# TAUCS A Library of Sparse Linear Solvers

Sivan Toledo
School of Computer Science
Tel-Aviv University
stoledo@tau.ac.il
http://www.tau.ac.il/~stoledo/taucs

4th September 2003

With contributions by:

| | |
|---|---|
| Doron Chen | Maximum-weight-basis and Vaidya preconditioners |
| Vladimir Rotkin | Out-of-core sparse Cholesky factorization |
| Omer Meshar | New configuration system |

This document is the user manual for version 2.2 of Taucs. Version 2.2 is the first to support multithreading.
The main new innovations in Version 2.1 were a new build and configuration system, and a unified interface to all the linear solvers. Smaller innovations in 2.1 include compilation with no warnings on several platforms, and out-of-the-box builds for Windows and MacOS (in addition to Linux, Irix, and Solaris, which were already well supported).
Version 2.0 was the first version to support both real and complex data type (both in single and double precisions). As a consequence, the interfaces to subroutines this version are somewhat different than in version 1.0.

## Contents

# 1 Preliminaries

## 1.1 Introduction

TAUCS is a C library of sparse linear solvers. The current version of the library includes the following functionality:

**Multifrontal Supernodal Cholesky Factorization.** This code is quite fast (several times faster than MATLAB 6's sparse Cholesky). It uses the BLAS and LAPACK to factor and compute updates from supernodes. It uses relaxed and amalgamated supernodes. This routine is multithreaded.

**Left-Looking Supernodal Cholesky Factorization.** Slower than the multifrontal solver but uses less memory.

**Out-of-core Sparse Choleksy Factorization.** This is a supernodal left-looking factorization code with an associated solve routine that can solve very large problems by storing the Cholesky factor on disk. See [13] for further details.

**Out-of-core Sparse Pivoting LU Factorization.** This is a supernodal left-looking factorization code with an associated solve routine that can solve very large problems by storing the LU factors on disk. The algorithm is a supernodal version of the algorithm described in [8].

**Drop-Tolerance Incomplete-Cholesky Factorization.** Much slower than the supernodal solvers when it factors a matrix completely, but it can drop small elements from the factorization. It can also modify the diagonal elements to maintain row sums. The code uses a column-based left-looking approach with row lists.

**$LDL^T$ Factorization.** Column-based left-looking with row lists. Use the supernodal codes instead, since they are faster, unless you really need an $LDL^T$ factorization and not an $LL^T$ Cholesky factorization.

**Ordering Codes and Interfaces to Existing Ordering Codes.** The library includes a unified interface to several ordering codes, mostly existing ones. The ordering codes include Joseph Liu's `genmmd` (a minimum-degree code in Fortran), Tim Davis's `amd` codes (approximate minimum degree), METIS (a nested-dissection/minimum-degree code by George Karypis and Vipin Kumar), and a special-purpose minimum-degree code for no-fill ordering of tree-structured matrices. All of these are symmetric orderings. The library also includes an interface to Tim Davis's `colamd` column ordering code for LU factorization with partial pivoting.

**Matrix Operations.** Matrix-vector multiplication, triangular solvers, matrix reordering.

**Matrix Input/Output.** Routines to read and write sparse matrices using a simple file format with one line per nonzero, specifying the row, column, and value.

**Matrix Generators.** Routines that generate finite-differences discretizations of 2- and 3-dimensional partial differential equations. Useful for testing the solvers.

**Iterative Solvers.** Preconditioned conjugate-gradients and preconditioned MINRES (See [1], for example).

**Support-Graph Preconditioners.** These preconditioners construct a matrix larger than the coefficient matrix and use the Schur complement of the larger matrix as the preconditioner. The construction routine can construct Gremban-Miller preconditioners [9, 10] along with other (yet undocumented) variants.

**Vaidya's Preconditioners.** Augmented Maximum-weight-basis and Maximum-spanning-tree preconditioners [2, 4, 6, 16]. These preconditioners work by dropping nonzeros from the coefficient matrix and them factoring the preconditioner directly.

**Recursive Vaidya's Preconditioners.** These preconditioners [3, 12, 16] also drop nonzeros, but they don't factor the resulting matrix completely. Instead, they eliminate rows and columns which can be eliminated without producing much fill. They then form the Schur complement of the matrix with respect to these rows and columns and drop elements from the Schur complement, and so on. During the preconditioning operation, we solve for the Schur complement elements iteratively.

**Utility Routines.** Timers (wall-clock and CPU time), physical-memory estimator, and logging.

The routines that you are not likely to find in other libraries of sparse linear solvers are the direct supernodal solvers, the out-of-core solvers, and Vaidya's preconditioners. The supernodal solvers are fast and not many libraries include them; in particular, I don't think any freely-distributed library includes a sparse Cholesky factorization that is as fast as TAUCS's multifrontal code. I am not aware of any othe library at all that includes efficient out-of-core sparse factorizations.

As of version 2.0, the direct solvers work on real and complex matrices, single or double precision. The iterative solvers work on real matrices only.

To get a sense of the speed of the in-core multifrontal sparse Cholesky routine, let's compare it to MATLAB 6's sparse Cholesky solver. On a $600 \times 600$ mode problem (matrix order is 360000) TAUCS reorders the matrix using a minimum degree code that results in a Cholesky factor with approximately 12 million nonzeros. TAUCS factors the reordered matrix in 15.6 seconds, whereas MATLAB 6 takes 81.6 seconds to perform the same factorization, more than 5 times slower. The ratio is probably even higher on 3D meshes. (These experiments were performed with version 1.0 of the library on one processor of a 600MHz dual-Pentium III computer running Linux.)

TAUCS is easy to use and easy to cut up in pieces. It uses a nearly trivial design with only one externally-visible structure. If you need to use just a few routines from the library (say, the supernodal solvers), you should be able to compile and use almost only the files that include these routines; there are not many dependences among source files. The new configuration system, introduced in Version 2.1 makes it almost trivial to build a subset library that contains only the routines that you need (and the ones they depend on).

Two minor design goals that the library does attempt to achieve is avoidance of name-space pollution and clean failures. All the C routines in the library start with the prefix `taucs` and so do the name of structures and preprocessor macros. Therefore, you should not have any problems using the library together with other libraries. Also, the library attempts to free all the memory it allocates even if it fails, so you should not worry about memory leaks. This also allows you to try to call a solver in your program, and if it fails, simply call another. The failed call to the first solver should not have any side effects. In particular, starting in version 2.0 we use special infrastructure to find and eliminate memory leaks. This infrastructure allows us to ensure that no memory remains allocated after the user's program calls the appropriate `free` routines, and that no memory remains allocated in case of failures. This infrastructure also allows us to artificially induce failures; we use this feature to test the parts of the code that handle failures (e.g., failures of `malloc`), parts that are normally very rarely used.

The library is currently sequential. You can use parallelized BLAS, which may give some speedup on shared-memory multiprocessors. We have an experimental parallel version of the multifrontal Cholesky factorization, but it is not part of this release.

### A Preview of Things to Come

The next versions of the library should include

- In-core and out-of-core sparse symmetric indefinite factorizations.
- High-performance multithreaded LU factorization for unsymmetric matrices.
- A drop-tolerance incomplete LU factorization and nonsymmetric iterative solvers. The code is written but some of it needs to be converted from Fortran to C and it needs to be integrated into the library.

More distant versions may include

- A multithreaded version of the supernodal Cholesky factorizations.

Your input is welcome regarding which features you would like to see. We have implemented quite a few features as a direct response to users's requests (e.g., the complex routines and the out-of-core sparse LU), so don't be shy!

### 1.2 License

TAUCS comes with no warranty whatsoever and is distributed under the GNU LGPL (Library or Lesser GNU Public Library). The license is available in `www.gnu.org`. Alternatively, you can also elect to use TAUCS under the following UMFPACK-style license, which is simpler to understand than the LGPL:

> TAUCS Version 1.0, November 29, 2001. Copyright (c) 2001 by Sivan Toledo, Tel-Aviv Univesity, stoledo@tau.ac.il. All Rights Reserved.
>
> TAUCS License:
>
> Your use or distribution of TAUCS or any derivative code implies that you agree to this License OR to the GNU LGPL.
>
> THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED OR IMPLIED. ANY USE IS AT YOUR OWN RISK.
>
> Permission is hereby granted to use or copy this program, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any derivative code must cite the Copyright, this License, the Availability note, and "Used by permission." If this code or any derivative code is accessible from within MATLAB, then typing "help taucs" must cite the Copyright, and "type taucs" must also cite this License and the Availability note. Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. This software is provided to you free of charge.

The distribution also includes the AMD symmetric ordering routines, which come under a different, more restrictive license. Please consult this license in the source files (say `src/amdtru.f`). You can compile and use the library without these routines if you cannot accept their license.

## 2 Installation and Configuration

This section explains how to build TAUCS and how to configure it to suit different needs and different platforms. The configuration and build system described here was introduced in Version 2.1 of TAUCS. This system simplifies the installation process, allows the user to control the configuration of the library, and allows the user to maintain in a single diretory tree builds with different configurations and for different platforms.

