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

Project Representative

Akira Nishida 21st Century COE Program, Chuo University

Authors

Akira Nishida [*1, 2], Akira Nukada [*2, 3], Hisashi Kotakemori [*2, 3] and Tamito Kajiyama [*2, 3]

[*1] 21st Century COE Program, Chuo University

[*2] Core Research for Evolutional Science and Technology (CREST) Program, Japan Science and Technology Agency

[*3] Department of Computer Science, the University of Tokyo

The object of the project, which started in 2002 as a study named the Scalable Software Infrastructure (SSI) project of Core Research for Evolutional Science and Technology at Japan Science and Technology, is to develop basic libraries for large scale scientific simulations which have previously been developed separately in each field and to integrate them into a scalable software infrastructure. The targets include iterative solvers for linear systems, fast integral transforms, and computing environment-independent frameworks for scientific computing. The project adopted an object-oriented programming interface to enable users to build complex libraries and to develop highly parallel algorithms from a scalability viewpoint, as has been achieved at various supercomputing sites around the world. Our results have been freely available since September 2005 through networks for researchers in various fields, and their feedback has been reflected in the improved functionality and usability of the software. Since 2006, as a member of the FY2006 Earth Simulator collaboration project, we have begun tuning our software for highly parallel vector computing environments.

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

1. Overview

Recent progress in science and technology has made numerical simulation an important approach for studies in various fields. The object of the Scalable Software Infrastructure (SSI) project, funded by the Japan Science and Technology Agency since 2002, is to develop basic libraries of solutions and algorithms required for large scale scientific simulations which have been previously developed separately in each field, and to integrate them into a scalable software infrastructure. The components include a scalable iterative solvers library for linear systems having multiple solvers; preconditioners; matrix storage formats that are flexibly combinable; a fast Fourier transform library, which outperforms some vendor-provided FFT libraries; and a language- and computing environment-independent matrix computation framework. We report below our tuning efforts in each field.

2. Fast Fourier Transform Library

We have developed a high performance two-dimensional (2-D) Fast Fourier Transform (FFT) for the Earth Simulator. The FFT is one of the most important computations in many fields of the simulations. Therefore, its efficient implementation on computers is always required.

The Earth Simulator is a distributed-memory, massively-parallel vector computer. In the case of the computation of the FFT algorithm on distributed-memory systems, the data transfer between the nodes often occupies a large percentage of the total execution time. This is very serious problem, especially for PC clusters with narrow interconnects. The Earth Simulator has very high speed interconnects, however, and reduces the ratio occupied by data transfer to less than half. With it, the time required for the data transfer can be hidden by overlapping the communication and the computation. We used a transpose split algorithm [1] to overlap the communication of the FFT. The data transfer is divided into multiple stages, and the computation for one stage is overlapped with the communication for the next stage. Figure 1 shows the time lines of the algorithm in the case of four stages. For efficient data transfer, the data to be sent must be stored in a contiguous address, and for this reason the pack and unpack operations are required. Those operations are simple memory copy operations; therefore, they can be executed in one step of the computation of the FFT.

A typical implementation of parallel FFT uses MPI_Alltoall API for data transfer between the nodes. To

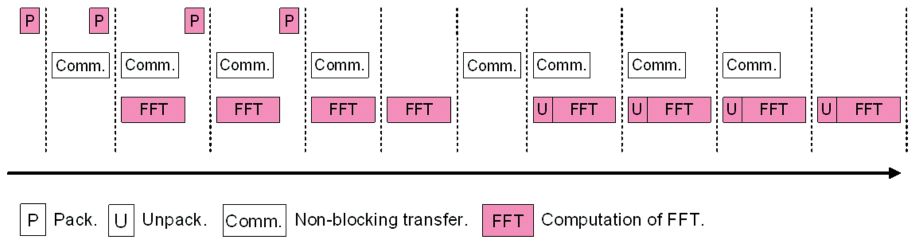

Fig. 1 The computation of the FFT using overlapped communication. (4 stages)

Table 1 The data transfer rates (GB/s) for all-to-all communications.

