# Development of General Purpose Numerical Software Infrastructure for Large Scale Scientific Computing

Project Representative
Akira Nishida Institute of Science and Technology, Chuo University

Author
Akira Nishida Institute of Science and Technology, Chuo University

The Development of Software Infrastructure for Large Scale Scientific Simulation project, or the Scalable Software Infrastructure (SSI) project for short, was initiated in November 2002, for the purpose of constructing a scalable software infrastructure to expand large scale computing environments to replace existing implementations of parallel algorithms and implementations in individual scientific fields. It contains iterative solvers for linear systems, fast integral transforms, their effective implementations to incorporate high-performance computers of various types, and joint studies performed with institutes and computer vendors, in order to implement the developed libraries in advanced environments to become major platforms in the next ten years. An object-oriented programming model was adopted to enable complex libraries to be built by combining elementary mathematical operations. Implemented algorithms are selected from the viewpoint of scalability on massively parallel computing environments. Some libraries adopted automatic performance tuning mechanisms to fit their kernels on targeting architectures. We also proposed a parallel scripting language SILC as an interface to these libraries, patents for the main concepts of which are in application. The libraries are distributed freely via the Internet, and intended be improved by feedback from the users. Since the first announcement in September 2005, we have been updating the libraries to reflect demands of users. Since 2006, we have started a joint project with the Earth Simulator Center, to enable our libraries to run on massively parallel vector architectures, and released the results of our study as software.

**Keywords**: high performance computing, parallel algorithms, scalability, object-oriented programming, network distribution

## 1. Overview

To construct a software infrastructure for highly parallel computing environments, we must precisely predict future hardware technologies, and design scalable and portable software for these technologies. Based on detailed research, we installed various types of parallel computers to implement latest results in computing methods, maintaining scalability and portability. The installed architectures include a shared-memory parallel computer (SGI Altix 3700), a distributed-memory parallel computer (Cray XT3), a Linux-based PC cluster, and a personal vector computer (NEC SX-6i). Since 2003, we have signed contracts with the IBM T. J. Watson Research Center on the joint study of library implementation on massively parallel environments with tens of thousands of processors, using IBM Blue Gene/L. Since 2006, the SSI project has been selected for joint research with the Earth Simulator Center to port our libraries on parallel vector computing environments. The results of the SSI project will be evaluated on larger computers in the near future.

Based on the above policies, we have to carefully design the libraries so as to maintain portability and usability. In the SSI project, we have studied the object-oriented implementation of libraries and the languages for such implementation. The results are displayed on the object-oriented interface of the iterative solver library Lis and the autotuning mechanism of the fast Fourier transform library FFTSS. The libraries are written in C and are equipped with a Fortran interface, and we can add interfaces for higher-level languages if necessary. In addition, we have developed SILC, a simple interface for library collections, to be used in parallel environments. Patents are in application for the specifications and the extension of SILC to a scripting language.

## 2. Iterative Solvers for Linear Sytems

The members of this group hosted many formal and informal meetings to promote basic research and published many results on the conjugate gradient and conjugate-residual-based iterative solvers, and their preconditioning for linear equations and eigenvalue problems. In addition, this group released Lis, a library of iterative solvers for linear systems, including various solvers and preconditioners for many sparse matrix storage formats. These solvers and preconditioners are listed below.

Table 1  Solvers.

| | | | |
|---|---|---|---|
| 1.0.x | CG | Added in 1.1.0 | CR |
| | BiCG | | BiCR |
| | CGS | | CRS |
| | BiCGSTAB | | BiCRSTAB |
| | BiCGSTAB(l) | | GPBiCR |
| | GPBiCG | | BiCRSafe |
| | Orthomin(m) | | FGMRES(m) |
| | GMRES(m) | | |
| | TFQMR | | |
| | Jacobi | | |
| | Gauss-Seidel | | |
| | SOR | | |
| | IDR(s) | | |

Table 2  Preconditioners.

