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

Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Thursday, October 31st, 2024, 8:30h - 12:30h

---

Teacher/Lärare: Hari Abram, tel:+46 76 752 03 85. Exam hall visits by teaching staff are planned approximately 1h and 3h after examination start.

Examiner: Miquel Pericàs, tel: +46 31 772 17 05, miquelp@chalmers.se

Language/Språk: Answers shall be given in English.

Solutions/Lösningar: Solutions will be posted on Friday, November 1st, 2024 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-authorized aids are allowed during examination. This includes pencils, erasers, rulers, and dictionaries. Electronic dictionaries and graphing calculators are not allowed. Remember that you can leave your answers in terms of mathematical expressions.

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 (16 points)

This problem covers theoretical concepts related to parallel programming models.

(a) The three programming models introduced in DAT400 (OpenMP, MPI, CUDA) support some form of data parallelism. Briefly describe what is data parallelism and what is its main advantage. Next, list one challenge to exploit (i) task parallelism, (ii) pipeline parallelism, and (iii) instruction level parallelism (list one challenge for each type of parallelism).

(b) SIMT (Single Instruction Multiple Threads) is a variation of SPMD (Single Program Multiple Data) used in CUDA GPUs. In up to two sentences each, describe (1) how does CUDA SIMT differ from the SPMD implementations in OpenMP and MPI, and (2) how is SIMT related to SIMD (Single Instruction Multiple Data).

(c) An application is given that is composed of several parallel loops, as shown in the code snippet below. The application is to be run on a multicore system with only private per-core caches. The iterations in the loops have different execution time. Across loops, corresponding iterations process the same data. Initially a static schedule was used to parallelize, but the programmer discovered that it led to load imbalance. To address the problem, a dynamic schedule (`#pragma omp parallel for schedule(dynamic)`) has been used instead. To their surprise, they observe that the total execution time actually degrades. Propose two potential causes for the performance degradation resulting from the dynamic schedule.

```
int i;
DATA_t data[N]; // N is very large
#pragma omp parallel for schedule(dynamic)
for(i = 0; i < N; i++) // loop 1
    func1(&data[i]); // Exec time depends on data[i]

#pragma omp parallel for schedule(dynamic)
for(i = 0; i < N; i++) // loop 2
    func2(&data[i]); // Exec time depends on data[i]

#pragma omp parallel for schedule(dynamic)
for(i = 0; i < N; i++) // loop 3
    func3(&data[i]); // Exec time depends on data[i]

....
```

(d) An HPC company has decided to implement the CUDA programming model using User-Level Threads (ULTs) so that they can run CUDA applications efficiently on a shared memory multiprocessor. List one general challenge of working with ULTs. Next, list two specific challenges of implementing CUDA using ULTs. Assume the N:M model of thread execution.

(e) Assume an application that adds three arrays `A[]`, `B[]`, `C[]` and stores the result in a fourth array `D[]`, i.e. `D[] = A[] + B[] + C[]`. The input data is initially located in disk, in two possible formats described by the two C structs shown in Figure 1 and Figure 2.

The application is to be run on a distributed memory multiprocessor system with four multicore nodes. Each multicore node has its own local memory, and the four nodes are interconnected by a message passing network. To execute the addition, the data

```
struct{
    double A[SIZE];
    double B[SIZE];
    double C[SIZE];
    double D[SIZE];
} data;
```

Figure 1: Structure of Arrays (SoA)

```
struct {
    double A;
    double B;
    double C;
    double D;
    } data[SIZE];
```

Figure 2: Array of Structures (AoS)

needs to be loaded into each node's local memory. For each of the two data arrangements (AoS and SoA), assess the suitability of the following data distribution methods: blockwise, cyclic, or block-cyclic. Consider that data can only be distributed at a granularity of a full C `struct`, both across nodes and within nodes (i.e. across the cores of the multicore node).

Problem 2 (7 points)

Consider the following code that computes the Floyd-Warshall algorithm, which finds the shortest paths between all pairs of vertices in a weighted graph represented by an adjacency matrix `dist`.

```
void floyd_warshall(int dist[N][N]) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            for (int i = 0; i < N; i++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}
```

Tasks:

(a) Discuss cache performance issues associated with the current implementation of the Floyd-Warshall algorithm, focusing on memory access patterns and their impact on execution time.

