Distributed SILC: An Easy-to-Use Interface for MPI-Based Parallel Matrix Computation Libraries

Tamito KAJIYAMA[1,2], Akira NUKADA[1,2], Reiji SUDA[2,1], Hidehiko HASEGAWA[3], and Akira NISHIDA[4,1]

[1] CREST, Japan Science and Technology Agency, Saitama 332–0012, Japan
[2] The University of Tokyo, Tokyo 113–8656, Japan
[3] University of Tsukuba, Tsukuba 305–8550, Japan
[4] 21st Century COE Program, Chuo University, Tokyo 112–8551, Japan

Abstract. The present paper describes the design and implementation of distributed SILC (Simple Interface for Library Collections) that gives users access to a variety of MPI-based parallel matrix computation libraries in a flexible and environment-independent manner. Distributed SILC allows users to make use of MPI-based parallel matrix computation libraries not only in MPI-based parallel user programs but also in sequential user programs. Since user programs for SILC are free of a source-level dependency on particular libraries and computing environments, users can easily utilize alternative libraries and computing environments without any modification in the user programs. The experimental results of two test problems showed that the implemented SILC system achieved speedups of 2.69 and 7.54 using MPI-based parallel matrix computation libraries with 16 processes.

1 Introduction

The traditional way of using matrix computation libraries based directly on library-specific application programming interfaces usually leads to a source-level dependency on the libraries in use. This source-level dependency is the primary reason why users (i.e., application programmers) are often required to make a considerable amount of modifications to their user programs, for example when porting them to other computing environments or when trying out other libraries having different sets of solvers, matrix storage formats, arithmetic precisions, and so on. To address this issue inherent in the traditional programming style, we have been proposing an easy-to-use application framework named Simple Interface for Library Collections (SILC) [1, 2]. A user program in the SILC framework first deposits data such as matrices and vectors into a separate memory space. Next, the user program makes requests for computation by means of mathematical expressions in the form of text. These requests are translated into calls for appropriate library functions, which are carried out in the separate memory space independently of the user program. Finally, the user program fetches the results of computation from the separate memory space.

```
double *A, *B;
int desc_A[9], desc_B[9], *ipiv, info;
/* create matrix A and vector B */
PDGESV(N, NRHS, A, IA, JA, desc_A, ipiv, B,
       IB, JB, desc_B, &info);
/* solution X is stored in B */
```

(a)

```
silc_envelope_t A, b, x;
/* create matrix A and vector B */
SILC_PUT("A", &A);
SILC_PUT("b", &b);
SILC_EXEC("x = A \\ b"); /* call PDGESV() */
SILC_GET(&x, "x");
```

(b)

Fig. 1. A comparison of two C programs (a) in the traditional programming style and (b) in the SILC framework, both making use of ScaLAPACK to solve a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$.

Figure 1 shows two user programs written in C, one in the traditional programming style and the other in the SILC framework. The traditional user program shown in Fig. 1 (a) prepares matrix A and vector $\boldsymbol{b}$ using library-specific data structures and makes a call for a library function in ScaLAPACK [3] to solve a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$. The user program for SILC shown in Fig. 1 (b) realizes the same computation using the following three routines: `SILC_PUT` to deposit A and $\boldsymbol{b}$ into a separate memory space, `SILC_EXEC` to issue a request for solution of the linear system by means of a mathematical expression in the form of text, and `SILC_GET` to retrieve the solution $\boldsymbol{x}$. The mathematical expression specified as the argument of `SILC_EXEC` is translated into a call for the library function in ScaLAPACK for example, and carried out in the separate memory space.

