Exam

DVA336 Parallel Systems

8:10–12:30

January 9, 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 Thomas Larsson, 073-6620847.
For questions on problems 2 and 8, please contact Björn Lisper, 021-151709.

Good luck!

1. (a) [2 p] What is high performance computing (HPC)?
   (b) [2 p] Why is high performance computing needed?
   (c) [2 p] What type of parallelism is usually present when algorithms show good scalability even on systems with thousands of processors? Motivate your answer!
   (d) [2 p] Give an example of an embarrassingly parallel problem. Also, explain why your example is appropriate.

Solution:

(a) HPC is about parallel computer systems and software solutions that give much higher performance compared to what you get from an ordinary PC or laptop. HPC is also a research area that involves parallel algorithms, parallel programming paradigms, languages, and APIs.

(b) HPC makes it possible to solve bigger problems and run larger simulations in many areas in, e.g., science, engineering, and business. Clearly, this enables progress and new knowledge in numerous problem areas.

(c) Data parallelism. For large datasets, the dataset can be partitioned across many processors, and then the same operations are performed on the subsets in parallel, which lead to good scalability in many cases.

(d) For example, converting a large color image to grayscale is an embarrassingly parallel operation. Each pixel can be processed independently by applying a simple formula to find the intensity. Thus, by assigning an equal share of pixels to the processors, we can expect good load balance and scalability.

2. (a) [2 p] What is the parallel time complexity of the `scan` operation "`scan(op,X)`" when implemented according to what we have covered during the lectures (and provided that the binary operator "`op`" takes unit time to compute)?
   (b) [2 p] Why is this parallel algorithm for computing `scan` *not* a good sequential algorithm?
   (c) [3 p] Consider the following code. It provides an alternative implementation of scan:

```
seq_scan(op,X) =
  for i = 1 to Length(X) - 1 do
    X[i] := X[i-1] op X[i]
```

   Explain why (1) this is a good sequential algorithm for computing scan, and (2) why it is *not* a good parallel algorithm for this purpose!

Solution:

(a) $O(\log n)$ time on $O(n)$ processors, where n is the number of elements in the array.

(b) It uses $O(n \log n)$ operations, which is its sequential time complexity. There are algorithms for `scan` that have a lower such complexity.

(c) The algorithm uses $O(n)$ operations in total, so it's a good sequential algorithm. However each loop iteration depends on results from the previous one, so the algorithm is completely sequential and cannot be parallelised as it stands. Also its parallel complexity is $O(n)$.

3. (a) [4 p] Describe Flynn's taxonomy of computer architectures.
   (b) [2 p] How do current multi-core CPUs relate to Flynn's taxonomy?
   (c) [2 p] These CPUs are dependent on a complex memory system. Explain why this is needed and describe the most salient parts of these memory systems.

Solution:

(a) In Flynn's taxonomy, the architectures are classified by the number of instruction and data streams that can be processed simultaneously. The different types are:

- Single instruction stream, single data stream (SISD): A sequential architecture that corresponds to the von Neumann model.
- Single instruction stream, multiple data streams (SIMD): Used in vector architectures, multimedia/graphics extensions, and GPUs.
- Multiple instruction streams, single data stream (MISD): This is an uncommon architecture.
- Multiple instruction streams, multiple data streams (MIMD): An architecture that has a number of processors that run independently.

(b) Clearly, current multi-core CPUs are of type MIMD, since each core can run different code segments (instructions) and process unique data elements. Multi-threaded programming is the usual way to accomplish this kind of parallel execution. Furthermore, the cores have vector units supporting SIMD instructions, which can be used to further enhance the parallel execution.

(c) The reason is the so called memory wall, i.e., the increasing gap between the rate at which the processor can perform computations and the rate at which data can be read from main memory (RAM). To hide the latency of too slow memory fetches, multilevel cache systems with prefetching techniques have been developed. Today three-level caches are common, where e.g. the first level consists of very fast local memories, private to the cores.

4. Consider a perfectly parallelized computation that runs on a computer with 1024

cores. However, 2% of the execution time is spent in an inherently sequential part of the computation.

(a) [4 p] Use Amdahl's law to predict the parallel speedup we can get on the mentioned computer, when the number of available cores is set to 10, 100, and 1000, respectively.

(b) [4 p] Suppose that the computation involves a broadcast operation that adds a parallel overhead of $0.005p$, where p is the number of cores used. Show how this would affect the predicted speedups. Also, determine the exact number of cores that gives the maximum speedup in this particular case.

(c) [2 p] It seems that Amdahl's law suggests that the benefit of running a computation on, say, 1000 cores, or more, is very limited. Why are such parallel systems built anyway?

