Cloth Simulation in the SILC Matrix Computation Framework: A Case Study

Tamito KAJIYAMA[1,2], Akira NUKADA[1,2], Reiji SUDA[2,1], Hidehiko HASEGAWA[3], and Akira NISHIDA[2,1]

[1] CREST, Japan Science and Technology Agency, Saitama 332–0012, Japan
[2] The University of Tokyo, Tokyo 113–8656, Japan
[3] University of Tsukuba, Ibaraki 305–8550, Japan

Abstract. This paper presents a case study of numerical simulations in an easy-to-use matrix computation framework named Simple Interface for Library Collections (SILC), which allows users to use various matrix computation libraries in an environment- and language-independent manner. As a practical example of numerical simulations in SILC, we selected cloth simulation based on a mass-spring model and the implicit backward Euler method. We constructed two SILC-based versions of an existing cloth simulation code according to two proposed application styles of SILC. Experimental results showed that both versions achieved some performance gains, thereby demonstrating the feasibility of numerical simulations in SILC and the usability of the proposed application styles.

1 Introduction

Matrix computations such as solutions of linear systems and eigenvalue analyses are key components of numerical simulations being conducted in various scientific and industrial fields; thus, an increasing number of matrix computation libraries have been developed to facilitate the development of numerical simulation codes. However, the application programming interfaces (APIs) of the libraries are not generally uniform, which makes it quite costly to employ the libraries to develop and maintain simulation codes. It is burdensome for users to have to learn a number of different library-specific APIs in order to write simulation codes using these libraries. In addition, there are numerous reasons why users must switch from one library to another (e.g., in order to switch computing environments, to try out alternative libraries for better performance, and so on). In such cases, users are required to make considerable modifications to the simulation codes, since the codes usually depend on the particular libraries in use.

To relieve the burden of writing simulation codes based directly on library-specific APIs, the authors have been proposing an easy-to-use matrix computation framework named Simple Interface for Library Collections (SILC) [1]. In short, SILC is a piece of middleware that gives access to various matrix computation libraries in an environment- and language-independent manner. With the aim of setting some guidelines for SILC users, we have also been proposing two applications styles for writing simulation codes within the SILC framework [2].

The purpose of the present paper is to verify the effectiveness of the proposed framework and its application styles through a case study in numerical simulation. As a practical example of numerical simulations in SILC, we have selected cloth simulation that is an important technology widely used in many academic and industrial fields including computer graphics and the fashion industry. We have developed two SILC-based versions of an existing cloth simulation code written in C according to the proposed application styles. In the rest of the paper, we describe how SILC was applied to the original cloth simulation code, and present some experimental results. We also make a brief survey of related work and finally draw some conclusions.

2 Overview of the SILC Matrix Computation Framework

Simple Interface for Library Collections (SILC) is a matrix computation framework that allows users to use various matrix computation libraries independently of particular libraries, computing environments, and programming languages. SILC is currently implemented based on a client-server architecture. Instead of using matrix computation libraries through library-specific APIs, user programs for SILC (i.e., simulation codes in the SILC framework) utilize libraries in the following three steps. First, the user programs deposit data such as matrices and vectors into a SILC server, together with names for later reference. Next, the user programs make requests for computation by means of mathematical expressions in the form of text. These computation requests are translated into calls for appropriate library functions and carried out on the server side. Finally, the user programs retrieve the results of the computation (if necessary) from the server by specifying the names of the computation results to be retrieved. The computation results are kept in the server unless they are explicitly deleted.

