Suggested solutions for the IN3200/IN4200 exam of spring 2019

Problem 1.1: Improving code performance

Explain why the following code will probably run very slow. How will you modify the code such that the same computation is done but the performance is considerably improved?

```
for (int i=0; i<n; i++) {
  c[i] = exp(1.0*i/n)+sin(3.1415926*i/n);
  for (int j=0; j<n; j++)
    a[j][i] = b[j][i] + d[j]*e[i];
}
```

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

Suggested solution: The main problem with the above code is that entries of the 2D arrays `a` and `b` are accessed in a column-major order, contrary to their row-major storage in memory. In practice, when n is large enough, every access of `a[j][i]` and `b[j][i]` incurs a cache miss, seriously enlarging the amount of data movement in the entire memory system. A secondary problem with the above code is that not all the calls to the expensive math functions `exp` and `sin` are necessary. Some of them can be either avoided or replaced with simpler floating-point operations.

To reduce the volume of memory traffic, we can modify the original code segment as:

```
int i, j;
for (i=0; i<n; i++)
  c[i] = exp(1.0*i/n)+sin(3.1415926*i/n);

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = b[i][j] + d[i]*e[j];
```

Now, all the arrays except `e` are accessed in the most economic order. To reduce the memory traffic due to repeatedly loading the entire `e` array, we can unroll the `i`-indexed for-loop such as:

```
for (i=0; i<n; i+=4)
  for (j=0; j<n; j++) {
    a[i][j] = b[i][j] + d[i]*e[j];
```

```
    a[i+1][j] = b[i+1][j] + d[i+1]*e[j];
    a[i+2][j] = b[i+2][j] + d[i+2]*e[j];
    a[i+3][j] = b[i+3][j] + d[i+3]*e[j];
  }
// plus a "remainder code" for the case of (n%4)>0
for (i=n-(n%4); i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = b[i][j] + d[i]*e[j];
```

To simplify the computation associated with calculating the `c` array, we use the observations that $\sin(x) = \sin(\pi - x)$ and $e^{\frac{i}{n}} = \left(e^{\frac{1}{n}}\right)^i$. Therefore, a more efficient implementation is as follows:

```
double ev, exp_div_n=exp(1.0/n), pi_div_n=3.1415926/n;
c[0] = 0.;
for (i=1; i<=(n/2); i++)
  c[i] = sin(i*pi_div_n);
for (i=(n/2)+1; i<n; i++)
  c[i] = c[n-i];  // taking advantage of sin(pi-x)=sin(x)

ev = 1.0;
for (i=0; i<n; i++) {
  c[i] += ev;
  ev *= exp_div_n;
}
```

It should be noted that additional memory traffic is incurred for the purpose of saving half of the calls to the `sin` function. But this should be worthwhile in total.

Problem 1.2: Estimating time usage

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. Can you estimate how much time the following code needs on this CPU, when $n = 10^{10}$?

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

We assume that array `a` is of length n and contains double-precision values.

Suggested solution: The corresponding machine balance B_{m} can be calculated as

$$B_{\mathrm{m}} = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak performance [GFlops/sec]}} = \frac{5 \text{ [GWords/sec]}}{100 \text{ [GFlops/sec]}} = 0.05$$

Note: We have used the fact that each double-precision value occupies 8 bytes in memory.

The code balance B_c of the above code segment is 0.5, because each iteration loads one value of array `a` from memory and executes two floating-point operations. (It is assumed that the variable `s` is kept in register, thus no memory traffic associated with `s` during the iterations.)

Since B_c is much larger than B_m, the speed of the computation is determined by the memory traffic. Therefore, the time required is estimated as

$$\frac{10^{10} \times 8 \text{ bytes}}{40 \times 10^9 \text{ bytes/sec}} = 2 \text{ seconds}$$

Note: To actually achieve the time as described above, we assume that the compiler has enabled the standard optimization of *prefetch*.

Problem 2.1: Processing dependent tasks

Figure 1 shows the dependency relationship between 16 tasks: $T_{1,1}, T_{1,2}, \ldots, T_{4,4}$. Each directed edge in the graph connects a pair of "source" and "destination" tasks. A destination task cannot be started until all its source tasks are carried out. All the 16 tasks are equally time-consuming, requiring one hour of a worker. (We also assume that it is not possible to let two or more workers collaborate on one task for a faster execution.)

