

```
cilk_for (declaration;
                  conditional expression;
                  increment expression)
body
```

The following applies:

- The declaration must declare and initialize a single variable, called the control variable. The constructor's syntactic form does not matter. If the variable type has a default constructor, no explicit initial value is needed.
- The conditional expression must compare the control variable to a "termination expression" using one of the following comparison operators: `< <= != >= >`
- The termination expression and control variable can appear on either side of the comparison operator, but the control variable cannot occur within the termination expression. The termination expression value must not change from one iteration to the next.
- The increment expression must add to or subtract from the control variable using one of the following supported operations:
  - `+=`
  - `-=`
  - `++` (prefix or postfix)
  - `--` (prefix or postfix)

  The value added to (or subtracted from) the control variable, like the loop termination expression, must not change from one iteration to the next.

The runtime converts a `cilk_for` loop into an efficient divide-and-conquer recursive traversal over the loop iterations.

Sample `cilk_for` loops include:

```
cilk_for (int i = begin; i < end; i += 2)
     f(i);
cilk_for (T::iterator i(vec.begin()); i != vec.end(); ++i)
     g(i);
```

In C, but not C++, the loop control variable can be declared in advance:

```
 int i;
 cilk_for (i = begin; i < end; i += 2)
      f(i);
```

The serialization of a valid Intel® Cilk™ Plus program has the same behavior as the similar C/C++ program, where the serialization of `cilk_for` is the result of replacing `cilk_for` with for. Therefore, a `cilk_for` loop must be a valid C/C++ for loop, but `cilk_for` loops have several constraints compared to C/C++ for loops.

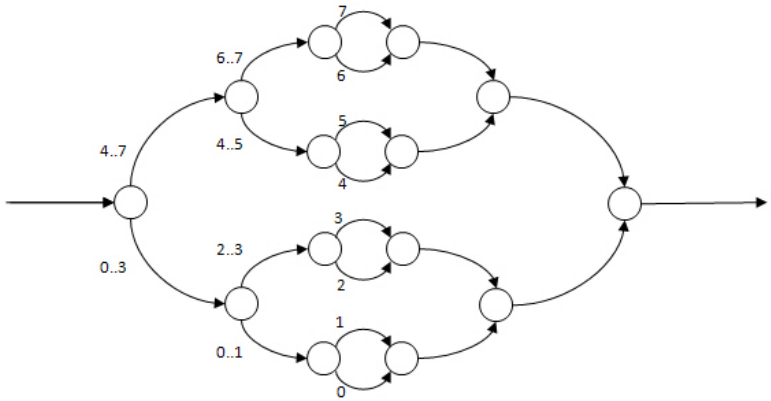
Since the loop body is executed in parallel, it must not modify the control variable nor should it modify a nonlocal variable, as that would cause a **data race**. A reducer can often be used to prevent a race.

Serial/parallel structure of cilk_for

Using `cilk_for` is not the same as spawning each loop iteration. In fact, the Intel compiler converts the loop body to a function that is called recursively using a divide-and-conquer strategy; this provides

significantly better performance. The difference can be seen clearly in the Directed Acyclic Graph (DAG) for the two strategies.

First, consider the DAG for a `cilk_for`, assuming N=8 iterations and a grain size of 1. The numbers label the serial sequence of instructions, known as **strands**; these numbers indicate which loop iteration is handled by each strand.

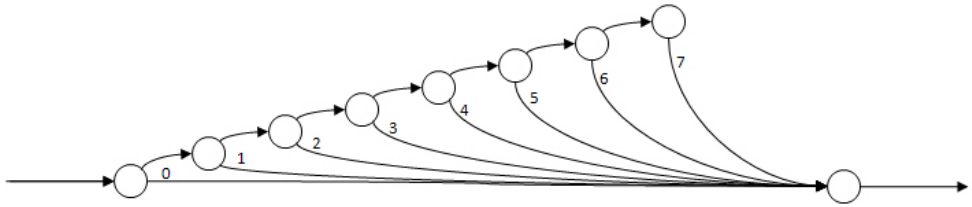


At each division of work, half of the remaining work is done in the child and half in the continuation. Importantly, the overhead of both the loop itself and of spawning new work is divided evenly along with the cost of the loop body.

If each iteration takes the same amount of time T to execute, then the **span**, which is the most expensive path extending from the beginning to the end of the program, is $\log_2(N) * T$, or $3 * T$ for 8 iterations. The run-time behavior is well balanced regardless of the number of iterations or number of workers.