We have developed a SILC system for sequential and shared-memory parallel computing environments [1, 2]. The current implementation of SILC is based on a client-server architecture, in which a user program is a client of a SILC server running in a remote computing environment. Since a user program for SILC does not contain any library-specific code, no modification to the user program is required to utilize alternative matrix computation libraries. Moreover, users can automatically gain the advantages of parallel computation by using a SILC server that runs in a parallel computing environment. The main overhead in using SILC, on the other hand, is the cost of data communications between a user program and a SILC server. However, it is not difficult to reduce the relative amount of communication overhead, since the time complexities of matrix computations tend to be larger than their space complexities. For instance, solving a dense linear system with N unknowns takes $O(N^3)$ time, while the time necessary for data communications is of $O(N^2)$. Consequently, in many cases the use of a faster matrix computation library and computing environment results in good speedups even at the cost of data communications.

2 SILC for Distributed Parallel Computing Environments

We have been developing a SILC system for distributed parallel computing environments that allows users to make use of MPI-based matrix computation libraries in a flexible and computing environment-independent manner. The pri-

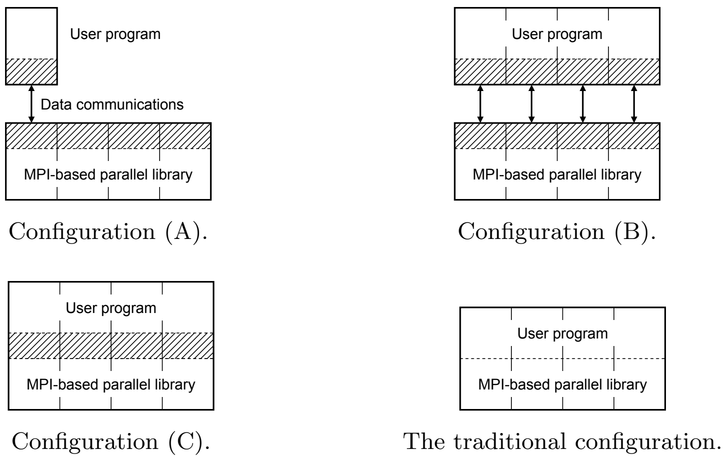
User program
Data communications
MPI-based parallel library
Configuration (A).
User program
MPI-based parallel library
Configuration (B).
User program
MPI-based parallel library
Configuration (C).
User program
MPI-based parallel library
The traditional configuration.

Fig. 2. Three system configurations of distributed SILC, compared with the configuration of an MPI-based parallel program in the traditional programming style.

mary goal in the design of distributed SILC is to support as many MPI-based parallel matrix computation libraries and computing environments as possible, since SILC is a piece of middleware placed between user programs and matrix computation libraries, serving as an abstraction layer that hides the details of the libraries and underlying computing environments. Having this design goal in mind, we consider three system configurations shown in Fig. 2. The shaded parts in the figure show the components that SILC provides. For comparison, the figure also shows the configuration of a user program in the traditional programming style. The traditional user program in this example consists of four MPI processes.

Configurations (A) and (B) are based on a client-server architecture in which a user program is a client of an MPI-based parallel SILC server. The user program establishes a TCP connection to the SILC server and makes use of MPI-based parallel matrix computation libraries managed by the server. The user program in (A) is sequential, while the user program in (B) is an MPI-based parallel program. Both the server and the user program in (B) shown in Fig. 2 consist of four MPI processes.

In Configuration (A), the user program makes a connection to one of the server processes through which both data and requests for computation are transferred. The server distributes the received data among the server processes by means of a data redistribution mechanism, keeping the data among the server processes in a distributed manner. The data redistribution mechanism is also utilized to make a change in data distributions in the following two situations. One situation is when the server handles requests for computation, where the data is passed to a library function as an argument in a different data distribution the library function accepts. The other situation is when the user program fetches the results of preceding computation requests, where the data is transferred in the data distribution that the user program requires. Computation requests by means of textual mathematical expressions are handled by the library interface in the server. The interface incorporates an interpreter that translates the expressions into calls for appropriate library functions, which are carried out within the server processes.

