Exam in HPPS

January 21—25, 2022

Preamble

This document consists of 7 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 2.

- 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 Your task

Your overall task is to implement various matrix algorithms in C, analyse their performance both empirically and analytically, and optimise them through improving locality and adding parallel execution. You will be judged on the overall quality of your code and your response to the questions in section 2.

In `matlib.c` you will implement a collection of C functions declared in `matlib.h`. You must not modify `matlib.h`. Each of the following subtasks involves implementing one or more functions.

These general principles apply:

- See `matlib.h` for the specific types of the functions.
- All matrices passed to functions or produced by functions are assumed to be in row-major order.
- Memory for result matrices must be allocated by the caller. Functions may allocate memory for internal intermediate results, but should free it accordingly. Your programs must be free of memory leaks.
- When parallelising a function you must use OpenMP. You may change the code in order to improve the parallelisation, but don't add additional unrelated optimisations (e.g. locality) unless the task specifically calls for it. It is up to you to decide how to parallelise.
- The functions must not modify their `const` arguments. Make a copy of the value if you feel the need to make changes to it.
- Even if you do not manage to implement all of the functions, you can still answer the questions in section 2 with the functions you did manage to implement.
- **Make sure to compile your code with optimisations enabled when benchmarking.**

The specific functions you must implement follow below.

1.1 Transposition

Implement `transpose()`

The function `transpose()` must transpose a matrix using the standard algorithm as described in pseudocode:

```
for i in 0 ... n-1:
  for j in 0 ... m-1:
    B[j,i] = A[i,j]
```

Implement `transpose_parallel()`

The function `transpose_parallel()` must use the same algorithm as `transpose()`, but should be parallelised with OpenMP.

Implement `transpose_blocked()`

The function `transpose()` must transpose a matrix using an algorithm described by the following pseudocode:

```
for ii in 0, T ... n-1:
  for jj in 0, T ... m-1:
    for i = ii ... ii+T-1:
      for j = jj ... jj+T-1:
        B[j,i] = A[i,j]
```

The notation `ii in 0,T..n-1` means that `ii` is increased by `T` after every iteration. For simplicity, you need only handle the case where `n` and `m` are divisible by `T`. `T` is called a *tile size* and can be chosen arbitrarily, but its value will affect the performance of the function. Later you will be asked to try various values of `T` and observe the resulting performance.

Implement `transpose_blocked_parallel()`

The function `transpose_blocked_parallel()` must use the same algorithm as `transpose_blocked()`, but should be parallelised with OpenMP.

1.2 Matrix multiplication

Implement `matmul()`

The function `matmul()` must multiply two matrices using the standard algorithm, again as pseudocode:

```
for i in 0 .. n-1:
  for j in 0 .. k-1:
    acc = 0
    for p in 0 ... m-1:
      acc += A[i,p] * B[p,j]
    C[i,j] = acc
```

Implement `matmul_locality()`

The function `matmul_locality()` must multiply two matrices using the following pseudocode algorithm:

```
...initialize C to all zeroes
for i in 0 .. n-1:
  for p in 0 .. m-1:
    a = A[i,p]
    for j in 0 ... k-1:
      C[i,j] += a * B[p,j]
```

Implement `matmul_transpose()`

The function `matmul_transpose()` must multiply two matrices A, B using the following algorithm:

1. Transpose B to obtain B^T.
2. Perform a conventional matrix multiplication (as in `matmul()`), but where you are traversing the rows of B^T instead of the columns of B.

Use your fastest *non-parallel* transposition function.

Implement `matmul_parallel()`

The function `matmul_parallel()` must use the same algorithm as `matmul()`, but parallelised with OpenMP.

Implement `matmul_locality_parallel()`

The function `matmul_locality_parallel()` must use the same algorithm as `matmul_locality()`, but parallelised with OpenMP.

Implement `matmul_transpose_parallel()`

The function `matmul_transpose_parallel()` must use the same algorithm as `matmul_transpose()`, but parallelised with OpenMP. Use your fastest *parallel* transposition function.

2 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, 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:

a) Show the runtimes of all your transposition functions for various input sizes. Use only a single value of `T` (the best one you can find). Explain the performance differences.

b) Show the runtime of `transpose_blocked()` for various values of `T`. Why does `T` affect performance?

c) Show the *speedup* of your best parallel transposition function compared to your best sequential transposition function. Show how the speedup changes with the number of threads used. Explain how you chose the workload (i.e. input sizes) for these measurements. Does your implementation show strong or weak scaling?

d) Show the runtimes of all your matrix multiplication functions for various input sizes.

e) Explain the reason for the the performance differences (if any) between `matmul()`, `matmul_locality()`, and `matmul_transpose()`.

f) Using all available cores/threads, show the speedup of your parallel matrix multiplication functions compared to their corresponding sequential functions (i.e. `matmul_parallel()` versus `matmul()`, `matmul_locality_parallel()` versus `matmul_locality()`, and `matmul_transpose_parallel()` versus `matmul_transpose()`).

g) How did you decide which loops to parallelise in the matrix multiplications functions? Which OpenMP clauses did you use?

h) Does `matmul_locality_parallel()` exploit as much parallelism as `matmul_transpose_parallel()`? If there is a difference, is the difference likely to matter in practice?

i) Show the *speedup* of your best parallel matrix multiplication function compared to your best sequential matrix multiplication function. Show how the speedup changes with the number of threads used. Explain how you chose the workload (i.e. input sizes) for these measurements. Does your implementation show strong or weak scaling?

Advice:

- It is up to you to decide on proper workloads (i.e. input sizes) that can provide good answers to these questions. 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.
- All else being equal, **a short report is a good report**.

3 The code handout

`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.

`benchmark.c`: Example code for how to instrument and benchmark your functions.

`Makefile`: You may want to modify the `CFLAGS` for debugging and benchmarking. You are also free to add targets for new benchmark programs.

Feel free to add more programs for benchmarking.