Serial/parallel structure when spawning within a serial loop

Here is the DAG for a serial loop that spawns each iteration. In this case, the work is not well balanced, because each child does the work of only one iteration before incurring the scheduling overhead inherent in entering a sync. For a short loop, or a loop in which the work in the body is much greater than the control and spawn overhead, there will be little measurable performance difference. However, for a loop of many cheap iterations, the overhead cost will overwhelm any advantage provided by parallelism.



cilk_for body

The body of a `cilk_for` loop defines a special region that limits the scope of `cilk_for` and `cilk_sync` statements within it. A `cilk_sync` statement within a `cilk_for` waits for completion only of the children that were spawned within the same loop iteration. It will not sync with any other iteration, nor will it sync with any other children of the surrounding function. In addition, there is an implicit `cilk_sync` at the end of every loop iteration (after block-scoped variable destructors are invoked). As a result, if a function has outstanding

children when entering a `cilk_for` loop, it will have the same outstanding children when exiting the `cilk_for` loop. Any children that were spawned within the `cilk_for` loop are guaranteed to have synchronized before the loop terminates. Conversely, none of the children that existed before entering the loop will be synchronized during loop execution. This quality of a `cilk_for` loop can be used to your advantage (see Exception Handling).

If an exception is thrown from within a `cilk_for` loop body (and not handled within the same iteration), then some of the loop iterations may not run. Unlike a serial execution, it is not completely predictable which iterations will run and which will not. No iteration (other than the one throwing the exception) is aborted in the middle.

Windows OS: There are restrictions when using Microsoft structured exception handling (specifically, the `/EHa` compiler option and the `__try, __except, __finally` and `__leave` extensions to C/C++). See Windows* Structured Exception Handling in Exception Handling.

cilk_for Type Requirements

With care, you may use custom C++ data types for the `cilk_for` control variable. For each custom data type, you need to provide some methods to help the runtime system compute the loop range size so that it can be divided. Types such as integer types and STL random-access iterators have an integral difference type already, and so require no additional work.

Suppose the control variable is declared with type `variable_type` and the loop termination expression has type `termination_type`, as shown here:

```
extern termination_type end;
extern int incr;
cilk_for (variable_type var; var != end; var += incr) ;
```

You must provide one or two functions to tell the compiler how many times the loop executes; these functions allow the compiler to compute the integer difference between `variable_type` and `termination_type` variables:

```
difference_type operator-(termination_type, variable_type);
difference_type operator-(variable_type, termination_type);
```

The following applies:

- The argument types need not be exact, but must be convertible from `termination_type` or `variable_type`.
- The first form of operator- is required if the loop could count up; the second is required if the loop could count down.
- The arguments may be passed by const reference or value.
- The program will call one or the other function at runtime depending on whether the increment is positive or negative.
- You can pick any integral type as the `difference_type` return value, but it must be the same for both functions.
- It does not matter if the `difference_type` is signed or unsigned.

Also, you need to tell the system how to add to the control variable by defining:

```
variable_type::operator+=(difference_type);
```

If you wrote `-=` or `--` instead of `+=` or `++` in the loop, define `operator-=` instead of `operator+=`.

These operator functions must be consistent with ordinary arithmetic. The compiler assumes that adding one twice is the same as adding two once, and if

```
X - Y == 10
```

then


```
Y + 10 == X
```

cilk_for Restrictions

In order to parallelize a loop using the divide and conquer technique, the runtime system must pre-compute the total number of iterations and must be able to pre-compute the value of the loop control variable at every iteration. To enable this computation, the control variable must act as an integer with respect to addition, subtraction, and comparison, even if it is a user-defined type. Integers, pointers, and random access iterators from the standard template library all have integer behavior and thus satisfy this requirement.

In addition, a `cilk_for` loop has the following limitations, which are not present for a standard C/C++ for loop. The compiler will report an error or warning for violations of the following.

- There must be exactly one loop control variable, and the loop initialization clause must assign the value. The following form is not supported:
  ```
  cilk_for (unsigned int i, j = 42; j < 1; i++, j++)
  ```
- The loop control variable must not be modified in the loop body. The following form is not supported:
  ```
  cilk_for (unsigned int i = 1; i < 16; ++i) i = f();
  ```
- The termination and increment values are evaluated once before starting the loop and will not be re-evaluated at each iteration. Therefore, modifying either value within the loop body will not add or remove iterations. The following form is not supported:
  ```
  cilk_for (unsigned int i = 1; i < x; ++i) x = f();
  ```
- In C++, the control variable must be declared in the loop header, not outside the loop. The following form is supported for C, but not C++:
  ```
  int i; cilk_for (i = 0; i < 100; i++)
  ```
- A break or return statement will not work within the body of a `cilk_for` loop; the compiler will generate an error message. break and return in this context are reserved for future speculative parallelism support.
- A goto can only be used within the body of a `cilk_for` loop if the target is within the loop body. The compiler will generate an error message if there is a goto transfer into or out of a `cilk_for` loop body. Similarly, a goto cannot jump into the body of a `cilk_for` loop from outside the loop.
- A `cilk_for` loop may not "wrap around." For example, in C/C++ you can write: `for (unsigned int i = 0; i != 1; i += 3);`
  and this has well-defined, if surprising, behavior. It means execute the loop 2,863,311,531 times. Such a loop produces unpredictable results when converted to a `cilk_for` loop.
- A `cilk_for` loop may not be an infinite loop such as:
  ```
  cilk_for (unsigned int i = 0; i != i; i += 0);
  ```

cilk_for Grain Size

The `cilk_for` statement divides the loop into chunks containing one or more loop iterations. Each chunk is executed serially, and is spawned as a chunk during the execution of the loop. The maximum number of iterations in each chunk is the **grain size**.

