International Conference on Computational Methods (ICCM 2007)
April 4-6, 2007, Hiroshima, Japan

Numerical Simulations in the SILC Matrix Computation Framework

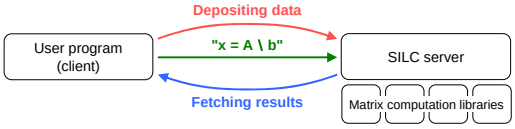
Tamito KAJIYAMA (JST / University of Tokyo, Japan)
Akira NUKADA (JST / University of Tokyo, Japan)
Reiji SUDA (University of Tokyo / JST, Japan)
Hidehiko HASEGAWA (University of Tsukuba, Japan)
Akira NISHIDA (Chuo University / JST, Japan)

Outline

- The SILC matrix computation framework
  - Easy-to-use interface for matrix computation libraries
- Proposal of two application modes for SILC
- Two applications in numerical simulations
  - Cloth simulation in C
  - MPS-based simulation in Java
- Experimental results
- Concluding remarks

Overview of the SILC framework

- **S**imple **I**nterface for **L**ibrary **C**ollections
  - Independent of libraries, environments & languages
  - Easy to use
- Three steps to use libraries
  - **Depositing data** (matrices, vectors, etc.) to a server
  - **Making requests for computation** by means of mathematical expressions
  - **Fetching the results of computation** if necessary

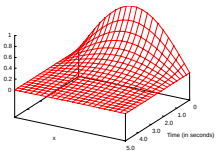


Example: Using SILC in C

```
silc_envelope_t A, C, x;

/* create matrices A, C and vector x_0 */

SILC_PUT("A", &A);
SILC_PUT("C", &C);
SILC_PUT("x", &x); /* x_0 */
for (k = 1; k <= n_steps; k++)
{
    SILC_EXEC("b = C * x");
    SILC_EXEC("x = A \\ b");
    SILC_GET(&x, "x"); /* x_k */

    /* output solution x_k at time t_k */

}
```

Solve the initial value problem of 1D diffusion equation below using the Crank-Nicolson method:

$$\frac{\partial \varphi}{\partial t} = \frac{\partial^2 \varphi}{\partial x^2} \quad (t \geq 0, 0 \leq x \leq \pi)$$
$$\varphi = \sin x \quad (t = 0, 0 \leq x \leq \pi)$$
$$\varphi = 0 \quad (t > 0, x = 0)$$
$$\varphi = 0 \quad (t > 0, x = \pi)$$



Functionalities of SILC

- Data structures for matrix computations
  - Matrices (dense, band, sparse), vectors, etc.
- Math operators, functions, and subscript
  - 2-norm of vector $\boldsymbol{x}$: `sqrt(x' * x)`
  - 5×5 submatrix of A: `A[1:5, 1:5]`
- No loops and conditional branching
  - These are realized with the languages you use to write user programs for SILC

Main characteristics of SILC

- Independence from programming languages
  - User programs for SILC in your favorite languages
- Independence from libraries and environments
  - Using alternative libraries and environments requires no modification in user programs
  - Flexible combinations of client & server environments

User program (client)	SILC server
Sequential	Sequential
Sequential	Shared-memory parallel (OpenMP)
Sequential	Distributed parallel (MPI)
Distributed parallel (MPI)	Distributed parallel (MPI)

Proposal of two application modes

- **Limited application mode**
  - Use SILC only in the most time-consuming, computationally intensive part of a program
- **Comprehensive application mode**
  - Move all relevant data to a SILC server, and implement overall simulations using SILC's mathematical expressions

Abbreviations:
- **LTD** for the limited application mode
- **CMP** for the comprehensive application mode

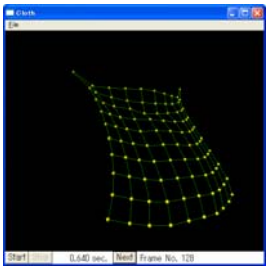
Limited application mode

Use SILC only in the most time-consuming, computationally intensive part of a program

- Pros
  - Easy to apply (especially to existing user programs)
- Cons
  - Smaller data size due to a limited amount of main memory in client environments
  - Frequent data communications (larger overheads)

Application #1

- Time-dependent simulation of cloth motion
  - Mass-spring model
  - Implicit Euler method
  - Solving a sparse linear system is necessary for each time step
- Original code
  - Sequential program in C
  - Linear solver: CG method
  - Visualization: OpenGL



Original code

For each time step:

1. Calculate force $\boldsymbol{f}$ and its derivatives $\partial \boldsymbol{f} / \partial \boldsymbol{x}$ and $\partial \boldsymbol{f} / \partial \boldsymbol{v}$ (Jacobian matrices).

