Suggested solutions for the IN3200/IN4200 final home exam of spring 2021

This home exam accounts for 60% of the final grade

The first and second home exams account for 20% each. The final grade is A–F.

1 Estimating memory traffic (10 points)

Explain how much memory traffic (in bytes, as a function of N), between the main memory and the last-level cache, can maximumly be caused by running the following code segment:

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    z[i+j] = z[i+j] + x[i]*y[j];
  }
}
```

Here, `z` is a 1D array of length $2N-1$, while `x` and `y` are 1D arrays of length N. All the arrays contain "double-precision floating point" values.

Suggested solution: The maximum volume of memory traffic, detailed below, will happen when the last-level cache is not large enough to hold at least $2N$ double-precision floating point values (each occupies 8 bytes in memory). In such cases, the total memory traffic will consist of the following parts:

- The total memory *load* traffic associated with the `y` array will be $8N^2$ bytes. This is because for each `i`-indexed outer iteration, the entire `y` array has to be reloaded from the main memory (while going through the `j`-indexed inner for loop).
- The total memory *load* traffic associated with the `z` array will also be $8N^2$ bytes. This is because for each `i`-indexed outer iteration, N values of the `z` array have to be reloaded from the main memory.
- The total memory *store* traffic associated with the `z` array will be the same: $8N^2$ bytes.
- For the `x` array, however, each of its values loaded from the memory will remain in cache (possibly also in register) throughout an entire `j`-indexed for loop. Thus the total memory *load* traffic associated with the `x` array is only $8N$ bytes.

The total amount of memory traffic is therefore estimated as $24N^2 + 8N$ bytes. This estimate is in fact not 100% accurate due to the following reason.

Cacheline considerations: The actual memory traffic volume is always a multiple of the cacheline size, because the unit of all memory traffic is a cache line. So, if N is not divisible by the number of double-precision values that fit into a cache line, there will be *slightly* more memory traffic than the above estimate. This is because the last cache line loaded/stored for `x`, `y`, `z` will not be fully used. A similar cacheline "inefficiency" can also happen with the first cache line loaded/stored, if the start memory addresses of `x` and `y` are not cacheline-aligned. During each `i`-indexed outer iteration, the work on the `z` array starts with the its `z[i]` entry, whose memory address is in most cases not cacheline-aligned.

2 Processing dependent tasks (10 points)

The figure below shows the dependency relationship between 15 tasks. Each directed edge in the figure connects a pair of "source" and "destination" tasks. A destination task cannot be started until all its source tasks are carried out. Each task requires ten time units of a worker to complete. (We also assume that it is not possible to let two or more workers collaborate on one task for a faster execution.)

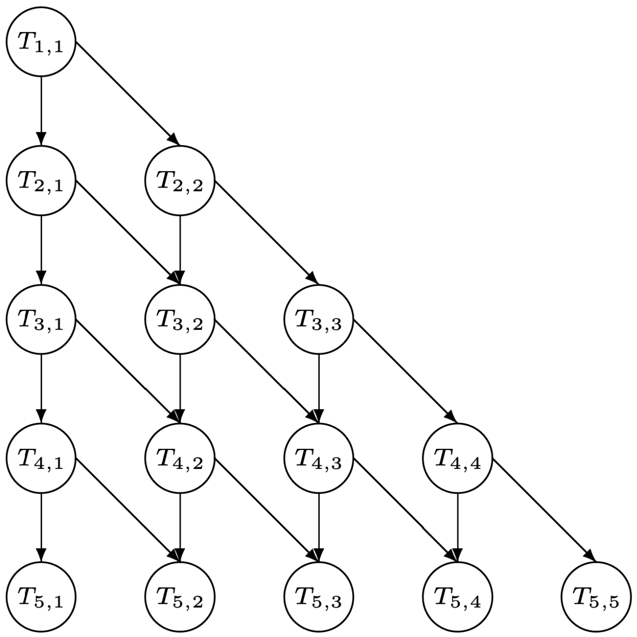

Figure 1: 15 dependent tasks