Solution:

(a) The parallel speedup $S(p)$ for p cores according to Amdahl's law is given by

$$S(p) = \frac{1}{(1-f) + f/p},$$

where f denotes the parallel portion of the runtime. Thus, with $f = 0.98$, we get

$$S(10) = \frac{1}{(1-0.98) + 0.98/10} = \frac{1}{0.118} \approx 8.5,$$
$$S(100) = \frac{1}{(1-0.98) + 0.98/100} = \frac{1}{0.0298} \approx 33.6,$$
$$S(1000) = \frac{1}{(1-0.98) + 0.98/1000} = \frac{1}{0.02098} \approx 47.7.$$

(b) We can add the overhead in the denominator of the speedup formula:

$$S(p) = \frac{1}{(1-0.98) + 0.98/p + 0.005p}.$$

The reduced speedups are then given by

$$S(10) = \frac{1}{(1-0.98) + 0.98/10 + 0.005 * 10} = \frac{1}{0.168} \approx 6.0,$$
$$S(100) = \frac{1}{(1-0.98) + 0.98/100 + 0.005 * 100} = \frac{1}{0.5298} \approx 1.9,$$
$$S(1000) = \frac{1}{(1-0.98) + 0.98/1000 + 0.005 * 1000} = \frac{1}{5.02098} \approx 0.2.$$

The maximum speedup occurs when the denominator is minimized. The first (partial) derivative of the denominator is

$$\frac{\partial}{\partial p}(1-0.98) + 0.98/p + 0.005p = 0.005 - \frac{0.98}{p^2},$$

and it is minimized when the derivative is zero. Hence,

$$0.005 - \frac{0.98}{p^2} = 0 \quad \Longrightarrow \quad p = \sqrt{146} = 14,$$

and the maximum speedup is

$$S(14) = \frac{1}{(1 - 0.98) + 0.98/14 + 0.005 * 14} = \frac{1}{0.16} = 6.25.$$

(c) Amdahl's law is only applicable in certain circumstances. In particular, it treats the problem size as a constant. In practice, however, increasing the problem size often leads to a smaller serial fraction of the program, which improves scalability.

5. Starting from any positive integer n, a sequence of numbers can be generated by following two simple rules. If n is even, let $n = n/2$ or if n is odd, let $n = 3n + 1$. For example, starting from $n = 19$ gives the sequence

$$19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$$

In fact, it appears that the sequence always reaches down to 1, no matter what n we start from, which is something that was conjectured by Collatz already in the year 1937, and the conjecture still remains unresolved. It is easy, however, to verify that Collatz conjecture is true for, say, all positive integers $m \leq 10^9$ by brute force testing in a computer program using 64-bit unsigned integers. Consider the following program written in C:

```
void collatzSteps(unsigned long long n, unsigned long long & steps) {
  steps = 0;
  while (n > 1) {
    if (n % 2 == 0) n = n / 2;
    else n = 3 * n + 1;
    steps++;
  }
}

bool verifyCollatzConjecture(unsigned long long max) {
  unsigned long long steps = 0, totSteps = 0;

  for (unsigned long long i = 1; i <= max; i++) {
    collatzSteps(i, steps);
    totSteps += steps;
  }
  printf("Total steps: %llu\n", totSteps);
  return true;
}

void main(void) {
```

```
  bool valid = verifyCollatzConjecture(1000000000); // 1000 M
}
```

If this testing procedure terminates, the conjecture is true for the tested range of numbers. On the other hand, if a counter example is found, the loop continues forever.

(a) [6 p] Create a parallel version of the given brute force testing procedure by using either OpenMP or Pthreads. For full credits, the issue of load balancing must be dealt with in an appropriate way.

(b) [2 p] There is in fact a risk that overflow can occur in the generation of certain sequences. Thus, even if the testing procedure terminates, the result can not really be trusted. How would you change the code so that it reports whether any overflow occurred or not?

(c) [2 p] There are several algorithmic methods that can be used to accelerate the verification of Collatz conjecture. Suggest at least one way to speed up the sequential testing procedure. Also, discuss in what way your parallel testing procedure can or cannot benefit from the method you proposed.

Solution:

(a) OpenMP makes it easy to create an efficient parallel version of the given brute force method. Since the complexity of the iterations varies in a way that is difficult to predict, dynamic scheduling is preferable. A possible solution is given below.

```
bool verifyCollatzConjecture_OMP(unsigned long long max) {
  unsigned long long totSteps = 0;

#pragma omp parallel
{
  unsigned long long loc_steps = 0, loc_totSteps = 0;

#pragma omp for nowait schedule(dynamic, 100)
  for (int i = 1; i <= max; i++) {
    collatzSteps(i, loc_steps);
    loc_totSteps += loc_steps;
  }

#pragma omp critical
  totSteps += loc_totSteps;

}

  printf("Total steps: %llu\n", totSteps);
  return true;
}
```

(b) Let the function `collatzSteps` signal overflow if n becomes larger than the constant $M = (L - 1)/3$, where L is the largest possible unsigned 64-bit integer.

