Solvers User Guide

Version T-2022.03, March 2022

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Contents

Contents

About This Guide

This user guide provides information about the solvers that are available as part of Synopsys® TCAD software. These solvers can be used with the Synopsys Sentaurus™ Device, Sentaurus Interconnect, and Sentaurus Process tools.

For additional information, see:

- The TCAD Sentaurus release notes, available on the Synopsys SolvNetPlus support site (see Accessing SolvNetPlus)
- Documentation available on the SolvNetPlus support site

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier font`	Identifies text that is displayed on the screen or that the user must type. It identifies the names of files, directories, paths, parameters, keywords, and variables.
Italicized text	Used for emphasis, the titles of books and journals, and non-English words. It also identifies components of an equation or a formula, a placeholder, or an identifier.

Customer Support

Customer support is available through the Synopsys SolvNetPlus support site and by contacting the Synopsys support center.

Accessing SolvNetPlus

The SolvNetPlus support site includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The site also gives you access to a wide range of Synopsys online services, which include downloading software, viewing documentation, and entering a call to the Support Center.

To access the SolvNetPlus site:

1. Go to https://solvnetplus.synopsys.com.
2. Enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register.)

Contacting Synopsys Support

If you have problems, questions, or suggestions, you can contact Synopsys support in the following ways:

- Go to the Synopsys Global Support Centers site on www.synopsys.com. There you can find email addresses and telephone numbers for Synopsys support centers throughout the world.
- Go to either the Synopsys SolvNetPlus site or the Synopsys Global Support Centers site and open a case (Synopsys user name and password required).

Contacting Your Local TCAD Support Team Directly

Send an email message to:

- support-tcad-us@synopsys.com from within North America and South America
- support-tcad-eu@synopsys.com from within Europe
- support-tcad-ap@synopsys.com from within Asia Pacific (China, Taiwan, Singapore, Malaysia, India, Australia)
- support-tcad-kr@synopsys.com from Korea
- support-tcad-jp@synopsys.com from Japan

Acknowledgments

ILS was codeveloped by Integrated Systems Laboratory of ETH Zurich in the joint research project NUMERIK II with financial support by the Swiss funding agency CTI.

Part I: PARDISO

This part contains information about the direct linear solver PARDISO and is intended for users of Sentaurus Device, Sentaurus Process, and Sentaurus Interconnect:

Chapter 1, Using PARDISO provides background information on PARDISO

1

Using PARDISO

PARDISO is a high-performance, robust, and easy to use software package for solving large sparse symmetric or nonsymmetric systems of linear equations in parallel.

The rapid and widespread acceptance of shared-memory multiprocessors has created a demand for parallel semiconductor device and process simulation on such shared-memory multiprocessors.

PARDISO [1][2] can be used as a serial package, or in a shared-memory multiprocessor environment as an efficient, scalable, parallel, direct solver. PARDISO is tuned for general use in Sentaurus Device, Sentaurus Process, and Sentaurus Interconnect, which means user intervention is not necessary.

Algorithms

The process of obtaining a direct solution of a sparse system of linear equations of the form $Ax = b$ consists of the following important phases [3][4]:

- *Nonsymmetric matrix permutation and scaling* places large matrix entries on the diagonal and aims to maximize the elements on the diagonal of the matrix.

  This step greatly enhances the reliability and accuracy of the numeric factorization process. More details can be found in the literature [5][6][7].

- *Ordering* determines the permutation of the coefficient matrix A such that the factorization incurs low fill-in.

  The reordering strategy of PARDISO features state-of-the-art techniques, for example, multilevel recursive bisection from METIS [8] or minimum degree–based approaches [9][10] for the fill-in reduction. The nested dissection approach integrated in PARDISO is substantially better than the multiple minimum degree algorithm for large problem sizes. This applies especially to three-dimensional problems.

- *Numeric factorization* is the actual factorization step that performs arithmetic operations on the coefficient matrix A to produce the factors L and U such that $A = LU$. Complete block diagonal supernode pivoting allows for dynamic interchanges of columns and rows.

  PARDISO exploits the memory hierarchy of the architecture by using the clique structure of the elimination graph by supernode algorithms, thereby improving memory locality [11]. The numeric factorization algorithm of the package utilizes the supernode structure of the numeric factors L and U to reduce the number of memory references with Level 3 BLAS [12][13]. The result is a greatly increased, sequential, factorization performance.

  Furthermore, PARDISO uses an integrated, scalable, left-right-looking, supernode algorithm [14][15] for the parallel sparse numeric factorization on shared-memory multiprocessors. This left-right-looking supernode algorithm significantly reduces the communication rate for pipelining parallelism.

- *Solution of triangular systems* produces the solution by performing forward and backward eliminations.

The combination of block techniques, parallel processing, and global fill-in reduction methods for 3D semiconductor devices results in a significant improvement in computational performance.

Parallel Solution on Shared-Memory Multiprocessors

The use of vendor-optimized BLAS and LAPACK subroutines ensures high computational performance on a large scale of different computer architectures. The parallelization technique is based on OpenMP [16], which is an industrywide standard for directive-based parallel programming of shared-memory parallelization (SMP) systems. Most SMP vendors are committed to OpenMP, thereby making OpenMP programs portable across an increasing range of SMP platforms.

A parallel version of PARDISO is available on Red Hat Enterprise Linux (64-bit).

Multiple cores on machines that support hyperthreading are treated in the same way as multiple CPUs.

A sufficient process stack size is required for the proper execution of PARDISO. To check the UNIX stack size limit, in `csh`, enter the command:

```
limit
```

or, in `bash` or `sh`, enter the command:

```
ulimit -a
```

The stack size limit can be increased, in `csh`, by using the command:

```
limit stacksize unlimited
```

or, in `bash` or `sh`, by entering the command:

```
ulimit -s unlimited
```

Selecting PARDISO in Sentaurus Device

PARDISO is activated in Sentaurus Device by specifying in the command file:

```
Math {
   ...
   Method = Blocked
   SubMethod = Pardiso
   WallClock
   ...
}
```

For single-device simulations only, you can specify `Method=Pardiso` instead of `Method=Blocked SubMethod=Pardiso`.

The keyword `WallClock` is used to print the wallclock times of the Newton solver. This is useful and recommended when investigating the performance of the parallel execution.

PARDISO accepts options that can be specified in parentheses:

```
Pardiso (<options>)
```

Table 1 PARDISO options

Option	Description	Default
`IterativeRefinement`	Performs up to two iterative refinement steps to improve the accuracy of the solution.	off
`MultipleRHS`	PARDISO solves linear systems with multiple right-hand sides. This option applies to AC analysis only. It might produce minor performance improvements.	off
`NonsymmetricPermutation`	Computes an initial nonsymmetric matrix permutation and scaling, which places large matrix entries on the diagonal.	on
`RecomputeNonsymmetricPermutation`	Computes a nonsymmetric matrix permutation and scaling before each factorization.	off

To switch off any option, use a minus sign, for example, `-NonsymmetricPermutation`.

The default options `-IterativeRefinement`, `NonsymmetricPermutation`, and `-RecomputeNonsymmetricPermutation` provide the best compromise between speed and accuracy. However, note the following:

- To improve speed, use `-NonsymmetricPermutation`.
- To improve accuracy at the expense of speed, use `IterativeRefinement`, or `RecomputeNonsymmetricPermutation`, or both.

The number of threads for PARDISO can be specified in the `Math` section of the Sentaurus Device command file as follows:

```
Math {
    ...
    Number_of_Threads = 2
    Number_of_Solver_Threads = 2
    ...
}
```

The keyword `Number_of_Threads` defines the number of threads for both the matrix assembly and PARDISO, and `Number_of_Solver_Threads` defines only the number of threads for PARDISO itself. Instead of a constant number of threads, you can specify `maximum`. In this case, the number of threads is set equal to the number of processors available on the execution platform.

If no specification appears in the `Math` section, Sentaurus Device checks the values of the following UNIX environment variables (in order of decreasing priority):

```
SDEVICE_NUMBER_OF_SOLVER_THREADS

SDEVICE_NUMBER_OF_THREADS

SNPS_NUMBER_OF_THREADS

OMP_NUM_THREADS
```

For example, to obtain parallel execution with two threads, you can define `OMP_NUM_THREADS` as follows (in a C shell):

```
setenv OMP_NUM_THREADS 2
```

In a Bourne shell, the equivalent commands are:

```
OMP_NUM_THREADS=2
export OMP_NUM_THREADS
```

Selecting PARDISO in Sentaurus Process

In Sentaurus Process, the PARDISO solver is the default for 1D simulations and 2D mechanics simulations, and also can be used in 2D diffuse simulations and some 3D simulations by specifying:

```
math diffuse dim=2 pardiso
math diffuse dim=3 pardiso
```

or:

```
math flow dim=3 pardiso
```

for diffusion simulations or mechanics simulations, respectively.

The number of threads must be specified in the `math` command. For example:

```
math numThreadsPardiso=2
```

Note:

For Sentaurus Process, PARDISO no longer depends on the OpenMP environment variable `OMP_NUM_THREADS`, and you no longer need to specify this variable.

For Sentaurus Process, by default, PARDISO uses multiple minimum degree (MMD) ordering in 2D simulations and nested dissection (ND) ordering in 3D simulations. You can change the ordering using the `Pardiso.Ordering` parameter to specify ND ordering (`2`) or MMD ordering (`0`):

```
pdbSetDouble Pardiso.Ordering 2
pdbSetDouble Pardiso.Ordering 0
```

Selecting PARDISO in Sentaurus Interconnect

In Sentaurus Interconnect, the PARDISO solver is the default for 1D simulations and 2D mechanics simulations, and also can be used in 2D solve steps and some 3D simulations by specifying:

```
math compute dim=2 pardiso
math compute dim=3 pardiso
```

or:

```
math flow dim=3 pardiso
```

for solve steps in 2D, 3D, or mechanics simulations, respectively.

The number of threads must be specified in the `math` command. For example:

```
math numThreadsPardiso=2
```

For Sentaurus Interconnect, by default, PARDISO uses MMD ordering in 2D simulations and ND ordering in 3D simulations. You can change the ordering using the `Pardiso.Ordering` parameter to specify ND ordering (`2`) or MMD ordering (`0`):

```
pdbSetDouble Pardiso.Ordering 2
pdbSetDouble Pardiso.Ordering 0
```

References

[1] O. Schenk, *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*, Series in Microelectronics, vol. 89, Konstanz, Germany: Hartung-Gorre, 2000.

[2] O. Schenk, K. Gärtner, and W. Fichtner, “Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors,” *BIT*, vol. 40, no. 1, pp. 158–176, 2000.

[3] P. Matstoms, “Parallel sparse QR factorization on shared memory architectures,” *Parallel Computing*, vol. 21, no. 3, pp. 473–486, 1995.

[4] A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[5] O. Schenk, M. Hagemann, and S. Röllin, “Recent advances in sparse linear solver technology for semiconductor device simulation matrices,” in *International Conference on Simulations of Semiconductor Processes and Devices (SISPAD)*, Boston, MA, USA, pp. 103–108, September 2003.

[6] O. Schenk, S. Röllin, and A. Gupta, “The Effects of Unsymmetric Matrix Permutations and Scalings in Semiconductor Device and Circuit Simulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 3, pp. 400–411, 2004.

[7] O. Schenk and K. Gärtner, “Solving unsymmetric sparse systems of linear equations with PARDISO,” *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.

