HPPS Re-exam

April 19—22, 2022

Preamble

This document consists of 6 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

LU decomposition factors a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U.

$$A = LU$$

For example:

$$A = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 4 & 3 \\ 0 & -1.5 \end{bmatrix}$$

Your overall task is to implement LU matrix decomposition in C, analyse their performance both empirically and analytically, and optimise them through blocking 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 Specific tasks

Implement `lu_decompose()`

The function `lu_decompose()` must decompose a nxn matrix A into lower triangular matrix L and an upper triangular matrix U using the doolittle algorithm as described in pseudocode:

```
for i in 0 .. n-1:
 L[i, i] = 1
 for j in i .. n-1:
   U[j, i] = A[j, i]
   for k in 0 .. i-1:
     U[j, i] = U[j, i] - U[j, k] * L[k, i];
 for j in i+1 .. n-1:
   L[i, j] = A[i, j] / U[i, i]
   for k in 0 .. i-1:
     L[i, j] = L[i, j] - ((U[i, k] * L[k, j]) / U[i, i])
```

Implement `lu_decompose_parallel()`

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

Implement `lu_decompose_blocked()`

The function `lu_decompose_blocked()` must decompose a nxn matrix A into lower triangular matrix L and an upper triangular matrix U using the blocked doolittle algorithm with blocks of size T, as described in pseudocode:

```
U = A

int T = 2;    // size of blocks
c = n/T;  // count of blocks

for b in 0 .. (n/T)-1:
 for k in 0 .. T-1:
   L[k, k] = 1
   for i in k+1 .. (n-T*b)-1:
     L[k, i] = U[k, i] / U[k, k]
     U[k, i] = 0

     for j in k+1 .. (n-T*b)-1:
       L[j, i] = 0
       U[j, i] -= L[k, i] * U[j, k]
```

You may assume that the size of the input array is divisible by T.

Implement `lu_decompose_parallel_blocked()`

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

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) Do you believe that your functions produce correct LU decompositions? On what do you base your belief?

b) Show the runtimes of all your LU decomposition functions for various input sizes. Explain any performance differences.

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

d) Explain the reason for the the performance differences (if any) between `lu_decompose()`, and `lu_decompose_blocked()`.

e) Using all available cores/threads, show the speedup of your parallel LU decomposition functions compared to their corresponding sequential functions (i.e. `lu_decompose()` versus `lu_decompose_parallel()`, `lu_decompose_blocked()` versus `lu_decompose_parallel_blocked()`).

f) How did you decide which loops to parallelise? Which OpenMP clauses did you use?

g) What difficulties did you have in parallelising these algorithms? Describe what you tried that did not work, and what limitations these algorithms have for parallelisation.

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 or testing.