# Solutions for the Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Friday, October 28th, 2022, 14:00h - 18:00h

## Problem 1 (10 points)

Consider a program that consists of the following main loop:

```
float sum = 0.0;
float a[N], b[N], c[N], d[N+1];

// main loop:
for (auto i = 0; i < N; i++){
    c[i] = a[i] + b[i];
    sum += a[i] * b[i];
    d[i+1]=d[i];
    }
```

### Tasks:

(a) Discuss whether this code can be parallelized and/or vectorized. If not, list the code transformations required to enable its parallelization and/or vectorization.

(b) The programmer is now tasked with parallelizing the code over a multicore processor. To select the unit of scheduling, they have been given three options: processes, kernel-level threads or user-level threads. Describe each option (2-3 lines) and list (at least) one advantage and one disadvantage of each approach.

(c) In the previous code, if each iteration is to become an independent scheduling unit, should the iteration be mapped to a distinct process, a distinct kernel-level thread, or a distinct user-level thread? Discuss each option separately from a scheduling perspective.

(d) Finally, from a data perspective, list two downsides of creating a new scheduling unit for each iteration. Given the discussion, suggest a new chunk size that will lead to high performance for a system with *P* processors and cache line size of 64 bytes.

### Answers

**Part (a)** The first statement can be parallelized and vectorized. The second statement can be parallelized and vectorized, but to deal with the reduction, privatization, and an outer reduction loop are required. The third statement cannot be parallelized or vectorized due to the loop carried dependency. One option is to move it out of the loop and execute it in its own loop. But note that what this loop does is copy d[0] to all other positions. hence the statement is equivalent to `d[i+1] = d[0]` which can both be parallelized and vectorized.

**Part (b)** Processes: own address space. Strong progress guarantees. Large creation and context switch overhead.
KLT: shared address space with kernel scheduling. Creation and context switch overheads are reduced. Strong progress guarantees.
ULT: shared address space with library scheduling. Very low creation and context switch overheads. Weak progress guarantees (a blocked thread blocks all other threads, and no guarantee that the library will ever schedule the ULT for execution)

**Part (c)** Because a single iteration is extremely fine grained, the only reasonable option would be to use user level threads.

**Part (d)** Two downsides are that (a) it will lead to false sharing, and (b) it will increase the amount of synchronization to deal with the reduction
The best possible chunk size would be max(N/P, 64/4). N/P because we want large chunks that do not limit parallelism, and 64/4 because with want to avoid false sharing across chunks.

## Problem 2 (10p)

STREAM is a widely used benchmark to measure the memory bandwidth of a system. In this problem you will use it to study the effect of different kinds of OpenMP schedulers on performance. Consider the following code:

```
#pragma omp parallel for
for (int i=0; i<STREAM_ARRAY_SIZE; i++) {
    c[i] = a[i];
}
```

Assume a programmer wants to run the loop on a system with 4 processors. `STREAM_ARRAY_SIZE` is set to 256.

**Tasks:**

(a) The programmer is considering the following four schedulers:

- Static schedule
- Dynamic schedule: "self-scheduling"
- Dynamic schedule: "chunk scheduling", with chunk size 50
- Guided schedule with minimum chunk size of 15: "GSS(15)"

How many chunks will be generated by each scheduler? And what is the number of iterations in each chunk?

(b) The programmer has found that each iteration executes for 1ns on the system. Based on your analysis in part (a), what are the execution times on the system when using the four scheduling schemes listed above?

(c) Now consider only self-scheduling and chunk scheduling. Assuming the dispatch overhead for one iteration is 0.1 ns, and the dispatch overhead for scheduling a chunk of size 50 is 4.5 ns, which of the two scheduling policies will execute faster?

(d) The programmer decides to use the static scheduling strategy for the STREAM COPY problem and thinks the execution time using static schedule for problem size 256 is satisfactory. Now they are able to access a larger system with 32 processors. However, creating the parallel threads on the new system incurs overhead, which depends on the number of processors (P). The time cost (overhead) is T = 1 ns + P $\times$ 0.5 ns. For simplicity, assume that the new system offers unlimited memory bandwidth. What is the largest problem size `STREAM_ARRAY_SIZE` that can be addressed on the new system without increasing the execution time?

