

Suggested solutions for the IN3200/IN4200 exam of spring 2023

Question 1: Distribution of tasks (10 points)

There are in total 100 tasks of different sizes. Task 1 requires 1 minute to complete by one worker, task 2 requires 2 minutes, task 3 requires 3 minutes, and so on.

If there are seven workers, how much time minimum is needed for the seven workers to complete all the 100 tasks? Please explain your answer. (Note: All the tasks are independent of each other, and each task can only be executed by one worker.)

Suggested solution: We can create 49 *task pairs* by grouping task 3 and task 100 together, task 4 and task 99 together, task 5 and task 98 together, and so on. Note that these 49 task pairs all require 103 minutes to complete by one worker. Thus the seven workers can each take 7 such task pairs, needing $7 \times 103 = 721$ minutes to complete. Before assigning the remaining two tasks (task 1 and task 2), we let the worker (say worker 1) who has got task 3 to exchange with the worker (say worker 2) who has got task 4. After the exchange worker 1 needs 722 minutes to complete its assigned tasks, whereas worker 2 needs 720 minutes. Then task 2 can be assigned to worker 2 who will now also need 722 minutes in total. Task 1 can be given to any of the five other workers. Therefore, in total **722** minutes are needed to complete all the 100 tasks by seven workers.

Question 2: SPMD (5 points)

What does "SPMD" stand for in the context of parallel programming? Please give a short explanation.

Suggested solution: "SPMD" stands for *single program multiple data* in the context of parallel programming. It mainly refers to the observation that many computational problems can be parallelized by dividing the data among multiple processing units, which work concurrently on the different parts of the data. It also means that all the processing units execute the *same* program (that is, there is no need to implement different programs for the different processing units).

Question 3: Loop fusion (5 points)

Please give one example of loop fusion. What are the benefits and possible disadvantages of loop fusion?

Suggested solution: Loop fusion means merging several adjacent loops into one. The following is an example of loop fusion:

```
/* before loop fusion */
for (i=0; i<n; i++)
    a[i] = b[i] + c[i];

for (i=0; i<n; i++)
    d[i] = b[i] + e[i];

/* after loop fusion */
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    d[i] = b[i] + e[i];
}
```

Loop fusion may reduce the memory traffic (as shown in the above example related to $b[i]$), while also reducing the loop overhead. If, however, each iteration of a fused loop becomes too complicated and thus requires using too many registers, loop fusion may lead to slower speed (due to register spilling).

Question 4: Volumes of memory traffic (15 points)

```
int i, j;
for (i=0; i<n; i++) {
    double tmp = 0.;
    for (j=ptr[i]; j<ptr[i+1]; j++) {
        tmp += arr[j]*vec1[pos[j]];
    }
    vec2[i] = tmp;
}
```

In the above code segment, 'vec1' and 'vec2' are both 1D arrays of length n and contain double-precision values. 'arr' is a 1D array of length m and contains double-precision values, whereas 'pos' is a 1D array of length m and contains int values. (You can assume that m is larger n , and each value in the 'pos'-array is between 0 and $n - 1$.) Moreover, 'ptr' is a 1D array of length $n + 1$ and contains int values, and we also have

$$0 == \text{ptr}[0] \leq \text{ptr}[1] \leq \text{ptr}[2] \leq \dots \leq \text{ptr}[n] == m.$$

On a cache-based microprocessor, where the cache line size is 64 bytes, what will be the minimum (best-case) volume and maximum (worst-case) volume of memory traffic when the above code is executed? Please explain your answer. (You can assume that none of the arrays were cached before the execution.)

Suggested solution: Note that the code segment implements a sparse matrix vector multiply using the compressed row storage format for the matrix (see Section 3.6.1 in the textbook).

The entries in the arrays of `arr` and `pos` are all loaded from memory only once and used only once (no data reuse). Similarly, the entries in array `vec2` are each stored to memory only once. The entries in

the array of `ptr` are each loaded only once and used (up to) twice, and fully cached. The memory load volume associated with the arrays of `arr`, `pos` and `ptr` is thus deterministic as follows:

$$\left\lceil \frac{m \cdot 8}{64} \right\rceil \cdot 64 \text{ bytes} + \left\lceil \frac{m \cdot 4}{64} \right\rceil \cdot 64 \text{ bytes} + \left\lceil \frac{(n+1) \cdot 4}{64} \right\rceil \cdot 64 \text{ bytes}$$

Note: Each double-precision value requires 8 bytes in memory, and each int value requires 4 bytes. The ceiling function $\lceil \cdot \rceil$ is used because the traffic from (to) the memory is in the unit of cache lines.

