OpenMP exercise: LU factorization

December 1, 2014

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<NB; i++){
   panel(A,i);
   for(j=i+1; j<NB; j++){
      update(A,i,j);
   }
}
backperm(A);
```

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

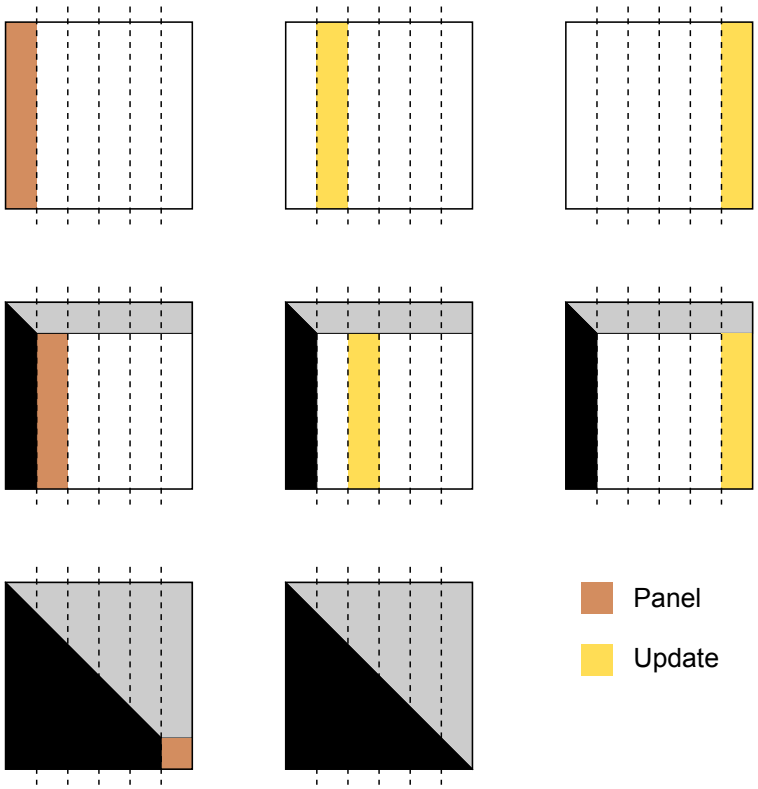

Figure 1: The steps of an LU factorization by block-columns

The routines in the algorithm above are defined as such:

- `panel(A,i)`: 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`**.
- `update(A,i,j)`: this operation applies to block-column `j` the transformation computed by the `panel(i)` operation. This routine **reads block-columns `i` and `j` and modifies block-column `j`**.
- `backperm(A)`: this routine applies all the P_i permutations computed in the factorization main loop to the L factor.

The package contains 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_loop.c`: this file has to be modified to achieve the first parallelization described in Part 1. At the beginning this file is an exact copy of the `lu_seq.c` file and the parallelization is obtained by adding OpenMP directives.
- `lu_par_dag.c`: this file contains a more advanced version of the algorithm presented above especially designed to achieve a better parallelization. This is the subject of Part 2.
- `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. 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). The number of threads can be controlled through the `OMP_NUM_THREADS` environment variable, like this