## Answer to Problem 2:

**Part (a)**

- Static Schedule: 4 chunks, each 256/4 = 64 iterations
- Dynamic schedule: "self-scheduling": 256 chunk, 1 iteration per chunk
- Dynamic schedule: "chunk scheduling", with chunk size 50: 5 chunks with 50 iterations, 1 chunk with 6 iterations
- Guided schedule with minimum chunk size of 15: "GSS(15)": 9 chunks, number of iterations: 64, 48, 36, 27, 21, 15, 15, 15, 15

**Part (b)**

- Static schedule: 64 $\times$ 1 = 64 ns
- Dynamic schedule: "self-scheduling": 256 / 4 $\times$ 1 = 64 ns
- Dynamic schedule: "chunk scheduling", with chunk size 50: 2 $\times$ 50 $\times$ 1 = 100 ns
- Guided schedule with minimum chunk size of 15: "GSS(15)": 36+ 15 +15 = 66 ns

Chunk assignment:

| | | | | |
|---|---|---|---|---|
| P1 | 0 (64) | 64 | 64 | 64 |
| P2 | 0 (48) | 48 (15) | 63 | 63 |
| P3 | 0 (36) | 36 (15) | 51 (15) | 66 |
| P4 | 0 (27) | 27 (21) | 48 (15) | 63 |

**Part (c)**

- Self-scheduling: 256 / 4 * (1 + 0.1) = 70.4 ns (winner)
- Chunk Scheduling: 100 + 2 * 4.5 = 109 ns

**Part (d)** Solution 1: considering no time cost when running on 4-processor system. According to Gustafson's law, the execution time 64ns obtained in part (b) is constant. The time cost = 1 + 32 * 0.5 = 17 ns on 32-processor system. Assume problem size is x, then:

$$\frac{x}{32} + 17ns = 64ns \tag{1}$$

By solving equation 1, x = 1504.
Solution 2: If you instead consider the time cost on the 4-processor system, then the new execution time becomes 64 + 1 + 4 * 0.5 = 67s. Assume problem size is x, then:

$$\frac{x}{32} + 17ns = 67ns \tag{2}$$

By solving equation 2, x = 1600.

## Problem 3 (10p)

Consider an application consisting of the following kernel:

```
#define NDIM 1000

void 2dfunc(float next[NDIM][NDIM],
            float prev[NDIM][NDIM])
{
  for(j=1;j<NDIM-1;j++){
    for(i=1;i<NDIM-1;i++){
      next[i][j] = -4.0*prev[i][j] +
                        prev[i][j-1] + prev[i][j+1] +
                        prev[i-1][j] + prev[i+1][j];
      next[i][j] *= next[i][j];
    }
  }
}
```

**Tasks:**

(a) What is the arithmetic intensity (AI) of this code? Now 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 (execution time) if the code is run on a single core or on 8 cores? Note: a single `float` consists of 4 bytes.

(b) The above kernel is included in an outer loop that evolves the execution over multiple timesteps (see the code below). This outer loop includes a serial component that takes 0.5 ms. What is the upper limit to the speed-up (compared to a single core) according to Amdahl's law? For the above 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.

```
float a[NDIM][NDIM];
float b[NDIM][NDIM];

void run()
{
// assume input vectors are initialized
float *tmp, *future=b, *current=a;
for(int ts = 0; ts < NTIMESTEPS; ts++){
  serial_computation(current);
  2dfunc(future, current);
// swap pointers
  tmp = future;
  future = current;
  current = tmp;
  }
}
```

(c) Assume now that the developers are happy with the execution time of the current system. They believe that by replacing the current chip with a larger multicore chip they will be able to solve a larger problem in the same time. In 2-3 lines, discuss the suitability of this proposal.

(d) Part of the `serial_computation()` in (b) consists of the following Matrix-vector multiplication code:

```
int i, j, A[N][N], B[N], C[N];
for (i = 0; i < N; i++) {
  C[i] = 0;
  for (j = 0; j < N; j++) {
    C[i] = C[i] + A[i][j] * B[j];
  }
}
```

Propose two loop transformations (not involving parallelization) to improve the performance of this code on a system with caches. Rewrite the loop code with the proposed transformations.