2. Solve a linear system $A \Delta \boldsymbol{v} = \boldsymbol{b}$, where

$$A = \left\{ M - \Delta t^2 \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} - \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} \right\}$$

$$\boldsymbol{b} = \left\{ \boldsymbol{f}_0 + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \boldsymbol{v}_0 \right\} \Delta t$$

3. Update particle motion.

$$\boldsymbol{v} = \boldsymbol{v}_0 + \Delta \boldsymbol{v}$$
$$\boldsymbol{x} = \boldsymbol{x}_0 + \Delta t \, \boldsymbol{v}$$

New code in the LTD mode

Original code using Lis*

```
LIS_MATRIX A; LIS_VECTOR b, dv;

for (k = 1; k <= n_steps; k++)
{
  /* 1. Calculate f, ∂f/∂x and ∂f/∂v */
     :
  /* 2. Solve AΔv = b */
  lis_solve(A, b, dv, /* Δv */
            lis_params,
            lis_options,
            lis_status);
  /* 3. Update particle motion */
     :
}
```

New code using SILC

```
silc_envelope_t A, b, dv;

for (k = 1; k <= n_steps; k++)
{
  /* 1. Calculate 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"); /* Δv */
  /* 3. Update particle motion */
     :
}
```

* An iterative solvers library written in C.

Comprehensive application mode

Move all relevant data to a SILC server, and implement overall simulations using SILC's mathematical expressions

- Pros
  - User programs are free from massive data
  - Reduced data communications (smaller overheads)
- Cons
  - Existing user programs may require a major rewrite

Computations in Application #1

- Force

$$\boldsymbol{f}_i = \sum_{j \in P_i} (\boldsymbol{f}_{ij} + \boldsymbol{d}_{ij})$$

$$\boldsymbol{f}_{ij} = b_k \left(|\boldsymbol{x}_j - \boldsymbol{x}_i| - l_k \right) \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|} \quad \text{(spring force)}$$

$$\boldsymbol{d}_{ij} = -h_k (\boldsymbol{v}_i - \boldsymbol{v}_j) \quad \text{(damping)}$$

- Derivatives (Jacobian matrices)

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

Elements of the derivatives

- Off-diagonal blocks (3×3 submatrices)

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

- Diagonal blocks (3×3 submatrices)

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

- In SILC, write coarse-grain matrix computations
  - Computing the blocks one after another is not a good idea (too fine-grain to parallelize)

```
/* 1. Calculate f, ∂f/∂x and ∂f/∂v */
SILC_EXEC("p = Y_L * x - Y_R * x");
SILC_EXEC("P = sparse(P_row, P_col, p, 3*s, s)");
SILC_EXEC("z = sqrt(diagvec(P' * P))");
SILC_EXEC("fij = P * (K_stiff *@ (z - L) /@ z)");
SILC_EXEC("q = Y_L * v - Y_R * v");
SILC_EXEC("Q = sparse(P_row, P_col, q, 3*s, s)");
SILC_EXEC("dij = Q * K_damp");
SILC_EXEC("f = Sum_f * (fij + dij) - M * g");

SILC_EXEC("zhat = ones(s, 1) /@ z; Pzhat = P * 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("tmp = sqrt(zhat *@ K_stiff *@ L)");
SILC_EXEC("C1 = diag(X * sqrt(K_stiff))");
SILC_EXEC("C2 = diag(X * tmp); D = diag(tmp)");
SILC_EXEC("A1 = Y_LT * C1 - Y_RT * C1; T1 = -A1 * A1'");
SILC_EXEC("A2 = Y_LT * C2 - Y_RT * C2; T2 = -A2 * A2'");
SILC_EXEC("A3 = U_L * D - U_R * D; T3 = -A3 * A3'");
SILC_EXEC("DfDx = T1 - T2 + T3");

/* 2. 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");

/* 3. Update particle motion */
SILC_EXEC("v += dv *@ fixed");
SILC_EXEC("x += dt * v");
```

Code in the CMP mode: All computations done by **coarse-grain matrix computations** so as to be parallelized by a parallel SILC server.

Solving $A\Delta\boldsymbol{v} = \boldsymbol{b}$ is done in the same way as the LTD mode.