[8] G. Karypis and V. Kumar, *Analysis of Multilevel Graph Partitioning*, Technical Report 95-037, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, USA, 1995.

[9] J. W. H. Liu, “Modification of the Minimum-Degree Algorithm by Multiple Elimination,” *ACM Transactions on Mathematical Software*, vol. 11, no. 2, pp. 141–153, 1985.

[10] M. Yannakakis, “Computing the Minimum Fill-in Is NP-Complete,” *SIAM Journal on Algebraic and Discrete Methods*, vol. 2, no. 1, pp. 77–79, 1981.

[11] E. Rothberg, *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization*, PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[12] J. J. Dongarra *et al*., “A Set of Level 3 Basic Linear Algebra Subprograms,” *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.

[13] C. L. Lawson *et al*., “Basic Linear Algebra Subprograms for Fortran Usage,” *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[14] O. Schenk, K. Gärtner, and W. Fichtner, “Scalable Parallel Sparse Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors,” in *High-Performance Computing Networking, 7th International Conference, HPCN Europe*, Amsterdam, The Netherlands, pp. 221–230, April 1999.

[15] O. Schenk, K. Gärtner, and W. Fichtner, *Application of Parallel Sparse Direct Methods in Semiconductor Device and Process Simulation*, Technical Report 99/7, Integrated Systems Laboratory, ETH, Zurich, Switzerland, 1999.

[16] L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

Part II: SUPER

This part contains information about the direct linear solver SUPER and is intended for users of Sentaurus Device:

- Chapter 2, Using SUPER provides background information on SUPER
- Chapter 3, Implementing SUPER discusses the algorithms used in SUPER

2

Using SUPER

SUPER is a library that contains a set of block-oriented and nonblock-oriented, supernodal, factorization algorithms for the direct solution of sparse structurally symmetric linear systems.

Features of the SUPER Solver

SUPER is a fast direct solver for the semiconductor device simulator Sentaurus Device, where the solution of sparse structurally symmetric linear systems of equations (typically written in the form $Ax = b$) is the main task consuming most of the processor time.

Advances in sparse matrix technology have resulted in supernodal linear solvers. The key concept behind this technique is based on the concept of a supernode [1]. In the course of the factorization of the coefficient matrix, supernodes are identified as a set of consecutive columns in the factor L of the LU decomposition with the following structural properties.

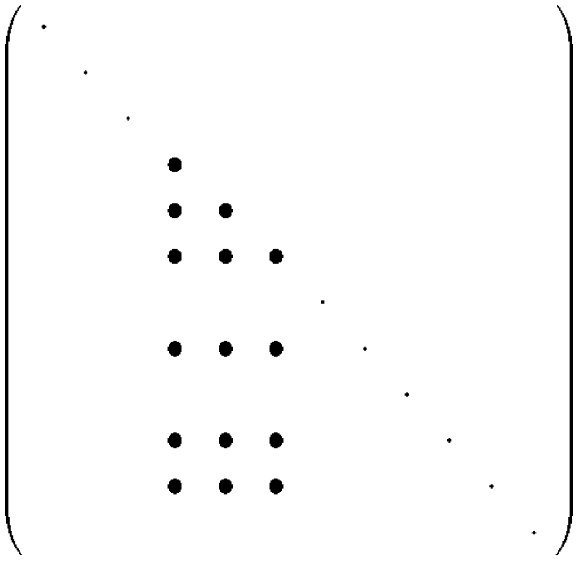
Assume $\{k, k+1, \ldots, k+r\}$ is a set of consecutive columns and $\eta(k)$ denotes the number of nonzero entries in column k of the factor L. If all $k+i,\ i = 0 \ldots r$ columns share the same sparsity structure below row $k+r$ and $\eta(k+i) = \eta(k+r) + r - i,\ i = 0 \ldots r$, then the set $\{k, k+1, \ldots, k+r\}$ forms a supernode [2].

In other words, a supernode formed by s adjacent columns consists of two blocks: a dense diagonal block of size $s \times s$ and a block of width s below the diagonal block where all columns share the same sparsity pattern. Due to structural symmetry, the term *supernode* can also apply to the rows of the factor U. For simplification, this user guide restricts its considerations mainly to the columns of factor L. Figure 1 on page 18 illustrates a supernode.

Supernodes offer a significant advantage for numeric factorization: a column j being computed is modified by either all or none of the columns of a supernode S, which updates column j [3]. In addition, if column j has an identical sparsity structure compared to the columns of supernode S below row j, updating column j is a dense operation, meaning that no index list is needed to reference the various elements. This is also true for column updates within the same supernode. The fact that dense linear algebra operations can be performed in those cases reduces memory traffic and increases computational efficiency. This is documented in a number of papers [1][4][5].

SUPER incorporates the advances in supernodal sparse matrix technology towards the most efficient solution of a given linear system. SUPER provides different supernodal factorization methods that give excellent performance on both RISC and vector machines.

Figure 1 Example of a supernode



You can fine-tune SUPER although this is not necessary, since all tunable parameters have built-in default values or are set automatically during execution. Some parameters relate to measured times during execution; therefore, they influence the computational behavior on different hardware platforms.

Customizing SUPER: The .superrc File

You can tailor SUPER behavior to your own preferences by modifying the parameters specific to SUPER in the `.superrc` file. The software uses the following procedure to search for this configuration file. First, SUPER checks whether the environment variable SUPERRC is set. This environment variable must contain the absolute path of the directory, which contains the `.superrc` file. SUPER checks whether the `.superrc` file exists; if so, the configuration file is used. If the environment variable SUPERRC is not set or the directory specified does not contain a `.superrc` file, the home directory of the user is sought. Finally, if neither location contains a `.superrc` file, the configuration file is sought in the current directory.

This hierarchical concept allows for the following:

- A group of users can share a common `.superrc` file by specifying its location in the SUPERRC environment variable.
- Individual users can have their own personal global `.superrc` file in their home directory.
- Individual configuration files can be used when put into the current working directories.

SUPER uses default settings if no configuration file is found.

Grammar of the Input Language

Terminal symbols are presented in Courier font and nonterminal symbols are uppercase and italicized:

```
STATEMENTS            ←  STATEMENT

                      |      STATEMENTS, STATEMENT

STATEMENT             ←  factorization_type = FACTORIZATION_METHOD
                      |  write { INTEGER_LIST }
                      |  write ( FORMAT )
                      |  write ( FORMAT ) { INTEGER_LIST }
                      |  write

FACTORIZATION_METHOD  ←  column_supernode_0
                      |  column_supernode_1
                      |  column_supernode_2
                      |  column_supernode_3
                      |  block_supernode_0
                      |  block_supernode_1
                      |  block_supernode_2
                      |  block_supernode_3
                      |  block_supernode_4

FORMAT                ←  blsmp
                      |  matlab

INTEGER_LIST          ←  INTEGER
                      |  INTEGER_LIST : INTEGER
```

The value of `factorization_type` specifies the factorization to be used. The factorization within SUPER is performed using supernodal algorithms. Generally, two types of supernodal algorithms are available: column supernode and block supernode (see Sparse Supernodal Factorization Algorithms on page 30).

There are four column supernode algorithms and five block supernode algorithms. In terms of memory consumption, column supernode algorithms are preferred over block supernode algorithms. The algorithm `column_supernode_2` uses minimal space and the algorithm

`block_supernode_1` requires maximal space. Conversely, if speed is an important consideration, block supernode algorithms should be considered because they reduce memory traffic and support data locality. By default, SUPER uses `column_supernode_1`.

The `write` statement is used to write linear systems in ASCII representation to files. The parameter `INTEGER_LIST` must contain nonnegative numbers separated by colons. It determines at which invocation of SUPER the output files should be generated. The list does not have to be in increasing order. If `INTEGER_LIST` is missing, the first ten invocations of SUPER generate the file output.

The parameter `FORMAT` determines the format of the output:

- If `blsmp` is selected, then the matrix (the right-hand side) and the solution of the linear system are written to either the `nsuper_blsmp_real_index.txt` file or the `nsuper_blsmp_complex_index.txt` file.
- If `matlab` is selected, then output is sent to either the `nsuper_matlab_real_index.m` file or the `nsuper_matlab_complex_index.m` file.

By default, no output is generated.

In many cases, you can completely ignore setting up a special `.superrc` file and can rely on the defaults. Conversely, there is no way to change the default settings without modifying the corresponding parameter in the `.superrc` file. In addition, the `.superrc` file is read only once, at the initial invocation of SUPER.

Example of a .superrc File

```
factorization_type = block_supernode_4,
write (blsmp) {5:9}
```

These settings instruct SUPER to use the factorization algorithm `block_supernode_4` and to generate ASCII files, in `blsmp` format, of the fifth and ninth linear systems solved.

References

[1] C. C. Ashcraft *et al*., “Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers,” *The International Journal of Supercomputer Applications*, vol. 1, no. 4, pp. 10–30, 1987.

[2] J. W. H. Liu, E. Ng, and B. W. Peyton, *On Finding Supernodes for Sparse Matrix Computations*, Technical Report ORNL/TM-11563, Oak Ridge National Laboratory, Oak Ridge, TN, USA, June 1990.

[3] E. Rothberg and A. Gupta, *An Evaluation of Left-Looking, Right-Looking and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchical-Memory Machines*, Technical Report STAN-CS-91-1377, Department of Computer Science, Stanford University, Stanford, CA, USA, August 1991.

[4] P. Arbenz and W. Gander, A *Survey of Direct Parallel Algorithms for Banded Linear Systems*, Technical Report 221, Institute of Scientific Computing ETH, Zurich, Switzerland, October 1994.

[5] E. G. Ng and B. W. Peyton, *A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors*, Technical Report ORNL/TM-11814, Oak Ridge National Laboratory, Oak Ridge, TN, USA, April 1991.

3

Implementing SUPER

This chapter describes the algorithms in SUPER.

Algorithms

Typically, you want to solve a linear system of the form:

$$Ax = b \tag{1}$$

where A is the structurally symmetric coefficient matrix of the system, b denotes the solution vector or the right-hand side, and x is the vector of all unknowns, commonly referred to as the solution. A permutation matrix P is used to apply row and column permutations to the coefficient matrix A. Now, the linear system Equation 1 becomes:

$$PAP^T\tilde{x} = \tilde{b} \tag{2}$$

where $\tilde{x} = Px$ and $\tilde{b} = Pb$. The permuted coefficient matrix PAP^T is decomposed into two triangular factors L and U. For example:

$$PAP^T = LU \tag{3}$$

Eventually, the linear system Equation 2 is solved by forward and backward substitution:

$$\begin{aligned} Ly &= Pb \\ U\tilde{x} &= y \end{aligned} \tag{4}$$

Finally, the solution x of the original linear system Equation 1 is obtained by left-multiplying $\tilde{x}$, the solution of Equation 2, with P^T [1].

Technically, the solution process of SUPER has the following distinct phases leading to a modular code that is easier to maintain and optimize (this approach has been used in other solver packages such as SPARSPAK [2] and YSMP [3]):

- Structure input
- Reordering
- Symbolic factorization

- Numeric value input
- Numeric factorization
- Numeric solution

Structure Input

During the *structure input* phase, the solver reads the nonzero structure of the lower triangle of the coefficient matrix A and generates a full adjacency structure of A, which passes to the reordering phase.

