Solvers User Guide

Version N-2017.09, September 2017

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.

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.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

About This Guide

This user guide provides information about the solvers that are available as part of Synopsys TCAD software. It is organized into the following parts:

- Part I PARDISO
- Part II SUPER
- Part III ILS

These solvers can be used with the Synopsys Sentaurus™ Device, Sentaurus Interconnect, and Sentaurus Process tools.

Related Publications

For additional information, see:

- The TCAD Sentaurus release notes, available on the Synopsys SolvNet® support site (see Accessing SolvNet on page vi).
- Documentation available on SolvNet at https://solvnet.synopsys.com/DocsOnWeb.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Blue text	Identifies a cross-reference (only on the screen).
Bold text	Identifies a selectable icon, button, menu, or tab. It also indicates the name of a field or an option.
`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 SolvNet customer support website and by contacting the Synopsys support center.

Accessing SolvNet

The SolvNet 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 SolvNet site:

1. Go to the web page at https://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register.)

If you need help using the site, click **Help** on the menu bar.

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

Contacting Your Local TCAD Support Team Directly

Send an e-mail 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.

METIS is a software package for unstructured graph partitioning and sparse matrix orderings. It was developed by G. Karypis and V. Kumar, Department of Computer Science, University of Minnesota (karypis,kumar@cs.umn.edu), and is copyrighted by the regents of the University of Minnesota (http://glaros.dtc.umn.edu/gkhome/views/metis).

The TCAD Sentaurus software uses the LAPACK linear algebra package, which requires the following copyright notice:

ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The TCAD Sentaurus software uses the ATLAS (Automatically Tuned Linear Algebra Software) package, which requires the following copyright notice:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the ATLAS group or the names of its contributors may not be used to endorse or promote products derived from this software without specific written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE ATLAS GROUP OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part I PARDISO

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

Chapter 1 Using PARDISO on page 3 provides background information on PARDISO.

CHAPTER 1 Using PARDISO

PARDISO [1][2] 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 can be used as a serial package, or in a shared-memory multiprocessor environment as an efficient, scalable, parallel, direct solver.

Algorithms

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

- Nonsymmetric matrix permutation and scaling – This places large matrix entries on the diagonal.
- Ordering – This determines the permutation of the coefficient matrix A such that the factorization incurs low fill-in.
- Numeric factorization – This 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.
- Solution of triangular systems – This produces the solution by performing forward and backward eliminations.

The nonsymmetric matrix permutation and scaling 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].

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 that is integrated in PARDISO is substantially better than the multiple minimum degree algorithm for large problem sizes. This applies especially to three-dimensional problems.

PARDISO exploits the memory hierarchy of the architecture by using the clique structure of the elimination graph by supernode algorithms, thus 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.

The combination of block techniques, parallel processing, and global fill-in reduction methods for three-dimensional 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 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`, type the command:

```
limit
```

or, in `bash` or `sh`, type 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 typing the command:

```
ulimit -s unlimited
```

PARDISO is tuned for general use in Sentaurus Device and Sentaurus Process. This means that user intervention is not necessary.

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, it is also possible to specify `Method = Pardiso` instead of `Method = Blocked SubMethod = Pardiso`.

PARDISO accepts options that can be specified in parentheses: `Pardiso (<options>)`. Table 1 lists the available 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 may 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:

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

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

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, whereas `Number_of_Solver_Threads` only defines the number of threads for PARDISO itself. Instead of a constant number of threads, it is possible to 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 will check 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 the environment variable `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.

By default for Sentaurus Process, PARDISO uses the multiple minimum degree (MMD) ordering in 2D and the nested dissection (ND) ordering in 3D. The ordering can be changed using the following commands, which specify ND ordering and MMD ordering, respectively:

```
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
```

