SILC: Simple Interface for Library Collections

Version: 1.3 (October 31, 2007)

Introduction

This package contains the source code of Simple Interface for Library Collections (SILC), sample programs, and documentations.

SILC is a framework that allows you to use various matrix computation libraries independently of particular computing environments and programming languages.

If you write user programs in the traditional programming style based on direct calls of library functions, the user programs depend on particular libraries in use. This dependency requires you to make a considerable amount of modification to the user programs when you want to use different libraries, possibly in other computing environments.

User programs in the SILC framework, on the other hand, utilize libraries by depositing data (such as matrices and vectors) to a SILC server together with user-defined names for later reference, and making requests for computation by means of textual mathematical expressions. The mathematical expressions are translated into calls of appropriate library functions, which are carried out in the server's memory space independently of the user programs. When the results of computation are needed, the user programs retrieve them by specifying the names of data to be fetched.

By using SILC, you can easily make use of alternative libraries and computing environments without modification in user programs. In addition, user programs for SILC can be written in various programming languages because of the use of mathematical expressions to represent requests for computation.

Please refer to the following documents for the concepts, design, and implementation of SILC.

- T. Kajiyama, A. Nukada, H. Hasegawa, R. Suda, and A. Nishida. SILC: a Flexible and Environment Independent Interface to Matrix Computation Libraries. In Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005), LNCS 3911, pp.928-935, 2006.

  http://www.ssisc.org/ppam2005/paper.pdf

- T. Kajiyama, A. Nukada, H. Hasegawa, R. Suda, and A. Nishida. LAPACK in SILC: Use of a Flexible Application Framework for Matrix Computation Libraries. In Proceedings of the 8th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia 2005), November 2005.

  http://www.ssisc.org/HPCAsia2005/Kajiyama_paper.pdf

Getting started with SILC

SILC runs in Unix-like environments and Microsoft Windows. In a Unix-like environment, you need to compile the source code of SILC to obtain executable files. For Windows, a precompiled

binary package of ready-made executable files is available. See README.win32.en for more information on the binary package for Windows.

The following pieces of software are required to compile the source code of SILC in Unix-like and Microsoft Windows environments:

- C and Fortran compilers
- GNU Make
- GNU Bison
- GNU Flex

You are able to use the following matrix computation libraries by building them into SILC.

- BLAS and LAPACK (http://www.netlib.org/)
- Lis 1.0.2 (http://www.ssisc.org/lis/)
- FFTSS (http://www.ssisc.org/fftss/)

You can obtain the source code of SILC (in a compressed archive file named `silc-1.3.tar.gz`) from the following location:

> http://www.ssisc.org/silc/

The contents of the compressed archive file can be extracted as follows:

```
gzip -cd silc-1.3.tar.gz | tar xvf -
```

The source code of SILC is stored in the `silc-1.3/src/` directory in the current working directory. Move to `silc-1.3/src/` as follows:

```
cd silc-1.3/src
```

Next, you need to create a file named `make.inc`. This file defines environment-specific compiler options, locations of library files, and so on. There are example files for several computing environments in the `inc/` directory.

If you have a GNU/Linux system, you can use `inc/make.gcc` to compile SILC with GCC. Create the `make.inc` file as follows:

```
cp inc/make.gcc make.inc
```

On Microsoft Windows, you can use GCC and GNU Make in MinGW (http://www.mingw.org/) and GNU Bison and GNU Flex in GnuWin32 (http://gnuwin32.sourceforge.net/) to compile SILC. The following installer packages are used to test the current version of SILC:

- http://downloads.sourceforge.net/mingw/MSYS-1.0.10.exe
- http://downloads.sourceforge.net/mingw/MinGW-3.1.0-1.exe
- http://downloads.sourceforge.net/gnuwin32/bison-2.1.exe
- http://downloads.sourceforge.net/gnuwin32/flex-2.5.4a-1.exe

After installing these packages, create the `make.inc` file from `inc/make.mingw` as follows:

```
cp inc/make.mingw make.inc
```

For other computing environments, you need to create a `make.inc` file of your own. Please take a look at Section 2.1, "Compiling a SILC server", in SILC User's Manual for more information.

After you have created the `make.inc` file, run the make command to compile SILC:

```
make
```

Before you run sample programs, you need to start a SILC server as follows (NB: the following examples are for Unix/Linux environments; on Windows, please replace `/` [slash] with `\` [backslash]):

```
cd src/server
./server
```

Then start a sample program (for example, `src/client/demo3.c`) as follows:

```
cd src/client
./demo3
```

This program solves a system of linear equations Ax = b, where A is a tridiagonal matrix. If you get an output message like ||b-Ax|| = 3.784025e-12, the program works fine (the decimal value in the output message may vary according to the computing environment in use).

Moreover, you can use the console program (`src/client/console.c`) to interactively carry out computation by mathematical expressions:

```
./console
```

The following example shows how to compute matrix-vector products `A * b` and `b' * A` using the console program, where A is a 2-by-2 matrix, b is a 2-vector, and the single quote `'` means transposition:

```
> A = {1, 2; 3, 4}
> b = {5, 6}
> x = A * b
> pprint x
column vector, 2 elements of int
  [1] = 23
  [2] = 34
> x = b' * A
> pprint x
row vector, 2 elements of int
  [1] = 17
  [2] = 39
>
```

Letters after `>` of each line are user's input. Type Ctrl-D (in Unix-like environments, or Ctrl-Z on Windows) to exit. For more information, take a look at the README.console.en file.

Creating user programs for SILC

Please consult SILC User's Manual for the development of application programs (referred to as user programs) for SILC. The manual deals with user programs written in C and Fortran. The development of user programs for SILC on Microsoft Windows is also covered in the README.win32.en file.

You can also write user programs for SILC in object-oriented scripting language Python (http://www.python.org/). At the moment, however, there is no document about the development of user programs in Python. It is worth noting that there is a good correspondence between user programs in Python and those in C. You can find sample programs in Python in the `src/client/python/` directory.

Sample programs

There is a number of sample programs for SILC in the `src/client/` directory. Some programs are described as follows.

- `demo3_ssi_cg.c`
- `demo3.c`
- `fortran/fdemo3.f`

  These programs solve a system of linear equations Ax = b, where A is a tridiagonal matrix. In `demo3_ssi_cg.c`, library function `ssi_cg()` is directly called (i.e., without SILC) to solve the linear system with the Conjugate Gradient (CG) method; in `demo3.c` and `fortran/fdemo3.f`, the linear system is solved by means of SILC. In addition, a Python version of `demo3.c` can be found in `python/demo3.py`.

- `silc_cg.c`

  This program implements the CG method by means of mathematical expressions of SILC. The same system of linear equations as `demo3.c` is used as a problem to be solved.

- `mm_crs.c`
- `mm_band.c`

  These programs read a matrix from a file in the Matrix Market format (see http://math.nist.gov/MatrixMarket/) and solve systems of linear equations via SILC. In `mm_crs.c`, the matrix is transferred to a SILC server in the Compressed Row Storage (CRS) format, and the systems of linear equations are solved with an iterative solver (such as the CG method). In `mm_band.c`, the matrix is sent in LAPACK's banded storage format, and the linear systems are solved with LU decomposition. You can change matrix storage formats through a command-line option.

- `console.c`

  This program reads mathematical expressions from the standard input and sends them to a SILC server. That is, you can interactively use the server like a desktop calculator. Type "pprint <name>" to see the results of computation. For more information, take a look at the README.console.en file.

Open issues

The software is still under active development, so that some features are not fully implemented. Major features not implemented yet are as follows:

- Restricted assignment to sparse matrices in the CRS format (namely, assignment statements with subscript in the left-hand side).
- Elementwise multiplication and division.
- Cubic arrays.

Regression tests

There is a set of regression tests for validating the features of SILC. These tests are user programs for SILC themselves, and are written in Python. You need Python 2.4 or later to run the programs. To carry out the regression tests, simply run one program after another as follows:

```
cd src/client/python
python test_dense.py
python test_sparse_crs.py
python test_leq.py
python test_augmented.py
python test_subscript.py
python test_concat.py
python test_put_matrix.py
python test_matrix_format.py
```

Request for citation

We would like to ask you to cite the following documents in the papers you write with regard to the results of research using SILC.

- Hidehiko HASEGAWA, Reiji SUDA, Akira NUKADA, Tamito KAJIYAMA, Kengo NAKAJIMA, Daisuke TAKAHASHI, Hisashi KOTAKEMORI, Akihiro FUJII, Akira NISHIDA. Computing environment independent interface for matrix computation library. In IPSJ SIG Notes, 2004-HPC-100, pp.37-42, 2004.
- Akira NISHIDA. SSI: Overview of simulation software infrastructure for large scale scientific applications. In IPSJ SIG Notes, 2004-HPC-098, pp.25-30, 2004.

We would also appreciate if you send us any kind of results (papers, application programs, etc.) that you have using this software. Our postal and e-mail addresses are as follows:

Akira Nishida
Graduate School of Information Science and Technology,
The University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan.
E-mail: devel **at** ssisc.org

Copyright and license

This software contains the following pieces of software by other people and projects. Please use them according to the license terms and conditions specified by their copyright holders.

- `src/server/dgefa.f`, `src/server/dgesl.f`

  LINPACK from the Netlib (http://www.netlib.org/).

- `src/lib/mt/mt19937ar.c`, `src/lib/mt/mt19937ar.h`

  These files are part of Mersenne Twister, which is freely available at the following location:

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

Contact

We appreciate your comments, bug reports, feature requests, and so on. Please feel free to use the following address to write us:

The SSI Project <devel **at** ssisc.org>

What's new?

- Version 1.3 (October 31, 2007)
  - src/client/console.c: An extended version of SILC's command language was implemented. See README.console.en for the details of language extensions.
  - The following built-in functions were added.
    - coremodule: sqrt(v) for computing the square root of each element in a vector.
    - sparse_crs: sparse(i, j, v, m, n) for generating a sparse matrix.
    - sparse_crs: diag(v) and diag(v, k) for generating diagonal matrices.
    - sparse_crs: diagvec(m) for retrieving diagonal elements of a matrix.
  - The following modules were added.
    - leq_smsamg: an (experimental) arithmetic module for Super Matrix Solver AMG version 3 (VINAS Co., Ltd.)
    - leq_mp: an (experimental) arithmetic module for mp_crs, a small library of GNU MP-based multiple-precision iterative solvers.
  - New constant "inf" for expressing the double precision infinity was added.
  - An experimental implementation of the COCG method was added to the leq_lis module.
  - Elementwise multiplication and division of vectors and sparse matrices in the CRS format were implemented.
  - A bug of function norm2 in coremodule with regard to complex vectors was fixed.
  - A bug in transposes of rectangular matrices in the CRS format was fixed.
  - Bugs in matrix addition and subtraction in the CRS format were fixed.
  - The handling of modifiers (i.e., negation, transposition, complex conjugate, and subscript) was fixed so that modifiers attached to arguments of functions and procedures are correctly handled.
  - A race condition in the handlers of PUT and EXEC requests in a multithreading server was removed. This race condition could result in unexpected alterations of deposited data when PUT and EXEC requests were alternately issued.
  - src/client/mm_crs.c: A bug in computing a residual norm was fixed.
  - Lots of minor bug fixes and improvements were made.
- Version 1.2 (November 12, 2006)
  - The following modules were added.
    - leq_lis: an arithmetic module for the Lis iterative solvers library (see http://www.ssisc.org/lis/).
    - fftss: another arithmetic module for the FFTSS fast Fourier Transform library (see http://www.ssisc.org/fftss/).
    - sparse_jds: an (experimental) storage format module for the Jagged Diagonal Storage (JDS) format.
  - The performance of data communications in PUT/GET requests was improved.
  - The maximum string length of EXEC requests was increased to 4096.
  - The SILC_EXEC routine now returns an error code when syntax errors are found in EXEC requests.
  - Bugs in matrix subtraction, matrix-vector product, and matrix product for sparse matrices in the CRS format were fixed.
  - The performance of the transpose operation in the CRS format was improved.

