Exam in HPPS

January 20–25, 2023

Preamble

This document consists of 14 pages including this preamble; make sure you have them all. Your solution is expected to consist of a *short* report in PDF format, as well as a `.zip` or `.tar.gz` archive containing your source code, such that it can be compiled immediately with `make` (i.e. include the non-modified parts of the handout as well).

You are expected to upload *two* files in total for the entire exam. The report must have the specific structure outlined in section 3.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- If you believe there is an ambiguity or error in the exam text, *contact one of the teachers*. If you are right, we will announce a correction.
- Your final grade is based on an overall evaluation of your solution, but *as a guiding principle* each task and question is weighted equally.

Make sure to read the entire text before starting your work. There are useful hints at the end.

1 Library design

The overall theme of this exam problem is to implement a small C library for manipulating objects representing scalars, vectors, and matrices. Further, the library supports *serialising* these objects to and from files. The disk format is precisely specified below. The support for serialisation is used to implement a range of small programs that manipulate files as if they were mathematical objects. Most of these programs have already been written for you, and serve as examples of how to use the library.

Concretely, in `matlib.c` you will implement a collection of C functions and datatypes declared in `matlib.h`. Each of the subtasks involves implementing one or more functions.

These general principles apply:

- You must not modify `matlib.h`.
- See `matlib.h` for the specific types of the functions.
- Even if you do not manage to implement all of the functions, you can still answer the questions in section 3 with the functions you did manage to implement.
- **Make sure to compile your code with optimisations enabled when benchmarking.** See also section 5.

The specific functions you must implement are listed below, but first we must discuss which object formats are supported.

1.1 Supported objects

The library supports four different types of objects, implemented as pointers to four different C types: `struct scalar`, `struct vector struct matrix_dense`, `struct matrix_csr`.

Any object can be stored in a file. A valid file starts with a four-byte integer called a *magic number* that indicates the type of object contained in the file. Each object type has a distinct magic number, and for convenience they correspond to four-character ASCII strings. For clarity, the examples in this exam also use the magic number (in ASCII form) as a file extension, but to the library, file names do not matter. After the magic number comes the actual *representation* of the object, in an object-specific format.

1.1.1 Type `struct scalar`

This type respresents single 64-bit floating-point numbers. No operations are supported beyond reading and writing to disk, and converting to a C `double`. They are produced when computing the dot product of two vectors.

A file storing this type of object contains the following, in order:

1. A magic number corresponding to the ASCII string `"SCLR"`.
2. Eight bytes storing a `double`.

1.1.2 `struct vector`

This type represents a vector of n double-precision floating-point numbers.

A file storing this type of object contains the following, in order:

1. A magic number corresponding to the ASCII string `"VDNS"`.
2. Four bytes storing an `int`, which is the length n of the vector.
3. n `doubles` (that is, $8 \cdot n$ bytes), which are the elements of the vector.

1.1.3 `struct matrix_dense`

This type represents a matrix of $n \cdot m$ double-precision floating-point numbers, stored in row-major order.

A file storing this type of object contains the following, in order:

1. A magic number corresponding to the ASCII string `"MDNS"`.
2. Four bytes storing an `int`, which is the number of rows n of the matrix.
3. Four bytes storing an `int`, which is the number of columns m of the matrix.
4. $n \cdot m$ `doubles` (that is, $8 \cdot n \cdot m$ bytes), which are the elements of the matrix in row-major order.

1.1.4 `struct matrix_csr`

This type represents a *sparse* matrix of $n \cdot m$ double-precision floating-point numbers in Compressed-Sparse-Row (CSR) format. A sparse matrix is a matrix where we store only the *nonzero* elements, which means it uses much less memory than the usual "dense" representations when the matrix is mostly zeroes. There are many formats for sparse matrices, but for this exam we will use the CSR format.

A matrix in CSR format consists of three integers:

- n, the number of rows.
- m, the number of columns.
- nnz, the number of nonzero elements.

And three single-dimensional arrays:

- v, an array of size nnz containing double precision numbers. Note that despite the name, some of these elements may actually be zero—that is not a problem.
- v_{col}, an array of size nnz containing for each corresponding element in v, its column as an integer.
- row_{start}, an array of size n such that $row_{start}[i]$ contains the integer index where the first element of row i occurs in v.