In Configuration (B), each process of the MPI-based parallel user program makes a separate connection to one of the server processes; that is, multiple connections are established between the user program and the server. Data is retained in a distributed manner in both the user program and the SILC server, and parallel data transfer is performed between the user program and the server through the multiple connections. The data redistribution mechanism is employed in the same manner as Configuration (A), when the server needs to change distributions of data. Requests for computation, on the other hand, are sent to the server from one process on behalf of the user program. Since a user program in Configuration (B) is an ordinary MPI-based program, it can be executed, for example, as follows:

```
mpirun -np n ./my_silc_application
```

where n is the number of processes on which the program runs. At the moment, the number of processes of the user program must be smaller than or equal to the number of the server's processes.

Configuration (C) is prepared for some restrictive computing environments in which the client-server architecture cannot be adopted. In this configuration, the data redistribution mechanism and library interface of the SILC server are implemented as a library, which is linked to MPI-based parallel user programs together with MPI-based parallel matrix computation libraries. There is no source-level difference between a user program in Configuration (B) and another program in (C); that is, the source code of the two programs is the same, so that these configurations can be exchanged without any modification to the source code. Unlike Configurations (A) and (B), on the other hand, library functions in this configuration are carried out within the processes of a user program.

3 Experiments

To determine whether the implemented SILC system is capable of achieving speedups when compared with the traditional programming style, we conducted experiments with regard to the following two test problems.

Problem 1. A dense linear system $A\boldsymbol{x} = \boldsymbol{b}$.
Problem 2. An initial value problem of a partial differential equation (PDE).

Table 1 is a summary of the computing environments used in the experiments. These computing environments are in the same Gigabit Ethernet (GbE) LAN. Both Xeon4 and Xeon8 consist of a disjoint set of nodes in the same GbE-based PC cluster. Only one core of each node was used. Computation was done in double precision real throughout the experiments.

Table 1. The computing environments used in the experiments.

Host name	Specifications
Xeon4	IBM eServer xSeries 335 (dual Intel Xeon 2.8 GHz, L2 cache 512 KB, Memory 1 GB) $\times$ 4, Red Hat Linux 8.0, LAM/MPI 7.0
Xeon8	Different 8 nodes in the same PC cluster as Xeon4
Altix	SGI Altix 3700 (Intel Itanium2 1.3 GHz $\times$ 32, L2 cache 256 KB, Memory 32 GB), Red Hat Linux Advanced Server 2.1, SGI MPI 4.4 (MPT 1.9.1)

3.1 Problem 1: Solution of a Dense Linear System

Consider solving a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ using the `PDGESV` routine in ScaLAPACK, where A is an $N \times N$ dense matrix and $\boldsymbol{b}$ and $\boldsymbol{x}$ are N-vectors. We prepared the following two user programs, both of which are MPI-based parallel programs written in C.

Program P_1 in the traditional programming style. The program first prepares A and $\boldsymbol{b}$ in the two-dimensional block-cyclic distribution [3]. The elements of A are random numbers, while those of $\boldsymbol{b}$ are given so that all elements of solution $\boldsymbol{x}$ will be 1. Then, the program makes a call for `PDGESV` to solve the linear system. The time elapsed in the ScaLAPACK routine was measured as the execution time of the program.

Program P_2 in the SILC framework. The program also prepares A and $\boldsymbol{b}$ in the same way as P_1. Next, the program makes two calls for `SILC_PUT` to deposit A and $\boldsymbol{b}$ into a SILC server, respectively, and another call for `SILC_EXEC` to request the solution of the linear system. This request is translated into a call for the `PDGESV` routine, which is carried out on the server side. Finally, the program calls for `SILC_GET` to retrieve the solution $\boldsymbol{x}$ from the server. We prepared three SILC servers running in Xeon4, in Xeon8, and in Altix. The elapsed time from the connection to a server to the data transfer of $\boldsymbol{x}$ was measured as the execution time of the program.