    - Several built-in functions and procedures were added, including:
        - split: a version for sparse matrices in the CRS format (in addition to the existing one for dense matrices).
        - ones: generates a dense matrix whose elements are 1.
        - srand, rand: for initializing the Mersenne Twister random number generator and for generating a dense matrix consisting of random numbers.
        - svd: computes the singular value decomposition (SVD) using LAPACK.
    - Notations for specifying precisions of scalar literals were introduced. In addition, the precision of the constant "i" for representing the imaginary unit was changed from double precision to single precision.
    - Makefile.mingw for Microsoft Windows (MinGW) was added.
    - Lots of minor bugs were also fixed.
- Version 1.1 (November 25, 2005)
    - English translations of README (this file) and "SILC User's Manual" (doc/users_en.txt) were added.
    - Now SILC supports computing environments that do not have the feature of dynamic loading of shared object files (such as NEC SX-6i).
    - Makefile.sx for NEC SX-6i was added.
    - Some minor bugs were fixed.
- Version 1.0 (September 20, 2005)
    - The first public release.

$Id: README.en,v 1.16 2007/10/31 05:18:48 kajiyama Exp $

SILC Binary Package for Win32

Version: 1.3 (October 31, 2007)

Introduction

This file describes how to use a binary package of SILC precompiled for Win32 environments. A binary package is distributed in the form of a zipped archive having a file name like `silc-1.3-mingw32.zip`. The package contains a set of executable files, including both a SILC server and sample user programs for SILC. The package has been tested on Windows XP SP2. If you have a Windows machine and you want to try out SILC by playing with some sample user programs, the binary package will meet your needs.

Quick start

1. Extract all files in the binary package into a directory. In this document, it is assumed that the extracted files exist in the `C:\silc-1.3\` directory.

2. Open a command prompt (e.g., by selecting the *Start* menu >> *All Programs* >> *Accessories* >> *Command Prompt*) and change the current directory to the `C:\silc-1.3\src\server\` directory as follows:

   ```
   > C:
   > cd \silc-1.3\src\server
   ```

   Start a SILC server by running `server.exe` as follows:

   ```
   > server
   ```

   The server runs in the foreground, printing some debugging information as illustrated below:

   ```
   single thread
   load_modules("./modules/formats")
   silc_register_format("SILC:dense (column major)")
   silc_register_module("dense")
   silc_register_format("SILC:Band")
   silc_register_module("sparse_band")
   silc_register_format("SILC:CRS")
   silc_register_module("sparse_crs")
   silc_register_format("SILC:JDS")
   silc_register_module("sparse_jds")
   load_modules("./modules")
   silc_register_module("blasmodule")
   silc_register_module("coremodule")
   silc_register_module("leq_cg")
   silc_register_module("leq_gs")
   silc_register_module("leq_lis")
   silc_register_module("leq_lu")
   silc_register_module("linpackmodule")
   ```

3. Open another command prompt and change the current directory to `C:\silc-1.3\src\client\`. This directory contains executable files of sample user programs for SILC:

```
> C:
> cd \silc-1.3\src\client
```

Run a user program (`demo3.exe` for instance) in the new command prompt as follows:

```
> demo3
```

This program establishes a connection to the SILC server that is running in the other command prompt, and solves a system of linear equations Ax = b by means of an appropriate linear solver that the server makes available. The program prints some debugging information and terminates after displaying its execution time (in seconds) and the solution's residual norm ||b-Ax|| as illustrated below:

```
0.063935s
||b-Ax|| = 3.839510e-015
```

The actual numbers may vary; if you get similar results, the program works fine.

4. You can also use the console program (`console.exe`) in the same directory to interactively carry out matrix computations by means of SILC's mathematical expressions:

```
> console
```

The following example shows how to compute matrix-vector products `A * b` and `b' * A` using the console program, where A is a 2-by-2 matrix, b is a 2-vector, and the single quote `'` means transposition:

```
> A = {1, 2; 3, 4}
> b = {5, 6}
> x = A * b
> pprint x
column vector, 2 elements of int
  [1] = 23
  [2] = 34
> x = b' * A
> pprint x
row vector, 2 elements of int
  [1] = 17
  [2] = 39
>
```

Letters after `>` of each line are user's input. Type Ctrl-Z to exit. For more information, take a look at the README.console.en file.

Note: If you get some error message with regard to .dll files, try setting the PATH environment variable as follows:

```
> set PATH=C:\silc-1.3\src\server;%PATH%
```

Creating user programs for SILC

SILC User's Manual describes how to develop application programs (referred to as user programs) for SILC. The manual deals with user programs written in C and Fortran.

You can also write user programs for SILC in object-oriented scripting language Python (http://www.python.org/). At the moment, however, there is no document about the development

of user programs in Python. It is worth noting that there is a good correspondence between user programs in Python and those in C. You can find sample programs in Python in the `src\client\python\` directory.

What you need to do to create a user program in C and Fortran is summarized as follows:

1. Include a header file for SILC in the beginning of the user program: `src\client\client.h` is the header file for C, and `src\client\fortran\client.h` is the one for Fortran.
2. Link either `src\client\client.c` or `src\client\libsilc.a` to the user program, together with the WinSock2 library.

The rest of this section describes how to compile a sample program `mmul.c`, which computes a square of matrix A (i.e., a matrix-matrix product), by using Microsoft Visual Studio .NET 2003 as well as MinGW (GCC for Windows). The source code of `mmul.c` is as follows (please consult "SILC User's Manual" for the details of functions and constants in the code):

```
#include <stdio.h>

#include "client.h"

int main(int argc, char *argv[])
{
  silc_envelope_t object;
  double A[4] = {1.0, 2.0, 3.0, 4.0};
  double X[4];

  SILC_INIT();

  object.v = A;
  object.type = SILC_MATRIX_TYPE;
  object.format = SILC_FORMAT_DENSE;
  object.precision = SILC_DOUBLE;
  object.m = object.n = 2;
  SILC_PUT("A", &object);

  SILC_EXEC("X = A * A");

  object.v = X;
  SILC_GET(&object, "X");

  SILC_FINALIZE();

  printf("A =\n");
  printf(" %e %e\n", A[0], A[2]);
  printf(" %e %e\n", A[1], A[3]);
  printf("A * A =\n");
  printf(" %e %e\n", X[0], X[2]);
  printf(" %e %e\n", X[1], X[3]);
}
```

- Using Microsoft Visual Studio .NET 2003

  The main points of the instructions to be presented are as follows:

  - Add the following header file and C file to the project that you will be creating:
    - `src\client\client.h`
    - `src\client\client.c`
  - Change configuration properties of the project as described below:
    - *Configuration Properties* >> *Linker* >> *Input*: Add the WinSock2 library `ws2_32.lib` to *Additional Dependencies*.
    - *Configuration Properties* >> *C/C++* >> *Precompiled Headers*: Select *Not Using Precompiled Headers* from the choices in *Create/Use Precompiled Header*.

  Detailed instructions are as follows:

  1. Start Microsoft Visual Studio .NET 2003 and create a new project by selecting *Visual*

C++ Projects >> *Win32 Console Project*. Enter the project's name (`mmul` for example) in the *Name* entry.

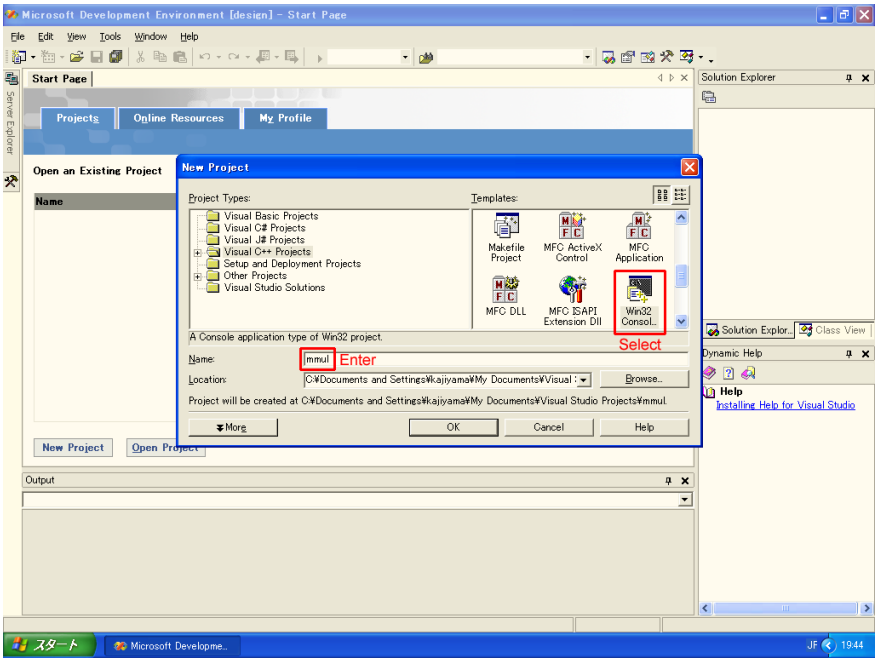

Figure 1: Creating a new project.

2. The following files are automatically created when you create the new project. These files are unnecessary in this tutorial, so delete them as well as their folders.

   - `Source Files\mmul.cpp`
   - `Source Files\stdafx.cpp`
   - `Header Files\stdafx.h`
   - `ReadMe.txt`

   Note that only references to the files are deleted in this step. The actual files can be deleted in the next step.

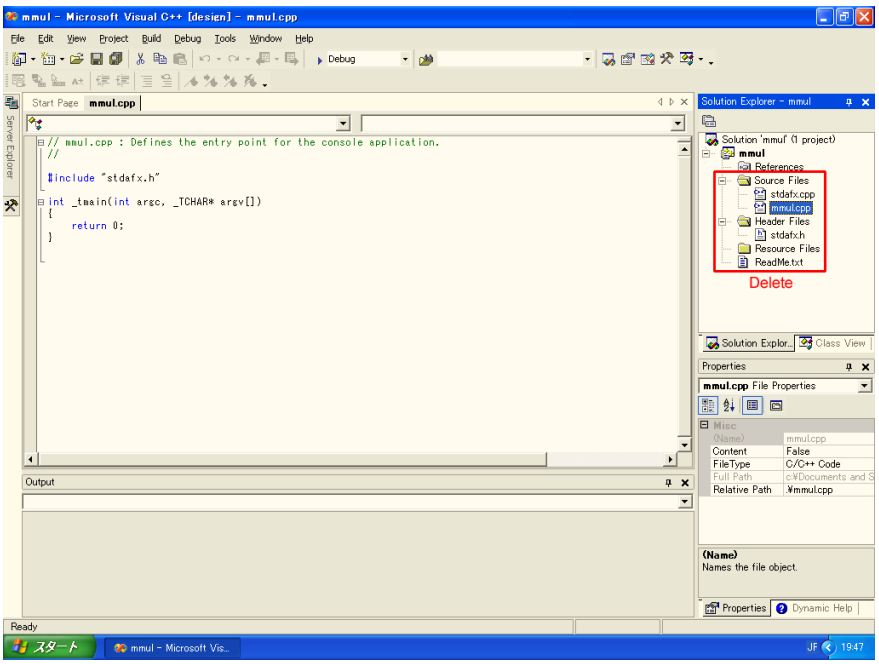

Figure 2: Removing unnecessary files.

