Exam

DVA336 Parallel Systems

8:10–12:30

August 17, 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 *data parallelism* and *task parallelism*.

   (b) [3 p] Provide (at least) an example of problems suitable for such kinds of parallelism.

   (c) [3 p] Provide (at least) an example of architectures implementing such kinds of parallelism.

Solution: Data parallelism refers to situations where the same operation can be performed at the same time on the elements of the input set, e.g., array. Task parallelism consists in divide the computation into a set of processes/tasks/threads for its concurrent execution.

Example of a problem suitable for data parallelism is compute the upper-case of a string. While computing different statistical indicator (e.g., average, median, min, max, etc.) o the same dataset can be considered a suitable case of task parallelism.

Multicores can be considered an example of task parallelism since each core can execute (possibly) different tasks on different data. The SIMD instruction set and vector processors, instead, are two examples of data parallel architectures.

2. [7 p] In the course we bring up six different data parallel primitives, which recur in different data parallel settings. Name at least four of them! For each one explain briefly what it does, and give its parallel time complexity assuming the simple machine model for data parallelism that we have covered in the course.

Solution: We list here all six primitives (four of these are enough to get full credits):

- Elementwise application of scalar operation: to apply an operator or function to all elements in a data structure at once. $O(1)$ parallel time complexity.
- Get communication (parallel read): each processor simultaneously fetches an element from some given position in a data structure. $O(1)$ parallel time complexity.
- Send communication (parallel write): each processor simultaneously sends its element in some data structure to some given position in another data structure. $O(1)$ parallel time complexity.
- Replication: to duplicate a single datum to many processors. $O(1)$ parallel time complexity.
- Masking (selection): to select part of a data structure for some data parallel operation using a boolean "mask". $O(1)$ parallel time complexity.

- Reduce and scan: to "sum" all elements in a data structure with respect to some binary, associative operation (or, for scan, to form all "partial sume"). $O(\log n)$ parallel time complexity, where n is the number of elements in the data structure.

3. Consider the following C++ function and assume that `n` is multiple of 40:

```
float getavg(float const * array, const int n) {
   float sum = array[0];
   for(int i=1; i<n; ++i)
      sum += array[i];
   return sum/n;
}
```

(a) [5 p] Write the SIMD version of `getavg(...)`.

(b) [3 p] Write the SIMD version of `getavg(...)` with loop-unrolling.

(c) [2 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: part (a):

```
float getavg(float const * array, const int n) {
   __m256 sum = _mm256_setzero_ps();
   for(int i=0; i<n; i+=8)
      sum = _mm256_add_ps(sum, _mm256_load_ps(array+i));
   sum = _mm256_add_ps(sum, _mm256_permute2f128_ps(sum, sum, 0x81));
   sum = _mm256_add_ps(sum, _mm256_shuffle_ps(sum, sum, 0x0e));
   sum = _mm256_add_ps(sum, _mm256_shuffle_ps(sum, sum, 0x01));
   return (*((float*)&sum))/n;
}
```

part (b):

```
float getavg(float const * array, const int n) {
   __m256 sum = _mm256_setzero_ps();
   for(int i=0; i<n; i+=40) {
      sum = _mm256_add_ps(sum, _mm256_load_ps(array+i));
      sum = _mm256_add_ps(sum, _mm256_load_ps(array+i+8));
      sum = _mm256_add_ps(sum, _mm256_load_ps(array+i+16));
      sum = _mm256_add_ps(sum, _mm256_load_ps(array+i+24));
      sum = _mm256_add_ps(sum, _mm256_load_ps(array+i+32));
  }
   sum = _mm256_add_ps(sum, _mm256_permute2f128_ps(sum, sum, 0x81));
   sum = _mm256_add_ps(sum, _mm256_shuffle_ps(sum, sum, 0x0e));
   sum = _mm256_add_ps(sum, _mm256_shuffle_ps(sum, sum, 0x01));
   return (*((float*)&sum))/n;
}
```

AVX instructions have a theoretical speedup equal to 8× since they compute 8 floats simultaneously. As a consequence, we can only estimate that the theoretical speedup of the two functions is 8×.

4. (a) [4 p] Define the *Amdahl's Law* for parallel programs.

   (b) [6 p] Consider a 'master-slave' algorithm, in which the master processor first distributes work to a set of slaves, then collects and processes the results produced by the slaves. Assume that the master's task takes a factor $s = 0.10$ of the total execution time. Calculate the maximum theoretical speedup that can be achieved by changing the number of slaves.

Solution: The Amdahl's Law defines the speedup as $S(p) = \frac{T}{T(p)} = \frac{1}{s+\frac{1-s}{p}}$, where s is the fraction of the program which is serial and p is the degree of parallelism, e.g., number of processors.

In this case we have that $S(p) = \frac{1}{0.10+\frac{1-0.10}{p}}$. By increasing the number of slaves, p, the denominator tends to $s = 0.10$ since $\frac{1-0.10}{p} \to 0$. As a consequence the maximum theoretical speedup is $\frac{1}{s}$ that is $10\times$.