	16	32	64	128	256	512
MPI_Alltoall	7.06	7.71	7.80	7.33	6.97	6.17
MPI_Isend	3.95	3.28	3.07	2.72	2.36	2.20
MPI_Put	10.60	11.00	11.19	10.18	11.42	11.43

overlap the communication with the computation, we used MPI_Put, a one-side transfer API provided by MPI-2. Table 1 shows the data transfer rates for all-to-all communications with MPI_Alltoall, with MPI_Put, and with MPI_Isend. The size of the message sent to each node is 1MB. The result shows that the implementation with MPI_Put is the best solution for all-to-all communication on the Earth Simulator.

Each node has eight vector processors and a Remote Control Unit (RCU), which actually executes non-blocking data transfers between the nodes. Therefore, we could use all vector processors for the computation during the data transfer. The main memory of each node is shared with the vector processors. In order to avoid any additional memory copies, the computation within the node is parallelized using OpenMP--that is, the hybrid parallel programming model of OpenMP + MPI is used in our implementation.

The best number of the stages should be selected depending on the size of input data and the number of nodes. As shown in Fig. 1, the communication in one stage is not overlapped, and therefore the number of stages should be large. On the other hand, the overhead of the communication becomes large if the number of stages is large.

Figure 2 shows the performance of $2^{14} \times 2^{14}$ 2-D FFT with 1~16 nodes. The overlapping improved the performance by a maximum of 50%.

Our implementation also includes many additional optimizations such as a loop exchange to enlarge the iteration counts for efficient vector processing, insertion of padding to avoid bank conflicts, and the elimination of memory copies, to mention only a few. As the result, the performance of 16.3 Tflops was achieved in computing $2^{19} \times 2^{18}$ 2-D FFT with 512 nodes (49.6% of peak).

3. Iterative Solvers Library Lis

To solve the linear equations for large sparse systems, we are developing a library called Lis (Library of Iterative Solvers for linear systems). It is written in C and Fortran 90, and consists of the serial, the OpenMP, and the OpenMP+MPI hybrid version. Lis supports the following features:

- 12 iterative solvers, 8 preconditioners, and their combinations
- 11 matrix storage formats

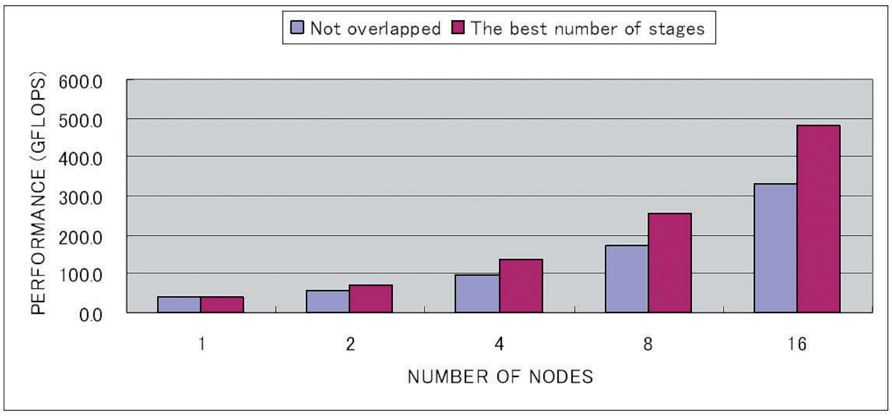

Fig. 2 The performance improvement by the overlapping.