```
export OMP_NUM_THREADS=4
```

for setting the number of threads to 4 (for shells other than bash you should use `setenv OMP_NUM_THREADS 4`).

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.

2 Part 1: simple inner loop parallelization

Assume that the `backperm` operation can be ignored and choose a small example matrix of size 3 or 4 block-columns: can you draw a graph of dependencies for the algorithm above?

Based on the graph of dependencies you drew, can you identify which operations can be performed independently and in parallel?

- Modify the `lu_par_loop.c` file to achieve a parallelization using the OpenMP `#pragma omp parallel for` construct. Compile the main program and run it. Verify the correctness (the printed residual should be smaller than 10^{-10}) and analyze the performance and scalability of your code using 2 and 4 threads. Is the parallel factorization faster than the sequential?
- Note that creating and destroying a parallel region has a cost and should be avoided within a loop. Modify your parallel code in order to create the parallel region only once and then execute multiple parallel loops in it. This can be done by splitting the `#pragma omp parallel for` into the two constructs `#pragma omp parallel` and `#pragma omp for` and placing these two in the right position. Pay special attention to the synchronization between threads. Compile the main program and run it. Verify the correctness (the printed residual should be smaller than 10^{-10}) and analyze the performance and scalability of your code using 2 and 4 threads. Is the parallel factorization faster than the sequential?

Running the main program also generates trace files. A trace is an image showing which operations are executed by the threads in time as in Figure 2. Each row shows the operations executed by one thread in time; within each row, a rectangle shows an operation (brown is for panel, yellow is for update and green is for backperm) and its length is proportional to the operation execution time.

Figure 2: A part of an execution trace with 4 threads

Open the trace file `trace_par_loop.svg` of your parallel code with the `inkscape` or `eog` programs and analyze it. Are all the threads working? is the work fairly distributed among the threads? can you identify inefficiencies?

3 Part 2: A complex, efficient DAG based parallelization

Analyzing the traces produced by the parallel code developed in Part 1, you should remark that there are empty gaps when panel operations are being executed by one thread. White spaces in the traces mean that some threads are idle (i.e., not working) waiting for some event to happen (in this case, the execution of the corresponding panel operation). Therefore, white spaces represent inefficiencies and should be removed as much as possible.

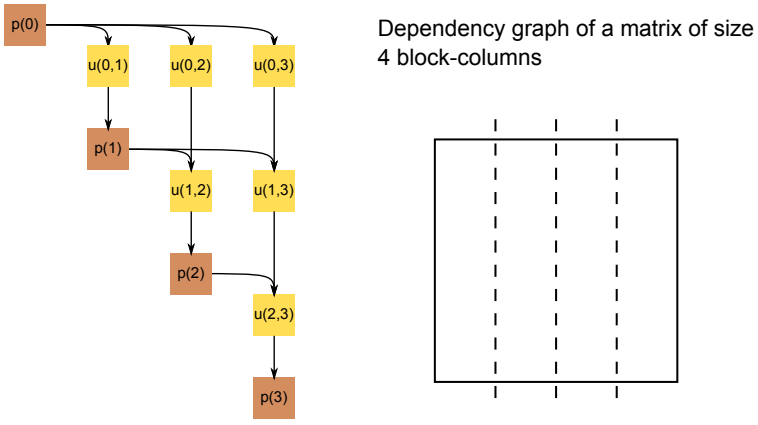

Figure 3: The dependency graph for a matrix of size 4 block-columns.

Figure 3 shows the dependency graph of the LU factorization for a matrix of size 4 block-columns. Looking at this graph of dependencies, is there any way we can remove some of the white gaps that appear in the

execution traces of Part 1? Note that, in the parallel code developed in Part1, the execution of the operation `panel(A,i)` is started only after all of the updates `update(A,i-1,i), update(A,i-1,i+1),...,update(A,i-1,NB)` are completed. Is this really necessary? Can we start the execution of `panel(A,i)` immediately after `update(A,i-1,i)` or do we have to wait for all the other updates to finish?

The answer is that `panel(A,i)` can be started as soon as `update(A,i-1,i)` is done. Its execution can thus be overlapped with the execution the other updates related to previous panels. This can be done either statically, with the well known technique of *lookahead* or dynamically, as described below.

3.1 Parallel, dynamic LU factorization

Imagine that we have a representation of the dependency graph in Figure 3 that we use to keep track of the already executed operations during the LU factorization. At any moment, by looking at this graph, we are able to identify operations that are ready to be executed, i.e., operations that depend on other operations that have been completed. Imagine, for example, that in the graph of Figure 3, operations `panel(A,0)` `update(A,0,1)` and `update(A,0,2)` have been executed; this is shown in Figure 4 with gray nodes. At this moment, it is possible to execute either `panel(A,1)` or `update(A,0,3)` marked with a thick red border.

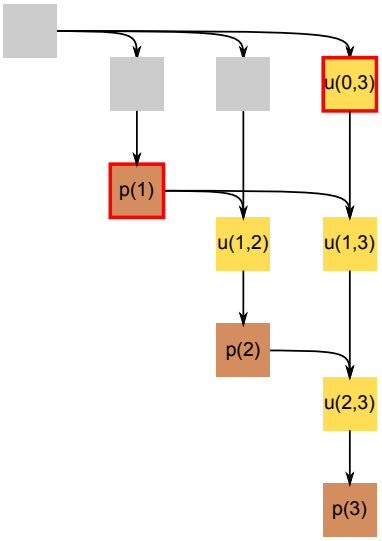

Figure 4: The dependency graph allows to identify operations that are ready to be executed.

The main idea of the parallel, dynamic LU factorization is that every thread continuously looks at the dependency graph and, as soon as it finds a ready operation, it executes the operation and consequently updates the dependency graph. This goes on until all the operations have been executed.

In the file `lu_par_dag.c` you will find a sequential version of this algorithm. You have to make it parallel by conveniently adding OpenMP directives. Keep in mind that this takes as little as five lines of code so the solution is quite simple (it only requires some preliminary thinking).

The dependency graph and the state of the factorization are represented by a simple array of size NB (number of block-columns in the matrix) called *progress table* (`ptable`). At the beginning of the factorization, all the entries of this array are equal to -1. During the factorization the progress table is updated as such:

- if the operation `panel(A,i)` is executed, then `ptable[i]` is set to `i`;
- if the operation `update(A,i,j)` is executed, then `ptable[j]` is set to `i`.

The progress table associated with the partial factorization in Figure 4 is thus

```
ptable = {0, 0, 0, -1}
```

The following rules can be defined:

- a panel operation `panel(A,i)` can be executed only if `ptable[i]==i-1`; in the example of Figure 4, it is thus possible to perform `panel(A,1)` because `ptable[1]==0`.
- an update operation `update(A,i,j)` can be executed only if `ptable[j]==i-1` and `ptable[i]==i`; in the example of Figure 4, it is thus possible to perform `update(A,0,3)` because `ptable[3]==-1` and `ptable[0]==0`.
- if `ptable[NB]==NB`, then the factorization is finished.

In the file `lu_par_dag.c` you will find two routines:

- `task_fetch`: this routine looks at the progress table and returns the identifier of a ready-to-execute operation (if it exists) according to the rules defined above.
- `lu_par_dag`: this is the main factorization driver.

Parallelize the code, compile and run it. Verify that the result is correct and then analyze the performance and scalability with 2 and 4 threads. Is the new parallel version faster than the one developed in Part 1? It should. Analyze the traces in the `trace_par_dag.svg` (open with `inkscape` or `eog`). Can you see the difference between these traces and the previous ones? have the white gaps disappeared?

3.2 Improving the scheduling

One question that was left unanswered in the previous part is: in case there are multiple operations ready to be executed, which one should I choose?

Do you believe this is an important question? why? Think about the concept of *critical path* in the graph of dependencies. Because the graph of dependencies of the LU factorization only has one entry point and one exit point, the critical path can simply be defined as the longest path connecting the entry and exit points. Can you identify the critical path in the graph of Figure 3? can you identify a type of operations that always lie along the critical path? Do you believe that these operations should be executed with higher or lower priority?

Modify the `task_fetch` routine accordingly. Is the resulting code faster? It should.