| | | | |
|---|---|---|---|
| 1.0.x | Jacobi | Added in 1.1.0 | Crout ILU |
| | ILU(k) | | ILUT |
| | SSOR | | Additive Schwarz |
| | Hybrid | | User defined preconditioner |
| | I+S | | |
| | SA-AMG | | |
| | SAINV | | |

Table 3  Matrix storage formats.

| | |
|---|---|
| Point | Compressed Row Storage |
| | Compressed Column Storage |
| | Modified Compressed Sparse Row |
| | Diagonal |
| | Ellpack-Itpack generalized diagonal |
| | Jagged Diagonal Storage |
| | Dense |
| | Coordinate |
| Block | Block Sparse Row |
| | Block Sparse Column |
| | Variable Block Row |

```
LIS_MATRIX        A;
LIS_VECTOR        b,x;
LIS_SOLVER        solver;
int               iter;
double            times,itimes,ptimes;

lis_initialize(argc, argv);
lis_matrix_create(LIS_COMM_WORLD,&A);
lis_vector_create(LIS_COMM_WORLD,&b);
lis_vector_create(LIS_COMM_WORLD,&x);
lis_solver_create(&solver);
lis_input(A,b,x,argv[1]);
lis_vector_set_all(1.0,b);
lis_solver_set_optionC(solver);
lis_solve(A,b,x,solver);
lis_solver_get_iters(solver,&iter);
lis_solver_get_times(solver,&times, &itimes,&ptimes);
printf("iter = %d time = %e (p=%e i=%e)\n",iter,times, ptimes, itimes);
lis_finalize();
```

Fig. 1 Example of the C program using Lis.

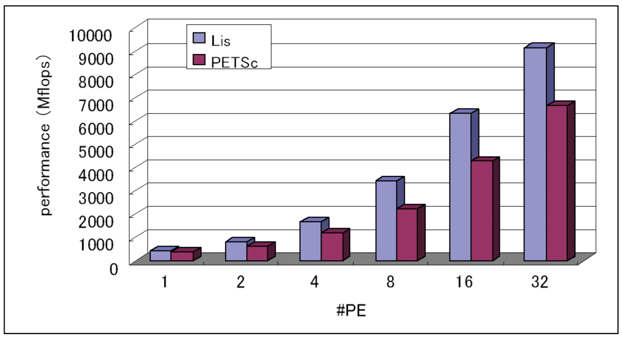

Fig. 2 Comparison of the MPI version of Lis and PETSc.

We present an example of the program using Lis in Fig. 1.

Feedback on Lis has been received from users, and Lis has gone through many improvements. Lis can be used on both small PC clusters and massively parallel computers, such as the Earth Simulator, IBM Blue Gene, and Cray XT. The code of Lis 1.1.2 has attained the vectorization ratio of 99.1% and the parallelization ratio of 99.99%. We show a comparison with the MPI version of Lis and PETSc, a library developed by Argonne National Laboratory, using the three-dimensional Poisson equation (size: one million, number of nonzero entries: 26,207,180) on an SGI Altix with 32 processors in Fig. 2.

In fields such as fluid dynamics and structural analysis, we must solve large-scale systems of linear equations to compute numerical solutions of partial differential equations, and the demand for efficient algorithms is great. In the SSI project, we have designed and implemented scalable and robust algorithms of iterative solvers for linear equations and their preconditioning, derived from such physical applications.

In recent years, multilevel algorithms for large-scale linear equations, such as algebraic multigrid (AMG), have been investigated in numerous studies. In most cases, multigrid methods show linear scalability, and the number of iteration counts is O(n) for a problem of size n. The algebraic multigrid method is based on a principle similar to the geometric multigrid, which utilizes the spatial information on physical problems, but this method differs from the geometric multigrid by considering the coefficient as a vertex-edge incidence matrix, In addition, by using only the information on the elements and their relations, this method generates coarser level matrices without higher frequency errors. The complexity of AMG is equivalent to geometric multigrid and can be applied to irregular or anisotropic problems. A conceptual image of AMG is shown in Fig. 3.