Table 2 summarizes the computing environments used for Problem 1. Both P_1 and P_2 were executed with 4 processes in Xeon4, whereas the SILC servers used by P_2 were executed with 4 processes in Xeon4, with 8 processes in Xeon8, and with 16 processes in Altix. Since P_2 is an MPI-based parallel program, this configuration corresponds to Configuration (B) shown in Fig. 2. Timing was done by the `gettimeofday` system call.

Table 3 shows the experimental results, where T is execution time in seconds, S is speedup (i.e., a ratio of the execution time of P_1 to that of P_2), and C is a proportion of communication overhead to the execution time of P_2 (we assumed $C = (T - T_{comp})/T$, where T_{comp} is a computation time measured on the server side). The execution time of P_2 includes the time for the data

Table 2. The computing environments used for Problem 1.

Label	User program	SILC server	Configuration
Trad.	P_1 in Xeon4 (4 PEs)	–	–
SILC (local)	P_2 in Xeon4 (4 PEs)	Xeon8 (4 PEs)	(B)
SILC (remote #1)	P_2 in Xeon4 (4 PEs)	Xeon8 (8 PEs)	(B)
SILC (remote #2)	P_2 in Xeon4 (4 PEs)	Altix (16 PEs)	(B)

Table 3. The results of Problem 1 (solution of a dense linear system $A\boldsymbol{x} = \boldsymbol{b}$). T is execution time in seconds, S is speedup, and C is communication overhead.

	Trad.	SILC (local)		SILC (remote #1)		SILC (remote #2)	
N	T	T (S)	C	T (S)	C	T (S)	C
1,000	1.592	1.417 (1.12)	7.8%	2.179 (0.73)	9.5%	0.790 (2.02)	16.0%
2,000	5.403	5.453 (0.99)	6.5%	5.789 (0.93)	11.1%	2.827 (1.91)	13.8%
4,000	27.153	30.145 (0.90)	4.0%	22.626 (1.20)	9.8%	14.235 (1.91)	10.2%
8,000	186.991	208.880 (0.90)	2.3%	130.892 (1.43)	6.5%	69.481 (2.69)	8.3%

transfer and distribution of A and $\boldsymbol{b}$, as well as the time for the collection and data transfer of $\boldsymbol{x}$. These data communications constitute the major overhead in using SILC. However, as indicated by the proportion C in Table 3, the cost of data communications becomes relatively smaller as dimension N increases, because the solution of the dense linear system requires a computation time on the order of $O(N^3)$, while the data communications take only $O(N^2)$ time. Since the computation time can be significantly reduced by using a faster computing environment via SILC, some speedups are expected to be achieved even at the cost of data communications when N is large. This holds true for the experimental results shown in Table 3 – the speedups in the case of $N = 8,000$ were 1.43 with the SILC server in Xeon8 and 2.69 with the server in Altix.

3.2 Problem 2: Solution of an Initial Value Problem of a PDE

We solve the two-dimensional time-dependent diffusion equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ $(t \geq 0)$ in the region $0 \leq x \leq 1$ and $0 \leq y \leq 1$ subject to the initial condition

$$u(x,\, y,\, 0) = \begin{cases} 1 & \text{if } |x - 0.5| < 0.1 \text{ and } |y - 0.5| < 0.1, \\ 0 & \text{otherwise,} \end{cases}$$

and boundary conditions $u(0,\, y,\, t) = u(1,\, y,\, t) = u(x,\, 0,\, t) = u(x,\, 1,\, t) = 0$ for $t > 0$, using the Crank-Nicolson method [4]. Suppose $t_0 = 0$ is the initial time and $\Delta t > 0$ is a constant time interval, and define the k-th time step as $t_k = t_{k-1} + \Delta t$. In using the Crank-Nicolson method, we have to solve a system of linear equations $A\boldsymbol{x}_k = \boldsymbol{b}_k$ for each time step, where A is an $N \times N$ sparse matrix and $\boldsymbol{b}_k$ and $\boldsymbol{x}_k$ are N-vectors. $\boldsymbol{b}_k$ is defined as $\boldsymbol{b}_k = C\boldsymbol{x}_{k-1}$, i.e. the matrix-vector product of another $N \times N$ sparse matrix C and the solution $\boldsymbol{x}_{k-1}$ at t_{k-1}. We prepared the following two user programs, both of which are sequential programs written in C.

