

# Suggested solutions for the IN3200/IN4200 exam of spring 2024

**Note:** The suggested solution for Question 5 was improved on May 23, 2025 for better clarity and precision. The MPI implementation for Question 8 (IN3200 version) was also corrected for small bugs.

## Question 1: Improve the speed (10 points)

Please rewrite the following code segment to improve the computing time.

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            A[i][j][k] = pow(a, (i+j+k));
```

Note: A is a 3D array of dimension  $n \times n \times n$ , a is a constant, and function  $\text{pow}(a, (i+j+k))$  computes  $a^{(i+j+k)}$ .

**Suggested solution:** The performance problem with the above code segment is due to the repeated calculations of the time-costly `pow` function. (In total  $n^3$  times!) To save time, we can create a *lookup table* of length  $3n - 2$ , which can be used to pre-calculate for each possible input value of  $i + j + k$ .

An improved code segment can be as follows:

```
double* lookup = malloc((3*n-2)*sizeof(double));
lookup[0] = 1.0;
for (i=1; i<=3*n-3; i++)
    lookup[i] = a*lookup[i-1];

for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            A[i][j][k] = lookup[i+j+k];
```

Note: We have completely replaced all the time-costly `pow` function calls with simple multiplications in the above code segment.

## Question 2: SIMD vectorization (12 points)

Please explain why the following code cannot be SIMD vectorized.

```
for (i=1; i<n; i++)  
    v[i] = v[i-1]*v[i-1];
```

Can the following code be SIMD vectorized? Please provide an explanation.

```
for (i=10; i<n; i++)  
    v[i] = v[i-10]*v[i-10];
```

**Suggested solution:** SIMD vectorization is a hardware technology available in modern processors to carry out a single instruction simultaneously on multiple (neighboring) data points. The first loop cannot be vectorized because the computation of  $v[i]$  cannot start until  $v[i-1]$  is completed. That is,  $v[i-1]$  and  $v[i]$  cannot be computed at the same time.

The second loop, however, is vectorizable as long as the vector length is shorter than 10. (For example, if the vector length is 4, the values of  $v[i-3]$ ,  $v[i-2]$ ,  $v[i-1]$ ,  $v[i]$  can be computed simultaneously.)

## Question 3: Completing 200 tasks (12 points) (IN3200 version)

There are 200 identical tasks, and there are three workers of different skill levels. Specifically, worker 1 needs 1 hour to complete one task, worker 2 needs 2 hours to complete one task, whereas worker 3 needs 4 hours to complete one task.

How many hours minimum do the three workers need to complete all the 200 tasks? (Note: The tasks are independent of each other, and it is not possible to let multiple workers collaborate on one task.)

**Suggested solution:** First, let us notice that the three workers can complete 7 tasks in 4 hours (4 tasks by worker 1, 2 tasks by worker 2 and 1 task by worker 3). Therefore, the three workers can complete 196 tasks in  $\frac{196}{7} \times 4 = 112$  hours. The remaining 4 tasks can be completed in 3 hours (3 tasks for worker 1 and 1 task for worker 2). Therefore, the total time needed is 115 hours.

## Question 3: Completing $m$ tasks (12 points) (IN4200 version)

There are  $m$  identical tasks, and there are three workers of different skill levels. Specifically, worker 1 needs 1 hour to complete one task, worker 2 needs 2 ours to complete one task, whereas worker 3 needs 4 hours to complete one task.

How many hours minimum do the three workers need to complete all the  $m$  tasks? (Note: The tasks are independent of each other, and it is not possible to let multiple workers collaborate on one task.)

**Suggested solution:** First, let us notice that the three workers can complete 7 tasks in 4 hours (4 tasks by worker 1, 2 tasks by worker 2 and 1 task by worker 3). Therefore, if  $m$  is divisible by 7, the total time needed is  $\frac{m}{7} \times 4$ . If  $m$  is not divisible by 7, the remainder of the integer division can only be between 1 and 6. The following table shows the number of hours needed for the three workers to complete 1,2,3,4,5,6 tasks.

| Tasks | Hours | Worker 1 | Worker 2 | Worker 3 |
|-------|-------|----------|----------|----------|
| 1     | 1     | 1        | none     | none     |
| 2     | 2     | 2        | none     | none     |
| 3     | 2     | 2        | 1        | none     |
| 4     | 3     | 3        | 1        | none     |
| 5     | 4     | 4        | 1        | none     |
| 6     | 4     | 4        | 2        | none     |

## Question 4: Estimating memory traffic (13 points)

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        A[i][j] = B[j][i];
```

Estimate the amount of data traffic to and from the main memory due to running above code segment. Please consider two situations: (1) the cache is very large; (2) the cache is very small relative to the size of A and B.