By default for Sentaurus Interconnect, PARDISO uses the multiple minimum degree (MMD) ordering in 2D and the nested dissection (ND) ordering in 3D. The ordering can be changed using the following commands, which specify ND ordering and MMD ordering, respectively:

```
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*, Ph.D. 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 chapters regarding the direct linear solver SUPER and is intended for users of Sentaurus Device:

Chapter 2 Using SUPER on page 13 provides background information on SUPER.

Chapter 3 Customizing SUPER on page 15 describes the `.superrc` file, which is used to customize SUPER.

Chapter 4 Implementing SUPER on page 19 discusses the algorithms used in SUPER.

CHAPTER 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.

Overview

It is a fast direct solver for the multidimensional semiconductor device simulator Sentaurus Device, where the solution of structurally symmetric sparse 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 idea 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 $\eta(k+i) = \eta(k+r) + r - i,\ i = 0 \ldots r$, 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 illustrates a supernode.

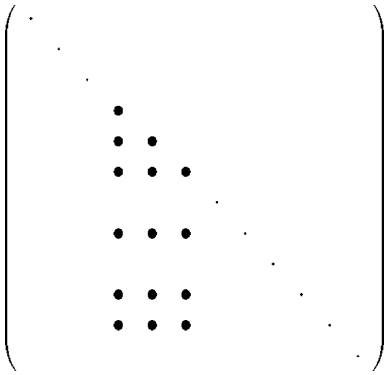

Figure 1 Example of a supernode

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]. Additionally, 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 the 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 contains a set of nine supernodal factorization methods that provide excellent performance on both RISC and vector machines.

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

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.

CHAPTER 3 Customizing SUPER

This chapter discusses the customization that is possible for 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 name 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:

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

SUPER uses default settings if no configuration file is found.

In this section, the grammar of the input language is presented. 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 techniques. Generally, two types of supernodal approaches are available: column supernode and block supernode (see Sparse Supernodal Factorization Algorithms on page 27).

SUPER contains four versions of the column supernode approach and five versions of the block supernode approach. In terms of memory consumption, column supernode methods 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 approaches should be considered as 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 file(s) is to 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 (`blsmp` or `matlab`). If the `blsmp` format is selected, the matrix (the right-hand side) and the solution of the linear system are written to the file `nsuper_blsmp_real_index.txt` or the file `nsuper_blsmp_complex_index.txt`. If the `matlab` format is selected, the output is sent to the file `nsuper_matlab_real_index.m` or `nsuper_matlab_complex_index.m`. 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.

An example of a .superrc file is:

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

These settings in the .superrc file instruct SUPER to use the factorization algorithm block_supernode_4. The write statement instructs SUPER to generate ASCII files, in blsmp format, of the fifth and ninth linear systems solved.

CHAPTER 4 Implementing SUPER

This chapter describes the algorithms in SUPER.

Overview

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

Technically, the solution process of SUPER has six 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]. The phases are:

- Structure input
- Reordering
- Symbolic factorization
- Numeric value input
- Numeric factorization
- Numeric solution

During *structure input*, 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 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 may want to retain the nonzero structure of the coefficient matrix in its factors or at least 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]. In this solver, an enhanced minimum degree algorithm called the multiple minimum degree (MMD) algorithm is used [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 were eliminated simultaneously, hereby producing supernodes as a side effect [8]. How Multiple Minimum Degree (MMD) Works on page 22 presents the MMD algorithms in detail.

When the coefficient matrix is reordered, it is desirable 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.

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

The *numeric factorization* is the most time-consuming part 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 (also referred to as supernode–node and supernode–supernode, respectively) algorithms are implemented. Both methods are schematically depicted in Figure 2 on page 21.

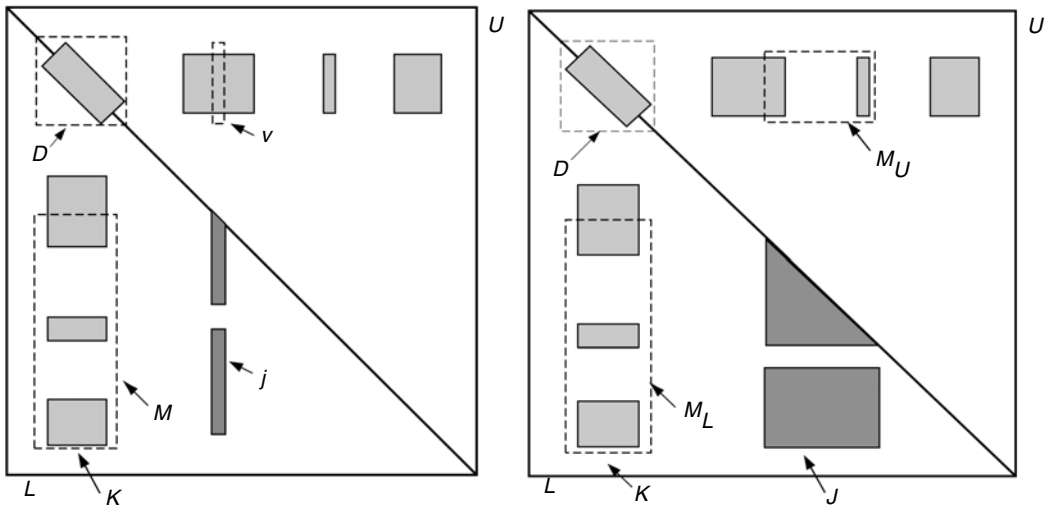

Figure 2 Illustration of supernode column (*left*) and block supernode updating (*right*)

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 mathematically expressed in the term:

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

also 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 need to 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:

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

representing a DGEMM operation [12]. Block supernode methods mainly involve dense matrix–matrix multiplications, hereby reducing memory traffic. Analogous to column supernode methods, 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 the decomposition. Sparse Supernodal Factorization Algorithms on page 27 describes all supernodal algorithms implemented in SUPER.

The final step in the solution process is the *numeric solution* phase. 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 (MMD) 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[1]

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[2].)

As previously mentioned, 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 minimum degree are eliminated simultaneously. For indistinguishable nodes, it can be shown that they are eliminated consecutively when MD is used.

Algorithm 1 on page 23 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 minimum degree. 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.

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

1. The concept of indistinguishable nodes is covered extensively in the literature [2].
2. Practically, this defines the term *clique* where all nodes are connected to each other.
3. MA27 (Harwell Laboratories), SPARSPAK (University of Waterloo), and YSMP (Yale University).

After an element $y \in T$ is chosen, the algorithm determines the set Y that contains all elements of T indistinguishable from y [1]. 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 $S = V$.