Reordering

Reordering is a very important phase in the solution process. The goal of applying row and column permutations to the coefficient matrix is to minimize the size of its factors L and U. Any additional nonzero entry in the decomposition is called a fill-in entry. In terms of computational cost (that is, memory consumption and execution time), you might want to retain the nonzero structure of the coefficient matrix in its factors or at least to reduce growth to a minimum. Although there is no minimum fill-in reordering scheme [4], a number of heuristics, mainly using graph theoretical approaches, produce near-to-optimal reorderings. Among these approaches, the minimum degree reordering heuristic has proven to be most effective [5].

The SUPER solver uses an enhanced minimum degree algorithm called the multiple minimum degree (MMD) algorithm [6][7]. Its motivation is based on the observation that in the course of elimination, expensive degree updates can be saved if nodes of the same degree are eliminated simultaneously, thereby producing supernodes as a side effect [8]. See How Multiple Minimum Degree Works on page 25.

Symbolic Factorization

When the coefficient matrix is reordered, you want to predetermine the structure of its factors L and U. This process is referred to as *symbolic factorization* [9]. Knowing the factor structure, you can preallocate the necessary memory space for the remainder of the solution process.

Numeric Value Input

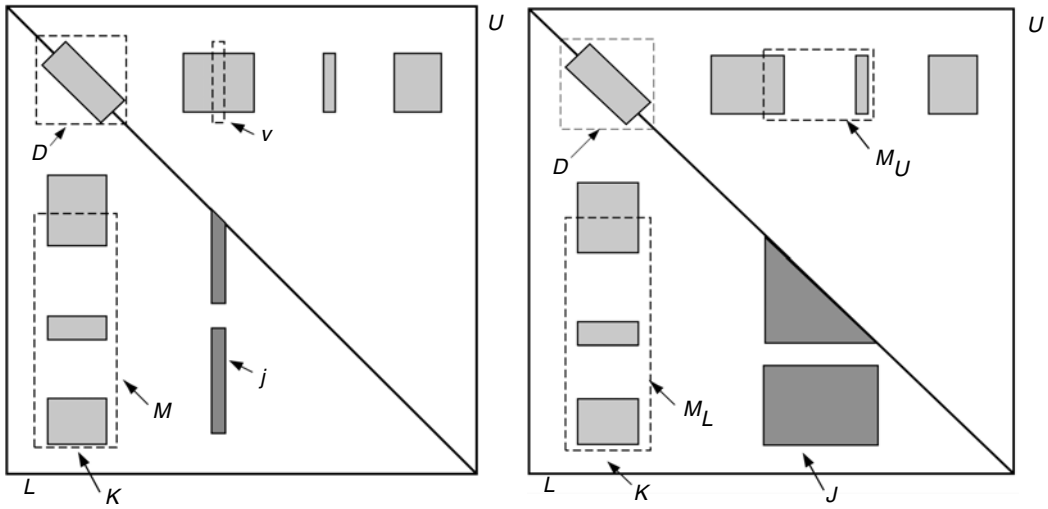
So far, only preliminary steps toward the numeric solution of the linear system have been performed. The *numeric value input* phase is the preparation step for numeric computation.

The numeric values of the coefficient matrix A are read into their memory locations, simultaneously applying the row and column permutations found in the reordering phase.

Numeric Factorization

Numeric factorization is the most time-consuming phase of the solution process. Extensive research to find optimal performance in terms of speed and memory requirements has lead to supernodal techniques [10]. Column supernode and block supernode algorithms are implemented (see Figure 2).

Figure 2 Illustration of (left) column supernode and (right) block supernode updating



Column supernode updating describes a technique where only one column of the factor L is computed at a time. Consider Figure 2 (*left*): column j is updated by supernode S. Computing this update is expressed mathematically in the term:

$$j = j - M(Dv) \tag{5}$$

This is known as a DGEMV operation in BLAS terminology [11]. Computing $M(Dv)$ is a dense operation that requires no indirect addressing.

When the result of this matrix–vector product is subtracted from vector j, the elements of the resulting vector must be scattered into their corresponding positions only.

Block supernode factorization operates on groups of columns or a complete supernode at the same time instead of merely focusing on a single column. It must compute:

$$J = J - M_L(DM_U) \tag{6}$$

This represents a DGEMM operation [12]. Block supernode algorithms mainly involve dense matrix–matrix multiplications, thereby reducing memory traffic. Analogous to column

supernode algorithms, indirect addressing is necessary when the results of the dense matrix–matrix multiplication are scattered into the updated supernode. Since DGEMV and DGEMM operations are highly efficient computational kernel routines, their use during numeric factorization significantly speeds up decomposition. Sparse Supernodal Factorization Algorithms on page 30 describes all the supernodal algorithms implemented in SUPER.

Numeric Solution

The *numeric solution* phase is the final step in the solution process. The solution is found using forward and backward substitution to exploit the supernodal partitioning of the factors. Detailed discussions of this are documented in the literature [8][13][14][15].

How Multiple Minimum Degree Works

Before going into detail, a few preliminary terms must be defined for subsequent use.

Let $G = (V, E)$ be a graph.

Def.: adjacency set

Let $v \in V; adj(v) = \{w \in V | (v, w) \in E\}$

The adjacency set $adj(v)$ for any $v \in V$ consists of all nodes $w \in V$, which are directly connected with v through an edge from set E.

Def.: indistinguishable

Let $v, w \in V$; v is indistinguishable from $w : \Leftrightarrow adj(v) \cup \{v\} = adj(w) \cup \{w\}$

Two nodes $v, w \in V$ are said to be indistinguishable if and only if v and w have identical adjacency sets and each node is contained in the other's adjacency set.

Note:

The concept of *indistinguishable* nodes is covered extensively in the literature [2].

Practically, the *adjacent set* defines the term *clique* where all nodes are connected to each other.

As previously mentioned, multiple minimum degree (MMD) is a variant of the minimum degree (MD) ordering algorithm. Its concept is based on the observation that, during elimination, expensive degree updates can be saved if nodes of the same MD are eliminated simultaneously. For indistinguishable nodes, it can be shown that they are eliminated consecutively when MD is used.

Figure 3 lists the MMD algorithm. Initially, S is set equal to the empty set and the degrees of all nodes in V are computed. Next, a set T is determined, which contains all nodes from V to S that have MD. Mass elimination is performed over all elements of T. On entry, all elements (nodes) are unflagged (unmarked). Next, a node $y \in T$ must be selected. The criteria that set out how to select elements from T are called tie-breaking strategies.

Figure 3 Multiple minimum degree (MMD) algorithm

```
S = ∅
for x ∈ V do
   δ(x) = |adj(x)|
end for
while S ≠ V do
   set T = {y ∈ V - S|δ(x) = min_{x ∈ V - S} δ(x)}
   for y ∈ T do
      if y is not marked do
         set Y = {x ∈ T|x indistinguishable from y}
         for all nodes x ∈ Y do
             order x next
         end for
         mark all nodes in adj(Y) and Y
         S = S ∪ Y
      end if
   end for
   eliminate all marked nodes in S from the graph
   for all marked nodes x ∈ V - S do
      δ(x) = |adj(x)|
   end for
   unmark all nodes
end while
```

Effective tie-breaking is known to improve numeric factorization since the fill-in of the factor L can be reduced significantly [5]. SUPER does not implement any of the commonly used tie-breaking strategies used in other well-known solver packages, such as MA27 (Harwell Laboratories), SPARSPAK (University of Waterloo), and YSMP (Yale University).

Instead, SUPER uses random tie-breaking, which is the selection of elements without intelligence; mostly implied by the underlying data structure.

After an element $y \in T$ is chosen, the algorithm determines the set Y that contains all elements of T indistinguishable from y (element y is trivially indistinguishable from itself). When Y is computed, all elements of Y and the adjacency set of Y, $adj(Y)$, are flagged. There are two reasons for this. First, flagging the nodes of set Y prevents double-accessing indistinguishable nodes, that is, nodes found to be indistinguishable from y, the current node, do not have to be looked at while mass elimination proceeds, because they are eliminated with y. Second, nodes that lie in $adj(Y)$ must be marked for a degree update, because some of their neighbors, some or all elements of Y, are eliminated. This means their current degree was modified.

Finally, set S is unified with set Y and mass elimination starts over with another element $y \in T$ until no unflagged element remains. Then, the graph representation of the remaining nodes from V to S is computed. Simultaneously, all flagged nodes in V to S undergo a degree update. Finally, the non-eliminated nodes are unmarked and the algorithm continues until $S = V$.

Example of Executing Multiple Minimum Degree

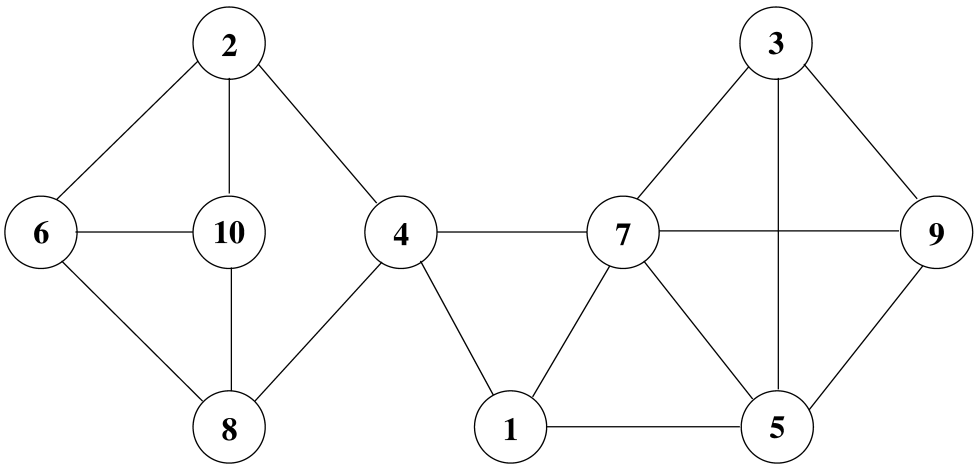
Figure 4 provides the symmetric pattern of the matrix A where • denotes a nonzero entry.

Figure 4 Sample sparse matrix A

$$\mathbf{A} = \begin{pmatrix} 1 & & \bullet & \bullet & & \bullet & & & \\ & 2 & \bullet & & \bullet & & & & \bullet \\ & & 3 & & \bullet & & \bullet & & \bullet \\ \bullet & \bullet & & 4 & & & \bullet & \bullet & \\ \bullet & & \bullet & & 5 & & \bullet & & \bullet \\ & \bullet & & & & 6 & & \bullet & \bullet \\ \bullet & & \bullet & \bullet & \bullet & & 7 & & \bullet \\ & & & \bullet & & \bullet & & 8 & \bullet \\ & & \bullet & & \bullet & & \bullet & & 9 \\ & \bullet & & & & \bullet & & \bullet & & 10 \end{pmatrix}$$

Figure 5 illustrates the graph representation of A.

