OpenMP exercise: LU factorization with tasks

1 The LU factorization by block columns

This programming assignment consists in developing two different parallelizations of the matrix $PA = LU$ factorization by block columns. Assuming that the matrix is of size NB block-columns, this operation can be roughly described with the following pseudo-code:

```
for(i=0; i<info.NB; i++){
  /* Do the panel */
  panel(A[i], i, info);
  for(j=i+1; j<info.NB; j++){
    /* Do all the corresponding updates */
    update(A[i], A[j], i, j, info);
  }
}
backperm(A, info);
```

Note that the result of the factorization (i.e., the L and U factors) overwrites the input matrix A. The steps of this algorithm are depicted in Figure 1.

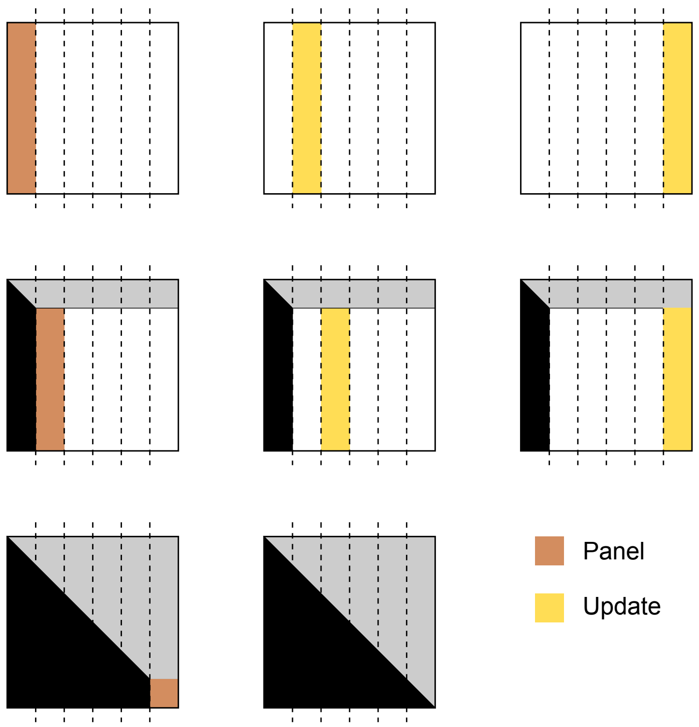

Figure 1: The steps of an LU factorization by block-columns

The routines in the algorithm above are defined as such:

- `panel(A[i], i, info)`: this routine computes the reduction (unblocked, inefficient LU factorization) of a block column `i`: $P_i * A(:, i) = L_i * U_i$. This routine **reads and writes block-column `i`, i.e. `A[i]`**. The `i` scalar and `info` data structure are only used in **read** mode.
- `update(A[i], A[j], i, j, info)`: this operation applies to block-column `j` the transformation computed by the panel operation on block column `A[i]`. This routine **reads block-columns `A[i]` and `A[j]` and modifies block-column `A[j]`**. The `i` and `j` scalars and `info` data structure are only used in **read** mode.
- `backperm(A, info)`: this routine applies all the P_i permutations computed in the factorization main loop to the L factor.

2 Package content

In the `lu_tasks` directory you will find the following files:

- `lu_seq.c`: this file contains a sequential version of the LU factorization by block-columns. This is, essentially, the same as the pseudo-code reported above. This file should not be modified and only serves as a reference to compare with the two parallel versions to be developed.
- `lu_par_tasks.c`: this file has to be modified to achieve the parallelization. At the beginning this file is an exact copy of the `lu_seq.c` file and the parallelization is obtained by adding OpenMP directives.
- `main.c`: this file contains a main program which creates and initializes the matrix and the calls the sequential and the two parallel versions of the factorization. For each of them the program also computed the execution time, the performance rate in Gflops/s (billion of floating-point operations per second) and checks the correctness of the factorization.
- `aux.c`, `auxf.f90`, `common.h`, `kernels.c`, `trace.c` and `trace.h`: this are auxiliary files and should not be modified.

The main program can be compiled by typing the `make` command; this will generate an executable file `main` that can be run as such:

```
./main B NB
```

where `B` is the size of a block-column and `NB` is the number of block-columns the matrix is made of.

By compiling with the command `make main_dbg` instead, the resulting program will also print additional information showing the order in which panel and update operations are executed and which thread executed each of them. This can be very useful to verify that the operations are executed in the correct order.

3 Assignment

- ⌨ The objective of this exercise is to parallelize the *LU* factorization code presented above. **This as to be done using the OpenMP `task` directive**. At the beginning the `lu_par_tasks` is a copy of the `lu_seq` routine; it has to be modified to achieve the parallelization.
- ✎ Report the execution times for the implemented parallel version with 1, 2 and 4 threads and for different array sizes. Analyze and comment on your results: is the achieved speedup reasonable or not? Report your answer in the `responses.txt` file.

Advice

- For verifying the correctness of your code, choose moderate values for B and NB (for example B=20 and NB=5). For analyzing the performance and scalability of your parallelization choose bigger values (for example, B=100 and NB=40).
- When using the OpenMP `task` directive, two ways of handling dependencies among tasks consist in using either the `taskwait` directive or the `depend` option of the `task` directive.