## Answers

**Part (a)** code has 6 flops and two memory accesses, so the AI is 6 flops / 8 bytes = 0.75 FLOPS/byte.
Performance on system with 1 core min(4 GFLOPS, 0.75 $\times$ 16GB/s) = min(4 GFLOPS, 12 GFLOPS) = 4 GFLOPS
Performance on system with 8 cores min(32 GFLOPS, 0.75 $\times$ 16GB/s) = min(32 GFLOPS, 12 GFLOPS) = 12 GFLOPS
The number of flops of the algorithm is 6 x NDIM x NDIM = 6000000. Hence execution times are:
serial: 6e6 / 4e9 = 1.5e-3 seconds = 1.5 ms parallel: 6e6 / 12e9 = .5e-3 seconds = 0.5 ms

**Part (b)** serial fraction = .5 ms, ie 25% of total, the upper speed-up limit is 4x. For the case of 8 cores, the speed-up is 2x.
Based on the previous result, the kernel is memory bound. Hence (i) is not a good option. By improving the AI the performance of the kernel would yield approx 3x speed-up. Hence option (ii) is reasonable. As the serial part is not negligible (25%), option (iii) should also be considered to achieve scalability beyond a few cores.

**Part (c)** This is an application of Gustafson's law. However, Gustafson's law requires the system to scale ideally. In this case, we are proposing the increase the number of cores, but the memory bus remains the same. Hence, this will not work.

**Part (d)** Two transformations that will improve the caching behavior are loop blocking and interchange (i.e. loop tiling). Given a block size M, the final resulting code is as follows:

```
for (i = 0; i < N; i += M) {
  C[i... i+M-1] = 0;
  for (j = 0; j < N; j += M) {
    for (x = i; x < min(i + M, N); x++) {
      for (y = j; y < min(j + M, N); y++) {
        C[x] = C[x] + A[x][y] * B[y];
      }
    }
  }
}
```

## Problem 4 (10p)

The code in the below listing shows a serial implementation of the Cholesky decomposition. There are three steps to perform a Cholesky decomposition on a matrix: (1) The upper triangle of the matrix is replaced with zeroes, (2) the diagonal elements are calculated, and (3) the elements below the diagonal are computed. The goal of this problem is to develop an OpenMP implementation of the code.

```
double ** cholesky(double ** L, int n) {
    int i, j, k;
    for (j = 0; j < n; j++) {
        #pragma omp parallel
        {
            //Step 1. Set upper triangle to 0
            for (i = 0; i < j; i++)
                L[i][j] = 0;
            //Step 2: Calculate diagonal elements
            //OMP Loop 1
            //________________________
            for (k = 0; k < i; k++)
                    L[j][j] = L[j][j] - L[j][k] * L[j][k];
            //Should be done by one thread
            L[j][j] = sqrt(L[j][j]);
            //Step 3: calculate Lower triangular elements
            //OMP Loop 2
            //________________________
            for (i = j+1; i < n; i++) {
                for (k = 0; k < j; k++)
                    L[i][j] = L[i][j] - L[i][k] * L[j][k];
                L[i][j] = L[i][j] / L[j][j];
            }
        }
    }
    return L;
}
```

**Tasks:**

(a) The code `L[j][j] = sqrt(L[j][j])` should be executed by one thread. Show two ways to achieve this using OpenMP.

(b) Write the missing OpenMP pragmas for the loops "OMP Loop 1" and "OMP Loop 2". Explain your choices of shared and private list of variables.

(c) The loop "OMP Loop 1" contains a race condition. (1) Explain what causes the race condition, and (2) provide at least two ways to solve the race condition.

(d) Finally we ask you to consider if the outer j-loop can be parallelized. Explain in less than 3 lines why this is, or why this is not possible.

**Answer to Problem 4:**

**Tasks:**

(a) Two possible ways to execute the code by one thread are shown in the listing below.

(b) See the listing below.

(c) See the listing below.

(d) Loop j cannot be parallelized because values of loop variables (i and k) are dependent on j and these variable are used as index to write on L, creating read after write dependency among the threads.

```
//Solution of Task a
#pragma omp single
L[j][j] = sqrt(L[j][j]);
//OR
if(omp_get_thread_num() == 0) // enforcing only one thread
    L[j][j] = sqrt(L[j][j]);
#pragma omp barrier
```

```
//Solution of Task b
//OMP Loop 1
#pragma omp for shared(L) private(k)
    for (k = 0; k < i; k++)
....
//OMP Loop 2
#pragma omp for shared(L) private(i, k)
for (i = j+1; i < n; i++)
```

