SILC: A Flexible and Environment-Independent Interface for Matrix Computation Libraries

Tamito Kajiyama[1,2], Akira Nukada[1,2], Hidehiko Hasegawa[1,3], Reiji Suda[1,2], and Akira Nishida[1,2]

[1] CREST, Japan Science and Technology Agency, Saitama 332–0012, Japan
[2] The University of Tokyo, Tokyo 113–8656, Japan
{kajiyama, nukada, reiji, nishida}@is.s.u-tokyo.ac.jp
[3] University of Tsukuba, Tsukuba 305–8550, Japan
hasegawa@slis.tsukuba.ac.jp

Abstract. We propose a new framework, named Simple Interface for Library Collections (SILC), that gives users access to matrix computation libraries in a flexible and environment-independent manner. SILC achieves source-level independence between user programs and libraries by (1) separating a function call into data transfer and a request for computation, (2) requesting the computation by means of mathematical expressions in the form of text, and (3) using a separate memory space to carry out library functions independently of the user programs. Using SILC, users can easily access various libraries without any modification of the user programs. This paper describes the design and implementation of SILC based on a client-server architecture, and presents some experimental results on the performance of the implemented system in different computing environments.

1 Introduction

Solutions of systems of linear equations and other matrix computations take a major proportion of execution time and memory resources in many large-scale scientific applications. As a result, a large number of matrix computation libraries have been developed [1, 2, 3] to facilitate the rapid development of user programs. Each library offers a different set of solvers and matrix storage formats and has its own application programming interface that is incompatible with other libraries.

The traditional way to use matrix computation libraries, i.e., through function calls, makes user programs dependent on the libraries. Users of a library have to prepare input matrices in a specific storage format and to make a function call using a library-specific function name together with a number of arguments in a prescribed order. Although such function calls are plain and intuitive, they result in source-level dependency on the library, making it difficult to replace one library with another.

There are various computing environments, such as personal computers, symmetric multiprocessor (SMP) systems, high-end supercomputers, and clusters, each of which has its own highly optimized libraries. Therefore, users must modify their user programs significantly in order to port them from one computing environment to another. Moreover, there are various solvers and matrix storage formats, the best combination of which varies according to the computing environment in use and the problem to be solved. However, because of the source-level dependency resulting from the use of a solver and matrix storage format in the form of conventional function calls, users who want to find the most efficient solver and matrix storage format must make considerable modifications in the user programs to change solvers and matrix storage formats.

To address those issues that arise from the source-level dependency on specific libraries, we propose a new framework that allows users to easily utilize matrix computation libraries in a flexible and computing environment-independent manner. The framework, named Simple Interface for Library Collections (SILC), is based on the following three design decisions.

– To separate a function call into data transfer (to and from a separate memory space) and a request for computation.
– To request the computation by means of mathematical expressions in the form of text.
– To use the separate memory space to carry out library functions independently of user programs.

In our framework, a function call occurs through three steps: sending input data (i.e., arguments), requesting computation by means of mathematical expressions, and receiving the results of the computation. The operators that comprise the expressions are translated into a series of function calls and are carried out in a separate memory space. The results of the computation are sent back only when they are required by user programs.

The main benefits of employing SILC are as follows.

– User programs will be free of source-level dependency on specific libraries, so that users won't need to modify their programs when changing libraries according to the computing environment and the problem to be solved.
– Users need to prepare only the smallest amount of data. Temporary memory space used for carrying out library functions is automatically allocated before the library functions are called.
– A variety of computing environments and programming languages can be used, since computation is requested by means of mathematical expressions in the form of text.

2 Design and Implementation

We have been developing a SILC system for shared-memory parallel computing environments. Figure 1 shows an architectural overview of the SILC system. It is based on a client-server architecture. The SILC server and user programs can run either on the same machine or on different machines on a network.

A user program connects to the SILC server and utilizes the features of matrix computation libraries by sending three types of requests described below.

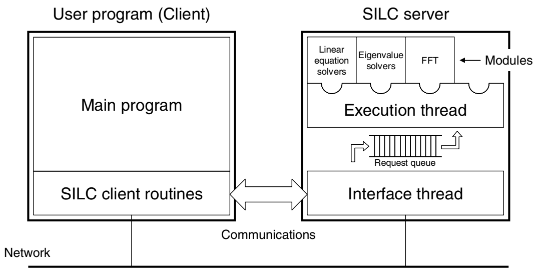

Fig. 1. Architectural overview of SILC

PUT deposits data such as matrices and vectors, together with names for later reference. The data are converted and transferred into the server's memory space, which is independent of the user program. The deposited data remain there until deleted explicitly.