3. Move or copy the following files to the `Visual Studio Project\mmul\` directory in *My Documents*:

   - `mmul.c`
   - `src\client\client.c`
   - `src\client\client.h`

   Also delete the four files listed in Step 2 at this point of time.

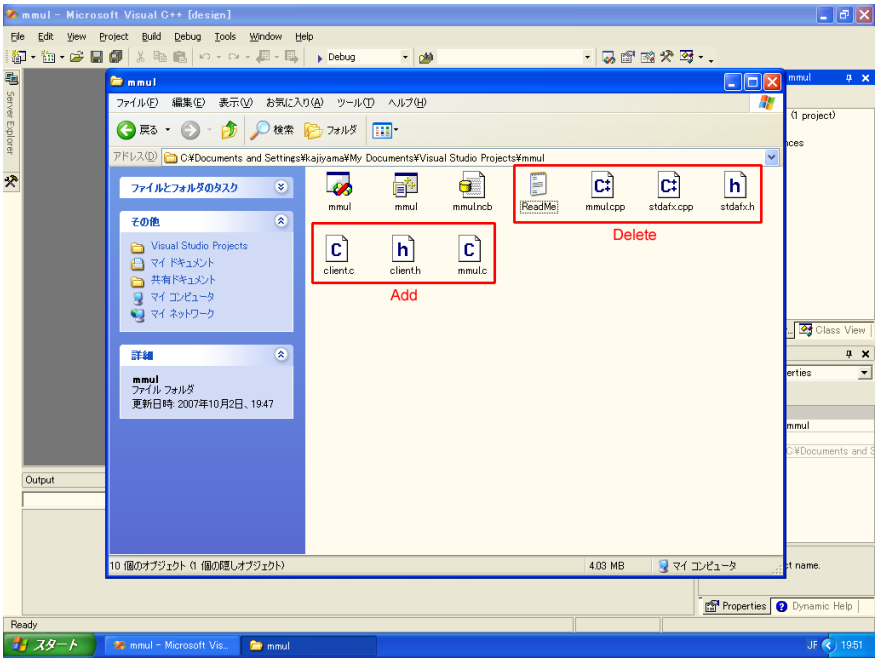

Figure 3: Adding and removing files.

4. Add the three files listed in Step 3 by selecting *Project* >> *Add Existing Item*.

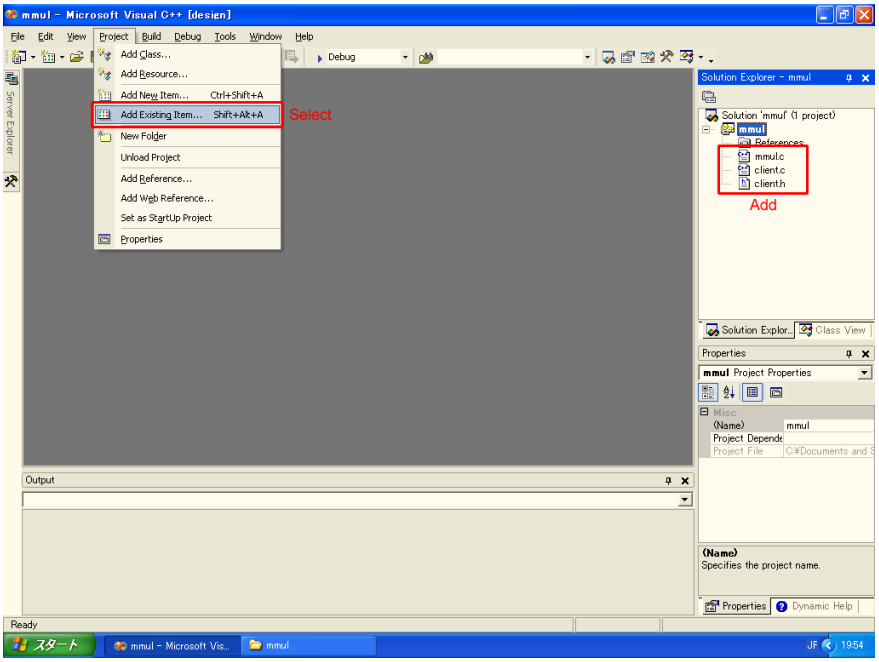

Figure 4: Adding existing items.

5. Select *Project* >> *Properties* (or press the highlighted button in Figure 5) and change properties as described below:

   - *Configuration Properties* >> *Linker* >> *Input*: Add the WinSock2 library `ws2_32.lib` to *Additional Dependencies*.
   - *Configuration Properties* >> *C/C++* >> *Precompiled Headers*: Select *Not Using Precompiled Headers* from the choices in *Create/Use Precompiled Header*.

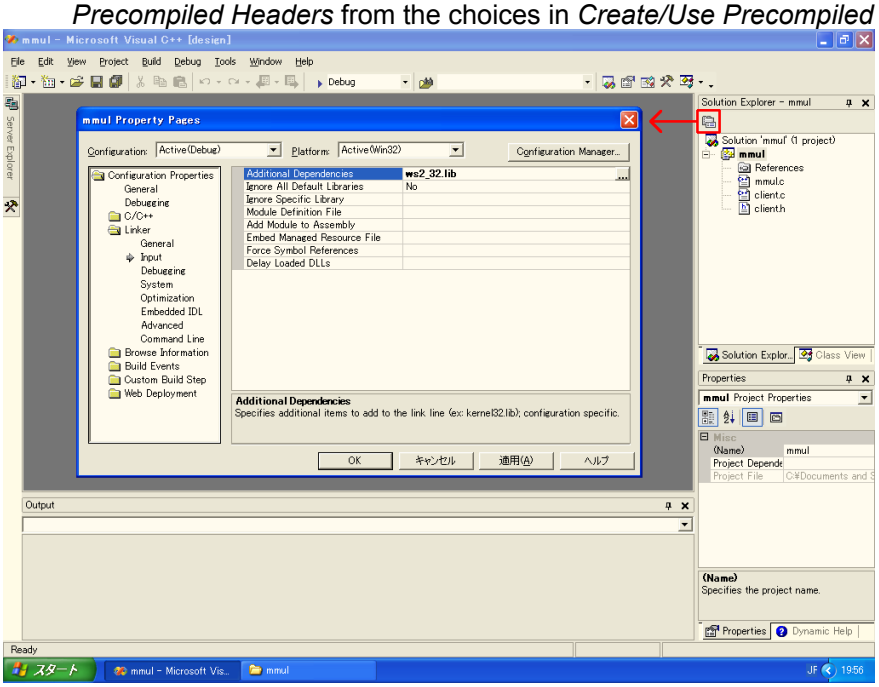

Figure 5: Changing properties of the project.

6. Compile and link the program by selecting *Build* >> *Build Solution*. If compilation and linking have succeeded, executable file `mmul.exe` is created in the `Visual Studio Projects\mmul\Debug\` directory in *My Documents*.

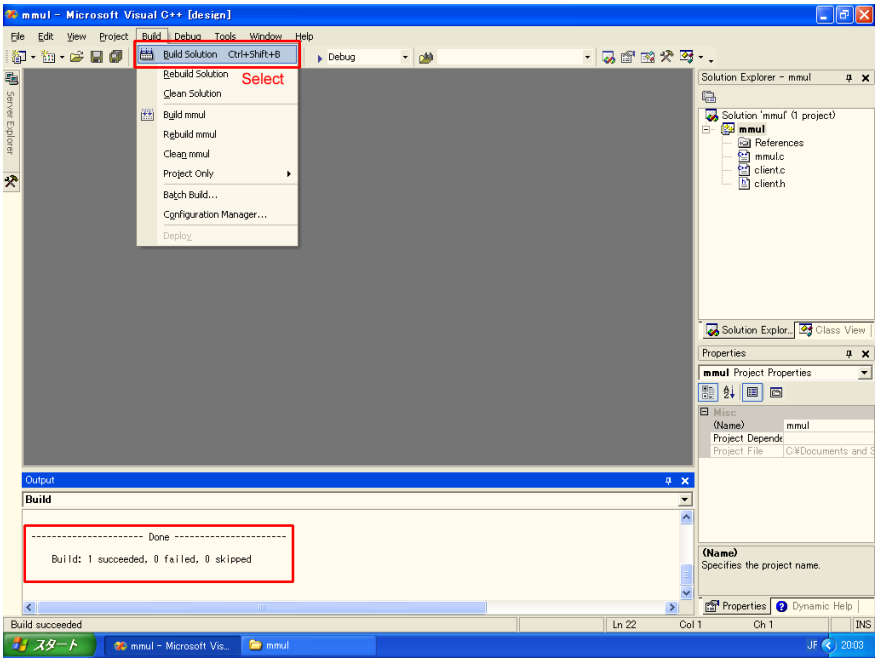

Figure 6: Building the user program.

7. Open a command prompt by selecting the *Start* menu >> *All Programs* >>

Accessories >> *Command Prompt* and start a SILC server as follows:

```
> C:
> cd \silc-1.3\src\server
> server
```

Open another command prompt and run the user program as follows:

```
> C:
> cd "My Documents\Visual Studio Projects\mmul\Debug"
> mmul
connected to kajiyama on port 1639
number of formats = 4
  0: "SILC:dense (column major)"
  1: "SILC:Band"
  2: "SILC:CRS"
  3: "SILC:JDS"
Request: >A
Response: 200 OK
Request: :9:X = A * A
Response: 200 OK
Request: <X
Response: 200 OK
A =
 1.000000e+000 3.000000e+000
 2.000000e+000 4.000000e+000
A * A =
 7.000000e+000 1.500000e+001
 1.000000e+001 2.200000e+001
>
```

If you get similar results to the above, the program works fine.

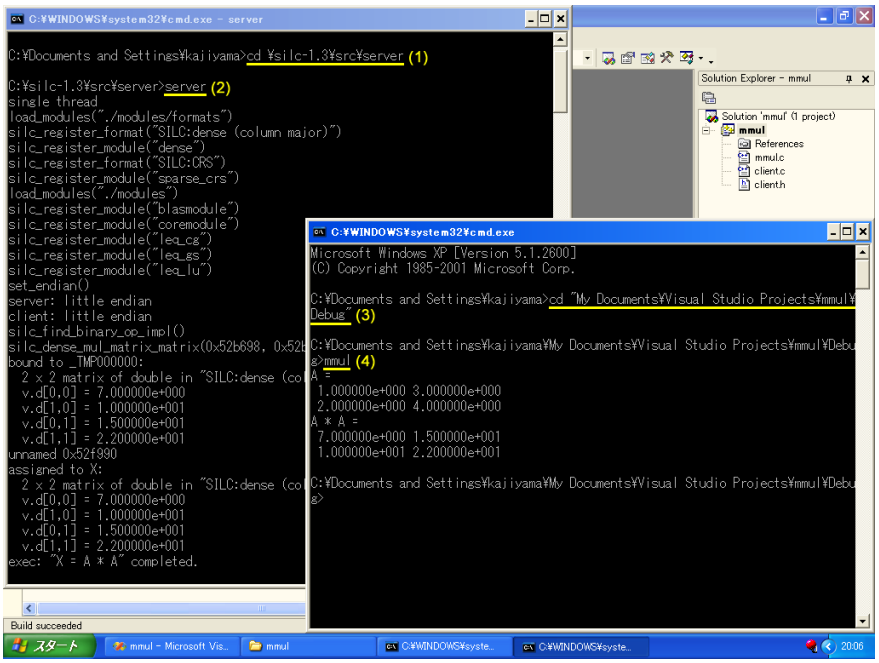

Figure 7: Running a SILC server and the user program.

