Exam

DVA336 Parallel Systems

8:10–12:30

June 10, 2019

This exam includes 8 problems, for a total of 80 points.

General information

1. This exam is closed book.
2. A simple non-programmable calculator is permitted.
3. Write your answers in English.
4. Make sure your handwriting is readable.
5. Answer each problem on a separate sheet.
6. Write your identification code on all sheets as required for anonymous exams.
7. All answers must be motivated. If necessary, make suitable assumptions.

Grading

The passing marks are 90% for grade 5, 70% for grade 4, and 50% for grade 3. This corresponds to the passing grades A, C, and E in the ECTS grading system.

Contact information

For questions on problems 1, 3-7, please contact Gabriele Capannini, 021-101458.
For questions on problems 2 and 8, please contact Björn Lisper, 021-151709.

Good luck!

1. HW Multithreading allows multiple threads to coexist and share the resources of a single processor. Explain how the following approaches work with related pros and cons:
   (a) [4 p] Coarse-grained Multithreading,
   (b) [4 p] Fine-grained Multithreading,
   (c) [4 p] Simultaneous Multithreading.

Solution:

(a) *Coarse-grained MT* : processor issues instruction from a single thread and switches only on costly stalls (for example a cache miss). Pros: threads continue the execution until a costly stall occurs. Cons: cannot overcome short stalls: when a stall occurs, the pipeline must be reset.

(b) *Fine-grained MT* : processor issues instructions iterating (e.g., round-robin) among different threads by skipping those that are stalled. Pros: it hides the throughput losses due to short or long stalls. Cons: interleaved execution slows down the execution of threads that are ready to execute.

(c) *Simultaneous multithreading* (SMT) exploits dynamic multiple-issue processors: multiple instructions from different threads are issued and any dependence among them is handled dynamically.

(for a more detailed answer, please refer the book paragraphs: Computer Organization and Design – 6.4, Introduction to Parallel Computing – Chap 2)

2. (a) [7 p] Sort the following sequence using Radix Sort:

$$15, 4, 13, 2, 2, 0, 30, 3$$

Show the different steps in your computation.

Solution:

(a) Showing each number as a bit array, we see how the numbers are shifted according to the respective bits in the current "bit slice" (0's to the left, 1's to the right) where the bit slices are taken from the least significant bit and up:

15	**4**	**13**	**2**	**2**	**0**	**30**	**3**		**4**	**2**	**2**	**0**	**30**	**15**	**13**	**3**	
1	0	1	0	0	0	0	1		0	0	0	0	0	1	1	1	
1	0	0	1	1	0	1	1		0	1	1	0	1	1	0	1	
1	1	1	0	0	0	1	0	$\rightarrow$	1	0	0	0	1	1	1	0	$\rightarrow$
1	0	1	0	0	0	1	0		0	0	0	0	1	1	1	0	
0	0	0	0	0	0	1	0		0	0	0	0	1	0	0	0	

4	**0**	**13**	**2**	**2**	**30**	**15**	**3**
0	0	1	0	0	0	1	1
0	0	0	1	1	1	1	1
1	0	1	0	0	1	1	0
0	0	1	0	0	1	1	0
0	0	0	0	0	1	0	0

→

0	**2**	**2**	**3**	**4**	**13**	**30**	**15**
0	0	0	1	0	1	0	1
0	1	1	1	0	0	1	1
0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0

→

0	**2**	**2**	**3**	**4**	**13**	**30**	**15**
0	0	0	1	0	1	0	1
0	1	1	1	0	0	1	1
0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0

→

0	**2**	**2**	**3**	**4**	**13**	**15**	**30**
0	0	0	1	0	1	1	0
0	1	1	1	0	0	1	1
0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	1

Solution: ...

3. (a) [6 p] Describe the following three data layouts and how are they related to AVX/SSE programming: *SoA*, *AoS*, and *AoSoA*.

   (b) [4 p] Consider the following C code:

```
struct point3d_t {
  float x, y, z;
};
point3d_t * mesh;
```

   To which layout corresponds the `mesh` array? Provide two distinct versions of the code in which `mesh` is stored in the two remaining types of data layout.

Solution:

(a) They are different approaches for storing arrays of composite data types in memory. The AoS layout keeps each record as one contiguous block of memory, as is conventional in object-oriented systems. SoA is a layout gathering the data related to the different fields of the 'struct' in different arrays: in this way it is possible to easy use load/store packed SIMD instructions to access data in memory. The AoSoA layout is a mix of the two previous ones and keeps chunks of records as one contiguous block of memory.

