International Workshop on Automatic Performance Tuning (iWAPT 2007)
September 20–21, 2007, University of Tokyo, Tokyo, Japan

Toward Automatic Performance Tuning for 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 (University of Tokyo / JST, Japan)

Outline

- The SILC matrix computation framework
  - Easy-to-use interface for matrix computation libraries
- Automatic performance tuning in SILC
  - Performance modeling
  - Related work
- Experimental results
- Concluding remarks

Numerical simulations

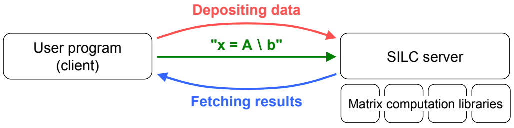
- A common feature: Appearance of PDEs
  - Discretization of PDEs results in linear systems, which are solved by linear solvers
- Some simulations require only one linear system to be solved
  - E.g. Steady-state simulations in linear problems
- Others require many linear systems to be solved repeatedly
  - E.g. Steady-state simulations in nonlinear problems, time-dependent simulations

Matrix computation libraries

- Key components of numerical simulations
- Problem: Using libraries is not easy
  - Many libraries having different APIs
  - Diversity of computing environments
  - Interoperability issues of various programming languages
- Solution: The SILC framework

Simple **I**nterface for **L**ibrary **C**ollections

- Benefits
  - 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, u;

/* create matrices A, C and vector u0 */

SILC_PUT("A", &A);
SILC_PUT("C", &C);
SILC_PUT("u", &u); /* u0 */
for (k = 1; k <= n_steps; k++)
{
   SILC_EXEC("b = C * u");
   SILC_EXEC(“u = A \\ b");
   SILC_GET(&u, "u"); /* uk */

   /* output solution uk at time tk */

}
```

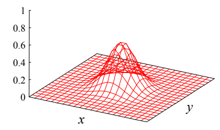
Solve the initial value problem of 2D diffusion equation below using the Crank-Nicolson method:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \quad x, y \in (0, 1),$$

$$u(x, y, t) = 0, \quad t > 0,$$

$$u(x, y, 0) = \begin{cases} 1 & \text{if } x, y \in (0.4, 0.6) \\ 0 & \text{otherwise} \end{cases}$$

$u(x, y, 0.004)$



Functionalities of SILC

- Data structures for matrix computations
  - Matrices (dense, band, sparse), vectors, etc.
- Math operators, functions, and subscript
  - 2-norm of vector ***x***: `sqrt(x' * x)`
  - 5×5 submatrix of *A*: `A[1:5, k:k+4]`
- No loops and conditional branching
  - These are realized with the languages used 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)

Cost in SILC: Communication time

- Likely to be smaller than computation time

Matrix (Solver)	Time for solving a linear system $Ax = b$	Time for depositing A and b and fetching x
Dense (LU decomposition)	$O(N^3)$	$O(N^2)$
Sparse (CG method)	$O(CZ)$	$O(Z)$

(N: dimension, C: iteration count, Z: number of non-zero elements)

- Possible speedups by parallel computation even at the cost of data communications

Automatic performance tuning (APT)

- SILC needs APT
  - To achieve as much speedup as possible in order to relatively minimize the cost of data communications
  - Using all available processors (or threads) is not always optimal
- SILC is an ideal framework in which APT is implemented
  - SILC servers can carry out various types of APT independently of user programs

Purposes of the present research

- Performance modeling of time-dependent simulations in SILC
- Outline of an APT mechanism for SILC

- Assumptions
  - A sequential user program, running with
  - A shared-memory parallel SILC server

Performance modeling

- The execution time (in seconds) of a user program is modeled as a function of p (the number of threads) as follows:

$$f(p) = a/p + bp + c$$

a/p : time for parallelized computations

bp : parallelization overhead

c : time for sequential computations

$(a, b, c > 0)$

The least squares method

- Suppose we have measured the execution time of the user program with n different numbers of threads (e.g., $p_i = 2^{i-1}$, $i = 1, \ldots, n$)

Number of threads	p_1	p_2	…	p_n
Execution time [sec.]	f_1	f_2	…	f_n

- By using the least squares method, we can find a, b, and c that minimize

$$q = \sum_{i=1}^{n} \{f_i - f(p_i)\}^2$$

The optimal number of threads p_{opt}

- With a performance model $f(p)$, we can predict the optimal number of threads p_{opt} that leads to the minimum execution time
- Since $f > 0$, p_{opt} satisfies

$$\frac{df}{dp} = -\frac{a}{p^2} + b = 0$$

- By solving the equation for p, we have $p = \sqrt{a/b}$ and thus

$$p_{\mathrm{opt}} = \begin{cases} \lfloor p \rfloor & \text{if } f(\lfloor p \rfloor) < f(\lceil p \rceil) \\ \lceil p \rceil & \text{otherwise} \end{cases}$$

Proposed performance modeling

1. Measure the execution time of a user program with different numbers of threads
2. Learn a performance model

   $$f(p) = a/p + bp + c$$

   with the 3 parameters a, b, and c determined by the least squares method
3. Predict the optimal number of threads p_{opt}

Related work

- Various approaches to detailed performance modeling
- E.g. Performance Analysis and Characterization Environment (PACE) by Kerbyson *et al.*
  - Semi-automated code analysis of user programs
  - Predefined hardware models

- Our performance modeling is much simpler
  - Due to the primary objective of SILC: Independence from libraries, environments, and languages

Numerical experiments

- Purpose: Validation of the performance modeling
- Example applications
  1. Cloth simulation based on the implicit Euler method
  2. CFD simulation based on the Moving Particle Semi-implicit (MPS) method
  3. An initial value problem of the 2-dimensional diffusion equation

Test environments

User programs (clients)	Parallel SILC server
Dell Dimension 8400 Intel Pentium 4 3.4 GHz, 1 GB RAM, Microsoft Windows XP SP2	SGI Altix 3700 32 Intel Itanium 2 1.3 GHz, 32 GB RAM (cc-NUMA), Red Hat Linux AS 2.1
MinGW (GCC 3.2.3) `-O3` option enabled	Intel C Compiler 9.1 `-O3` option enabled

- Both machines in the same Gigabit Ethernet LAN

Validation criterion

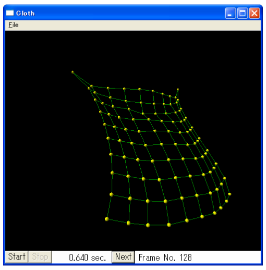
- Relative error $\varepsilon_{\mathrm{rel}}$ in the execution time t measured with p_{opt}

$$\varepsilon_{\mathrm{rel}} = \frac{|t_{\mathrm{true}} - t|}{t_{\mathrm{true}}}$$

where t_{true} is the execution time measured with the true optimal number of threads ($\varepsilon_{\mathrm{rel}} = 0$ if p_{opt} is truly optimal)

Example #1

- Time-dependent simulation of cloth motion
  - A mass-spring model for representing cloth
  - The implicit Euler method for computing cloth motion
  - A sparse linear system is solved for each time step
  - Solver: CG method in the Lis iterative solvers library
  - Visualization via OpenGL



Outline of the simulation #1

Do some initialization (defining cloth geometry, etc.)
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

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

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

3. Update particle motion.

$$\boldsymbol{v} \leftarrow \boldsymbol{v} + \Delta \boldsymbol{v}$$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} + \Delta t\, \boldsymbol{v}$$

Force and its derivatives

- Force

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

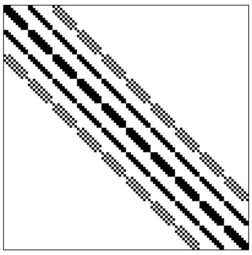
$$\boldsymbol{f}_{ij} = b_k \left(|\boldsymbol{x}_j - \boldsymbol{x}_i| - l_k \right) \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|} \qquad \text{(spring force)}$$

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

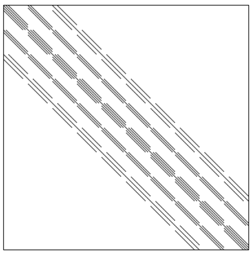
- Derivatives (Jacobian matrices)

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

Non-zero patterns of the derivatives



$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$$



$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}}$$

Elements of the derivatives

- Off-diagonal blocks (3×3 submatrices)

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

- Diagonal blocks (3×3 submatrices)

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

- All computations can be implemented by means of SILC's mathematical expressions

The body of the loop over time steps