- Using MinGW (GCC 3.2.3)

  MinGW (http://www.mingw.org/) is a Windows version of GCC. You can obtain a complete installation of GCC 3.2.3 by downloading and running the following installers in this order:

  - http://downloads.sourceforge.net/mingw/MSYS-1.0.10.exe

- http://downloads.sourceforge.net/mingw/MinGW-3.1.0-1.exe

Instructions on how to compile `mmul.c` using MinGW are shown below:

1. Open a command prompt by selecting the *Start* menu >> *All Programs* >> *Accessories* >> *Command Prompt* and change the current working directory to the directory where `mmul.c` exists.

2. Run the gcc command to compile `mmul.c` as follows:

```
> gcc -IC:/silc-1.3/src/client -c mmul.c
> gcc -LC:/silc-1.3/src/client -o mmul mmul.o -lsilc -lws2_32
```

   The `-I` and `-L` options specify the directories where `client.h` and `libsilc.a` exist. The `-lws2_32` option is needed to link the WinSock2 library to the user program.

   If you want to modify the user program and compile it repeatedly, it is worth creating `Makefile` shown below:

```
all: mmul

SILC=    C:/silc-1.3

CC=      gcc
CFLAGS=  -I$(SILC)/src/client
LDFLAGS= -L$(SILC)/src/client
LIBS=    -lws2_32

mmul: mmul.c
        $(CC) $(CFLAGS) -c mmul.c
        $(CC) $(LDFLAGS) -o $@ mmul.o -lsilc $(LIBS)
```

   With the above file placed in the same directory as `mmul.c`, run the make command to compile the user program as follows:

```
> make
```

3. Open another command prompt and run a SILC server:

```
> C:
> cd \silc-1.3\src\server
> server
```

   Then, run the user program in the first command prompt:

```
> mmul
connected to kajiyama on port 1639
number of formats = 4
  0: "SILC:dense (column major)"
  1: "SILC:Band"
  2: "SILC:CRS"
  3: "SILC:JDS"
Request: >A
Response: 200 OK
Request: :9:X = A * A
Response: 200 OK
Request: <X
Response: 200 OK
A =
 1.000000e+000 3.000000e+000
 2.000000e+000 4.000000e+000
A * A =
 7.000000e+000 1.500000e+001
```

```
 1.000000e+001 2.200000e+001
>
```

If you get similar results to the above, the program works fine.

Copyright and license

The copyright holder of this software is the SSI Project. This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY. See the `LICENSE` file for the precise terms and conditions for the use of this software.

This software contains the following pieces of software by other people and projects. Please use them according to the license terms and conditions specified by their copyright holders.

- `src\server\libblas.dll`, `src\server\liblapack.dll`

  BLAS and LAPACK from the Netlib (http://www.netlib.org/).

- `src\server\dgefa.f`, `src\server\dgesl.f`

  LINPACK from the Netlib (http://www.netlib.org/).

- `src\lib\mt\mt19937ar.c`, `src\lib\mt\mt19937ar.h`

  These files are part of Mersenne Twister, which is freely available at the following location:

  http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

Contact

We appreciate your comments, bug reports, feature requests, and so on. Please feel free to use the following address to write us:

The SSI Project <devel **at** ssisc.org>

$Id: README.win32.en,v 1.11 2007/10/31 05:18:48 kajiyama Exp $

Command language extensions in the console program

Last modified: October 31, 2007

Introduction

This document describes the extensions of SILC's command language in the console program (`src/client/console.c`). The console program allows you to (1) interactively execute mathematical expressions and (2) execute a sequence of mathematical expressions stored in a file. The program has made several language extensions to the command language so that conditional branching and loops can be written. Using this program you can write user programs without compilers of other programming languages such as C and Fortran.

Conditional branching

Conditional branching is written in the following form:

```
if (cond_expr) {
  stmt; ...
} else if (cond_expr) {
  stmt; ...
} else {
  stmt; ...
}
```

where `cond_expr` is a conditional expression (described below) and `stmt` is a statement. Else clauses may be omitted. Braces cannot be omitted.

Loops, continue and break statements

Loops are written in the following form:

```
while (cond_expr) {
  stmt; ...
}
```

where `cond_expr` is a conditional expression (described below) and `stmt` is a statement. Braces cannot be omitted. Within loops you can use continue and break statements.

Conditional expressions

Conditional expressions (`cond_expr`) are composed of the following comparison operators.

- expr < expr
- expr > expr
- expr <= expr
- expr >= expr
- expr == expr
- expr != expr

where `expr` is a mathematical expression in SILC's command language. The value of the expression must be a scalar value. Values of other data types (such as vectors and matrices) result in a runtime error. In addition, conditional expressions can be combined with the following boolean operators and parentheses.

- cond_expr or cond_expr
- cond_expr and cond_expr
- not cond_expr
- (cond_expr)

These comparison and boolean operators yield boolean values. Any expressions that do not have boolean values cannot be used as conditional expressions. Moreover, `true` and `false` can be used as boolean constants.

Conditional expressions are handled in the console program as follows:

1. Execute an assignment statement for each of the two expressions specified as the left- and right-hand sides of a comparison operator, and store the value of the expression to a temporary variable `_` as follows:

   ```
   SILC_EXEC("_ = expr")
   ```

2. Fetch the value of the temporary variable from the SILC server:

   ```
   SILC_GET(&tmp, "_")
   ```

3. Compare the values of the two expressions (fetched one after another) on the client side.

Extended system statements

In addition to the system statements defined in SILC's command language, the following extended system statements can be used.

- load "filename", variable

  Read data (such as matrices and vectors) from a specified file in the Matrix Market format, and deposit the data to the SILC server with a specified variable name. Examples:

  ```
  load "filename.mtx", A
  load "filename.mtx", b
  ```

- save "filename", variable

  Fetch the data of a specified variable name from the server, and store the data into a file in the Matrix Market format. Example:

  ```
  save "filename.mtx", x
  ```

- pprint expr

  Fetch the value of a specified mathematical expression `expr` from the server, and pretty-print it on the client's standard output.

- message "string"

  Print a specified string on the client's standard output.

Miscellaneous

Conditional branching, loops, and the extended system statements are carried out on the client side. Any other statements (namely, ordinary statements in SILC's command language) are carried out in the SILC server. This implies that all data are maintained by the server (that is, the console program has no mechanism for data management).

Statements can be concatenated by semicolons. The last semicolon at the end of a line can be omitted.

Lines can be concatenated by placing `\` at the end of a line. For example, the following statement is treated as written in a single line:

```
B = {1, 2, 3; \
     4, 5, 6; \
     7, 8, 9}
```

Any input from `#` to the end of a line is treated as a comment (simply ignored).

Example

The following is a script that realizes the Conjugate Gradient (CG) method using the aforementioned extended command language:

```
# create tridiagonal matrix A and vector b
n = 400
A = diag(2.0 * ones(n, 1)) - diag(ones(n-1, 1), 1) - diag(ones(n-1, 1), -1)
b = A * (-ones(n, 1))

# solve the linear system Ax=b with the CG method
rho_old = 1.0
p = zeros(n, 1)
x = zeros(n, 1)
r = b
bnrm2 = 1.0 / norm2(b)
iter = 1
while (iter <= n) {
  rho = r' * r
  beta = rho / rho_old
  p = r + beta * p
  q = A * p
  alpha = rho / (p' * q)
  r = r - alpha * q
  nrm2 = norm2(r) * bnrm2
  x = x + alpha * p
  if (nrm2 <= 1.0e-12) {
    break
  }
  rho_old = rho
```

```
  iter += 1
}

# store the solution x into a file
save "sol.mtx", x

# print the iteration count
message "number of iterations:"
pprint iter
```

To run this script, store the script into a file (`silc_cg.txt` for example), start a SILC server, and run the console program as follows:

```
console silc_cg.txt
```

If everything works fine, the script terminates when the number of iterations reaches 200.

Revision history

- October 31, 2007
  - File names were updated according to the new directory structures in SILC v1.3.
- February 13, 2007
  - Initial version.

$Id: README.console.en,v 1.5 2007/10/31 05:18:48 kajiyama Exp $

SILC User's Manual

The SSI Project

Revision: October 31, 2007 (for SILC v1.3)

Table of Contents

1. Introduction

This document provides technical details required to create application programs (referred to as user programs) for SILC. The document first explains how to compile and run a SILC server and how to create user programs in C and Fortran. Then, the specifications of SILC's mathematical expressions are described, and finally the details of APIs are summarized.

2. SILC server

2.1. Compiling a SILC server

The following pieces of software are required to compile a SILC server and run it in a Unix-like or Microsoft Windows environment:

- C and Fortran compilers
- GNU Make
- GNU Bison

- GNU Flex

You are able to use the following matrix computation libraries by building them into the SILC server.

- BLAS, LAPACK (http://www.netlib.org/)
- Lis 1.0.2 (http://www.ssisc.org/lis/)
- FFTSS (http://www.ssisc.org/fftss/)

SILC has been tested in the following computing environments (in the parentheses, templates of the `make.inc` file (described below) for each environment are shown), together with the operating systems and compilers shown in Table 1. The column "OpenMP" shows whether support for OpenMP is available in each platform.

Table 1. Tested platforms

Computing environment	OS	Compilers	OpenMP
Sun Fire 3800 (make.sunfire)	Solaris 9 (sparc)	Sun ONE Studio 7	Yes
SGI Altix 3700 (make.altix)	Red Hat Linux Advanced Server 2.1	Intel C 9.1 Intel Fortran 9.1	Yes
IBM eServer xSeries 335 (make.linux-icc-32)	Red Hat Linux 8.0	Intel C 9.0 Intel Fortran 9.0	Yes
Dell PowerEdge SC 1420 (make.linux-icc-64)	Fedora Core 4	Intel C 9.0 (EM64T) Intel Fortran 9.0 (EM64T)	Yes
IBM OpenPower 710 (make.openpower)	SuSE Linux Enterprise Server 9 (ppc)	IBM XL C 7.0 IBM XL Fortran 9.1	Yes
Apple PowerMac G5 (make.g5)	Mac OS X 10.4.2	IBM XL C 6.0 IBM XL Fortran 8.1	Yes
NEC SX-6i (make.sx)	SUPER-UX 13.1 SX-6	C++/SX 1.0 for SX-6 FORTRAN90/SX 2.0 for SX-6	No
Panasonic CF-R3 (make.gcc)	Fedora Core 3	GCC 3.4.2	No
IBM ThinkPad T42 (make.gcc)	KNOPPIX 4.0 LinuxTag Japanese Edition	GCC 3.3.6	No
Dell Dimension 84000 (make.mingw)	Microsoft Windows XP Professional SP2	MinGW (GCC 3.2.3)	No

You can obtain the source code of SILC (in a compressed archive file named `silc-1.3.tar.gz`) from the following location:

http://www.ssisc.org/silc/

The contents of the compressed archive file can be extracted as follows. The source code of SILC is stored in the `silc-1.3/src/` directory in the current working directory.

```
$ gzip -cd silc-1.3.tar.gz | tar xvf -
```

To compile the source code of SILC, you need to create a file named `make.inc` in the `silc-1.3/src/` directory. This file defines environment-specific compiler options, locations of library files, and so on. There are template files for several computing environments in the `silc-1.3/src/inc/` directory. See Table 1 for the names of the template files, target platforms, operating systems, and compilers to be used.

In the `make.inc` file, you define a series of macros that are referenced from `Makefile`, the input file of the **make** command. The macros are roughly divided into two groups, namely a group of general macros that

do not depend on SILC (Table 2) and another of SILC-specific macros (Table 3).