Figure 1 shows a user program written in C in the SILC framework. This program solves an initial value problem of a two-dimensional diffusion equation using the Crank-Nicolson method. Suppose that t_0 is the initial time and $\Delta t > 0$ is a constant time interval. The Crank-Nicolson method requires the solution of a linear system $A\boldsymbol{x}_k = C\boldsymbol{x}_{k-1}$ for each time step $t_k = t_{k-1} + \Delta t$ ($k = 1, 2, 3, \ldots$), where A and C are sparse matrices. The user program first deposits A, C, and the initial values $\boldsymbol{x}_0$ at t_0 into a SILC server by three separate calls for the `SILC_PUT` routine. Then for each time step t_k, the program issues a request to solve $A\boldsymbol{x}_k = C\boldsymbol{x}_{k-1}$ using the `SILC_EXEC` routine. The computation request results in calls for some library functions, which are carried out in the server. After that, the program fetches the solution $\boldsymbol{x}_k$ at t_k by the `SILC_GET` routine.

The primary benefit of using SILC is independence from matrix computation libraries, computing environments, and programming languages. User programs for SILC do not depend on particular libraries and their underlying computing environments, as illustrated by the user program in Fig. 1. Sequential user programs can automatically obtain performance gains by simply using a parallel SILC server. SILC is also independent of programming languages in the sense that the same mathematical expressions can be used to make computation

```
silc_envelope_t A, C, x;

/* create matrices A, C, and the initial values x_0 at time t_0 */

SILC_PUT("A", &A);
SILC_PUT("C", &C);
SILC_PUT("x", &x); /* x_0 */
for (k = 1; k <= num_time_steps; k++) {
    SILC_EXEC("x = A \\ (C * x)");
    SILC_GET(&x, "x"); /* solution x_k at time t_k */
    /* output x_k */
}
```

Fig. 1. An example of a user program for SILC, written in C, which solves an initial value problem using the Crank-Nicolson method. The backslash operator for solving linear systems is represented by a backslash, which is used to escape special characters in string literals in C. Therefore, the operator is written as "\\" in the program.

requests from user programs in any programming language. Another benefit is ease of use. SILC makes it easy to write user programs that utilize matrix computation libraries, relieving users from the burden of using library-specific APIs that differ prominently in terms of data structures for matrices and vectors, parameters of library functions, compilation and linking procedures, and so on.

SILC comprises useful functionalities for matrix computations. Supported data types include dense, band, and sparse matrices and vectors. Mathematical expressions are composed of various math operators (such as arithmetic operators and the backslash operator for solving linear systems), built-in functions (e.g., function `norm2` computes the 2-norm of a vector), and subscripts (for example, `A[1:5,k:k+4]` yields a 5×5 submatrix of A). There is no construct for loops and conditional branching in the mathematical expressions of SILC, since SILC is intended to be a replacement for library calls. Control flows are expressed by the languages in which user programs are written, as shown in Fig. 1.

3 Two Application Styles of SILC

As a few basic guidelines on writing user programs for numerical simulations in the SILC framework, we have been proposing two different application styles (see Table 1 for a comparison of the two application styles).

Limited Application Style. User programs in this application style realize the most time-consuming, computationally intensive part of the user programs by depositing data into a SILC server, making requests for computation, and retrieving the results of the computation from the server. Those computations that are hard to realize in terms of matrix computations are implemented in the user programs by fetching data from the server and sending the results of computation back to the server. The limited application style is easy to use, although it imposes some communication overheads due to frequent data transfer between the user programs and the server. In addition, the maximum amount of data that can be handled is largely restricted by the memory capacity of a user program.

Table 1. A comparison of the limited and comprehensive application styles.

	Limited	Comprehensive
Ease of application	Easy	Hard
The amount of data maintained by a user program	Large	Small
The amount of data maintained by a SILC server	Small	Large
The amount of data transfer	Large	Small
The amount of parallelizable computation	Small	Large

Comprehensive Application Style. User programs in this application style first move all relevant data to a SILC server. After that, the user programs issue a series of computation requests to control the server-side computations, while having few data communications with the server in the middle of the simulations. The comprehensive application style can be difficult to employ since all computations are not necessarily easy to realize by means of SILC's mathematical expressions. On the other hand, the comprehensive application style imposes fewer communication overheads than the limited application style. The maximum amount of data mainly depends on the server's memory capacity, so that this application style allows a larger amount of data to be handled than the limited application style. Moreover, the amount of parallelizable computation is larger than in the limited application style, since most computations are done on the server side.