### 2.1 Quick Start

In the top-level directory that you unpacked, type `configure` and then type `make` (or `nmake` on windows). This should build the library and a few test programs. If this fails, or if you need to tune the library to your needs, you'll have to read a bit more. If this succeeds, the build process will put the resulting library in the directory `lib/OSTYPE`, where `OSTYPE` stands for the operating system, such as `win32` (Windows), `darwin` (MacOS X), `linux`, `solaris`, `irix`, `aix`, and so on. Test programs will reside in `bin/OSTYPE`, and object files, which you probably do not need ,will be stored in `obj/OSTYPE`.

The command `make clean` removes the object files, binaries, and libraries.

To build the software on a new operating system (in the same directory tree), simply run `configure` and `make` again; the two builds will resided in completely different subdirectories. If you later need to build again the first distribution, use `make -f build/OSTYPE/makefile` (or `nmake /F` on Windows). The makefile in the top-level directory is essentially a link to the most-recently configured makefile, but older makefiles remain in the `build` directory.

## 2.2 An Overview of the Configuration and Build Process

Building the TAUCS involves two stages, a configuration stage and a build stage. The `configure` script performs the configuration stage. It does essentially three things:

1. determines the operating system you are running on,
2. builds a program, called `configurator`, that will build a makefile appropriate for that operating system, and
3. runs `configurator`.

This stage usually runs without a problem. The only likely problem is when the script cannot figure out which operating system you are running on. It was tested to work correctly on Windows, Linux, MacOS X, Solaris, Irix, and AIX, and it will probably work on most other Unix systems. If it fails and asks you to set the environment variable `OSTYPE`, set it and try again (and please let me know about this problem).

Typing `make` (on Windows, both GNU `make` and `nmake` work) performs the second stage, which builds the library and test programs. To build the library, `make` needs to know how to run the compiler and library manger, and to build the test programs, `make` also needs to know how to run the linker and where to find libraries that TAUCS calls.

The `make` process reads the platform- and site-dependent information that it needs from a file called `config/OSTYPE.mk`, where `OSTYPE` stands for the operating system. The distribution comes with `OSTYPE.mk` files containing reasonable defaults for several operating systems, but you may need to edit these files (or override them as explained below in the secion on variant builds). **In particular, supplying references to high-performance of LAPACK the Basic Linear-Algebra Subroutines (the BLAS) is crucial**; if the build process cannot find these libraries, the build will fail, and if you supply references to low-performance implementations, the libraries and test programs will build, but will be unnecessarily slow.

You can control both stages of the configuration and build process. You can control the first stage by instructing `configure` what parts of the package to build. For example, you can create a makefile that will build some or none of the test programs, you can create a makefile that will build your own program, you can create a makefile that will create a library with subset functionality (e.g., only out-of-core direct solvers for complex data types), and so on. You can create files that represent these configurations, so you can build the same configurations on multiple platforms.

You can control the build stage in two ways. The first is to change the definition of macros in the `OSTYPE.mk` files. The second is to define *build variants*. A build variant is a set of macro definitions that override those in the `OSTYPE.mk` files. By creating several variants, you can maintain versions of the library (and of client programs) that use different compilers, different compiler options, or different libraries, all on the same platform. For example, you can use variants to maintain libraries that link with several different BLAS implementations, debug and release libraries, and so on.

## 2.3 Configuration

The `configure` script is responsible for determing `OSTYPE`, for building the configuration program `configurator/configurator`, and for running it. It passes any command-line argument it

received to it. The `configurator` program is the one that actually builds the makefile according to a given configuration.

The package (the library and test programs) is broken into *modules*, and a *configuration* specifies which modules are included in a build and which are not. There are several kinds of modules:

- *Number-type modules* that control whether or not the library supports certain number types. The four number types TAUCS supports are double- and single-precision floating-point real numbers, and double- and single-precision complex numbers (modules names `DREAL`, `SREAL`, `DCOMPLEX`, `SCOMPLEX`).

- *Library-functionalify modules*. These are the most common modules. They control which subroutines are compiled and packaged in the final library. These modules include the in-core sparse Cholesky factorizations (`LLT`), out-of-core factorizations (`OOC_LLT` and `OOC_LU`), and so on. Several modules control access to external ordering codes (`AMD`, `GENMMD`, `COLAMD`, `METIS`). If you include the first three, the actual ordering code is included in the library. If you include the `METIS` module, TAUCS will be able to call METIS, but you will need to provide METIS in a separate library.

- *Test-program modules*. This modules control which one of the test programs are built. The test programs need quite a lot of functionality, so a configuration that builds them may be too large for some applications. For example, the test programs need the `MATRIX_IO` module, which contains routines for reading and writing matrices to various file formats. When you call TAUCS from an application code, the application code provides the matrices and uses the solvers, but it may not need to read or write matrices to files. There is one special test-program module, called `AD_HOC_TEST`, which we use for building small test programs. Do you include it in standard configurations.

The specification of a module includes a list of modules it depends on. They are automatically included in the configuration if the dependent module is included.

There are several ways to specify a configuration. When you run `configure` without any arguments, it will use a built-in configuration that includes essentially all the modules. If you run `configure` with the option `interactive`, it will ask you interactively whether or not to include each module. It will skip modules that must be included because of dependecies by other modules you already included. If you just hit ENTER as a reply, the program will use the built-in default as a reply.

You can also run `configurate` with an option `in=`*filename*, in which case it will read a configuration from *filename*. Configuration files have two useful features that are worth knowning about. First, they can specify a configuration by first including all modules and then excluding the ones that are not part of the configuration, or they can first exclude all modules and then include the ones that are needed. This feature helps create almost-full and almost-empty configurations. Second, The file can contain other text in addition to the configuration specification. This allow you to embed a configuration specification within comments in C files, and so on. The precise format of the configuration file is explained in the on-line documentation, which is printed out when you run `configure help`. You can also create a configuration file using the `out=`*filename* option, which saves the configuration (say one generated interactively) to a file.

The last way to specify a configuration, or more precisely to tune one, is with `module=`*modulename* or `module=!`*modulename*. These options include or exclude a specific module. Because options are processed in a sequence, this allows you to override the built-in configuration or a configuration read from a file. For example, the command `configure in=myconfig module=!METIS` excludes one module from a stored configuration. You cannot violate dependencies this way, since the set of required module is recomputed after every configuration change. You will need to escape the exclamation mark (which signals module exclusion) in most Unix shells. That

is, in Linux and Unix you probably need to run `configure in=myconfig module=\!METIS`. You can mix several `in=` and `module=` options in one invocation.

## 2.4 Controlling Build Parameters

The `configure` script generates a makefile and an include file. The include file defines a preprocessor variable for each included module, which allows the sources to know what parts of the library are available and what parts are missing from the current configuration. The makefile is almost completely generic: only the first few lines differ from one `OSTYPE` to another (the only differences are the definition of the `OSTYPE` macro itself, which is needed in case it is not set in the environment, and the path separator character in the names of included makefiles). To build TAUCS on different platforms, the makefile utilizes a number of macros that define the names and command-line options of the compilers and other build-related tools, and the names of libraries needed to build executables.

The external libraries that you need to point to are BLAS and LAPACK, the C libraries (some linkers include them automatically, but not always; on Unix and Linux systems you need to specify `-lm` to link with the library of mathematical routines), the Fortran 77 libraries, and METIS. The first three are always required, but the last two are not strictly required. You only need to point to METIS if the `METIS` module is included. You only need the Fortran libraries if you use a Fortran compiler or if you use the `MATRIX_IO` module.

TAUCS comes with a few Fortran sources. Most are ordering codes, and one contains a set of routines that read matrices in Harwell-Boeing format. The distribution comes with automatically-produced C translations for all of them. The translations were generated by the `f2c` translator[1]. To use the translations, simply define the macros that define the Fortran compiler to be the same to those defining the C compiler, and define the macro `F2CEXT` to be `.c`. The ordering codes perform no I/O, so the C versions do not need any external libraries. The Harwell-Boeing interface routines, however, need the Fortran standard library, so if you include the `MATRIX_IO` module, you will need to point to the Fortran libraries if you actually use a Fortran compiler, or to the `f2c` library if you only use a C compiler.

The "full" distribution of TAUCS contains a complete set of freely-available binary libraries for some platforms in the `external/lib/OSTYPE` directory. These libraries should allow you to build the default configuration of TAUCS without any external libraries. However, the performance of these bundled libraries may be suboptimal. In particular, try to find the best implementation of the BLAS and LAPACK, because their performance strongly influences TAUCS's. If you use the ATLAS implementation[2], try to build it on your computer, or to download a version suited for your processor.

## 2.5 Variant Builds

*Variants* allow you to maintain several builds with different configurations and build parameters simulteneously. You create a variant by running `configure` with the option `variant=`*variantname*. This option does two things: it changes the names of the subdirectories containing object files, binary executables, and binary libraries to include the variant's name, and it instructs the makefile to load a second macro-definition file after `config/OSTYPE.mk`. The name of the second file is `config/OSTYPE`*variantname*`.mk`.

For example, suppose we want to create a Linux build that uses Intel's MKL (an implementation of LAPACK and the BLAS) instead of ATLAS, which we normally use. We create a variant called `_mkl` by running `configure variant=_mkl`. Then we create a file `config/linux_mkl.mk` that redefines the macros that point to LAPACK and the BLAS. When we run `make`, it will use the redefined

[1] `http://netlib.bell-labs.com/netlib/f2c/`
[2] `http://math-atlas.sourceforge.net/`

macros, and will place the objects, binaries, and libraries in `obj/linux_mkl`, `bin/linux_mkl`, and `lib/linux_mkl`, instead of in `*/linux`.