Figure 5 Graph representation of sample matrix A



The numbering in the graph is equal to the line numbering of the matrix. The initial minimum degree of the graph is 3 (self-loops are neglected). Therefore, the ordering algorithm starts with:

$$S = \varnothing \qquad T = \{10, 9, 8, 6, 3, 2, 1\} \tag{7}$$

Now, $y = 10$ is chosen from T. The only indistinguishable node from $y = 10$ is the node with the number 6, yielding $Y = \{(10, 6)\}$ (parentheses are used only to identify groups of indistinguishable nodes). The adjacency set $adj(Y)$ contains the nodes 2 and 8 that, therefore, are flagged (indicated by +). S becomes $S = \{(10, 6)\}$. After the first loop through the mass elimination step:

$$S = \{(10, 6)\} \qquad T = \{10^{+}, 9, 8^{+}, 6^{+}, 3, 2^{+}, 1\} \tag{8}$$

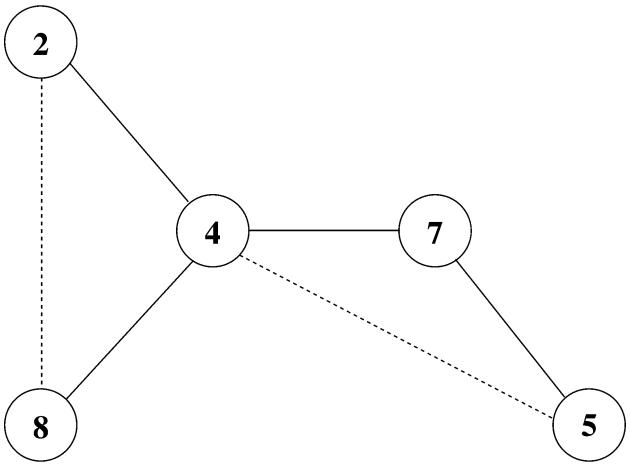
The second loop finds $y = 9$ and $Y = \{(9, 3)\}$, since node 3 is indistinguishable from node 9. Nodes 7 and 5 are marked because they are adjacent to Y. By the end of the loop:

$$S = \{(10, 6), (9, 3)\} \qquad T = \{10^{+}, 9^{+}, 8^{+}, 6^{+}, 3^{+}, 2^{+}, 1\} \tag{9}$$

Node $y = 1$ is the only unflagged node left in T. $y = 1$ has no indistinguishable nodes besides itself. Therefore, only $y = 1$ is eliminated, leaving adjacent node 4 flagged. All elements of T are now flagged and the algorithm proceeds to the degree update step.

Figure 6 shows the graph representation of the remaining nodes all of which had their degree updated because they were all flagged.

Figure 6 Elimination graph after first loop through multiple mass elimination



The new minimum degree is 2, which yields:

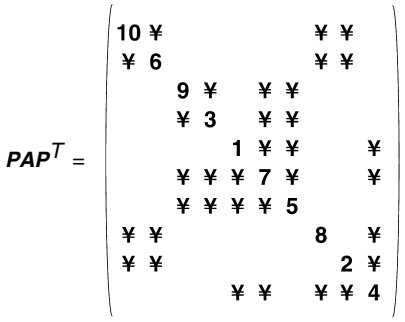
$$S = \{(10, 6), (9, 3), 1\} \qquad T = \{7, 8, 5, 2\} \tag{10}$$

The algorithm finds nodes 7 and 5 as well as nodes 8 and 2 to be indistinguishable, respectively. They are eliminated leaving only node 4. The reordering sequence or permutation is now computed to be:

$$P = (10, 6, 9, 3, 1, 7, 5, 8, 2, 4) \tag{11}$$

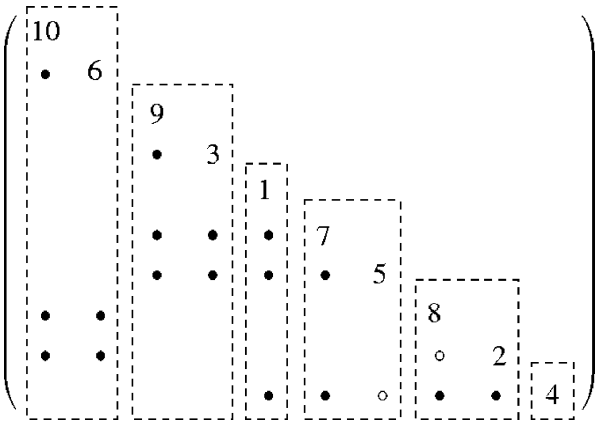
Applying this permutation to the matrix A results in the structure shown in Figure 7.

Figure 7 Sample matrix A reordered with MMD



Performing symbolic factorization on this matrix reveals the sparsity pattern of the factor L, which is depicted in Figure 8 where the columns have been renumbered.

Figure 8 Sparsity structure of factor L of A



Note:

The sparsity structures for PAP^T and L are similar. L has two additional nonzero fill-in entries (indicated by ‘o’) and consists of groups of columns that share the same sparsity pattern, such as columns 10 and 6, or 9 and 3 (indicated by the dashed rectangles).

These groups of columns correspond to the sets Y of indistinguishable nodes as they are found in the course of the mass elimination step. These groups form supernodes [8], which

have an important role in improving the performance of the numeric factorization. SUPER focuses entirely on the supernodal update scheme. You can take advantage of the fact that a column update depends on all previous columns of the same supernode and on all nodes of other supernodes that update this column.

Using BLAS terminology [11][12][16], the first type of update mentioned involves dense SAXPY operations, while the second type performs so-called indexed SAXPY or SAXPYI operations [13][17].

In addition, updating a column j by a supernode S requires one gather and one scatter operation, while node–node updates require as many operations as there are nodes in S of each [13]. Therefore, memory traffic is reduced and numeric factorization is accelerated, especially on machines with hardware-supported gather-and-scatter operations.

Sparse Supernodal Factorization Algorithms

Generally, matrix reordering and numeric factorization are the parts of a direct solver package where most of the execution time is spent. Depending on the algorithm and its implementation, the time necessary to reorder the input matrix can vary significantly and might even dominate the factorization time. Nevertheless, these are rare cases, since the reordering algorithm does not have to deal with any fill-in that occurs during LU decomposition. This leaves numeric factorization as the part to focus on for performance improvements.

Factorization algorithms based on supernodal techniques have proven to be superior over former general approaches [8][13][18][19].

The next subsections describe the supernodal factorization algorithms implemented in SUPER. These algorithms fall into two types of approaches: column supernode and block supernode.

Table 2 Symbols used in algorithms

Symbol	Description
J, K	Supernodes of the LDU decomposition
j, k	Nodes, that is, columns or rows of a supernode
N_S	Number of supernodes
t_L, t_U	Temporary work vectors
T_L, T_U	Temporary blocks of workspace
$A_{*,j}, A_{j,*}$	A column or row of the coefficient matrix A

Table 2 Symbols used in algorithms (Continued)

Symbol	Description
$A_{*,J}, A_{J,*}$	A column or row block of the coefficient matrix A
$L_{*,j}(L_{*,J})$	A (block) column of the factor L
$U_{j,*}(U_{J,*})$	A (block) row of the factor U
$c_i(r_i)$	*i*-th element of column (row) vector $c(r)$
d_j	*j*-th diagonal element of the matrix D of the LDU decomposition
im, ri	Index vectors
$[L_{*,j}]_{im}$	Scattering into column $L_{*,j}$ is performed using index map im
n	Number of equations of the linear system
ne	Number of off-diagonal nonzero entries in the lower-half or upper-half of A
$\|L\|$	Number of nonzero entries in the factor L
$\|S\|$	Number of supernodes
maxcol	Maximum number of nonzero entries in a column of L
maxsup	Maximum number of columns in a supernode

Column Supernode Algorithms

Column supernode updating describes a technique where only one column or row of the factors L and U is computed at a time, although the corresponding supernode can consist of several columns or rows.

The column_supernode_0 Algorithm

Figure 9 on page 32 lists the `column_supernode_0` algorithm. Initially, the algorithm reveals the general form of column supernode algorithms: a triple-nested for-loop (indicated with indices c0.1 to c0.3). The outermost loop runs over all supernodes J that were generated in the reordering and symbolic factorization steps. The next for-loop (c0.2) proceeds one level deeper and scans over all nodes j of the current supernode J starting with the smallest index.

Note:

The product of the loop lengths for loop c0.1 and c0.2 is always equal to the dimension of the matrix A.

Finally, the innermost loop (c0.3) handles the contribution of all updating supernodes K to the current node j.

Furthermore, three computationally intensive kernels *CRmod_d* and *CRmod_i* (see Figure 10 on page 33) and *CRdiv* (see Figure 11 on page 33 (*left*)) are typical for LU decomposition methods [8][20].

Figure 9 Algorithm: column_supernode_0

```
t_L ← 0; t_U ← 0
for J = 1 to N_S do                                        (c0.1)
   for j ∈ J (in order) do                                 (c0.2)
      [t_L]_ind ← A_*,j
      [t_U]_ind ← A_j,*
      for all K updating j do                              (c0.3)
         if (K and J have same sparsity pattern)           (c0.4)
            collect dense updates
         else
            for k ∈ K do                                   (c0.5)
               CRmod_i(t_L, t_U, ind, j, k)
            end for
         end if
      end for
      [L_*,j]_ind ← t_L; t_L ← 0
      [U_j,*]_ind ← t_U; t_U ← 0
      for all dense updates k do                           (c0.6)
          CRmod_d(L_*,j, U_j,*, j, k)
      end for
      CRdiv(j)
   end for
end for
```

CRmod_i and *CRmod_d* describe the necessary operations to calculate the update of column $L_{*,k}$ and row $U_{k,*}$ on the current column j using indexed SAXPY [13][21] and dense SAXPY [16] operations, respectively. The contribution of these two vectors is then accumulated into the column vector c and the row vector r. *CRdiv* describes the scaling procedure after column or row j has been updated. All of these kernel routines can be vectorized, thereby running very efficiently on machines with vector capabilities.

A third task, which is also common to all algorithms implemented in SUPER, is the determination of the row structure of the factor L (or, identically, the determination of the column structure of U). This row structure is required to find all supernodes updating the current column j (see loop c0.3 in Figure 9). As described [8], it is not necessary to

calculate the row structure of L beforehand, since it can be efficiently generated during factorization.

Specific to this algorithm is the use of the temporary vectors t_L and t_U, and, as a result, the implementation of *CRmod_{i,d}* and *CRdiv*. Vectors t_L and t_U contain intermediate results for the factors L and U, respectively. Both vectors are of length n where n is the dimension of the matrix A of the linear system. Initially, t_L and t_U are set to zero. Then, for every column or row j to be computed (loop c0.2), column $A_{*,j}$ is loaded into t_L and row $A_{j,*}$ is loaded into t_U.

Figure 10 (Left) CRmod_d kernel and (right) CRmod_i kernel

```
for i = j to n do
   c_i = c_i - l_{i,k} d_k u_{k,j}
   r_i = r_i - u_{k,i} d_k l_{j,k}
end for
```

```
for i = j to n do
    l = ind(i)
    c_l = c_l - l_{i,k} d_k u_{k,j}
   r_l = r_l - u_{k,i} d_k l_{j,k}
end for
```

Figure 11 (Left) CRdiv kernel and (right) setup for vector im

```
d_j = l_{j,j}
for i = j + 1 to n do
   l_{i,j} = l_{i,j} / d_j
   u_{j,i} = u_{j,i} / d_j
end for
```

```
i = 0
 for all row indices k of j do
    im(k) = i
        i = i + 1
 end for
```

This is performed by expanding (scattering) the densely stored column or row elements of A into their corresponding positions into t_L and t_U. Hereby, it is possible to accumulate all indexed updates to column j without repeatedly storing the contents of the temporary vectors t_L and t_U into factor storage and simultaneously zeroing out both vectors. Additionally, the index vector *ind* (loop c0.5) simply holds the row structure of the current column j, which does not have to be computed, since it is provided by the symbolic factorization. Doing this significantly reduces memory traffic at the cost of comparably little storage overhead (compared to the fill-in size).

