CHALMERS TEKNISKA HÖGSKOLA
Institutionen för data- och informationsteknik
Avdelningen för dator- och nätverkssystem

Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Friday, October 27th, 2023, 14:00h - 18:00h

---

Teacher/Lärare: Miquel Pericas, tel:+46-31-772-17-05. Exam hall visits by teaching staff are planned approx. 1h and 3h after examination start.

Language/Språk: Answers shall be given in English.

Solutions/Lösningar: Solutions will be posted on Friday, November 3rd, 2023 on the Canvas page.

Exam review/Granskning: The review date will be posted on the course Canvas page by the time you receive the email from LADOK.

Aids/Hjälpmedel: Only Chalmers-authorised aids are allowed during examination. This includes pencils, erasers, rulers and dictionaries. Electronic dictionaries or graphing calculators are not allowed.

Grades:

Chalmers				
Points	0-23	24-35	36-47	48-60
Grade	Failed	3	4	5

GU				
Points	0-23	24-41	42-60	
Grade	Failed	G	VG	

Problem 1 (10 points)

In this problem we ask you to reason about how user level threads and kernel level threads impact performance, how threading potentially impacts MPI processing, and how CUDA threads relate to these concepts.

Tasks:

(a) Assume a multicore system on which we want to run a parallel application. The runtime library manages user level threads (ULT) with a context switch overhead of 1 time unit. Threads managed by the operating system (kernel level threads, KLT) have a context switch overhead of 5 units, which can be triggered either by I/O (disk, network, etc.) or because all ULTs assigned to a running KLT have completed. The total work of the application is 1000 tasks, such that each task takes 10 units. The application is to be executed on 10 cores. Each ULT executes a single task. The developers consider the following three models: (1) `N:1` (i.e. all ULTs mapped to 1 KLT), (2) `1:1` (each ULT mapped to a different KLT), and `N:M`, with a 10:1 ratio (i.e. 10 ULTs mapped to 1 KLT). What are the execution times for all these three cases?

(b) The program is modified so that each task now includes a checkpointing stage at the end of the computation. The time it takes to perform the checkpoint is 10 time units. Checkpointing consists in writing the tasks' data to disk. Reason about the suitability of the three threading models `N:1`, `1:1` and `N:M` (with both 100:1 and 10:1 ratios).

(c) Instead of checkpointing, the developers now intend to use MPI in order to delegate checkpointing to a different MPI process running on a different node. Assume an MPI library that has been written with only one kernel level thread (KLT) in mind. Discuss how the various threading models (`1:1`, `N:M` and `N:1`) potentially impact the program. Depending on the case, will it be necessary to modify the MPI library or the application?

(d) Now let's consider the CUDA threading model. CUDA supports very fine-grained threads. How do CUDA threads compare to ULTs/KLTs? How does CUDA effectively support the fine thread granularity? Explain also in 1-2 lines how the CUDA threading model relates to SIMD.

Problem 2 (10p)

Consider the following example of a nested loop with depth 2.

```
#define N 64
for(int j=0;j<N;j++)
{
    for(int k=0;k<N;k++)
    {
        a[j*N+k] = (j+1)*N+k;
    }
}
```

Tasks:

(a) Consider the following schedulers: (a) static scheduling, dynamic scheduling with (b) "self-scheduling", and (c) "chunk-scheduling with a chunk size of 6", and (d) "guided scheduling with a minimum chunk size of 4". The above loop is to be executed on a given system with 8 processors. Considering the above schedulers applied to a parallelization of the outer loop. How many time units will it take to complete the given code? Assume that each iteration takes 2.0 time units to complete.

(b) Next assume the programmers have bought a system with 100 processors. Propose a loop transformation for the above code to offer enough parallelism to this new system with 100 processors. Show the transformed code.

(c) Now assume that a fixed overhead is paid to create the parallel loop (the transformed loop from part (b)). This overhead includes the generation of all the threads needed for the parallel execution, and the destruction of those threads after the execution. The overhead depends on the number of processors (P), and is $P \times 0.5$ time units. Considering the case of static scheduling: after which value of P does adding new processors result in a slowdown instead of a speed-up?

Problem 3 (10p)

Now consider the following matrix-matrix multiplication code:

```
#define M 100
#define N 100
#define K 100
void gemm( float *A, float *B, float *C)
{
    int i,j,k;
    for(i = 0; i < M; ++i){
        for(k = 0; k < K; ++k){
            for(j = 0; j < N; ++j){
                C[i*N+j] +=  A[i*K+k]*B[k*N+j];
            }
        }
    }
}
```

Tasks:

(a) What is the Arithmetic Intensity of this code? Consider a multicore chip in which each core has a performance of 4 GFLOPS (single precision) and a shared memory bus of 16GB/s. Using the simplified DRAM roofline model, what will be the performance (i.e. the execution time) if the code is run on a single core or on 8 cores? Note: a single `float` consists of 4 bytes.

(b) By the use of SIMD, the performance of each core is multiplied by four. In this new regime, what is the next bottleneck that should be addressed: FLOPS or Memory (GB/s)? Explain your reasoning. Further, assume that the developers have bought a 32-core machine, with each core having the same SIMD unit. Will that change the bottleneck that should be addressed?