Table 4. The computing environments used for Problem 2.

Label	User program	SILC server	Configuration
Trad.	P_1 in Xeon4 (1 PE)	–	–
SILC (local)	P_2 in Xeon4 (1 PE)	Xeon4 (4 PEs)	(A)
SILC (remote #1)	P_2 in Xeon4 (1 PE)	Xeon8 (8 PEs)	(A)
SILC (remote #2)	P_2 in Xeon4 (1 PE)	Altix (16 PEs)	(A)

Table 5. The results of Problem 2 (solution of an initial value problem of a PDE). T is execution time in seconds, S is speedup, and C is communication overhead.

	Trad.	SILC (local)		SILC (remote #1)		SILC (remote #2)	
N	T	T (S)	C	T (S)	C	T (S)	C
10,000	0.432	0.693 (0.62)	46.54%	1.040 (0.42)	41.34%	0.423 (1.02)	56.8%
40,000	5.019	3.164 (1.59)	33.55%	3.756 (1.34)	39.32%	1.209 (4.15)	38.2%
90,000	19.206	8.587 (2.24)	28.83%	7.402 (2.59)	24.08%	2.981 (6.44)	31.1%
160,000	43.118	17.617 (2.45)	22.72%	13.850 (3.11)	22.69%	6.078 (7.09)	27.1%
250,000	82.798	30.627 (2.70)	17.78%	22.505 (3.68)	19.77%	10.987 (7.54)	23.1%

Program P_1 in the traditional programming style.

1. Prepare matrices A and C and the initial solution $\boldsymbol{x}_0$ at t_0. The matrices are stored in the Compressed Row Storage (CRS) format [5].
2. For each time step t_k ($k = 1, 2, 3, \ldots$):
   (a) Compute $\boldsymbol{b}_k = C\boldsymbol{x}_{k-1}$ using the sparse matrix-vector product routine in the sequential version of an iterative solvers library Lis [6].
   (b) Solve $A\boldsymbol{x}_k = \boldsymbol{b}_k$ using the Conjugate Gradient (CG) method [5] in Lis with a zero initial guess.

Program P_2 in the SILC framework.

1. Prepare matrices A and C and vector $\boldsymbol{x}_0$ in the same way as P_1.
2. Send A, C, and $\boldsymbol{x}_0$ to a SILC server using `SILC_PUT`. In the server, the data is distributed among the server processes.
3. For each time step t_k ($k = 1, 2, 3, \ldots$):
   (a) Send a request for computation by `SILC_EXEC` to compute $\boldsymbol{b}_k = C\boldsymbol{x}_{k-1}$ using a parallel sparse matrix-vector product routine.
   (b) Send another request with `SILC_EXEC` to solve $A\boldsymbol{x}_k = \boldsymbol{b}_k$ using the CG method in the MPI-based parallel version of Lis with a zero initial guess.
   (c) Receive $\boldsymbol{x}_k$ from the SILC server using `SILC_GET`.

The library routines used by P_1 are sequential, whereas those carried out by the SILC server are MPI-based parallel routines. Table 4 shows the computing environments used for Problem 2. We executed both P_1 and P_2 in a node of Xeon4 using a single processor, and measured their execution times for the first 40 time steps using the `gettimeofday` system call. We used the same SILC servers as those in Problem 1 to run P_2. Since P_2 is a sequential program, this configuration corresponds to Configuration (A) shown in Fig. 2.