You can also use variants to maintain different configurations. For example, suppose that you want to maintain both a default configuration and a configuration which doesn't use METIS on Windows. You first run `configure` and then `make` to build the default configuration. Then, you create an empty file `config/win32_nometis.mk` and run `configure variant=_nometis module=!METIS`. When you now run `make`, it will build the METIS-less configuration and put the results in `*/win32_nometis`. You can also, of course, create variants that represent both special configurations and special build parameters.

Note that if you use variants to represent your local build parameters, instead of changing the provided `config/OSTYPE.mk` files, unpacking new versions of TAUCS will not overwrite your changes.

## 2.6 Building a Multithreaded Library

Starting in version 2.2, some of the routines in TAUCS are multithreaded (in 2.2, only the multifrontal Cholesky factorization is multithreaded). TAUCS uses Cilk[3] [7, 14], a parallel programming language, to implement the multithreaded algorithms. Cilk is a programming environment that supports a fairly minimal parallel extension of the C programming language using a Cilk-to-C translator and a specialized run-time system. It is specifically designed to parallelize recursive codes. One of the most important aspects of using Cilk is the fact that it performs automatic dynamic scheduling that leads to both load balancing and locality of reference.

To enable multithreading in TAUCS, simply define the macros `CILKC`, `CILKFLAGS`, and `CILKOUTFLG` in `config/OSTYPE.mk`, or better yet, in the variant `.mk` file. The TAUCS distribution comes with `_cilk` variant files for Linux, Solaris (Sun), and Irix (SGI), so you can use these as templates. You should probably define the linker (the `LD` macro) to be the Cilk compiler, to allow the compiler to link with the Cilk run-time libraries. If these macros indeed point to a Cilk compiler and appropriate flags, some routines in TAUCS will be multithreaded. TAUCS parallelize both sparse operations and dense operations, so you do not need to link Cilk builds of TAUCS with a multithreaded BLAS library. (On the other hand, if you do not want to bother with a Cilk build, a multithreaded BLAS will provided some speedup, but not as much as a Cilk build.)

To control the level of multithreading, pass the option `taucs.cilk.nproc=`*number* to `taucs_linsolve`.

For more information on our implementation of the multifrontal sparse Cholesky algorithm in Cilk, including performance results, see [11] (a preprint should be in the `doc` directory). The implementation that we distribute is simpler than the implementation described in [11] (but easier to maintain, which is the reason we distribute it), but follows same principles except for the blocked and recursive dense data layouts.

The Cilk support is currently experimental.

TAUCS does not support MPI or other distributed-memory programming models.

## 2.7 Testing

Typing `testscript` runs a script that builds a set of test programs, runs them, and records the results in a file called `testscript.log` in the top-level directory. Each program configures TAUCS differently, so for each test program the library is compiled from scratch. Therefore, the script takes a while to run.

You can control which modules are included in these test builds and what compiler options and libraries are used. The `testscript` script passes its command-line argument to the `configure` script, after the `in=`*filename* argument that specifies the build. This allows you to add or remove

[3] `http://supertech.lcs.mit.edu/cilk/`

modules and to use variants. For example, suppose that you have a variant called `_intel` that uses the Intel compilers and the Intel BLAS, and that you do not have METIS on your system. By running `testscript module=!METIS variant=_intel` you will test the `_intel` variant without the METIS module.

The testing programs are an on-going effort and they currently do not exhaustively test the library.

### 2.8 Obscure Details

As the title of this section indicates, you should probably skip it, unless you are curious about details of the configuration and build process, or you run into trouble.

**Testing the Capabilities of the Compilers and External Libraries** TAUCS uses C preprocessor macros defined in two automatically-produced header files to control compile-time options. One file, `build/OSTYPE/taucs_config_build.h`, defines a macro for each module that is included in the configuration. This allows TAUCS's sources to know whether a given module is included or excluded. The other file, `build/OSTYPE/taucs_config_tests.h`, defines macros that indicate the presense of various capabilities of the compiler and of external libraries.

The header file `build/OSTYPE/taucs_config_tests.h` is created by TAUCS's makefile. To create it, the makefile tries to compile, link, and run several programs. Each such program tests for one capability. If the program compiles, links, and runs correctly, it write a macro definition to the header file. If it fails to compile, link, or run, the macro definition is simply not written to the header file.

As of version 2.2, there are four such programs. One tests whether the Fortran BLAS subroutines can be called from C as if they were compiled by the C compiler; a sister program tests whether the Fortran BLAS can be called from C by adding an underscore to the subroutine names (this is a convension of many Fortran compilers). Typically, one of these programs succeeds and the other fails to link with the BLAS. If both fail, the entire build of TAUCS will fail. This failure typically indicates that the linker cannot find the BLAS at all. A third program tests whether the C compiler supports C99 complex numbers, and a fourth tests whether the nominal Cilk compiler is indeed a Cilk compiler, or simply a C compiler. The repertoire of these test programs is likely to change between versions of TAUCS.

If ones of these programs fail even though you think they should have succeeded, check your `.mk` file. The problem can usually be traced to inappropriate compiler, compiler flags, or library specification. For example, if you specify the C compiler flags for the GCC compiler as `-std=c89`, the C99-complex-numbers test will fail, but if you specify `-std=c99` it will succeed.

## 3 Using TAUCS without Programming

There are three ways to use TAUCS without much programming: to use a read-made executable, to use a simple interface that allows MATLAB to call TAUCS, or to use the TAUCS routines that are built into products like MATHEMATICA and MATLAB.

### 3.1 Ready-to-use Executables

The `progs` directory contains example programs that you can use to test TAUCS without writing any code, and to guide you in calling the library from your own programs. These programs can generate matrices or read them from files, and they can employ several solvers. The programs print out detailed usage instructions when invoked with no arguments. The main programs are

**taucs_run** The main sample executable, which can solve linear systems iteratively or directly. It solves the system by calling `taucs_linsolve` and passing to it its command-line arguments, so you can control the solution method using the `taucs_linsolve` arguments. It can generate test matrices using the command-line arguments `taucs_run.mesh2d=`*size* and `taucs_run.mesh3d=`*size*, or it can read matrices from files using the arguments `taucs_run.ijv=`*filename* or `taucs_run.hb=`*filename*.

**direct** An old test routine for direct solvers. It is useful as an example if you need to call solvers directly rather than through `taucs_linsolve`.

**iter** An old test routine for iterative solvers.

## 3.2 Calling TAUCS from MATLAB

The TAUCS distribution contains code that allows MATLAB to call TAUCS routines. The code consists of several MATLAB functions (`.m` files), a set of special input-output routines within TAUCS, and the above-mentioned executables. Using the code is simple: using the provided MATLAB functions, you write the coefficient matrix and right-hand side of a linear system of equations to files. You then invoke, from within MATLAB, an executable that uses the special input-output routines to read these files, solves the linear system using a TAUCS routine, and writes the solution to a third file. Another MATLAB function reads the solution vector from the file. There is also a sample MATLAB function that automates the entire procedure, so to solve $Ax = b$ you simply invoke `x=taucs_ooc_solve(A,b)`. You can easily modify that function to invoke other linear solvers in TAUCS.

The files that are used to pass matrices and vectors between MATLAB and TAUCS are binary files. Data is essentially just dumped from memory into these files, so writing and reading them is typically fast and does not represent a significant performance overhead. Obviously, the more expensive the linear solver, the less important this overhead becomes.

To build the executable that these MATLAB scripts call, run `configure in=progs/ooc_factor_solve.c` and then `make -f build/OSTYPE/makefile` (on Windows, replace `/` by `\` and replace `make -f` by `nmake /F`).

We currently do not maintain interface codes in C that would allow MATLAB to call TAUCS routines packaged into dynamically-linked library without storing matrices and vectors into files. The diversity of routines in TAUCS, at least in versions 2.0 and earlier, makes it quite hard to maintain such codes. Version 2.1 introduces a new unified routine, so perhaps in the future we will provide such an interface code.

## 3.3 TAUCS Routines in other Software

TAUCS routines are built into some other software packages. They are usually invoked automatically when appropriate.

MATHEMATICA 5 uses TAUCS's in-core sparse Cholesky factorization to factor sparse matrices within MATHEMATICA's linear solver (`LinearSolve[A,Method->Cholesky]`), and within the interior-point linear-programming solver.

MATLAB 7 will use TAUCS's in-core sparse Cholesky factorization within the backslash linear solver.

# 4 TAUCS Fundamentals

## 4.1 Sparse Matrix Representation and Interface Conventions

TAUCS uses the following compressed-column-storage (CCS) structure to represent sparse matrices. Like other TAUCS data structures and data types, it is defined in `src/taucs.h`, which must be included in source files that call TAUCS routines.

```
typedef struct {
  int     n;       number of columns
  int     m;       number of rows
  int     flags;   see below
  int*    colptr;  pointers to where columns begin in rowind and values
                   0-based, length is (n+1)
  int*    rowind;  row indices, 0-based
  union {
    void*           v;
    taucs_double*   d;
    taucs_single*   s;
    taucs_dcomplex* z;
    taucs_scomplex* c; }
  values;          numerical values
} taucs_ccs_matrix;
```

(Comments are set in italics). Before version 2.0, the type of `values` was `double*`; since version 2.0, `values` is a union, to support multiple data types. The data types `taucs_double`, `taucs_single`, `taucs_scomplex`, and `taucs_dcomplex` correspond to C's native `float` and `double` and to arrays of two such numbers to represent the real and imaginary parts of complex numbers. In C compilers that support complex arithmetic, the build process uses native complex representations for `taucs_scomplex`, and `taucs_dcomplex` (`gcc` support complex arithmetic; in the future, we expect most C compilers to support complex arithmetic since this is part of the new C99 standard for the C language). Otherwise, we use arrays of two `float`s or `double`s.