4 Cloth Simulation in SILC

With the aim of exemplifying the usability of SILC in numerical simulations, we applied the two application styles to an existing sequential cloth simulation code written in C. The simulation code employs a mass-spring model to represent the geometry of cloth and computes the motion of the cloth (governed by Newton's law of motion) based on the implicit backward Euler method [3].

The mass-spring model represents cloth as a mesh of n particles connected by springs. Let $\boldsymbol{x}_i \in \mathbf{R}^3$ be a position vector that specifies the location of particle i. We simply represent the geometry of the entire cloth by $\boldsymbol{x} \in \mathbf{R}^{3n}$. Similarly, we represent the velocity of particle i by $\boldsymbol{v}_i \in \mathbf{R}^3$ and those of all particles by $\boldsymbol{v} \in \mathbf{R}^{3n}$. Two particles are connected by a weightless spring k with a spring constant b_k, a damping constant h_k, and a natural length l_k.

In the implicit backward Euler method, we need to solve a linear system for each time step. Let $\boldsymbol{x}_0$ and $\boldsymbol{v}_0$ be the position and velocity of the cloth at the end of the previous time step. The main loop over time steps in the simulation code consists of the following three steps:

Step 1. Compute force $\boldsymbol{f} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v})$ and its derivatives $\partial \boldsymbol{f}/\partial \boldsymbol{x}$ and $\partial \boldsymbol{f}/\partial \boldsymbol{v}$. The force $\boldsymbol{f} \in \mathbf{R}^{3n}$ that acts on the cloth is calculated particle-wise as follows. Let P_i be the set of particles that are connected to particle i; then the force $\boldsymbol{f}_i \in \mathbf{R}^3$ that acts on particle i is defined as a sum of spring force $\boldsymbol{f}_{ij}$ and damping force

$\boldsymbol{d}_{ij}$ between each pair of particles i and j connected by spring k:

$$\begin{aligned}\boldsymbol{f}_i &= \sum_{j \in P_i} (\boldsymbol{f}_{ij} + \boldsymbol{d}_{ij}) \\ \boldsymbol{f}_{ij} &= b_k(|\boldsymbol{x}_j - \boldsymbol{x}_i| - l_k)\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|} \\ \boldsymbol{d}_{ij} &= -h_k(\boldsymbol{v}_i - \boldsymbol{v}_j)\end{aligned}$$

The derivatives $\partial \boldsymbol{f}/\partial \boldsymbol{x}$ and $\partial \boldsymbol{f}/\partial \boldsymbol{v}$ are Jacobian matrices [4], each of which consists of n^2 submatrices as follows:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}_1} & \cdots & \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{x}_n} \\ \vdots & & \vdots \\ \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{x}_1} & \cdots & \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{x}_n} \end{pmatrix}, \quad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} = \begin{pmatrix} \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{v}_1} & \cdots & \frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{v}_n} \\ \vdots & & \vdots \\ \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{v}_1} & \cdots & \frac{\partial \boldsymbol{f}_n}{\partial \boldsymbol{v}_n} \end{pmatrix}$$

Off-diagonal submatrices are defined as follows:

$$\frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j} = b_k I - \frac{b_k l_k}{|\boldsymbol{x}_j - \boldsymbol{x}_i|}\left\{ I - \frac{(\boldsymbol{x}_j - \boldsymbol{x}_i)(\boldsymbol{x}_j - \boldsymbol{x}_i)^T}{|\boldsymbol{x}_j - \boldsymbol{x}_i|^2} \right\}, \quad \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{v}_j} = h_k I$$

Diagonal submatrices are defined in terms of off-diagonal ones as follows:

$$\frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_i} = -\sum_{j \in P_i} \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j}, \quad \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{v}_i} = -\sum_{j \in P_i} \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{v}_j}$$