In addition to saving memory transfers, algorithm `column_supernode_0` increases computational efficiency by collecting all dense updates (collected in statement c0.4) and executing them in one block in loop c0.6. This requires additional storage to keep track of all nodes that share the same sparsity pattern as column/row j, but provides for a compact dense update procedure. After column j has been computed, it must be scaled by its diagonal dj. This is performed in the kernel routine *CRdiv*.

Note:

The computation of the scaling diagonal dj is performed along with the column/row $L_{*,j}/U_{j,*}$ instead of calculating its value separately. The data structures used were dimensioned to have extra space for the diagonal element, thus exploiting vectorization capabilities on the different hardware platforms.

The column_supernode_1 Algorithm

Figure 12 is an enhanced version of the previous algorithm. In this case, it was feasible to reduce the storage overhead introduced by the temporary vectors t_L and t_U.

Figure 12 Algorithm: column_supernode_1

```
t_L ← 0     t_U ← 0    im ← 0
for J = 1 to N_S do
   setup vector im                                          (c1.1)
   for j ∈ J (in order) do
      [t_L]_im ← A_*,j                                      (c1.2)
      [t_U]_im ← A_j,*
      for all K updating j do
         if (K and J have same sparsity pattern)
             collect dense updates
         else
            for k ∈ K do                                    (c1.3)
                CRmod_i(t_L, t_U, im, j, k)
            end for
         end if
      end for
      L_*,j ← t_L     t_L ← 0
      U_j,* ← t_U     t_U ← 0                               (c1.4)
      for all dense updates k do
         CRmod_d(L_*,j, U_j,*, j, k)
      end for
      CRdiv(j)
   end for
end for
```

Instead of occupying space for $2*n$ real numbers, algorithm `column_supernode_1` needs only $2*(MAXCOL+1)$ where $MAXCOL$ denotes the maximal number of nonzero entries in a column of L excluding the diagonal element. $(MAXCOL+1)$ is needed here to hold the diagonal element of the current column.

In 2D and 3D device simulations, where n is typically greater than 5000, $MAXCOL$ is much smaller than n [22]. (Experimental results revealed $MAXCOL$ to be less than 10% of n in 2D device simulation.)

Conversely, the relative indexing technique was utilized [15][23] so that the algorithm `column_supernode_1` can use smaller temporary vectors. Relative indexing introduced an additional vector im of length n (c1.1), where im stands for *index map*. Nevertheless, the total amount of overhead storage required for algorithm `column_supernode_1` is approximately 60% of that used in `column_supernode_0`.

Figure 11 on page 33 (*right*) shows the vector im setup. Basically, the row index vector for the first column j of supernode J is scanned and the corresponding position in vector im is set to the value of the integer variable i, which is incremented by one after each

assignment starting with zero. Thereby, referencing im_k for a row index K returns the relative position of the corresponding column element c_k within t_L.

Note:

The row index vector is stored in decreasing order (looking at the column from the bottom) by the symbolic factorization phase of the solver.

Vector im is then used to copy the nonzero elements of column or row $A_{*,j}/A_{j,*}$ into t_L and t_U (c1.2) and to perform the indexed updates in loop c1.3. Both operations take advantage of the fact that the set of row indices for $A_{*,j}$ and the updating supernodes K up to row j from a subset of column j's set of row indices in the factor L [24].

This is also the reason why im does not have to be reset to zero when all nodes j of supernode J have been computed; this reduces memory traffic. Finally, storing the contents of t_L and t_U into factor storage (c1.4) does not require indirect addressing and can be performed one by one, because t_L/t_U and $L_{*,j}/U_{j,*}$ share the same sparsity pattern.

The column_supernode_2 Algorithm

The algorithm `column_supernode_2` (see Figure 13 on page 36) implements a major change compared to algorithm `column_supernode_1`. Instead of loading column or row $A_{*,j}/A_{j,*}$ of the coefficient matrix A into a temporary work space, the contents are directly transferred into the appropriate places of $L_{*,j}$ and $U_{j,*}$, respectively (see c2.1).

In this case, since the temporary work vectors t_L and t_U are not required, it is possible to further reduce memory consumption. Since all computation is performed within factor space, additional data transfers, and scatter and add operations caused by intermediate results can also be saved (see c1.4 in Figure 12 on page 34). Consequently, algorithm `column_supernode_2` uses the least amount of memory of all algorithms considered in this section.

Figure 13 Algorithm: column_supernode_2

```
im ← 0
for J = 1 to N_S do
   setup vector im
   for j ∈ J (in order) do
      [L_{*,j}]_im ← A_{*,j}                                   (c2.1)
      [U_{j,*}]_im ← A_{j,*}
      for all K updating j do
         if (K and J have same sparsity pattern)
             collect dense updates
         else
            for k ∈ K do                                       (c2.2)
                CRmod_i(L_{*,j}, U_{j,*}, im, j, k)
            end for
         end if
      end for
      for all dense updates k do
         CRmod_d(L_{*,j}, U_{j,*}, j, k)
      end for
      CRdiv(j)
   end for
end for
```

The column_supernode_3 Algorithm

Figure 14 on page 37 shows another variant of column supernode LU factorization. This algorithm requires the same amount of storage overhead as the `column_supernode_1` algorithm, but implements two significant changes computing supernode K's update on column j (see c3.2 and c3.3).

First, like algorithm `column_supernode_2`, column or row $A_{*,j}/A_{j,*}$ of the coefficient matrix A are *not* loaded into the temporary work space but into their appropriate places in $L_{*,j}$ and $U_{j,*}$, respectively (see c3.1). This is not necessarily advantageous concerning memory traffic, since the algorithm still uses temporary work vectors (t_L and t_U), which must be merged into factor storage. The advantage over the other algorithms is assumed to unfold in the fact that you can compute supernode K's contribution updating column j as a dense SAXPY operation (see c3.2), therefore revealing the second major difference mentioned earlier.

Unfortunately, after t_L and t_U have been computed, their contents must be scattered and added to column or row $L_{*,j}/U_{j,*}$ using the index map im of supernode J. This is the cost when using dense SAXPY operations to calculate t_L and t_U. Experiments with real device simulation test cases have shown that computational efficiency suffers from the resulting memory transfers. In addition, t_L and t_U must be reset to zero for the next supernode to update column j (see c3.3). The remainder of algorithm `column_supernode_3` is identical to the algorithms previously discussed.

Figure 14 Algorithm: column_supernode_3

```
t_L ← 0    t_U ← 0    im ← 0
for J = 1 to N_S do
   setup vector im
   for j ∈ J (in order) do
      [L_*,j]_im ← A_*,j                                   (c3.1)
      [U_j,*]_im ← A_j,*
      for all K updating j do
         if (K and J have same sparsity pattern)
             collect dense updates
         else
            for k ∈ K do                                   (c3.2)
                CRmod_d(t_L, t_U, j, k)
            end for
            [L_*,j]_im ← t_L    t_L ← 0                    (c3.3)
            [U_j,*]_im ← t_U    t_U ← 0
         end if
      end for
      for all dense updates k do
         CRmod_d(L_*,j, U_j,*, j, k)
      end for
      CRdiv(j)
   end for
end for
```

Summary of Column Supernode Algorithms

Looking at all the column supernode algorithms discussed reveals that, in all cases, dense updates and column or row scaling are treated equally. Therefore, you can conclude that the data structures involved as well as the execution time necessary for the two operations do not differ (at least not significantly) in all four cases. This leaves the indexed updates and the memory references through gather-and-scatter operations for the temporary vectors t_L and t_U as the critical points for measuring how efficiently the algorithms run on different machines.

In terms of storage overhead and memory transfers, algorithm `column_supernode_2` clearly is the first choice. Although, if execution time is important, most machines seem to prefer algorithm `column_supernode_1` to the others.

In the next section, the number of gather-and-scatter operations is reduced by working on blocks of columns of the same supernode simultaneously.

Block Supernode Algorithms

Block supernode factorization operates on groups of columns or rows, or an entire supernode at the same time instead of merely focusing on a single node. This does not

reduce the number of references to memory, but by grouping them together, memory fetch and store can be more efficient, that is, using the same index map only once throughout a loop cycle. In addition, in terms of vectorization, block supernode factorization does not lengthen the vectorizable loops, thereby increasing the average vector length, but it nests the vectorizable loops one level deeper, which collapses vector work and avoids vector startup overhead.

On the other hand, block supernode factorization increases storage overhead considerably, since the intermediate results for more than one column or row must be retained and, to support this technique, other data structures must be added. Furthermore, the time needed to perform the setup and administration of these data structures cannot be neglected.

The block_supernode_0 Algorithm

Figure 15 shows the first approach implementing the block supernode factorization technique. Obviously, the algorithms in this section consist of a double-nested loop construct compared to the three-level nesting of column supernode algorithms. The third level of nesting has not vanished but is hidden in the kernels *CRmod_d* and *CRmod_i*.

Figure 15 Algorithm: block_supernode_0

```
T_L ← 0     T_U ← 0    im ← 0
for J = 1 to N_S do
   setup vector im
      [T_L]_im ← A_*,j                                    (b0.1)
      [T_U]_im ← A_j,*
      for all K updating j do
         determine all j ∈ J being updated by K           (b0.2)
         CRmod_d(T_L, T_U, J, K)                          (b0.3)
         CRmod_i(T_L, T_U, im, J, K)
      end for
      for j ∈ J (in order) do                             (b0.4)
         CRmod_d(T_L, T_U, j, J)
         L_*,j ← L_*,j + T_L(j)     T_L(j) ← 0
         U_j,* ← U_j,* + T_U(j)     T_U(j) ← 0
         CRdiv(j)
   end for
end for
```

These kernels now consist of a double-nested loop where the inner loop remains the same as in Figure 10 on page 33; the outer loop usually runs over all nodes j being updated by a supernode K. (This node is sometimes split into nodes that can be updated densely and nodes that require indexed updating.)

The temporary vectors t_L and t_U had to be enlarged to hold a complete supernode.

Their counterparts in this section are denoted by T_L and T_U; both of length $(MAXCOL + 1)*MAXSUP$ where $MAXSUP$ holds the size of the largest system supernode. For each supernode being updated, T_L and T_U are loaded with the corresponding values from the coefficient matrix A (denoted $A_{*,j}/A_{j,*}$) using the index vector $imap$.

When this is finished, `block_supernode_0` determines the set of nodes j of supernode J, which are updated by supernode K (see b0.2). This set is formed by reverse scanning all column indices of supernode K and adding the corresponding node j of supernode J to the set. At the same time, the algorithm marks those nodes j, which can be computed using dense operations. Then, the dense and indexed updates are performed where the order of execution is merely implied by the underlying data structures (see b0.3).

After all supernodes K updating supernode J have been processed, supernode J needs to update *itself* (see b0.4). This is a dense operation involving each node of J. Loop b0.4 shows all operations necessary to complete the factorization of supernode J. Unfortunately, these operations cannot be applied to all nodes of J at the same time.

