

Suggested solutions for the IN3200/IN4200 exam of spring 2025

Question 1 (5 points)

Which of the following hardware techniques are available on modern CPUs?
Select one or more alternatives:

- Thread warping
- Pipelining
- SPMD
- Prefetching
- Branch prediction

Suggested solution: Three of the above five alternatives are hardware technologies available on modern CPUs: pipelining, prefetching & branch prediction.

Question 2 (5 points)

Which of the following statements are incorrect?
Select one or more alternatives:

- False sharing can happen in both OpenMP and MPI programs.
- Deadlocks can happen in both OpenMP and MPI programs.
- MPI programs can only run on distributed-memory systems.
- All variables in an OpenMP program will be shared between the threads inside a parallel region.
- Every MPI communicator includes all the processes that are started.

Suggested solution: All the above five statements, except "Deadlocks can happen in both OpenMP and MPI programs", are incorrect.

Question 3 (7 points)

Please optimise the following code segment with respect to time usage and provide a short explanation.

```
for (i=0; i<n; i++) {  
    if (a[i]>0)  
        d[i] = b[i];  
    else  
        d[i] = c[i];  
}
```

Note: a, b, c, d are all 1D arrays of length n, containing values of type 'int'. Specifically, each value in array a is either 0 or 1, but we don't know the distribution of 0s and 1s.

Suggested solution: The above code segment will run slowly due to the if-test in each iteration. The following reformulated code completely avoids the if-test and will thus improve the speed (despite additional computations):

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

Question 4 (8 points)

What is defined as code balance? Provide a short discussion about how code balance can be used to estimate the achievable performance of a given code on a given computer.

Suggested solution: Code balance is defined, for a given code, as the ratio between the amount of memory traffic and the number floating-point operations executed. If the code balance is calculated/estimated as B_c , and a given computer has a realistically achievable memory bandwidth as b_S , then the expected performance of the code, in terms of number of floating-point operations executed per second, is

$$\min\left(\frac{b_S}{B_c}, P_{\max}\right)$$

where $P_{\max}$ denotes the given computer's theoretical peak floating-point performance.

Question 5 (12 points)

For the following code segment, please estimate the total amount of memory traffic for two extreme scenarios: (1) Worst case: the cache capacity is very small with respect to the problem size; (2) Best case: the cache capacity is very large with respect to the problem size.

```

for (i=0; i<M; i++)
    for (j=0; j<N; j++) {
        double tmp = 0.0;
        for (k=0; k<L; k++)
            tmp += A[i][k]*B[k][j];
        C[i][j] = C[i][j] + tmp;
    }

```

Note: A is a 2D array of dimension $M \times L$, B is a 2D array of dimension $L \times N$, and C is a 2D array of dimension $M \times N$. All these 2D arrays have row-major storage, and contain values of type 'double'. You can assume that all the caches are empty before the code segment is executed, and the size of each cache line is 64 bytes.

Suggested solution: We note that each row of A is used N times during the computation, whereas each column of B is used M times. Therefore, for the “worst-case” scenario, each row of A needs to be (re)loaded from memory N times, and each column of B needs to be (re)loaded from memory M times. This means that the “worst-case” memory traffic volume associated with A is (note: each 'double' value is 8 bytes)

$$M \times N \times L \times 8 \text{ bytes}$$

whereas the “worst-case” memory traffic volume associated with B is

$$M \times N \times L \times 64 \text{ bytes}$$

The reason is that traversing each column of B is done in an extremely “memory-unfriendly” way for a row-major 2D array, which means that (re)loading each value of B, in the worst case, needs (re)loading an entire cacheline (64 bytes). Associated with C, there will be $M \times N \times 8$ bytes load traffic and a same amount of store traffic. (Note: Updating the help variable `tmp` will be done using a register, so no memory traffic will incur.)