We proposed an efficient parallel implementation of AMG preconditioned conjugate gradient method based on smoothed aggregation (SA-AMGCG) and found that the proposed implementation provides better performance as the problem size becomes larger.

Currently, AMG is the most effective algorithm for general-purpose preconditioning, and its scalability is also remarkable. We have implemented AMG in Lis and have tested AMG in massively parallel environments. We pre-

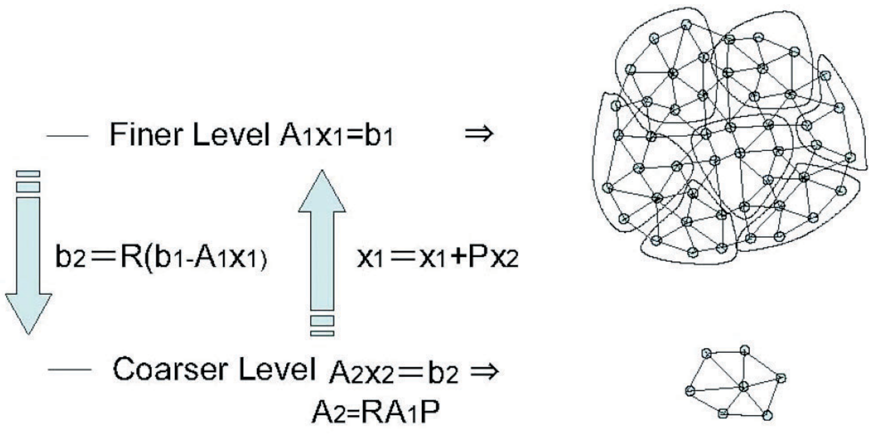

Fig. 3 Conceptual Image of the SA-AMG method.

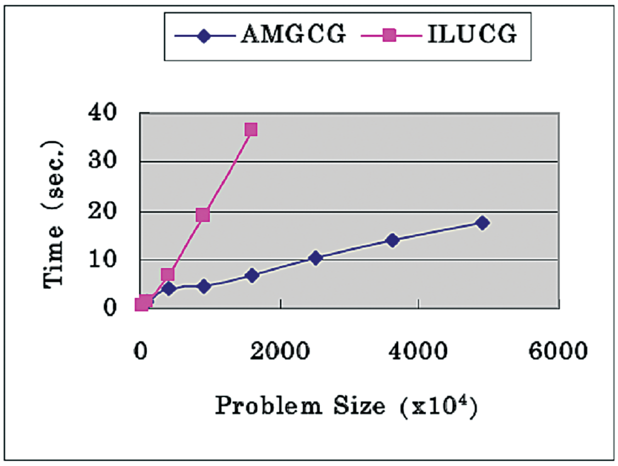

Fig. 4 Comparison of AMGCG and ILUCG.

sented results for a two-dimensional Poisson equation of dimension 49 million on 1,024 nodes of IBM Blue Gene/L in Fig. 4.

We also proposed the BiCR method, described in Fig. 5, which extends the CR method for symmetric problems to nonsymmetric problems, and GPBiCR, its application to a product type algorithm. In addition, we showed that a smoother convergence was achieved, as compared with BiCG. The convergence histories of some problems from Matrix Market (WATT1, WATT2, and petroleum engineering) using BiCR are shown in Fig. 6. These algorithms are also implemented on Lis 1.1.

```
set $\mathbf{x}_0$ is an initial guess, $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\beta_{-1} = 0$,
$\mathbf{r}_0^*$ is an arbitrary vector, such that $(\mathbf{r}_0^*, \mathbf{r}_0) \neq 0.$, e.g., $\mathbf{r}_0^* = \mathbf{r}_0$,
for $n = 0, \Lambda$ until $\|\mathbf{r}_n\| \leq \varepsilon \|\mathbf{r}_0\|$ do :
begin
    $\mathbf{p}_n = \mathbf{r}_n + \beta_{n-1}\mathbf{p}_{n-1}$,        $\mathbf{p}_n^* = \mathbf{r}_n^* + \beta_{n-1}\mathbf{p}_{n-1}^*$,
    $(\mathbf{A}\mathbf{p}_n = \mathbf{A}\mathbf{r}_n + \beta_{n-1}\mathbf{A}\mathbf{p}_{n-1})$,
    $\alpha_n = \dfrac{(\mathbf{r}_n^*, \mathbf{A}\,\mathbf{r}_n)}{(\mathbf{A}^{\mathrm{T}}\mathbf{p}_n^*, \mathbf{A}\mathbf{p}_n)}$,
    $\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{p}_n$,        $\mathbf{r}_{n+1}^* = \mathbf{r}_n^* - \alpha_n \mathbf{A}^{\mathrm{T}}\mathbf{p}_n^*$,
    $\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{p}_n$,
    $\beta_n = \dfrac{(\mathbf{r}_{n+1}^*, \mathbf{A}\,\mathbf{r}_{n+1})}{(\mathbf{r}_n^*, \mathbf{A}\,\mathbf{r}_n)}$,
end
```

