Exam

DVA336 Parallel Systems

8:10–12:30

August 16, 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. Consider the terms: SIMD, MIMD and SIMT. Then:
   (a) [4 p] Give their definitions and make some examples of real architectures that implement them.
   (b) [6 p] What is the main difference among them when you write a program on systems corresponding to each of them?

Solution: These terms refer to the most common architectural approaches for nowadays processing units. SIMD stands for Single Instruction flow, Multiple Data flow, for example a processor which is capable of executing the AVX/SSE instruction set. Typical application are number crunching computing applications. Here the same instruction is applied to a register made up of several values packed together in special registers.The program, however, follows a unique instruction flow. The term MIMD (Multiple Instruction flow, Multiple Data flow) refers to parallel processors capable to run different instruction flows on different data (for example several threads executing different instruction on different portions of data) like multi-core CPUs do. SIMT architectures (which is a term invented for GPUs) are similar to SIMD but values don't need to be 'packed' together and threads can follow different paths. However, when it happens, performance can be heavily penalized. For example, since threads of the same warp are synchronized on the same instruction, when they follow different paths of an if statement, the execution is serialized which penalizes the overall efficiency. Furthermore, when threads access non-coalesced data in the memory, the bandwidth of the communications usually degrades.

2. (a) [2 p] What is the parallel time complexity for sorting n numbers, which need at most m bits each to be represented, for bitonic sort?
   (b) [2 p] Same question, for radix sort?
   (c) [3 p] Now assume that you are to sort 10^9 numbers, all in the range $[0 \ldots 255]$, on a massively parallel machine. Would you use bitonic sort or radix sort to do this? Motivate your answer!

Solution:

(a) $O(\log^2 n)$

(b) $O(\log m \cdot \log n)$

(c) I would choose radix sort. The reason is the parallel time complexities of the algorithms. Since all numbers are in the range $[0 \ldots 255]$, m is small. Furthermore, n is very large. It is therefore likely that radix sort will be much faster than bitonic sort.

3. In geometry, the bounding box of a given set S of points is the smallest box within which all the points in S lie. The Axis Aligned Bounding Box (AABB) for S is its minimum bounding box of which edges are parallel to the coordinate axes. It is simply defined, for each dimension, by the minimal and maximal value of the corresponding coordinate assumed by the points in S, as shown in the code below:

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

#define D 3 //number of dimensions (D>0)

struct point {
  float v[D];
  void printme() const {
    printf("(%f", v[0]);
    for(int i=1; i<D; ++i) printf(",%f", v[i]);
    printf(")");
  }
};

struct AABB {
  void printme() const {
    printf("min="); min.printme(); printf(",\n");
    printf("max="); max.printme(); printf(".\n");
  }
  point min, max;
};

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

void init(point* S, int n) {
  for(int i=0; i<n; ++i)
    for(int j=0; j<D; ++j)
      S[i].v[j] = rand()/(RAND_MAX+1.0f);
};

AABB calc(point* S, int n) {
  AABB aabb;
  for(int j=0; j<D; ++j)
    aabb.min.v[j] = aabb.max.v[j] = S[0].v[j];
  for(int i=1; i<n; ++i)
    for(int j=0; j<D; ++j) {
      if(S[i].v[j]<aabb.min.v[j]) aabb.min.v[j] = S[i].v[j];
      if(S[i].v[j]>aabb.max.v[j]) aabb.max.v[j] = S[i].v[j];
    }
  return aabb;
}

int main(int argc, char* argv[]) {
  int n = atoi(argv[1]);
    point* S = alloc(n);
  init(S, n);
  AABB aabb = calc(S, n);
  printf("AABB of S is defined by:\n");
  aabb.printme();
```

```
  return 0;
}
```

(a) [8 p] Provide an equivalent SIMD (AVX/SSE) version of the program and explain what modifications are required (assume `n`>0 and multiple of 8).

(b) [2 p] What is the expected speedup for the SIMD version of the `calc` 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, which means to have a distinct array for each coordinate axis instead of an array of points; *ii*) allocate aligned memory; *iii*) rewrite the function `calc`.