For example, the matrix

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

is represented as

$$\begin{array}{lcl} v & = & [5, 8, 3, 6] \\ v_{col} & = & [0, 1, 2, 1] \\ row_{start} & = & [0, 1, 3, 3] \end{array}$$

Note:

- An empty row has the same start index as the following row.
- We can find the (exclusive) end of the values for a row i in $row_{start}[i+1]$, except for the last row $i = n-1$, where the (exclusive) end is nnz.

A file storing CSR matrices contains the following, in order:

1. A magic number corresponding to the ASCII string `"MCSR"`.
2. Four bytes storing an `int` (n).
3. Four bytes storing an `int` (m).
4. Four bytes storing an `int` (nnz).
5. An sequence of nnz `double`s (the v array).
6. An sequence of nnz `int`s (the v_{col} array).
7. An sequence of n `int`s (the row_{start} array).

1.2 Helper programs

The code handout contains a number of small programs that make use of `matlib.c`. Compile them by running `make`. They read and write files corresponding to library objects. Initially few of the programs will work, but as you implement more of the library, more of them will become functional. You can use them for testing and benchmarking. Their behaviour is summarised below. For full details, see their source code—they are quite small. You can also consult the usage examples in section 6.

The first four programs will work without any changes to `matlib.c` and can be used to generate test data.

`./random-scalar FILE`
: Generate a random scalar and write it to `FILE`.

`./random-vector` n `FILE`
: Generate a random vector of n elements and write it to `FILE`.

`./random-matrix-dense` n m `FILE`
: Generate a random matrix of $n \times m$ elements and write it to `FILE`.

`./random-matrix-csr` n m a b `FILE`

Generate a random sparse matrix of $n \times m$ elements and write it to `FILE`. The a and b arguments must be between 0 and 1 and control the sparsity—in the first row, each element has probability a of being nonzero, while in the last row the probability is b. For the intervening rows, the probability is a smooth interpolation of a and b. If $a = b$ the nonzero entries are uniformly distributed.

The following four programs will work once you have implemented the various reading and indexing functions. These programs are useful for checking whether the arithmetic functions are correct, as you use them to show their results.

`./print-scalar FILE`

Print the scalar in the provided file.

`./print-vector FILE`

Print the vector in the provided file.

`./print-matrix-dense FILE`

Print the dense matrix in the provided file.

`./print-matrix-csr FILE`

Print the sparse matrix in the provided file.

The following five programs will require implementation of writing and arithmetic functions.

`./mul-vv Z X Y`

Read vectors from `X` and `Y`, compute their dot product, and write it to `Z`.

`./mul-mv Z X Y`

Read dense matrix from `X` and vector from `Y`, compute the product $Z = XY$, and write it to `Z`.

`./mul-mTv Z X Y`

Read dense matrix from `X` and vector from `Y`, compute the product $Z = X^T Y$, and write it to `Z`.

`./mul-spmv Z X Y`
: Read sparse matrix from `X` and vector from `Y`, compute the product $Z = XY$, and write it to `Z`.

`./mul-spmTv Z X Y`
: Read sparse matrix from `X` and vector from `Y`, compute the product $Z = X^TY$, and write it to `Z`.

Finally, you are provided with five benchmarking programs:

`./benchmark-mul-vv X Y RUNS`
: Compute mean runtime of vector-vector product.

`./benchmark-mul-mv X Y RUNS`
: Compute mean runtime of matrix-vector product.

`./benchmark-mul-mTv X Y RUNS`
: Compute mean runtime of transposed-matrix-vector product.

`./benchmark-mul-spmv Z X RUNS`
: Compute mean runtime of sparse matrix-vector product.

`./benchmark-mul-spmTv Z X RUNS`
: Compute mean runtime of transposed sparse matrix-vector product.

You are free to modify any of the programs if you wish. In particular, you may want to modify the benchmarking programs, or write new ones based on them.

2 Your task

You are given definitions for the C structs, as well as a complete implementation of functions for the `struct scalar` type. Your implementation task is to implement the remaining functions for vectors, dense matrices, and sparse matrices. The header file contains information about the error cases you must also handle. Use parallelism (with OpenMP) as appropriate[1].

The precise functions to implement are listed below. After your implementation is done, you will be performing performance analysis (see section 3).