Fig. 5 Algorithm of the BiCR method.

In material science fields, such as solid-state physics and quantum chemistry, large-scale simulations derived from density functional theory and first-principles calculation are often required. In these fields, there is a strong demand for efficient algorithms to solve large-scale eigenproblems, and cooperation with such fields is desirable in order to develop scalable eigensolvers. There are several methods to compute eigenvalues of large-scale sparse matrices, including the Lanczos method for symmetric problems, the Arnoldi method, its extension for nonsymmetric problems, the Davidson method originally proposed for quantum chemistry, and the Jacobi-Davidson method, a derivative of the Davidson method. We proposed a parallel implementation of the Jacobi-Davidson method, which resulted in research on performance and bottlenecks of iterative solvers and parallel implementations of the Jacobi-Davidson method. Based on observations, we proposed that the scalability of the conjugate gradient method for linear equations can improve the performance of eigensolvers in parallel environments, where the extreme eigenvalues of a generalized eigenproblem

$Ax = \lambda Bx$,

or an equivalent problem

$Bx = \mu Ax$, $\mu = 1/\lambda$

can be solved by reducing these problems to the calculation of the local maximum or local minimum of the Rayleigh quotients combined with appropriate preconditioners, such as the algebraic multigrid. We are currently verifying the effect of this procedure using Lis.

## 3. Fast Integral Transforms

In fields such as hydrodynamics and weather forecasting, we need to solve problems on a spherical surface, which derives the demands for high-performance fast integral transforms. In the present study, we have developed high-performance libraries, which have practical performance in real computing environments, such as fast Fourier transform.

The fast Fourier transform is an implementation the discrete Fourier transform and is used in many fields, ranging from large-scale scientific computing to image processing. Although many improvements have been proposed since the discovery of the FFT algorithm, recent rapid progress in processor architecture requires new FFT kernels.

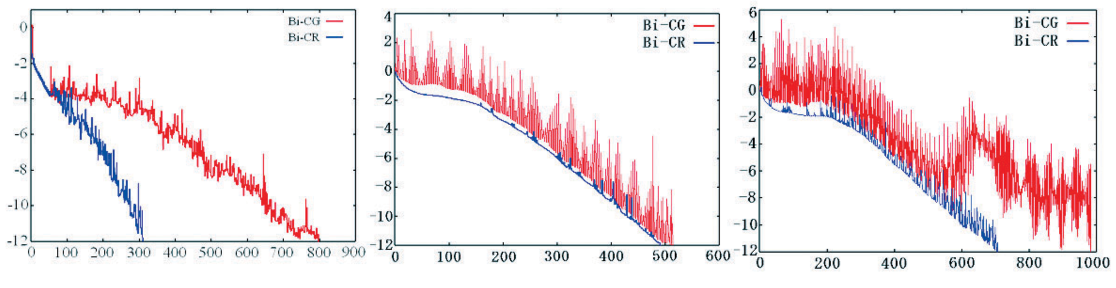

Fig. 6 Comparison of BiCG and BiCR.