The block_supernode_1 Algorithm

In the `block_supernode_1` algorithm (see Figure 16 on page 40), an attempt has been made to increase computational efficiency by collecting the dense updates from all updating supernodes K and process them in one separate loop (see b1.1 and b1.2).

It is clear that this approach costs more in terms of both storage and computation to implement. As a result, this algorithm is only efficient if the amount of dense updates is (much) greater than the indexed one to trade off for the additional storage and computing overhead.

Figure 16 Algorithm: block_supernode_1

```
T_L ← 0    T_U ← 0    im ← 0
for J = 1 to N_S do
   set up vector im
      [T_L]_im ← A_*,J
      [T_U]_im ← A_J,*
      for all K updating j do
         determine all j ∈ J being updated by K      (b1.1)
         and collect dense updates
         CRmod_i(T_L, T_U, im, J, K)
      end for
      for all dense updates do                        (b1.2)
         CRmod_d(T_L, T_U, J, K)
      end for
      for j ∈ J (in order) do
         CRmod_d(T_L, T_U, j, J)
         L_*,j ← L_*,j + T_L(j)    T_L(j) ← 0
         U_j,* ← U_j,* + T_U(j)    T_U(j) ← 0
         CRdiv(j)
   end for
end for
```

The block_supernode_2 Algorithm

The `block_supernode_2` algorithm (see Figure 17 on page 41) is designed so that it does not need to perform any indexed updates. Primarily, the matrix elements of supernode J are stored into factor storage using the index map im (see b2.1). In the next loop over all updating supernodes K, first, another index vector ri is set up. Vector ri comprises the relative indices of supernode K's column structure in relation to supernode J's column structure. ri_k provides an offset from the bottom of a node j of J, which maps the k-th element of a node of K to the corresponding position within j. The index vector ri can, therefore, be regarded as a compact form of im applied to some supernode K updating J (see b2.2).

After ri is set up, the contribution of supernode *K* to the factorization of supernode J is accumulated as a dense operation in the temporary work arrays T_L and T_U as a dense operation. The result is then scattered and added into factor storage using ri (see b2.3 and b2.4; internally, the algorithm is more sophisticated at this point, since it knows which K shares the same sparsity pattern as J and then adds the contents of T_L and T_U with stride one).

Finally, the factorization of supernode J is completed by dense computations in factor storage (see b2.5). The algorithm is most efficient when there are only a few large supernodes updating another supernode. Otherwise, memory access penalties will decrease performance.

Figure 17 Algorithm: block_supernode_2

```
T_L ← 0    T_U ← 0    im_1 ← 0
for J = 1 to N_S do
   set up vector im
      [L_*,J]_im1 ← A_*,J                                    (b2.1)
      [U_J,*]_im1 ← A_J,*
      for all K updating J do
         determine all j ∈ J being updated by K              (b2.2)
         simultaneously setting up vector
         CRmod_d(T_L, T_U, J, K)                             (b2.3)
         [L_*,J]_ri ← [L_*,J]_ri + T_L    T_L ← 0
         [U_J,*]_ri ← [U_J,*]_ri + T_U    T_U ← 0            (b2.4)
      end for
      for j ∈ J (in order) do
         CRmod_d(L_*,J, U_J,*, j, J)                         (b2.5)
         CRdiv(j)
   end for
end for
```

The block_supernode_3 and block_supernode_4 Algorithms

The `block_supernode_3` algorithm is a variant of `block_supernode_2`. In this case, the second index map ri is omitted and indirect addressing is used explicitly (see b3.1). Furthermore, a modified version of the *CRmod_{d,i}* kernels is used.

Figure 18 Algorithm: block_supernode_3

```
T_L ← 0    T_U ← 0    im ← 0
for J = 1 to N_S do
   set up vector im
      [L_*,J]_im ← A_*,J
      [U_J,*]_im ← A_J,*
      for all K updating j do
         determine all j ∈ J being updated by K
         simultaneously set up vector ri
         CRmod_d(T_L, T_U, J, K)
         [L_*,J]_im_ind ← [L_*,J]_im_ind + T_L    T_L ← 0   (b3.1)
         [U_J,*]_im_ind ← [U_J,*]_im_ind + T_U    T_U ← 0
      end for
      for j ∈ J (in order) do
         CRmod_d(L_*,J, U_J,*, j, J)
         CRdiv(j)
   end for
end for
```

In the previously mentioned algorithms, the products $d_k * U_{k,j}$ and $d_k * L_{j,k}$ are precomputed immediately after setting up the index map im, and their results are stored in a temporary work space for later use. This has been changed for algorithms

`block_supernode_3` and `block_supernode_4`. Both algorithms use the kernels *CRmod_d* and *CRmod_i* as shown in Figure 10 on page 33. This leads to reduced memory requirements. Consequently, algorithms `block_supernode_3` and `block_supernode_4` use less space than the other block supernode algorithms.

Figure 19 Algorithm: block_supernode_4

```
T_L ← 0    T_U ← 0    im ← 0
for J = 1 to N_S do
   set up vector im
      [T_L]_im ← A_*,J
      [T_U]_im ← A_J,*
      for all K updating J do
         determine all j ∈ J being updated by K
         CRmod_i(T_L, T_U, im, J, K)                    (b4.1)
      end for
      for j ∈ J (in order) do
         CRmod_d(T_L, T_U, j, J)
         L_*,j ← L_*,j + T_L(j)      T_L(j) ← 0
         U_j,* ← U_j,* + T_U(j)      T_U(j) ← 0
         CRdiv(j)
   end for
end for
```

References

[1] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford: Clarendon Press, 1986.

[2] A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[3] S. C. Eisenstat *et al*., “The (New) Yale Sparse Matrix Package,” in *Elliptic Problem Solvers II* (Proceedings of the Elliptic Problem Solvers Conference), Monterey, CA, USA, pp. 45–52, January 1983.

[4] H. A. van der Vorst, *Lecture notes on iterative methods*, Utrecht, The Netherlands: University Utrecht, 1993.

[5] A. George and J. W. H. Liu, “The Evolution of the Minimum Degree Ordering Algorithm,” *SIAM Review*, vol. 31, no. 1, pp. 1–19, 1989.

[6] J. W. H. Liu, “Modification of the Minimum-Degree Algorithm by Multiple Elimination,” *ACM Transactions on Mathematical Software*, vol. 11, no. 2, pp. 141–153, 1985.

[7] A. Liegmann, *The Application of Supernodal Techniques on the Solution of Structurally Symmetric Systems*, Technical Report 92/5, Integrated Systems Laboratory, ETH, Zurich, Switzerland, 1992.

[8] E. G. Ng and B. W. Peyton, *A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors*, Technical Report ORNL/TM-11814, Oak Ridge National Laboratory, Oak Ridge, TN, USA, April 1991.

[9] E. Ng, *Supernodal Symbolic Cholesky Factorization on a Local-Memory Multiprocessor*, Technical Report ORNL/TM-11836, Oak Ridge National Laboratory, Oak Ridge, TN, USA, June 1991.

[10] J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, New York: Plenum Press, 1988.

[11] J. J. Dongarra *et al*., “An Extended Set of FORTRAN Basic Linear Algebra Subprograms,” *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 1–17, 1988.

[12] J. J. Dongarra *et al*., “A Set of Level 3 Basic Linear Algebra Subprograms,” *ACM Transactions on Mathematical Software*, vol. 16, no. 1, pp. 1–17, 1990.

[13] C. C. Ashcraft *et al*., “Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers,” *The International Journal of Supercomputer Applications*, vol. 1, no. 4, pp. 10–30, 1987.

[14] A. Liegmann and W. Fichtner, *The Application of Supernodal Factorization Algorithms for Structurally Symmetric Linear Systems in Semiconductor Device Simulation*, Technical Report 92/17, Integrated Systems Laboratory, ETH, Zurich, Switzerland, 1992.

[15] E. G. Ng and B. W. Peyton, *Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers*, Technical Report ORNL/TM-11960, Oak Ridge National Laboratory, Oak Ridge, TN, USA, December 1991.

[16] A. George, J. W. H. Liu, and E. Ng, “Communication results for parallel sparse Cholesky factorization on a hypercube,” *Parallel Computing*, vol. 10, no. 1, pp. 287–298, 1989.

[17] J. G. Lewis and H. D. Simon, “The Impact of Hardware Gather/Scatter on Sparse Gaussian Elimination,” *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 2, pp. 304–311, 1988.

[18] E. Rothberg and A. Gupta, “Techniques for Improving the Performance of Sparse Matrix Factorization on Multiprocessor Workstations,” in *Proceedings of Supercomputing ‘90*, New York, NY, USA, pp. 232–241, November 1990.

[19] E. Rothberg and A. Gupta, “Efficient Sparse Matrix Factorization on High-Performance Workstations—Exploiting the Memory Hierarchy,” *ACM Transactions on Mathematical Software*, vol. 17, no. 3, pp. 313–334, 1991.

[20] A. George, M. T. Heath, and J. Liu, “Parallel Cholesky Factorization on a Shared-Memory Multiprocessor,” *Linear Algebra and Its Applications*, vol. 77, pp. 165–187, 1986.

[21] D. S. Dodson, R. G. Grimes, and J. G. Lewis, “Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms,” *ACM Transactions on Mathematical Software*, vol. 17, no. 2, pp. 253–263, 1991.

[22] C. Pommerell, *Solution of Large Unsymmetric Systems of Linear Equations*, PhD thesis, ETH, Zurich, Switzerland, 1992.

[23] R. Schreiber, “A New Implementation of Sparse Gaussian Elimination,” *ACM Transactions on Mathematical Software*, vol. 8, no. 3, pp. 256–276, 1982.

[24] J. W. H. Liu, “The Role of Elimination Trees in Sparse Factorization,” *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.

Part III: ILS

This part contains information about the iterative linear solver ILS and is intended for users of Sentaurus Device, Sentaurus Process, and Sentaurus Interconnect:

- Chapter 4, Using ILS describes how to select ILS in Sentaurus Device, Sentaurus Process, and Sentaurus Interconnect, and how to control the parallel execution
- Chapter 5, Customizing ILS describes the parameters of ILS

4

Using ILS

The package ILS (iterative linear solver) is a library to solve sparse linear systems iteratively.

Features of the ILS Solver

ILS provides several iterative methods and different kinds of preconditioner. Recent techniques to reorder and scale the linear systems are used in the package to achieve good convergence results and high performance.

On shared-memory architectures, you can run ILS in parallel. Similar techniques to those in direct methods are used to achieve good accelerations. Parallelization of ILS is performed with OpenMP [1], which is an industry standard for parallel programming on shared-memory parallelization (SMP) systems. Most vendors of shared-memory architectures support OpenMP.

A parallel version of ILS is available on Red Hat Enterprise Linux (64-bit).

Multiple cores on machines that support hyperthreading are treated in the same way as multiple CPUs.

Selecting ILS in Sentaurus Device

You can activate ILS in Sentaurus Device by specifying:

```
Math {
   ...
   Method = Blocked
   SubMethod = ILS
   ILSrc = "
      set (...) {
      iterative (...);
      preconditioning (...);
      ordering (...);
      options (...);
   };
   ...
```

```
"
   WallClock
   ...
}
```

For single-device simulations only, you can specify `Method=ILS` instead of `Method=Blocked SubMethod=ILS`.

The keyword `WallClock` is used to print the wallclock times of the Newton solver. This is useful and recommended when investigating the performance of parallel execution.

