Suggested solutions for the IN3200/IN4200 exam of spring 2022

Question 1: Cache line (5 points)

Give a short explanation about ”cache line”.

Suggested solution: On a modern computer, caches are low-capacity, high-speed storages between the processing units and the main memory. A cache line is the basic unit of data transfer from/to the main memory or within the cache hierarchy. Each cache line contains multiple data values that are stored *contiguously* in memory. Spatial locality is important for effectively using the cache lines, that is, all the values on a cache line should ideally be used (preferably several times) before the cache line is evicted.

Question 2: Performance improvement (10 points)

Please name one data-access optimization technique that is very effective for improving the performance of the following code when n is large. Explain your answer.

```
for (i=0; i<n; i++) {
   double t = 0.0;
   for (j=0; j<n; j++)
       t += a[i][j]*b[j];
   c[i] = t;
}
```

We assume that `a` is a row-major 2D array of dimension $n \times n$, while `b`, `c` are 1D arrays of length n.

Suggested solution: *Loop unroll and jam* (see Section 3.5.2 in the textbook) is an effective optimization technique, for example,

```
for (i=0; i<n; i+=4) {
   double t0 = 0.0, t1 = 0.0, t2 = 0.0, t3 = 0.0;
   for (j=0; j<n; j++) {
       t0 += a[i][j]*b[j];
       t1 += a[i+1][j]*b[j];
       t2 += a[i+2][j]*b[j];
       t3 += a[i+3][j]*b[j];
```

```
    }
    c[i] = t0; c[i+1] = t1; c[i+2] = t2; c[i+3] = t3;
}
```

The purpose is to reduce the amount of memory traffic related to the array `b` when the size of n is large (such that a cache cannot keep the entire array `b`). The unroll-and-jam optimization shown above leads to loading $n/4$ times the array `b`, instead of n times in the original code. Note: We have assumed that n is divisible by the unroll depth (which is 4 for this example). The unroll depth can be different from 4 (but not too large).

Question 3: Estimating time usage (10 points)

Assume that we have a CPU with 40 GB/s as the theoretical memory bandwidth and 100 GFLOP/s (in double precision) as the theoretical peak floating-point performance. It is also known that each cache line is 64 bytes in length. Can you estimate how much time the following code needs on this CPU, when $n = 10^5$? (If needed, you can make additional assumptions about the CPU.)

```
#define stride 2

double s = 0.0;
for (i=0; i<n; i++)
  for (j=0; j<n; j+=stride)
     s += a[i][j]*a[i][j];
```

We assume that `a` is a row-major 2D array of dimension $n \times n$ and contains double-precision values.

Suggested solution: To estimate the time needed to load the required values of array `a` from memory, we can count the number of cache lines transferred. (Note that even with `stride=2`, entire cache lines are still loaded.) This can be roughly calculated as

$$n \cdot \frac{n \cdot \text{sizeof}(\texttt{double})}{64} = 10^5 \cdot \frac{10^5 \cdot 8}{64} = 1.25 \cdot 10^9 \text{ cache lines}$$

Thus the time needed to transfer so many cache lines can be calculated as

$$\frac{1.25 \cdot 10^9 \cdot 64 \text{ bytes}}{40 \cdot 10^9 \text{ bytes/second}} = 2 \text{ seconds}$$

The time needed for the floating-point operations can be estimated as

$$\frac{2 \cdot n^2/2}{100 \text{ GFLOP/s}} = \frac{10^{10}}{100 \cdot 10^9} = 0.1 \text{ seconds}$$

Due to hardware pipelining that is available on all modern processors, the cost of the floating-point operations is "hidden" by the memory traffic cost. Therefore, the total time needed is 2 seconds.

Question for IN4200 students

The IN4200 students are requested to answer the same question, but with `stride` defined as 9.

Suggested solution: Since the value of `stride` is now 9, which is larger than 8 (number of double-precision values stored per cache line), one out of every 9 cache lines in memory will be skipped over. Therefore, the total amount of memory traffic will be reduced by a factor of $\frac{1}{9}$, so the time for data transfer can be roughly calculated as

$$\frac{\frac{8}{9} \cdot 1.25 \cdot 10^9 \cdot 64 \text{ bytes}}{40 \cdot 10^9 \text{ bytes/second}} = 1.78 \text{ seconds}$$

The time needed for the floating-point operations can be estimated as

$$\frac{2 \cdot n^2/9}{100 \cdot 10^9} = 0.022 \text{ seconds}$$

The total time needed is 1.78 seconds (see the explanation above).