5. In the following code, the function `seq` counts the number of occurrences of `ch` in the file `filename`.

```
int seq(char const * filename, const char ch) {
  FILE *f = fopen(filename, "r");
  int counter = 0;
  if(f) {
    while(!feof(f)) {
      ssize_t nread;
      size_t buffersize = 0;
      char * buffer = NULL;
      nread = getline(&buffer, &buffersize, f);
      for(ssize_t i=0; i<nread; ++i)
        if(buffer[i]==ch) ++counter;
      free(buffer);
    }
    fclose(f);
  }
  return counter;
}

int main(int argc, char *argv[]) {
  if(argc!=3) exit(1);
  int counter = seq(argv[1], *argv[2]);
  printf("%c occurs %d times in %s\n", *argv[2], counter, argv[1]);
  return 0;
}
```

Note that `getline(char **ptr, ssize_t *n, FILE *stream)` reads an entire line from `stream`, storing the address of the buffer containing the text into `*ptr`. The buffer is null-terminated and includes the newline character, if one was found. If `*ptr` is set to `NULL` and `*n` is set 0 before the call, then `getline()` will allocate a buffer for storing the line. This buffer should be freed by the user program. `getline()` is a multithread-safe function.

(a) [5 p] Write the OpenMP version of `seq`.

(b) [5 p] Write the OpenMP version of `seq` assuming that `getline()` is multithread-unsafe, i.e, not safe to call in the presence of other threads.

Solution: part (b):

```
int par(char const * filename, const char ch) {
  FILE *f = fopen(filename, "r");
  int counter = 0;
  if(f) {
    #pragma omp parallel reduction (+:counter)
    while(!feof(f)) {
      ssize_t nread;
      size_t buffersize = 0;
      char * buffer = nullptr;
      #pragma omp critical
      { nread = getline(&buffer, &buffersize, f); }
      for(ssize_t i=0; i<nread; ++i)
        if(buffer[i]==ch) ++counter;
      free(buffer);
    }
    fclose(f);
  }
  return counter;
}
```

For part (a), remove '`#pragma omp critical`' line.

6. Almost every MPI function can be implemented by means of the basic sending and receiving functions:
`MPI_Send(..., int tag, MPI_Comm comm)`,
`MPI_Recv(..., int tag, MPI_Comm comm, MPI_Status* status)`.

(a) [3 p] Define a communicator is in MPI.

(b) [3 p] Define what a tag is in MPI.

(c) [4 p] Explain the differences between tags and communicators.

Solution: A communicator is an MPI term that means an ordered group of MPI processes. Each process gets a unique integer rank identifier in that communicator in the range $[0, n-1)$ where n is the number of processes.

A tag is simply an arbitrary integer that is used to delineate the matching of point-to-point messages.

Communicators are heavy objects: creating a communicator requires time and internal MPI resources. Tags are light-weight objects theyre just integers, and can be chosen arbitrarily.

7. Let A be an array of n float values with $n > 0$, and let x be a float value.

   (a) [6 p] Define a CUDA kernel for finding the positions of A storing x. Example:
   $$A = \langle 1, 2, 4, 1, 3, 8, 0, 2, 9, 7 \rangle \text{ and } x = 1 \text{ returns } \langle 0, 3 \rangle$$

   (b) [4 p] Show an example of how the kernel is invoked in by a host function (to this end, do the required assumptions to ensure the correct behavior of the program).

Solution:

```
__global__ void foo(int * idxs, int * nidxs, float x, float * A, int n) {
   int i = blockIdx.x*blockDim.x+threadIdx.x;
   if(i<n) {
    float v = A[i];
    if(v==x) {
      int pos = atomicAdd(nidxs, 1);
      idxs[pos] = i;
    }
   }
}

#define T 512

int main() {
   ...
   foo <<<(n+T-1)/T,T>>> (idxs, nidxs, 1.0, A, n);
   ...
   return 0;
}
```

Assume that `nidxs` is initialized to zero and all pointers used by in the kernel invocation are allocated on the device memory. Moreover, the `idxs` capacity is n and n is less than the max no. threads multiplied by the max no. blocks allowed by CUDA.

8. [13 p] When identifying objects in a picture, the first step is often *thresholding*: to set all pixels whose values are below a threshold to white, and the ones above it to black. If the threshold is selected in the right way, this can make the delimitations of the objects in the picture more visible and they can be found by a subsequent edge detection.

   Now assume that a black and white picture is stored in a matrix `b`, where the elements are integers in the range 0 to 255. The elements represent the pixels, where the numbers represent increasingly grey shades (0 = white, 255 = black). Write data parallel code representing a data parallel algorithm that:

   1. computes the mean value of the pixel values in `b`, and
   2. thresholds the picture in `b` using the mean value as threshold.

   Use the notation for data parallel algorithms that we have used in the lectures on data parallelism and parallel algorithms (not Peril-L, not OpenMP, not MPI). Hint:

to express that a data parallel operation is performed over all elements of a matrix, with indices `(i,j)`, you can write "`For all (i,j)...`".

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: The mean value of the pixels is easily computed using reduction to compute the sum of the pixel values, and counting the elements. In two subsequent masked data parallel statements, the thresholding is then done.

The following is code written in our pseudocode notation from the lectures:

```
For all (i,j) in parallel do
   n = count
   m = reduce(+,b)/n
For all (i,j) where b(i,j) < m in parallel do b(i,j) = 0
For all (i,j) where b(i,j) >= m in parallel do b(i,j) = 255
```

We perform four data parallel statements over a matrix with `n` elements. Two of them are reductions (computing `n`, and `m`), where each can be done in parallel in $O(\log \mathtt{n})$ time, and two are elementwise assignments (computing new values of `b` according to the threshold) that can be done in parallel in $O(1)$ time. In total, the parallel time complexity is $O(\log \mathtt{n})$. This is significantly lower than the time complexity of the best sequential algorithm to perform the thresholding, which is $O(\mathtt{n})$.

Total: 80