Note that in case a pair of "source" and "destination" tasks are assigned to two different workers, it will incur an extra data communication cost of one time unit. (No communication is needed if a destination node and all its source node are assigned to the same worker.) Each needed communication will automatically start as soon as the source task is completed, and multiple communications can take place by themselves simultaneously.

Explain how many time units minimum do 3 workers need to finish all the 15 tasks.

Suggested solution: Table 1 shows the details of one strategy for executing the 15 tasks by using three workers. (There are other equally fast strategies.) The minimum time units needed is 62. Here, it is assumed that each communication does not involve the worker who completes the "source" task. Another interpretation is that the worker who completes the "source" task has to spend 1 time unit to execute the communication. For such an interpretation, Table 2 shows the details of another strategy for executing the 15 tasks by using three workers, for which the minimum time units needed is 65.

Both strategies will be considered correct, due to the different interpretations of whether a communication executes itself automatically, or involves the worker who completes the "source" task.

$T_{1,1}$ by worker 1 start: 0, finish: 10 auto comm. to $T_{2,2}$ done: 11				
$T_{2,1}$ by worker 1 start: 10, finish: 20 auto comm. to $T_{3,2}$ done: 21	$T_{2,2}$ by worker 2 start: 11, finish: 21 auto comm. to $T_{3,3}$ done: 22			
$T_{3,1}$ by worker 1 start: 20, finish: 30 auto comm. to $T_{4,2}$ done: 31	$T_{3,2}$ by worker 2 start: 21, finish: 31	$T_{3,3}$ by worker 3 start: 22, finish: 32 auto comm. to $T_{4,3}$ done: 33		
$T_{4,1}$ by worker 1, start: 30, finish: 40	$T_{4,2}$ by worker 2, start: 31, finish: 41 auto comm. to $T_{5,2}$ done: 42	$T_{4,3}$ by worker 2 start: 41, finish: 51 auto comm. to $T_{5,4}$ done: 52	$T_{4,4}$ by worker 3 start: 32, finish: 42	
$T_{5,1}$ by worker 1 start: 40, finish: 50	$T_{5,2}$ by worker 1 start: 50, finish: 60	$T_{5,3}$ by worker 2 start: 51, finish: 61	$T_{5,4}$ by worker 3 start: 52, finish: 62	$T_{5,5}$ by worker 3 start: 42, finish: 52

Table 1: One way of executing the 15 tasks by using three workers. Here, the assumption is that the worker who completes the "source" task does not need to involve himself in the *automatic* communication.

$T_{1,1}$ by worker 1 start: 0, finish: 10 comm. to $T_{2,1}$ by w1, done: 11				
$T_{2,1}$ by worker 1 start: 11, finish: 21 comm. to $T_{3,2}$ by w1, done: 22	$T_{2,2}$ by worker 2 start: 11, finish: 21 comm. to $T_{3,3}$ by w2, done: 22			
$T_{3,1}$ by worker 1 start: 22, finish: 32 comm. to $T_{4,2}$ by w1, done: 33	$T_{3,2}$ by worker 2 start: 22, finish: 32	$T_{3,3}$ by worker 3 start: 22, finish: 32 comm. to $T_{4,3}$ by w3, done: 33		
$T_{4,1}$ by worker 1, start: 33, finish: 43	$T_{4,2}$ by worker 2, start: 33, finish: 43 comm. to $T_{5,2}$ by w2, done: 44	$T_{4,3}$ by worker 2 start: 44, finish: 54 comm. to $T_{5,4}$ by w2, done: 55	$T_{4,4}$ by worker 3 start: 33, finish: 43	
$T_{5,1}$ by worker 1 start: 43, finish: 53	$T_{5,2}$ by worker 1 start: 53, finish: 63	$T_{5,3}$ by worker 2 start: 55, finish: 65	$T_{5,4}$ by worker 3 start: 55, finish: 65	$T_{5,5}$ by worker 3 start: 43, finish: 53

Table 2: Another way of executing the 15 tasks by using three workers. Here, the assumption is that the worker who completes the "source" task has to spend 1 time unit to execute the communication.

3 Understanding a given function (3 points)