- a common interface for both the serial and the parallel processing which enables users to switch seamlessly from the serial to the parallel computing environments
- quadruple precision operations

For iterative solvers, Lis supports stationary (e.g., SOR) and nonstationary (e.g., GPBiCG) iterative solvers for real general matrices (Table 2). As for preconditioners, it includes the Jacobi; Incomplete LU factorization (ILU); an algebraic multigrid based on smoothed aggregation (SA-AMG), the I+S for stationary iterative solvers; the Hybrid, which combines iterative solvers like SOR; SAINV, which approximates A^{-1} based on the A-diagonalization; and the Crout version of the ILU preconditioner, which gives us more stable factorization than the ILU (Table 3). It also supports multiple matrix storage formats like CRS (Table 4). We can easily combine the solvers, the preconditioners, and the matrix storage formats.

The iterative solvers based on the Krylov subspace method are implemented by matrix-vector products, vector inner products, and axpy type vector-vector operations. To port them on the Earth Simulator, we supported vectorization and the JDS (Jagged Diagonal Storage) matrix storage format. Since the standard format for Lis is CRS (Compressed Row Storage), we needed to transform the data into the JDS format. The vectorization of the CRS-to-JDS transformation is implemented by the loop interchange shown in Fig. 3.

Furthermore, to overlap the computation and communication of the matrix vector product y = Ax, we partitioned A

Table 2 Iterative Solvers

unstationary	CG
	BiCG
	CGS
	BiCGSTAB
	BiCGSTAB(1)
	GPBiCG
	TFQMR
	Orthomin(k)
	GMRES(k)
stationary	Jacobi
	Gauss-Seidel
	SOR

Table 3 Preconditioners

Jacobi
SSOR
ILU(k)
I+S
SA-AMG
Hybrid
SAINV
Crout ILU

Table 4 Matrix Storage Formats

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

```
for(i=is;i<ie;i++)    {
  js = Ain->ptr[perm[i]];
  je = Ain->ptr[perm[i]+1];
  for(j=js;j<je;j++) {
    l = ptr[my_rank*(maxnzr+1) + j-js]
          +i-is;
    value[l]     = Ain->value[j];
    index[l]     = Ain->index[j];
  }
}
```

(a)Previous Transformation

```
for(j=0;j<maxnzr;j++) {
  js = ptr[my_rank*(maxnzr+1) + j];
  je = ptr[my_rank*(maxnzr+1) + j+1];
  #pragma cdir nodep
  for(i=js;i<je;i++) {
    l = Ain->ptr[perm[is+i-js]] + j;
    value[i]     = Ain->value[l];
    index[i]     = Ain->index[l];
    }
}
```

(b)Transformation for vector machines

Fig. 3 Vectorization of CRS-to-JDS matrix storage formats.

into D-L-U (D is the diagonal part of A, and -L and -U are the lower and upper triangular parts of A, respectively). We can overlap 2) and 3) in the following four steps:
1) Communication of the elements of x, required for the computation of (D-L)x
2) Communication of the elements of x, required for the computation of Ux
3) Computation of y = (D–L)x
4) Computation of y –= Ux

4. Matrix Computation Framework

The authors have been developing an easy-to-use matrix computation framework named Simple Interface for Library Collections (SILC) [2], which 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 making calls for library functions based directly on a library-specific application programming interface (API), user programs for SILC utilize matrix computation libraries in the following three steps. First, the user programs deposit data such as matrices and vectors into a SILC server. Next, the user programs make 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 SILC server independently of the user programs. Finally, the user programs fetch the results of the computation (if necessary) from the server.

With the aim of providing support for the Earth Simulator in the SILC framework, we pursued the following three research directions:

- To introduce a system configuration for computing environments based on batch processing systems.
- To implement a SILC server with matrix computation libraries linked statically to the server.
- To seek improvements on high-performance matrix computations in vector machines by means of vectorization techniques.