For the “best-case” scenario, each value of A is loaded once and (re)used N times (fully cached). Similarly, each value of B is loaded once and (re)used M times (fully cached). So the best-case memory traffic volume associated with A is $M \times L \times 8$ bytes, and $N \times L \times 8$ bytes associated with B. The same memory traffic volume as above is still associated with C.

Question 6 (8 points)

Describe an optimisation method (with sufficient details) that can be applied to the code segment in the previous question, with the purpose of reducing the amount of memory traffic for the situation of cache being small.

Suggested solution: There are many strategies that can reduce the “worst-case” memory traffic volume. The following method is to apply “unroll-and-jam”, with unroll depth equal 2, to the j-indexed loop level:

```

for (i=0; i<M; i++)
    for (j=0; j<N; j+=2) {
        double tmp0 = 0.0, tmp1 = 0.0;

```

```

for (k=0; k<L; k++) {
    tmp0 += A[i][k]*B[k][j];
    tmp1 += A[i][k]*B[k][j+1];
}
C[i][j] = C[i][j] + tmp0;
C[i][j+1] = C[i][j+1] + tmp1;
}

```

Even for a very small cache, the above improvement will reduce the memory traffic associated with A and B both with a factor of 2. (Larger unroll depths may lead to larger benefits with respect to the memory traffic.)

Question 7 (10 points)

On a shared memory system, the obtainable memory bandwidth does not scale perfectly with the number of threads used. This is why running the following OpenMP parallelised loop, with 1, 2, 3, 4 threads, gives in the time measurements shown in the table.

```

#pragma omp parallel for
for (int i=0; i<1000000000; i++)
    A[i] = 1.0*i; // A: 1D array of size 1000000000, containing values of type 'double'

```

Threads used	1	2	3	4
Time usage	0.08 seconds	0.04 seconds	0.03 seconds	0.025 seconds

How will you estimate the time usage, when the following OpenMP code is run with 1,2,3,4 threads, respectively?

```

#pragma omp parallel for
for (int i=0; i<10000; i++)
    for (int j=0; j<=i; j++)
        B[i][j] = 1.0*(i+j); // size of B: 10000x10000, row-major storage, with 'double' values

```

Note: You can ignore the OpenMP and looping overheads.

Suggested solution: From the table, the (aggregate) memory bandwidth achievable by using 1,2,3,4 threads, can be calculated as $B_{1 \text{ thread}} = 10$, $B_{2 \text{ threads}} = 20$, $B_{3 \text{ threads}} = 26.7$ and $B_{4 \text{ threads}} = 32 \text{ GB/s}$, respectively. **It is important to note that these bandwidths are measured when the work (memory traffic) is evenly distributed between the threads.**

The challenge with the above OpenMP code is that the work (memory traffic) distribution between the threads is uneven, although the distribution of the iterations is even. Before detailing the actual work division, let us first discuss how to estimate the time for uneven work division between the threads.

If there are two threads, one with V_1 as the volume of traffic and the other with V_2 (with $V_2 \geq V_1$). The thread with V_1 will finish first, requiring the time usage of $T_1 = (2 \times V_1) / B_{2 \text{ threads}}$. The thread with V_2 will finish second, requiring $T_2 = T_1 + (V_2 - V_1) / B_{1 \text{ thread}}$.

For three threads with volumes $V_1 \leq V_2 \leq V_3$, the time usages for the three threads will be

$$T_1 = \frac{3 \times V_1}{B_{3 \text{ threads}}}, \quad T_2 = T_1 + \frac{2 \times (V_2 - V_1)}{B_{2 \text{ threads}}}, \quad T_3 = T_2 + \frac{V_3 - V_2}{B_{1 \text{ thread}}}.$$