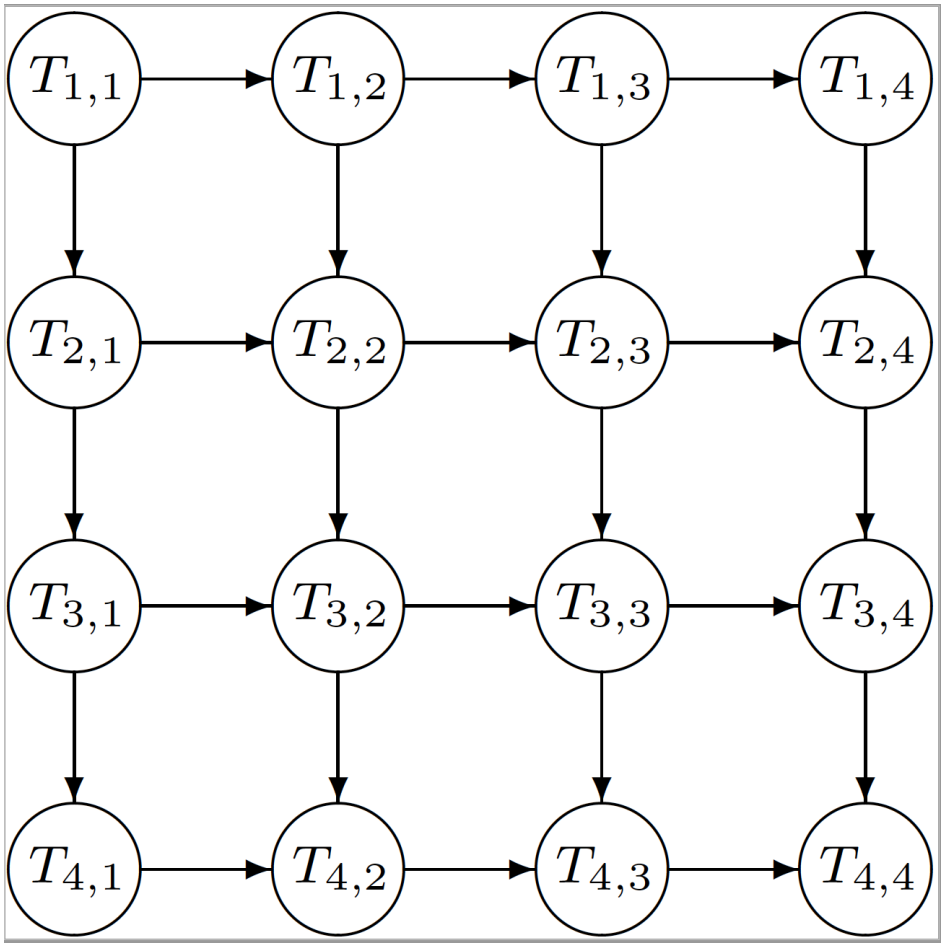

Figure 1: The dependency relationship between 16 tasks.

Explain how many hours minimum do 3 workers need to finish all the 16 tasks.

Suggested solution: The dependency relationship between the tasks means that tasks lying on the same diagonal "wavefront" can be executed simultaneously. The following table shows one efficient way of executing the 16 tasks by 3 workers:

During hour 1: $T_{1,1}$
During hour 2: $T_{2,1}$ and $T_{1,2}$
During hour 3: $T_{3,1}$, $T_{2,2}$ and $T_{1,3}$
During hour 4: $T_{4,1}$, $T_{3,2}$ and $T_{2,3}$
During hour 5: $T_{1,4}$, $T_{4,2}$ and $T_{3,3}$
During hour 6: $T_{2,4}$ and $T_{4,3}$
During hour 7: $T_{3,4}$
During hour 8: $T_{4,4}$

Therefore, 3 workers need 8 hours to finish all the 16 tasks.