Question 4: Difficulty for straightforward OpenMP parallelization (10 points)

Question for IN3200 students

```
for (i=0; i<n-1; i++) {
    a[i] = b[i] + c[i];
    d[i+1] = d[i] + a[i]*c[i];
}
```

Explain why the above code segment cannot be directly parallelized by only inserting "`#pragma omp parallel for`" before the for-loop.

Note that `a`, `b`, `c`, `d` are 1D arrays of length n.

Suggested solution: There is loop-carried dependency between the iterations, particularly due to

```
    d[i+1] = d[i] + a[i]*c[i];
```

Therefore, if multiple threads concurrently execute different segments of the iterations, the final result in the array `d` will be wrong.

Question for IN4200 students

Explain why the following code segment cannot be directly parallelized by only inserting "`#pragma omp parallel for`" before either of the two for-loops.

```
for (i=1; i<n-1; i++)
   for (j=1; j<n-1; j++)
       a[i][j] = 0.25*(a[i-1][j]+a[i][j-1]+a[i][j+1]+a[i+1][j]);
```

We assume that `a` is a row-major 2D array of dimension $n \times n$.

Suggested solution: There is loop-carried dependency between the `i`-indexed iterations, as well as between the `j`-indexed iterations. Applying directly `#pragma omp parallel for` before either of the two loops will lead to wrong final result.

Question 5: OpenMP parallelisation (10 points)

Parallelize the code in the previous question by using OpenMP.

(Hint for IN3200 students: You need to first restructure the code. Remember to use comments for explaining the parallelization.)

(Hint for IN4200 students: You need to first restructure the code and identify "wave fronts". Remember to use comments for explaining the parallelization.)

For IN3200 students

Suggested solution: A simple (but imperfect) solution is to split the loop into two loops, where the first loop is parallelized by OpenMP, and the second loop is executed serially.

```
#pragma omp parallel for
for (i=0; i<n-1; i++)
  a[i] = b[i] + c[i];

for (i=0; i<n-1; i++)  // serial execution
  d[i+1] = d[i] + a[i]*c[i];
```

A more sophisticated solution is to realize that the final values in the array `d` are actually

```
  d[1] = d[0] + a[0]*c[0];
  d[2] = d[0] + a[0]*c[0] + a[1]*c[1];
  d[3] = d[0] + a[0]*c[0] + a[1]*c[1] + a[2]*c[2];
  ...
```

This can lead to the following OpenMP parallelization:

```
#pragma omp parallel
{
  // declaration of thread-private variables
  int i, istart, iend, num_threads, thread_id;

  num_threads = omp_get_num_threads();
  thread_id = omp_get_thread_num();

  // calculate start and end indices for each thread
  istart = thread_id*(n-1)/num_threads; iend = (thread_id+1)*(n-1)/num_threads;

  d[iend] = d[0];
  for (i=istart; i<iend; i++) {
    a[i] = b[i] + c[i];
    d[iend] += a[i]*c[i];
  }
```

```
#pragma omp barrier

#pragma omp single
{
  for (i=1; i<num_threads; i++)
     d[(i+1)*(n-1)/num_threads] += d[i*(n-1)/num_threads] - d[0];
}

  for (i=istart; i<iend-1; i++)  // very important for each thread to stop at i=iend-2
     d[i+1] = d[i] + a[i]*c[i];
}  // end of parallel region
```

For IN4200 students

Suggested solution: We can use the same strategy of forming "wavefronts" described in Section 6.3 in the textbook.

```
#pragma omp parallel
{
  // declaration of thread-private variables
  int i, j, m, jstart, jend, num_threads, thread_id;

  num_threads = omp_get_num_threads();
  thread_id = omp_get_thread_num();

  // start and end indices for each thread
  jstart = thread_id*(n-2)/num_threads + 1;
  jend = (thread_id+1)*(n-2)/num_threads + 1;

  for (m=1; m<n+num_threads-2; m++) {
     i = m - thread_id;
     if (i > 0 && i<n-1) {
       for (j=jstart; j<jend; j++)
          a[i][j] = 0.25*(a[i-1][j]+a[i][j-1]+a[i][j+1]+a[i+1][j]);
     }
#pragma omp barrier   // very important
  }
}  // end of parallel region
```

Question 6: Speedup discussion (5 points)

Discuss the achievable speedup as a function of the number of threads.

For IN3200 students

Suggested solution: Suppose the simple (but imperfect) OpenMP parallelization is used. If we only consider the perspective of floating-point operations, then the expected speedup due to using P threads

can be calculated as

$$\frac{3\cdot(n-1)}{\frac{n-1}{P}+2\cdot(n-1)}=\frac{3P}{1+2P}$$

Suppose the sophisticated OpenMP parallelization is used. From the perspective of floating-point operations (more operations are executed in total), the speedup due to using P threads can be calculated as

$$\frac{3\cdot(n-1)}{\frac{5\cdot(n-1)}{P}}=\frac{3}{5}P$$

For simplicity, we do not calculate the speedup from the perspective of memory traffic and available memory bandwidth per thread.