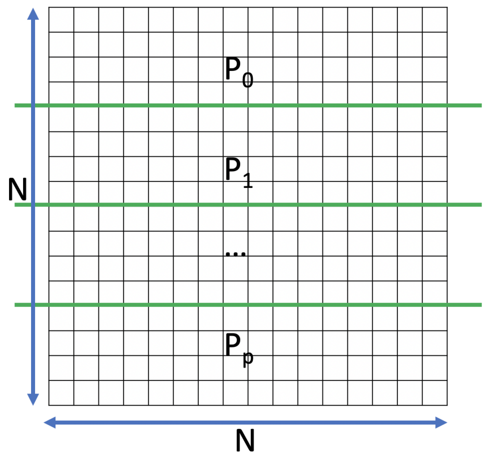
```
//Solution of task c
for(k = 0; k < i; k++)
    #pragma omp critical
    L[j][j] -= L[j][k] * L[j][k]; //Critical section.
//OR
for(k = 0; k < i; k++)
    float a = L[j][k] * L[j][k];
    #pragma omp atomic
    L[j][j] -= a; //Critical section.
```

## Problem 5 (10p)

The following code is used to approximate the solution of the Poisson problem $\nabla^2 u = f$ on a square matrix:

```
void compute(float** anew, float** aold, int j_start, int j_end,
             int i_start, int i_end){
    for(int j=j_start; j<j_end; ++j) {
      for(int i=i_start; i<i_end; ++i)
        anew[i][j] = (aold[i][j]+
                      aold[i-1][j] + aold[i+1][j] +
                      aold[i][j-1] + aold[i][j+1]) / 5.0;
    }
}
int main() {
  ...
  for (int time = 0; time < MAX_TIME; time++){
    compute(pnew, pold, j_start, j_end, i_start, i_end);
    pold = pnew; // update pold for next iteration
  }
  ...
}
```

This is a 2D 5-point stencil computation. The size of the matrix is $N^2$ and it is divided among *P* MPI processes as shown in the figure below.



Domain decomposition among *P* processes

**Tasks:**

(a) How many matrix elements are exchanged in each iteration? Provide the solution as a function of *N* and *P*.

(b) Add the necessary MPI calls to `main()` in order to exchange information. Make sure it does not cause deadlock.

(c) Now consider that each process checks for the convergence condition on its own region. The computation should stop if all processes agree that the solution has converged. An implementation of this is shown in the code below. Suggest an MPI implementation (in Psuedo code) that breaks the *time loop* (line 17) if all processes indicate convergence.

```
int compute(float** anew, float** aold, int j_start,
            int j_end, int i_start, int i_end){
    for(int j=j_start; j<j_end; ++j) {
      for(int i=i_start; i<i_end; ++i)
        anew[i][j] = (aold[i][j]+
                      aold[i-1][j] + aold[i+1][j] +
                      aold[i][j-1] + aold[i][j+1]) / 5.0;
    }
    if(delta(anew, aold) < 0.05)
        return 1; //solution converged
    else
        return 0;
}

int main() {
  ...
  for (int time = 0; time < MAX_TIME; time++){
    converge = compute(pnew, pold, j_start, j_end,
                        i_start, i_end);
    if(converge) // each process should check convergence
        break;
    else
        pold = pnew; // update pold for next iteration
  }
  ...
}
```

(d) Add the necessary code to measure the execution time in each iteration. How can you make sure that all processes have completed before measuring the time?

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

```
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_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
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)
```

**Answer to Problem 5:**

(a) 2N elements need to be exchanged per process for exchanging first and last row of their local matrix. However, first and last process will exchange only one row.

(b) The code below shows the solution using MPI. For the purpose of exam correction, explaining a high-level pseudo-code will be considered enough as long as the main calls to MPI routines and the exchange buffers are correctly identified.

(c) MPI_GATHER (convergence, root) followed by MPI_BROADCAST(convergence , root....). Alternatively, use MPI_Allgather() to achieve the same effect.