```
#include <stdlib.h>
#include <stdio.h>
#include <mm_malloc.h>
#include <immintrin.h>

#define D 3 //number of dimensions

struct point {
  float v[D];
  void printme() const {
    printf("(%f", v[0]);
    for(int i=1; i<D; ++i) printf(",%f", v[i]);
    printf(")");
  }
};

struct AABB {
  void printme() const {
    printf("min="); min.printme(); printf(",\n");
    printf("max="); max.printme(); printf(".\n");
  }
  point min, max;
};

struct avxpoints {
  float * v[D];
  int n;
};

avxpoints avxalloc(int n) {
  avxpoints S;
  for(int d=0; d<D; ++d)
    S.v[d] = (float*)_mm_malloc(sizeof(float)*n,32);
  S.n = n;
  return S;
};

void avxinit(avxpoints & S) {
  for(int i=0; i<S.n; ++i)
    for(int d=0; d<D; ++d)
      S.v[d][i] = rand()/(RAND_MAX+1.0f);
};
```

```
float getmin(__m256 p) {
  p = _mm256_min_ps(p, _mm256_permute2f128_ps(p, p, 0x81));
  p = _mm256_min_ps(p, _mm256_shuffle_ps(p, p, 0x0e));
  p = _mm256_min_ps(p, _mm256_shuffle_ps(p, p, 0x01));
  return *((float*)&p);
}

float getmax(__m256 p) {
  p = _mm256_max_ps(p, _mm256_permute2f128_ps(p, p, 0x81));
  p = _mm256_max_ps(p, _mm256_shuffle_ps(p, p, 0x0e));
  p = _mm256_max_ps(p, _mm256_shuffle_ps(p, p, 0x01));
  return *((float*)&p);
}

AABB avxcalc(avxpoints & S) {
  AABB aabb;
  for(int d=0; d<D; ++d) {
    __m256 __p, __min, __max;
    __min = __max = _mm256_load_ps(S.v[d]);
    for(int i=8; i<S.n; i+=8) {
      __p = _mm256_load_ps(&S.v[d][i]);
      __min = _mm256_min_ps(__min, __p);
      __max = _mm256_max_ps(__max, __p);
    }
    aabb.min.v[d] = getmin(__min);
    aabb.max.v[d] = getmax(__max);
  }
  return aabb;
}