For four threads with volumes $V_1 \leq V_2 \leq V_3 \leq V_4$, the time usages for the four threads will be

$$T_1 = \frac{4 \times V_1}{B_{4 \text{ threads}}}, \quad T_2 = T_1 + \frac{3 \times (V_2 - V_1)}{B_{3 \text{ threads}}}, \quad T_3 = T_2 + \frac{2 \times (V_3 - V_2)}{B_{2 \text{ threads}}}, \quad T_4 = T_3 + \frac{V_4 - V_3}{B_{1 \text{ thread}}}.$$

Now we are ready to estimate the time of the OpenMP code when 1,2,3,4 threads are used. When a single thread is used, the total memory traffic volume is $(1 + 2 + 3 + 4 + \dots + 10^4) \times 8 \text{ bytes} \approx 4 \times 10^8$ bytes. So the time usage is $(4 \times 10^8) / 10 \text{ GB/s} = \mathbf{0.04}$ seconds.

When 2 threads are used, the first thread has its total volume as $(1 + 2 + 3 + 4 + \dots + 5000) \times 8 \text{ bytes} \approx 10^8$ bytes, whereas the second thread has its total volume as $(5001 + 5002 + \dots + 10^4) \times 8 \text{ bytes} \approx 3 \times 10^8$ bytes. So the time usage of the slower thread is **0.03** seconds.

When 3 threads are used, the three threads have approximately the individual total volumes as 4.44×10^7 , 1.33×10^8 and 2.22×10^8 bytes. So the slowest thread will have time usage as about **0.0228** seconds.

When 4 threads are used, the four threads have approximately the individual total volumes as 2.5×10^7 , 7.5×10^7 , 1.25×10^8 , 1.75×10^8 bytes. So the slowest thread will have time usage as about **0.0187** seconds.

Question 8 (10 points)

How will you modify the OpenMP code from the previous question, for the purpose of optimising the speed when using multiple threads?

Please briefly explain the idea first, before modifying the OpenMP code.

Suggested solution: The idea is to enforce an uneven distribution of the iterations, such that an even distribution of the memory traffic is achieved.

```
#pragma omp parallel
{
    int my_start, my_stop, thread_id, num_threads, i, j;

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

    my_start = sqrt(1.0 * thread_id / num_threads) * 10000;
    my_stop = sqrt(1.0 * (thread_id+1) / num_threads) * 10000;

    for (i=my_start; i<my_stop; i++)
        for (j=0; j<=i; j++)
            B[i][j] = 1.0*(i+j);
}
```

Question 9 (10 points)

As described in Chapter 6 of the textbook, the Gauss-Seidel computation

```
for (int k=1; k<kmax-1; k++)
  for (int j=1; j<jmax-1; j++)
    for (int i=1; i<imax-1; i++)
      phi[k][j][i] = (phi[k-1][j][i] + phi[k][j-1][i] + phi[k][j][i-1]
                     + phi[k][j][i+1] + phi[k][j+1][i] + phi[k+1][j][i])/6.0;
```

can be parallelised using wavefronts (as shown in the figure below), where the work partitioning is through dividing the computational points in the j direction as blocks.

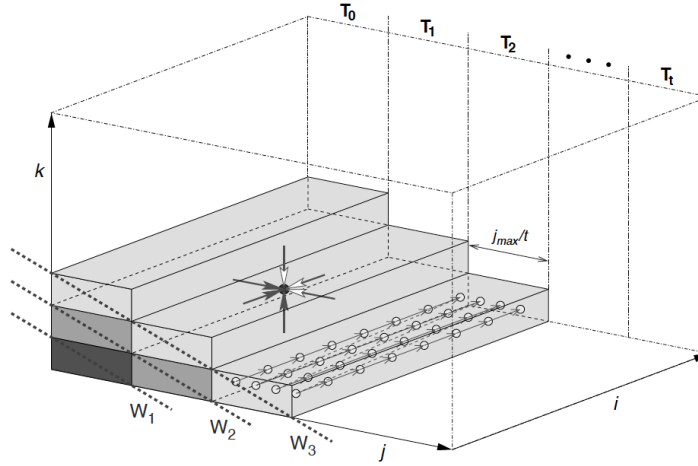


Figure 6.4: Pipeline parallel processing (PPP), a.k.a. wavefront parallelization, for the Gauss-Seidel algorithm in 3D (wind-up phase). In order to fulfill the dependency constraints of each stencil update, successive wavefronts ($W_1, W_2, \dots, W_n$) must be performed consecutively, but multiple threads can work in parallel on each individual wavefront. Up until the end of the wind-up phase, only a subset of all t threads can participate.