```
void func (int N, int *arr)
{
  int pass, smallIndex, j, temp;
  for (pass = 0; pass < N-1; pass++)
  {
    smallIndex = pass;
    for (j = pass+1; j < N; j++) {
      if (arr[j] < arr[smallIndex])
      smallIndex = j;
    }
    temp = arr[pass];
    arr[pass] = arr[smallIndex];
    arr[smallIndex] = temp;
  }
}
```

If an array of four integer values $\{3, 1, 9, 5\}$ is passed as input to the above function ”func”, what will be the content of the integer array after the function call? Please explain your finding.

Suggested solution: If an array of four values is passed to the function ”`func`”, the for-loop indexed by `pass` will be executed with three iterations. During the first iteration, the index for value “1” is found, because “1” is the smallest among the four values. So at the end of the first iteration, the array will contain values of $\{1, 3, 9, 5\}$. At the end of the second iteration, the array will still contain values of $\{1, 3, 9, 5\}$, because “3” is already the smallest value among the three last values. At the end of the third iteration, the array will contain values of $\{1, 3, 5, 9\}$. That is, the result of calling the function “`func`” is that the values of the input array are sorted in an increasing order.

Important notice: There was a typo in the function “`func`” in the original exam text. Specifically, there was an unncessary semicolon immediately after the `if` conditional:

```
for (j = pass+1; j < N; j++) {
  if (arr[j] < arr[smallIndex]);
  smallIndex = j;
}
```

So it is also considered to be correct if the student answers, with a sufficient explanation, that the resulting content of the array is $\{5, 3, 1, 9\}$.

4 Parallelism (5 points)

Please explain what kind of parallelism is found in function “`func`”.

Suggested solution: The iterations of the outer for loop (indexed by `pass`) have to be executed in sequence. Parallelism is found with the inner for loop, which finds the index of the smallest array entry, from position `pass` to position `N-1`. This is data parallelism, in the sense that the array entries can be divided among multiple threads, which can *individually* work on the different segments of the array. Each thread first finds the "per-thread local" index of the smallest array entry in its segment. At the end, a parallel reduction can be carried out among the threads, or simply using one thread, to find the "global" index of the smallest array entry.

Important notice: Considering the unnecessary semicolon in the original text of function "`func`", it is also correct if the student argues that there is no parallelism.

5 OpenMP parallelization (7 points)

Write an OpenMP parallelized version of the function "`func`".

Important notice: Due to the potential confusion caused by the unnecessary semicolon in the original text of function "`func`", all students will automatically get 7 points. The following OpenMP parallelization is merely for reference.

Suggested solution:

```
#define max_num_threads 128
#define offset 32

void para_func (int N, int *arr)
{
  int pass, smallIndex, j, temp;
  int index_array[max_num_threads*offset];

#pragma omp parallel private(pass, smallIndex, j)
 {
  int thread_id, num_threads;
  thread_id = omp_get_thread_num();
  num_threads = omp_get_num_threads();

  for (pass = 0; pass < N-1; pass++)
  {
    smallIndex = pass;
    #pragma omp for nowait
    for (j = pass+1; j < N; j++) {
      if (arr[j] < arr[smallIndex])
        smallIndex = j;
    }

    // each thread registers the index of its smallest entry
    index_array[thread_id*offset] = smallIndex;

    #pragma omp barrier
    #pragma omp single
    {
      smallIndex = index_array[0];
      for (j=1; j<num_threads; j++)
```

```
        if (arr[index_array[j*offset]] < arr[smallIndex])
          smallIndex = index_array[j*offset];

      temp = arr[pass];
      arr[pass] = arr[smallIndex];
      arr[smallIndex] = temp;
    }  // end of single
  }
 }  // end of parallel region
}
```

6 Detecting MPI errors (15 points)

There are a few errors in the following MPI program. Can you find the errors and correct them? Please explain your corrections.

When the errors are corrected, what will be result of running the MPI program with 5 MPI processes?