(c) As soon as any number in a sequence becomes smaller than the starting value, there is no need to finish the sequence, since it has already been done in a previous iteration. In the multi-threaded version, we can do the same, since there is always a thread that either has finished the sequence or will finish it later. However, if the correct number of total steps needs to be reported (as is done in the given program), auxiliary storage and communication issues must be handled.

6. Consider the following program that uses a sequential function called `findSum` to compute the sum of all values in an array:

```
const int ARR_SIZE = 10000000; // 10M

__declspec(align(32)) float arr[ARR_SIZE];

float findSum(float * a, int n) {
  float sum = 0.0f;
  for (int i = 0; i < n; i++) {
    sum += a[i];
  }
  return sum;
}

void initArray(float * a, int n, float val) {
  for (int i = 0; i < n; i++) {
    a[i] = val;
  }
}

void main(void) {
  initArray(arr, ARR_SIZE, 0.000001f);
  float sum = findSum(arr, ARR_SIZE);
}
```

(a) [3 p] Give a new version of the function that computes the sum where the loop has been unrolled four times in such a way that four independent parts of the total sum are maintained in the loop body, which are then added together at the end.

(b) [3 p] Improve your unrolled version by creating a new version that uses vectorized SIMD instructions to maintain the four parts of the sum. You can assume that convenient compiler intrinsics are available to access the SIMD instructions.

(c) [3 p] As an extension of your SIMD version, create a new multi-threaded version of the function by using OpenMP to realize the multi-threading.

(d) [3 p] Discuss and motivate the performance that can be expected for each one of the versions you have created compared to the sequential baseline. In particular, discuss what kind of architectural hardware features they can benefit from.

Solution:

(a) The unrolled version can be implemented like this:

```
float findSum_unrl4(float * a, int n) {
  int i;
  float sum;
  float s[4] = {0.0f, 0.0f, 0.0f, 0.0f};
  for (i = 0; i < n-3; i+=4) {
    s[0] += a[i];
    s[1] += a[i+1];
    s[2] += a[i+2];
    s[3] += a[i+3];
  }
  sum = s[0] + s[1] + s[2] + s[3];

  for (; i < n; i++) {
    sum += a[i];
  }
  return sum;
}
```

(b) The SIMD version using SSE instructions can be implemented like this:

```
float findSum_SSE(float * a, int n) {
  int i;
  float sum;
  __m128 S = { 0.0f, 0.0f, 0.0f, 0.0f };
  __m128 A;
  for (i = 0; i < n - 3; i += 4) {
    A = _mm_load_ps(&a[i]);
    S = _mm_add_ps(S, A);
  }
  sum = S.m128_f32[0] + S.m128_f32[1] +
        S.m128_f32[2] + S.m128_f32[3];

  for (; i < n; i++) {
    sum += a[i];
  }
  return sum;
}
```

(c) A combined OpenMP and SSE version can be implemented like this:

```
float findSum_OMP_SSE(float * a, int n) {
  float sum = 0.0f;
#pragma omp parallel shared(sum, n) num_threads(NUM_THREADS)
  {
    float sum_p;
    __m128 S = { 0.0f, 0.0f, 0.0f, 0.0f };
    __m128 A;
#pragma omp for nowait
    for (int i = 0; i < n - 3; i += 4) {
      A = _mm_load_ps(&a[i]);
      S = _mm_add_ps(S, A);
    }
```

```
        sum_p = S.m128_f32[0] + S.m128_f32[1] +
                S.m128_f32[2] + S.m128_f32[3];
    #pragma omp critical
        {
          sum += sum_p;
        }
      }
      int i = n - n % 4;
      for (; i < n; i++) {
        sum += a[i];
      }
      return sum;
    }
```

(d) The unrolled version may exploit pipelining and superscalar architecture efficiently since the add operations in the loop body are completely independent. Therefore, it is likely to be significantly faster than a strictly sequential version. The SIMD version explicitly utilizes fine-grained instruction parallelism at the hardware level, thereby effectively executing four add operations simultaneously, which suggests an ideal speed-up of 4 compared to a strictly sequential version. And of course, the multi-threaded version benefits from multi-core processors. Assuming 4 cores, where each one utilizes 4 float wide SIMD operations, gives an ideal speed-up of 16, compared to the sequential case.

7. Study the following CUDA kernel that computes the vector addition $c = a + b$ for vectors (arrays) of length n.

```
__global__ void add(float * a, float * b, int n, float * c) {
  int id = threadIdx.x + blockIdx.x * blockDim.x;

  if (id < n) {
    c[id] = a[id] + b[id];
  }
}
```

(a) [2 p] Now suppose the associated kernel launch code is written as follows:

```
add<<<10, 512>>>(a, b, 10000, c);
```

When the code is executed, it will produce the wrong result. Why is that? Explain in detail.