(c) Let's assume the above code is included in a program that has a serial component that takes 0.5 ms. Consider the multi-core chip from part (a). What is the upper limit to the speed-up (compared to a single core) according to Amdahl's law? For the 8 core system, compute the speed-up over a single core. Discuss, individually, the suitability of (i) executing the kernel on even more cores, (ii) optimizing the kernel to achieve a higher AI, or (iii) optimizing the serial part of the computation.

Problem 4 (10p)

The following two code listings show two different loops.

Loop 1:

```
for(int i = 0; i < N; i++)
{
    A[i] += A[N-1-i];
}
```

Loop 2:

```
int quad = 0

int N = 1000;
int range = 1;
float dx = (float)range/(float) N;
float* x = (float *) malloc(sizeof(float)*N);
x[0] = 0;
float temp = 0;

for(int i = 1; i < N; i++)
{
    x[i] = i* dx;
    temp = dx/(1+x[i]);

    quad += temp;
}
```

Tasks:

(a) Can Loop 1 be vectorized? why? why not?

(b) Based on your answer for task (a), apply the relevant changes to the code and add the relevant OpenMP constructs to vectorize the loop.

(c) Loop 2 presents the numerical integration $I = \int_0^1 \frac{1}{1+x} dx = \log(2)$. List three methods of parallelizing the given code using OpenMP. Write the pragmas you would use for these methods, along with the data sharing attributes for the different variables.

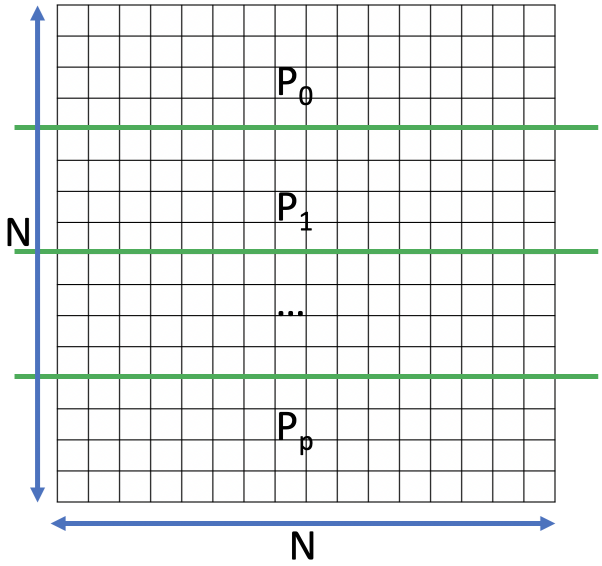
(d) How do you expect the performance of the three methods listed in task (c) to differ? For each of the methods, briefly describe one way in which the OpenMP compiler could implement these methods.

Problem 5 (10p)

For this problem consider a set of P processors, each on a separate node, and without shared memory (i.e. with distributed memory).

Tasks:

(a) Consider the matrix shown below, and its partitioning over the set of P processors. The matrix is initially stored in the DRAM memory of one processor. Give at least two different ways to distribute the different blocks of the matrix to the P different processors. The answer should be in the form of a pseudo-code.



(b) Next consider an array of M = 10000 integers (type `int`). Write an MPI code, to be run on P processors, which counts the number of zeros (0) in the given array. While doing so, you should minimize the data communication between the P processes.

(c) Finally, consider an MPI program which uses the MPI communication primitives *SSend* and *SSrecv*. We know that this program does not deadlock. Later on these are changed to non-blocking *Isend* and *Irecv* with *Wait* calls added immediately after each *Isend* and *Irecv*. Will the new program deadlock from communication? Explain your reasoning.

A subset of MPI calls that are useful for this problem are shown below:

```
1  int MPI_Send (const void *buf, int count, MPI_Datatype datatype,
2      int dest, int tag, MPI_Comm comm)
3  int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
4      int dest, int tag, MPI_Comm comm, MPI_Status *status)
5  int MPI_Comm_rank(MPI_Comm comm, int *rank)
6  int MPI_Comm_size(MPI_Comm comm, int *size)
7  int MPI_Barrier( MPI_Comm comm )
8  int MPI_Bcast(void* data, int count, MPI_Datatype datatype,
9          int root, MPI_Comm communicator)
10 int MPI_Gather(const void *sendbuf, int sendcount,
11         MPI_Datatype sendtype, void *recvbuf, int recvcount,
12         MPI_Datatype recvtype, int root, MPI_Comm comm)
13 int MPI_Allgather(const void *sendbuf, int sendcount,
14         MPI_Datatype sendtype, void *recvbuf, int recvcount,
```