Table 2. General macros

CC	Command name of a C compiler.
FC	Command name of a Fortran compiler.
LINK.f	Command name of a linker to be used for linking Fortran programs (default: the command name specified by FC).
BISON	Command name of GNU Bison.
FLEX	Command name of GNU Flex.
CFLAGS	Compiler options for the C compiler.
FFLAGS	Compiler options for the Fortran compiler.
LDFLAGS	Linker options for the C and Fortran compilers.
SHAREDCFLAGS	C compiler options required when compiling shared libraries.
SHAREDFFLAGS	Fortran compiler options required when compiling shared libraries.
SHAREDLDFLAGS	C/Fortran compiler options required when linking shared libraries.
RANLIB	Name of the **ranlib** command (for example, the **echo** command can be specified if the target computing environment does not have the **ranlib** command).
RM	Name of the **rm** command (usually not needed defining).

Table 3. SILC-specific macros

OMPFLAGS	C/Fortran compiler options for enabling OpenMP-based parallelization.
LIBS	Linker options for linking the SILC server and sample user programs (e.g., libraries to be linked).
PLATFORM_MODULES	A list of environment-specific arithmetic matrix storage modules modules that you want to build into the SILC server. The following modules can be specified (the libraries that each module depends on, if any, are shown in the parentheses). See Section 4.7 for the details of the modules. • $(FORMAT_DIR)/sparse_band.so (BLAS, LAPACK) • $(MODULE_DIR)/blasmodule.so (BLAS, LAPACK) • $(MODULE_DIR)/leq_lis.so (Lis) • $(MODULE_DIR)/fftss.so (FFTSS) • $(MODULE_DIR)/linpackmodule.so (LINPACK)
PLATFORM_TESTS	A list of environment-specific sample user programs that you want to build. The following programs can be specified (the libraries that each program depends on, if any, are shown in the parentheses). • test_dense (BLAS) • test_dense_sa (BLAS) • test_dot • test_dot_sa (BLAS) • test_band • test_band_sa (BLAS, LAPACK)

	• mm_band (BLAS) • mm_band_sa (BLAS, LAPACK) • mm_lis (Lis)
BLAS_DRIVER	BLAS driver name (described below).
BLAS_CFLAGS	C compiler options for compiling the programs that use BLAS.
BLAS_LIBS	Linker options for linking the programs that use BLAS (e.g., libraries to be linked).
LAPACK_DRIVER	LAPACK driver name (described below).
LAPACK_CFLAGS	C compiler options for compiling the programs that use LAPACK.
LAPACK_LIBS	Linker options for linking the programs that use LAPACK (e.g., libraries to be linked).
LIS_CFLAGS	C compiler options for compiling the programs that use Lis.
LIS_LD	Name of a linker for linking the programs that use Lis.
LIS_LIBS	Linker options for linking the programs that use Lis (e.g., libraries to be linked).
LINPACK_LIBS	Linker options for linking the programs that use LINPACK (e.g., libraries to be linked).

Subroutines of BLAS and LAPACK are used through a driver file. There are three pairs of BLAS and LAPACK drivers shown below. Choose one of them according to the implementations of BLAS and LAPACK you want to use.

- Drivers for Intel Math Kernel Library (MKL)

  `blas_intelmkl.c`, `lapack_intelmkl.c`

- Drivers for Sun Performance Library

  `blas_sunperf.c`, `lapack_sunperf.c`

- Dumb (generic) drivers

  `blas_dumb.c`, `lapack_dumb.c`

You have to define the following macros in the `make.inc` file to specify the drivers, compiler options and linker options.

BLAS_DRIVER	BLAS driver file name
BLAS_CFLAGS	C compiler options for BLAS
BLAS_LIBS	Linker options for BLAS
LAPACK_DRIVER	LAPACK driver file name
LAPACK_CFLAGS	C compiler options for LAPACK
LAPACK_LIBS	Linker options for LAPACK

Examples of macro definitions for the dumb drivers are shown below.

```
BLAS_DRIVER=    blas_dumb.c
BLAS_CFLAGS=
BLAS_LIBS=      /opt/LAPACK/blas.a

LAPACK_DRIVER=  lapack_dumb.c
LAPACK_CFLAGS=  $(BLAS_CFLAGS)
LAPACK_LIBS=    /opt/LAPACK/lapack.a $(BLAS_LIBS)
```

The SILC server and user programs will generate a lot of debugging messages, since the template files in the `silc-1.3/src/inc/` directory define the `CFLAGS` macro with the `-DDEBUG` option enabled by default.

You can suppress the messages by editing `make.inc` and removing the option.

After you have created `make.inc` in the `silc-1.3/src/` directory, run the **make** command to compile the SILC server and related programs as follows.

```
$ make
```

2.2. Running and stopping a SILC server

Before you run a user program for SILC, you need to start a SILC server. Type the following command to run a SILC server (you also need to change the current working directory to the directory where the executable file of the SILC server exists). The server starts in the foreground.

```
$ cd src/server
$ ./server
```

In shared-memory parallel computing environments, you can use an OpenMP-based parallel SILC server. The `OMP_NUM_THREADS` environment variable is used to specify the number of threads as follows.

```
$ env OMP_NUM_THREADS=4 ./server
```

Unless explicitly specified, the port number through which the server accepts connections from user programs will vary every time you restart the server. The `SILC_PORT` environment variable is used to specify the port number as illustrated below.

```
$ env SILC_PORT=32000 ./server
```

The server creates a plain text file `~/.silc` (if it does not exist) in your home directory and writes out the server's host name and port number. The `SILC_INIT` routine (Section 3.1) reads the file and establishes a connection to the server according to the host name and port number in the file. The contents of the file is as follows.

```
hostName portNumber [EOF]
```

If the SILC server and user programs run in different computing environments with separate file systems, you have to manually create `~/.silc` in the file system from which the user programs (or precisely speaking, the `SILC_INIT` routine) will read the file. The **echo** command will do for this task, as shown in the following example.

```
$ echo hostName portNumber > ~/.silc
```

To stop the SILC server, simply type **Ctrl-C** or use the **kill** command as follows.

```
$ kill -9 processNumber
```

3. User programs for SILC

3.1. Writing user programs

User programs for SILC establish a connection to a SILC server over networks, and make use of matrix computation libraries the SILC server maintains, by calling the following routines. These routines are referred as client routines in this document. The arguments of the client routines are described in detail in Section 5.

Client routines for C:

- `SILC_INIT`

  Establishes a connection to a SILC server.

- `SILC_PUT`

  Sends data (such as matrices and vectors) to the server.

- `SILC_EXEC`

  Sends a request for computation by means of mathematical expressions in the form of text. The mathematical expressions are written in SILC's command language (described in Section 4).

- `SILC_GET`

  Receives the results of the computation.

- `SILC_FINALIZE`

  Closes the connection to the server.

Client routines for Fortran:

- `SILC_INIT`

  Establishes a connection to a SILC server.

- `SILC_PUT_SCALAR`, `SILC_PUT_MATRIX`, `SILC_PUT_MATRIX_CRS`, etc.

  Send data (such as matrices and vectors) to the server. Unlike the client routines for C, there is a separate PUT routine for each data type.

- `SILC_EXEC`

  Sends a request for computation by means of mathematical expressions in the form of text. The mathematical expressions are written in SILC's command language (described in Section 4).

- `SILC_GET_SCALAR`, `SILC_GET_MATRIX`, `SILC_GET_MATRIX_CRS`, etc.

  Receive the results of the computation. Like the PUT routines above, there is a separate GET routine for each data type.

- `SILC_FINALIZE`

  Closes the connection to the server.

The client routines for both C and Fortran are defined in `silc-1.3/src/client/libsilc.a`. By linking the library file to user programs, these client routines are made available. `libsilc.a` is built together with the SILC server.

In addition, data structures and constants used with the client routines (see Section 5). are defined in `silc-`

`1.3/src/client/client.h` (for C) and `silc-1.3/src/client/fortran/client.h` (for Fortran).

Although user programs can be either sequential or multithreading parallel programs, all calls for client routines must be made in the same thread.

An example of a user program in C (`solve.c`) is shown below. This program solves a system of linear equation Ax = b, where A is a sparse matrix in the Compressed Row Storage (CRS) format.

```
#include "client.h"

int main(int argc, char *argv[])
{
  silc_envelope_t object;  /* a structure used for data communications */
  double *value, *b, *x;
  int *index, *row;

  /* Create sparse matrix A (in the CRS format) and vector b */

  SILC_INIT();

  object.v = value;
  object.type = SILC_MATRIX_TYPE;
  object.format = SILC_FORMAT_CRS;
  object.precision = SILC_DOUBLE;
  object.m = object.n = N;  /* dimensions */
  object.nnz = NNZ;  /* the number of non-zero elements */
  object.row = row;
  object.index = index;
  SILC_PUT("A", &object);

  object.v = b;
  object.type = SILC_COLUMN_VECTOR_TYPE;
  object.precision = SILC_DOUBLE;
  object.length = N;
  SILC_PUT("b", &object);

  /* Solve a system of linear equations Ax = b */
  SILC_EXEC("x = A \\ b");

  object.v = x;
  SILC_GET(&object, "x");

  SILC_FINALIZE();

  /* Output the solution (vector x) */
}
```

A Fortran program (`solve.f`) that carries out the same computation as the C program above is shown below.

```
    INCLUDE 'client.h'

    REAL *8 VALUE(NNZ), B(N), X(N)
    INTEGER *4 ROW(N+1), INDEX(NNZ), IERR

C   Create sparse matrix A (in the CRS format) and vector b

    CALL SILC_INIT(IERR)

    CALL SILC_PUT_MATRIX_CRS('A', VALUE, N, N, NNZ, ROW, INDEX,
   &                         SILC_DOUBLE, IERR)

    CALL SILC_PUT_COLUMN_VECTOR('b', B, N, SILC_DOUBLE, IERR)

C   Solve a system of linear equations Ax = b
    CALL SILC_EXEC('x = A \ b', IERR)

    CALL SILC_GET_COLUMN_VECTOR('x', X, IERR)
```

```
      CALL SILC_FINALIZE(IERR)

C     Output the solution (vector x)
```

3.2. Compiling user programs

To compile user programs for SILC with a C/Fortran compiler, you need to specify (1) compiler options for locating the header file (`client.h`) and library file (`libsilc.a`), and (2) linker options for specifying the libraries to be linked (`libsilc.a` and additional libraries described below).

For example, GNU C compiler is used to compile the C program (`solve.c`) in the previous section by running the **gcc** command as follows. In this example, it is assumed that the source code of SILC is placed in the `~/silc-1.3/src/` directory in your home directory.

```
$ gcc -I~/silc-1.3/src/client -c solve.c
$ gcc -L~/silc-1.3/src/client -o solve solve.o -lsilc
```

The `-I` and `-L` options specify the locations of `client.h` and `libsilc.a` respectively, and `-lsilc` is a linker option for linking `libsilc.a` to the user program.