Step 2. Solve a linear system $A\Delta\boldsymbol{v} = \boldsymbol{b}$ to find a change in velocity $\Delta\boldsymbol{v}$, where

$$\begin{aligned} A &= M - \Delta t^2 \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} - \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} \\ \boldsymbol{b} &= \left\{ \boldsymbol{f} + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \boldsymbol{v}_0 \right\} \Delta t \end{aligned}$$

and M is a diagonal matrix that represents the mass of particles. The linear system is solved by the Conjugate Gradient (CG) method [5] since A is sparse and symmetric positive definite.

Step 3. Update velocity $\boldsymbol{v}$ and position $\boldsymbol{x}$ as follows:

$$\begin{aligned} \boldsymbol{v} &= \boldsymbol{v}_0 + \Delta\boldsymbol{v} \\ \boldsymbol{x} &= \boldsymbol{x}_0 + \boldsymbol{v}\Delta t \end{aligned}$$

The most time-consuming, computationally intensive part of the original simulation code is the second step, where solving linear systems takes about 80% of the original code's execution time. Therefore, we developed a SILC-based code in the limited application style by rewriting the second step of the original code by means of the SILC framework. Figure 2 (a) is part of the original code that calls for `lis_solve`, a library function of the Lis iterative solvers library [6], to

```
LIS_MATRIX A; LIS_VECTOR b, dv;

for (k = 1; k <= num_time_steps; k++) {
   /* 1. Compute f, ∂f/∂x, and ∂f/∂v */
      :
   /* 2. Solve AΔv = b */
   lis_solve(A, b, dv,
         lis_params,
         lis_options,
         lis_status);
   /* 3. Update velocity v and position x */
      :
}
```

(a) The original code

```
silc_evelope_t A, b, dv;

for (k = 1; k <= num_time_steps; k++) {
   /* 1. Compute f, ∂f/∂x, and ∂f/∂v */
      :
   /* 2. Solve AΔv = b */
   SILC_PUT("A", &A);
   SILC_PUT("b", &b);
   SILC_EXEC("dv = A \\ b");
   SILC_GET(&dv, "dv");
   /* 3. Update velocity v and position x */
      :
}
```

(b) The SILC-based code

Fig. 2. The original code and the SILC-based code in the limited application style.

solve a linear system $A\Delta\boldsymbol{v} = \boldsymbol{b}$ with the CG method. Figure 2 (b) is the same part of the SILC-based code in the limited application style, where the linear system is solved by depositing A and $\boldsymbol{b}$ into a SILC server, making a request for it to solve the linear system, and fetching the solution $\Delta\boldsymbol{v}$ from the server. In both codes, A is stored in the Compressed Row Storage (CRS) format [7].

We also developed a SILC-based code in the comprehensive application style, in which all relevant data is moved to a SILC server at the beginning of the simulation code and all computations are realized by means of SILC's mathematical expressions. The data transfer is performed in the initialization part of the code. Some computations are also carried out during the initialization, although most computations are concentrated in the main loop over time steps. Figure 3 shows the iterative part of the code in the comprehensive application style. The Jacobian $\partial\boldsymbol{f}/\partial\boldsymbol{v}$, referred to as `DfDv` in the figure, is computed during the initialization since it is constant. A few additional constant vectors and matrices are also defined in the initialization part. All matrices involved in the code are sparse and stored in the CRS format.

The mathematical expressions in Fig. 3 have been written so that data parallelism can be exploited as much as possible. For example, the mathematical expressions in the first call for `SILC_EXEC` are requests for computing the distance between two particles connected by a spring. Let s be the number of springs; then matrix `Y` is a linear map for transforming $\boldsymbol{x} \in \mathbf{R}^{3n}$ into $\boldsymbol{p} \in \mathbf{R}^{3s}$ so that each three elements of $\boldsymbol{p}$ represent $\boldsymbol{x}_j - \boldsymbol{x}_i$. The expression `p @* p` stands for elementwise multiplication, and `X_T` is another linear map from $\mathbf{R}^{3s}$ to $\mathbf{R}^{s}$ such that multiplying it by a vector sums up each three elements of the vector. Finally, function `sqrt` computes the square root of each element in a given vector. All these computations can be parallelized in a data-parallel manner.