The current implementation of SILC is based on a client-server architecture. Figure 4 (a) and (b) show two configurations of the implemented SILC system. Figure 4 (a) depicts a SILC system that is composed of a sequential user program and an MPI-based parallel SILC server running on four MPI processes, while Fig. 4 (b) shows another configuration in which both a user program and a SILC server are MPI-based parallel programs running on four MPI processes. The shaded parts in the figure indicate the components that SILC provides, which are responsible for data communications between the user program and the SILC server as well as for managing computation requests to be made by the user program. On the other hand, the client-server architecture is not applicable in some computing environments (including the Earth Simulator) mainly because of their being managed by batch processing systems such as the Network Queuing System (NQS). Therefore, we introduced a third system configuration, shown in Fig. 4 (c), which is not based on the client-server model. In this system configuration, an MPI-based user program is statically linked with SILC's components as well as the matrix computation libraries to be used. The implementation of the third system configuration is in progress.

We also implemented a SILC server that did not rely on the functionalities of multithreading and dynamic linking available in many operating systems. In some restrictive computing environments (including the Earth Simulator), user programs must be single-threaded programs, and the libraries used in the user programs must be statically linked. On the other hand, SILC servers are originally multithreaded programs, and libraries are dynamically linked with the servers in the form of plug-in modules such that users can specify the libraries they want to utilize at run time. Therefore, we developed a single-threaded SILC server in which all libraries were statically linked while the functionality of plug-in modules was kept unchanged. The implemented SILC server was tested on NEC SX-6i, a vector computer without the functionalities of multithreading and dynamic linking.

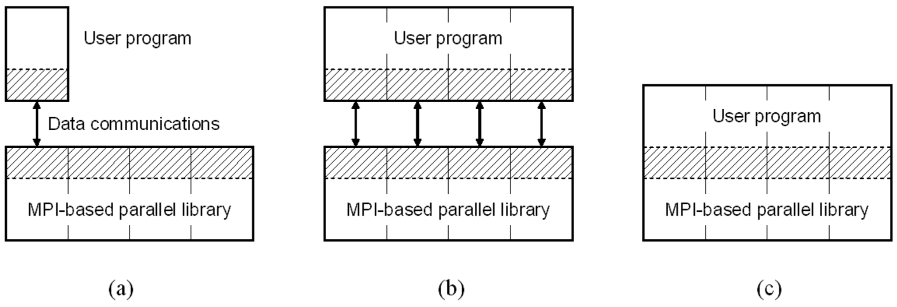
User program
Data communications
MPI-based parallel library
User program
MPI-based parallel library
User program
MPI-based parallel library
(a)
(b)
(c)

Fig. 4 Three system configurations of SILC. Configurations (a) and (b) are based on a client-server architecture, while Configuration (c) is for restrictive computing environments in which a client-server architecture is not applicable.

Moreover, in order to provide better support for vector computers, sparse matrices in the Jagged Diagonal Storage (JDS) format [3] were newly supported, and the single-threaded SILC server was vectorized by vectorization directives inserted into the source code of the server.

We conducted preliminary experiments to examine the effectiveness of the aforementioned extensions, using the user program for SILC shown in Fig. 5. The user program solves a system of linear equations $Ax = b$ with the Conjugate Gradient (CG) method [3] written in SILC's mathematical expressions. Matrix A in the user program was originally stored in the Compressed Row Storage (CRS) format [3]; the use of this matrix storage format, however, resulted in poor performance in NEC SX-6i due to the facts that (i) the most computationally intensive part in the CG method is a matrix-vector product; and (ii) when the CRS format is in use, the innermost loop in the matrix-vector product over an array of non-zero elements tends to be quite short compared with the dimension of A. Therefore, we changed matrix storage formats from CRS to JDS so that the innermost loop in the matrix-vector product was likely to be as long as the dimension. It is worth noting that no modification was required in the code of the CG method with regard to the change in matrix storage formats since the code written in SILC's mathematical expressions was independent of matrix storage formats.

```
silc_envelope_t  object;
double nrm2;

/*  A : matrix,  b : vector  */

SILC_EXEC("rho_old  = 1.0");
SILC_EXEC("n  = length(b)");
SILC_EXEC("p  = zeros(n , 1)");
SILC_EXEC("x  = zeros(n , 1)");
SILC_EXEC("r  = b");
SILC_EXEC("bnrm2 = 1.0 / norm2(b)");
for (i = 1; i <=  maxiter ; i++) {
  SILC_EXEC("rho  = r' * r");
  SILC_EXEC("beta  = rho / rho_old ");
  SILC_EXEC("p  = r + beta * p");
  SILC_EXEC("q  = A * p");
  SILC_EXEC("alpha  = rho / (p' * q)");
  SILC_EXEC("r  = r - alpha * q");
  SILC_EXEC("nrm2 = norm2(r) * bnrm2");
  SILC_EXEC("x  = x + alpha * p");
  /*  convergence check  */
  object.v  = &nrm2;
  SILC_GET(&object , "nrm2");
  if (nrm2 <= EPSILON)
    break;
  SILC_EXEC("rho_old  = rho");
}

/*  x : solution  */
```