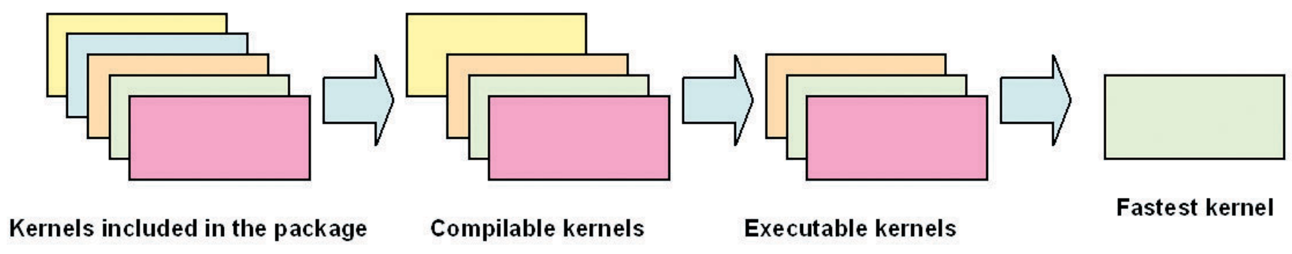

Fig. 7 Automatic performance tuning mechanism of FFTSS.

The existing algorithms, which propose efficient use of cache memory, adopt algorithms that do not require bit-reverse, such as Stockham FFT. In the SSI project, we overlapped the bit-reverse process with memory access to enable an in-place algorithm, and we have shown that the latency can be eliminated.

Processors like Intel's IA-64 and IBM's POWER, which have two multiply-add units and operate four floating point calculations per cycle, are becoming the mainstream. The multiply-add operation is a combination of multiply and add, while a single multiply or a single add operation also uses the multiply-add unit. This implies that we must combine as many multiplies and adds as possible in order to utilize the units efficiently. We proposed an 8-radix FFT kernel with the least number of multiply-add operations, which requires a smaller twiddle factor table and a smaller number of twiddle factors to be loaded. The result is reflected in the FFTSS library, which we developed as an FFT library for superscalar processors with automatic performance tuning mechanism, as shown in Fig. 7.

A program example and the performance of FFTSS for one-dimensional FFT, as compared with commercial libraries, and for OpenMP-based two-dimensional parallel FFT supporting padding, as compared with FFTW, are shown in Fig. 8 and 9–11, respectively.

In fields such as hydrodynamics and weather forecasting, we must solve problems on spherical surfaces, which requires high-performance fast integral transforms. In the present research, we developed an MPI version of FFTSS and implemented a vector processor version of FFT, overlapping a huge number of all-to-all communications and computation, as part of a joint study with the Earth Simulator Center. The FFTSS library showed the best performance of 16.3 TFLOPS with the double-precision FFT on 512 nodes of the Earth Simulator, which is 49.6% of peak.

```
max_threads = omp_get_num_procs();
fftss_plan_with_nthreads(max_threads);
plan = fftss_plan_dft_2d(nx, ny, py, vin, vout,
FFTSS_FORWARD, FFTSS_MEASURE);
{ /* Initialization of array */ }
for (nthreads = 1; nthreads <= max_threads; nthreads ++) {
fftss_plan_with_nthreads(nthreads);
t = fftss_get_wtime();
fftss_execute(plan);
t = fftss_get_wtime() - t;
printf("%lf sec. with %d threads.\n", nthreads, t);
}
```

Fig. 8 Program example of FFTSS.

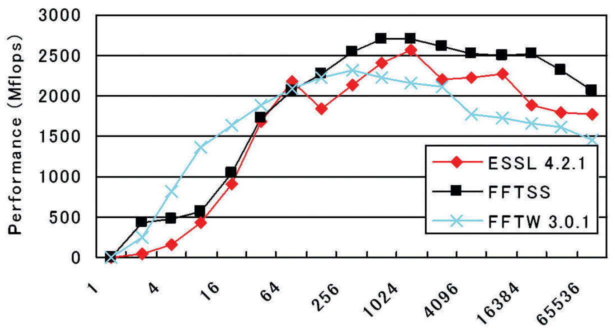

Fig. 9 Performance comparison of 1-dimensional FFT kernels on IBM POWER5.