5 Numerical Experiments

We conducted numerical experiments to investigate the performance of the SILC-based simulation codes. Table 2 shows the computing environments used for the experiments. We compared the performance of the original code and the

```
silc_envelope_t v, x;

/* Compute force f and Jacobian ∂f/∂x */
SILC_EXEC("p = Y * x; z = sqrt(X_T * (p *@ p))");
SILC_EXEC("fij = p *@ (X * (K_stiff *@ (z - L) /@ z))");
SILC_EXEC("dij = (Y * v) *@ (X * K_damp)");
SILC_EXEC("f = Mg - Y_T * (fij + dij)");

SILC_EXEC("zhat = ones(s, 1) /@ z");
SILC_EXEC("pzhat = p *@ (X * zhat)");
SILC_EXEC("U_L = sparse(U_L_row, U_col, pzhat, 3*n, s)");
SILC_EXEC("U_R = sparse(U_R_row, U_col, pzhat, 3*n, s)");
SILC_EXEC("U = U_L - U_R");
SILC_EXEC("tmp = zhat *@ K_stiff *@ L");
SILC_EXEC("T2 = Y_T * diag(X * tmp) * -Y");
SILC_EXEC("T3 = -U * diag(tmp) * U'");
SILC_EXEC("DfDx = T1 - T2 + T3");

/* Solve AΔv = b */
SILC_EXEC("A = M - (dt * dt) * DfDx - dt * DfDv");
SILC_EXEC("b = dt * (f + dt * (DfDx * v))");
SILC_EXEC("dv = A \\ b");

/* Update velocity v and position x */
SILC_EXEC("v += dv *@ fixed; x += dt * v");
SILC_GET(&v, "v");
SILC_GET(&x, "x");
```

Fig. 3. The SILC-based code in the comprehensive application style. Only the code segment within the main loop of the simulation is shown.

SILC-based codes by running them on the same PC (Dell Dimension 8400) and measuring their execution time for the first 20 time steps. The cloth used for the experiments consisted of 10,000 particles. The SILC-based codes were tested with two SILC servers, one in the same PC and another in SGI Altix 3700, both in the same Gigabit Ethernet LAN. The original code and local SILC server in the PC utilized a sequential version of the Lis iterative solvers library to solve linear systems, while the remote server in Altix employed an OpenMP-based parallel version of the same library. The execution time of the original code was 11.80 seconds; solving the linear systems took 81.70% of the execution time.

Table 3 shows the performance results of the SILC-based code in the limited application style using the local SILC server in the same PC and the remote SILC server running on different numbers of threads. The time spent for client-side computations and the time for data transfer were almost constant regardless of the number of threads, while the time spent for server-side computations was significantly reduced by the multithreaded server. As a result, a total execution time of 8.33 seconds was achieved by using the remote SILC server running on 16 threads. It constituted a 1.90 times speedup compared to the execution time with the remote SILC server on one thread, and a 1.42 times speedup compared to the original code.

Table 4 shows the performance results of the SILC-based code in the comprehensive application style using the local and remote SILC servers. There is no client-side computation since all data is maintained by a SILC server; only the velocity $\boldsymbol{v}$ and position $\boldsymbol{x}$ are retrieved from the server. The code showed good scalability, thanks to the mathematical expressions written so as to ex-

Table 2. The computing environments used for the experiments.