(b) The code uses the SoA approach which is more intuitive but less SSE/AVX-oriented. It's better to use the so-called SoA (or AoSoA) and keep data contiguous in memory:

```
struct mesh_t {
  float *x, *y, *z;
```

```
};
mesh_t mesh; //mesh consists of 3 different arrays (SoA)

template <const int n> struct vector3d_t {
  float x[n], y[n], z[n];
};
vector3d_t<8> *mesh; //mesh is an array of chunks (AoSoA)
```

4. Give the definition of the following performance metrics:
   (a) [2 p] *parallel time*,
   (b) [2 p] *parallel work*,
   (c) [2 p] *parallel cost*,
   (d) [2 p] *work-optimal*,
   (e) [2 p] *cost-optimal*.

Solution: Given an algorithm A for an input of size n executed on a m-PRAM. Let S be the sequential algorithm to solve the same problem of A, we have that:

(a) *parallel time* $t_A(n, m)$ denotes the number of parallel time steps required to execute A (a.k.a. *depth*).

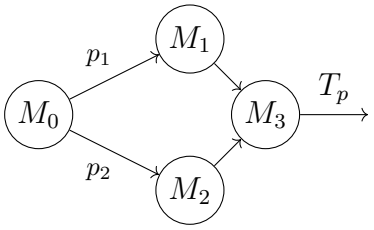
(b) *parallel work* $w_A(n, m)$ denotes the number of instructions performed to execute A.

(c) *parallel cost* $c_A(n, m)$ is defined as $m \cdot t_A(n, m)$.

(d) A is *work-optimal* if $w_A(n, m) = O(t_S(n))$.

(e) A is *cost-optimal* if $c_A(n, m) = O(t_S(n))$.

5. Let S be a system of 4 modules $M_{0..3}$ computing, respectively, the functions $f_{0..3}$ having latency $12t$, $20t$, $10t$, and $15t$ where t is a generic time unit. The modules operate according to the 'OR-logic' and M_0 sends an element to M_1 with probability $p_1 = 2/5$ and to M_2 with probability $p_2 = 1 - p_1$:



   (a) [6 p] What is the interdeparture time T_p of S?

(b) [6 p] What is the interdeparture time T_p of S if $f_3 = 10t$?

Solution:

(a) First, we apply the multiple servers theorem:

$$T_{a_1} = \frac{T_{p_0}}{p_1} = 30t$$

$$T_{a_2} = \frac{T_{p_0}}{p_2} = 20t$$

Hence, we compute the *traffic intensity* for M_1 and M_2 as follows:

$$\rho_1 = \frac{20t}{30t} < 1 \Rightarrow T_{p_1} = 30t$$

$$\rho_2 = \frac{10t}{20t} < 1 \Rightarrow T_{p_2} = 20t$$

Finally, we apply the multiple clients theorem:

$$\frac{1}{T_{a_3}} = \frac{1}{T_{p_1}} + \frac{1}{T_{p_2}} \Rightarrow T_{a_3} = 12t$$

Hence, $\rho_3 = \frac{15}{12} > 1$ so that $T_{p_3} = 15t$ which is also the throughput of S.

(b) Nothing changes until the calculation of ρ_3 which, in this case, becomes $\rho_3 = \frac{10}{12} < 1$ so that $T_{p_3} = 12t$ which is also the throughput of S.

6. Consider the following C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define P 8

void count(int *buckets, const int *A, const int n) {
  #pragma omp parallel num_threads(P)
  {
    const int id = omp_get_thread_num();
    buckets[id] = 0;
    for(int i=0; i<n; ++i) if(A[i]==id) ++buckets[id];
  }
}

int main() {
  int n = 10000000;
  int buckets[P];
  int *A = (int*)malloc(sizeof(int)*n);
```

```
  for(int i=0; i<n; ++i) A[i] = rand()%P;
  count(buckets, A, n);
  for(int i=0; i<P; ++i) printf("buckets[%d]=%d\n", i, buckets[i]);
  free(A);
  return 0;
}
```

(a) [4 p] Describe the run-time behavior of the threads in the `count` function.

(b) [5 p] Discuss the problem affects the performance of the `count` function and how it could be solved.

Solution:

(a) Each one of the `P` threads scan the whole `A` array and counts the number of entries of which value equals the thread's `id`.

(b) The problem which affects the performance of the function is called *false sharing*. It is a phenomenon degrading the performance of the multi-threaded program and it is related to the cache coherence mechanism. When a line of cache is shared among different "sibling" caches and one of the copies is modified by the owning processor, the system invalidates all the other copies by forcing the other processors to reload the cache line. This happens even if the modified variable are not really shared but it fits in the same cache line with other variables, e.g., the `buckets[]` array. To eliminate the *false sharing* we can replace the `buckets[P]` array with a 2D array (i.e., `buckets[P][L]` so that `L` equals the size of the cache line in bytes divided by `sizeof(int)`) storing each counter in `buckets[P][0]` and having each line long as the cache line, i.e., `L` equals the size of the cache line. A different solution is to use a local variable as counter and atomically update the sum before to end the parallel session.

Moreover the function does not improve the complexity w.r.t. the sequential case. To this end, the for should be palatalized by assigning a part of it to each thread. To manage `buckets[P]` that is shared by the threads we can use an atomic increment (which introduce an overhead due to the cache coherence mechanisms) or, even better, instantiate a local copy of `buckets[]` for each thread and, when the loop ends, update (atomically) the shared `buckets[]`.

7. Consider the following MPI function:

```
void MPI_Function(float *px, const int root, MPI_Comm comm) {
  int rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  if(root>=size)
    return;
  if(rank!=root) {
    MPI_Send(px, 1, MPI_FLOAT, root, 100, comm);
```

```
    MPI_Recv(px, 1, MPI_FLOAT, root, 100, comm, MPI_STATUS_IGNORE);
  } else {
    float y, z = 0.0f;
    for(int i=1; i<size; ++i) {
      MPI_Recv(&y, 1, MPI_FLOAT, (rank+i)%size, 100, comm, MPI_STATUS_IGNORE);
      if(i==1 || y>z) z = y;
    }
    *px += z;
    for(int i=1; i<size; ++i)
      MPI_Send(px, 1, MPI_FLOAT, (rank+i)%size, 100, comm);
  }
}
```

(a) [3 p] What computation is performed by `MPI_Function`?

(b) [4 p] What is the complexity of `MPI_Function`? How could it be improved? (to this end, use the model described during the course)

Solution:

(a) Assuming `root<size` (otherwise the function returns and nothing is done), the function adds `*px` of the root process to the maximum of the `*px` values of the processes that are not `root`. Then the result is sent to all processes.

(b) The function performs twice a communication among the root and the each other `size`-1 processes belonging to the communicator `comm`, hence the complexity is linear in the number of processes. The complexity could be improved by using a tree-based schema of communications. In this way, the complexity becomes logarithmic in the number of processes belonging to the communicator `comm`.

8. (a) [13 p] *Median filtering* is a digital filtering technique to reduce noise in a signal or picture. In the one-dimensional case (for instance a sampled audio signal), each element y_i in the filtered signal y is computed as the median of the elements in the input signal $x = \{x_0, x_i, \ldots, x_{n-1}\}$ that are contained in a time window centered around its i'th element x_i. For a time window of size three, each y_i is computed as follows:

$$\begin{aligned} y_0 &= x_0 \\ y_{n-1} &= x_{n-1} \\ y_i &= median(x_{i-1}, x_i, x_{i+1}), \quad 0 < i < n-1 \end{aligned}$$

Define a data parallel algorithm that performs 1-D median filtering with a time window of size three! Use the notation for data parallel algorithms that we have used in the lectures on data parallelism and parallel algorithms (not OpenMP or such). You can not assume that a function to compute the median is given.

What is the parallel time complexity for your algorithm? To get full credits, your solution must have a parallel time complexity that is significantly lower than the sequential complexity for the straightforward sequential solution.

Hint: it is not entirely straightforward to define a function to compute the median. It might be better to define and use a predicate $between(x, y, z)$ that is true if x is in-between y and z such that it holds the median value, and false otherwise.

Solution: We define a function `median` that takes a data parallel array of size `n` (indexed from `0`), and returns a data parallel array that contains the median-filtered signal of the input array. The function uses an algorithm that works by cases, depending on which of the three elements in the time window is the median value. First we define the `between` predicate:

```
between(x,y,z) = (y <= x and x <= z) or (z <= x and x <= y)
```

Then pseudocode for the `median` function:

```
median(x) =
forall i where 0 < i < n - 1 and between(x[i-1],x[i],x[i+1])
  in parallel do y[i] = x[i-1]
forall i where 0 < i < n - 1 and between(x[i],x[i-1],x[i+1])
  in parallel do y[i] = x[i]
forall i where 0 < i < n - 1 and between(x[i+1],x[i-1],x[i])
  in parallel do y[i] = x[i+1]
return y
```

The parallel time complexity for the masked elementwise assignments are all $O(1)$. Thus, the complexity for the whole algorithm is $O(1)$ as well. A sequential algorithm must have at least the time complexity $O(n)$ since it will have to compute n different elements in the filtered signal. Thus, the parallel algorithm has significantly lower time complexity.

Solution: ...

Total: 80