The flags member contains the bitwise or of several symbolic constants that describe the matrix:

```
TAUCS_INT          matrix contains integer data
TAUCS_SINGLE       matrix contains single-precision real data
TAUCS_DOUBLE       matrix contains double-precision real data
TAUCS_SCOMPLEX     matrix contains single-precision complex data
TAUCS_DCOMPLEX     matrix contains double-precision complex data
TAUCS_PATTERN      matrix contains no numric values, only a nonzero pattern

TAUCS_TRIANGULAR   matrix is triangular
TAUCS_SYMMETRIC    matrix is symmetric
TAUCS_HERMITIAN    matrix is hermitian

TAUCS_LOWER        matrix is lower triangular (if TAUCS_TRIANGULAR is set)
                   or the lower part of a triangular/hermitian matrix
TAUCS_UPPER        upper triangular or upper part of symmetric/hermitian
```

In symmetric and hermitian matrices we store only one triangle, normally the lower one. Most of the routines fail if their argument contain the upper triangle of a symmetric/hermitian matrix.

### Generic and Type-Specific Routines

Most of the computational and data-structure-related rourines in TAUCS have five entry points, one for each data type (real/complex, single/double), and one generic. The generic routine calls one of the four specific routines based on the data type of the actual arguments. For example, the following five routines compute the Cholesky factorization of a matrix $A$.

```
void* taucs_sccs_factor_llt_mf (taucs_ccs_matrix* A);
void* taucs_dccs_factor_llt_mf (taucs_ccs_matrix* A);
void* taucs_cccs_factor_llt_mf (taucs_ccs_matrix* A);
void* taucs_zccs_factor_llt_mf (taucs_ccs_matrix* A);
void* taucs_ccs_factor_llt_mf  (taucs_ccs_matrix* A);
```

Each of the first four routines operate on a single data type. Each one of them expects $A$'s elements to be of a specific data type. For example, `taucs_zccs_factor_llt_mf` expects $A$'s elements to be of type `taucs_dcomplex`. Names of type-specific routines always start with `taucs_s`, `taucs_d`, `taucs_c`, or `taucs_z`. The fifth declaration is for the generic routine, which determines the data type using `A->flags` and calls the appropriate type-specific routine. Calling the generic routine incurs a small overhead compared to calling the appropriate type-specific routine, but this overhead is negligible in most cases. User codes that call TAUCS should call the generic routines.

The rest of the documentation only documents generic routines.

### Creating and Deleting Sparse Matrices

The following routines create and delete sparse matrices.

```
taucs_ccs_matrix* taucs_ccs_create(int m, int n, int nnz, flags);
void              taucs_ccs_free  (taucs_ccs_matrix* A);
```

The first routine, `taucs_ccs_create`, allocates memory for an $m$-by-$n$ matrix with space for `nnz` nonzeros. Its last argument specifies the data type for the matrix, and can also specify other properties, such as symmetry. The interface to `taucs_ccs_create` changed in version 2.0! The matrix is not initialized in any way apart from setting the flags. The second routine frees a matrix and all the memory associated with it.

### Reading and Writing Sparse Matrices

TAUCS includes a number of routines to read and write sparse matrices from and to files in various formats. The first pair of routines hanle `ijv` files, which have a simple textual format: each line contains the row index, column index, and numerical value of one matrix entry. Indices are 1-based. The file does not contain any flags regarding symmetry and so on, so you have to pass both data type and structural flags to `taucs_ccs_read_ijv`., which reads a matrix from a file.

```
taucs_ccs_matrix* taucs_ccs_read_ijv (char* filename,int flags);
int               taucs_ccs_write_ijv(taucs_ccs_matrix* A, char* filename);
```

The `ijv`-reading routine assumes that the lower part of symmetric and hermitian matrices is stored in the file; if the upper part is also stored, the routine simply ignores it. The `ijv`-writing routine always writes all the matrix's entries into the file. You can read `ijv` files into MATLAB using the command

```
read 'Afile.ijv' -ascii; A=spconvert(Afile);
```

The next format, the `mtx` format, is almost identical to the `ijv` format, but the first line in the file contains the number of rows and columns, and nonzeros in the matrix.

```
taucs_ccs_matrix* taucs_ccs_read_mtx (char* filename,int flags);
int               taucs_ccs_write_mtx(taucs_ccs_matrix* A, char* filename);
```

The `ccs` format is also a textual format. The first integer in the file store the matrix's dimension `n`. It is followed the integers in the arrays `colptr` and `rowind` in the CCS data structure, and then the array of real or complex values. This is essentially a textual representation for square CCS matrices, but excluding the `flags`.

```
taucs_ccs_matrix* taucs_ccs_read_ccs (char* filename,int flags);
int               taucs_ccs_write_ccs(taucs_ccs_matrix* A, char* filename);
```

The `binary` format simply dumps a `taucs_ccs_matrix` into (or from) a binary file. This format is not archival (it may change in future versions of TAUCS), but it can be used to transfer matrices quickly between TAUCS clients and MATLAB or other programs (we have MATLAB routines to read and write such matrices). The current version of TAUCS includes only a `binary`-reading routine. Since the `flags` are stored in the file, there is no `flags` argument to the routine.

```
taucs_ccs_matrix* taucs_ccs_read_binary(char* filename);
```

Finally, the following routine reads a matrix stored in Harwell-Boeing format, which is used in matrix collections such as MatrixMarket and Tim Davis's. Harwell-Boeing files contain structural infomation (e.g., symmetry) and distinguish between real and complex matrices, so the `flags` argument to this routine only specifies whether the resulting matrix will be single or double precision. If the Harwell-Boeing matrix contains only a nonzero pattern, the routine creates a matrix with random elements in the specified positions (if the Harwell-Boeing matrix is symmetric the diagonal elements are not set to random values but to values that ensure that the matrix is diagonally dominant).

```
taucs_ccs_matrix* taucs_ccs_read_hb(char* filename, int flags);
```

## 4.2 Vectors

TAUCS represents vectors as simple arrays of numbers, with no type or length information. If one of the arguments to a generic routine is a matrix and the other is a vector, the routine determines the length and type of the vector from the information associated with the matrix. The following routine, for example, multiplies a sparse matrix $A$ by a vector `x` and stores the result in another vector, `b`.

```
void taucs_ccs_times_vec  (taucs_ccs_matrix* A,
                           void*             x,
                           void*             b);
```

The pointers `x` and `b` must point to arrays of numbers with the same type as $A$'s elements. That is, if `TAUCS_DCOMPLEX` is set in `A->flags`, then `x` and `b` must point to arrays of `taucs_dcomplex` elements. The size of `x` and `b` must match the number of columns in $A$.

Vector handing routines that have no matrix argument have explicit arguments that specify the data type and length of input and output vectors. For example, the next two routines read and write vectors from and to binary (non archival) files.

```
void* taucs_vec_read_binary (int n, int flags, char* filename);
int   taucs_vec_write_binary(int n, int flags, void* v, char* filename);
```

## 4.3 Utility Routines

TAUCS routines print information to a log file using a special routine,

```
int    taucs_printf(char *fmt, ...);
```

Another routine,

```
void   taucs_logfile(char* file_prefix);
```

sets the name of the log file. The names `stdout`, `stderr` and `none` are acceptable, as are actual file names. If you do not call this routine or if you set log file to `none`, the library produces no printed output at all.

```
int    taucs_printf(char *fmt, ...);
```

TAUCS can usually determine the amount of memory available. This can be useful when calling an out-of-core solver, which needs this information in order to plan its schedule. This information can also be useful for determining whether an in-core direct solver is likely to run out of memory or not before calling it.

```
double taucs_system_memory_size();
double taucs_available_memory_size();
```

The first routine attempts to determine how much physical memory the computer has, in bytes. The second reports the amount of memory in bytes that you can actually allocate and use. It returns the minimum of 0.75 of the physical memory if it can determine the amount of physical memory, and the amount that it actually managed to allocate and use. You should use the second routine, since the first may fail or may report more memory than your program can actually allocate.

The next routines measure time.

```
double taucs_wtime();
double taucs_ctime();
```

The first routine returns the time in seconds from some fixed time in the past (so-called wall-clock time). The second returns the CPU time in seconds that the process used since it started. The CPU time is mostly useful for determining that the wall-clock measurements are not reliable due to other processes, paging, I/O, etc.