int main(int argc, char * argv[]) {
  int n = atoi(argv[1]);
  avxpoints S = avxalloc(n);
  avxinit(S);
  AABB aabb = avxcalc(S);
  printf("(avx) AABB of S is defined by:\n");
  aabb.printme();
  return 0;
}
```

For large values of `n`, the speedup of the `avxcalc` function above should be $\sim 8\times$ since AVX instructions can handle 8 values at once.

4. (a) [4 p] State and explain how to compute the speedup of a parallel program according to the *Amdahl's Law*.

   (b) [3 p] What is the speedup if we parallelize 40% of a program by making it 8 times faster? And if we make it 80 times faster?

   (c) [3 p] It seems that Amdahl's law heavily bounds the the benefit of running a computation on high parallel systems. Why, in HPC, such systems are built anyway?

Solution:

(a) Amdahl's law gives an estimation of the speedup that we can achieve in solving a problem when more processors are added to the system. The parallel speedup $S(p)$ with p processors is given by:

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

where f denotes the portion of execution time which benefits from the parallelization. When $p \to \infty$, we get an estimation of the maximum speedup we can achieve that is $1/(1-f)$ since $f/p \to 0$.

(b) Here p is not given, instead, we have the speedup of the parallel computation. Hence the calculation is as it follows:

$$8\times \text{ faster} \Rightarrow S = \frac{1}{(1-0.4) + \frac{0.4}{8}} = \frac{1}{0.65} \simeq 1.54\times$$

$$80\times \text{ faster} \Rightarrow S = \frac{1}{(1-0.4) + \frac{0.4}{80}} = \frac{1}{0.605} \simeq 1.65\times$$

(c) The Amdahl's law is calculated by keeping fixed the problem size while, nowadays, the amount data to manage grows constantly so that more powerful computers are used to solve problems with growing-size input.

5. A Binary Search Tree (BST) is a binary tree which satisfies the binary search property. In particular, each node of the tree stores a key k, and has two distinct sub-trees so that the key of the nodes stored on the left sub-tree are less than k while any key stored in the right sub-tree is greater than k. The code below implements a simple BST to count the occurrences of each character in an input file:

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

struct bst_node {
  bst_node(char key) : key(key) {
  }
  ~bst_node() {
    delete left;
    delete right;
  }
  char key;
  int counter = 0;
  bst_node* left = NULL;
  bst_node* right = NULL;
};

void set(bst_node* & ptnode, char key) {
```

```
  if(!ptnode)
    ptnode = new bst_node(key);
  if(ptnode->key!=key)
    set(key<ptnode->key?ptnode->left:ptnode->right, key);
  else
    ptnode->counter += 1;
}

void sorted_print(bst_node* ptnode) {
  if(!ptnode)
    return;
  sorted_print(ptnode->left);
  printf("'%c' %dx\n", ptnode->key, ptnode->counter);
  sorted_print(ptnode->right);
}

struct bst {
  public:
    ~bst() { delete root; }
    void Set(char key) { set(root, key); }
    void Print() { sorted_print(root); }
  private:
    bst_node* root = NULL;
};

bst tree;

FILE* f = NULL;

int main(int argc, char* argv[]) {
  f = fopen(argv[1], "r");
  for(char ch=fgetc(f); ch!=EOF; ch=fgetc(f)) tree.Set(ch);
  fclose(f);
  tree.Print();
}
```

(a) [12 p] Modify the code in order to make thread-safe* the insertion/update operation on the `bst` data structure. Note that the solution must allow threads to use such a data structure simultaneously as much as possible.

(b) [4 p] Provide a new `main` function that uses your multi-threaded version of `bst`. Note that `fgetc()`, which is used for reading a character at-a-time from the file, is not thread-safe.

Solution:

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

struct bst_node {
  bst_node(char key) : key(key) {
```

* Thread safety means that the multi-threaded code manipulates shared data structures in a manner that ensures that all threads interact properly and fulfill the design specifications.

```
    pthread_mutex_init(&lock, NULL);
  }
  ~bst_node() {
    delete left;
    delete right;
    pthread_mutex_destroy(&lock);
  }
  char key;
  int counter = 0;
  bst_node* left = NULL;
  bst_node* right = NULL;
  pthread_mutex_t lock;
};

void set(bst_node* & ptnode, char key, pthread_mutex_t & lock) {
  pthread_mutex_lock(&lock);
  if(!ptnode)
    ptnode = new bst_node(key);
  pthread_mutex_unlock(&lock);
  if(ptnode->key!=key)
    set(key<ptnode->key?ptnode->left:ptnode->right, key, ptnode->lock);
  else {
    pthread_mutex_lock(&ptnode->lock);
    ptnode->counter += 1;
    pthread_mutex_unlock(&ptnode->lock);
  }
}

void sorted_print(bst_node* ptnode) {
  if(!ptnode)
    return;
  sorted_print(ptnode->left);
  printf("'%c' %dx\n", ptnode->key, ptnode->counter);
  sorted_print(ptnode->right);
}

struct bst {
  public:
    bst() {
      pthread_mutex_init(&lock, NULL);
    }
    ~bst() {
      delete root;
      pthread_mutex_destroy(&lock);
    }
    void Set(char key) { set(root, key, lock); }
    void Print() { sorted_print(root); }
  private:
    bst_node* root = NULL;
    pthread_mutex_t lock;
};

bst tree;