Name	Specifications
Dell Dimension 8400	Intel Pentium 4 3.4 GHz, 1 GB RAM, Windows XP SP2
SGI Altix 3700	Intel Itanium 2 1.3 GHz × 32, 32 GB RAM (cc-NUMA), Red Hat Linux Advanced Server 2.1

Table 3. Performance results of the SILC-based code in the limited application style. The three rows of client-side computations, data transfer, and server-side computations show the breakdowns of the total execution times (in seconds).

SILC server	Local	Remote					
Number of threads	–	1	2	4	8	16	32
Client-side computations	1.94	2.13	2.17	2.08	2.11	2.10	2.04
Data transfer	3.36	4.11	4.05	4.21	5.24	5.20	5.59
Server-side computations	9.98	9.58	6.29	3.61	1.57	1.03	1.57
Total execution time	15.28	15.82	12.52	9.90	8.92	8.33	9.20
Speedup	–	1.00	1.26	1.60	1.77	1.90	1.72

ploit data parallelism. However, the code was 2.09 times slower than the original code even with the remote SILC server running on 32 threads. This is mainly because of extra non-floating point operations present in the SILC-based code. For example, the mathematical expressions in Fig. 3 include four matrix-matrix multiplications, a transposition (by the ' operator), and two calls for the `sparse` function. All these operations create a sparse matrix in the CRS format as a result of computation through a number of non-floating point operations such as counting non-zero elements to be generated and packing them per row.

As described in Section 3, on the other hand, the comprehensive application style has certain advantages with regard to the amount of data on the client side and the amount of data transfer. The amount of data maintained by the SILC-based code in the limited application style is 16.8 MB, while the amount of data in the SILC-based code in the comprehensive application style is 3.82 MB when data is deposited into a SILC server at the beginning of the code and is reduced to 0.763 MB after the initialization. Similarly, the amount of data transfer per time step is 17.7 MB in the case of the limited application style, whereas it is 0.458 MB in the case of the comprehensive application style. These advantages of the comprehensive application style allow a larger piece of cloth to be simulated even in a PC with a restrictive memory capacity, together with a remote SILC server running in a high-performance parallel computer.

6 Related Work

Since SILC is a piece of middleware based on a client-server architecture, it is related to Grid RPC middleware such as Ninf-G [8] and NetSolve [9]. In these systems, numerical simulation codes are usually parallelized in a task-parallel manner. It is common in Grid computing to employ geographically distributed

Table 4. Performance results of the SILC-based code in the comprehensive application style. The two rows of data transfer and server-side computations show the breakdowns of the total execution times (in seconds).

SILC server	Local	Remote					
Number of threads	–	1	2	4	8	16	32
Data transfer	1.66	2.35	1.71	1.23	1.05	1.06	1.40
Server-side computation	434.91	335.09	238.49	114.67	56.46	32.23	23.27
Total execution time	436.57	337.44	240.20	115.90	57.51	33.28	24.67
Speedup	–	1.00	1.40	2.91	5.87	10.14	13.68

servers, which makes it prohibitive to exchange data among the servers to perform computations in a data-parallel manner. To address this issue, Tanaka *et al.* [10] proposed a hybrid programming model which combines Ninf-G and MPI. In this model, tasks are sent to multiple servers via Grid RPC, while each task is carried out within a server in a data-parallel manner based on MPI. In Ninf-G, however, it is the user's responsibility to write the MPI-based parallel codes the servers perform. In SILC, computation requests are expressed by means of SILC's mathematical expressions and automatically parallelized in a data-parallel manner, so that users can write numerical simulation codes without knowing the details of parallel computations on the server side.