```
#include <mpi.h>
#include <stdio.h>

int main (int nargs, char **args)
{
  int rank, size, len, i, *arr_out, *arr_in;
  int send_to, recv_from;
  MPI_Status status;

  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  if (rank==0) {
    len = 100;
    MPI_Bcast (&len, 1, MPI_INT, 0, MPI_COMM_WORLD);
  }
  else
    MPI_Recv (&len, 1, MPI_INT, 0, 101, MPI_COMM_WORLD, &status);

  arr_out = (int*)malloc(len*sizeof(int));
  arr_in = (int*)malloc(len*sizeof(int));
  for (i=0; i<len; i++)
    arr_out[i] = rank*len + i;

  recv_from = (rank+1)%size;
  send_to = (rank-1+size)%size;

  MPI_Recv (&arr_in, len, MPI_INT, recv_from, 100*rank, MPI_COMM_WORLD, &status);
  MPI_Send (&arr_out, len, MPI_INT, send_to, 100*rank, MPI_COMM_WORLD);
  MPI_Finalize ();
  free(arr_in);
  free(arr_out);
  printf("On rank <%d>, first value received=%d\n", arr_in[0]);

  return 0;
}
```

Suggested solution: The above MPI program has the following errors:

1. The first call to `MPI_Recv`, which is invoked by all the processes with `rank` larger than 0, is a mis-match with the `MPI_Bcast` call on process 0. This will lead to a deadlock of all the processes with `rank` larger than 0. (To correct, all the processes should simply call `MPI_Bcast`.)
2. The second call to `MPI_Recv`, which is invoked by all the processes and followed by `MPI_Send`, will lead to a guaranteed deadlock of all processes (no matter how much data is inside the message). One solution to avoiding the deadlock is to let even-numbered processes start with `MPI_Recv` whereas the other processes start with `MPI_Send`.
3. For the second call to `MPI_Recv`, the initial address of the receive buffer should be `arr_in` (instead of `&arr_in`). Similarly, for `MPI_Send`, the initial address of the send buffer should be `arr_out` (instead of `&arr_out`).
4. For each pair of `MPI_Send` and `MPI_Recv` calls, there is a mis-match with respect to the message tags used. A correction can be to instead use `100*send_to` as the message tag for the call to `MPI_Send`.
5. The call to function `printf` should include `rank` as the second parameter, while `arr_in[0]` should be used as the third parameter.
6. The function `MPI_Finalize` is called too early (should be moved to the end of the program, before the `return` statement).
7. The call to `free(arr_in)` is invoked too early (should be moved after `printf`, but before `MPI_Finalize`).

If the above errors are corrected, the execution result of running five MPI processes will be as follows (the sequence is however non-deterministic):

```
On rank <0>, first value received=100
On rank <1>, first value received=200
On rank <2>, first value received=300
On rank <3>, first value received=400
On rank <4>, first value received=0
```

7 When is peak performance possible? (10 points)

Please provide your viewpoints on what characteristics should a code have, in order to have the possibility of achieving max FLOPS performance on a modern multicore CPU.

Suggested solution: The max FLOPS performance on a modern multicore CPU is only achievable when all the CPU cores are constantly busy with vectorized floating-point operations. Specifically, the following characteristics (not an exhaustive list) are required:

1. The code balance $B_{\mathrm{c}} = \frac{\text{data traffic}}{\text{floating point ops}}$ of the code is lower than the machine balance for the entire multicore CPU, so that the performance is not limited by the data movement inside the memory system.

2. The computations are implemented as very long loops (so that the operations are fully pipelined with negligible loop overhead), where the same floating-point operations are applied to different data elements. The latter is important for achieving the SIMD vectorized capability of modern processors (which is the main source of max FLOPS performance). The loops should regularly access the memory (to allow effective hardware prefetch) and should not contain conditionals that can lead to branch-prediction errors on the hardware.

3. The computations are in the form of simple arithmetic operations of (to avoid pipeline bubbles). Also, the amounts of additions and multiplications should be interleaved and evenly matched, so that the *fused-multiply-add* (FMA) capability of a modern CPU is fully utilized.

4. The computations involved should have sufficient parallelism and that the entire computational work can be evenly divided between the CPU cores (to avoid load imbalance).

5. The overhead due to communication needed between the CPU cores should be negligible (or completely hidden).