For your convenience, if you repeatedly modify the user program and compile it, it is worth creating `Makefile` like the one shown below.

```
all: solve

CC=      gcc
CFLAGS=  -I$$HOME/silc-1.3/src/client
LDFLAGS= -L$$HOME/silc-1.3/src/client
LIBS=

solve: solve.c
        $(CC) $(CFLAGS) -c solve.c
        $(CC) $(LDFLAGS) -o $@ solve.o -lsilc $(LIBS)
```

To compile `solve.c` using `Makefile` above, run the **make** command as follows:

```
$ make
```

GNU Fortran compiler is used to compile the Fortran program (`solve.f`) in the previous section by running the **g77** command as follows:

```
$ g77 -I~/silc-1.3/src/client/fortran -fno-second-underscore -c solve.f
$ g77 -L~/silc-1.3/src/client -o solve solve.o -lsilc
```

As it is in the case of the aforementioned C program, it is convenient to create `Makefile` like the following one:

```
all: solve

FC=      g77 -fno-second-underscore
FFLAGS=  -I$$HOME/silc-1.3/src/client/fortran
LDFLAGS= -L$$HOME/silc-1.3/src/client
LIBS=

solve: solve.f
        $(FC) $(FFLAGS) -c solve.f
        $(FC) $(LDFLAGS) -o $@ solve.o -lsilc $(LIBS)
```

Some computing environments require additional libraries to be linked to user programs together with `libsilc.a` in order to use networking facilities. Several computing environments and additional libraries to be required are shown below. In the case of `Makefile` for C and Fortran shown above, specify the additional libraries with the `LIBS` macro.

Table 4. Several computing environments and additional libraries for networking.

Computing environment	Additional libraries
Solaris	`-lsocket -lnsl`
GNU/Linux	None
Microsoft Windows (MinGW)	`-lws2_32`
Mac OS X	None

3.3. Data types and precisions

There are five types of data that are transferred between a SILC server and user programs. The constants (shown in the parentheses) are used to specify the data types when calling the PUT/GET routines. These constants are defined in `client.h` (see Section 5).

- Scalar (`SILC_SCALAR_TYPE`)
- Column vector (`SILC_COLUMN_VECTOR_TYPE`)
- Row vector (`SILC_ROW_VECTOR_TYPE`)
- Matrix (`SILC_MATRIX_TYPE`)
- Cubic (3-dimensional) array (`SILC_CUBIC_ARRAY_TYPE`)

The table below summarizes precisions supported in SILC. The constants (shown in the parentheses) are used to specify the precisions when calling the PUT/GET routines. The corresponding data types in C and Fortran for each precision are also shown.

Table 5. Precisions in SILC.

Precision	C	Fortran
Single precision integer (`SILC_INT`)	`int`	`INTEGER*4`
Double precision integer (`SILC_LONG`)	`long`	`INTEGER*8`
Single precision real (`SILC_FLOAT`)	`float`	`REAL*4`
Double precision real (`SILC_DOUBLE`)	`double`	`REAL*8`
Single precision complex (`SILC_COMPLEX`)	`float`[a]	`COMPLEX*8`
Double precision complex (`SILC_DOUBLE_COMPLEX`)	`double`[a]	`COMPLEX*16`

[a] In C, both the real and imaginary parts of a complex number are represented by a real number. An array of N complex numbers are represented as an array of $2N$ real numbers that stores real and imaginary parts alternatively.

3.4. Matrix storage formats

When transferring a matrix, you have to specify the storage format of the matrix together with the data type (`SILC_MATRIX_TYPE`). Two matrix storage formats are currently supported as described below.

- `SILC_FORMAT_DENSE`

  This format is used to represent dense matrices. Elements of a dense matrix are stored in a 2-

dimensional array in the Fortran style. The format consists of the following attributes and array.

m

The number of rows (integer).

n

The number of columns (integer).

value

An array of m-by-n elements (any precision).

The elements are stored in the Fortran style (i.e., stored column by column). Note that 2-dimensional arrays in the C style store elements row by row.

- `SILC_FORMAT_CRS`

This format, called the Compressed Row Storage (CRS) format, is used to represent sparse matrices compactly by storing only non-zero elements (those whose values are not zero) row by row. The format consists of the following attributes and arrays.

m

The number of rows (integer).

n

The number of columns (integer).

nnz

The number of non-zero elements (integer).

value

An array of *nnz* elements (any precision).

This array stores non-zero elements row by row, leaving no space.

row

An array of *m*+1 elements (single precision integer).

This array stores a pointer (i.e., an index in the `value` array) to the first element of each row; that is, `row[0]` stores a pointer to the first element of row 1, `row[1]` stores a pointer to the first element of row 2, and so forth. `row[m]` stores the number of non-zero elements (i.e., *nnz*).

index

An array of *nnz* elements (single precision integer).

Each element of this array represents a column number of the non-zero element stored in the corresponding index in the `value` array. For example, if `value[7]` holds the non-zero element at column 5 of a row in a matrix, then `index[7]` holds the column number 5.

Example: The following 4-by-4 matrix is represented by the three arrays value, row, index, as shown below.

```
| 11  0  0  0 |
|  0 22  0  0 |
|  0 32 33  0 |
| 41  0  0 44 |
```

- value: 11, 22, 32, 33, 41, 44 (length 6)
- row: 0, 1, 2, 4, 6 (length 5)
- index: 1, 2, 2, 3, 1, 4 (length 6)

4. The command language

The argument of the `SILC_EXEC` routine is a mathematical expression in the form of text that instructs a SILC server to carry out matrix computations. A mathematical expression is a kind of programs written in SILC's command language.

The unit of computation to be carried out at once is a statement, which is an assignment statement, a procedure call, or a system statement.

- Assignment statement

  An assignment statement stores a value to a variable. A variable name is specified in the left-hand side of an equal sign (=), while an expression that yields the assigned value is written in the right-hand side. Variables are used without type declaration, and they can retain any types of values. If a new value is stored in an existing variable, its old value is deleted.

- Procedure call

  A procedure call instructs a call for a procedure.

- System statement

  System statements are used to control a SILC server's behavior. There is only the prefer statement (Section 4.3) at the moment.

You can pass multiple statements to `SILC_EXEC` at once by concatenating them with semicolon (;).

4.1. Assignment statements

There are two types of assignment statements.

- Simple assignment

  This is an assignment in the form of "*`name = expression`*". The value of the expression is stored into the variable of the specified name. A new variable is defined unless already defined; otherwise, the value of the variable is replaced with the new value. When *`expression`* is a variable name as in "`A = B`", the value of `B` is duplicated and assigned to `A`.

- Augmented assignment

  This is an assignment with a binary operation. There are seven augmented assignments as listed below. For example, "*`name += expression`*" is equivalent to "*`name = name + expression`*". The variable name in the left-hand side must be defined in advance.

  `name += expression`

Addition.

name `-=` *expression*

Subtraction.

name `*=` *expression*

Multiplication.

name `/=` *expression*

Division.

name `%=` *expression*

Remainder.

name `*@=` *expression*

Elementwise multiplication.

name `/@=` *expression*

Elementwise division.

The right-hand side of an assignment statement is an expression, whose components are described below. In the following descriptions, x, y, e_1, e_2, ..., and e_N are arbitrary expressions. See Section 4.4 for the data types that are acceptable as operands of unary and binary operators, and Section 5.3 for available built-in functions and procedures.

- Unary operators

  x`~`

  Complex conjugates.

  x`'`

  Conjugate transposes.

  x`'~` or x`~'`

  Transposes.

  `-`x

  Negation.

- Binary operators

  x `+` y

  Addition.

  x `-` y

  Subtraction.

  x `*` y

Multiplication.

`x / y`

Division.

`x % y`

Remainder.

`A \ b`

Solution of systems of linear equations $Ax = b$, where A is an N-by-N matrix, and b is either a vector of length N or an N-by-M matrix.

`x *@ y`

Elementwise multiplication.

`x /@ y`

Elementwise division.

- Function calls

  $f(e_1, e_2, ..., e_N)$

  f is a function name, and e_1, e_2, ..., e_N are arbitrary expressions (i.e., arguments).

- Concatenation

  $\{e_1, e_2, ..., e_N\}$

  The values of the expressions are concatenated vertically. If all e_1, e_2, ..., e_N are scalars, the result of concatenation is a column vector.

  $\{e_1; e_2; ...; e_N\}$

  The values of the expressions are concatenated horizontally. If all e_1, e_2, ..., e_N are scalars, the result of concatenation is a row vector.

  $\{e_1;; e_2;; ...;; e_N\}$

  This always results in a cubic array.

- Range

  $\{e_1:e_2\}$

  This generates a column vector whose elements are integer scalars from e_1 to e_2. The values of the two expressions must be integer scalars.

- Literals

  Numbers are treated as double precision real (`SILC_DOUBLE`) if they have a dot; otherwise, numbers are treated as single precision integer (`SILC_INT`).

- Constants

`e`

Napier's constant.

`i`

The imaginary unit (complex numbers can be obtained by binary operators as in "`3 - 5 * i`".)

`pi`

The ratio of a circle's circumference to its diameter.

`inf`

Infinity.

- Miscellaneous
  - Variable names are expressions.
  - You can use parentheses in compound expressions to specify which part of an expression should be evaluated first.
  - You can also use subscript (Section 4.6) to refer to part of data.

4.2. Procedure calls

A procedure call consists of a procedure name and a list of arguments, as illustrated below.

```
split(A, L, D, U)
```

This procedure divides matrix `A` into the three parts of lower triangle, main diagonal, and upper triangle. These parts are stored in variables `L`, `D`, and `U`, respectively.

Procedures have three types of arguments as follows.

in

The arguments of this type deliver input data to procedures. The value of these arguments are never modified.

out

The arguments of this type are used to receive output data from procedures.

inout

The arguments of this type are used for both delivering input data and receiving output data.

In the case of the procedure split above, the first argument is an in-argument, and the other three are out-arguments.

4.3. System statements

System statements are used to control a SILC server's behavior. There is only one system statement at the moment.

```
prefer moduleName
```

This statement reorders the search path of module functions by bringing the specified module to the beginning of the list of modules. See Section 4.7 for more information on the way of looking up module functions.

4.4. Operands of operators

Although operands of an operator can be arbitrary expressions, valid data types of the operands vary according to the operator. The acceptable combinations of data types for each operator are summarized as follows. The same restrictions on binary operators are applied to augmented assignments (Section 4.1).

Table 6. Unary operators.

Complex conjugates[a]	Scalar, column/row vector, matrix, cubic array
Conjugate transposes[a][b]	Scalar, column/row vector, matrix
Transposes[b]	Scalar, column/row vector, matrix
Negation	Scalar, column/row vector, matrix, cubic array
[a] If the precision of the operand is either integer or real, the result equals to the operand. [b] If the operand is a scalar, the result equals to the operand.	

Table 7. Binary operators.

Operator	Left operand	Right operand	Result
Addition/subtraction	Scalar	Scalar	Scalar
	Column vector	Column vector	Column vector
	Row vector	Row vector	Row vector
	Matrix	Matrix	Matrix
	Cubic array	Cubic array	Cubic array
Multiplication	Scalar	Scalar	Scalar
	Scalar	Column vector	Column vector
	Scalar	Row vector	Row vector
	Scalar	Matrix	Matrix
	Column vector	Scalar	Column vector
	Column vector	Row vector	Matrix
	Row vector	Scalar	Row vector
	Row vector	Column vector	Scalar[a]
	Row vector	Matrix	Row vector
	Matrix	Scalar	Matrix
	Matrix	Column vector	Column vector
	Matrix	Matrix	Matrix
	Cubic array	Scalar	Cubic array
Division	Scalar	Scalar	Scalar
	Row vector	Matrix	Column vector
	Matrix	Matrix	Matrix
Remainder[b]	Scalar	Scalar	Scalar
Solution of systems of linear equations	Scalar	Scalar	Scalar
	Matrix	Column vector	Column vector
	Matrix	Matrix	Matrix
Elementwise multiplication/division	Scalar	Scalar	Scalar

	Column vector	Column vector	Column vector
	Row vector	Row vector	Row vector
	Matrix	Matrix	Matrix
	Cubic array	Cubic array	Cubic array

[a] The result represents the inner product of the two vectors.

[b] The precisions of the two operands must be integer.