A and B are both 2D arrays of dimension  $n \times n$  containing double-precision floating-point values. The cacheline size is assumed to be 64 bytes. The caches are empty in the beginning.

**Suggested solution:** First, we notice that the values of the 2D array A are traversed row-by-row, that is, the most memory-efficient order for 2D arrays in the C language. No cache-to-memory traffic related to A will be wasted. The total memory traffic due to storing the 2D array A to memory is thus

$$n^2 \times 8 \text{ bytes}$$

Note: Each double-precision floating-point value occupies 8 bytes in memory. The above estimate is **independent** of the cache capacity (as long as the last-level cache has at least two cachelines).

Second, we notice that the column-by-column loading of values of array B from memory is the most memory-inefficient, due to potentially very big strides between the memory accesses. The amount of memory traffic due to loading B values thus depends on the size of cache relative to the array dimension  $n$ . If  $n$  is very small, the cache capacity can be large enough such that each loaded cacheline of B values is not evicted before all the values on the cacheline are used. In such a case, the total memory traffic for loading the 2D array B is also

$$n^2 \times 8 \text{ bytes}$$

However, if  $n$  is very large, we can have a very unfortunate situation that every access to  $B[j][i]$  requires (re-)loading an entire cacheline of 64 bytes from memory. In other words, the total memory traffic related to loading the B values becomes

$$n^2 \times 64 \text{ bytes}$$

## Question 5: Reducing memory for the case of small cache (13 points)

Restructure the code segment given in the last question, with the purpose of reducing the amount of data traffic for the case of very small caches. Estimate the new amount of data traffic due to restructuring the code (by adding comments in the restructured code).

**Suggested solution:** Applying the technique of “unroll and jam”, with an unroll depth  $m$ , on the outer for-loop level will give the following restructured code:

```
for (i=0; i<n; i+=m) // working simultaneously on m rows of A
    for (j=0; j<n; j++) {
        A[i][j] = B[j][i]; // note B[j][i], B[j][i+1]... are neighboring values in memory
        A[i+1][j] = B[j][i+1];
        // ...
        A[i+m-1][j] = B[j][i+m-1];
    }
```

Assume there are at least  $m + 1$  cachelines in the last-level cache (even though the capacity can still be very small), the amount of memory traffic due to storing A values to memory remains the same at  $n^2 \times 8$  bytes for the restructured code. This is because in theory  $m$  cachelines can be used for  $m$  different rows of the A values, while one cacheline is used for the B values. Compared with the worst-case scenerio described in the previous question, the amount of traffic due to loading B values from memory will be reduced by a factor of up to 8, **because each 64-byte cacheline has space for just 8 double-precision values**. Specifically, even for a very small last-level cache (having at least  $m + 1$  cachelines), with  $m = 2$ , the memory traffic amount due to loading B values is reduced to  $n^2 \times 32$  bytes (**because two values will be used every time a cacheline is loaded from memory**); with  $m = 4$  the traffic is reduced to  $n^2 \times 16$  bytes (**four values will be used per loaded cacheline**); whereas with  $m = 8$  the B traffic amounts to  $n^2 \times 8$  bytes (the lowest possible). **Note: In the above, we have assumed that the start memory address of each row of the 2D array B aligns with a cache line. Otherwise there will be more traffic.**

For other values of  $m$ , the benefit of “unroll and jam” needs a closer look. Take  $m = 3$  for instance. **The cacheline containing  $B[0][0], \dots, B[0][7]$  will be loaded from memory three times. (This is because  $B[0][0], B[0][1], B[0][2]$  are used when the cacheline is loaded the first time;  $B[0][3], B[0][4], B[0][5]$  are used when the cacheline is loaded the second times;  $B[0][6], B[0][7]$  are used when the cacheline is loaded the third time, whereas  $B[0][8]$  requires loading the next cacheline that contains  $B[0][8], \dots, B[0][15]$  already once.) A detailed count will reveal that  $\frac{2}{3}$  of the cachelines will be loaded three times from memory and  $\frac{1}{3}$  of the cachelines will be loaded four times. So the total memory traffic associated with array B is about  $n^2 \times 26.7$  bytes using ‘unroll and jam’ with  $m = 3$ . For  $m = 5$ , the total memory traffic will be  $n^2 \times 19.2$ , and for both  $m = 6$  and  $m = 7$ , the total traffic will be  $n^2 \times 16$ . Using an unroll depth  $m > 8$  will actually reduce the benefit enjoyed by  $m = 8$ .**