```
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
```

Algorithm 1 Multiple minimum degree (MMD) algorithm

1. Element y is trivially indistinguishable from itself.

Example: MMD Execution

Figure 3 provides the symmetric pattern of the matrix A where '•' denotes a nonzero entry.

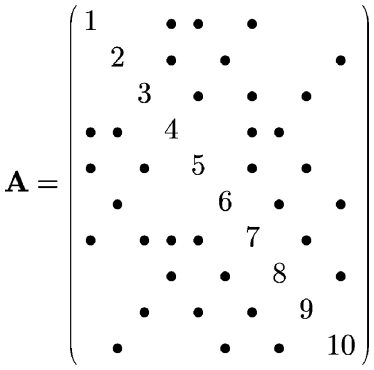

Figure 3 Sample sparse matrix *A*

Figure 4 illustrates the graph representation of A .

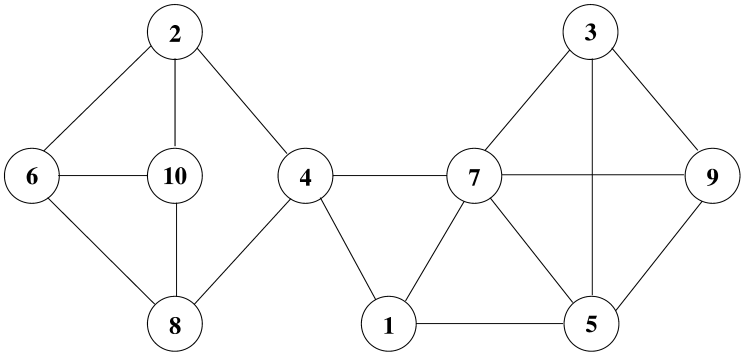

Figure 4 Graph representation of sample matrix *A*

The numbering in the graph is equal to the line numbering of the matrix. The initial minimum degree of the graph is 3[1]. Therefore, the ordering algorithm starts with:

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

1. Self-loops are neglected.

Now, $y = 10$ is chosen from T. The only indistinguishable node from $y = 10$ is the node with the number 6, yielding $Y = \{(10, 6)\}$ [1]. 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:

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

The 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 5 shows the graph representation of the remaining nodes all of which had their degree updated because they were all flagged.

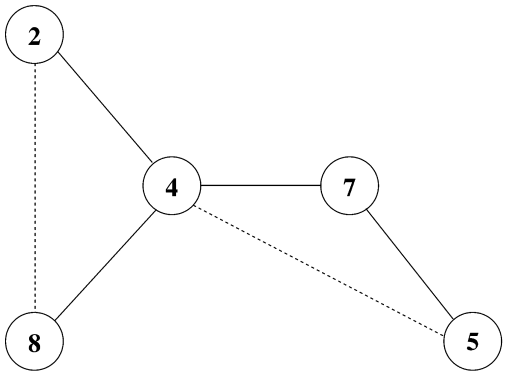

Figure 5 Elimination graph after first loop through multiple mass elimination

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:

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

1. Parentheses are only used to identify groups of indistinguishable nodes.

Applying this permutation to the matrix A results in the structure shown in Figure 6.

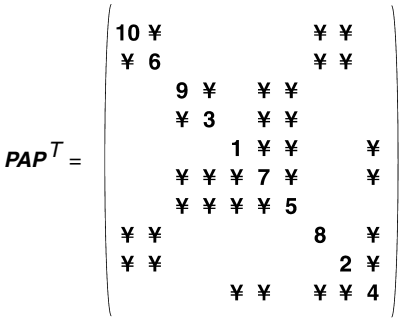

Figure 6 Sample matrix *A* reordered with MMD

Performing symbolic factorization on this matrix reveals the sparsity pattern of the factor L, which is depicted in Figure 7 where the columns have been renumbered.

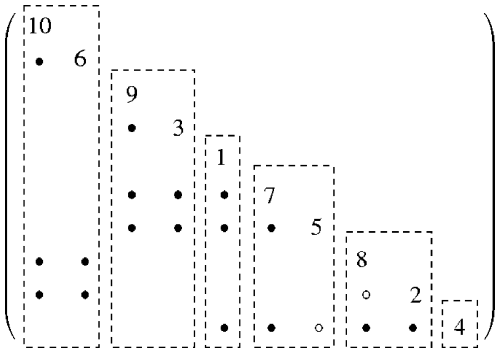

Figure 7 Sparsity structure of factor *L* of *A*

NOTE The sparsity structures for PAP^T and L are similar; L has two additional nonzero fill-in entries (indicated by 'o'). In addition, L 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]. Supernodes play an important role in improving the performance of the numeric factorization. SUPER is focused 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, whereas the second type performs so-called indexed SAXPY or SAXPYI operations [13][17].

Additionally, to update updating a column j by a supernode S requires one gather and one scatter operation, whereas 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 can 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 following subsections describe several supernodal factorization algorithms implemented in SUPER. These algorithms fall into two different classes: column and block supernode update schemes. Table 2 lists the symbols used here.

Table 2 List of symbols

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
$A_{*,J}, A_{J,*}$	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

Table 2 List of symbols

Symbol	Description
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

Supernode–node 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 may consist of several columns or rows. Algorithm 2 lists the first algorithm implementing this technique.

