HYLU User Guide

(Version 20260126)

Xiaoming Chen (chenxiaoming@ict.ac.cn)

HYLU (Hybrid Parallel Sparse LU Factorization) is a general-purpose parallel solver for efficiently solving sparse linear systems ($\mathbf{Ax} = \mathbf{b}$) on multi-core shared-memory architectures. It employs an innovative parallel up-looking LU factorization algorithm, which dynamically adapts to varying matrix sparsity patterns by leveraging hybrid numerical kernels. The numerical stability is maintained by combining static pivoting, dynamic diagonal supernode pivoting, dynamic scaling, and iterative refinement.

HYLU delivers high-performance LU factorization for large-scale sparse linear systems from multiple engineering and scientific domains, including circuit simulation, power systems, computational fluid dynamics, electromagnetics, and structural analysis. The solver efficiently handles linear systems arising from finite element analysis, 2D/3D modeling, and optimization problems.

HYLU is implemented in `C` and offers seamless integration with `C/C++` applications. The multi-threading of *HYLU* is implemented based on operating system's native threading interface (OpenMP is NOT needed).

1. Functions

HYLU offers 7 groups of functions for solving sparse linear systems. Each group provides both 32-bit and 64-bit integer versions for both real number and complex number systems. The following table lists the format of the *HYLU* function names and their corresponding versions. In the following text, the function names of the 32-bit integer/real number version will be used to explain the functionality of *HYLU*. In the 32-bit integer version, only the input matrix indices use 32-bit integers, while internal LU factor data structures maintain 64-bit integers for scalability. For detailed function prototypes and argument descriptions of the *HYLU* functions, please refer to `<hylu.h>`.

Function name	**Version**	**Library**
`HYLU_XXXX`	32-bit integer/real number	libhylu.so|hylu.dll
`HYLU_L_XXXX`	64-bit integer/real number	libhylu_l.so|hylu_l.dll
`HYLU_C_XXXX`	32-bit integer/complex number	libhylu_c.so|hylu_c.dll
`HYLU_CL_XXXX`	64-bit integer/complex number	libhylu_cl.so|hylu_cl.dll

(1) `HYLU_CreateSolver`

This function creates a solver instance, retrieves a pointer to the parameter array, and spawns working threads for parallel execution. Input parameters are initialized to their default values during this process. The number of working threads should not exceed the number of idle cores.

(2) `HYLU_DestroySolver`

This function frees all allocated memory, terminates the working threads, and destroys the solver instance.

(3) `HYLU_Analyze`

This function performs preprocessing steps including static pivoting, matrix reordering, and symbolic factorization. The matrix format is **compressed sparse row (CSR)**, where column indexes in each row are **NOT required to be sorted**. If the input matrix is symmetric, the CSR representation must still contain the **full** set of matrix entries.

The second argument, '`repeat`', specifies whether linear systems will be solved repeatedly with an identical matrix structure. This is common in applications such as circuit simulation. In such a scenario, **preprocessing is needed only once**. When enabled, this argument impacts the default values of certain parameters. Specifically, in the repeated solving scenario, *HYLU* prioritizes minimizing nonzeros in the LU factors, which may increase preprocessing time but improves subsequent factorization efficiency.

It is optional to provide the matrix values (the argument '`ax`') to preprocessing, but providing values is strongly recommended. Without matrix values, static pivoting and static scaling cannot be carried out.

(4) `HYLU_Analyze2`

This function provides similar functionality to `HYLU_Analyze` with user-provided ordering. The user-provided ordering must ensure **a zero-free diagonal** in the permuted matrix. Static pivoting and static scaling are disabled.

(5) `HYLU_Factorize`

This function computes the numerical LU factorization of the preprocessed matrix, decomposing it into a lower triangular matrix (**L**) and an upper triangular matrix (**U**).

Dynamic diagonal supernode pivoting is performed during this process. If no suitable pivot can be found in a row, a pivot perturbation strategy is applied.