EXEC requests computation by means of mathematical expressions in the form of text. The names defined by preceding PUT requests are used in the expressions to refer to the deposited data. The computation is carried out on the server asynchronously. The results of the computation and names defined by the expressions are retained in the server's memory space.

GET fetches data from the server. Names are used to specify the data to be fetched, and those data are sent back into the memory space of the user program. The data are kept undeleted on the server.

In case a user program is written in C, the following three client routines are used to issue the PUT, EXEC, and GET requests, respectively:

- `SILC_PUT(`⟨*name*⟩`,` ⟨*data*⟩`)`
- `SILC_EXEC(`⟨*expr*⟩`)`
- `SILC_GET(`⟨*data*⟩`,` ⟨*name*⟩`)`

where ⟨*name*⟩ is a data name and ⟨*expr*⟩ is a mathematical expression (whose syntax is described later), each specified by a string. ⟨*data*⟩ is a pointer to the `silc_envelope_t` structure that is used for data communications between the user program and the SILC server.

The requests from the user program are received by the interface thread in the SILC server. PUT and GET requests are handled by the interface thread, while EXEC requests are stored in the request queue and processed by the execution thread one after another. The user program and the interface thread run synchronously, while the execution thread runs asynchronously.

Figure 2 (a) shows a user program that calls a library function `ssi_cg` [3] to solve a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ with the CG method [4]. The input data of the library function are matrix A and vector $\boldsymbol{b}$ as well as some solver-specific parameters, while the output is the solution $\boldsymbol{x}$. These data are represented by library-specific data structures and are passed to the library function through its arguments in a prescribed order. On the other hand, in the

```
SSI_MATRIX A;
SSI_SCALAR *b, *x, work[N*6], params[2];
int options[6], status;

/* Create matrix A and vectors b and x */
status = ssi_cg(b, x, work,
               params, options, &A, NULL);
```

(a)

```
silc_envelope_t A, b, x;

/* Create matrix A and vectors b and x */
SILC_PUT("A", &A);
SILC_PUT("b", &b);
SILC_EXEC("x = A \\ b"); /* Call a solver */
SILC_GET(&x, "x");
```

(b)

Fig. 2. Comparison between the two ways of using a library function. (a) is a user program that makes use of a library function in the traditional manner. (b) is another user program written in the framework of SILC.

framework of SILC, the same computation can be achieved as shown in Fig. 2 (b). In this framework, the input data are deposited by two separate calls of the client routine `SILC_PUT`. After the input data are deposited, computation is requested by the `SILC_EXEC` routine. This routine's argument is a mathematical expression in the form of text that instructs the solution of the system of linear equations using an appropriate library function (e.g., `ssi_cg`). Finally, the output data are fetched by `SILC_GET`. The source code shown in Fig. 2 (b) does not contain any code that is specific to the actual library to be used for the computation.

The SILC system is equipped with a simple command language to represent a request for computation in the form of mathematical expressions. The unit of computation that is carried out at once is a statement, which is either an assignment or a procedure call. The left-hand side of an assignment statement is a variable name, which can be used without declaring a data type. The right-hand side of the assignment statement is an expression, which consists of variable names, operators, and function calls. Some of the operators are binary arithmetic operators (`+`, `-`, `*`, `/`, `%`), solutions of systems of linear equations (e.g., `A \ b` obtains the solution $\boldsymbol{x}$ as in $A\boldsymbol{x} = \boldsymbol{b}$), complex conjugates (`A~`), and conjugate transposes (`A'`). There is no control statement in the command language; loops and conditional branching are achieved by the programming languages in which user programs are written.

Every operator, function, and procedure[1] that appears in a mathematical expression is translated into a call of a library function with the help of a wrapper that actually calls the library function. The wrapper provides a unified interface to the library function so that the SILC server can invoke all library functions in the same manner. Related wrappers are grouped into an arithmetic module, and all modules are dynamically loaded into the SILC server at startup. When loading arithmetic modules, the server constructs a mapping table that relates operators, functions, and procedures to certain wrappers. The server then handles each of the operators, functions, and procedures used in a given mathematical expression by invoking a corresponding wrapper, which results in a call

[1] Functions return values, while procedures do not. Moreover, procedures can change the values of arguments, whereas functions cannot. The distinction between functions and procedures is introduced to eliminate ambiguities from mathematical expressions.