FILE* f = NULL;
```

```
#define NTHREADS 8

pthread_t threads[NTHREADS];

pthread_mutex_t flock;

char mutex_fgetc(FILE* f) {
  pthread_mutex_lock(&flock);
  char ch = fgetc(f);
  pthread_mutex_unlock(&flock);
  return ch;
}

void* Run(void* arg) {
  for(char ch=mutex_fgetc(f); ch!=EOF; ch=mutex_fgetc(f)) tree.Set(ch);
  return NULL;
}

int main(int argc, char* argv[]) {
  f = fopen(argv[1], "r");
  pthread_mutex_init(&flock, NULL);
  for(int i=0; i!=NTHREADS; ++i)
    pthread_create(&threads[i], NULL, &Run, NULL);
  for(int i=0; i!=NTHREADS; ++i)
    pthread_join(threads[i], NULL);
  pthread_mutex_destroy(&flock);
  fclose(f);
  tree.Print();
}
```

Explanation: to allow different threads to insert/update simultaneously the tree, each node is provided with a `lock` variable (plus one used to lock the `root` pointer). Function calls for creating and destroying the `lock` variables are included, respectively, in the constructor and destructor of each data structure. During the execution of the `set` function, we have to handle three different cases:

- creation of a new node: we have to preventively acquire the `lock` passed as argument (which belongs, in general, to the last node visited since the pointer to the new node is stored there) and, once the new node has been created, release `lock`.
- counter update: to update the `counter` variable we must acquire the `lock` of the current node (`ptnode->lock`) where the counter is stored. Once the counter has been increased, we release `ptnode->lock`.
- recursive step: no modification is done (mutex is not required) so that the function continues to search the target `key` by moving to the next node.

The new function `main` creates the threads that read in mutual-exclusion a character at-a-time from the file. Mutual exclusion is ensured by the `flock` variable used to implement a thread-safe version of `fgetc()`. Once the computation is ended for all the threads, we can print the tree.

6. (a) [8 p] Write an MPI program that gets an integer `n` (`n`>2) as input and find the set of all the prime numbers between 0 and `n`. Moreover, the set must be stored in the address space of the master process (i.e., the process with rank equal to 0) when all prime numbers have been found. To test if a number `x` is prime, the following function is given:

```
bool isprime(int x) {
  //return true is x is prime, false otherwise.
  if (x<2) return false;
  if (x==2) return true;
  for (int i=3; i<x; i+=2)
    if (x%i==0) return false;
  return true;
}
```

Solution:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "isprime.hpp" // isprime() implementation

struct array {
  ~array() {
    free(data);
  }
  void push(int item) {
    if (size==maxsize) {
      maxsize = 2*maxsize+1;
      data = (int*) realloc(data, sizeof(int)*maxsize);
    }
    data[size++] = item;
  }
  void send(int dst) {
    MPI_Send(&size, 1, MPI_INT, 0, 100, MPI_COMM_WORLD);
    if (size==0) return;
    MPI_Send(data, size, MPI_INT, 0, 101, MPI_COMM_WORLD);
  }
  void recv(int src) {
    int n;
    MPI_Status st;
    MPI_Recv(&n, 1, MPI_INT, src, 100, MPI_COMM_WORLD, &st);
    if (n==0) return;
    if (size+n>maxsize) {
      maxsize = size+n;
      data = (int*) realloc(data, sizeof(int)*maxsize);
    }
    MPI_Recv(data+size, n, MPI_INT, src, 101, MPI_COMM_WORLD, &st);
    size += n;
  }
  void print() { //optional function
    for (int i=0; i<size; ++i) printf("%d\n", data[i]);
  }
  int* data = NULL;
  int size = 0;
```

```
  int maxsize = 0;
};

int main (int argc, char *argv[]) {
  MPI_Init(&argc,&argv);
  int r, p, n = atoi(argv[1]);
  MPI_Comm_rank(MPI_COMM_WORLD,&r);
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  array primes;
  for (int i=2*r+3; i<=n; i+=2*p)
    if (isprime(i)) primes.push(i);
  if(r==0) {
    primes.push(2);
    for(int i=1; i<p; ++i) primes.recv(i);
    primes.print(); //optional
  } else
    primes.send(0);
  MPI_Finalize();
  return 0;
}
```

