HYLU User Guide

(Version 20250903)

Xiaoming Chen (chenxiaoming@ict.ac.cn)

HYLU (Hybrid Parallel Sparse LU Factorization) is a general-purpose parallel solver designed for efficiently solving sparse linear systems (**Ax** = **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 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 5 user-friendly functions for solving sparse linear systems. The library provides both 32-bit and 64-bit integer versions (denoted by the `_L` suffix for 64-bit). 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 of *HYLU* and argument descriptions of the functions, please refer to `<hylu.h>`.

(1) `HYLU(_L)_CreateSolver`

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

(2) `HYLU(_L)_DestroySolver`

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

(3) `HYLU(_L)_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**.

The second argument, '`repeat`', specifies whether the linear system will be solved repeatedly with an identical matrix structure. This scenario frequently occurs in practical 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 cannot be carried out.

(4) `HYLU(_L)_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, a pivot perturbation strategy will be applied.

(5) `HYLU(_L)_Solve`

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

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

2. Parameters

This table describes all individual components of the parameter array. Input parameters are initialized to their default values (marked with an asterisk *) upon calling `HYLU(_L)_CreateSolver`. All modifications (if needed) to input parameters should be completed prior to invoking `HYLU(_L)_Analyze`.

Parameter	Description	
`parm[0]`: output	Version.	
`parm[1]`: **input**	Timer control. When enabled, `parm[7]` will return 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(_L)_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 will switch to a constrained approximate minimum degree method. Smaller `parm[3]` values generally produce better ordering results of nested dissection, with	

	longer ordering time.	
	0*	Automatic control. The value will be determined according to the 'repeat' argument of HYLU(_L)_Analyze.
	>=64	Allowable range.
parm[4]: output	Selected ordering method (1-4), reported by HYLU(_L)_Analyze.	
parm[5]: **input**	Minimum number of columns of a supernode. A supernode should have at least parm[5] columns.	
	32*	Default value.
	>=8	Allowable range.
parm[6]: **input**	Maximum number of rows of a supernode. A supernode can have at most parm[6] rows. A larger supernode will be split into multiple supernodes.	
	0*	Automatic control. The value will be determined by whether worker 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(_L)_Factorize.	
parm[9]: output	Number of supernodes, reported by HYLU(_L)_Analyze.	
parm[10]: **input**	Pivot perturbation coefficient. Zero or small pivot will be replaced by $\mathrm{sign(pivot)} \times 10^{\mathrm{parm[10]}} \times \|\mathbf{A}\|_\infty$.	
	-15*	Default value.
	<0	Allowable range.
parm[11]: output	Number of perturbed pivots, reported by HYLU(_L)_Factorize.	
parm[12]: output	Current memory usage (in bytes), or needed memory size (in bytes) when -4 is returned, reported by HYLU(_L)_Analyze.	
parm[13]: output	Maximum memory usage (in bytes), reported by HYLU(_L)_Analyze.	
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 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(_L)_Solve.	
parm[17]: output	Number of nonzeros in **L** (including diagonal), reported by HYLU(_L)_Analyze.	

`parm[18]`: output	Number of nonzeros in **U** (excluding diagonal), reported by `HYLU(_L)_Analyze`.	
`parm[19]`: output	Number of floating-point operations of factorization (excluding scaling), reported by `HYLU(_L)_Analyze`.	
`parm[20]`: output	Number of floating-point operations of solving (excluding scaling), reported by `HYLU(_L)_Analyze`.	
`parm[21]`: **input**	Matrix scaling.	
	>0*	Dynamic matrix value scaling.
	0	Disabled.
	<0	Static matrix value scaling.
`parm[22]`: **input**	Symmetric symbolic factorization. Symmetric symbolic factorization helps reduce preprocessing time, but will increase fill-ins for structurally unsymmetric matrices.	
	0	Disabled.
	>0	Enabled.
	<0*	Automatic control.

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_L_*` functions.
-10	Internal error.

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

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

(5) The *HYLU* libraries which use MKL BLAS have memory leaks, which are caused by MKL BLAS functions.