(b) One optimization to consider is loop unrolling, unroll the innermost loop two times and write down the pseudo-code.

(c) List at least two loop optimisations that can be performed on this code, other than loop unrolling, and specify their advantages in a single sentence. Apply the optimizations in the simplest possible way and write down the resulting pseudo-code.

Problem 3 (11 points)

Consider a scientific computing application that processes a large dataset and consists of both parallelizable and non-parallelizable components.

Tasks:

(a) The application spends 80% of its time performing an algorithm shown below (which is parallelizable) and 20% on sequential data preparation and result analysis. Using Amdahl's law, calculate the theoretical maximum speedup achievable for this program. If using an upgraded hardware improves the algorithm speed by a factor of 6x, what would be the overall speedup according to Amdahl's law?

(b) Cost and efficiency are two key metrics used to assess the performance of parallel programs. The serial execution time of the application is given as Z seconds. A programmer runs the application on an 8-core machine, achieving an execution time of K seconds. Calculate the cost and efficiency based on these values.

Consider the algorithm implementation (Note: a `float` is 4 bytes):

```
1  #define M 10000
2  #define N 10000
3
4  void algorithm(float *A, float *B, float *x, float *y) {
5      for (int i = 0; i < M; i++) {
6          for (int j = 0; j < N; j++) {
7              y[i * N + j] += A[i * N + j] * x[j] + B[j];
8          }
9      }
10 }
```

(c) Calculate the arithmetic intensity of this algorithm implementation.

(d) Assume a multicore processor where each core has a performance of 8 GFLOPS (single precision) and a shared memory bus of 32 GB/s. Using the simplified DRAM roofline model, estimate the execution time if this algorithm runs on 1 core and on 8 cores. According to your results, what are the primary bottlenecks: FLOPS or Memory bandwidth of running on 1 core and 8 cores respectively? Regarding the execution time calculation, you can simply provide the formula.

(e) If the shared memory bus bandwidth is doubled to 64 GB/s, how would this affect the performance of the algorithm on 1 core and 8 cores? Recalculate the execution times and discuss the implications. What strategies can be used to improve the performance based on your analysis of the results?

Problem 4 (20 points)

This problem deals with parallelization of the "power iteration", an algorithm to find the eigenvalues and eigenvector of a diagonalizable square (NxN) matrix A. A sequential C code is shown below. The code first multiplies `matrix[N][N]` with `vector[N]` (matrix-vector product). The product `result[N]` is then normalized and copied back into `vector[N]`, to be used for the next iteration, unless the convergence check is satisfied or a maximum number of iterations is reached. The results are returned in `eigenvector[N]` and `eigenvalue`.

The goal of the problem is to devise MPI and CUDA versions of the program. The MPI and CUDA parts are independent of each other, and can be solved independently. To simplify the parallelization across MPI nodes and CUDA device, we will accelerate only on the matrix-vector product. All other operations are to be performed by one processor (either the *rank 0* in MPI, or the *host* in CUDA).

```
1  void power_iteration(double matrix[N][N], double vector[N], double *eigenvalue, double
       eigenvector[N]) {
2      double result[N], norm;
3      int iter, i, j;
4
5      for (i = 0; i < N; i++) {    // Initialize the vector with arbitrary values
6          vector[i] = 1.0;
7      }
8
9      for (iter = 0; iter < MAX_ITER; iter++) {
10         // Matrix-vector multiplication
11         for (i = 0; i < N; i++) {
12             result[i] = 0.0;
13             for (j = 0; j < N; j++) {
14                 result[i] += matrix[i][j] * vector[j];
15             }
16         }
17
18         // Calculate the norm of the result vector
19         norm = 0.0;
20         for (i = 0; i < N; i++) {
21             norm += result[i] * result[i];
22         }
23         norm = sqrt(norm);
24
25         // Normalize the result vector and copy it back to the input vector
26         for (i = 0; i < N; i++) {
27             vector[i] = result[i] / norm;
28         }
29
30         // Check for convergence
31         if (fabs(norm - *eigenvalue) < TOL) {
32             break;
33         }
34
35         *eigenvalue = norm;
36     }
37
38     // Copy the final vector to the eigenvector
39     for (i = 0; i < N; i++) {
40         eigenvector[i] = vector[i];
41     }
42 }
```

