**Q: What's the difference between the open-source and close-source NICSLU versions?**
A: The open-source version is old, has bugs, and we will never update it. It can be used for study purpose. The new version improves the performance up to 3X. Major improvements include new ordering algorithms and new numerical kernels. Another major difference is the thread synchronization method. The old version uses busy waiting so slaves always consume CPU when they have no work to do, but the new version uses blocked waiting by default.

**Q: How to do parallel LU factorization?**
A: After `NicsLU_Analyze`, call `NicsLU_CreateThreads` (only once), and then you can call `NicsLU_Factorize`, `NicsLU_ReFactorize` or `NicsLU_FactorizeMatrix` with a proper number of threads specified in the last argument to execute parallel factorization. The created threads will not exit unless `NicsLU_DestroyThreads`, `NicsLU_Free` or `NicsLU_Analyze` is called.

**Q: I specified the number of threads when calling factorization functions, but the CPU usage was always 1 core. It seems that NICSLU did not perform parallel factorization.**
A: `NicsLU_Factorize`, `NicsLU_ReFactorize` and `NicsLU_FactorizeMatrix` can automatically determine the number of threads to achieve the optimal performance. This feature can be turned off by setting `cfg[13]=0` (default is `cfg[13]=1`).

**Q: What's the difference between `NicsLU_Factorize`, `NicsLU_ReFactorize` and `NicsLU_FactorizeMatrix`?**
A: `NicsLU_Factorize` always performs pivoting, and `NicsLU_ReFactorize` doesn't perform any pivoting. `NicsLU_ReFactorize` must be called after `NicsLU_Factorize` is called at least once, otherwise `NicsLU_ReFactorize` just calls `NicsLU_Factorize` for the first time. `NicsLU_FactorizeMatrix` selects to call `NicsLU_Factorize` or `NicsLU_ReFactorize` by a heuristic method. For `NicsLU_FactorizeMatrix` with identical matrix data, the first call invokes `NicsLU_Factorize`, and subsequent calls invoke `NicsLU_ReFactorize`. `NicsLU_FactorizeMatrix` is recommended in practice.

**Q: What's the difference between `NicsLU_Solve` and `NicsLU_SolveAndRefine`?**
A: `NicsLU_Solve` doesn't perform refinement. `NicsLU_SolveAndRefine` automatically decides if refinement will be called by a heuristic method. `NicsLU_Solve` is recommended in practice.

**Q: How to do benchmarking?**
A: You just need to measure the WALL time of `NicsLU_Analyze` for analysis time, `NicsLU_Factorize`, `NicsLU_ReFactorize` or `NicsLU_FactorizeMatrix` for factorization time, and `NicsLU_Solve` or `NicsLU_SolveMatrix` for solving time.
If you are simulating a SPICE-like circuit simulation application, please measure the runtime of the 2nd `NicsLU_Factorize` for factorization time and 2nd `NicsLU_ReFactorize` for re-factorization time. If you are using `NicsLU_FactorizeMatrix`, measure the runtime of the 3rd `NicsLU_FactorizeMatrix` for re-factorization time. The main reason of giving up the very first iterations is that the 1st factorization or re-factorization has some one-time initialization

work so it will cause a little longer time than subsequent factorizations or re-factorizations.

**Q: The solution is completely wrong for any test case. What's the reason?**
A: Please note that NICSLU uses row-based compressed format (CSR), not column-based format. If your matrix is stored in column-based order, set the proper `_matrix_type_t` argument when calling `NicsLU_Analyze`, then NICSLU will solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$.

**Q: Do I need to change the platform-related configurations of NICSLU?**
A: All the platform-related configurations have been carefully adjusted to achieve a good performance on latest CPUs. Generally speaking, if you just use the default configurations, you will get a reasonable performance on most modern platforms. However, we only tested NICSLU on limited platforms. It is possible to get a poor performance on some platforms. In this case, you will need to adjust the platform-related configurations.

**Q: Do I need to change the matrix-related configurations of NICSLU?**
A: Possibly yes for some ill-conditioned matrices but no for general cases. Some features like scaling may have a big impact on the performance for specific matrices. This is mainly because of numerical stability. Keeping high numerical stability during factorization will help reduce off-diagonal pivots so factorization time is also reduced. Generally speaking, if a matrix is near singular, you can try to enable the scaling options and see the performance difference.

**Q: `NicsLU_Analyze` requires matrix values, but I don't have them when calling `NicsLU_Analyze` (values are available only when calling LU factorization). How can I do?**
A: You can simply give a `NULL` to the argument `ax` when calling `NicsLU_Analyze`, and then NICSLU will do a zero-free permutation purely based on the symbolic pattern. However, this will disable a feature of static pivoting, which is expected to significantly improve the performance. We provide another solution to solve this problem. Put `NicsLU_Analyze` into your wrapper of factorization, and let `NicsLU_Analyze` be called before the first factorization. Set a flag to indicate whether analysis is called (you should ensure that analysis is called only once), like this:

```
int my_lu_factor(...)
{
    static int analysis_done = 0;

    if (!analysis_done)
    {
        int err = NicsLU_Analyze(...);
        if (__FAIL(err)) ...;
        analysis_done = 1;
    }

    //do factorization here
    int err = NicsLU_FactorizeMatrix(...);
}
```