Please derive the achievable speedup when using P OpenMP threads or MPI processes. (You can ignore the OpenMP or MPI overhead.)

Suggested solution: Note that there are in total $kmax+P-3$ wavefronts. The time usage needed by each wavefront, when ignoring the OpenMP or MPI overhead, is proportional to $(jmax-2)/P$. (Here, we have assumed that the number of computational points in the j -direction, $jmax-2$, is divisible by P .) Therefore, the parallel time usage is proportional to

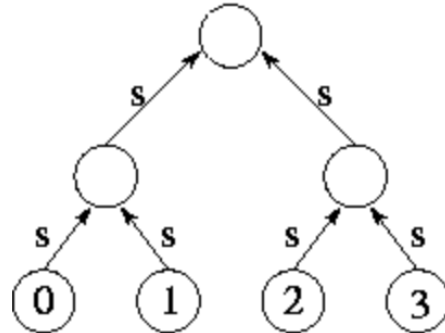
$$(kmax + P - 3) \times \frac{jmax-2}{P},$$

whereas the sequential time usage is proportional to $(kmax-2) \times (jmax-2)$. Thus, the achievable speedup is

$$\frac{kmax - 2}{kmax + P - 3} \times P.$$

Question 10 (IN3200 version, 10 points)

The following figure illustrates how the principle of 'reverse recursive bisection' can be used to let 4 MPI processes collaborate to sum up 4 values (one value per MPI process) using point-to-point communications. The summation result is obtained on only one process in the end.



Please extend the situation to 8 MPI processes each initially having one value. Describe in words which processes need to communicate with which processes during the different steps. You can assume that the final result of summation should be obtained on process 0.

Suggested solution: Step 1: **simultaneously**, process 1 sends its initial value to process 0, process 3 sends its initial value to process 2, process 5 sends its initial value to process 4, process 7 sends its initial value to process 6; whereas processes 0, 2, 4, 6 each receives one value and adds it on top of its initial value.

Step 2: **simultaneously**, process 2 sends its current value to process 0, process 6 sends its current value to process 4; whereas processes 0 and 4 each receives one value and adds it on top of its current value.

Step 3: process 4 sends its current value to process 0, which then adds the received value on top of its current value.

Question 11 (IN3200 version, 15 points)

Follow the idea from the previous question and implement the following function, which is supposed to be called by 8 MPI processes, each with a unique 'my_rank' between 0 and 7 and having its own 'my_number' as input.

```
int para_sum (int my_rank, int my_number)
```

The returned value from the function on process 0 should be the sum of these 8 input values. (The returned value on the other processes can be arbitrary.) Note: You can only use point-to-point MPI communication, not collective communication. (A brief overview of major MPI commands can be found in the attached document.)

Please insert comments where necessary.