Problem 3.1: OpenMP parallelization

Explain why the following code segment cannot be directly parallelized by inserting "`#pragma omp parallel for`" before the for-loop. If the computation involved in function "`func()`" is very time consuming, how will you modify the code such that OpenMP parallelization becomes possible to speed up the entire computation?

```
for (i=0; i<n-1; i++) {
  u[i] = func(u[i+1]);
}
```

Suggested solution: The problem is that there is loop-carried dependence between the iterations. More specifically, `u[i+1]` should not be updated before `u[i]`, but this is no longer guaranteed if the iterations are divided among OpenMP threads.

In case the function call `func(u[i+1])` is very time consuming, it may pay off to introduce a help array `v` as follows:

```
double *v = (double*)malloc((n-1)*sizeof(double));

#pragma omp parallel for
for (i=0; i<n-1; i++)
  v[i] = func(u[i+1]);

#pragma omp parallel for
for (i=0; i<n-1; i++) {
  u[i] = v[i];
}

free (v);
```

Note: Due to introducing the help array `v`, OpenMP parallelization can now be enforced. The performance benefit because of OpenMP parallelization is assumed to exceed the extra time and memory usage incurred by array `v`.

Problem 3.2: False sharing

In the context of OpenMP programming, what does **false sharing** mean? Please give a very simple example of false sharing.

Suggested solution: **False sharing** happens when two or more OpenMP threads load the *same* cache line from memory to their respective private cache, and that the OpenMP threads update the *different* locations on the same cache line. Due to the *cache coherence* policy, expensive memory traffic has to be carried out and it effectively serializes the updates. A simple example of false sharing is shown in the example below:

```
# assume that 'counter' is an integer array of length num_threads
# assume that 'a' is an integer array of length n

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

Remarks:

- False sharing is not the same as race condition.
- False sharing is not specific for NUMA, it also happens for UMA.

Problem 4.1: Understanding an MPI program

What will be the result of running the following MPI program using 8 MPI processes?

```
#include <mpi.h>
#include <stdio.h>
int main (int nargs, char **args)
{
  int own_value, in_value, out_value, i;
  int rank, size;
  int send_to, recv_from;
  MPI_Status recv_status;
  MPI_Request recv_req;

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

  own_value = rank;
```

```
  recv_from = (rank+2)%size;
  send_to = (rank-2+size)%size;
  out_value = own_value;

  for (i=0; i<(size/2)-1; i++) {
    MPI_Irecv(&in_value, 1, MPI_INT, recv_from, 0, MPI_COMM_WORLD, &recv_req);
    MPI_Send (&out_value, 1, MPI_INT, send_to, 0, MPI_COMM_WORLD);
    MPI_Wait (&recv_req, &recv_status);
    own_value += in_value;
    out_value = in_value;
  }

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

Suggested solution: As long as the total number of MPI processes is an even number, the processes with ranks (0,2,4,...) will form a group, while the processes with ranks (1,3,5,...) will form another group. Within each group, an all-reduction is carried out to sum up all the rank values. For the specific case of 8 MPI processes, the above MPI program will output the following 8 lines (sequence not deterministic):

```
On rank <0>, own_value=12
On rank <1>, own_value=16
On rank <2>, own_value=12
On rank <3>, own_value=16
On rank <4>, own_value=12
On rank <5>, own_value=16
On rank <6>, own_value=12
On rank <7>, own_value=16
```

Problem 5.1: Function `count_occurrence`

Write a serial C function with the following syntax:

```
int count_occurrence (const char *text_string, const char *pattern);
```

Note that this function takes two input arguments: `text_string` and `pattern`. The purpose is to find out many times pattern appears in `text_string`. Some concrete examples are as follows:

```
count_occurrence ("ATTTGCGCAGACCTAAGCA", "GCA")  will return 2
count_occurrence ("AAABCDEFGAAGEREAANMT", "AA")  will return 4
count_occurrence ("ABCDEFGHIJKLMNOPQRSTUVWXYZ", "BTTT")  will return 0
```

Hint: The following two C standard functions from "`string.h`" can be useful:

- The C library function `size_t strlen(const char *str)` computes the length of the string `str` up to, but not including the terminating null character. (For example, `strlen("GCA")` returns 3.)