(6) `HYLU_Solve`

This function computes the solution vector by performing forward ($\mathbf{Ly} = \mathbf{b}$) and backward ($\mathbf{Ux}=\mathbf{y}$) substitutions using the computed LU factors.

If pivot perturbation has occurred in the last numerical LU factorization, iterative refinement is automatically performed to correct the solution.

(7) `HYLU_MSolve`

This function solves multiple linear systems that share the same coefficient matrix. If working threads have been created, it performs parallel solves across different right-hand-side vectors.

2. Parameters

The following table describes the parameter array. Input parameters are initialized to their default values (marked with an asterisk *) upon calling `HYLU_CreateSolver`. All modifications (if needed) to input parameters should be completed prior to invoking `HYLU_Analyze` or `HYLU_Analyze2`.

Parameter	Description	
`parm[0]`: output	Version.	
`parm[1]`: **input**	Timer control. When enabled, `parm[7]` records the wall time of the last function call.	
	0*	Disabled.
	>0	High-precision timer (microsecond precision).
	<0	Low-precision timer (millisecond precision).
`parm[2]`: **input**	Ordering method for fill-in reduction.	
	0*	The ordering method is automatically decided according to the matrix dimension and the '`repeat`' argument of `HYLU_Analyze`.
	1	Approximate minimum degree.
	2	Approximate minimum degree variant.
	3	Nested dissection method 1.
	4	Nested dissection method 2.
	5	Best of 1 and 2.
	6	Best of 3 and 4.
	7	Best of 1-4.
`parm[3]`: **input**	Ordering method switch point. For nested dissection, once a partition size is smaller than `parm[3]`, it switches to a constrained approximate minimum degree method. Smaller `parm[3]` values generally yield better ordering results of nested dissection at the cost of increased ordering time.	
	0*	Automatic control. The value is determined according to the '`repeat`' argument of `HYLU_Analyze`.
	>=64	Allowable range.
`parm[4]`: output	Selected ordering method (1-4), reported by `HYLU_Analyze`.	
`parm[5]`: **input**	Minimum number of columns of a supernode. A supernode must contain at least `parm[5]` columns.	
	32*	Default value.
	>=8	Allowable range.
`parm[6]`: **input**	Maximum number of rows of a supernode. A supernode may contain at most `parm[6]` rows. A larger supernode is split into multiple supernodes.	
	0*	Automatic control. The value is determined by whether working threads have been spawned.
	>=8	Allowable range.
`parm[7]`: output	Wall time (in **microseconds**) of last function call.	

`parm[8]`: output	Number of off-diagonal pivots, reported by `HYLU_Factorize`.	
`parm[9]`: output	Number of supernodes, reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[10]`: **input**	Pivot perturbation coefficient. Zero or small pivots will be replaced by $\text{sign(pivot)} \times 10^{\text{parm[10]}} \times \lVert\mathbf{A}\rVert_\infty$.	
	-15*	Default value.
	<0	Allowable range.
`parm[11]`: output	Number of perturbed pivots, reported by `HYLU_Factorize`.	
`parm[12]`: output	Current memory usage (in bytes), or required memory size (in bytes) when -4 is returned, reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[13]`: output	Maximum memory usage (in bytes), reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[14]`: output	Thread numbers stored in 3 shorts. Use the following method to get them: `const short *threads = (short *)&parm[14];` `threads[0]`: number of physical cores (may be incorrect). `threads[1]`: number of logical cores. `threads[2]`: number of created working threads.	
`parm[15]`: **input**	Maximum number of refinement iterations.	
	0*	Whether to perform iterative refinement and the iteration count are determined automatically.
	>0	If *HYLU* decides to perform iterative refinement, execute `parm[15]` iterations.
	<0	Execute `-parm[15]` iterations.
`parm[16]`: output	Number of refinement iterations performed, reported by `HYLU_Solve`.	
`parm[17]`: output	Number of nonzeros in **L** (including diagonal), reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[18]`: output	Number of nonzeros in **U** (excluding diagonal), reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[19]`: output	Number of floating-point operations of factorization (excluding scaling), reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[20]`: output	Number of floating-point operations of solving (excluding scaling), reported by `HYLU_Analyze` or `HYLU_Analyze2`.	
`parm[21]`: **input**	Matrix scaling method.	
	>0*	Dynamic matrix value scaling.
	0	Disabled.
	<0	Static matrix value scaling.
`parm[22]`: **input**	Symbolic factorization method. Symmetric symbolic factorization helps reduce preprocessing time, but increases fill-ins for structurally unsymmetric matrices.	
	>0	Unsymmetric symbolic factorization.
	<0	Symmetric symbolic factorization.
	0*	Automatic control.