Similarly, the memory store volume associated with the array of `vec2` is also deterministic as follows:

$$\left\lceil \frac{n \cdot 8}{64} \right\rceil \cdot 64 \text{ bytes}$$

Note: Any compiler will keep the temporary `tmp` variable in the register, thus there is no memory store traffic associated with `tmp`. (On some computers, storing the array of `vec2` to memory will require first loading the same amount from memory. We ignore such extra load traffic here.)

The load traffic related to the array of `vec1`, however, depends on the reuse of its entries in the cache. In the best-case scenario, each needed entry of `vec1` is loaded only once from memory and is then reused inside the cache throughout the computation, so the minimum load volume related to the array of `vec1` (assuming each entry is used at least once) is

$$\left\lceil \frac{n \cdot 8}{64} \right\rceil \cdot 64 \text{ bytes}$$

(In an extreme case, if only entries of `vec1` that are located on the same cache line are repeated used, then the minimum load volume related to `vec1` can be as low as a single cache line, that is, 64 bytes.)

The worst-case scenario is that each time an entry in `vec1` is used, an entire cache line is (re)loaded from memory (that is, no cache reuse of `vec1`), so the maximum load volume related to the array of `vec1` is

$$m \cdot 64 \text{ bytes}$$

To summarize:

$$\text{best-case memory traffic} = \left(\left\lceil \frac{m \cdot 8}{64} \right\rceil + \left\lceil \frac{m \cdot 4}{64} \right\rceil + \left\lceil \frac{(n+1) \cdot 4}{64} \right\rceil + 2 \cdot \left\lceil \frac{n \cdot 8}{64} \right\rceil \right) \cdot 64 \text{ bytes}$$

or

$$\text{best-case memory traffic} = \left(\left\lceil \frac{m \cdot 8}{64} \right\rceil + \left\lceil \frac{m \cdot 4}{64} \right\rceil + \left\lceil \frac{(n+1) \cdot 4}{64} \right\rceil + \left\lceil \frac{n \cdot 8}{64} \right\rceil + 1 \right) \cdot 64 \text{ bytes}$$

$$\text{worst-case memory traffic} = \left(\left\lceil \frac{m \cdot 8}{64} \right\rceil + \left\lceil \frac{m \cdot 4}{64} \right\rceil + \left\lceil \frac{(n+1) \cdot 4}{64} \right\rceil + \left\lceil \frac{n \cdot 8}{64} \right\rceil + m \right) \cdot 64 \text{ bytes}$$

Question 5: OpenMP parallelisation (5 points)

Parallelize the code segment in the previous question by using OpenMP, with the goal of achieving good speedup. (Please add comments if needed.)

Suggested solution: Parallelization is straightforward by parallelizing the outer for-loop:

```
int i, j;
#pragma omp parallel for private(j) schedule(static,8)
for (i=0; i<n; i++) {
    double tmp = 0.;
    for (j=ptr[i]; j<ptr[i+1]; j++) {
        tmp += arr[j]*vec1[pos[j]];
    }
    vec2[i] = tmp;
}
```

Remark: The amount of work per i-iteration can vary considerably, thus a small chunk size is preferred (instead of the default largest-possible chunk size) when using the `static` scheduler. Using the `dynamic` scheduler is another possibility, giving a better load balance but probably at a (much) higher scheduling overhead. Parallelizing the inner for-loop with a `reduction` clause is not recommended, because there might not be enough work and the scheduling/reduction overhead can be too high.