```
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
```

Algorithm 2 column_supernode_0

Initially, the algorithm reveals the general form of supernode–node updating 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_{i,d}* and *CRdiv* (see Algorithm 3 and Algorithm 4 on page 30, and Algorithm 5 on page 30) are typical for LU decomposition methods [8][20].

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 Algorithm 2 on page 28). 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.

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[1].

1. 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.

```
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
```

Algorithm 3 CRmod_d kernel

```
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
```

Algorithm 4 CRmod_i kernel

```
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
```

Algorithm 5 CRdiv kernel

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

Algorithm 6 Setup of vector *im*

Algorithm 7 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.

```
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
```

Algorithm 7 column_supernode_1

Instead of occupying space for $2*n$ real numbers, algorithm `column_supernode_1` needs only $2*(MAXCOL+1)$ [1] where $MAXCOL$ denotes the maximal number of nonzero entries in a column of L excluding the diagonal element. In 2D and 3D device simulation, where n is typically greater than 5000, $MAXCOL$ [2] is much smaller than n [22].

Conversely, we use a technique called relative indexing [15][23] so that algorithm `column_supernode_1` can use smaller temporary vectors. Relative indexing introduced an additional vector im [3] of length n (c1.1). Nevertheless, the total amount of overhead storage required for algorithm `column_supernode_1` is approximately 60% of that used in `column_supernode_0`.

1. $MAXCOL + 1$ is needed here to hold the diagonal element of the current column.
2. Experimental results revealed $MAXCOL$ to be less than 10% of n in 2D device simulation.
3. Acronym for *index map*.

Algorithm 6 on page 30 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.

Next, algorithm `column_supernode_2` (see Algorithm 8 on page 33) is introduced, which implements a major change compared to algorithm `column_supernode_1` dealt with previously. 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 Algorithm 7 on page 31). Consequently, algorithm `column_supernode_2` uses the least amount of memory of all algorithms considered in this section.