When providing your answers, you do not need to rewrite the full code again, but we ask you to clearly specify where in the sequential code your changes should be inserted (i.e. some number between 1 and 42) and what the changes are. For this problem you may provide your answers in pseudocode, but the call names and parameters should be accurate. A list of useful MPI and CUDA calls is shown below the questions.

The first set of questions deals with MPI:

(a) Modify the code with MPI calls to (1) distribute the matrix and vector to all ranks, (2) perform the matrix-vector product on each MPI rank, and (3) return the result. Use blocking point-to-point communication to solve this problem.

(b) Improve your solution to part (a) by using MPI collectives wherever applicable.

(c) An alternative way to achieve performance improvement is to target communication-computation overlap. Propose a code using non-blocking P2P communication to achieve this.

(d) Out of the two code variants developed in parts (b) and (c), which one do you expect to perform best? Briefly motivate why.

The second set of questions deals with CUDA:

(e) Write a CUDA host code that sends the matrix and vector to the CUDA device, executes the kernel and collects the resulting vector. You may assume that N is a multiple of 1024.

(f) Write a CUDA kernel that performs the matrix vector product. Do not use shared memory for this task.

(g) Improve performance by making use of shared memory. To simplify, you may assume that N <= 1024.

(h) Next, we want to improve the CUDA code with the goal of achieving computation-communication overlap. Briefly answer each of the following two questions: (1) What is a CUDA stream? (2) Draw a diagram showing a possible timeline for the overlapped execution, in which the different copies and kernel executions are visible.

(i) Finally, identify two parts of the code developed in part (e) that need to be changed and describe in words what needs to be changed.

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

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
    comm, MPI_Status *status)
int MPI_Isend (void *buffer, int count, MPI_Datatype type, int dest, int tag, MPI Comm
    comm, MPI_Request *request)
int MPI_Irecv (void *buffer, int count, MPI_Datatype type, int source, int tag, MPI Comm
    comm, MPI_Request *request)
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request,  MPI_Status *status)
int MPI_Barrier( MPI_Comm comm )
int MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm
    communicator)
```

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf
    , int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

A subset of CUDA calls that can be useful for this problem are shown below

```
cudaMalloc(void** devPtr, size_t size)
cudaFree(void* devPtr)
cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)
Kernel<<<dimGrid, dimBlock, shared_mem, stream>>>(parameters)
__syncthreads()
cudaStreamCreate(cudaStream_t *)
cudaStreamDestroy(cudaStream_t)
cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind,
    cudaStream_t streamx)
cudaMallocHost(void** devPtr, size_t size)
cudaFreeHost(void* devPtr)
```

Problem 5 (6 points)

Consider the following prefix sum code.

Listing 1: prefix sum

```
int N = 1000;
int A[N], prefix_sum[N];

// Initialize array
for (int i = 0; i < N; i++) {
    A[i] = i + 1;
}

prefix_sum[0] = A[0];

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

Listing 2: prefix sum algorithm rewritten

```
int N = 1000;
int A[N], prefix_sum[N];

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

prefix_sum[0] = A[0];

for (int d = 1; d < N; d *= 2) {
    int * read = (int*)malloc(sizeof(int)*N);

    memcpy(read, prefix_sum, sizeof(int)*N);

    for (int i = d; i < N; i += 1) {
        prefix_sum[i] += read[i - d];
    }
}
```

Tasks:

(a) Is the code in Listing 1 parallelizable using OpenMP. Why? Why not?

(b) To parallelize the code we have to rewrite the prefix sum algorithm, presented in Listing 2. Which `for` loops can be parallelized using OpenMP constructs? And why? Write down pseudo code.

(c) Will your code be faster than the serial version? Justify your answer.

(d) What are the main disadvantages of using synchronization constructs such as 'critical', 'barrier' and 'atomics' for this code?