It should be remarked that although the technique of “unroll and jam” is effective at reducing the amount of memory traffic. Another performance-unfriendly factor still remains. That is, the memory accesses related to both A and B now have very big jumps (when the array dimension  $n$  is large). Such big jumps will result in that the so-called translation look-aside buffer (TLB), which records the mapping between logical and physical memory pages, is more frequently updated, leading to bad performance. (Please refer to pages 75–76 of the textbook for a detailed explanation.) Therefore, the technique of “loop blocking”, as shown below, is even better for performance than “unroll and jam”, although the amount of memory traffic is not further reduced than  $n^2 \times 8$  bytes for both arrays A and B.

```

for (ii=0; ii<n; ii+=b) { // 4 is a typically good value for b
    istart = ii; iend = ii+b;
    for (jj=0; jj<n; jj+=b) {
        jstart = jj; jend = jj+b;

        for (i=istart; i<iend; i++)
            for (j=jstart; j<jend; j++)
                A[i][j] = B[j][i];
    }
}

```

## Question 6: Improving the speed of OpenMP code (15 points)

Please rewrite the following function to improve the speed.

```

void func (int *v, int n) {
    int counter = 0;

#pragma omp parallel
    {
        int num_threads, id, i;

        num_threads = omp_get_num_threads();
        id = omp_get_thread_num();

        for (i=id; i<n; i+=num_threads) {
            v[i] = v[i] + 1;

#pragma omp critical
            {
                if (v[i]>10)
                    counter = counter + 1;
            } // end of critical

        } // end of for loop
    } // end of parallel

    printf("counter=%d\n", counter);
}

```

Please insert an explanation about why the original code is slow as comments in the beginning of the new code.

**Suggested solution:** See the comment in the beginning of the following code for an explanation of why the original code is slow.

```

/*
There are two reasons for the original code to be slow.

1. The different threads have a high probability of updating the
neighboring values in array v, due to the way how the iterations
are assigned to the threads. This will cause the problem of false
sharing.

2. The critical section is over-used, leading to slow execution.
The same result can be obtained by using the reduction clause (much faster).
*/

void func (int *v, int n) {

```

```

int counter = 0, i;

#pragma omp parallel for reduction(+:counter)
for (i=0; i<n; i++) {
    v[i] = v[i] + 1;

    if (v[i]>10)
        counter = counter + 1;
} // end of for loop

printf("counter=%d\n", counter);
}

```

## Question 7: Parallelization (15 points) (IN3200 version)

We want to use MPI programming to parallelise the function below. Please explain how the data and work can be divided among the process, and what kind of MPI communication will be needed to enable the necessary data exchanges. (No detailed coding is needed here, which will be subject of the next question.)

```

void matvec (int n, int **A, int *x, int *y)
{
    int i, j, sum;

    for (i=0; i<n; i++) {
        sum = 0;

        for (j=0; j<n; j++)
            sum += A[i][j] * x[j];

        y[i] = sum;
    }
}

```

Note: A is a 2D array of dimension  $n \times n$ , x and y are 1D arrays of dimension  $n$ .

**Suggested solution:** The `matvec` function implements the mathematical operation of a dense matrix A multiplied with a vector x. The simplest parallelization is to divide the rows of matrix A block-wise to the other MPI processes. This data division also gives a corresponding work division in that each MPI process can individually compute a block of values for the result vector y, provided that vector x is duplicated on all the processes.

The necessary communication consists of (1) letting process 0 distribute the rows of matrix A to the other processes, using the command of `MPI_Scatterv`; (2) letting process 0 broadcast the entire x vector to the other processes, using the command of `MPI_Bcast`; (3) letting every process send back its computed block of vector y to process 0, using the command of `MPI_Gatherv`.

Note: 2D block-wise partitioning of matrix A should be considered only if the number of processes is larger than  $n$ , or if a 2D partitioning gives a much better load balancing than 1D row-wise block partitioning.

## Question 8: MPI implementation (10 points) (IN3200 version)

Based on the high-level sketch that was discussed in the previous question, please program the following MPI-parallel function

```
void matvec_mpi (int n, int **A, int *x, int *y)
```

This function should be called by all the MPI processes. As input,  $A$  is allocated of dimension  $n \times n$  and with values only on process 0, and both  $x$  and  $y$  are allocated of dimension  $n$  and with values only on process 0. (On all the other processes,  $A$ ,  $x$  and  $y$  are empty pointers as the input arguments.) When the function returns, the  $y$  array on process 0 should contain all the newly computed values.