Table 5 shows the experimental results. In comparison with Program P_2 for Problem 1, P_2 for this problem consumed a relatively large proportion of the execution time in depositing A, C, and $\boldsymbol{x}_0$ into the server and fetching $\boldsymbol{x}_k$ for each time step. Suppose $K = 40$ is the number of time steps, α is the iteration count of the CG method, and $\beta = 5N - 4\sqrt{N}$ is the number of non-zero elements in A and C. Then, the number of floating-point operations for sparse matrix-vector product $C\boldsymbol{x}_{k-1}$ is 2β, while that for solving $A\boldsymbol{x}_k = \boldsymbol{b}_k$ with the CG method is $4N + \alpha(2\beta + 12N + 3)$. Therefore, the matrix computations in P_1 and P_2 require a computation time on the order of $O(\alpha KN)$. On the other hand, data communications between P_2 and a SILC server require a communication time on the order of $O(KN)$. That is, the ratio of the computation time to the communication time is almost proportional to α, which is small compared to N in this test problem. In other words, this problem is somewhat disadvantageous to SILC in the sense that P_2 can hardly yield a speedup in the first place. However, in the experiments we observed speedups of 3.68 using the SILC server in Xeon8 and 7.54 using the server in Altix in the case of $N = 250,000$, by means of faster matrix computations in these remote computing environments.

3.3 Observations

The experimental results of the two test problems showed that the implemented SILC system is capable of achieving speedups when it deals with large problems. Although SILC imposes some communication overhead due to the data transfer between a user program and a SILC server, the overhead can be offset by speedups through the faster matrix computation libraries and computing environment that SILC makes available. The overhead can also be reduced by means of a faster interconnect between the user program and the server.

In addition to the quantitative benefit of speedups, SILC also provides a qualitative benefit in that it enables MPI-based parallel matrix computation libraries to be used not only in MPI-based parallel user programs but also in sequential user programs. In fact, P_2 in Problem 1 was an MPI-based parallel program, while P_2 in Problem 2 was a sequential program. The former used ScaLAPACK and the latter employed Lis, both in a remote MPI-based parallel computing environment. Since the SILC server to be used by a user program can be specified outside the user program, various computing environments as well as the matrix computation libraries that are available in the computing environments can be evaluated one after another without any modification to the user program.

4 Related Work

Improving the utility of matrix computation libraries is a major research topic in the areas of high-performance computing and Grid computing.

The Trilinos project [7] has been proposing a framework for integrating matrix computation libraries into a C++ class library and developing a number of libraries for numerical linear algebra. The application programming interface (API) of each library is consistent with others' in terms of (1) common data structures of matrices and vectors, and (2) common abstract classes based on which users define solvers by inheritance. The libraries are also organized as Trilinos packages by means of (3) common directory structures and installation procedures. However, the libraries vary in the details of their APIs; for example, the API of a dense direct solvers library and that of an iterative solvers library, both developed in the Trilinos project, are not exactly the same, so that users are required to modify their user programs to utilize one library instead of the other in use. In SILC, requests for computation are issued by means of textual mathematical expressions through which any libraries (even having incompatible APIs) can be utilized in the same way in any programming languages.

Amesos [8] is a C++ class library which gives access to various direct linear solvers through a common API. The library provides good support for many existing libraries based on different parallelization techniques, including sequential libraries such as LAPACK and parallel libraries such as ScaLAPACK. Amesos focuses on direct solvers for dense matrices, whereas SILC provides support for a wider range of matrix computations in a language-independent manner.