`parm[23]`: **input**	Consistent symbolic results between sequential and parallel executions.	
	!=0*	Enabled.
	0	Disabled.

3. Return Values

Every function of *HYLU* returns an integer to indicate the error code. This table describes the meanings of the return codes.

Return value	Description
0	Successful.
-1	Invalid instance handle.
-2	Function argument error (e.g., negative matrix dimension, `NULL` pointer).
-3	Invalid input matrix (e.g., matrix indices error).
-4	Out of memory. `parm[12]` will return the needed memory size.
-5	Structurally singular.
-6	Numerically singular.
-7	Threads error.
-8	Calling procedure error.
-9	Integer overflow, please use the `HYLU_(C)L_*` functions.
-10	Internal error.

4. Using *HYLU* Library

HYLU is provided as x64 dynamic-link libraries. **Running *HYLU* requires an x64 processor with FMA and AVX2 instruction sets support.** Processors that support the AVX-512 instruction set are recommended.

Please follow the platform-specific instructions below to link the *HYLU* library.

- **Linux**

When linking *HYLU* to the final executable, use the compiler flags `-L<library_path> -l<library_name>` to specify the location and name of the *HYLU* library.

Some system libraries may be required. It is often necessary to add `-lpthread`, `-lm`, and `-ldl` to the linker flags. On older Linux distributions, `-lrt` may also be needed. A typical linking command might look like: `gcc -o myprogram myprogram.c -L/path/to/hylu -lhylu -lpthread -lm -ldl`.

Before running the executable, ensure the system can locate the *HYLU* shared library. The typical method is to set the `LD_LIBRARY_PATH` environment variable: `export LD_LIBRARY_PATH=<library_path>:$LD_LIBRARY_PATH`.

- **Windows (using Visual Studio)**

Add the directive `#pragma comment(lib, "<library_name>.lib")` anywhere in the source code (commonly near the top). The corresponding `.lib` import library file is needed during the linking stage.

Only the `.dll` file is required to run the executable. The recommended and simplest method is to place the corresponding `.dll` file in the same directory as the executable.

5. Notes

(1) **User must not free the memory of the parameter array**, which is managed by *HYLU*.

(2) By default, *HYLU* operates in **row-major order**. For matrices stored in column-major order, set the argument '`transpose`' of `HYLU_Solve` to `true` to solve transposed systems ($\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$). However, the parallel scalability of column-wise solve is not as good as that of row-wise solve.

(3) If the input matrix is symmetric, the CSR representation must still contain the full set of matrix entries.

(4) Please ensure that the number of working threads created by *HYLU* is less than or equal to the number of idle cores, otherwise the performance of *HYLU* will degrade severely.

(5) Due to the limited pivoting range, for some linear systems *HYLU* may produce inaccurate solutions. In such cases, consider setting `parm[23]` to 0, adjusting `parm[6]` to a larger value, and utilizing sequential factorization. `parm[10]` and `parm[21]` may also affect the accuracy of results.

(6) Memory leaks have been observed in *HYLU* libraries linked with Intel MKL BLAS. These leaks are attributed to the MKL BLAS functions themselves.

(7) Setting `parm[23]!=0` ensures consistent symbolic results between sequential and parallel executions, but numerical results may be slightly different. When `parm[23]=0` and multithreading is used, the ordering results from parallel nested dissection are non-deterministic.