(A quick reference guide of the most important MPI functions is attached.)

### Suggested solution:

```
void matvec_mpi (int n, int **A, int *x, int *y)
{
    int i, j, sum;
    int num_procs, my_rank, start, stop, my_n, *counts, *displs, **my_A, *my_y;

    // assuming MPI_Init has been called before
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

    counts = malloc(num_procs * sizeof(int)); // Note: here was a bug in the solution from 2024
    displs = malloc(num_procs * sizeof(int)); // Note: here was a bug in the solution from 2024

    for (i=0; i<num_procs; i++) { // Note: here was a bug in the solution from 2024
        start = (i*n)/num_procs; stop = ((i+1)*n)/num_procs;
        if (i==my_rank)
            my_n = stop - start;
        counts[i] = (stop - start) * n;
        displs[i] = start * n;
    }

    // allocate my_A
    my_A = malloc(my_n * sizeof(int*));
    my_A[0] = malloc(my_n * n * sizeof(int));
    for (i=1; i<my_n; i++)
        my_A[i] = &(my_A[0][i*n]);

    // distribute matrix A
    MPI_Scatterv (A[0], counts, displs, MPI_INT, my_A[0], my_n*n, MPI_INT, 0, MPI_COMM_WORLD);

    if (my_rank>0)
        x = malloc(n*sizeof(int));
    // broadcast vector x
    MPI_Bcast (x, n, MPI_INT, 0, MPI_COMM_WORLD);

    // allocate my_y
    my_y = malloc(my_n * sizeof(int));

    // local computation per process
    for (i=0; i<my_n; i++) {
        sum = 0;

        for (j=0; j<n; j++)
            sum += my_A[i][j] * x[j];

        my_y[i] = sum;
    }

    for (i=0; i<num_procs; i++) { // Note: here was a bug in the solution from 2024
        counts[i] /= n; displs[i] /= n;
    }
    // process 0 gathers my_y from all processes
    MPI_Gatherv (my_y, my_n, MPI_INT, y, counts, displs, MPI_INT, 0, MPI_COMM_WORLD);
}
```

## Question 7: Parallelization (15 points) (IN4200 version)

We want to use MPI programming to parallelise the function below. Please explain how the data and work can be divided among the process, and what kind of MPI communication will be needed to enable the necessary data exchanges. (No detailed coding is needed here, which will be subject of the next question.)

```
void dijkstra (int n, int s, int **w, int *d)
{
    char *visited = malloc(n*sizeof(char));
    int i,u,v, min_d;

    for (i=0; i<n; i++)
        visited[i] = 0;

    for (i=0; i<n; i++)
        if (w[s][i]>0)
            d[i] = w[s][i];
        else
            d[i] = infinity;

    d[s] = 0;
    visited[s] = 1;

    for (i=1; i<n; i++) {
        u = -1;
        min_d = infinity;

        for (v=0; v<n; v++)
            if (!visited[v] && d[v]<min_d) {
                u = v; min_d = d[v];
            }

        if (u>-1) {
            visited[u] = 1;
            for (v=0; v<n; v++)
                if (!visited[v] && w[u][v]>0 && (d[v] > d[u]+w[u][v]))
                    d[v] = d[u]+w[u][v];
        }
    }

    free(visited);
}
```

Note:  $w$  is a 2D array of dimension  $n \times n$ ,  $d$  is a 1D array of dimension  $n$ .

**Suggested solution:** The work division is through dividing vector  $d$  into blocks and letting each process update one block. To achieve that, it is very important to let process 0 distribute matrix  $w$  **column-wise** in blocks to the other processes. Note, however, that the values of  $w$  to be distributed to each process are not contiguously located in memory, thus individual `MPI_Send` and `MPI_Recv` have to be used (because it doesn't work to simply call `MPI_Scatterv`).

Inside each  $i$ -indexed iteration, the  $v$ -indexed for-loop can be parallelized by first letting each process find a local `min_d` value over its designed part of vector  $d$ , and then using `MPI_Allreduce` (with `MPI_MINLOC`) to find the global `min_d` value. The global  $u$  index then needs to be broadcasted from the process that finds the global `min_d` value. Thereafter, updating each process's designated part of  $d$  can be carried out without any communication.

Finally, the entire  $d$  vector can be gathered from all the processes using `MPI_Gatherv`.