The use of mathematical expressions to express computation requests in an environment-independent manner is closely related to parallel implementations of Matlab. There are four major categories of parallel Matlab systems: (1) embarrassingly parallel, (2) message passing, (3) back-end support, and (4) Matlab compilers [11]. Among them, those systems in the second and third categories are relevant to SILC. A typical example in the second category is MatlabMPI [12], which allows users to write MPI-based parallel codes in Matlab. User programs for SILC can also be MPI-based parallel programs. In addition, SILC can be used in sequential programs; if that is the case, users can automatically gain the benefit of parallel computations by just using parallel SILC servers. The same approach is taken by parallel Matlab systems of the third category. A representative system in this category is Star-P [13, 14] which enables a parallel back-end server to be interactively utilized in Matlab. The most significant difference from SILC is that in Star-P, local data in Matlab and remote data in the back-end server are seamlessly handled in such a way that the data on both sides can be referred to and used in one mathematical expression without any restriction. In SILC, data management in a SILC server is completely separated from user programs. This design decision has made SILC's client-side API relatively simple, allowing the SILC framework to be used in a number of programming languages including C, Fortran, Java, Python, and GNU Octave. In addition, user programs for SILC can be executed in large-scale distributed parallel computing environments based on batch queuing systems [1]. In Star-P, the focus is on the seamless integration of a back-end server with Matlab and their interactive utilization, while SILC focuses on a higher degree of independence from computing environments and programming languages.

7 Concluding Remarks

This paper presented a case study of numerical simulations in the SILC framework. In this study, we adapted an existing cloth simulation code to the SILC framework and obtained two SILC-based versions of the code according to the proposed application styles. Using a remote parallel SILC server, the SILC-based code in the limited application style outperformed the original code while the SILC-based code in the comprehensive application style achieved good scalability. These results demonstrate the feasibility of numerical simulations within the SILC framework and the usability of the proposed application styles.

Our future work includes a performance evaluation of the SILC-based cloth simulation codes in MPI-based parallel computing environments, performance tuning of SILC servers for faster execution of user programs in the comprehensive application style, and further case studies with other types of numerical simulations such as computational fluid dynamics.

Acknowledgment. This research was supported by a grant-in-aid project [6] in the Core Research for Evolutional Science and Technology (CREST) program of Japan Science and Technology Agency.

References

1. Kajiyama, T., Nukada, A., Suda, R., Hasegawa, H., Nishida, A.: Distributed SILC: An easy-to-use interface for MPI-based parallel matrix computation libraries. In: Proc. Para '06, LNCS, Springer, in press. (2006) http://ssi.is.s.u-tokyo.ac.jp/silc/.
2. Kajiyama, T., Nukada, A., Suda, R., Hasegawa, H., Nishida, A.: Numerical simulations in the SILC matrix computation framework. In: Proc. ICCM 2007. (2007)
3. Baraff, D., Witkin, A.: Large steps in cloth simulation. In: Proc. ACM SIGGRAPH '98. (1998) 43–54
4. Lang, S.: Calculus of Several Variables, Second Edition. Addison-Wesley (1979)
5. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards **49** (1952) 409–436
6. Nishida, A., Kotakemori, H., Kajiyama, T., Nukada, A.: Scalable software infrastructure project. In: Proc. SC06, poster. (2006) http://ssi.is.s.u-tokyo.ac.jp/.
7. Barrett, R., *et al.*: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM (1994)
8. Ninf Project: http://ninf.apgrid.org/.
9. NetSolve: http://icl.cs.utk.edu/netsolve/.
10. Tanaka, Y., Takemiyia, H., Nakada, H., Sekiguchi, S.: Design and implementation of flexible, robust and efficient Grid-enabled hybrid QM/MD simulation. Computational Methods in Science and Technology **12** (2006) 79–87
11. Choy, R., Edelman, A.: Parallel MATLAB: Doing it right. Proceedings of the IEEE **93** (2005) 331–341
12. Kepner, J., Ahalt, S.: MatlabMPI. Journal of Parallel and Distributed Computing **64** (2004) 997–1005
13. Shah, V., Gilbert, J.R.: Sparse matrices in MATLAB*P: Design and implementation. In: Proc. HiPC 2004, LNCS 3296. (2004) 144–155
14. Interactive Supercomputing, Inc.: http://www.interactivesupercomputing.com/.