```
im ← 0
for J = 1 to N_S do
   setup vector im
   for j ∈ J (in order) do
      [L_{*,j}]_im ← A_{*,j}
      [U_{j,*}]_im ← A_{j,*}                                   (c2.1)
      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
```

Algorithm 8 column_supernode_2

Algorithm 9 on page 34 shows another variant of column supernode LU factorization. This algorithm requires the same amount of storage overhead as algorithm `column_supernode_1`, 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 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 have to be merged into factor storage. The advantage over the other algorithms is assumed to unfold in the fact that we can compute supernode K's contribution updating column j as a dense SAXPY operation (see c3.2), therefore revealing the second major difference mentioned above.

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 price for being able to use dense SAXPY operations to calculate t_L and t_U. Experiments with real device simulation test cases have shown that the 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.

```
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}
      [U_{j,*}]_{im} ← A_{j,*}                                   (c3.1)
      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
            [U_{j,*}]_{im} ← t_U      t_U ← 0                    (c3.3)
         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
```

Algorithm 9 column_supernode_3

Looking at all the supernode–node updating algorithms previously discussed reveals that, in all cases, dense updates and column/row scaling are treated equally. Thus, we 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 `column_supernode_1` to the others. In the next section, we reduce the number of scatter/ gather operations 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 by any means, but by grouping them together, memory fetch and store can be made more efficient, that is, using the same index map only once throughout a loop cycle. In addition, in terms of vectorization, supernode–supernode updating does not make the vectorizable loops longer, thus 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, supernode–supernode factorization increases storage overhead considerably, since the intermediate results for more than one column or row must be retained and, in order to support this technique, other data structures must be added. Furthermore, the time necessary to perform the setup and administration of these data structures cannot be neglected.

Algorithm 10 (`block_supernode_0`) shows a first approach implementing this block supernodal factorization technique. Obviously, the algorithms in this section consist of a double-nested loop construct compared to the three-level nesting of supernode–node algorithms. The third level of nesting has not vanished, but is hidden in the kernels *CRmod_d* and *CRmod_i*.

$T_L \leftarrow 0 \qquad T_U \leftarrow 0 \qquad im \leftarrow 0$
for $J = 1$ **to** N_S **do**
  set up vector im
  $[T_L]_{im} \leftarrow A_{*,J}$
  $[T_U]_{im} \leftarrow A_{J,*}$ (b0.1)
  for all K updating J **do**
    determine all $j \in J$ 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 $j \in J$ (in order) **do** (b0.4)
    $CRmod\_d(T_L, T_U, j, J)$
    $L_{*,j} \leftarrow L_{*,j} + T_L(j) \qquad T_L(j) \leftarrow 0$
    $U_{j,*} \leftarrow U_{j,*} + T_U(j) \qquad T_U(j) \leftarrow 0$
    $CRdiv(j)$
  end for
end for

Algorithm 10 block_supernode_0

These kernels now consist of a double-nested loop where the inner loop remains the same as in Algorithm 3 and Algorithm 4 on page 30; the outer loop usually runs over all nodes j being updated by a supernode K[1]. 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.

In Algorithm 11 on page 37 (`block_supernode_1`), an attempt was 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.

1. This node is sometimes split into nodes that can be updated densely and nodes that require indexed updating.

