# Exam

## DVA336 Parallel Systems

8:10–12:30

June 4, 2018

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.
8. Inefficient solutions may lead to a lower grade.

## 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. (a) [8 p] Compare *multithreading* and *distributed computing* (i.e. a set of processes cooperating on a network of computers) by stating pros and cons of the two approaches.

**Solution:** The advantage of using *multithreading* is that threads share the same address space so that sharing information and synchronizations are faster than among different processes. On the other hand, *multithreading* is limited to a processor while *distributed computing* enhances scalability.

2. (a) [2 p] what is the parallel time complexity of Radix Sort?

   (b) [7 p] Sort the following sequence in ascending order using Radix Sort:

$$6, 2, 1, 6, 7, 0$$

Show the different steps in your computation.

**Solution:** The time complexity of Radix Sort is $O(\log m \cdot \log n)$, whewre $m$ is the size of the largest key and $n$ is the number of items to be sorted.

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):

$$\begin{array}{cccccc} \mathbf{6} & \mathbf{2} & \mathbf{1} & \mathbf{6} & \mathbf{7} & \mathbf{0} \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{array} \rightarrow \begin{array}{cccccc} \mathbf{6} & \mathbf{2} & \mathbf{6} & \mathbf{0} & \mathbf{1} & \mathbf{7} \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{array} \rightarrow$$

$$\begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{6} & \mathbf{2} & \mathbf{6} & \mathbf{7} \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array} \rightarrow \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{6} & \mathbf{6} & \mathbf{7} \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

3. (a) [8 p] Look at the code below then provide an equivalent SIMD (AVX/SSE) version of it (you can assume `n` is multiple of 8).

```
struct point { float x, y; };

point * alloc(int n) { return (point*) malloc(sizeof(point)*n); };

void init(point * ps, int n) {
  for(int i=0; i<n; ++i)
    ps[i].x = rand()/(RAND_MAX+1.0f),
    ps[i].y = rand()/(RAND_MAX+1.0f);
};
```

```
float sq(float x) { return x*x; };

float * dist(point p, point * ps, int n) {
  float * dists = (float*) malloc(sizeof(float)*n);
  for(int i=0; i<n; ++i)
    dists[i] = sqrtf(sq(ps[i].x-p.x)+sq(ps[i].y-p.y));
  return dists;
};

int main (int argc, char * argv[]) {
  point p;
  p.x = atof(argv[2]);
  p.y = atof(argv[3]);
  int n = atoi(argv[1]);
  point * ps = alloc(n);
  init(ps, n);
  float * dists = dist(p, ps, n);
  for(int i=0; i<n; ++i) printf(" %.3f", dists[i]);
  return 0;
}
```

(b) [2 p] What is the expected speedup for the new version of the `dist` function?

**Solution:** The following code uses of the AVX instruction set. To this end we have to: *i*) reorganize the data structure for fetching 8 values at once; *ii*) allocate aligned memory; *iii*) rewrite the function `dist`. The result is shown below:

```
struct avxpoints {
  float * x;
  float * y;
};

avxpoints avxalloc(int n) {
  avxpoints res;
  res.x = (float*)_mm_malloc(sizeof(float)*n,32);
  res.y = (float*)_mm_malloc(sizeof(float)*n,32);
  return res;
};

void avxinit(avxpoints & ps, int n) {
  for(int i=0; i<n; ++i)
    ps.x[i] = rand()/(RAND_MAX+1.0f),
    ps.y[i] = rand()/(RAND_MAX+1.0f);
};

float * avxdist(point p, avxpoints & ps, int n) {
  float * dists = (float*) _mm_malloc(sizeof(float)*n,32);
  __m256 px = _mm256_set1_ps(p.x);
  __m256 py = _mm256_set1_ps(p.y);
  __m256 x, y, dx, dy, d;
  for(int i=0; i<n; i+=8) {
    x = _mm256_load_ps(&ps.x[i]);
    y = _mm256_load_ps(&ps.y[i]);
    dx = _mm256_sub_ps(x, px);
    dx = _mm256_mul_ps(dx, dx);
    dy = _mm256_sub_ps(y, py);
    dy = _mm256_mul_ps(dy, dy);
    d = _mm256_sqrt_ps(_mm256_add_ps(dx, dy));
    _mm256_store_ps(&dists[i], d);
  }
  return dists;
};

int main (int argc, char * argv[]) {
  point p;
  p.x = atof(argv[2]);
  p.y = atof(argv[3]);
  int n = atoi(argv[1]);
  avxpoints ps = avxalloc(n);
  avxinit(ps, n);
  float * dists = avxdist(p, ps, n);
  for(int i=0; i<n; ++i) printf(" %.3f", dists[i]);
  return 0;
}
```

Speedup should be $\sim 8\times$ since AVX instructions can handle 8 values at once.