Suggested solution:

```

int para_sum (int my_rank, int my_number)
{
    int source, dest, value;
    MPI_Status status;

    // step 1
    if (my_rank%2) {
        dest = my_rank-1;
        MPI_Send (&my_number, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
    }
    else {
        source = my_rank+1;
        MPI_Recv (&value, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        my_number = my_number + value;
    }

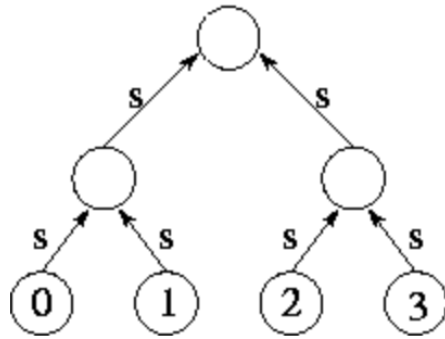
    // step 2
    if (my_rank==2 || my_rank==6) {
        dest = my_rank-2;
        MPI_Send (&my_number, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
    }
    else if (my_rank==0 || my_rank==4) {
        source = my_rank+2;
        MPI_Recv (&value, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
        my_number = my_number + value;
    }

    // step 3
    if (my_rank==4)
        MPI_Send (&my_number, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
    else if (my_rank==0) {
        MPI_Recv (&value, 1, MPI_INT, 4, 3, MPI_COMM_WORLD, &status);
        my_number = my_number + value;
        return my_number;
    }
}

```

Question 10 (IN4200 version, 10 points)

The following figure illustrates how the principle of 'reverse recursive bisection' can be used to let 4 MPI processes collaborate to sum up 4 values (one value per MPI process) using point-to-point communications. The summation result is obtained on only one process in the end.



Please extend the situation to 8 MPI processes each initially having one value. In addition, please also modify the strategy such that the summation result becomes available on all the 8 processes. Describe in words which processes need to communicate with which processes during the different steps.

Suggested solution: Step 1: **simultaneously**, processes 1 & 0 exchange their initial values (by sending one message from process 1 to process 0 and another message from process 0 to process 1); processes 3 & 2 exchange their initial values; processes 5 & 4 exchange their initial values; processes 7 & 6 exchange their initial values. Then, each process adds the received value to its initial value.

Step 2 is the same as Step 1 but with a different pairing of the processes: **simultaneously**, processes 2 & 0 exchange their current values; processes 3 & 1 exchange their current values; processes 6 & 4 exchange their current values; processes 7 & 5 exchange their current values. Then, each process adds the received value to its current value.

Step 3 uses yet another pairing of the processes: **simultaneously**, processes 4 & 0 exchange their current values; processes 5 & 1 exchange their current values; processes 6 & 2 exchange their current values; processes 7 & 3 exchange their current values. Then, each process adds the received value to its current value.

Question 11 (IN4200 version, 15 points)

Follow the idea from the previous question and implement the following function which is supposed to be called by 8 MPI processes, each with a unique 'my_rank' between 0 and 7 and having its own 'my_number' as input.

```
int para_sum (int my_rank, int my_number)
```

The returned value from the function should be the sum of these 8 input values on all the 8 processes. Note: You can only use point-to-point MPI communication, not collective communication. (A brief overview of major MPI commands can be found in the attached document.)

Please insert comments where necessary.

Suggested solution:

```
int para_sum (int my_rank, int my_number)
{
    int other_rank, value;
```

```

MPI_Status status;

// step 1
if (my_rank%2)
    other_rank = my_rank-1;
else
    other_rank = my_rank+1;
MPI_Sendrecv (&my_number,1,MPI_INT,other_rank,1,
               &value,1,MPI_INT,other_rank,1,MPI_COMM_WORLD,&status);
my_number = my_number + value;

// step 2
if ((my_rank%4)<=1)
    other_rank = my_rank+2;
else
    other_rank = my_rank-2;
MPI_Sendrecv (&my_number,1,MPI_INT,other_rank,2,
               &value,1,MPI_INT,other_rank,2,MPI_COMM_WORLD,&status);
my_number = my_number + value;

// step 3
if (my_rank<4)
    other_rank = my_rank+4;
else
    other_rank = my_rank-4;
MPI_Sendrecv (&my_number,1,MPI_INT,other_rank,3,
               &value,1,MPI_INT,other_rank,3,MPI_COMM_WORLD,&status);
my_number = my_number + value;

return my_number;
}

```