```
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
```

Algorithm 11 block_supernode_1

Algorithm 12 on page 38 (`block_supernode_2`) 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)[1].

1. 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.

$T_L \leftarrow 0 \qquad T_U \leftarrow 0 \qquad im_1 \leftarrow 0$
for J = 1 **to** N_S **do**
  set up vector im
  $[L_{*,J}]_{im_1} \leftarrow A_{*,J}$
  $[U_{j,*}]_{im_1} \leftarrow A_{J,*}$ (b2.1)
  for all K updating J **do**
    determine all $j \in J$ being updated by K (b2.2)
    simultaneously setting up vector
    $CRmod\_d(T_L, T_U, J, K)$ (b2.3)
    $[L_{*,J}]_{ri} \leftarrow [L_{*,J}]_{ri} + T_L \qquad T_L \leftarrow 0$
    $[U_{J,*}]_{ri} \leftarrow [U_{J,*}]_{ri} + T_U \qquad T_U \leftarrow 0$ (b2.4)
  end for
  for $j \in J$ (in order) **do**
    $CRmod\_d(L_{*,J}, U_{J,*}, j, J)$ (b2.5)
    $CRdiv(j)$
  end for
end for

Algorithm 12 block_supernode_2

Algorithm 13 on page 39 (`block_supernode_3`) 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. In the algorithms previously presented, 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` (see Algorithm 14 on page 39). Both algorithms use the kernels *CRmod_d* and *CRmod_i* as they are depicted in Algorithm 3 on page 30. This leads to reduced memory requirements. Consequently, algorithms `block_supernode_3` and `block_supernode_4` use less space than the previously presented block supernode algorithms.

$T_L \leftarrow 0 \qquad T_U \leftarrow 0 \qquad im \leftarrow 0$
for $J = 1$ **to** N_S **do**
  set up vector im
  $[L_{*,J}]_{im} \leftarrow A_{*,J}$
  $[U_{j,*}]_{im} \leftarrow A_{J,*}$
  for all K updating J **do**
    determine all $j \in J$ being updated by K
    simultaneously setup vector ri
    $CRmod\_d(T_L, T_U, J, K)$
    $[L_{*,J}]_{im_{ind}} \leftarrow [L_{*,J}]_{im_{ind}} + T_L \qquad T_L \leftarrow 0$
    $[U_{J,*}]_{im_{ind}} \leftarrow [U_{J,*}]_{im_{ind}} + T_U \qquad T_U \leftarrow 0$ (b3.1)
  end for
  for $j \in J$ (in order) **do**
    $CRmod\_d(L_{*,J}, U_{J,*}, j, J)$
    $CRdiv(j)$
  end for
end for

Algorithm 13 block_supernode_3

$T_L \leftarrow 0 \qquad T_U \leftarrow 0 \qquad im \leftarrow 0$
for $J = 1$ **to** N_S **do**
  set up vector im
  $[T_L]_{im} \leftarrow A_{*,J}$
  $[T_U]_{im} \leftarrow A_{J,*}$
  for all K updating J **do**
    determine all $j \in J$ being updated by K
    $CRmod\_i(T_L, T_U, im, J, K)$ (b4.1)
  end for
  for $j \in J$ (in order) **do**
    $CRmod\_d(T_L, T_U, j, J)$
    $L_{*,j} \leftarrow L_{*,j} + T_L(j) \qquad T_L(j) \leftarrow 0$
    $U_{j,*} \leftarrow U_{j,*} + T_U(j) \qquad T_U(j) \leftarrow 0$
    $CRdiv(j)$
  end for
end for

Algorithm 14 block_supernode_4

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*, Ph.D. 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 chapters regarding the iterative linear solver ILS and is intended for users of Sentaurus Device, Sentaurus Process, and Sentaurus Interconnect:

Chapter 5 Using ILS on page 45 describes how to select ILS in Sentaurus Device, Sentaurus Process, and Sentaurus Interconnect, and how to control the parallel execution.

Chapter 6 Customizing ILS on page 51 describes the parameters of ILS.

CHAPTER 5 Using ILS

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

Overview

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

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 enable ILS in Sentaurus Device by specifying:

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

```
    WallClock
    ...
}
```

For single-device simulations only, it is also possible to specify `Method=ILS` instead of `Method=Blocked SubMethod=ILS`.

ILS accepts options that can be specified in parentheses: `ILS (<options>)`. Table 3 lists the available 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 may produce minor performance improvements or slightly more accurate results.	off
`Set=<integer>`	Uses the ILS options from the specified set.	1

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, verbose=0, 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, verbose=0, refineresidual=1);
};

set (3) { // for 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, verbose=0, 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);
};
```

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

Sets 1–9 are reserved for the built-in defaults. User-defined sets can be assigned to numbers 10 and higher.