4.5. Rules for precision conversion

If the two operands of a binary operator have a different precision, they are converted to the same precision before computation. The following rules are applied to precision conversion.

- The following relation (i.e., a partial order) is defined among the six precisions. If X → Y, then Y is said a precision higher than X. Moreover, if X → Y and Y → Z, then X → Z.

```
SILC_LONG   →   SILC_DOUBLE   →   SILC_DOUBLE_COMPLEX
    ↑               ↑                   ↑
SILC_INT    →   SILC_FLOAT    →   SILC_COMPLEX
```

- If the precision of one operand is higher than that of the other, then the latter operand is converted to the precision of the former.

  For example, if one operand is of `SILC_DOUBLE` and the other is of `SILC_FLOAT`, then the latter is converted to `SILC_DOUBLE` since it is a higher precision than `SILC_FLOAT`.

- If neither operand has a precision higher than that of the other, then both operands are converted to a common higher precision.

  For example, if one operand is of `SILC_DOUBLE` and the other is of `SILC_COMPLEX`, then both operands are converted to a common higher precision, that is `SILC_DOUBLE_COMPLEX`.

4.6. Subscript

You can use subscript with either a variable name on the left-hand side of an assignment statement, or arbitrary expressions (including variable names). A variable name with subscript takes one of the following three forms:

- `A[`*x*`]`

  `A` must be a column/row vector.

- `A[`*x*`,`*y*`]`

  `A` must be a matrix.

- `A[`*x*`,`*y*`,`*z*`]`

  `A` must be a cubic array.

where *x*, *y*, and *z* are expressions whose values are either an integer scalar or an integer column vector. Expressions with other kinds of values result in a runtime error.

When you use a range as a subscript expression, you can omit the initial and/or end value(s) of the range.

For example, if `A` is a vector of length *N*, then `A[:5]` equals to `A[1:5]`, `A[5:]` to `A[5:N]`, and `A[:]` to `A[1:N]`, respectively.

The following table summarizes the functions of subscript, the contexts in which subscript can be used, and some examples of mathematical expressions with subscript.

Table 8. Functions of subscript in various contexts.

Function	Context	Example
Partial reference	The right-hand side of an assignment	`B = A[1:5]` A partial vector (length 5) of `A` is stored to `B`.
	In-arguments of functions and procedures	`Y = F(A[1:5,1:5])` Only a 5-by-5 submatrix of `A` is passed to function `F`. Matrix `A` does not change.
Restricted assignment	The left-hand side of an assignment	`A[1:5,1:5] = B` A 5-by-5 submatrix of `A` is replaced with the elements of matrix `B`.
	Out-arguments of procedures	`P1(x, A[1:5])` Procedure `P1` partially modifies vector `A`.
Partial reference + restricted assignment	Inout-arguments of procedures	`P2(A[1:5,1:5])` Procedure `P2` takes a 5-by-5 submatrix of `A` and modifies its elements.

4.7. Modules

Every operator, function, and procedure in the command language of SILC is carried out through a call for a module function, which is a "wrapper" that actually calls a library function. Related module functions are grouped into a module. There are several standard modules as shown below.

- coremodule

  Contains module functions for all data types except matrices.

- dense

  Contains module functions for dense matrices.

- sparse_crs

  Contains module functions for sparse matrices in the CRS format.

- leq_lis

  Provides access to the Lis iterative solvers library.

- leq_cg

  Implements the CG method for dense matrices and sparse matrices in the CRS format.

- leq_gs

  Implements the Gauss-Seidel method for dense matrices.

- leq_lu

  Implements the LU decomposition method for dense matrices.

- blasmodule

  Contains module functions for matrix-vector product and other operators based on BLAS (Basic Linear Algebra Subprograms).

- fftss

  Contains module functions for the FFTSS fast Fourier Transform library.

There are also a few experimental modules.

- sparse_band

  Implements the banded matrix storage format and the LU decomposition method based on LAPACK (Linear Algebra PACKage).

- sparse_jds

  Implements the the Jagged Diagonal Storage (JDS) format.

- leq_smsamg

  Contains module functions for VINAS Super Matrix Solver AMG version 3.

- leq_mp

  Contains module functions for the mp_crs multiple precision iterative solvers library. This module requires the GNU MP library.

- linpackmodule

  Contains module functions for the LINPACK benchmark.

Every operator, function, and procedure is handled by one of module functions in the modules above. Some operators have multiple module functions that can handle them. For example, the three modules of leq_cg, leq_gs, and leq_lu contain a module function for the backslash operator (i.e., solution of systems of linear equations). Modules are maintained in a SILC server by means of a serial linked list. When the SILC server needs to handle an operator, the server picks a module function by searching one module after another from the beginning of the module list and selecting the first module function found in the list. Module functions for functions and procedures are also looked up in the same way.

You can change the order of modules in the module list using the prefer statement (Section 4.3) in a call for the `SILC_EXEC` routine. The prefer statement moves the specified module to the beginning of the list. By changing the order of modules, you can easily specify the relations between operators (functions, or procedures) and module functions.

5. API reference

5.1. Client routines for C

- Header file: `client.h`

  You use this header file when creating user programs for SILC.

- Structure: `silc_envelope_t`

  This structure is used as an argument of the `SILC_PUT` and `SILC_GET` routines. The structure has the

following members, and only some of them are used according to the type of data to be transferred.

- Members common to all data types:

  `int type;`

  Data type (Section 3.3). The value of this member must be one of the following constants.

`SILC_SCALAR_TYPE`	(scalar)
`SILC_ROW_VECTOR_TYPE`	(row vector)
`SILC_COLUMN_VECTOR_TYPE`	(column vector)
`SILC_MATRIX_TYPE`	(matrix)
`SILC_CUBIC_ARRAY_TYPE`	(cubic array)

  `int precision;`

  Precision (Section 3.3). The value of this member must be one of the following constants.

`SILC_INT`	(single precision integer)
`SILC_LONG`	(double precision integer)
`SILC_FLOAT`	(single precision real)
`SILC_DOUBLE`	(double precision real)
`SILC_COMPLEX`	(single precision complex)
`SILC_DOUBLE_COMPLEX`	(double precision complex)

  `const char *format;`

  Matrix storage format (Section 3.4). This member is used only when the data type is `SILC_MATRIX_TYPE`. The value must be one of the following constants.

`SILC_FORMAT_DENSE`	(dense matrix)
`SILC_FORMAT_CRS`	(sparse matrix in the CRS format)

- For `SILC_SCALAR_TYPE`:

  `void *v;`

  A pointer to the scalar value. (*)

- For `SILC_ROW_VECTOR_TYPE` and `SILC_COLUMN_VECTOR_TYPE`:

  `size_t length;`

  The length of the vector.

  `void *v;`

  A pointer to an array that stores the elements of the vector. (*)

- For `SILC_MATRIX_TYPE` (`SILC_FORMAT_DENSE`):

  `size_t m, n;`

  The dimensions of the matrix.

`void *v;`

A pointer to a Fortran-style 2-dimensional array that stores the elements of the matrix column by column. (*)

- For `SILC_MATRIX_TYPE` (`SILC_FORMAT_CRS`):

`size_t m, n;`

The dimensions of the matrix.

`size_t nnz;`

The number of non-zero elements.

`void *v;`

A pointer to an array of non-zero elements. (*)

`int *row;`

A pointer to an array of row pointers. (*)

`int *index;`

A pointer to an array of column indexes. (*)

The base (or origin) of elements in `row` and `index` must be zero.

- For `SILC_CUBIC_ARRAY_TYPE`:

`size_t l, m, n;`

The dimensions of the cubic array.

`void *v;`

A pointer to a Fortran-style 3-dimensional array that stores the elements of the cubic array. (*)

When calling `SILC_GET`, you need to initialize only the members with the asterisk mark (*). When calling `SILC_PUT`, you have to set a valid value to all members.

The following client routines return 0 if no error occurs; otherwise, they return −1.

- `int SILC_INIT(void);`

  This routine establishes a connection to a SILC server. Call this routine before you start using the SILC server (for example, at the beginning of a user program).

- `int SILC_FINALIZE(void);`

  This routine closes the connection to the SILC server. Call this routine when you stop using the SILC server (for example, at the end of a user program).

- `int SILC_EXEC(const char *expr);`

  This routine sends a request for computation to the SILC server, where expr is a string of a mathematical expression written in the command language (Section 4).

- `int SILC_PUT(const char *name, silc_envelope_t *envelope);`

  This routine associates a name with data and deposits the named data to the SILC server. The fist argument is a string that represents the data name, and the second is a pointer to the `silc_envelope_t` structure that holds the information on the data to be sent.

- `int SILC_GET(silc_envelope_t *envelope, const char *name);`

  This routine fetches data from the SILC server by specifying the name of the data to be received. The first argument is a pointer to the `silc_envelope_t` structure, and the second argument is a string that represents the data name.

  You can allocate buffers for receiving data in either of the following two ways.

  a. Allocating all buffers in advance.

     If the member `v` of the `silc_envelope_t` structure is not `NULL`, then received data are stored in the buffer pointed by `v` (as well as the buffers pointed by `row` and `index` in case of sparse matrices in the CRS format).

  b. Allowing the `SILC_GET` routine to allocate buffers automatically.

     If the member `v` of the `silc_envelope_t` structure is `NULL`, then the `SILC_GET` routine automatically allocates buffers of an appropriate size. You have to free the buffer pointed by `v` (as well as the buffers pointed by `row` and `index` in case of sparse matrices in the CRS format) if the received data are no longer needed.

5.2. Client routines for Fortran

- Header file: `client.h`

  You use this header file when creating user programs for SILC.

The following arguments are common to the client routines for Fortran.

- `INTEGER*4 precision`

  Precision (Section 3.3). The value of this member must be one of the following constants.

`SILC_INT`	(single precision integer)
`SILC_LONG`	(double precision integer)
`SILC_FLOAT`	(single precision real)
`SILC_DOUBLE`	(double precision real)
`SILC_COMPLEX`	(single precision complex)
`SILC_DOUBLE_COMPLEX`	(double precision complex)

- `INTEGER*4 status`

  An exit status of a client routine. The value is 0 if no error occurs, and −1 otherwise.

In the following descriptions, *`type`* can be one of the following data types.

`INTEGER*4`	(for `SILC_INT`)
`INTEGER*8`	(for `SILC_LONG`)
`REAL*4`	(for `SILC_FLOAT`)
`REAL*8`	(for `SILC_DOUBLE`)
`COMPLEX*8`	(for `SILC_COMPLEX`)

`COMPLEX*16` (for `SILC_DOUBLE_COMPLEX`)

- `SILC_INIT(status)`

  This routine establishes a connection to a SILC server. Call this routine before you start using the SILC server (for example, at the beginning of a user program).

  Output:

  `INTEGER*4 status`

  An exit status.

- `SILC_FINALIZE(status)`

  This routine closes the connection to the SILC server. Call this routine when you stop using the SILC server (for example, at the end of a user program).

  Output:

  `INTEGER*4 status`

  An exit status.

- `SILC_EXEC(expr, status)`

  This routine sends a request for computation to the SILC server.

  Input:

  `CHARACTER*`*`size`* `expr`

  A mathematical expression written in the command language (Section 4).

  Output:

  `INTEGER*4 status`

  An exit status.

- `SILC_PUT_SCALAR(name, value, precision, status)`

  This routine associates a name with a scalar value and deposits it to the SILC server.

  Input:

  `CHARACTER*`*`size`* `name`

  A name.

  `type` `value`

  The scalar value.

  `INTEGER*4 precision`

  The precision of the scalar value.

  Output:

`INTEGER*4 status`

An exit status.