Table 1. Four server configurations used for preliminary experiments

Environment	Specifications	Thread(s)
(a) A notebook PC	CPU: Intel Pentium M 733 1.1 GHz, Memory: 768 MB, OS: Fedora Core 3	–
(b) SGI Altix 3700	CPU: Intel Itanium 2 1.3 GHz × 32, Memory: 32 GB, OS: Red Hat Linux Advanced Server 2.1	1
(c) IBM eServer OpenPower 710	CPU: IBM Power5 1.65 GHz × 2 (4 logical CPUs), Memory: 1 GB, OS: SuSE Linux Enterprise Server 9	4
(d) SGI Altix 3700	Same as (b)	16

of a particular library function. Matrix storage formats are implemented in a similar way, and their wrappers are grouped into a storage format module.

Support for a new matrix storage format can be incorporated into the SILC system by providing a storage format module coupled with an arithmetic module. Both arithmetic and storage format modules are parallelized with OpenMP in shared-memory parallel computing environments.

To assess the performance of SILC servers in different computing environments, we conducted preliminary experiments using the four configurations of SILC servers summarized in Table 1. The three computing environments of the notebook PC, Altix 3700, and OpenPower 710 were interconnected via a 100 Base–TX network. We used the user program shown in Fig. 2 (b), which solves a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ using the CG method, where A is an $N \times N$ tridiagonal matrix of double precision in the Compressed Row Storage (CRS) format [4]. An arithmetic module that includes a wrapper for the library function `ssi_cg` was used to solve the system of linear equations. The user program was carried out on the notebook PC for all cases. Since the server to which the user program connects is specified in a configuration file, the same user program was used without any modification during the experiments. Figure 3 shows the results of the experiments, with dimension N on the horizontal axis and execution time in seconds (including communication time for establishing connection and transferring data) on the vertical axis. The experimental results proved that

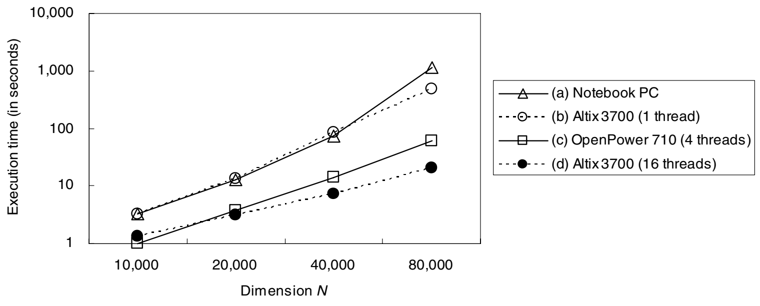

Fig. 3. Experimental results with the four server configurations shown in Table 1

(1) the computing environment that achieves the best performance varies according to dimension N, and (2) the execution times in the cases of (c) and (d) are much shorter than those in the case of (a) even at the cost of depositing matrix A and vector $\boldsymbol{b}$ and fetching the solution $\boldsymbol{x}$ through the relatively slow network. Since communication time is roughly on the order of $O(N)$ while computation time is on the order of $O(N^2)$, the observation (2) agrees with the expectation that the communication time will take a smaller fraction of the whole execution time as dimension N increases.

3 Usability and Benefits

3.1 User Programs

User programs in the framework of SILC are independent of the computing environment in which a SILC server runs. This allows users to develop their programs without being concerned about the details of various computing environments. Users can port their programs from one computing environment to another without any modification to the programs.

SILC separates data transfer from a request for computation, permitting PUT, EXEC, and GET requests to be sent from different user programs separately. Moreover, a SILC server accepts connections from multiple user programs, allowing those programs to use the server as an in-memory database through which they exchange data. Therefore, in the framework of SILC, separate user programs can be easily combined in such a manner that a mesh generation program issues PUT requests, a solution program sends EXEC requests, and a visualization program uses GET requests.

Since data transfer and requests for computation in the form of text are relatively lightweight tasks, less-powerful computing environments, such as personal computers and mobile environments, will suffice to run user programs for SILC. This characteristic allows combinations in which, for example, a personal computer is used to control computations on high-performance parallel computers.

3.2 Libraries

An exchange of libraries is required mainly in either of two situations: (1) users wish to change computing environments or (2) users wish to use different solvers and matrix storage formats provided in other libraries. There are a variety of optimized matrix computation libraries that are only available in a particular computing environment. Moreover, the most efficient solver and matrix storage format depend strongly on the computing environment to be used as well as on the problem to be solved. Aside from the considerable costs of modifying user programs in order to switch libraries, it is burdensome for users to maintain multiple versions of the same program, each of which is written for a specific computing environment and a problem based on a specific combination of a solver and matrix storage format. In the framework of SILC, on the other hand, users can easily change libraries either by using different SILC servers running in other computing environments or by modifying the mapping table in a SILC server so that different library functions are used.