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

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

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

The keyword `WallClock` can be used to print the wallclock times of the Newton solver. This is useful and recommended when investigating the performance of 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, whereas `Number_of_Solver_Threads` only defines the number of threads for ILS itself. Instead of a constant number of threads, it is possible to 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 will check 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 the environment variable `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 the `pdbSet` commands. For details, see Sentaurus™ Process User Guide, Setting Parameters of the Iterative Solver ILS on page 884.

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 the convergence. In such cases, it is recommended to 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 (the default value for 3D diffusion) to 2e-4, all the way to 1e-5. You can increase the parameter `ILS.gmres.restart` from 60 to 100 (the default value 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` – 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 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 in Version J-2014.09, K-2015.06, L-2016.03, M-2016.12, and N-2017.09:

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

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.

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 the `pdbSet` commands. For details, see Sentaurus™ Interconnect User Guide, Setting Parameters of the Iterative Solver ILS on page 330.

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 the convergence. In such cases, it is recommended to 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 (the default value 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 Sentaurus Interconnect 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
```

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.

CHAPTER 6 Customizing ILS

This chapter discusses the customization that is possible for ILS.

Configuration

In Sentaurus Process and Sentaurus Interconnect, the parameters of ILS can specified using the `pdbSet` commands (see Chapter 5 on page 45).

In Sentaurus Device, the behavior of ILS can be controlled with 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 four 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.

For each step, there are several options, which are described in the following sections. 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 default. The syntax of a set specification is:

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

where `<...>` represents a subspecification, `[...]` is an optional block, and '`|`' defines a choice. The meaning of `parent(i)` is that all of 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` should be the first statement of a set.

A description of the four other blocks is given in the following sections.

Nonsymmetric Ordering

The first step in the solution process of a linear system is the computation of a nonsymmetric ordering and scaling [1][2][3], such that the reordered and scaled system is better conditioned. The three different options for this step are the default version is column oriented (MPSILST), the second version is row oriented (MPSILS), and the third possibility is to omit the nonsymmetric ordering by specifying `none`. The syntax to select the nonsymmetric ordering is given in the next section, Symmetric Ordering.

Symmetric Ordering

As in direct methods, the linear systems are reordered before the preconditioner is computed. The purpose of the 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 orderings are available in ILS:

- Reverse Cuthill–McKee (RCM) [4]
- Multiple minimum degree (MMD) [5]
- Multilevel nested dissection (ND) [6]
- A combination of ND and RCM (NDRCM)

The ordering to be used depends on the preconditioner and an application. The best choice for an ILU(0) factorization is often the RCM ordering [7][8]. For an incomplete LU factorization, where the dropping is entirely based on the numeric values (ILUT), the 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, it is mandatory to use either ND (default) or NDRCM, since these orderings allow for the parallel computation and application of incomplete LU factorizations. It is also possible to use MMD for the parallel solver, but the performance is better using the other orderings.

The syntax is:

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

Example

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

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. Several preconditioners exist in ILS, from simple techniques such as a diagonal preconditioner, over different incomplete LU factorizations, to sparse approximate inverse preconditioners.

An overview of the syntax to select a preconditioner is presented and the various possibilities are described.

The syntax is:

```
preconditioning( < none | diagonal | ilu0 |
                 ilut( <double>, <integer> ) |
                 spai0 | spai1 |
                 spai( <double>, <integer>, <integer>, <integer> ) >
                 [, < left | right > ] );
```

If `none` is specified, the linear system is solved without a preconditioner. If a preconditioner is used, it can be applied from either the `left` (default) or `right` by specifying the according option. 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 Factorizations

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, three different incomplete LU factorizations are implemented: ILU(0), ILUT(ε,q), and ILUPT(p,ε). They are described in Table 4. Parallel versions of the first two incomplete factorizations exist. ILUPT is currently not parallelized.

Table 4 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 currently this value is ignored. The smaller ε is, the more accurate the preconditioner becomes. However, the computation, memory requirements, and application of the preconditioner is increasing in this case.
ILUPT(p,ε)	A combination of ILU(p) (generalization of ILU(0)) and ILUT. Increasing p or lowering ε improves the accuracy of the preconditioner, but the same consequences as for lowering ε in ILUT(ε,q) hold. Note that the parameters p and ε are reversed in the configuration string.

Sparse Approximate Inverses

These preconditioners approximate directly the inverse of the given linear system. Three different versions exist in ILS: SPAI(0), SPAI(1), and SPAI(ε) [9]. The difference between these preconditioners is their structure. The first consists solely of a diagonal, the second has the same structure as the given linear system, and the structure of the third one is computed dynamically during the computation of the approximation.

The implementation of SPAI(ε) requires four arguments: `spai(epsilon, bs, ns, mn)`. They are described in Table 5. These preconditioners have the advantage that they can be computed and applied in parallel. However, their quality is not good enough to use them for semiconductor device simulations. For this reason, they are currently not available as parallel versions.

Table 5 Parameters for SPAI(ε)

Parameter	Description
`bs`	Block size to use.
`epsilon`	Threshold to limit fill-in.
`mn`	Maximum number of columns to add during one improvement step.
`ns`	Number of improvement steps.

Other Preconditioners

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

Iterative Methods

Unsymmetric sparse linear systems can be solved with different Krylov subspace methods.

The most famous methods are the biconjugate gradients stabilized (BICGSTAB) method [10] and the generalized minimal residual (GMRES(m)) method [11], 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.

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

Three additional general iterative methods, CGS [12], BiCGxMR2 [13], and FGMRES(m) (FlexibleGMRES), are available (use the keyword `bicgxmr2` to select the second one). Additionally for Sentaurus Process, special iterative methods, STCG2 and STS2 for solving 2D stress problems, as well as STCG3 and STS3 for solving 3D stress problems, are available. The stress solvers STS2 and STS3, which are based on improved orderings and preconditioners, are recommended for mechanics simulations in Sentaurus Process.

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

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 is restarted. The default value `m` is 100 in Sentaurus Device, and `m` is 50 in Sentaurus Process. Larger values of `m` usually help GMRES to converge, but at the expense of higher memory and execution time.

If there are convergence problems, it is recommended to decrease the threshold parameter `<eps>` or to increase the number of backvectors `m` or both. Conversely, in the case of huge simulations, `m` can be decreased to fit the available memory of the computer.

The syntax is:

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

Different stopping criteria are available for the iterative methods. If one of these is satisfied, the iterative method stops. The first possibility is to specify 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 if the preconditioned residual becomes smaller than `tolabs`. With the option `tolunprec`, the reduction of the unpreconditioned residual can be monitored (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. A limit of the number of iterations can be specified with `maxit`.

Table 6 lists the default values for the different stopping criteria.

Table 6 Default values for stopping criteria for iterative methods

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

Example

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

Options

There are additional options that you can specify for the linear solver. One of these should be used 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 of this option is `compact=yes`.

The option `refineresidual=m` with a specified positive `m` will force the GMRES method to perform `m` additional iterations and, on exiting, will improve iteratively the final residual for the original (unpreconditioned/non-reordered) linear system. This option may be recommended if there are convergence problems in Sentaurus Device or Sentaurus Process.

It is also useful in Sentaurus Process when diffusion steps converge in a few (for example, 2–3) iterations. In such a situation, specifying additional `m=1,2` iterations may improve the accuracy of the solution. The default of this option is `refineresidual=0`.

The option `refineiterate=1` is used to improve the final iteration, that is, the computed approximate solution of the original (unpreconditioned/non-reordered) linear system. This option differs from the option `refineresidual=m`, but it is recommended if there are convergence problems in Sentaurus Device or Sentaurus Process. The default of this option is `refineiterate=0`.

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

The verbosity of ILS is controlled with the option `verbose`. With a value of `0`, all output is suppressed. If a value of `1` is specified, the accumulated numbers of calls, iterations, and execution times are printed to standard output. The most basic information is printed with `verbose=2` and this should be sufficient for the needs of most users. Higher values print additional information about the solution and preconditioners.

The syntax is:

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

Examples

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

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

General Remarks

The parser of the configuration string is case insensitive. Comments can be made 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[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] 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.

[7] 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.

[8] 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.

[9] M. J. Grote and T. Huckle, "Parallel Preconditioning with Sparse Approximate Inverses," *SIAM Journal on Scientific Computing*, vol. 18, no. 3, pp. 838–853, 1997.

[10] 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.

[11] 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.

[12] 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.

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