## 4.4 Error Codes

Some TAUCS routines return an integer error code. Before version 2.1, routines that return an integer error code returned 0 to signal success and -1 to signal failure. Routines that return a pointer used the value `NULL` to signal failure. Version 2.1 introduces symbolic integer error codes, so that routines can report several kinds of errors.

| | |
|---|---|
| `TAUCS_SUCCESS` | the routine completed its task successfully; guaranteed to have value zero |
| `TAUCS_ERROR` | an error occured, but either none of the specific error codes is appropriate, or the routine could not determine which error code to return (this might happen if the error occured deep in the code) |
| `TAUCS_ERROR_NOMEM` | failure to dynamically allocate memory |
| `TAUCS_ERROR_BADARGS` | the arguments to the routine are invalid |
| `TAUCS_ERROR_MAXDEPTH` | recursion is too deep and might cause a stack overflow |
| `TAUCS_ERROR_INDEFINITE` | input matrix was supposed to be positive definite but appears to be indefinite |

# 5 The Unified Linear Solver

Version 2.1 introduces a unified linear-solver interface. The interface consists of a single C routine that allows the user to invoke almost all the linear solvers that TAUCS provides.

The new interface is designed to achieve two main objectives. First, it allows us to add new functionality without adding new routines and without modifying client codes. Suppose that we want to add a new way to handle zero pivots in a Cholesky solver. In the pre-2.1 interface style, we had two choices: (1) to add a new routine that would treat zero pivots in a new way, or (2) add an argument on an existing routine, an argument that would tell it how to treat zero pivots. Both options have flaws. The first option causes the number of externally-visible routines to explode, and the second breaks old client codes. The new interface style allows us to add features in a cleaner way.

The second objective of the new interface style is to allow multiple ways for client codes to specify how a linear system should be solved. The solver specification can be embeded in the client's source code, in configuration files, or even on the client's command line.

The main drawback of the new interface is some additional overhead. The convenience of the new interface is achieved mostly by parsing character strings that represent solver options, and parsing these arguments takes time. For most linear systems, this overhead is negligeable.

We plan to keep supporting the old interfaces, but we recommend that new codes use the new interface unless the new interface results in actual and significant performance penalty.

There are a few things that the old-style interfaces can do but the new ones cannot. The most important one is allow direct access to the factors of a matrix. We are still unsure as to how to do this with the new interface. We would welcome any suggestions from users.

## 5.1 Usage by Example

The new interface consists of a general-purpose linear-solver routine,

```
int taucs_linsolve(taucs_ccs_matrix* A,   input matrix
                   void** factorization,  an approximate inverse
                   int    nrhs,           number of right-hand sides
                   void*  X,              unknowns
                   void*  B,              right-hand sides
                   char*  options[]       options (what to do and how)
                   void*  arguments[]);   option arguments
```

The first argument is a coefficient matrix $A$. The second is the address of a pointer to a representation of an approximate inverse. As we shall see below, this representation can include a Cholesky factorization of $A$, an incomplete factorization of $A$, a factorization of an approximation to $A$, and so on. The next three arguments specify the number of linear system of equations with the same coefficient matrix that we wish to solve, an array of unknowns (`nrhs` columns of `A->n` elements, column oriented), and an array of right-hand sides. The next two arguments are an array of options in character-string format, and an optional array of option arguments; both should be `NULL` terminated. The routine returns `TAUCS_SUCCESS` or an error code.

Let us see how this interface works using a few examples. The first example simply solves a single linear system $Ax = b$ where $A$ is symmetric positive definite,

```
char* options = { "taucs.factor.LLT=true", NULL };
...
if (taucs_linsolve(A,NULL,1,x,b,options,NULL) != TAUCS_SUCCESS)
  handle_the_error_somehow();
```

The options array tells the routine to use a Cholesky factorization. The routine will find a fill reducing row and column permutation, permute the matrix, factor the reordered matrix, and solve the linear system using this factorization. Since we did not pass an address of a pointer to a factorization, the factorization will be destroyed before the routine returns. If we want to reuse the same factorization later, we can separate the factor and solve phases (from here on we omit the error checking from the examples),

```
char* options_factor = { "taucs.factor.LLT=true", NULL };
void* F = NULL;
...
taucs_linsolve(A,&F,0,NULL,NULL,options,NULL); factor
...
taucs_linsolve(A,&F,1,x,b,NULL,NULL);          solve
...
taucs_linsolve(NULL,&F,0,NULL,NULL,NULL,NULL); free the factorization
```

The first call creates the factor, but does not solve any linear system, and the second uses that factor to solve a linear system. The third call, which passes a factorization object to the routine but no linear system (no matrix and no right-hand sides), is by convension a request to free the factorization. The same routine can also solve linear systems iteratively,

```
char* opt_no_precond = { "taucs.solve.cg=true", NULL };
char* opt_incomplete = { "taucs.factor.LLT=true",
                         "taucs.factor.droptol=1e-2",
                         "taucs.solve.cg=true",
                         NULL };
char* opt_amwb       = { "taucs.approximate.amwb=true",
                         "taucs.approximate.amwb.subgraphs=300",
                         "taucs.factor.LLT=true",
                         "taucs.solve.cg=true",
                         NULL }; sivan toledo how are you sivan test test test test
```

The first options vector indicates an iterative solution with no preconditioner, the second using a drop tolerance incomplete Cholesky preconditioner, and the third using a complete Cholesky factorization of an approximation to the coefficient matrix.

## 5.2 More on Options and their Arguments

There are four types of kinds of options:boolean, numeric, strings, and pointers. There are two ways to specify the value of an option: witnin the option string itself, following an = sign, or in the arguments array.

For example, you can specify a boolean option with the string literals `true` and `false` within the option string, as the previous examples have demostrated. You can also specify a boolean option in the arguments array,

```
int   factor_llt = 1;
char* options_factor = { "taucs.factor.LLT=#0", NULL };
void* arguments[]    = { &factor_llt, NULL };
...
taucs_linsolve(A,&F,0,NULL,NULL,options,arguments);
```

The # sign implies that the option is specified in a given location of the arguments array, rather than in the character string itself. The nonnegative integer that follows the # sign indicates the

location within the arguments array, the first location (index 0) in this example. This style of passing arguments allows you to control the option-value that your program passes to TAUCS without converting it to a string.

When you use the # notataion, the element in the arguments array should be an `int*` when the option is boolean, `double*` when the option is numeric (even if only integer values make sense), `char*` when the option is a string, and `void*` when the option is a pointer.

You can define arrays of options within your source code, as these examples show. You can also pass them to your program on the command line, and have your program pass them to `taucs_linsolve` without any processing,

```
int main(int argc, char* argv[]) {
  ...
  taucs_linsolve(A,NULL,1,x,b,argv,NULL);
```

In ANSI C, the `argv` array must be `NULL` terminated, so you can pass it safely to `taucs_linsolve`. The routine will use the options it understand; it will complain about the ones it does not understand, but it will otherwise ignore them. If you want to suppress these warnings, remove from the options array strings that do not start with `taucs`. This mechanism allows you to take advantage of new options in future versions of TAUCS simply by relinking your code with the new version of TAUCS; you don't even have to recompile your program.

## 5.3 A Catalog of Options

The following table documents the options that `taucs_linsolve` understands. In most cases, values have reasonable defaults. For example, if you do not specify an ordering, the routine will select an appropriate ordering based on the kind of factorization and the configuration of the library.

| | | |
|---|---|---|
| `taucs.approximate.amwb` | *boolean* | maximum-weight-basis and Vaidya preconditioners |
| `taucs.approximate.amwb.subgraphs` | *double* | desired number of subgraphs in the tree partitioning |
| `taucs.approximate.amwb.randomseed` | *double* | random seed, specify an integer in $[0, 2^{31} - 1]$ |
| `taucs.factor.LLT` | *boolean* | Chokesky factorization |
| `taucs.factor.LU` | *boolean* | LU factorization with partial pivoting |
| `taucs.factor.ldlt` | *boolean* | $LDL^T$ factorization without pivoting |
| `taucs.factor.mf` | *boolean* | multifrontal factorization |
| `taucs.factor.ll` | *boolean* | left-looking factorization |
| `taucs.factor.symbolic` | *boolean* | if false, the code will use the symbolic factorization from an earlier code |
| `taucs.factor.numeric` | *boolean* | if false, the code will only perform a symbolic factorization |
| `taucs.factor.ordering` | *string* | what preordering to use |
| `taucs.ooc` | *boolean* | requests out-of-core processing |
| `taucs.ooc.basename` | *string* | path name and base file name for data files |
| `taucs.ooc.iohandle` | *pointer* | pointer to an open `taucs_io_handle`. Specify either this option or the `basename` option in out-of-core requests |
| `taucs.ooc.memory` | *double* | the amount of in-core memory to use |
| `taucs.solve` | *boolean* | specify false if you do not want to solve linear systems; same as requesting to solve zero systems |
| `taucs.solve.cg` | *boolean* | requests an iterative solve with the Conjugate Gradients algorithm; if you do not specify any iterative solver, the routine will use a direct solve (single application of the preconditioner or of the factorization) |

| | | |
|---|---|---|
| `taucs.solve.minres` | *boolean* | requests an iterative solve with the MINRES algorithm |
| `taucs.solve.maxits` | *double* | maximum number of iterations |
| `taucs.solve.convergetol` | *double* | reduction in the norm of the residual that is considered convergence |
| `taucs.maxdepth` | *double* | maximum depth of recursion in recursive routines that traverse elimination trees; can be used to reduce chances of stack overflows |
| `taucs.cilk.nproc` | *double* | number of threads to be used in multithreaded routines |

# 6 Matrix Reordering

Reordering the rows and columns of a matrix prior to factoring it can have a dramatic effect on the time and storage required to factor it. Reordering a matrix prior to an iterative linear solver can have a significant effect on the convergence rate of the solver and on the time each iteration takes (since the reordering affects the time matrix-vector multiplication takes). The following routine computes various permutations that can be used effectively to permute a matrix.

```
void              taucs_ccs_order(taucs_ccs_matrix* matrix,
                                  int** perm, int** invperm,
                                  char* which);
```

The string argument `which` can take one of the following values, all of which are fill-reducing permutations. All except the last are for for symmetric matrices, and the last is only for unsymmetric matrices.