(b) [2 p] Show how the kernel launch code can be corrected so that the given kernel computes the correct answer.

(c) [4 p] Show how the kernel can be rewritten so that each thread is able to compute more than one output value, depending on the total thread count. Also, explain the advantages of using this version of the kernel compared to using the original kernel defined above.

(d) [2 p] Even though the computation performed by this kernel may result in a

useful speed-up compared to a sequential solution, the gain can be expected to be quite modest. Why is that?

Solution:

(a) Too few threads are launched. For this kernel, one thread per array element is needed. Thus, at least 10000 threads are needed, but only 10 blocks with 512 threads per block are launched, i.e., a total of $10 \times 512 = 5120$ threads. Therefore, only the first 5120 elements of the vectors will be added.

(b) Launching 20 blocks with 512 threads in each block gives a total of $20 \times 512 = 10240$ threads, which fits the vector size $n = 10000$ well. Thus, we get the correct result by changing the launch code to

```
add<<<20, 512>>>(a, b, 10000, c);
```

(c) Replacing the if statement with a loop allows each thread to process several elements:

```
__global__ void add(float * a, float * b, int n, float * c) {
  int id = threadIdx.x + blockIdx.x * blockDim.x;
  int st = blockDim.x * gridDim.x;

  while (id < n) {
      c[id] = a[id] + b[id];
      id += st;
  }
}
```

With the new version, the correctness of the kernel computation is not dependent on that the user launches sufficiently many threads in the kernel launch code. Also, this allows more flexible performance tuning. By launching a varying number of threads per block and blocks per grid, we can find out what combination of launch parameters that gives the most efficient parallel execution. Furthermore, the kernel can now be used to add really large vectors. If looping is not used, and one thread is required per vector element, there will be a maximum vector size dictated by hardware limitations.

(d) Since only a single add operation is executed per two loads and one store in global memory, the execution speed of this kernel is bounded by the memory bandwidth. Furthermore, if the vectors must be transferred initially from CPU memory to the GPU memory over the PCIe bus, and after the computation, the resulting vector must be copied back, the overall performance is reduced severely. In such situations, it is important that the transfer costs can be amortized efficiently over the actual parallel computation.

8. [15 p] Sound is typically represented digitally by an sequence of sampled values, where each value is represented by an integer with some number of bits (16, 24, 32,...). Digital sound processing involves transformations of such sequences. Two important

operations, especially when working with recorded sound, are *removal of DC offset* and *normalisation*. Removal of DC offset works by first computing the average of all sampled values in the sequence, and then subtracting this such that the resulting signal is centered around zero. Normalisation means to scale the signal such that the maximal amplitude uses the full range of the chosen integer representation (e.g., if 16 bit signed integers are used to represent the sound samples, then the samples should be in the range [-32768,32767] with at least one sample equal to -32768 or 32767). Thus, an essential part of the normalisation is to compute a scale factor that yields this while not causing any samples to be clipped. Subsequently, all values in the signal are multiplied with that factor to form the normalised signal.

Define a data parallel algorithm that first removes the DC offset from a sound signal, and then normalises it! You can assume that the signal sample bit length is 16 bits, and that the signal is stored in a data parallel array of 16 bit signed integers. Use the pseudocode notation that we have been using during the lectures on data parallelism and parallel algorithms (not, for instance, OpenMP).

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.

Hint: You can assume two binary operators "`max`" and "`min`" that compute the maximal and minimal values of their arguments, respectively. You may also use the "`abs`" function that returns the absolute value of its argument.

Solution: We assume that the sound signal is stored in the array `A`, with `n` elements. The algorithm updates `A` in-place. The "`max`" and "`min`" operators are associative, and thus they can be used with the `reduce` functions to compute the maximal and minimal values of a data parallel array.

```
// DC offset removal:
average = reduce(+,A)/n
for all i in parallel do
 A[i] = A[i] - average
// Normalisation:
maxint = 32767
minint = -32768
max_signal = reduce(max,A)
min_signal = reduce(min,A)
if abs(max_signal) >= abs(min_signal)
  then scale = abs(maxint/max_signal)
  else scale = abs(minint/min_signal)
for all i in parallel do
  A[i] = A[i] * scale
```

What is the parallel time complexity? The `for all .. in parallel` statements are all $O(1)$ parallel time, on $O(n)$ processors. The calls to reduce are each

$O(\log n)$ parallel time, on $O(n)$ processors. There are no sequential loops. Thus, the total parallel time complexity is $O(\log n)$ on $O(n)$ processors. This is clearly significantly better than for a sequential algorithm, which has to be at least $O(n)$.

Total: 80