[1]Only in the arithmetic functions `mul_vv()`/`mul_mv`/`mul_mTv`/`mul_spmv`/`mul_spmTv`.

2.1 Task: Vectors

Implement the following functions in `matlib.c`:

- `read_vector()`
- `write_vector()`
- `free_vector()`
- `vector_n()`
- `vector_idx()`, retrieve the ith element of the vector.
- `mul_vv()`, which computes the dot product of two vectors and produces a single scalar. Parallelise this with OpenMP as appropriate.

2.2 Task: Dense Matrices

- `read_matrix_dense()`
- `write_matrix_dense()`
- `free_vector_dense()`
- `matrix_dense_n()`
- `matrix_dense_m()`
- `matrix_dense_idx()`, retrieve the element at position (i, j) in the matrix.
- `mul_mv()`, which multiplies an $n \times m$ matrix with an m-element vector in the conventional way, returning an n-element vector. Parallelise this with OpenMP as appropriate.
- `mul_mTv()`, which multiplies the *transpose* of an $n \times m$ matrix with an n-element vector, returning an m-element vector. Do not actually transpose the matrix in memory; instead traverse its columns. Parallelise this with OpenMP as appropriate.

2.3 Task: Compressed-Sparse-Row (CSR) matrices

- `read_matrix_csr()`
- `write_matrix_csr()`
- `free_vector_csr()`
- `matrix_csr_n()`
- `matrix_csr_m()`
- `matrix_csr_idx()` retrieves the element at position (i, j) in the sparse matrix. Note that element (i, j) may not correspond to an element in the v array, in which case 0 should be returned.
- `mul_spmv()` multiplies a $n \times m$ sparse matrix with an m-element vector, returning an n-element vector x vector. Pseudocode:

  ```
  for i in 0..n-1:
    start = row_start[i]
    if i+1 < n: end = row_start[i+1]
       else:    end = nnz
    y[i] = 0
    for l in start..end-1:
      y[i] = y[i] + v[l] * x[v_col[l]]
  ```

  Feel free to diverge from the pseudocode as you wish. Parallelise your function with OpenMP as appropriate.
- `mul_spmTv()` multiplies the transpose of an $n \times m$ sparse matrix with an n-element vector x, returning an m-element vector. Pseudocode:

  ```
  for j in 0..m-1:
    y[j] = 0
  for i in 0..n-1:
    start = row_start[i]
    if i+1 < n: end = row_start[i+1]
       else:    end = nnz
    for l in start..end-1:
      y[v_col[l]] = y[v_col[l]] + v[l] * x[i]
  ```

Feel free to diverge from the pseudocode as you wish. Parallelise your function with OpenMP as appropriate.

3 The structure of your report

Do not put your name in your report. The exam is supposed to be anonymous. Your report must be structured exactly as follows.

Introduction:
Briefly mention general concerns, your own estimation of the quality of your solution, whether it is fully functional, which programs you have written or modified in order to answer the questions, and possibly how to run your tests. Make sure to report the computer you are benchmarking on (in particular the core count).

Sections answering the following specific questions:

1. a) Does `mul_vv()` exhibit good temporal and/or spatial locality? Why or why not?
   b) How did you parallelise `mul_vv()`?
   c) Measure and show the weak and strong scalability of `mul_vv()` as you vary the number of threads. Explain how you chose the datasets.
2. a) Does `mul_mv()` exhibit good temporal and/or spatial locality? Why or why not?
   b) How did you parallelise `mul_mv()`?
   c) Measure and show the weak and strong scalability of `mul_mv()` as you vary the number of threads. Explain how you chose the datasets.
3. a) Does `mul_mTv()` exhibit good temporal and/or spatial locality? Why or why not?
   b) How did you parallelise `mul_mTv()`?
   c) Measure and show the weak and strong scalability of `mul_mTv()` as you vary the number of threads. Explain how you chose the datasets.

4. a) Does `mul_spmv()` exhibit good temporal and/or spatial locality? Why or why not?
   b) How did you parallelise `mul_spmv()`?
   c) Measure and show the weak and strong scalability of `mul_spmv()` as you vary the number of threads. Explain how you chose the datasets.
5. a) Does `mul_mTv()` exhibit good temporal and/or spatial locality? Why or why not?
   b) How did you parallelise `mul_spmTv()`?
   c) Measure and show the weak and strong scalability of `mul_spmTv()` as you vary the number of threads. Explain how you chose the datasets.