- The C library function `int strncmp(const char *str1, const char *str2, size_t n)` compares at most the first *n* bytes of `str1` and `str2`. In case `str1` and `str2` are identical for the first *n* bytes, the function will return 0, otherwise the function will return a non-zero value.

Suggested solution:

```
int count_occurrence (const char *text_string, const char *pattern)
{
  int n1, n2, i, m=0;
  n1 = strlen(text_string);
  n2 = strlen(pattern);
  for (i=0; i<n1-n2+1; i++)
    if (strncmp(&(text_string[i]), pattern, n2)==0)
      m++;
  return m;
}
```

Note: It is important that the `i` index stops before `n1-n2+1`. Otherwise it means that you will skip checking the final `n2` characters of string `text_string`.

Problem 5.2: OpenMP parallelization

Implement an OpenMP parallelization of function '`count_occurrence`'.

Suggested solution:

```
int count_occurrence (const char *text_string, const char *pattern)
{
  int n1, n2, i, m=0;
  n1 = strlen(text_string);
  n2 = strlen(pattern);
  #pragma omp parallel for reduction(+:m)
  for (i=0; i<n1-n2+1; i++)
    if (strncmp(&(text_string[i]), pattern, n2)==0)
      m++;
  return m;
}
```

Problem 5.3: MPI parallelization

Write an MPI version of '`count_occurrence`' with the following syntax:

```
int parallel_count_occurrence (const char *text_string, const char *pattern);
```

This function is to be called by all MPI processes, where the input arrays `text_string` and `pattern` are both empty pointers on all MPI processes except on MPI process with rank 0.

You can assume that '`MPI_Init`' has already been executed before function '`parallel_count_occurrence`' is called.

Suggested solution: The main steps involved in an MPI implementation are as follows:

- Process with rank 0 broadcasts the lengths of `text_string` and `pattern`. (Remember: Every process must call `MPI_Bcast`.)
- Every process, except with rank 0, allocates the `char` array `pattern`.
- Process with rank 0 broadcasts the content of array `pattern`.
- Every process prepares two arrays of `int` values, named `sendcounts` and `displs`, as follows:

```
int *sendcounts, *displs;
sendcounts = (int*)malloc(num_procs*sizeof(int));
displs = (int*)malloc(num_procs*sizeof(int));

// assume n1 is the length of text_string and n2 length of pattern
for (i=0; i<num_procs; i++)
  displs[i] = (i*(n1-n2+1))/num_procs;
for (i=0; i<num_procs-1; i++)
  sendcounts[i] = (displs[i+1]-displs[i])+n2-1;
sendcounts[num_procs-1] = n1-displs[num_procs-1];
```

- Every process then allocates an array of `char` values named `my_text_string` of length `sendcounts[my_rank]`.
- Every process thereafter calls `MPI_Scatterv` for the purpose of letting rank 0 distribute the content of array `text_string`.

```
MPI_Scatterv(text_string, sendcounts, displs, MPI_CHAR,
             my_text_string, sendcounts[my_rank], MPI_CHAR,
             0 /*rank of root process*/, MPI_COMM_WORLD);
```

- Every process counts the number of occurrences of `pattern` inside its `my_text_string`.
- Finally, every process calls `MPI_Allreduce` to sum up the global number of occurrences.

Note: It is not recommended to let process with rank 0 simply broadcast the entire `text_string` array, because this potentially leads to a huge waste of memory usage.

Problem 6.1: Overlapping communication with computation (Only for IN4200)

Explain why hybrid MPI+OpenMP programming, in comparison with pure MPI programming, will allow a better chance of asynchronous communication for the purpose of overlapping communication with computation.

Suggested solution: Although non-blocking MPI point-to-point communication routines (such as `MPI_Irecv`) are designed to allow computation to be carried out while the communication is taking place "in the background", a specific implementation of the MPI library may actually "serialize" communication with computation (thus no overlap between them). Hybrid MPI+OpenMP programming means that each MPI process spawns several OpenMP threads. Thus, inside an OpenMP parallel region (on every MPI process), it is now possible to dedicate one OpenMP thread (or a small number of threads) to the communication tasks, while the remaining threads independently carry out the computation tasks. The downside is that work division between the "computational threads" is no longer straightforward. (For example, `#pragma omp for` cannot be used.)