(d) MPI_BARRIER should be called at the end of each iteration then MPI_RANK=0 measure the time.

```
// For this solution, we consider that the MPI processes operate on a local
// matrix called local_A. local_A has a size of N/p+2 rows of N elements
// N/p rows for computing the next value, and two rows to hold the neighboring rows
// Thus, each rank operates on local_A[1][] to local_A[N/p][]
// The neighboring rows are indexed as local_A[0] and local_A[N/p+1]
// The solution below attempts to solve the deadlock issue by making use of
// buffered sends and receives

main(){
    // create a buffer for buffered sends, consisting of two rows of N elements
    buffer = malloc(2*N*sizeof(float));
            // in practice more space is needed, but this is not discussed
            // in DAT400 so we consider that this is correct
    MPI_Buffer_attach(buffer,2*N*sizeof(float));

    // Read local data for each process
    // start computation
    for (int time = 0; time < MAX_TIME; time++){
        j_start = 0; i_start = 0;
        j_end = N; i_end = N/p;
        int converge = compute(pnew, local_A, j_start, j_end, i_start, i_end);
        // gather converge value from all processes and
        // broadcast updated converge value

        // Label ranks as upper and lower
        upper = rank -1;
        lower = rank +1;

        if(upper > 0) MPI_Bsend(&local_A[1], N*sizeof(float), MPI_FLOAT,
                                upper, 1, MPI_COMM_WORLD, status);
        if(lower < p) MPI_Bsend(&local_A[(N/p)], N*sizeof(float), MPI_FLOAT,
                                lower, 2, MPI_COMM_WORLD, status);

        if(lower < p) MPI_Recv(&local_A[(N/p)+1], N*sizeof(float), MPI_FLOAT,
                                lower, 1, MPI_COMM_WORLD, status);
        if(upper > 0) MPI_Recv(&local_A[0], N*sizeof(float), MPI_FLOAT,
                                upper, 2, MPI_COMM_WORLD, status);
  }
}
```

## Problem 6 (10p)

The CUDA program below computes the outer product of two vectors, `u[]` and `v[]`. The outer product is a specific case of matrix multiplication in which the first matrix is a (column) vector of N elements and the second matrix is the transpose of a vector (also called a row vector) of N elements. When we multiply a N×1 matrix by a 1×N matrix, the result is an N×N matrix, which we call the outer product of the two vectors. The code below calculates the outer product of vector `u[]` with vector `v[]` and returns the answer as matrix `A[][]`.

```
#define BLOCK_DIM 32
__global__ outer_product_kernel (float* u, float* v, float* A,
  unsigned int N){
  /* Perform the outer product of u and v,
  * u is of size N x 1
  * v is of size 1 x N
  * A is of size N x N */
  unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
  unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row<N && col<N){
      A[row * N + col] = u[row] * v[col];
  }
}

void outer_product (float* u, float* v, float* A, unsigned int N)
{
  .....
  // sizeof(float) = 4 bytes
  cudaMalloc((void **) &u_d, N * sizeof(float));
  cudaMalloc((void **) &v_d, N * sizeof(float));
  cudaMalloc((void **) &A_d, N * N * sizeof(float));
  cudaMemcpy(1, 2, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(3, 4, N * sizeof(float), cudaMemcpyHostToDevice);
  dim3 blockDim (5);
  dim3 gridDim (6);
  outer_product_kernel<<<gridDim, blockDim>>>(u_d, v_d, A_d, N);
  cudaMemcpy(7, 8, N * N * sizeof(float), cudaMemcpyDeviceToHost);
  .....
}
```

**Tasks:**

(a) Considering the CUDA implementation above. Fill in the missing code pieces 1 to 8 in the function `outer_product`. You may find the following API helpful:
`cudaMemcpy(Destination, Source, Size in bytes to copy, Type of transfer)`

(b) Now we consider using tiling and shared memory to optimize the kernel. Each thread block computes a tile of matrix A. Parts of vectors `u[]` and `v[]` are loaded from global memory to shared memory. Assume the size of shared memory on the GPU is 256 bytes. How many elements of vectors `u[]` and `v[]` can be loaded into shared memory?

(c) Rewrite the CUDA kernel (`outer_product_kernel`) to make use of tiling and shared memory as described in part (b).

## Answer to Problem 6:

**Part (a)**

(1) u_d

(2) u

(3) v_d

(4) v

(5) BLOCK_DIM, BLOCK_DIM, 1 (Third dimension 1 is optional)

(6) (N-1)/BLOCK_DIM + 1, (N-1)/BLOCK_DIM + 1, 1 (Third dimension 1 is optional)

(7) A

(8) A_d

**Part (b)** 256 bytes / sizeof(float) 4 / 2 vectors = 32 elements per vector can fit into shared memory.

**Part (c)**

```
#define BLOCK_DIM 32
__global__ outer_product_kernel (float* u, float* v, float* A, unsigned
int N){
    __shared__ float shared_u[BLOCK_DIM];
    __shared__ float shared_v[BLOCK_DIM];
    unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
    for(int i = 0; i < ceil(N / BLOCK_DIM); i++){
        shared_u[threadIdx.y] = u[threadIdx.y + i * BLOCK_DIM];
        shared_v[threadIdx.x] = v[threadIdx.x + i * BLOCK_DIM];
        __syncthreads();
        A[row * N + col] = shared_u[threadIdx.y] * shared_v[threadIdx.x];
        __syncthreads();
    }
}
```