- `SILC_PUT_ROW_VECTOR(name, value, length, precision, status)`

This routine associates a name with a row vector and deposits it to the SILC server.

Input:

`CHARACTER*`*`size`* `name`

A name.

`type` `value(length)`

The elements of the vector.

`INTEGER*4 length`

The length of the vector.

`INTEGER*4 precision`

The precision of the elements.

Output:

`INTEGER*4 status`

An exit status.

- `SILC_PUT_COLUMN_VECTOR(name, value, length, precision, status)`

This routine associates a name with a column vector and deposits it to the SILC server.

Input:

`CHARACTER*`*`size`* `name`

A name.

`type` `value(length)`

The elements of the vector.

`INTEGER*4 length`

The length of the vector.

`INTEGER*4 precision`

The precision of the elements.

Output:

`INTEGER*4 status`

An exit status.

- `SILC_PUT_MATRIX(name, value, m, n, precision, status)`

  This routine associates a name with an M-by-N dense matrix and deposits it to the SILC server.

  Input:

  `CHARACTER*`*`size`* `name`

  A name.

  `type` `value(m, n)`

  The elements of the matrix.

  `INTEGER*4 m, n`

  The dimensions of the matrix.

  `INTEGER*4 precision`

  The precision of the elements.

  Output:

  `INTEGER*4 status`

  An exit status.

- `SILC_PUT_MATRIX_CRS(name, value, m, n, nnz, row, index, precision, status)`

  This routine associates a name with an M-by-N sparse matrix in the CRS format and deposits it to the SILC server. Unlike the C version of the CRS format, the base (or origin) of elements of the arrays *`row`* and *`index`* must be one.

  Input:

  `CHARACTER*`*`size`* `name`

  A name.

  `type` `value(nnz)`

  The non-zero elements of the matrix.

  `INTEGER*4 m, n`

  The dimensions of the matrix.

  `INTEGER*4 nnz`

  The number of the non-zero elements.

  `INTEGER*4 row(m+1)`

  An array of row pointers.

  `INTEGER*4 index(nnz)`

  An array of column indexes.

`INTEGER*4 precision`

The precision of the non-zero elements.

Output:

`INTEGER*4 status`

An exit status.

- `SILC_PUT_CUBIC_ARRAY(name, value, l, m, n, precision, status)`

This routine associates a name with an L-by-M-by-N cubic array and deposits it to the SILC server.

Input:

`CHARACTER*`*`size`* `name`

A name.

`type` `value(l, m, n)`

The elements of the cubic array.

`INTEGER*4 l, m, n`

The dimensions of the cubic array.

`INTEGER*4 precision`

The precision of the elements.

Output:

`INTEGER*4 status`

An exit status.

- `SILC_GET_SCALAR(name, value, status)`

This routine receives a scalar value from the SILC server by specifying the name of the scalar value to be fetched.

Input:

`CHARACTER*`*`size`* `name`

A name.

Output:

`type` `value`

The received scalar value.

`INTEGER*4 status`

An exit status.

- `SILC_GET_ROW_VECTOR(name, value, status)`

This routine receives a row vector from the SILC server by specifying the name of the vector to be fetched. You need to know the dimension of the vector (i.e., *length*) in advance in order to prepare the output array `value`.

Input:

`CHARACTER*`*size* `name`

A name.

Output:

type `value(`*length*`)`

The elements of the received vector.

`INTEGER*4 status`

An exit status.

- `SILC_GET_COLUMN_VECTOR(name, value, status)`

This routine receives a column vector from the SILC server by specifying the name of the vector to be fetched. You need to know the dimension of the vector (i.e., *length*) in advance in order to prepare the output array `value`.

Input:

`CHARACTER*`*size* `name`

A name.

Output:

type `value(`*length*`)`

The elements of the received vector.

`INTEGER*4 status`

An exit status.

- `SILC_GET_MATRIX(name, value, status)`

This routine receives a dense matrix from the SILC server by specifying the name of the matrix to be fetched. You need to know the dimensions of the matrix *m* and *n* in advance in order to prepare the output array `value`.

Input:

`CHARACTER*`*size* `name`

A name.

Output:

type `value(m, n)`

The elements of the received matrix.

`INTEGER*4 status`

An exit status.

- `SILC_GET_MATRIX_CRS(name, value, row, index, status)`

  This routine receives a sparse matrix in the CRS format from the SILC server by specifying the name of the matrix to be fetched. You need to know the dimensions of the matrix *m* and *n* as well as the number of non-zero elements *nnz* in advance in order to prepare the output arrays `value`, `row`, and `index`. The base (or origin) of elements of `row` and `index` is one.

  Input:

  `CHARACTER*`*size* `name`

  A name.

  Output:

  type `value(`*nnz*`)`

  The non-zero elements of the received matrix.

  `INTEGER*4 row(`*m*`+1)`

  An array of row pointers.

  `INTEGER*4 index(`*nnz*`)`

  An array of column indexes.

  `INTEGER*4 status`

  An exit status.

- `SILC_GET_CUBIC_ARRAY(name, value, status)`

  This routine receives a cubic array from the SILC server by specifying the name of the cubic array to be fetched. You need to know the sizes of the cubic array (i.e., *l*, *m*, and *n*) in order to prepare the output array `value`.

  Input:

  `CHARACTER*`*size* `name`

  A name.

  Output:

  type `value(`*l*`, `*m*`, `*n*`)`

  The elements of the received cubic array.

  `INTEGER*4 status`

  An exit status.

5.3. Built-in functions and procedures

Built-in functions and procedures are defined in modules. If two or more functions or procedures have the

same name, the function or procedure that is found first is called. By using the prefer statement (Section 4.3) to change the order of modules, you can use specific functions and procedures defined in particular modules.

5.3.1. coremodule

- *scalar* `dot(`*rowVector*`, `*columnVector*`)` [function]

  This function returns the inner product of the given vectors.

- *scalar* `sqrt(`*scalar*`)` [function]

  vector `sqrt(`*vector*`)` [function]

  This function returns the square root of the given scalar value or elements of the given vector.

- *scalar* `norm2(`*vector*`)` [function]

  This function returns the 2-norm of the given vector.

- *scalar* `length(`*vector*`)` [function]

  This function returns the length (i.e., the number of elements) of the given vector.

- *scalar* `time()` [function]

  This function returns the time in seconds in double precision real.

5.3.2. dense module

- *columnVector* `diagvec(`*matrix*`)` [function]

  This function returns a column vector that consists of elements in the main diagonal of the given matrix.

- *matrix* `zeros(`*scalar*`)` [function]

- *matrix* `zeros(`*scalar*`, `*scalar*`)` [function]

  These functions return a dense matrix whose elements are 0 in double precision real. The zeros function taking one argument returns a square matrix. The first and second arguments of the other function specify the number of rows and the number of columns, respectively.

- *matrix* `ones(`*scalar*`)` [function]

- *matrix* `ones(`*scalar*`, `*scalar*`)` [function]

  These functions return a dense matrix whose elements are 1 in double precision real. The ones function taking one argument returns a square matrix. The first and second arguments of the other function specify the number of rows and the number of columns, respectively.

- *matrix* `rand(`*scalar*`)` [function]

- *matrix* `rand(`*scalar*`, `*scalar*`)` [function]

  These functions return a dense matrix whose elements are random numbers in double precision real. The rand function taking one argument returns a square matrix. The first and second arguments of the other function specify the number of rows and the number of columns, respectively. The sequence of random numbers can be initialized with the srand procedure described below.

- *columnVector* `size(`*matrix*`)` [function]

This function returns a column vector of length 2. The elements of the vector represent the dimensions (i.e., the number of rows and the number of columns) of the given matrix.

- *scalar* size(*matrix*, *scalar*) [function]

  This function returns a dimension of the given matrix. The number of rows is returned if the second argument is 1, while the number of columns is returned if the second argument is 2.

- *matrix* full(*scalar*, *scalar*) [function]

  This function returns an integer matrix of double precision. The first and second arguments specify the number of rows and the number of columns, respectively. Elements of the matrix have non-zero values (but they are not random numbers).

- split(*matrix* IN, *matrix* OUT, *matrix* OUT, *matrix* OUT) [procedure]

  This procedure divides the matrix of the first argument into lower triangle, main diagonal, and upper triangle. These parts are stored in the three variables specified by the second, third, and forth arguments, respectively.

- srand(*scalar* IN) [procedure]

  This procedure (re)initializes the Mersenne Twister random number generator, specifying a seed for a new sequence of random numbers.

5.3.3. sparse_crs module

- *matrix* sparse(*vector r*, *vector c*, *vector v*, *scalar m*, *scalar n*) [function]
- *matrix* sparse(*vector r*, *vector c*, *scalar v*, *scalar m*, *scalar n*) [function]

  These functions return a sparse matrix in the CRS format. Vector *r* in the first argument is a list of row indices and vector *c* in the second argument is a list of column indices. If the third argument *v* is a vector, the *k*th elements of *r* and *c* represent the row and column indices of the matrix to which the *k*th element of *v* is stored. That is, *v* is a list of non-zero elements. In this case, the length of *r*, *c* and *v* must be the same. If *v* is scalar, all non-zero elements in the matrix will be the same value. In this case, the length of *r* and *c* must be the same. The fourth argument *m* is the number of rows and th fifth argument *m* is the number of columns. The precision of the matrix will be the same as the precision of *v*.

- *matrix* zeros(*scalar*) [function]
- *matrix* zeros(*scalar*, *scalar*) [function]

  These functions return a sparse matrix (in the CRS format) whose elements are 0 in double precision real. The zeros function taking one argument returns a square matrix. The first and second arguments of the other function specify the number of rows and the number of columns, respectively.

- *matrix* eye(*scalar*) [function]

  This function returns the identity matrix (in the CRS format) of the given dimension.

- *matrix* diag(*vector*) [function]
- *matrix* diag(*vector*, *scalar*) [function]

  These functions return a sparse diagonal matrix in the CRS format. Let *n* be the length of the given vector. If only one argument is specified, the diag function generates *n*-by-*n* diagonal matrix whose diagonal elements are given by the vector. If the second argument is specified, it represents an offset value (referred to as *k*). If $k = 0$, then diag(*v*, *k*) is equivalent to diag(*v*). If $k > 0$, a square matrix

with a diagonal starting from (1, *k*+1) is generated. If *k* < 0, the subdiagonal starts from (1-*k*, 1). The dimension of the generated matrix is *n*+*k*.

- `vector diagvec(matrix)` [function]

  This function returns a vector that consists of elements in the main diagonal of the given matrix (in the CRS format).

- `matrix fliplr(matrix)` [function]

  This function returns the given matrix (in the CRS format) with columns in the reversed order.

- `matrix flipud(matrix)` [function]

  This function returns the given matrix (in the CRS format) with rows in the reversed order.

5.3.4. sparse_band module

- `matrix CRS(matrix)` [function]

  This function converts the given matrix (in the banded storage format) to the CRS format and returns a new matrix.

- `matrix band(matrix)` [function]

  This function converts the given matrix (in the CRS format) to the banded storage format and returns a new matrix.

- `scalar rcond(matrix)` [function]

  This function returns the reciprocal of the estimated condition number of the given matrix (in the banded storage format).

A. Revision history

- October 31, 2007 (SILC v1.3)
  - Descriptions on how to compile a SILC server and user programs were added.
  - Some built-in functions were added to the API reference.
- November 12, 2006 (SILC v1.2)
  - The source file of this document was converted into the DocBook XML format.
  - Descriptions of some built-in functions and procedures were added.
- November 25, 2005 (SILC v1.1)
  - The first English edition.

```
$Id: users_en.xml,v 1.13 2007/10/31 06:31:35 kajiyama Exp $
```