Since SILC is a piece of middleware based on a client-server architecture, our framework shares some functionalities with Grid computing middleware such as Ninf-G [9] and NetSolve [10]. Ninf-G is a middleware system for realizing Remote Procedure Call (RPC) in Grid computing environments. Ninf-G allows user programs to carry out MPI-based parallel matrix computation libraries in remote distributed parallel computing environments. In Ninf-G, a particular call for a remote procedure takes place in one process; that is, RPC is carried out sequentially either in a sequential user program or in a process of an MPI-based parallel user program [11]. All input and output data is once gathered to one of the remote processes by which the remote procedure is carried out in parallel. In addition, users are required to specify the ways of distributing the input data to the other processes as well as of collecting output data to the sending process, both by means of Ninf-G's interface description language. In contrast, SILC enables data transfer between a user program and a SILC server to be performed in parallel by means of Configuration (B) shown in Fig. 2, allowing the user program and the server to avoid the data redistribution to/from one process before the data transfer. Moreover, users do not have to care about the details of the data redistribution on the server side as long as supported matrix storage formats are in use.

5 Concluding Remarks

This paper described the design and implementation of a SILC system for distributed parallel computing environments. By using this system, MPI-based parallel matrix computation libraries can be utilized not only in MPI-based parallel user programs but also in sequential user programs. Moreover, no modification to the user programs is required to make use of different computing environments.

The experimental results of two test problems showed that some speedups are feasible by using faster matrix computation libraries in distributed parallel computing environments via SILC, provided that the amount of matrix computations is large enough to reduce the relative amount of communication overhead due to data transfer between a user program and a SILC server. In the experiments, the implemented SILC system achieved speedups of 2.69 in Problem 1 using ScaLAPACK and 7.54 in Problem 2 using an iterative solvers library Lis, both through a remote SILC server that runs on 16 processes.

The primary subjects of our future study include an implementation of Configuration (C) shown in Fig. 2, a quantitative analysis concerning the cost of data communications, a proposal of a performance evaluation model for distributed SILC with emphasis on the communication overhead [12], and the development of plug-in modules for integrating various existing matrix computation libraries into the SILC framework.

Acknowledgments. The authors would like to thank the session chairperson Dr. Xiaoye Sherry Li and anonymous reviewers for insightful remarks. This research was supported by a grant [13] from the Core Research for Evolutional Science and Technology (CREST) of the Japan Science and Technology Agency.

References

1. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: SILC: Flexible and environment independent interface for matrix computation libraries. In: Proc. PPAM 2005, LNCS 3911. (2006) 928–935 http://ssi.is.s.u-tokyo.ac.jp/silc/.
2. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: LAPACK in SILC: Use of a flexible application framework for matrix computation libraries. In: Proc. HPC Asia 2005. (2005) 205–212
3. Blackford, L.S., *et al.*: ScaLAPACK Users' Guide. SIAM (1997)
4. Smith, G.D.: Numerical Solution of Partial Differential Equations. Oxford University Press (1965)
5. Barrett, R., *et al.*: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM (1994)
6. SSI Project: User's Manual for Lis 1.0.2. (2006) http://ssi.is.s.u-tokyo.ac.jp/lis/.
7. Heroux, M. A., *et al.*: An overview of the Trilinos project. ACM Transactions on Mathematical Software **31** (2005) 397–423
8. Sala, M.: On the design of interfaces to serial and parallel direct solver libraries. Technical Report SAND–2005–4239, Sandia National Laboratories (2005)
9. Ninf Project: http://ninf.apgrid.org/.
10. NetSolve: http://icl.cs.utk.edu/netsolve/.
11. Takemiya, H., Tanaka, Y., Nakada, H., Sekiguchi, S.: Development and execution of large scale grid applications using MPI and GridRPC: Hybrid QM/MD simulation. IPSJ Trans. on Advanced Computing Systems **46** (2005) 384–395 in Japanese.
12. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: A performance evaluation model for the SILC matrix computation framework. In: Proc. IFIP Intl. Conf. on Network and Parallel Computing. (2006) 93–103
13. Nishida, A., Kotakemori, H., Kajiyama, T., Nukada, A.: Scalable software infrastructure project. In: Proc. SC06, poster. (2006)