Experimental results

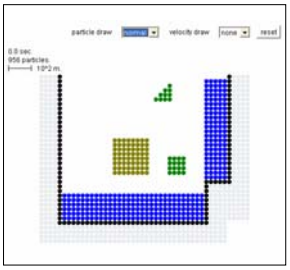
- 10^4 particles (7.998×10^8 springs), 100 time steps
- User programs on the same PC
- SILC servers on the same PC and on SGI Altix 3700
- LTD mode achieved ×2 speedup using a parallel server
- CMP mode was slow due to sparse matrix multiplication

User program	Original	LTD mode		CMP mode
SILC server	—	PC	Altix (8 threads)	Altix (8 threads)
Execution time [sec]	118.6	116.0	58.5	Not available
Speedup	1	×1.02 faster	×2.03 faster	×60 slower

PC: Intel Pentium 4 3.40 GHz, 1 GB RAM, Windows XP SP2
SGI Altix 3700: Intel Itanium 2 1.3 GHz × 32, 32 GB RAM (cc-NUMA), RH Linux AS 2.1

Application #2

- Time-dependent simulation of interaction among fluid, rigid and elastic bodies
  - Based on the Moving Particle Semi-implicit (MPS) method
  - Solving a sparse linear system is necessary for each time step
- Original code
  - Multithreaded program in pure Java
  - Solver: ICCG method



Original code

For each time step (k = 1, 2, 3, …):

1. Calculate source terms and particle motion.

$$\boldsymbol{u}^* = \boldsymbol{u}^k + \Delta t \left[\nu \nabla^2 \boldsymbol{u} + \boldsymbol{g} \right]^k$$
$$\boldsymbol{r}^* = \boldsymbol{r}^k + \Delta t \, \boldsymbol{u}^*$$

2. Solve pressure Poisson equation.

$$\nabla^2 P^{k+1} = \frac{\rho}{\Delta t^2} \frac{n^* - n'}{n^0}$$

3. Calculate pressure gradient terms.

$$\boldsymbol{u}' = -\frac{\Delta t}{\rho} \nabla P^{k+1}$$

4. Correct particle motion.

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^* + \boldsymbol{u}'$$
$$\boldsymbol{r}^{k+1} = \boldsymbol{r}^* + \Delta t \, \boldsymbol{u}'$$

Explicitly calculated

Implicitly calculated

Solving this equation by the ICCG method takes more than 60% of the original code's execution time.

Apply the LTD mode

Modified code in the LTD mode

- Added initialization and finalization
- Replaced the Java implementation of the ICCG method by a call for a linear solver via SILC

```
SILC client = new SILC();
client.INIT(host, port);
```

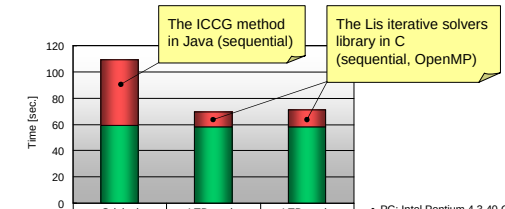
```
CRS mat = new CRS(mat_size, mat_size, a);
ColumnVector vec = new ColumnVector(mat_size, b);
ColumnVector sol = new ColumnVector();
client.PUT("A", mat);
client.PUT("b", vec);
client.EXEC("x = A \\ b");
client.GET(sol, "x");
```

CRS and ColumnVector convert user-defined data structures into SILC's ones.

```
client.FINALIZE();
```

Experimental results

- 956 particles (fluid: 296, rigid: 26, elastic: 64)
- Execution time of the first 300 time steps



	Original (PC)	LTD mode (PC)	LTD mode (Altix, 2 threads)
Solve $Ax=b$	**50.23**	**11.15**	**12.91**
Other	59.09	58.30	58.26

- PC: Intel Pentium 4 3.40 GHz, 1 GB RAM, Windows XP SP2
- SGI Altix 3700: Intel Itanium 2 1.3 GHz × 32, 32 GB RAM (cc-NUMA), RH Linux AS 2.1

Discussion

- What are these case studies intended for?
- Review: Basic ideas of SILC
  - User programs make computation requests using mathematical expressions
  - Parallel SILC servers fulfill the computation requests efficiently
  - Performance gain by parallel computation at the cost of data communications

Discussion (cont'd)

Our question: How should SILC be designed?	From users' viewpoint: How should users employ SILC?
• SILC servers should be able to properly parallelize computation requests • SILC's client API should be designed so that only parallelizable computation requests can be made	• Users should make computation requests so that SILC servers can parallelize them

Feedback from the present case studies

Summary

- Two application modes for SILC
  - Limited application mode
    - Ease of use
  - Comprehensive application mode
    - Data-free user programs
    - Reduced data communications
- Applied to cloth simulation in C and MPS-based simulation in Java
  - Feedback on design issues of SILC

Final remarks

- Future work
  - MPS-based simulation in the CMP mode
  - Analysis of various implementation choices in the CMP mode
  - More SILC applications in this & other areas
- Acknowledgment
  - Koji KOJIMA (University of Tokyo) for kindly providing the original Java code of the MPS-based simulation

Advertisement

- SILC v1.2 is freely available at

  http://ssi.is.s.u-tokyo.ac.jp/silc/

  – Source (Unix/Linux, Mac OS X, Windows)
  – Precompiled binary package for Windows
  – Documentation, sample programs