In a loop with many iterations, a relatively large grain size can significantly reduce overhead. Alternately, with a loop that has few iterations, a small grain size can increase the parallelism of the program and thus improve performance as the number of processors increases.

Setting the Grain Size

Use the `cilk grainsize` pragma to specify the grain size for one `cilk_for` loop:

```
#pragma cilk grainsize = expression
```

For example, you can write:

```
#pragma cilk grainsize = 1

cilk_for (int i=0; i<IMAX; ++i) { . . . }
```

If you do not specify a grain size, the system calculates a default that works well for most loops. The default value is set as if the following pragma were in effect:

```
#pragma cilk grainsize = min(512, N / (8*p))
```

where N is the number of loop iterations, and p is the number of workers created during the current program run. This formula will generate parallelism of at least 8 and at most 512. For loops with few iterations (less than 8 * workers) the grain size will be set to 1, and each loop iteration may run in parallel. For loops with more than (4096 * p) iterations, the grain size will be set to 512.

If you specify a grain size of zero, the default formula will be used. The result is undefined if you specify a grain size less than zero.

The expression in the pragma is evaluated at run time. For example, here is an example that sets the grain size based on the number of workers:

```
#pragma cilk grainsize = n/(4*__cilkrts_get_nworkers())
```

Loop Partitioning at Run Time

The number of chunks that are executed is approximately the number of iterations N divided by the grain size K.

The Intel compiler generates a divide and conquer recursion to execute the loop. In pseudo-code, the control structure looks like this:

```
void run_loop(first, last)
{
if (last - first) < grainsize)
{
   for (int i=first; i<last ++i) LOOP_BODY;
}
else
{
   int mid = (last-first)/2;
   cilk_spawn run_loop(first, mid);
   run_loop(mid, last);
  }
}
```

In other words, the loop is split in half repeatedly until the chunk remaining is less than or equal to the grain size. The actual number of iterations run as a chunk will often be less than the grain size.

For example, consider a `cilk_for` loop of 16 iterations:

```
cilk_for (int i=0; i<16; ++i) { ... }
```

With grain size of 4, this will execute exactly 4 chunks of 4 iterations each. However, if the grain size is set to 5, the division will result in 4 unequal chunks consisting of 5, 3, 5 and 3 iterations.

If you work through the algorithm in detail, you will see that for the same loop of 16 iterations, a grain size of 2 and 3 will both result in exactly the same partitioning of 8 chunks of 2 iterations each.

Selecting a Good Grain Size Value

The default grain size usually performs well. Use the following guidelines to select a different value:

- If the amount of work per iteration varies widely and if the longer iterations are likely to be unevenly distributed, it might make sense to reduce the grain size. This will decrease the likelihood that there is a time-consuming chunk that continues after other chunks have completed, which would result in idle workers with no work to steal.

- If the amount of work per iteration is uniformly small, then it might make sense to increase the grain size. However, the default usually works well in these cases, and you don't want to risk reducing parallelism.
- If you change the grain size, carry out performance testing to ensure that you've made the loop faster, not slower.

Parent topic: Intel(R) Cilk(TM) Plus Keywords

Question 33
Attached

1 Starting Code

```c
///< Contains all the information about a grayscale image.
typedef struct {
    int width;
    int height;
    unsigned char *data;
} GrayscaleImage;

/**
 * Performs a convolution over the image, averaging the surrounding pixels.
 * @param image The image to blur.
 * @param result The resulting output image.
 */
void blur(GrayscaleImage *image, GrayscaleImage *result)
{
    int width = image->width;
    int height = image->height;
    unsigned char *data = image->data;
    unsigned char *result_data = result->data;

    // Iterate over image height, excluding pixels at the boundary.
    for ( int y = 1; y < height - 1; y++ )
    {
        // Iterate over the image width, excluding pixels at the boundary.
        for ( int x = 1; x < width - 1 ; x++ )
        {
            unsigned int sum = 0;

            // The index of the pixel at coordinates (i,j)
            size_t pixel_index = y * width + x;

            // k denotes the y offset from the current pixel's y-coordinate.
            // iterating from -1 to 1 will include the current row and the rows above and below.
            for ( int k = -1; k <= 1; k++ )
            {
                sum += data[pixel_index + k * width - 1];
                sum += data[pixel_index + k * width];
                sum += data[pixel_index + k * width + 1];
            }

            float divisor = 1.0f / 9.0f;

            // Write the resulting pixel's value to the output image.
            result_data[pixel_index] = (unsigned char) (divisor * sum);
        }
    }
}
```