Fig. 5 The Conjugate Gradient (CG) method written in SILC's mathematical expressions.

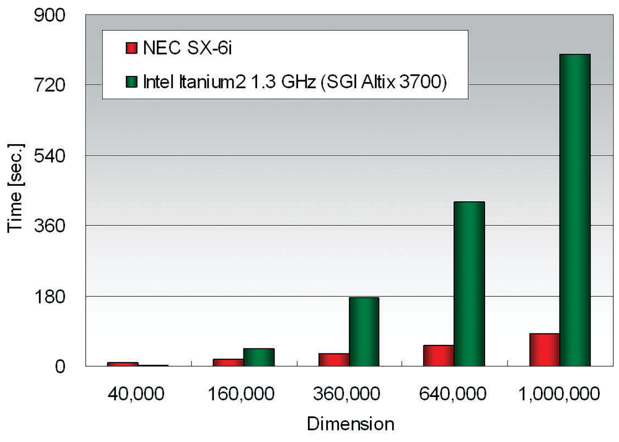

Fig. 6 Performance results of the Conjugate Gradient (CG) method for solving a system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$ using the Jagged Diagonal Storage (JDS) format to store a coefficient matrix A.

Figure 6 shows the experimental results with the dimension of matrix A on the horizontal axis and the execution time in seconds on the vertical axis, where A is a sparse matrix resulting from a regular five-point difference approximation of a two-dimensional Laplacian equation. We compared the following two cases: (A) the performance of the user program with a SILC server on NEC SX-6i, and (B) that of the same program with another SILC server on a compute node (having an Intel Itanium 2 1.3 GHz processor) of SGI Altix 3700. The user program in the case of (A) achieved better performance than the same program in the case of (B), especially when the dimension was large, thanks to the use of the JDS format for storing A.

References

[1] Calvin, C.: Implementation of parallel FFT algorithm on distributed memory machines with a minimum overhead of communication, Parallel Computing, Vol.22, pp.1255–1279 (1996).

[2] T. Kajiyama, A. Nukada, H. Hasegawa, R. Suda, and A. Nishida. SILC: A Flexible and Environment Independent Interface for Matrix Computation Libraries. In Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005), Lecture Notes in Computer Science 3911, pp.928–935, 2006.

[3] R. Barrett *et al*. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.

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21世紀COEプログラム・JST CREST

著者
西田　　晃　　中央大学21世紀COEプログラム・JST CREST
額田　　彰　　JST CREST・東京大学大学院　情報理工学系研究科
小武守　恒　　JST CREST・東京大学大学院　情報理工学系研究科
梶山　民人　　JST CREST・東京大学大学院　情報理工学系研究科

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指し、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対象に、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、平成14年度より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。また、本研究の成果はネットワークを通じて広く一般の研究者に配布し、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含むソフトウェア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年度は、これらの成果を背景に、地球シミュレータ共同プロジェクトに参加し、高並列なベクトル計算機環境への最適化を実施した。

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