## Question 8: MPI implementation (10 points) (IN4200 version)

Based on the high-level sketch that was discussed in the previous question, please program the following MPI-parallel function

```
void dijkstra_mpi (int n, int s, int **w, int *d)
```

This function should be called by all the MPI processes. As input,  $w$  is allocated of dimension  $n \times n$  (and with values) only on process 0, and  $d$  is allocated of dimension  $n$  only on process 0. (On all the other processes,  $w$  and  $d$  are empty pointers as the input arguments.) When the function returns, the  $d$  array on process 0 should contain all the computed values.

Hint: The `MPI_Allreduce` function also accepts a so-called `MPI_MINLOC` operation, which can be used to find out on which process the minimum value is located. The following is an example:

```
int input[2], result[2];
input[0] = my_value;
input[1] = my_rank;
MPI_Allreduce (input, result, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
printf("The minimum value is %d, found on process with rank %d\n", result[0], result[1]);
```

(A quick reference guide of the most important MPI functions is attached.)

### Suggested solution:

```
void dijkstra_mpi (int n, int s, int **w, int *d)
{
    int num_procs, my_rank, my_n, my_offset, **my_w, *my_d, *my_visited, my_min_d, my_u;
    int input[2], result[2];
    int i, u, v, min_d;

    // assuming MPI_Init has been called before
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);

    my_offset = (my_rank*n)/num_procs;
    my_n = ((my_rank+1)*n)/num_procs - my_offset;

    // allocate my_d on each process
    my_d = malloc(my_n * sizeof(int));

    // distribute matrix w from process 0 to all other processes
    if (my_rank==0) {
        int *sendbuf = malloc(n * (n/num_procs+1) * sizeof(int));
        for (i=1; i<num_procs; i++) {
            int counter = 0, start = (i*n)/num_procs; stop = ((i+1)*n)/num_procs;
            for (u=0; u<n; u++)
                for (v=start; v<stop; v++)
                    sendbuf[counter++] = w[u][v];
            MPI_Send (sendbuf, n*(stop-start), MPI_INT, i, 0, MPI_COMM_WORLD);
        }
        free (sendbuf);
        my_w = w; // use w directly on process 0
    }
    else { // my_rank > 0
        // allocate my_w
        my_w = malloc(n*sizeof(int*));
        my_w[0] = malloc(n * my_n * sizeof(int));
        for (i=1; i<n; i++)
            my_w[i] = &(my_w[0][i*my_n]);
        MPI_Status status;
        MPI_Recv (my_w[0], n*my_n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
}
```

```

my_visited = malloc(my_n*sizeof(char));

for (i=0; i<my_n; i++)
    my_visited[i] = 0;

for (i=0; i<my_n; i++)
    if (my_w[s][i]>0)
        my_d[i] = my_w[s][i];
    else
        my_d[i] = infinity;

if (s>=my_offset && s<my_offset+my_n) {
    my_d[s-my_offset] = 0;
    my_visited[s-my_offset] = 1;
}

for (i=1; i<n; i++) {
    my_u = -1;
    my_min_d = infinity;

    for (v=0; v<my_n; v++)
        if (!my_visited[v] && my_d[v]<my_min_d) {
            my_u = v; my_min_d = my_d[v];
        }

    input[0] = my_min_d; input[1] = my_rank;
    MPI_Allreduce (input, result, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
    min_d = result[0];

    if (min_d < infinity) {
        int root = result[1];
        if (my_rank==root) {
            my_visited[my_u] = 1;
            u = my_u + my_offset;
        }

        MPI_Bcast (&u, 1, MPI_INT, root, MPI_COMM_WORLD); // broadcast the global index u

        for (v=0; v<my_n; v++)
            if (!my_visited[v] && my_w[u][v]>0 && (my_d[v] > min_d+my_w[u][v]))
                my_d[v] = min_d+my_w[u][v];
    }
}

// gather the parts of d from all processes
int *counts = malloc(num_procs * sizeof(int));
int *displs = malloc(num_procs * sizeof(int));
for (i=0; i<num_procs; i++) {
    displs[i] = (i*n)/num_procs;
    conunts[i] = ((i+1)*n)/num_procs - displs[i];
}
MPI_Gatherv (my_d, my_n, MPI_INT, d, counts, displs, MPI_INT, 0, MPI_COMM_WORLD);

free(counts); free(displs); free(my_visited); free(my_d);
if (my_rank>0) {
    free(my_w[0]); free(my_w);
}
}

```