In addition, a SILC server has a separate memory space, which enables the server to convert deposited matrices into different storage formats independently of user programs. For example, it can be easily achieved that a user program uses the CRS format to deposit and receive matrices, while the server uses the Jogged Diagonal Storage (JDS) format [4] to carry out computations on them. Although conversion of storage formats takes some time and space according to the sizes of the matrices, there are cases where better performance is achieved if the matrices are converted into an appropriate storage format before the computations on them. Moreover, the conversion of storage formats allows users to choose solvers and storage formats from among a wide range of matrix computation libraries.

3.3 Programming Languages

In SILC, computation is requested by means of mathematical expressions in the form of text. In addition, the client routines of SILC to be linked with user programs are small and easy to implement. Therefore, various programming languages can be used to develop user programs. At the moment, three sets of client routines for C, Fortran, and Python are available. The main requirements for a programming language to implement the client routines are the capabilities of numerical computation, text processing, and socket-based interprocess communications; the major programming languages meet these requirements.

SILC permits various combinations of matrix computation libraries and programming languages. For example, user programs written in Python can easily make use of libraries written in C or Fortran in the same way.

4 Related Work

To improve the usability of matrix computation libraries, various approaches have been proposed based on Remote Procedure Call (RPC) [5, 6] and code generation techniques [7, 8, 9]. NetSolve [5] and Ninf–G [6] are middleware for realizing RPC in Grids. In these systems, RPCs are requested in a manner similar to traditional function calls. In contrast, SILC employs simple mathematical expressions to request matrix computations, allowing users to ignore complicated matters between user programs and matrix computation libraries. CMC [8] is a compiler that translates a user-defined function in MATLAB [10] into a subroutine in Fortran 90. CMC provides support for several sparse matrix storage formats and carries out a variety of source-level optimizations. Both CMC and SILC pursue the same goal of enhancing the utility of matrix computations. Whereas CMC focuses on the generation of optimized Fortran subroutines from MATLAB functions, SILC focuses on the use of various matrix computation libraries in a language-independent manner.

5 Concluding Remarks

The traditional manner of using matrix computation libraries through function calls usually results in source-level dependency on the libraries, making it impossible to switch libraries without modifying user programs. To address this issue, we have proposed a new framework for using matrix computation libraries in a flexible and environment-independent manner. In this paper, we presented the design and implementation of the proposed framework for shared-memory parallel computing environments based on a client-server architecture. We also reported the results of preliminary experiments assessing the performance of the implemented system, and discussed the usability and benefits of our proposal.

We plan to provide modules for major matrix computation libraries, allowing users to easily switch libraries and compare the performance of user programs with respect to different solvers and matrix storage formats. Implementation of a scripting language for SILC, run-time optimization of mathematical expressions, and MPI-based parallelization of the SILC system are also in our future plans.

Acknowledgments

This research was supported by a grant from the Core Research for Evolutional Science and Technology (CREST) of the Japan Science and Technology Agency [11].

References

1. Dongarra, J.: Freely available software for linear algebra on the Web (2004) http://www.netlib.org/utk/people/JackDongarra/la-sw.html.
2. Wu, K., Milne, B.: A survey of packages for large linear systems. Technical Report LBNL–45446, Lawrence Berkeley National Laboratory (2000)
3. Kotakemori, H., Hasegawa, H., Kajiyama, T., Nukada, A., Suda, R., Nishida, A.: Performance evaluation of parallel sparse matrix–vector products on SGI Altix3700. In: First International Workshop on OpenMP (IWOMP 2005). (to appear)
4. Barrett, R., *et al.*: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM (1994)
5. NetSolve: http://icl.cs.utk.edu/netsolve/ (2005)
6. Ninf Project: http://ninf.apgrid.org/ (2005)
7. De Rose, L., Padua, D.: Techniques for the translation of MATLAB programs into Fortran 90. ACM TOPLAS **21** (1999) 286–323
8. Kawabata, H., Suzuki, M., Kitamura, T.: A MATLAB-based code generator for sparse matrix computations. In: APLAS 2004, LNCS 3302. (2004) 280–295
9. Kennedy, K., *et al.*: Telescoping languages: A system for automatic generation of domain languages. Proceedings of the IEEE **93** (2005) 387–408
10. The MathWorks, Inc.: http://www.mathworks.com/ (2005)
11. Nishida, A.: SSI: Overview of simulation software infrastructure for large scale scientific applications. In: IPSJ SIG Notes, 2004–HPC–098. (2004) 25–30
12. Luszczek, P., Dongarra, J.: Design of interactive environment for numerically intensive parallel linear algebra calculations. In: ICCS 2004, LNCS 3039. (2004) 270–277