# Exam

## DVA336 Parallel Systems

8:10–12:30

June 5, 2017

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, 073-6620897.
For questions on problems 2 and 8, please contact Björn Lisper, 073-9607199.

Good luck!

1. (a) [4 p] Define and provide (at least) one example of the following architectures: SIMD, MIMD, SIMT.

   (b) [4 p] Explain the difference in how SIMD and SIMT handle an instruction flow.

**Solution:** In Single Instruction, Multiple Data (SIMD) architectures, a set of processing units executes the same instruction flow issued by a single control unit. In this way it is possible to create a scalable data-parallel architecture, as, for example, vector processors. The SIMD architectures are present in nowadays computer, like the SSE, SSE2, SSE3 and AVX instruction sets.

Despite to SIMD, the Multiple Instruction, Multiple Data (MIMD) architectures have a set of control units driving a set of processing units. An example are multicores. Here, a processor executes autonomously its own stream of instructions on the different set of data by possibly sharing information with other processors by means of a shared memory layer. There are also distributed systems, in which different machines cooperate by, for example, message passing protocols.

The term Single Instruction, Multiple Thread (SIMT) SIMT was coined by Nvidia in the beginning of 2000 for their G80 architecture. This architecture can be seen as an evolution of SIMD since they have some commonalities. Like SIMD, a SIMT processor has one instruction unit that issues the same instruction to multiple execution units (aka scalar processors). Here, however, it is possible specify the branching behavior of a single thread associated to a scalar processor. In practice, different threads can follow different instruction flows from the same program, but when it happens the execution is partially serialized. Moreover, each scalar processor has its own set of registers and local memory which allows to load/store uncoalesced addresses.

2. [7 p] Sort the following sequence in scending order using Radix sort:

$$7, 6, 5, 4, 3, 2, 1, 0$$

Show the different steps in your computation.

**Solution:** 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):

| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

$\rightarrow$

| **6** | **4** | **2** | **0** | **7** | **5** | **3** | **1** |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

$\rightarrow$

| **4** | **0** | **5** | **1** | **6** | **2** | **7** | **3** |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

$\rightarrow$

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

3. Consider the following C++ function:

```
float getmin(float const * array, const int n) {
   float min = array[0];
   for(int i=1; i<n; ++i)
      min = array[i]<min ? array[i] : min;
   return min;
}
```

   (a) [6 p] Write the SIMD version of the `getmin` function (assume that $n$ is positive and multiple of 32).

   (b) [3 p] Discuss and motivate the performance that can be expected for each one of the version you have created compared to the original one.

**Solution:**

```
float avxgetmin(float const * array, const int n) {
   __m256 min = _mm256_load_ps(array);
   //scan
   for(int i=8; i<n-7; i+=8)
      min = _mm256_min_ps(min, _mm256_load_ps(array+i));
   //reduction
   min = _mm256_min_ps(min, _mm256_permute2f128_ps(min, min, 0x81));
   min = _mm256_min_ps(min, _mm256_shuffle_ps(min, min, 0x0e));
   min = _mm256_min_ps(min, _mm256_shuffle_ps(min, min, 0x01));
   //return result
   return *((float*)&min);
}
```

Furthermore, since the AVX functions handle 8 float values at-a-time and $n$ is multiple of 32, there is the chance to unroll the loop and compute 32 values in each iteration. AVX instructions have a theoretical speedup equal to 8× since they compute 8 floats simultaneously. Unlikely, other factors, for example the operands fetching overhead, do not allow to reach the speedup peak in practice. As a consequence, we can only estimate that the theoretical speedup of `avxgetmin` is 8× as well, disregarding negligible delays like the final reduction or eventual tails when $n$ is not multiple of 8, for example.

4. (a) [4 p] Give the definition of the *Amdahl's Law* for parallel programs and explain the meaning of each part of the formula.