**identity** The identity permutation.

**genmmd** Multiple minimum degree. In my experience, this routine is often the fastest and it produces effective permutations on small- and medium-sized matrices.

**md, mmd, amd** True minimum degree, multiple minimum degree, and approximate minimum degree from the AMD package. In my experience they are slower then genmmd although they are supposed to be faster.

**metis** Hybrid nested-dissection minimum degree ordering from the METIS library. Quite fast and should be more effective than minimum degree codes alone on large problems.

**treeorder** No-fill ordering code for matrices whose graphs are trees. This is a special case of minimum degree but the code is faster than a general minimum degree code.

**colamd** Tim Davis's column approximate minimum-degree code. This ordering produces a column ordering that reduces fill in sparse LU factorizations with partial pivoting.

The next routine takes the permutation returned from `taucs_ccs_order` and permutes a matrix symmetrically. That is, the permutation is applied to both the rows and the columns.

```
taucs_ccs_matrix* taucs_ccs_permute_symmetrically(taucs_ccs_matrix* A,
                                                  int* perm, int* invperm);
```

The last two routines are auxiliary routines that permute a vector or inverse permute a vector. The interface to these routines changed in version 2.0!

```
void taucs_vec_permute (int n,
                        int flags,      data type
                        double v[],     input vector
                        double pv[],    permuted output vector
                        int    p[]);    permutation, 0-based

void taucs_vec_ipermute(int n,
                        int flags,      data type
                        double v[],     input vector
                        double pv[],    permuted output vector
                        int    invp[]); inverse permutation
```

# 7 Sparse Direct Linear Solvers

## 7.1 In-Core Sparse Symmetric Factorizations

The next routine factors a symmetric matrix $A$ completely or incompletely into a product of lower triangular matrix $L$ and its transpose $L^T$. If `droptol` is set to 0, the matrix is factored completely into $A = LL^T$. If `droptol` is positive, small elements are dropped from the factor $L$ after they are computed but before they update other coefficients. Elements are dropped if they are smaller than `droptol` times the norm of the column of $L$ and they are not on the diagonal and they are not in the nonzero pattern of $A$. If you set `modified` to true (nonzero value), the factorization is modified so that the row sums of $LL^T$ are equal to the row sums of $A$. A complete factorization should only break down numerically when $A$ is not positive definite. An incomplete factorization can break down even if $A$ is positive definite.

```
taucs_ccs_matrix* taucs_ccs_factor_llt(taucs_ccs_matrix* A,
                                       double droptol,
                                       int modified);
```

The factorization routine returns a lower triangular matrix which you can use to solve the linear system $LL^Tx = b$ (if the factorization is complete, that is, if $A = LL^T$, then this solves $Ax = b$). The formal type of the argument is `void*` but the routine really expects a `taucs_ccs_matrix*`, presumably one returned from `taucs_ccs_factor_llt`. The reason that we declare the argument to be `void*` is that all the solve routines that might be used as preconditioners must have the same type but each one accepts a different data type.

```
int               taucs_ccs_solve_llt  (void* L,
                                        double* x,
                                        double* b);
```

The routine `taucs_ccs_factor_llt` factors a matrix column by column. It is quite slow in terms of floating-point operations per second due to overhead associated with the sparse data structures and to cache misses. TAUCS also includes faster routines that can only factor matrices completely. These routines rely on an easy-to-compute decomposition of $L$ into so-called supernodes, or set of columns with similar structure. Exploiting supernodes allow these routines to reduce overhead and to utilize cache memories better.

```
void* taucs_ccs_factor_llt_mf(taucs_ccs_matrix* A);
void* taucs_ccs_factor_llt_ll(taucs_ccs_matrix* A);
```

The first routine (`_mf`) is a supernodal multifrontal routine and the second (`_ll`) is a supernodal left-looking routine. The multifrontal code is faster but uses more temporary storage. Both

routines return the factor in an opaque data structure that you can pass to the solve routine to solve $LL^T x = b$.

```
int taucs_supernodal_solve_llt(void* L,
                               double* x,
                               double* b);
```

The next routine deallocates the storage associated with such a factor.

```
void taucs_supernodal_factor_free(void* L);
```

You can also convert a supernodal factor structure to a compressed-column matrix using the following routine

```
taucs_ccs_matrix*
     taucs_supernodal_factor_to_ccs(void* L);
```

There may be two reason to perform this conversion. First, the compressed-column solve routine may be slightly faster than the supernodal solve routine due to cache effects and indexing overhead. Second, the only operations on supernodal factors are the solve and free routines, so if you want to perform another operation on the factor, such as writing it out to a file, you need to convert it to a compressed-column structure.

The following three routines are usefull when the application needs to factor several matrices with the same nonzero structure but different numerical values. These routines call the supernodal multifrontal factorization code. The first routine performs a symbolic elimination, which is a preprocessing steps that depends only on the nonzero structure of the input matrix. It returns a factor object, but with no numerical values (it cannot be yet used for solving linear systems).

```
void* taucs_ccs_factor_llt_symbolic(taucs_ccs_matrix* A);
```

The next routine takes a symbolic factor and a matrix and performs the numerical factorization. It returns 0 if the factorization succeeds, $-1$ otherwise. It appends the numeric values of the factors to the factor object, which can now be used to solve linear systems.

```
int   taucs_ccs_factor_llt_numeric(taucs_ccs_matrix* A,void* L);
```

If you want to reuse the symbolic factor, you can release the numeric information and call the previous routine with a different matrix, but with the same structure. The following routine releases the numeric information.

```
void taucs_supernodal_factor_free_numeric(void* L);
```

An auxiliary routine computes the elimination tree of a matrix (the graph of column dependences in the symmetric factorization) and the nonzero counts for rows of the complete factor L, columns of L, and all of L. This routine is used internally by the factorization routines, but it can be quite useful without them. In particular, computing the number of nonzeros can help a program determine whether there is enough memory for a complete factorization. Currently this routine is not as fast as it can be; it runs in time proportional to the number of nonzeros in L (which is still typically a lot less than the time to compute the factor). I hope to include a faster routine in future versions of TAUCS.

```
int taucs_ccs_etree(taucs_ccs_matrix* A, input matrix
                    int  parent[],       an n-vector to hold the etree
                    int  L_colcount[],   output; NULL is allowed
                    int  L_rowcount[],   output; NULL is allowed
                    int* L_nnz           output; NULL is allowed
                    );
```

You must pass the address of the output arguments if you want them or `NULL` if you do not need them.

The next routine factors a symmetric matrix A completely into a product $LDL^T$ where L is lower triangular and D is diagonal.

```
taucs_ccs_matrix* taucs_ccs_factor_ldlt(taucs_ccs_matrix* A);
```

The factorization routine returns a lower triangular matrix that packs both L and D into a single triangular, and which you can use to solve the linear system $LDL^Tx = b$. The formal type of the argument is `void*` but the routine really expects a `taucs_ccs_matrix*`, presumably one returned from `taucs_ccs_factor_llt`. The matrices L and D are packed into the matrix C that the routine returns in the following way: the diagonal of D is the diagonal of C, and the strictly lower triagular part of L is the strictly lower triangular part of C; the diagonal of L contains only 1, and is not represented explicitly. To solve linear systems you do not need to understand this packed format, only if you need to access elements of D or L.

```
int               taucs_ccs_solve_ldlt (void* L,
                                        double* x,
                                        double* b);
```

The routine `taucs_ccs_factor_ldlt` factors a matrix column by column. It is quite slow in terms of floating-point operations per second due to overhead associated with the sparse data structures and to cache misses.

## 7.2 Out-of-Core Sparse Symmetric Factorizations

TAUCS can factor a matrix whose factors are larger than main memory by storing the factor on disk files. The code works correctly even if the factor takes more than 4 GB of memory to store, even on a 32-bit computer (we have factored matrices whose factors took up to 46 GB of disk space on a Pentium-III computer running Linux). On matrices that can be factored by one of the supernodal in-core routines, the out-of-core code is usually faster if the in-core routines cause a significant amount of paging activity, but slower if there is little or no paging activity. As a rule of thumb, use the out-of-core routines if the in-core routines run out of memory or cause significant paging.

The basic sequence of operations to solve a linear system out-of-core is as follows:

1. Represent the coefficient matrix as a `taucs_ccs_matrix`.
2. Find a fill-reducing symmetric ordering and permute the matrix.
3. Create a file that will store the factor by calling `taucs_io_create_multifile`.
4. Factor the permuted coefficient matrix into the file by calling `taucs_ooc_factor_llt`. The Cholesky factor is now stored on disk files.
5. Solve one or more linear systems using the disk-resident factor by calling `taucs_ooc_solve_llt`.
6. Delete the factor from disk using `taucs_io_delete`, or just close the disk files by calling `taucs_io_close`. If you just close the file, you can keep it on disk and use it later to solve additional linear systems by opening it (`taucs_io_open_multifile`) and calling the solve routine.