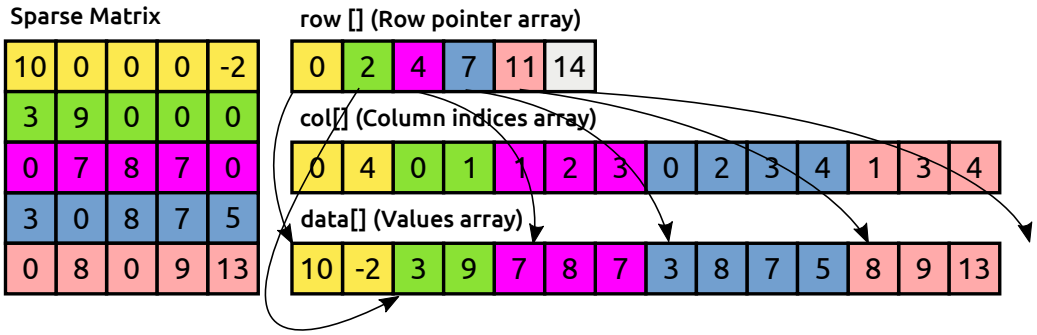
```
15          MPI_Datatype recvtype, MPI_Comm comm)
16  int MPI_Alltoall(const void *sendbuf, int sendcount,
17          MPI_Datatype sendtype, void *recvbuf, int recvcount,
18          MPI_Datatype recvtype, MPI_Comm comm)
19  int MPI_Scatter(const void *sendbuf, int sendcount,
20          MPI_Datatype sendtype, void *recvbuf, int recvcount,
21          MPI_Datatype recvtype, int root, MPI_Comm comm)
22  int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
23          MPI_Datatype datatype, MPI_Op op, int root,
24          MPI_Comm comm)
```

Problem 6 (10p)

The following CUDA code implements a Sparse Matrix Vector product, i.e. the product of a sparse matrix (a matrix that contains many zero elements) and a dense vector. To effectively store the matrix, all non-zero values are held in a vector `data[]`, sorted by rows and columns. The vector `row[]` contains indices into the vector `data[]` such that `row[x]` points to the first non-zero element of row 'x'. For example, `row[1]` points to the first non-zero value of row 1, while `row[2]-1` points to the last non-zero value. If row 1 contains no non-zero values, then `row[1] == row[2]`. Similarly, `col[]` holds the column indices corresponding to each value of `data[]`. See also the figure on the next page. In the below kernel code, each thread computes one row of the output vector `y[]`.

```
1  __global__ void spmv(int *row, int *col, float *data, float *x, float *y, int N) {
2      int tid = blockIdx.x * blockDim.x + threadIdx.x; // row ID
3      if (tid < N) {
4          float dot = 0.0;
5          for (int i = row[tid]; i < row[tid + 1]; i++) {
6              dot += data[i] * x[col[i]];
7          }
8          y[tid] = dot;
9      }
10 }
11 // Sparse Matrix-Vector Product: y[] = A[][] * x[]; A[][] is sparse, x[],y[] are dense
12 int main() {
13     int N;  // number of rows
14     int M;  // number of columns
15     int row[] = {...}; // Row pointer array
16     int col[] = {...}; // Column indices
17     float data[] = {...}; // Non-zero values
18     float x[M] = {...}; // Input vector
19     float y[N] = {...}; // Output vector
20
21     int *d_row, *d_col;
22     float *d_data, *d_x, *d_y;
23
24     // Allocate memory on the device
25     cudaMalloc(A, (N + 1) * sizeof(int));
26     cudaMalloc(B, sizeof(col));
27     cudaMalloc(C, sizeof(data));
28     cudaMalloc(D, M * sizeof(float));
29     cudaMalloc(E, N * sizeof(float));
30
31     // Copy data to device
32     cudaMemcpy(F, (N + 1) * sizeof(int), cudaMemcpyHostToDevice);
33     cudaMemcpy(G, sizeof(col), cudaMemcpyHostToDevice);
34     cudaMemcpy(H, sizeof(data), cudaMemcpyHostToDevice);
35     cudaMemcpy(I, N * sizeof(float), cudaMemcpyHostToDevice);
36
37     // Perform SpMV
38     spmv<<<J>>>(K);
39
40     // Copy the result back to the host
41     cudaMemcpy(L, N * sizeof(float), cudaMemcpyDeviceToHost);
42
43     // Free memory on the device
44     cudaFree(d_row);     cudaFree(d_col);
45     cudaFree(d_data);    cudaFree(d_x);
46     cudaFree(d_y);
47     return 0;
48 }
```

The clarify the storage format, the following figure shows a sample sparse matrix and its representation as the three arrays `data[]`, `row[]` and `col[]`.



Tasks:

(a) Complete the code of the `main()` function by filling the values of A through L.

(b) Next we want to use shared memory to improve the kernel. Identify which of the vectors offers temporal locality. Show how to modify the above CUDA kernel code to make use of shared memory. For this task you should assume that the length of the vector `x[]` (same as the number of columns of the matrix) is no more than 1024 .

(c) How would you change the code if the length of vector `x[]` is above 1024 but still fitting into shared memory? Provide the CUDA code the implements this modification

(d) How should you change the code if vector `x[]` is so large that it does not fit into the shared memory? Explain either in words or pseudocode.

(e) Is it possible to use CUDA streams to overlap computation and communication in the previous code? Explain how the code would need to be changed.