4. (a) [5 p] Explain and give the formula (when needed) of the following terms: *Execution Time*, *Speedup*, *Efficiency*, *Work*, and *Cost*.

   (b) [4 p] State when a parallel algorithm is *Work Optimal* and *Cost Optimal*.

   (c) [5 p] Consider a sequential algorithm which scans an array of $n$ integers and finds the maximum value. Describe (or give the pseudo-code) for a possible parallelization of such an algorithm. Then, assume to execute the parallel version on a machine with $p$ processors where $p = n$: what is *Execution Time*, *Work*, and *Cost* of the parallel algorithm? Is it *Work Optimal*? Is it *Cost Optimal*?

**Solution:**

- Execution Time ($T$) is the time spent to complete the computation.
- Speedup ($S = T_{seq}/T_p$) is the factor by how much faster we can solve a problem with $p$ processors instead of one.
- Efficiency ($E = S/p$) measures the fraction of time where the $p$ processors is usefully employed.
- Work ($W$) is the total number of elementary operations performed.
- A parallel algorithm is called Work Optimal if: $W_p = O(T_{seq})$.
- Cost ($C = p{\cdot}T_p$) reflects the sum of time spent by the $p$ processors.

- A parallel algorithm is called Cost Optimal if: $C_p = O(T_{seq})$.

A possible parallelization is to compute a tree-based reduction. The computation looks like traversing a tree with $n$ leaves from the bottom to the root. Nodes at the same level are computed in parallel:

```
**Data:** An array $x[\,]$ of $n$ values (assume $n$ is a power of 2).
for $d = 0$ to $\log_2 n - 1$ do
    for $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
        $x[k + 2^{d+1} - 1] = max(x[k + 2^d - 1], x[k + 2^{d+1} - 1])$;
    end
end
```

As a consequence $T_p = O(\log n)$, $W = O(n)$, $C = p{\cdot}T_p = O(n{\cdot}\log n)$. Since $T_{seq} = O(n)$, the parallel algorithm above is Work Optimal but not Cost Optimal.

5. (a) [6 p] Look at the program below and report any possible problem that might affect its execution and, if any, fix it.

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

pthread_mutex_t read_mutex;
pthread_mutex_t write_mutex;

void * twrite() {
  pthread_mutex_lock(&write_mutex);
  pthread_mutex_lock(&read_mutex);
  FILE * f = fopen("/tmp/msg.txt","w");
  fprintf(f, "something very important");
  fclose(f);
  pthread_mutex_unlock(&read_mutex);
  pthread_mutex_unlock(&write_mutex);
}

void * tread() {
  char str[40];
  pthread_mutex_lock(&read_mutex);
  pthread_mutex_lock(&write_mutex);
  FILE * f = fopen("/tmp/msg.txt","r");
  fscanf(f,"%s",str);
  //do something with str
  fclose(f);
  pthread_mutex_unlock(&write_mutex);
  pthread_mutex_unlock(&read_mutex);
}

int main() {
  pthread_t threads[2];
  pthread_create(&threads[0],NULL,&twrite,NULL);
  pthread_create(&threads[1],NULL,&tread,NULL);
  pthread_join(threads[0],NULL);
```

```
    pthread_join(threads[1],NULL);
    return 0;
  }
```

(b) [6 p] Define what a race condition is and provide a program as an example.

**Solution:** The program might not terminate since a deadlock can occur: the two threads could enter in a reciprocal-waiting state. In particular it happens if `twrite()` locks `write_mutex` and `tread()` locks `read_mutex` before that `twrite()` is able to lock it. To fix it is enough to switch the order of the `pthread_mutex_lock()` calls in `tread()`. A race condition arises when at least two threads simultaneously try to modify a shared data. The result is then affected by the temporal sequence with which the read/write operations are performed by the threads. An example is in the code labeled 'test08' provided in class.

6. (a) [6 p] Let be `f` a `C` function that gets in input an integer and returns a boolean (see the example in the code below). Implement a MPI collective function having the following signature:

   `MPI_IfBarrier(maskingfunction f, MPI_Comm comm)`

   that acts as a normal "barrier" but only for those processes in `comm` for which `f`, applied to their rank, returns true. You are not allowed to use other predefined MPI collective functions.

   (b) [4 p] Calculate the complexity of your solution. To this end assume that processes run on different nodes and nodes are equidistant and single-ported, i.e., each process can communicate with one different process at the same time.

For example, in the following `C` code, only the processes with even rank are blocked until all of them have reached the routine:

```
typedef bool (*maskingfunction)(int i);

bool iseven(int n) { return n%2==0; }

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  MPI_IfBarrier(iseven, MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

**Solution:** A possible solution is to centralize the computation in the first eligible root process:

```
void MPI_IfBarrier(maskfunction f, MPI_Comm comm) {
  int rank, size, root = 0;
  MPI_Comm_rank(comm, &rank);
  //early exit for the processes not in the barrier
  if(f(rank)==false)
    return;
  MPI_Comm_size(comm, &size);
  //find a process eligible to be root
  while(root<size && f(root)==false)
    ++root;
  //if root has been found (ow the maskarray contains only false)
  if(root<size) {
    int msg = 1;
    if(rank==root) {
      // root collects a token from all processes belonging to the barrier...
      for(int i=0; i<size; ++i)
        if(i!=rank && f(i)==true)
          MPI_Recv(&msg, 1, MPI_INT, i, 100, comm, MPI_STATUS_IGNORE);
      // ...then sends a token to them
      for(int i=0; i<size; ++i)
        if(i!=rank && f(i)==true)
          MPI_Ssend(&msg, 1, MPI_INT, i, 100, comm);
    } else {
      //each non-root process sends a token to the root then waits for a reply
      MPI_Ssend(&msg, 1, MPI_INT, root, 100, comm);
      MPI_Recv(&msg, 1, MPI_INT, root, 100, comm, MPI_STATUS_IGNORE);
    }
  }
}
```

Let $n$ be the number of processes involved in the barrier, the complexity of this version is given by the number of steps performed by the root for gathering the all tokens and send them back, namely $2(n-1)$. Hence the complexity is linear in the number of processes (even if a logarithmic solution can be implemented).

7. (a) [6 p] Give the definition of the *Little's Law* and how it is related to GPU programming.

**Solution:** Little's Law says that, under steady conditions, the average number of items in a queuing system equals the average rate at which items arrive multiplied by the average time that an item spends in the system. Let $L$ be the average number of items in the queuing system, $W$ be the average waiting time in the system for an item, and $\lambda$ be the average number of items arriving per unit time, the law is $L = \lambda W$.

It is used to improve throughput of our computation on GPUs by increasing istruction level of parallelism in the pipelined scalar processors (SPs).

8. (a) [11 p] Define a data parallel algorithm that implements a function `remove(s,c)`, which takes a string `s` and a character `c` and returns a string where all occur-

rences of `c` in `s` are removed while keeping the order of the other characters. For instance, it should hold that

`remove("AllanTarkakan",'a') = "AllnTrkkn"`

You can assume that the strings are stored in data parallel arrays, indexed from `0` and up. If the array is longer than the string stored in it, then the remaining part of the array is padded with a special character "`empty`". Use the notation for data parallel algorithms that we have used in the lectures on data parallelism and parallel algorithms (not OpenMP or such).

What is the parallel time complexity of 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.

**Solution:** We use a strategy similar to the data parallel imnplementation of Radix Sort, where we use the `enumerate` primitive to compute destination addresses where to send the characters in `s` that are to remain in the string. We compute `enumerate` under a mask that tests on inequality with `c`. Data parallel pseudocode:

```
remove(s,c) =
forall i in parallel do
  result[i] = empty
forall i where s[i] <> c in parallel do
  result[enumerate[i]] = s[i]
return(result)
```

What is the parallel complexity? The data parallel instructions are a broadcast ($O(1)$), `enumerate` which is a scan, and thus $O(\log n)$, and a parallel send ($O(1)$). The parallel complexity is thus $O(\log n)$ where $n$ is the number of characters in the string. A sequential algorithm must at least iterate through all elements in the string once, giving it a time complexity of at least $O(n)$. Thus, the parallel algorithm has significantly lower time complexity.

Total: 80