TAUCS stores the sparse factor in multiple files, each at most than one gigabyte in size. The file-creation routine,

```
taucs_io_handle* taucs_io_create_multifile(char* basename);
```

receives a string argument that is used to generate file names. For example, if the argument is `"/tmp/bcsstk38.L"`, then the factor will be stored in the files `/tmp/bcsstk38.L.0`, `/tmp/bcsstk38.L.1`, `/tmp/bcsstk38.L.2`, and so on. To open an existing collection of files that represent a sparse matrix, call

```
taucs_io_handle* taucs_io_open_multifile(char* basename);
```

If you want to stop the program but retain the contents of such files, you must close them explicitly,

```
int taucs_io_close(taucs_io_handle* h);
```

The argument is the handle that the create or open routine returned. This routine returns `-1` in case of failure and `0` in case of success. To delete an existing an open collection of files, and to release the memory associated with a handle to the files, call

```
int taucs_io_delete(taucs_io_handle* h);
```

There is no way to delete files that are not open; if you want to delete an exising on-disk matrix, open it and then delete it.

Using the out-of-core factor and solve routines is easy:

```
int taucs_ooc_factor_llt(taucs_ccs_matrix* A,
                         taucs_io_handle* L,
                         double memory);
int taucs_ooc_solve_llt (void* L, double* x, double* b);
```

The first argument of the factor routine is the matrix to be factored (permute it first!), the second is a handle to a newly created TAUCS file that will contain the factor upon return, and the third is the amount of main memory that the factor routine should use. In general, the value of the third argument should be only slightly smaller than the amount of physical main memory the computer has. The larger the argument, the less explicit I/O the factorization performs. But a value larger than the physical memory will cause explicit I/O in the form of paging activity and this typically slows down the factorization. If you do not know how much memory to allow the routine to use, just pass the value returned by `taucs_available_memory_size()`; in most cases, this will deliver near-optimal performance. The return value of both the factor and solve routines is `0` in case of success and `-1` otherwise.

The first argument of the solve routine is the handle to the file containing the factor. The formal argument is declared as `void*` to ensure a consistent interface to all the solve routines, but the actual argument must be of type `taucs_io_handle*`. Do not pass a filename!

In this version of TAUCS the out-of-core routines are not completely reliable in case of failure. They will generally print a correct error message, but they may not return immediately and they may not release all the disk space and memory that they have allocated. In particular, this may happen if they run out of disk space. We will attempt to rectify this in future versions.

Finally, this version of the documentation does not document the interfaces to the matrix I/O routines that the out-of-core codes use. If you need such documentation to develop additional out-of-core matrix algorithms using TAUCS's I/O infrastructure, please let me know.

## 7.3 Out-of-Core Sparse Unsymmetric Factorizations

TAUCS can solve unsymmetric linear systems using an out-of-core sparse LU factorization with partial pivoting.

```
int taucs_ooc_factor_lu (taucs_ccs_matrix* A,
                         int*              colperm,
                         taucs_io_handle*  LU,
                         double            memory);
int taucs_ooc_solve_lu  (taucs_io_handle*  LU,
                         void*             x,
                         void*             b);
```

The interface to these routines is similar to the interface of the out-of-core symmetric routines, except that you do not need to prepermute A and you do not need to permute b and x before and after the solve. The argument `colperm` is a fill-reducing column permutation that you can obtain by calling `taucs_ccs_order` with a `colamd` ordering-specification. These routines perform all the necessary permutations internally, so you do not have to perform any.

### 7.4 Inverse Factorizations

TAUCS can directly compute the sparse Cholesky factor of the inverse of a matrix. This factorization always fills more than the Cholesky factorization of the matrix itself, so it is usually not particularly useful, and is included mainly for research purposes. One interesting aspect of this factorization is that the solve phase involves two sparse matrix-vector multiplications, as opposed to two triangular solves that constitute the solve phase of convensional triangular factorizations. This fact may make the factorization useful in certain iterative solvers, such as solvers that use support trees as preconditioners [9, 10]. For further details about the factorization, see [15]; for a different perspective, along with an analysis of fill, see [5].

The first routine computes the factor of the inverse, the second uses this factor to solve a linear system. The interface is identical to the interface of the Cholesky routines.

```
taucs_ccs_matrix* taucs_ccs_factor_xxt(taucs_ccs_matrix* A);
int               taucs_ccs_solve_xxt (void* X,
                                       double* x,
                                       double* b);
```

## 8 Iterative Linear Solvers

The iterations of conjugate gradients are cheaper than the iterations of MINRES, but conjugate gradients is only guaranteed to work on symmetric positive-definite matrices, whereas MINRES should work on any symmetric matrix. The two iterative solver routines have identical interfaces. To solve a system $Ax = b$, you pass the sparse matrix A, the addresses of the right-hand side b and of the output x, the preconditioner, and the parameters of the stopping criteria `itermax` and `convergetol`.

The iterative algorithm stops when the maximum number of iterations reaches `itermax` or when the 2-norm of the residual $b - Ax$ drops by a factor of `convergetol` or more.

The preconditioner is specified using two arguments: the address of a routine that solves $Mz = r$ for z given M and r and the address of an opaque data structure that represents M. For example, if you construct an incomplete-Cholesky preconditioner by calling `taucs_ccs_factor_llt`, the value of `precond_fn` should be `taucs_ccs_solve_llt` and the value of `precond_arg` should be the address of the incomplete triangular factor returned by `taucs_ccs_factor_llt`.

```
int taucs_conjugate_gradients(
                  taucs_ccs_matrix*  A,
                  int     (*precond_fn)(void*,double z[],double r[]),
                  void*   precond_args,
                  double  x[],
                  double  b[],
                  int     itermax,
                  double  convergetol);
int taucs_minres(taucs_ccs_matrix*  A,
                  int     (*precond_fn)(void*,double z[],double r[]),
                  void*   precond_args,
                  double  x[],
                  double  b[],
                  int     itermax,
                  double  convergetol);
```

# 9 Preconditioners for Iterative Linear Solvers

This section describes TAUCS routines that construct preconditioners for iterative linear solvers.

## 9.1 Drop-Tolerance Incomplete Cholesky

As described in Section 7.1, `taucs_ccs_factor_llt` can construct relaxed-modified and unmodified incomplete Cholesky factorizations.

## 9.2 Maximum-Weight-Basis (Vaidya's) Preconditioners

The next routine constructs a so-called Vaidya preconditioner for a symmetric diagonally-dominant matrix with positive diagonal elements. The preconditioner $M$ that is returned is simply $A$ without some of the off-diagonal nonzeros dropped and with a certain diagonal modification. To be used as a preconditioner in an iterative linear solver, you normally have to factor $M$ into its Cholesky factors. The routine accepts two parameters that affect the resulting preconditioner. The construction of $M$ is randomized and `rnd` is used as a random value. Different values result in slightly different preconditioners. Subgraphs is a number that controls how many nonzeros are dropped from $A$ to form $M$. The value 1.0 results in the sparsest possible preconditioner that this routine can construct; it will have less than $n$ offdiagonal nonzeros (for an $n$-by-$n$ matrix) and it can be factored with $O(n)$ work and fill. If all the offdiagonal nonzeros in $A$ are negative, the graph of $M$ will be a tree. The value $n$ for `subgraphs` results in $M = A$. In-between values result in in-between levels of fill. The sparsity of $M$ is roughly, but not strictly, monotone in `subgraphs`.

The routine may fail due to several reasons: failure to allocate memory, an input matrix that is not symmetric or symmetric with only the upper part stored, or an input matrix with negative diagonal elemtents. In the first case the routine returns `NULL`, in all the other cases the address of $A$.

```
taucs_ccs_matrix* taucs_amwb_preonditioner_create(
                    taucs_ccs_matrix* A,
                    int rnd,
                    double subgraphs);
```

Note that the theory of Vaidya's preconditioner only applies to symmetric diagonally-dominant matrices with positive diagonal elements, but the routine works on any symmetric matrix with

positive diagonals. Furthermore, the returned preconditioner is always symmetric and positive definite, so it should always have a Cholesky factor and, at least in theory, it should always lead to Conjugate Gradients convergence if $A$ is symmetric positive definite. We enforce the diagonal dominance of the preconditioner by always constructing a preconditioner for $A + D$, where $D$ is a diagonal matrix that brings $A + D$ to diagonal dominance. However, when $A$ is not diagonally dominant, convergence may be slow.

The next set of routines creates a so-called recursive Vaidya preconditioner. It works in the following way. It drops elements from $A$. It then finds all the rows and columns in $A$ that can be eliminated without creating much fill (elimination of degree-1 and 2 vertices until all vertices have degree 3 or more). It then eliminates these rows and columns and computes the Schur complement of $A$ with respect to them. Now it drops elements again from the Schur complement and so on. When the sparsified Schur complement is small enough, it factors it directly. In a 2-level preconditioner, in which we drop elements, compute the Schur complement, drop elements from it, and factor it directly, each preconditioning iteration requires an iterative solve for the unknowns associated with the Schur complement. The preconditioner in the inner solve is an augmented-maximum-weight-basis preconditioner. In a 3-level preconditioner, the nesting of iterative solves inside iterative solves is deeper.

The creation routine returns both a preconditioner and the reordering permutation and its inverse.