Question 6: Improve the efficiency of OpenMP code (15 points)

Please improve the speed of the following OpenMP program:

```
int n, place;
// values of 'n' and 'place' are provided at run-time, 'place' is guaranteed smaller than 'n'
int *a = malloc(n*sizeof(int));

#pragma omp parallel
{
    int i;

    #pragma omp master
    { a[place] = -1; }

    #pragma omp for
    for (i=0; i<place; i++)
        a[i] = i;

    #pragma omp for
    for (i=place+1; i<n; i++)
        a[i] = i;
}
```

Suggested solution: It is much more efficient to first let all the threads parallelize the entire for-loop. Then, the master thread can assign `a[place]=-1` outside the parallel region.

```
int n, place;
// values of 'n' and 'place' are provided at run-time, 'place' is guaranteed smaller than 'n'
int *a = malloc(n*sizeof(int));
```

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    a[i] = i;

a[place] = -1;
```

Question 7: Improving MPI communication (15 points)

```
double *arr;
int my_rank, P;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
MPI_Comm_size (MPI_COMM_WORLD, &P);

if (my_rank==0) {
    MPI_Status;
    arr = malloc(P*sizeof(double));
    arr[0] = my_rank;
    for (int i=1; i<P; i++)
        MPI_Recv(&(arr[i]), 1, MPI_DOUBLE, i, i, MPI_COMM_WORLD, &status);
}
else {
    MPI_Send(&my_rank, 1, MPI_DOUBLE, 0, my_rank, MPI_COMM_WORLD);
}
```

Part 1: Please propose a new MPI implementation of the above code segment. The new code should still use point-to-point communication and be as efficient as possible. The new code should achieve the same result as the old code. You don't need to write a complete code, a pseudo code with comments is sufficient.

Part 2: How much time does the new code take when run on P processes? You can use the following formula to estimate the time needed to communicate one message of m bytes from one process to another:

$$L + \frac{m}{B}$$

where L and B are two given constants.

Suggested solution: The problem with the code segment is that process 0 can easily become the performance bottleneck. The total time usage will be $(P-1) \cdot (L + \frac{8}{B})$.

Part 1: To improve, we can do the following (assuming for simplicity P is a power of 2):

- First step: process 1 sends (using `MPI_Send`) its value to process 0 (which calls `MPI_Recv`), while process 3 sends its value to process 2, ..., and so on, while process $P-1$ sends its value to process $P-2$.
- Second step: process 2 sends a message containing two values (its own and that from process 3) to process 0, while process 6 sends a message containing two values (its own and that from process 7) to process 4, and so on.

- ...
- Step No. $\log_2 P$ (final step): process $P/2$ sends a message containing $P/2$ values to process 0.

Part 2: There are in total $\log_2 P$ steps, with the total time usage being

$$\sum_{k=1}^{\log_2 P} \left(L + \frac{2^{k-1} \cdot 8}{B} \right) = \log_2 P \cdot L + \frac{(P-1) \cdot 8}{B}$$

Impart remark: *Another* interpretation of this question could be that no communication is strictly speaking needed, because the particular values to be communicated (shown in the code segment) are known precisely by process 0.

Question 8: Implement a communication function (20 points)

We want to partition a 2D grid in the x-direction, such that each process is responsible for a part-grid. The following code segment shows the partitioning and the data structure per process:

```
int has_neigh_left = (my_rank > 0) ? 1 : 0;
int has_neigh_right = (my_rank < P - 1) ? 1 : 0;
int my_offset = (my_rank * nx) / P;
int my_nx = ((my_rank + 1) * nx) / P - my_offset;
double **my_u,
allocate_2D_array(&my_u, (my_nx + has_neigh_left + has_neigh_right), ny);
```

Note: the `allocate_2D_array` function allocates a row-major 2D array with `ny` rows and `(my_nx + has_neigh_left + has_neigh_right)` columns.

Please implement an MPI function with the following syntax:

```
void communicate_left_right (int my_rank, int P, int my_nx, int ny, double **my_u)
```

The purpose is to let each process communicate with its left neighbour (if any) and its right neighbour (if any), to get the "ghost" values from the neighbours.

Suggested solution:

```
void communicate_left_right (int my_rank, int P, int my_nx, int ny, double **my_u)
{
    int i;
    MPI_Status status;

    if (my_rank > 0) {
        double *buffer_in_left = malloc(ny * sizeof(double));
        double *buffer_out_left = malloc(ny * sizeof(double));

        for (i = 0; i < ny; i++)
            buffer_out_left[i] = my_u[i][1]; // packing message
```