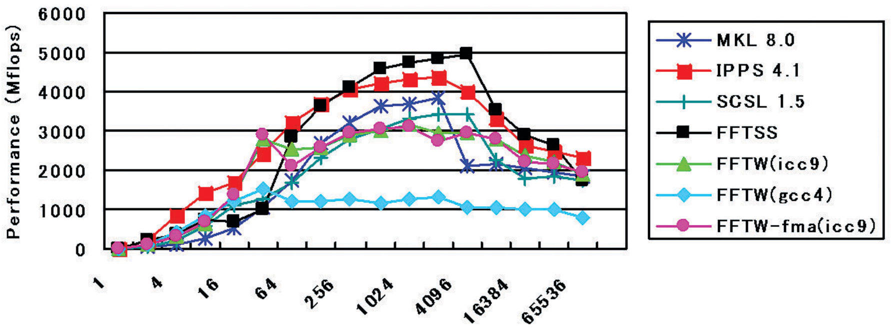

Fig.10 Performance comparison of 1-dimensional FFT kernels on Intel Itanium2.

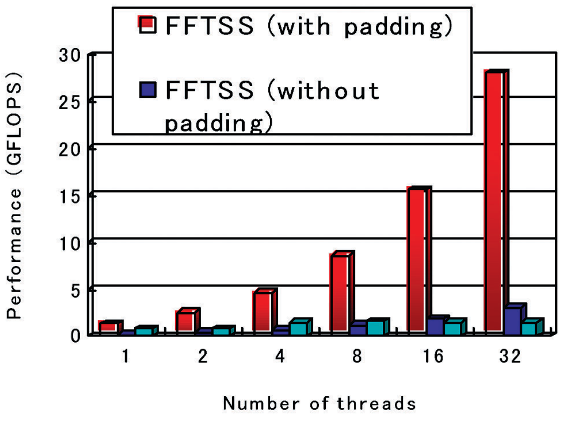

Fig.11 Performance of OpenMP-based two-dimensional parallel FFTSS on SGI Altix 3700.

## 4. Programming Environment

Libraries for matrix computation are indispensable to scientific computations, and several libraries have been proposed for their implementation. These libraries are provided with APIs to be used with other programs. For example, to solve a linear equation Ax = b, the user prepares a matrix A and a vector b in the format specified by the library and calls a function with specified arguments. In such cases, the program created by the user depends on the data structure and function calls of the specific library. In many cases, there are no compatibilities between the interfaces of the libraries, and the user must modify the program to use the routines provided by other libraries. These libraries are also applicable to cases with different preconditions or different computing precisions. There are also libraries for specific computing environments, which require libraries to be changed and codes to be rewritten. It is burdensome for the user to rewrite programs, and a more flexible method of using libraries is needed. To fulfill the demand, we have proposed an environment-independent matrix computation library SILC, a simple interface for library collections.

Apart from the former usage based on the specific interface of a library, SILC utilizes the features of the matrix computation libraries by sending three types of requests: (1) deposit of data to be input, (2) requests for computation by means of mathematical expressions in the form of text, and (3) fetch data to be output. The input data, such as matrices and vectors, are transferred to an independent memory space from the user program. The requests of computation by means of mathematical expressions are interpreted as appropriate function calls and are executed in the independent memory space. Finally, the results are returned to the memory space of the user program by request, as shown in Fig.12.

As an example, we present a C program in Fig.13, which calls a routine of LAPACK to solve a linear equation via the interface of SILC.

After making matrix A and vector b in LAPACK's format, this program calls the solver routine of LAPACK via the three routines SILC_PUT, SILC_EXEC, and SILC_GET provided by SILC.