Note: Answering the entire question in a general sense, that is, OpenMP parallelizing any code with threads, is also acceptable.

For IN4200 students

Suggested solution: The total number of wavefronts is $n-2+P-1=n+P-3$, where P denotes the number of threads used. On each wavefront, a thread is either completely idle or needs to compute $(n-2)/P$ points.

From the perspective of floating-point operations, if we suppose computing $n-2$ points by a single thread takes one time unit, then the speedup can be calculated as

$$\frac{(n-2)\cdot(n-2)}{(n+P-3)\cdot\frac{n-2}{P}}=P\cdot\frac{n-2}{n+P-3}$$

From the memory bandwidth perspective, however, the available memory bandwidth per thread is often lower the single-thread exclusive memory bandwidth. Therefore, the actual speedup will be lower than that calculated above.

Note: Answering the entire question in a general sense, that is, OpenMP parallelizing any code with threads, is also acceptable.

Question 7: Understanding an MPI program (10 points)

What will be the result of running the following MPI program using 8 MPI processes? (See the attachment for a quick reference of important MPI functions.)

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int sum = rank;
  int temp, stride = 1;

  while(stride <= size/2) {
    if (rank<size-stride) {
      MPI_Send(&sum, 1, MPI_INT, rank+stride,0,MPI_COMM_WORLD);
    }

    if (rank>=stride) {
      MPI_Recv(&temp, 1, MPI_INT, rank-stride,0,MPI_COMM_WORLD,&status);
      sum += temp;
    }

    stride = 2 * stride;
  }

  printf("On rank <%d>, sum=%d\n", rank, sum);
  MPI_Finalize();
  return 0;
}
```

Suggested solution:

```
On rank <0>, sum=0
On rank <1>, sum=1
On rank <2>, sum=3
On rank <3>, sum=6
On rank <4>, sum=10
On rank <5>, sum=15
On rank <6>, sum=21
On rank <7>, sum=28
```

Note: The order of the output from the different MPI processes may change from execution to execution.

Question 8: Implementation of function `find_min_with_index` (8 points)

Write a serial C function with the following syntax:

```
void find_min_with_index (const int *input_array, const int length, int *min_value, int *index);
```

The purpose of the function is to find the minimum value inside `input_array`, as well as the index of the minimum value found. Here is a concrete small example of usage:

```
  int an_array[6] = {19, 3, -2, 5, -2, 0};
  int min_v, ind;
  find_min_with_index (an_array, 6, &min_v, &ind);
```

```
  printf("minimum value=%d, index=%d\n", min_v, ind);
```

The result of printf will be

```
   minimum value=-2, index=2
```

Note: if there are multiple values having the same minimum value, the lowest index is to be found.

Suggested solution:

```
void find_min_with_index (const int *input_array, const int length, int *min_value, int *index)
{
  *min_value = input_array[0];
  *index = 0;

  for (int i=1; i<length; i++)  // starting from index 1
    if (input_array[i] < *min_value) {
      *index = i;
      *min_value = input_array[i];
    }
}
```

Question 9: OpenMP parallelization of `find_min_with_index` (12 points)

Implement an OpenMP parallelization of function `find_min_with_index` from the previous question.

Suggested solution:

```
void find_min_with_index (const int *input_array, const int length, int *min_value, int *index)
{
  *min_value = input_array[0];
  *index = 0;

#pragma omp parallel
{
  // declaration thread-private variables
  int my_min_value = input_array[0];
  int my_index = 0;

#pragma omp for nowait
  for (int i=1; i<length; i++)
    if (input_array[i] < my_min_value) {
      my_index = i;
      my_min_value = input_array[i];
    }
```

```
#pragma omp critical
{
  if (my_min_value < *min_value) {
    *index = my_index;
    *min_value = my_min_value;
  }
  else if (my_min_value == *min_value && my_index < *index) {
     *index = my_index;
  }
}

} // end of #pragma omp parallel

} // end of function
```

Question 10: Denoising a color image (10 points)

We want to extend the second mandatory project such that a color image can be denoised. More specifically, we assume that there is one 3D array of single-precision numerical values named "`color_image_data_bar`" and another 3D array named "`color_image_data`". The first two dimensions are the same as in the second project, that is, as the number of vertical pixels and horizontal pixels in the image. The third dimension is 3, containing the "red", "green", "blue" components of each pixel. Both the 3D arrays have an underlying 1D contiguous storage.

You are required to implement a serial function as follows:

```
void one_denoising_iteration
     (int m, int n, float ***color_image_data_bar, float ***color_image_data, float kappa)
```

which carries out one iteration of isotropic diffusion. The effect is that the red, green, blue components are denoised by

$$\begin{aligned}
\bar{u}_{i,j,r} &= u_{i,j,r} + \kappa\left(u_{i-1,j,r} + u_{i,j-1,r} - 4u_{i,j,r} + u_{i,j+1,r} + u_{i+1,j,r}\right), \\
\bar{u}_{i,j,g} &= u_{i,j,g} + \kappa\left(u_{i-1,j,g} + u_{i,j-1,g} - 4u_{i,j,g} + u_{i,j+1,g} + u_{i+1,j,g}\right), \\
\bar{u}_{i,j,b} &= u_{i,j,b} + \kappa\left(u_{i-1,j,b} + u_{i,j-1,b} - 4u_{i,j,b} + u_{i,j+1,b} + u_{i+1,j,b}\right),
\end{aligned}$$

where the indices are $r = 0$, $g = 1$, $b = 2$.

Suggested solution:

```
void one_denoising_iteration
     (int m, int n, float ***color_image_data_bar, float ***color_image_data, float kappa)
{
  for (i=1; i<m-1; i++)
    for (j=1; j<n-1; j++) {
       // red component
       color_image_data_bar[i][j][0] = color_image_data[i][j][0]
              + kappa*(color_image_data[i-1][j][0] + color_image_data[i][j-1][0]
               - 4*color_image_data[i][j][0]
```

```
            + color_image_data[i][j+1][0]+color_image_data[i+1][j][0]);
      // green component
      color_image_data_bar[i][j][1] = color_image_data[i][j][1]
            + kappa*(color_image_data[i-1][j][1] + color_image_data[i][j-1][1]
             - 4*color_image_data[i][j][1]
             + color_image_data[i][j+1][1]+color_image_data[i+1][j][1]);
      // blue component
      color_image_data_bar[i][j][2] = color_image_data[i][j][2]
            + kappa*(color_image_data[i-1][j][2] + color_image_data[i][j-1][2]
             - 4*color_image_data[i][j][2]
             + color_image_data[i][j+1][2]+color_image_data[i+1][j][2]);
    }
}
```

Question 11: Programming MPI communication (10 points)

Now we want to use MPI processes to parallelize the work of "one_denoising_iteration". Assume that a color image is already partitioned vertically such that each MPI process is assigned with 'my_m' rows of pixels (including two ghost rows). The numbers of pixel columns per process is still 'n'.

Implement the necessary MPI communication needed before every MPI process can call one_denoising_iteration restricted to its partitioned data (that is, my_m replaces m, my_color_image_data_bar replaces color_image_data_bar, and my_color_image_data replaces color_image_data).

If needed, please use comments to provide necessary explanations. (See the attachment for a quick reference of important MPI functions.)

Suggested solution: The *same* communication pattern as in the second mandatory project will be used. That is, each process has an "upper" neighbor (unless rank=0) and a "down" neighbor (unless rank equals number of processes minus 1). The only difference is that the size of the outgoing and incoming messages is three times that used in the second mandatory project.

Therefore, before each MPI process applies one_denoising_iteration to its part, the following MPI commands can be used to implement the necessary communication:

```
if (my_rank > 0) {
  MPI_Isend(my_color_image_data[1][0], 3*n, MPI_FLOAT, my_rank-1, 0, MPI_COMM_WORLD, &request0);
  MPI_Irecv(my_color_image_data[0][0], 3*n, MPI_FLOAT, my_rank-1, 0, MPI_COMM_WORLD, &request1);
}

if (my_rank < num_procs-1) {
  MPI_Isend(my_color_image_data[my_m-2][0], 3*n, MPI_FLOAT, my_rank+1, 0, MPI_COMM_WORLD, &request2);
  MPI_Irecv(my_color_image_data[my_m-1][0], 3*n, MPI_FLOAT, my_rank+1, 0, MPI_COMM_WORLD, &request3);
}

if (my_rank > 0) {
  MPI_Wait (request0, &status);
  MPI_Wait (request1, &status);
}
```

```
if (my_rank < num_procs-1) {
  MPI_Wait (request2, &status);
  MPI_Wait (request3, &status);
}
```

Note: We have used the fact the 3D arrays have am underlying 1D contiguous storage. It is also possible to use the `MPI_Sendrecv` function or suitably alternated calls to `MPI_Send` and `MPI_Recv`.