```

MPI_Sendrecv (buffer_out_left, ny, MPI_DOUBLE, my_rank-1, 0,
              buffer_in_left, ny, MPI_DOUBLE, my_rank-1, 0, MPI_COMM_WORLD, &status);

for (i=0; i<ny; i++)
    my_u[i][0] = buffer_in_left[i];    // unpacking message

free(buffer_in_left); free(buffer_out_left);
}

if (my_rank<P-1) {
    double *buffer_in_right = malloc(ny*sizeof(double));
    double *buffer_out_right = malloc(ny*sizeof(double));
    int offset = (my_rank==0) ? 0 : 1;

    for (i=0; i<ny; i++)
        buffer_out_right[i] = my_u[i][my_nx-1+offset];    // packing message

    MPI_Sendrecv (buffer_out_right, ny, MPI_DOUBLE, my_rank+1, 0,
                  buffer_in_right, ny, MPI_DOUBLE, my_rank+1, 0, MPI_COMM_WORLD, &status);

    for (i=0; i<ny; i++)
        my_u[i][my_nx+offset] = buffer_in_right[i];    // unpacking message

    free(buffer_in_right); free(buffer_out_right);
}
}

```

Remark: It is also possible to introduce a new MPI data type with help of `MPI_Type_vector`, such that packing/unpacking messages can be handled by MPI internally.

Question 9 (for IN3200 students): Speedup difficulties (10 points)

What can be the reasons for an OpenMP program to not achieve perfect speedup?

Suggested solution: There can be many reasons for an OpenMP program to not achieve perfect speedup:

- There can be non-parallelizable work, that is, can only be executed by a single thread.
- Creating and terminating a parallel region will incur some overhead.
- The scheduler (especially the dynamic or guided scheduler) will incur some overhead.
- Other OpenMP constructs will also incur overhead.
- There can be load imbalance between the threads.
- There can be (implicit) synchronization overhead between the threads.
- The achievable memory bandwidth does not grow proportionally to the number of threads.

- There can be non-local memory accesses by the threads on a NUMA system (especially if first-touch is not properly enforced).
- There can be false sharing (when there are multiple copies of a same cache line inside different private caches, and that one or more threads modify a different part of the same cache line).

Question 9 (for IN4200 students): Estimating time (10 points)

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

When the above OpenMP code is run using 1, 2, 3, and 4 threads on a machine with a single-socket 4-core processor, the time measurements are as follows:

when run with 1 thread: 10 seconds
 when run with 2 threads: 5 seconds
 when run with 3 threads: 4.5 seconds
 when run with 4 threads: 4 seconds

'a' and 'b' are 1D arrays of length 'n'.

How much time will the following OpenMP code need to complete when it runs on the same machine with 4 threads? Please explain your answer.

```
#pragma omp parallel
{
    int thread_id, i, my_start, my_stop;
    thread_id = omp_get_thread_num ();

    if (thread_id == 0) {
        my_start = 0; my_stop = n/10;
    } else if (thread_id == 1) {
        my_start = n/10; my_stop = 3*n/10;
    } else if (thread_id == 2) {
        my_start = 3*n/10; my_stop = 6*n/10;
    } else {
        my_start = 6*n/10; my_stop = n;
    }

    for (i=my_start; i<my_stop; i++)
        a[i] = b[i];
}
```

Suggested solution: We first note that thread 1 executes twice the amount of memory traffic as thread 0, thread 2 executes three times the amount, whereas thread 3 executes four times the amount. Therefore, thread 0 will finish first, followed respectively by threads 1, 2, 3.

When thread 0 finishes its share of memory traffic (10% of the total amount), due to fairness of competition, the other three threads should each have also completed 10% of the total amount. That is,

40% of the total amount is completed when thread 0 is done. According to the earlier 4-thread time measurement, it takes 4 seconds to complete the total amount when all the four threads compete against each other, it will thus take $40\% \times 4 = 1.6$ seconds when thread 0 is done in the current case. After thread 0 has completed, the remaining three threads will continue to work side-by-side, thus thread 1 will need an additional $30\% \times 4.5 = 1.35$ seconds to finish after thread 0. Similarly, thread 2 will need an additional $20\% \times 5 = 1$ second to finish after thread 1, whereas thread 3 will need an additional $10\% \times 10 = 1$ second to finish after thread 2. The total time needed by thread 3 (which finishes last) is thus

$$1.6 + 1.35 + 1 + 1 = 4.95 \text{ seconds}$$