   (b) [6 p] Consider the following sequential program:

```
int main() {
   int data[50];
   populate(data);
   for(int i=0; i<50; ++i)
      update(data[i]);
   finalize(data);
   return 0;
}
```

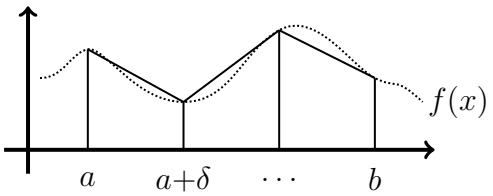
Let the time required to complete the sequential functions `populate`, `update`, and `finalize` be 15, 9, and 35 milliseconds, respectively. We provide a multi-threaded version of the program which fairly splits the computation of the central for-loop among the available processors. Calculate the expected speedup $S(p)$ of the parallel program according to the *Amdahl's Law* for a number of processors $p = 16, 32, 64$.

**Solution:** The parallel program requires $T(p) = 15 + (9 * 50)/p + 35$ ms (because only the central part can be parallelized), while its sequential version requires $T = T(1) = 15 + (9 * 50) + 35 = 500$ ms. The Amdahl's Law defines the speedup as $S(p) = \frac{T}{T(p)} = \frac{1}{(1-F)\frac{F}{p}}$, where $F$ is the fraction of the computation that can benefit from parallelization and $p$ is the degree of parallelism, e.g., number of processors. In this case $F = (9 * 50)/500 = 90\%$ and $p$ varies in $\{8, 16, 32, 64\}$.

| $F$ | $p$ | $S(p)$ |
|---|---|---|
| 0.1 | 16 | $500/\sim 78.125 = 6.4$ |
| 0.1 | 32 | $500/\sim 64.062 = 7.8$ |
| 0.1 | 64 | $500/\sim 59.000 = 8.5^*$ |

* since the `data` array has only 50 items, to use more than 50 processors does not improve $T(p)$ so that $T(64) = T(50)$, thus $S(64) = S(50) = 8.5\times$ which is the maximum speedup for this program. If `data` had an arbitrary size, the maximum speedup could be $10\times$.

5. The *definite integral* can be used to calculate the area of the region in the Cartesian plane between a function $f(\,)$ and an interval of the abscissa. When it is too hard to calculate, we can approximate the answer by means, for example, of the Trapezoidal Rule (TR). Given a continuous function $f : \mathbb{R} \mapsto \mathbb{R}^+$, let be $a$ and $b$ two real values with $a<b$, we can approximate $\int_a^b f(x)\,dx$ with the sum of the area of trapezoids as exemplified in the figure:




In particular, we divide the interval $[a, b]$ in $n$ segments so that $\delta=(b+a)/n$ and sum the area of $n$ trapezoids:

$$\frac{\delta}{2} \sum_{i=0}^{n-1} f(a + \delta i) + f(a + \delta(i+1))$$

(a) [6 p] Write the OpenMP version of the missing `tr` function which implements TR in the following program.

```
#include <iostream>
#include <math.h>
#include <omp.h>
double gauss(double x) {
   const double sigma = 1.0, mu = 3.0;
   double a = 1.0/(sigma*sqrt(2.0*M_PI));
   double n = -1.0*(x-mu)*(x-mu);
   double d = 2.0*sigma*sigma;
   return a*exp(n/d);
}
double tr(double(*f)(double), double a, double b, int n) {
   ... // TR implementation
}
int main() {
   double a = 0.0;
   double b = 6.0;
   int n = 1000000;
   std::cout << tr(gauss,a,b,n) << std::endl;
   return 0;
}
```

(b) [4 p] What speedup could be expected when your program runs on a quad-core processor? Motivate the answer.

**Solution:**

```
double tr(double(*f)(double), double a, double b, int n) {
   if(a>=b or n<1) exit(1);
   double delta=(a+b)/n, sum=0.0;
   #pragma omp parallel for reduction(+:sum)
   for(int i=0; i<n; ++i)
      sum += f(a+delta*i)+f(a+delta*(i+1));
   return sum*delta*0.5;
}
```

Assuming that the overhead for managing the thread is negligible, the expected speedup should be close to the ideal one, i.e., 4.