4 The code handout

The code handout contains the following.

- `timing.h`: Helper function for recording the passage of time. **Do not modify this file.**
- `matlib.h`: Function prototypes. **Do not modify this file.**
- `matlib.c`: Stub function definitions that you will have to fill out.
- `Makefile`: You may want to modify the `CFLAGS` for debugging and benchmarking. You are also free to add targets for new programs if you wish, or to modify the existing ones.
- The helper programs listed in section 1.2.

You can run `make datafiles` to produce a variety of random data files. These are not sufficient to answer the questions in section 3, but show how data generation works.

You can afterwards run `make benchmark` to run the included benchmark programs. Again, these are not sufficient to answer the questions in section 3, but can be used for inspiration.

Feel free to add more programs for benchmarking if you wish, or to modify the existing ones.

5 General advice

- You may assume that the machine uses little endian byte ordering.
- Use the implementation of the functions for the `struct scalar` type as inspiration for how to write the IO functions.
- Make sure that the code you hand in compiles. If some of your code is unfinished and does not compile, *comment it out* or remove it. The code in the handout is incomplete but it does compile. Do not hand in something that is less functional thah what you have been given.
- The IO functions are similar to the ones you worked with for A2.
- If may be a good idea to switch between implementation and report writing. If you find some part of the implementation difficult (e.g. sparse matrices), write report sections for the implementation parts you have already one.
- If you are short on time, skip error detection in the read/write functions.
- It is up to you to decide on proper workloads (i.e. input sizes) that can provide good answers to the questions asked in section 3. Make sure to report the workloads you use.
- You don't *have* to show your speedup results as graphs, although it is preferable. If you are short on time, tables with numbers will do.
- Use `OMP_NUM_THREADS` to change how many threads are used when running OpenMP programs.
- The Makefile contains two sets of `CFLAGS`: one that is good for debugging, and one that is good for benchmarking. You can uncomment the ones you wish to use. Remember to recompile everything afterwards (`make -B`).
- All else being equal, **a short report is a good report**.

6 Usage examples

None of the material in this section is normative with respect to the exam and you do not need to read it, but it may be useful.

6.1 Generating a scalar and printing it

```
$ ./random-scalar random.sclr
$ ./print-scalar random.sclr
0.840188
```

6.2 Generating two vectors and multiplying them

First generate the vectors.

```
$ ./random-vector 5 foo.vdns
$ ./random-vector 5 bar.vdns
```

Then check their contents.

```
$ ./print-vector foo.vdns
0.288008
0.296081
0.685892
0.635819
0.063751
$ ./print-vector bar.vdns
0.148586
0.830137
0.222124
0.637007
0.404760
```

Then take their dot product to produce a scalar.

```
$ ./mul-vv prod.sclr foo.vdns bar.vdns
```

Then print the scalar.

```
$ ./print-scalar prod.sclr
0.871760
```

We can verify this result by computing the same dot product in our favourite tool or programming language for matrix programming[2].

6.3 Multiplying dense matrix with vector

We reuse the vector `foo.vdns` generated above.

```
$ ./random-matrix-dense 3 5 foo.mdns
$ ./print-matrix-dense foo.mdns
0.707739 0.793380 0.636152 0.995961 0.726196
0.467314 0.639043 0.066773 0.467036 0.245445
0.101050 0.356289 0.223518 0.759802 0.930937
$ ./mul-mv prod.vdns foo.mdns foo.vdns
$ ./print-vector prod.vdns
1.554619
0.682197
0.830348
```

6.4 Multiplying dense matrix with vector

```
$ ./random-matrix-csr 3 5 0.3 0.5 foo.mcsr
$ ./print-matrix-csr foo.mcsr
0.000000 0.000000 0.000000 0.000000 0.000000
0.236727 0.000000 0.730262 0.000000 0.000000
0.367116 0.688950 0.553302 0.000000 0.404347
$  ./mul-spmv prod.vdns foo.mcsr foo.vdns
$ ./print-vector prod.vdns
0.000000
0.569061
0.715001
```

Note that when verifying the result, we don't have to care that the matrix is internally represented in CSR format - the mathematical object that is modelled is simply a matrix.

[2]`https://futhark-lang.org`