The construction depends on several parameters. The routine builds a preconditioner with at most `maxlevels` levels. It does not recurse if the matrix or Schur complement is smaller than `nsmall`. The parameters `c` and `epsilon` determine how may elements we drop from the matrix or from a Schur complement when building an augmented-maximum-weight-basis preconditioner them. A small `epsilon` $> 0$ will drop few elements, a large `epsilon` will drop many. A large `c` $< 1$ will drop few elements, a large `c` will drop many. The parameters `innerits` and `innerconv` control the accuracy of the inner iterative solves in terms of the maximum number of iteration and the convergence ratio.

We have not experimented extensively with these preconditioners and we are unsure when they are effective and how to control their construction. Therefore, the interface to the construction routine may change in the future.

```
void* taucs_recursive_mst_preconditioner_create(
        taucs_ccs_matrix* A,
        double c,
        double epsilon,
        int    nsmall,
        int    maxlevels,
        int    innerits,
        double innerconv,
        int**  perm,
        int**  invperm);
int
taucs_recursive_mst_preconditioner_solve(void* P,
                                         double* z,
                                         double* r);
```

## 9.3 Multilevel Support-Graph Preconditioners (Including Gremban-Miller Preconditioners)

TAUCS can construct a wide range of multilevel preconditioners that are called *support-graph* preconditioners. Such preconditioners were first proposed by Gremban and Miller [9, 10]. The next

routine constructs Gremban-Miller preconditioners, as well as a range of other multilevel preconditioners. This version of the documentation only documents the construction of Gremban-Miller preconditioners using this routine; its other capabilities will be described at a later date.

This routine relies on METIS and it will not work if you build the library with the `NOMETIS` option.

Also, the routine works only on symmetric diagonally-dominant matrices with negative offdiagonals.

The Gremban-Miller preconditioner is the Schur complement of a matrix whose graph is a tree. The leaves of the tree correspond to the unknowns, and the preconditioner is the Schur complement of the tree with respect to its leaves (in other words, all the internal vertices are eliminated and the reduced matrix on the leaves is the preconditioner). However, the Schur complement is not formed explicitly. Instead, the construction routine factors the entire tree matrix and uses this factor to apply the preconditioner implicitly. This ensures that the preconditioner can be factored and applied to a vector using $\Theta(n)$ work, where $n$ is the dimension of the linear system. The construction of the tree is quite expensive, however, since it involves repeated calls to graph partitioning routines in METIS.

```
void* taucs_sg_preconditioner_create(taucs_ccs_matrix *A,
                                     int* *perm,
                                     int* *invperm,
                                     char* ordering,
                                     char *gremban_command);
```

The first argument is the coefficient matrix of the linear system. The second and third arguments allow the routine to return a new ordering for the rows and columns of $A$. You should permute $A$ symmetrically using this ordering before calling the iterative solver. The third argument is ignored when this routine constructs Gremban Preconditioners; so you can pass `"identity"`. The last argument is a string that specifies the specific support-tree preconditioner that you want to construct. To construct a Gremban-Miller support tree, specify `"regular:GM:2"`. The integer at the end of the string specifies the degree of the tree's internal vertices, and we have found that high degrees lead to more efficient construction and to a more effective preconditioner (higher degrees increase the number of iterations, but reduce the cost of each iterations). It seems that values between 8 and 32 work well. The routine returns an opaque object that you can use to apply the preconditioner (or `NULL` if the construction fails):

```
int taucs_sg_preconditioner_solve(void* P,
                                  double* z,
                                  double* r);
```

The first argument of the solve routine should be the pointer that the construction routine returns. This routine solves the linear system $Pz = r$.

To free the memory associated with a support-tree preconditioner, call

```
void taucs_sg_preconditioner_free(void* P);
```

The ordering that the construction routine returns consists of two integer vectors that you can deallocate with `free()`.

# 10 Matrix Generators

TAUCS includes several matrix generators that we use to test linear solvers. The first creates a symmetric matrix that is a finite-differences discretization of $c_x \frac{\partial^2 u}{\partial x^2} + c_y \frac{\partial^2 u}{\partial y^2}$ in the unit square.

The argument `n` specifies the size of the mesh (the size of the matrix is $n^2$ and the string argument `which` specifies $c_x$, $c_y$, and the boundary conditions. The possible values of `which` are

**dirichlet** $u = 0$ on the boundary, $c_x = c_y$.

**neumann** $\frac{\partial u}{\partial n} = 0$ (the derivative in the direction normal to the boundary is 0), $c_x = c_y$. The diagonal is modified at one corner to make the matrix definite.

**anisotropic_x** $\frac{\partial u}{\partial n} = 0$, $c_x = 100c_y$, diagonal modification at a corner.

**anisotropic_y** $\frac{\partial u}{\partial n} = 0$, $100c_x = c_y$, diagonal modification at a corner.

```
taucs_ccs_matrix* taucs_ccs_generate_mesh2d(int n,char *which);
```

The second generator creates a finite-differences discretization of $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$ using an `X`-by-`Y`-by-`Z` mesh, with Neumann boundary conditions.

```
taucs_ccs_matrix* taucs_ccs_generate_mesh3d(int X, int Y, int Z);
```

The last generator creates a random `m`-by-`n` dense matrix. If `flags` is `TAU_SYMMETRIC`, the routine returns a symmetric matrix.

The library includes several additional generators that are not documented in this version.

# Changelog

**September 2003** Version 2.2. Added in this version:

- Multithreading using Cilk
- Better testing scripts and better testing programs

**August 2003** Version 2.1. Added in this version:

- A unified interface to all the linear solvers
- New configuration and build process
- Out-of-the-box support for Windows and MacOS X

**5 May 2002** Version 2.0. Added in this version:

- Complex routines, mutiple precisions, and generic routines
- Extensive automated testing for memory leaks and failure-handling

**21 January 2002** Added the $LDL^T$ factorization. It was mentioned in the documentation all along, but the code was missing from the distribution. I also added detailed information about the $LDL^T$ routines.

**12 December 2001** Version 1.0. Added in this version:

- Out-of-core sparse Cholesky and associated I/O routines.
- Relaxed and amalgamated supernodes.
- Cholesky factorization of the inverse.
- Gremban-Miller and other support-tree preconditioners (only the Gremban-Miller ones are fully documented, however).

- Faster construction of Vaidya's perconditioners when the input matrix has no positive elements outside the main diagonal. In such cases, TAUCS now uses a specialized routine that constructs a preconditioner based on maximum spanning trees rather than more general maximum weight bases. The savings depends on the matrix, but in our experiments with 2D problems the new routine is about 3 times faster than the old one.
- More matrix generators.

**26 July 2001** Added symbolic/numeric routines to allow efficient factorization of multiple systems with the same nonzero structure. Also some performance improvements to the construction of Vaidya preconditioners.

**28 June 2001** Added a routine to convert a supernodal factor to a compressed-column factor. Cleaned up memory management in construction of AMWB preconditioners; if they fail all the memory is deallocated before the routine returns.

**27 June 2001** Included missing Fortran sources in the tarball; Fixed a missing reference in the documentation; added routines to permute vectors.

**24 June 2001** Version 0.9. Initial release.

# References

[1] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadeplphia, PA, 1993.

[2] Marshall Bern, John R. Gilbert, Bruce Hendrickson, Nhat Nguyen, and Sivan Toledo. Support-graph preconditioners. Submitted to the *SIAM Journal on Matrix Analysis and Applications*, 29 pages, January 2001.

[3] Erik G. Boman. A note on recursive Vaidya preconditioners. Unpublished manuscript, February 2001.

[4] Erik G. Boman, Doron Chen, Bruce Hendrickson, and Sivan Toledo. Maximum-weight-basis preconditioners. To appear in *Numerical Linear Algebra with Applications*, 29 pages, June 2001.

[5] Robert Bridson and Wei-Pai Tang. Ordering, unisotropy, and factored sparse approximate inverses. *SIAM Journal on Scientific Computing*, 21(3):867–882, 1999.

[6] Doron Chen and Sivan Toledo. Vaidya's preconditioners: Implementation and experimental study. *Electronic Transactions on Numerical Analysis*, 16:30–49, 2003.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[8] John R. Gilbert and Sivan Toledo. High-performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San-Antonio, Texas, 1999. 10 pages on CDROM.

[9] K.D. Gremban, G.L. Miller, and M. Zagha. Performance evaluation of a parallel preconditioner. In *9th International Parallel Processing Symposium*, pages 65–69, Santa Barbara, April 1995. IEEE.

[10] Keith D. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1996. Technical Report CMU-CS-96-123.

[11] Dror Irony, Gil Shklarski, and Sivan Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. To appear in *Future Generation Computer Systems*, 16 pages, June 2002.

[12] J. H. Reif. Efficient approximate solution of sparse linear systems. *Computers and Mathematics with Applications*, 36(9):37–58, 1998.

[13] Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. Submitted to *ACM Transactions on Mathematical Software*, February 2003.

[14] Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MA. *Cilk-5.3.2 Reference Manual*, November 2001. Available online at `http://supertech.lcs.mit.edu/cilk`.

[15] H. M. Tufo and P. F. Fischer. Fast parallel direct solvers for coarse grid problems. To appear in the Journal of Parallel and Distributed Computing.

[16] Pravin M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Unpublished manuscript. A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, October 1991, Minneapolis.