6. Open MPI provides the `MPI_Barrier(comm)` routine which blocks the caller until all processes in the `comm` communicator have called it; that is, the call returns at any process only after all members of the communicator have entered the call.

(a) [6 p] Provide a possible implementation of the `MPI_Barrier` by means of the MPI point-to-point communication functions (`MPI_Send`, `MPI_Recv`, etc). Below there is a sketch of code which provides some useful MPI functions, the signature of the `Barrier` routine to implement, and an example of a point-to-point communication. Explain the expected behavior of the communication functions used.

```
#include <cstdlib>
#include <iostream>
#include <mpi.h>

void Barrier(MPI_Comm comm) {...}

int main (int argc, char * argv[]) {
   MPI_Init(&argc,&argv);
   int n, p, m = 3;
   MPI_Comm_size(MPI_COMM_WORLD,&n);
   if(n>1) {
      MPI_Status s;
      MPI_Comm_rank(MPI_COMM_WORLD,&p);
      if(p==0)
         MPI_Ssend(&m, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
      if(p==1)
         MPI_Recv(&m, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &s);
   }
   ... // do something before barrier
   Barrier(MPI_COMM_WORLD);
   ... // do something after barrier
   MPI_Finalize();
   return 0;
}
```

(b) [5 p] Assume to run the parallel program shown in the previous point (a) on $n$ machines with one process instantiated on each machine. Assume also that each machine has only one outgoing/incoming network link, so that a message cannot be sent/received by a process if the link is handling another communication for the same process. Let the latency of a point-to-point communication be $\ell$, estimate the performance of your `Barrier` routine in function of $\ell$.

**Solution:** We can implement the `Barrier` routine as a token ring communication among the $n$ processes.

```
void Barrier(MPI_Comm comm) {
   int n, p, token = 3;
   MPI_Comm_size(comm,&n);
   if(n<2) return;
   MPI_Comm_rank(comm,&p);
   MPI_Status status;
   if(p==0) {
      MPI_Ssend(&token, 1, MPI_INT, 1, 100, comm);
      MPI_Recv(&token, 1, MPI_INT, n-1, 100, comm, &status);
   } else {
      MPI_Recv(&token, 1, MPI_INT, p-1, 100, comm, &status);
      MPI_Ssend(&token, 1, MPI_INT, (p+1)%n, 100, comm);
   }
}
```

The routine is expected to require $n \cdot \ell$ since the ring consists of $n$ consecutive point-to-point synchronous communications. A better solution is a tree-based pattern of communication of which expected performance are $O(\log n) \cdot \ell$.

7. One method to estimate the value of $\pi$ is by using a Monte Carlo (MC) method. We generate $n$ uniformly distributed random points in any position within the square between $(0,0)$ and $(1,1)$. Let $h$ denote the number of points of which distance from the origin $(0,0)$ is not greater than 1. At the end of the procedure, we should get that $h/n \approx \pi/4$. The following CUDA kernel `doit` implements MC while the device function `rand01` returns an different sequence of values in $[0,1]$ for each thread.

```
__global__ void doit(float * pisum, int const * seeds, int n) {
   int u = seeds[blockIdx.x*blockDim.x+threadIdx.x];
   int h = 0;
   for(int i=0; i<n; ++i) {
      float x = rand01(u);
      float y = rand01(u);
      h += int(x*x+y*y<=1.0f);
   }
   float threadpi = 4.0f*h/n;
   atomicAdd(pisum, threadpi);
}

#define T 512
#define B 100

int main() {
   ... // init
   doit <<<B,T>>> (pisum, seeds, 10000);
   ... // finalize
   return 0;
}
```

Consider the program above and answer to the following questions by motivating each answer:

(a) [2 p] In which memory is allocated the value pointed by `pisum`?