ILS accepts options that can be specified in parentheses:

```
ILS (<options>)
```

Table 3 ILS options

Option	Description	Default
`MultipleRHS`	ILS solves linear systems with multiple right-hand sides. This option applies to AC analysis only. It might produce minor performance improvements or slightly more accurate results.	off
`Set=<integer>`	Uses the ILS options from the specified set.	1

ILSrc Statement

The optional `ILSrc` statement allows you to specify all ILS options within the `Math` section of Sentaurus Device. If the `ILSrc` statement is missing, Sentaurus Device uses the following built-in defaults:

```
set (1) { // default
   iterative (gmres(100), tolrel=1e-8, tolunprec=1e-4, tolabs=0,
     maxit=200);
   preconditioning (ilut(0.001,-1));
   ordering (symmetric=nd, nonsymmetric=mpsilst);
   options (compact=yes, refineresidual=0);
};
set (2) { // improved accuracy for AC analysis
   iterative (gmres(150), tolrel=1e-11, tolunprec=1e-8, tolabs=0,
     maxit=300);
   preconditioning (ilut(0.0001,-1), left);
   ordering (symmetric=nd, nonsymmetric=mpsilst);
   options (compact=yes, refineresidual=1);
};
set (3) { // for spherical harmonics expansion (SHE) distribution model
   iterative (gmres(150), tolrel=1e-11, tolunprec=1e-8, tolabs=0,
     maxit=150);
   preconditioning (ilut(0.0001,-1));
   ordering (symmetric=rcm, nonsymmetric=mpsilst);
```

```
   options (compact=yes, refinebasis=1);
};
set (4) { // for SHECoupled statement
   iterative (gmres(150), tolrel=1e-7, tolunprec=1e-4, tolabs=0,
     maxit=150);
   preconditioning (ilut(0.001,-1));
   ordering (symmetric=rcm, nonsymmetric=mpsilst);
   options (compact=yes, verbose=0, refinebasis=1, refineresidual=0);
};
set (5) { // for difficult 3D simulations (such as power devices)
   iterative (gmres(150), tolrel=1e-11, maxit=250);
   preconditioning (ilut(5e-5,-1), left);
   ordering (symmetric=nd, nonsymmetric=mpsilst);
   options (compact=yes, refineresidual=5);
};
set (6) { // for difficult 3D device simulations (stronger version
          // of set=5)
   iterative (gmres(150), tolrel=1e-11, maxit=250);
   preconditioning (ilut(2e-6,-1), left);
   ordering (symmetric=nd, nonsymmetric=mpsilst);
   options (compact=yes, refineresidual=30);
};
set (7) { // for 3D wide-bandgap simulations; for 2D, use ilut(5e-7,-1)
   iterative (gmres(150), tolrel=1e-10, maxit=200);
   preconditioning (ilut(5e-6,-1), right);
   ordering (symmetric=nd, nonsymmetric=mpsilst);
   options (compact=yes, refineresidual=30);
};
```

The parameters in set 1 give good results for most simulations.

Sets 1–19 are reserved for the built-in defaults. User-defined sets can be assigned to numbers 20 and higher.

If an `ILSrc` statement is specified in the `Math` section, it also must include the default sets as documented here.

Note:

It is not required to include build-in default sets in the `ILSrc` statement. A user-defined `ILSrc` statement will merge with the default sets, and both sets are available at runtime.

To improve the accuracy for AC analysis, set 2 is selected as follows:

```
Math {
   ACMethod = Blocked
   ACSubMethod = ILS (Set=2)
   ...
}
```

Parallel Execution

The number of threads for ILS can be specified in the `Math` section of the Sentaurus Device command file as follows:

```
Math {
   ...
   Number_of_Threads = 2
   Number_of_Solver_Threads = 2
   ...
}
```

The keyword `Number_of_Threads` defines the number of threads for both the matrix assembly and ILS, and `Number_of_Solver_Threads` defines only the number of threads for ILS itself. Instead of a constant number of threads, you can specify `maximum`. In this case, the number of threads is set equal to the number of processors available on the execution platform.

If no specification appears in the `Math` section, Sentaurus Device checks the values of the following UNIX environment variables (in order of decreasing priority):

```
SDEVICE_NUMBER_OF_SOLVER_THREADS

SDEVICE_NUMBER_OF_THREADS

SNPS_NUMBER_OF_THREADS

OMP_NUM_THREADS
```

For example, to obtain parallel execution with two threads, you can define `OMP_NUM_THREADS` as follows (in a C shell):

```
setenv OMP_NUM_THREADS 2
```

In a Bourne shell, the equivalent commands are:

```
OMP_NUM_THREADS=2
export OMP_NUM_THREADS
```

Selecting ILS in Sentaurus Process

You can enable ILS in Sentaurus Process by specifying the following commands for either diffusion simulations or mechanics simulations, respectively:

```
math diffuse dim=3 ils

math flow dim=3 ils
```

Use `dim=3` for 3D simulations or `dim=2` for 2D simulations.

You can set the parameters of the ILS solver using `pdbSet` commands. See *Sentaurus™ Process User Guide*, Setting Parameters of the Iterative Solver ILS.

The default set of ILS parameters used in Sentaurus Process is specified in the Parameter Database. These default parameters give good results for most simulations.

You can fine-tune some default parameters to improve convergence. In such cases, you should fine-tune the `ILS.ilut.tau` parameter, or the `ILS.gmres.restart` parameter, or both.

The `ILS.ilut.tau` parameter can be reduced, for example, from 2.e-3 (default for 3D diffusion) to 2e-4, all the way to 1e-5. You can increase the parameter `ILS.gmres.restart` from 60 to 100 (default is 60 for 3D diffusion). However, these two actions will increase memory use.

You can use the `pdbSet` command to activate the parameter `ILS.refine.sts`, which improves the convergence of the iterative mechanical solver STS3 in 3D simulations. The default value of `ILS.refine.sts` is 0, while the values 1 and 2 activate improvements made in Version H-2013.03 and Version I-2013.12, respectively. For example:

```
pdbSet Math Flow 3D ILS.refine.sts 2
```

Examples

```
pdbSet Math diffuse 3D ILS.ilut.tau 5e-5

pdbSet Math diffuse 2D ILS.ilut.tau 1e-5

pdbSet Math diffuse 3D ILS.gmres.restart 80
```

The number of threads must be specified in the `math` command. For example:

```
math numThreadsILS=2
```

For better ILS parallelization, you can specify the `pdbSet` command to activate the parameter `ILS.hpc.mode`, which is a high-performance computation mode that addresses multicore computers. This parameter helps to boost a parallel diffuse solver in Sentaurus Process when using many threads. The following values are available:

- The default value is 0 (no activation).
- A value of 1 activates algorithmic improvements made in Version E-2010.12.
- A value of 2 activates parallel improvements made in Versions F-2011.09 and G-2012.06.
- A value of 3 activates improvements made in Versions H-2013.03 and I-2013.12.
- A value of 4 activates algorithmic improvements made from Versions J-2014.09 to T-2022.03.

Note:
For Sentaurus Process, ILS no longer depends on the OpenMP environment variable `OMP_NUM_THREADS`, and you no longer need to specify this variable.

Example

```
pdbSet Math diffuse 3D ILS.hpc.mode 4
```

Selecting ILS in Sentaurus Interconnect

You can enable ILS in Sentaurus Interconnect by specifying the following commands for either solve steps or mechanics simulations, respectively:

```
math compute dim=3 ils

math flow dim=3 ils
```

Use `dim=3` for 3D simulations or `dim=2` for 2D simulations.

You can change the parameters of the ILS solver using `pdbSet` commands. See *Sentaurus™ Interconnect User Guide*, Setting Parameters of the Iterative Solver ILS.

The default set of ILS parameters used in Sentaurus Interconnect is specified in the Parameter Database. These default parameters give good results for most simulations.

You can fine-tune some default parameters to improve convergence. In such cases, you should fine-tune the `ILS.ilut.tau` parameter, or the `ILS.gmres.restart` parameter, or both.

The `ILS.ilut.tau` parameter can be reduced, for example, from 2.e-3 (default for 3D simulations) to 2e-4, all the way to 1e-5. You can increase the parameter `ILS.gmres.restart` to 120. However, these two actions will increase memory use.

Examples

```
pdbSet Math compute 3D ILS.ilut.tau 5e-5

pdbSet Math compute 2D ILS.ilut.tau 1e-5

pdbSet Math compute 3D ILS.gmres.restart 120
```

The number of threads must be specified in the `math` command. For example:

```
math numThreadsILS=4
```

For Sentaurus Interconnect, some ILS parameters have been tightened to provide better and faster convergence of iterative solvers (refer to the file `sinterconnect/sinterconnect/TclLib/SINTERCONNECT.models`):

```
pdbSet Math compute 1D ILS.refine.residual 3
pdbSet Math compute 2D ILS.refine.residual 2
pdbSet Math compute 3D ILS.refine.residual 2
pdbSet Math compute 1D ILS.ilut.tau 2.e-5
pdbSet Math compute 2D ILS.ilut.tau 5.e-5
pdbSet Math Flow    3D ILS.ilut.tau 1.0e-4
```

You can use the `pdbSet` command to activate the parameter `ILS.refine.sts`, which improves the convergence of the iterative mechanical solver STS3 in 3D simulations. The default value of `ILS.refine.sts` is `0`, while the values `1` and `2` activate improvements made in Version H-2013.03 and Version I-2013.12, respectively.

Example

```
pdbSet Math Flow 3D ILS.refine.sts 2
```

References

[1] L. Dagum and R. Menon, “OpenMP: An Industry-Standard API for Shared-Memory Programming,” *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

5

Customizing ILS

This chapter discusses the customization that is possible for ILS.

Configuration of ILS

In Sentaurus Process and Sentaurus Interconnect, the parameters of ILS are specified using `pdbSet` commands (see Chapter 4 on page 46).

In Sentaurus Device, the behavior of ILS is controlled using an `ILSrc` statement in the `Math` section. For example:

```
Math {
   Method = Blocked
   SubMethod = ILS
   ILSrc = "
      set (1) {
         iterative (gmres(100), tolrel=1e-8, tolunprec=1e-4, tolabs=0,
                    maxit=200);
         preconditioning (ilut(0.001,-1));
         ordering (symmetric=nd, nonsymmetric=mpsilst);
         options (verbose=0);
      };
   "
}
```

In ILS, the solution of a linear system consists of the following steps:

- Computation of a nonsymmetric ordering to improve the condition of the matrix
- Determination of a symmetric ordering to reduce the fill-in in the preconditioner
- Creation of a preconditioner to accelerate the convergence in the iterative method
- Calling an iterative method

ILS allows you to define sets of parameters. A configuration string defines one or more sets. Each set is identified with a number.

In Sentaurus Device, you can select a set with the following line in the command file:

```
Method = Blocked SubMethod = ILS (set = <integer>)
```

If a set is omitted, the number one (1) is taken as the default. The syntax of a set specification is:

```
set( <integer> ) {
   [ parent(<integer>); ]
   [ iterative(...) ]
   [ preconditioning(...) ]
   [ ordering(...) ]
   [ options(...) ]
};
```

where `<...>` represents a subspecification, `[...]` is an optional block, and a vertical bar (|) defines a choice. The meaning of `parent(i)` is that all the parameters of the set `i` are copied into the current set. This instruction can be used if two similar sets are specified, with only minor changes between them.