Explanation: the main point of this exercise is how to balance the workload among the processes. To this end, the range of values to test, i.e., $[0, \texttt{n}]$, is divided among the `p` processes so that the numbers of the form `2*r+3+2*p*`k, for any plausible value of $k \in \mathbb{N}$, are assigned to the process with rank `r`. For example, if `n`=25 and `p`=4, the process p with rank `r`=0, i.e. p_0, tests $\{3, 11, 19\}$, p_1 tests $\{5, 13, 21\}$, p_2 tests $\{7, 15, 23\}$, and p_3 tests $\{9, 17, 25\}$ (the prime number 2 is directly added by p_0). In this way the workload is reasonably balanced among the existing processes given that, except for the number 2, all primes are odd numbers and the complexity of the function `isprime(x)` is almost proportional to `x`. Finally, the p_0 collects the prime numbers sent by the other processes.

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 λ be the average number of items arriving per unit time, the law is $L = \lambda W$.

It is used to show how it is possible to improve throughput of the computation on GPUs by increasing instruction level of parallelism in the pipelined scalar processors (SPs).

8. (a) [13 p] A simplified, one-dimensional version of Conway's Game of Life can be defined as follows. There are n cells with positions $1, \ldots, n$. Each cell can be either alive or dead, For each step in the game, each cell changes status as follows:

   - a live cell with no live neighbor dies,
   - a live cell with two live neighbors dies,
   - a dead cell with one live neighbor becomes alive, and
   - in all other situations a cell keeps its status.

   Define a data parallel algorithm that given an initial array of cell values will play the game until no cells change their values anymore!

   Use data parallel arrays to represent the array of cells. You can assume that these arrays are indexed from 0 and up. 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 for a single iteration of your implementation (including the test if cells have changed)? 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: don't forget to handle the cells at the endpoint positions 1 and n in a correct manner.

Solution: We use a data parallel array `cell` to represent the n cells. A cell that is dead has the value `0`, and a live cell has the value `1`. The cells are represented by the array elements `cell[i]` at position $1, \ldots, n$, and we assume that the initial configuration of the game is stored in this sub-array. We initialize the elements `cell[0]` and `cell[n+1]` to `0`, keeping these values throughout the game: in that way, the summation of live neighbors will work also at the endpoints.

In each iteration we compute, in parallel, an array `newcell` holding the new cell confguration after applying the rules to the current configuration. We then compare `cell` and `newcell` element-wise, and we use reduction with `or` over the resulting boolean array to check whether the value of any cell has changed. If this is the case then we set `cell = newcell` and iterate one more time, otherwise we exit. Pseudocode is given below:

```
cell[0] = 0
cell[n+1] = 0
repeat
  forall i where 1 <= i <= n in parallel do
    newcell[i] = cell[i]
    if cell[i] = 1 and cell[i-1] + cell[i+1] == 0 then newcell[i] = 0
    if cell[i] = 1 and cell[i-1] + cell[i+1] == 2 then newcell[i] = 0
    if cell[i] = 0 and cell[i-1] + cell[i+1] == 1 then newcell[i] = 1
    diff[i] = newcell[i] <> cell[i]
    changed = reduce(diff,or)
until not changed
```

What is the parallel time complexity for an iteration? The elementiwise assignments are $O(1)$ time, and the reduction is $O(\log n)$ time. Thus, the parallel time complexity for an iteration is $O(\log n)$. A sequential algorithm must at least iterate through all elements in the cell array once, giving it a time complexity of at least $O(n)$. Thus, the parallel algorithm has significantly lower time complexity.

Total: 80