For scientific computing, libraries based on OpenMP and MPI are used in various parallel computing environments. SILC buffers the difference of computing environments between the user program and computing environments and enables us to use the libraries in a language- and environment-independent manner. We have assumed the following four situations:

(A) sequential client and sequential server
(B) sequential client and shared-memory parallel server
(C) sequential client and distributed-memory parallel server
(D) distributed-memory parallel client and distributed-memory parallel server

The system configurations of SILC are shown in Fig.14.

```
silc_envelope_t A, b, x;
/* make matrix A and vector b */
SILC_PUT("A", &A);
SILC_PUT("b", &b);
SILC_EXEC("x = A \\ b"); /* solve the linear equation */
SILC_GET(&x, "x");
```

Fig.13 C program calling a routine of LAPACK to solve a linear equation via the interface of SILC.

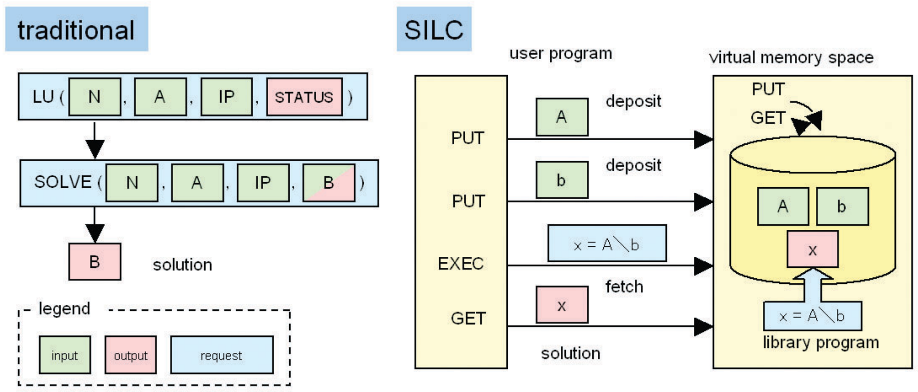

Fig.12 Concept of SILC, a simple interface for library collections.

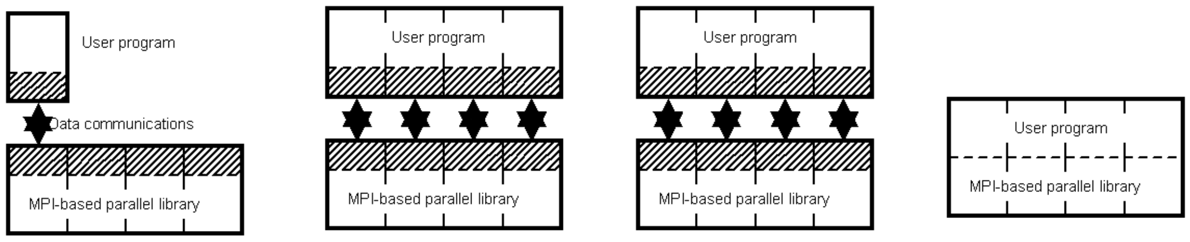
User program
Data communications
MPI-based parallel library
User program
MPI-based parallel library
User program
MPI-based parallel library
User program
MPI-based parallel library

Fig.14 System confugrations of SILC.

Libraries are used by linking to the user program. The user program transfers data and requests computations by connecting to a single process or multiple processes. The data transferred from the user program to the server is transformed into the requested data distribution manner and is retained in the server processes. The results returned to the user program are transformed again to the requested data distribution manner by the data redistribution mechanism.

Using remote distributed-memory parallel computing environments via the implemented system, we have observed better performance compared with the user program written in the traditional manner. The results of the solution of the initial value problem of a two-dimensional diffusion equation are shown using the finite difference method shown in Fig.15. Configuration is shown in Table. 4–5. We have used the conjugate gradient method without a preconditioner of Lis for the solution of linear equations, and have interpreted the request to solve the system into the function call of MPI-based Lis. Denoting the dimension by N and the number of iterations by I, the amount of communication is O(N) and the complexity is O(NI), which shows that we can solve the problems faster on the remote parallel server than on the local client when several iterations are required, even considering the communication cost.