Note:

The source set must be defined beforehand and `parent` must be the first statement of a set.

General Remarks

The parser of the configuration string is case insensitive. You can add comments in the configuration string, as in a C++ or C source file, that is, text that follows `//` up to the end of the line is ignored. Text between `/*` and `*/` is disregarded.

Iterative Methods

Unsymmetric sparse linear systems can be solved with different Krylov subspace methods. The most famous methods are the bi-conjugate gradients stabilized (Bi-CGSTAB) method [1] and the generalized minimal residual (GMRES(m)) method [2], which are both implemented in ILS. Usually, they give the best results in terms of the number of iterations and the time to compute the solution. In semiconductor device simulations, GMRES demonstrates better reliability.

In addition, general iterative methods, CGS [3], BiCG×MR2 [4], and FGMRES(m) (FlexibleGMRES), are available.

Note:

In Sentaurus Device, GMRES(100) is the default iterative solver.

Sentaurus Process has additional special iterative methods:

- STCG2 and STS2 for solving 2D stress problems
- STCG3 and STS3 for solving 3D stress problems

Both STS2 and STS3, which are based on improved orderings and preconditioners, are recommended for mechanics simulations in Sentaurus Process.

Note:

In Sentaurus Process, GMRES(60) is the default iterative solver for 3D diffusion, and STS3 is the default for 3D stress problems.

In the GMRES(m) method, the parameter `m`, which is the number of backvectors, is required to limit the memory demands of the method. After `m` iterations, GMRES restarts. The default value `m` is 100 in Sentaurus Device and 50 in Sentaurus Process. Larger values of `m` usually help GMRES to converge, but at the expense of higher memory and execution time.

If you encounter convergence problems, decrease the threshold parameter `<eps>`, or increase the number of backvectors `m`, or do both. Conversely, for very large simulations, decrease `m` to fit the available memory of the computer.

Syntax

```
iterative( < bicgstab | bicgxmr2 | cgs | fgmres(<integer>) |
             gmres(<integer>) | stcg2 | stcg3 | sts2 | sts3 >
           [, tolrel = <double> ]
           [, tolabs = <double> ]
           [, tolunprec = <double> ]
           [, maxit = <integer> ] );
```

Stopping Criteria for Iterative Methods

Different stopping criteria are available for the iterative methods. If one of these is satisfied, the iterative method stops. The first criterion specifies the relative tolerance of the norm of the preconditioned residual, that is, the iteration stops if the norm of the preconditioned residual is reduced by `tolrel`. The second criterion checks whether the preconditioned residual becomes smaller than `tolabs`. The option `tolunprec` monitors the reduction of the unpreconditioned residual (the left preconditioned `gmres` controls only a preconditioned residual). This option makes sense only if the preconditioner is applied from the left. Otherwise, the unpreconditioned and preconditioned residuals are the same and, therefore, this option corresponds to the first one. You can use `maxit` to limit the number of iterations.

Table 4 Default values for stopping criteria of iterative methods

Option	Default
`maxit`	200
`tolabs`	0
`tolrel`	1e-8
`tolunprec`	1e-4

Example

```
iterative( gmres(100), tolrel=1e-8, tolunprec=1e-4, maxit=200 );
```

Preconditioners

Iterative methods are usually combined with preconditioners to improve convergence rates. Especially for ill-conditioned matrices, iterative methods fail without the application of a preconditioner. Different preconditioners exist in ILS, including a diagonal preconditioner and different incomplete LU factorizations.

Syntax

```
preconditioning( < none | diagonal | ilu0 | ilut(<double>, <integer>) >
                 [, < left | right > ] );
```

If you specify `none`, then the linear system is solved without a preconditioner. If a preconditioner is used, it can be applied from either the `left` (default) or `right`. In the former case, the unpreconditioned residuals and the preconditioned residuals do not correspond, but the error is the same for both the preconditioned and unpreconditioned linear system. In the latter case, the situation is reversed.

Example

```
preconditioning( ilut(0.001,-1), right );
```

Incomplete LU Factorization Preconditioners

Direct solvers for linear systems decompose a given matrix A into triangular factors L and U, whose product is equal to the original matrix, that is, $LU = A$. One of the main concerns of direct methods is the high demand of memory to perform the factorization. As the factors L and U are not computed exactly, but some elements are disregarded, it is more economical to work with them.

Several strategies have been proposed in the literature to determine which elements should be dropped or kept. In ILS, different incomplete LU factorizations are implemented: ILU(0) and ILUT(ε,*q*) (see Table 5). Parallel versions of both incomplete LU factorizations exist.

Table 5 Incomplete LU factorizations

Factorization	Description
ILU(0)	The simplest incomplete LU factorization, where all elements but the entries from the linear system are dropped.
ILUT(ε,*q*)	Incomplete LU factorization where the dropping of elements is based on the values. Elements smaller than ε are dropped during the elimination. The second parameter is intended to limit the number of elements in a row in the triangular factors, but this value is ignored. The smaller ε is, the more accurate the preconditioner becomes. However, the computation, memory requirements, and application of the preconditioner increase in this case.

Other Preconditioners

A simple diagonal preconditioner is also available in ILS. This preconditioner is equal to the inverse of the diagonal of the given matrix.

Nonsymmetric Ordering

The first step in the solution process of a linear system is the computation of a nonsymmetric ordering and scaling [5][6][7], such that the reordered and scaled system is better conditioned. Available options for this step are column orientated (`nonsymmetric=mpsilst`) (default), row oriented (`nonsymmetric=mpsils`), or omitting the step (`nonsymmetric=none`). The syntax to select nonsymmetric ordering is given in Symmetric Ordering.

Symmetric Ordering

As in direct methods, linear systems are reordered before the preconditioner is computed. The purpose of symmetric ordering is twofold. The quality of the preconditioner depends on the ordering. On the other hand, the ordering also influences the amount of fill-in in the preconditioners and, therefore, the time for the application of the preconditioner in the iterative method.

The following options are available in ILS:

- Reverse Cuthill–McKee (RCM) (`symmetric=rcm`) [8]
- Multiple minimum degree (MMD) (`symmetric=mmd`) [9]
- Multilevel nested dissection (ND) (`symmetric=nd`) [10]
- Combination of ND and RCM (NDRCM) (`symmetric=ndrcm`)

The ordering to be used depends on the preconditioner and an application. The best choice for an ILU(0) factorization is often RCM ordering [11][12]. For an incomplete LU factorization, where the dropping is entirely based on the numeric values (ILUT), ND and NDRCM orderings are preferable. The approximate inverse preconditioners are independent of a symmetric ordering and, therefore, this step can be omitted for these preconditioners.

In parallel mode, you must use either ND (default) or NDRCM, since these orderings allow for the parallel computation and application of incomplete LU factorizations. You can also use MMD for the parallel solver, but the performance is better using the other orderings.

Syntax

```
ordering ( [ nonsymmetric = < none | mpsils | mpsilst > ]
           [, symmetric = < none | mmd | nd | ndrcm | rcm > ] );
```

Example

```
ordering( nonsymmetric=mpsilst, symmetric=nd );
```

Additional Options

You should use the `compact` option if the linear system to be solved contains many entries that are numerically zero. Especially for simulations with Sentaurus Device, this option should be switched on. The default is `compact=yes`.

The option `refineresidual=m`, with a specified positive `m`, forces the GMRES method to perform `m` additional iterations and, on exiting, improves iteratively the final residual for the original (unpreconditioned or non-reordered) linear system. This option is recommended if you encounter convergence problems in Sentaurus Device or Sentaurus Process.

It is also useful in Sentaurus Process when diffusion steps converge in a few iterations. In such a situation, specifying additional `m=1,2` iterations might improve the accuracy of the solution. The default is `refineresidual=0`.

The option `refineiterate=1` is used to improve the final iteration, that is, the computed approximate solution of the original (unpreconditioned or non-reordered) linear system. This option differs from `refineresidual=m`, but it is recommended if you encounter

convergence problems in Sentaurus Device or Sentaurus Process. The default is `refineiterate=0`.

The option `refinebasis=1` forces a partial reorthogonalization in the GMRES method, helping to improve the orthonormality of the backvectors and to obtain a more accurate solution. It is recommended if device simulations have convergence problems. In typical cases, these extra refinements are not required. The default is `refinebasis=0`.

The option `prep=1` forces a fixed initial ordering of the linear system in Sentaurus Device, which helps in many cases to reduce the overall time spent in both symmetric and unsymmetric ordering steps. The default is `prep=0`.

The option `pdeg=1` activates a faster implementation of the preconditioner and backvector orthogonalization steps in the GMRES method. In addition, the option `pdeg=2` activates a second-degree ILUT preconditioner (the generic preconditioner applied twice per iteration), which might give the GMRES method better convergence with fewer iterations. The default is `pdeg=0`.

Note:

The options `prep` and `pdeg` apply only to Sentaurus Device.

You control the verbosity of ILS with the `verbose` option as follows:

- If `verbose=0`, then all output is suppressed.
- If `verbose=1`, then the accumulated numbers of calls, iterations, and execution times are printed to standard output.
- The most basic information is printed with `verbose=2`, which should be sufficient for the needs of most users.
- Higher values print additional information about the solution and preconditioners.

Syntax

```
options([ compact=<no | yes>
        [, refineresidual=<integer>]
        [, refineiterate=<integer>]
        [, refinebasis=<integer>]
        [, prep=<integer>]
        [, pdeg=<integer>]
        [, verbose=<integer>] ]);
```

Examples

```
options(compact=yes, verbose=1);

options( compact=yes, refineresidual=2, verbose=1 );
```

References

[1] H. A. van der Vorst, “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[2] Y. Saad and M. H. Schultz, “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[3] P. Sonneveld, “CGS, A Fast Lanczos-type Solver for Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, pp. 36–52, 1989.

[4] S. Röllin and M. H. Gutknecht, “Variations of Zhang’s Lanczos-type product method,” *Applied Numerical Mathematics*, vol. 41, pp. 119–133, 2002.

[5] O. Schenk, S. Röllin, and A. Gupta, “The Effects of Unsymmetric Matrix Permutations and Scalings in Semiconductor Device and Circuit Simulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 3, pp. 400–411, 2004.

[6] M. Benzi, J. C. Haws, and M. Tuma, “Preconditioning Highly Indefinite and Nonsymmetric Matrices,” *SIAM Journal on Scientific Computing*, vol. 22, no. 4, pp. 1333–1353, 2000.

[7] I. S. Duff and J. Koster, “On Algorithms for Permuting Large Entries to the Diagonal of a Sparse Matrix,” *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 4, pp. 973–996, 2001.

[8] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proceedings of the 24th National Conference,* ACM, New York, USA, pp. 157–172, August 1969.

[9] A. George and J. W. H. Liu, “The Evolution of the Minimum Degree Ordering Algorithm,” *SIAM Review*, vol. 31, no. 1, pp. 1–19, 1989.

[10] G. Karypis and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[11] M. Benzi, W. Joubert, and G. Mateescu, “Numerical Experiments with Parallel Orderings for ILU Preconditioners,” *Electronic Transactions on Numerical Analysis*, vol. 8, pp. 88–114, 1999.

[12] M. Benzi, D. B. Szyld, and A. van Duin, “Orderings for Incomplete Factorization Preconditioning of Nonsymmetric Problems,” *SIAM Journal on Scientific Computing*, vol. 20, no. 5, pp. 1652–1670, 1999.