(b) [2 p] What is the minimum size (in bytes) of the `seeds` array?

(c) [6 p] Provide an improved version of the kernel which makes a more efficient use of the function `atomicAdd` and explain what is the improvement.

**Solution:** `pisum` points to a float in the device memory since the threads accumulate their results in that variable and only the device memory is shared among all CUDA threads. The size of `seeds` is at least `T*B*sizeof(int)` bytes, since each CUDA thread pick a distinct value from that array as shown in line of the code where `u` is initialized.

```
__global__ void doitV2(float * pisum, int const * seeds, int n) {
   __shared__ float blockpisum;
   if(threadIdx.x==0)
      blockpisum = 0;
   __syncthreads();
   int u = seeds[blockIdx.x*blockDim.x+threadIdx.x];
   int h = 0;
   for(int i=0; i<n; ++i) {
      float x = rand01(u);
      float y = rand01(u);
```

```
      h += int(x*x+y*y<=1.0f);
   }
   float threadpi = 4.0f*h/n;
   atomicAdd(&blockpisum, threadpi);
   __syncthreads();
   if(threadIdx.x==0)
      atomicAdd(pisum, blockpisum);
}
```

In this improved version, CUDA threads access the variable `pisum` in the device memory only `B` times, instead of `T*B`; that is, threads of each block accumulate their result in the variable `blockpipisum` in the shared memory (which has faster access time) and, at the end of the block computation, only thread #0 updates the value on the device memory.

8. [15 p] Let the string `s` have `n` characters, and let the string `pat` have `m` characters. Define a data parallel algorithm that counts the number of positions where `pat` equals the substring of `s` starting in that position. For instance, if `s = "cacababa"` and `pat = "aba"` then your algorithm should return the answer 2 since `"aba"` equals substrings of `s` starting at two positions.

   Use the pseudonotation for data parallel algorithms that we have used in the lectures to describe your algorithm on high level. You may use data parallel arrays to store the strings. Beware not to have array accesses that are out of bounds. You may want to use a sequential for-loop inside a parallel "in parallel do" statement: the semantics is then that each statement in the for loop is executed synchronously, in parallel, for each index value of the enclosing parallel statement. You may also want to use matrices: in that case you may use the notation `A[i,*]` to access row `i` of the matrix `A`, and `A[*,j]` to access column `j`.

   Also give the parallel time complexity for your algorithm: for full credits, this complexity must be significantly lower than the time complexity for a decent sequential algorithm. (You may assume that $\texttt{n} \gg \texttt{m}$.)

**Solution:** Our solution stores `s` and `pat` in data parallel arrays (indexed from 0). We also have a temporary array `match` with $\texttt{n} - \texttt{m} + 1$ elements. Our solution uses an inner,sequential loop to go through the elements in `pat` sequentially aand in each step in parallel check, for all `i`, whether the corresponding elements of `s` and `pat` are equal or not. (We could have parallelised also this loop, using a matrix of booleans, but the solution with a sequential inner loop also has good parallel complesity when $\texttt{n} \gg \texttt{m}$.) Here is our pseudocode for the algorithm:

```
forall i where 0 <= i <= n-m in parallel do
  match[i] = 1
  for j = 0 to m-1 do
    if s[i+j] <> pat[j] then match[i] = 0
```

```
matches = reduce(match,+)
return matches
```

Note that we constrain the range of `i` in the parallel statement in order to avoid accesses to `s` outside its range. (There can never be any match anyway that starts in a position greater than `n` − `m`.)

What about the parallel time complexity? The sequential inner loop contributes with $O(m)$ sequential steps, and the final reduction is $O(\log n)$. The other statements are $O(1)$. The resulting parallel complexity is thus $O(m + \log n)$. This is significantly better than for a straightforward sequential algorithm, which can be exepcted to have the time complexity $O(m \cdot n)$.

Total: 80