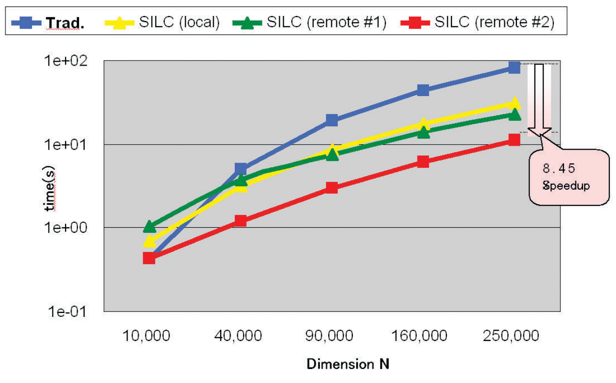

Fig.15 Performance of SILC on distributed-memory parallel computing environments.

To use control statements such as conditional branches and loops in SILC, the user must prepare a function, which includes the statements, and must make it callable from the library program. This makes it impossible to program arbitrary combinations of mathematical expressions and control statements. In the SSI project, we extended SILC to a scripting language with control statements, which interprets the user program and separates the mathematical expressions and the control statements and processes them using the existing SILC framework. By this extension, we are able to create more complicated programs, as shown in Fig.16.

Table 4 Configuration of computing environments.

| | User Program | SILC Server |
|---|---|---|
| traditional | Xeon4 (1 PE) | — |
| SILC (local) | Xeon4 (1 PE) | Xeon4 (4 PEs) |
| SILC (remote #1) | Xeon4 (1 PE) | Xeon8 (8 PEs) |
| SILC (remote #2) | Xeon4 (1 PE) | Altix (16 PEs) |

Table 5 Specification of machine architectures.

| Hosts | Specification |
|---|---|
| Xeon4 | IBM eServer xSeries 335 (dual Intel Xeon 2.8 GHz, 1.0 GB RAM) Red Hat Linux 8.0, LAM/MPI 7.0 |
| Xeon8 | Eight different nodes in the same PC cluster as Xeon4 |
| Atlix | Intel Itanium2 1.3 GHz x32, 32 GB memory, Red Hat Linux Advanced Server 2.1, SGI MPI 4.4 (MPT 1.9.1) |

```
# Conjugate Gradient Method
# make 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 linear equation Ax=b using the conjugate gradient 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
}

# save solution x
save "sol.mtx", x

# print number of iterations
message "number of iterations:"
pprint iter
```

Fig.16  An example of extension to scripting language.

# 大規模科学計算向け 汎用数値ソフトウェア基盤の開発

プロジェクト責任者
西田　晃　中央大学　理工学研究所

著者
西田　晃　中央大学　理工学研究所

本プロジェクトでは、従来それぞれの分野において別個に進められてきた並列アルゴリズムや実装に関する知見をもとに、大規模化が予想される今後の計算環境に対応したスケーラブルなソフトウェア基盤を整備することを目的として、反復解法、高速関数変換、及びその効果的な計算機上への実装手法を中心に、平成14年度より科学技術振興機構戦略的創造研究推進事業の一環として、多様な計算機環境を想定した開発を行っている。オブジェクト指向に基づくプログラミングインタフェースを採用し、複雑な機能を持つライブラリを容易に構築できるようにするとともに、スケーラビリティの観点から並列化に適したアルゴリズムを開発、実装し、高並列な環境での使用に耐えうるライブラリを実現している。本研究の成果はネットワークを通じて広く一般に配布し、フィードバックをもとにより汎用性の高いソフトウェアとしていく方針を採っており、平成17年9月よりソースコードを無償公開するとともに、ユーザの要望を反映した更新を適宜行なっている。平成18年度からは、地球シミュレータセンター共同プロジェクトの一環として、高並列なベクトル計算機環境への最適化を実施し、その成果をライブラリとして公開した。

キーワード：高性能計算, 並列アルゴリズム, スケーラビリティ, オブジェクト指向, ネットワーク配布