```
/* 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");
```

Example #1: Results

- The execution time of the first 20 time steps
  - Problem size: 1,024 particles (3,096 unknowns)

Number of threads	1	2	4	8	16
Execution time [sec.]	13.5	7.23	4.41	3.28	3.24

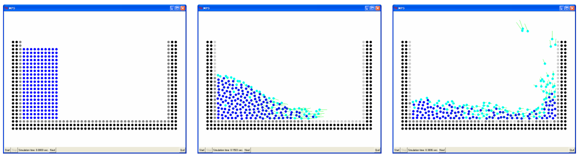
- Performance model

$$f(p) = 12.540/p + 0.103p + 0.838 \quad (q = 1.12\times10^{-2})$$

- The optimal number of threads $p_{\mathrm{opt}} = 11$
  - The true optimal number of threads was equal to p_{opt}
  - Relative error $\varepsilon_{\mathrm{rel}} = 0$

Example #2

- Simulation of incompressible flow based on the Moving Particle Semi-implicit (MPS) method
  - A sparse linear system is solved for each time step
  - Linear solver: ICCG method in Lis



t = 0 [sec.] t = 0.15 t = 0.3

Outline of the simulation #2

Do some initialization (generating particles, etc.)
For each time step ($k = 1, 2, 3, \ldots$):

Explicitly calculated

Implicitly calculated

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

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

Discretizing the equation results in a sparse linear system, which is solved by the ICCG method

3. Calculate pressure gradient terms

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

4. Correct particle motion

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

All computations can be implemented by means of SILC's mathematical expressions

Example #2: Results

- The execution time of the first 200 time steps
  - Problem size: 470 particles (470 unknowns)

Number of threads	1	2	4	8	16	32
Execution time [sec.]	29.4	19.6	14.4	12.1	12.2	20.5

- Performance model

$$f(p) = 23.467/p + 0.402p + 6.103 \quad (q = 5.97\times10^{0})$$

- The optimal number of threads $p_{\mathrm{opt}} = 8$
  - The true optimal number of threads was equal to p_{opt}
  - Relative error $\varepsilon_{\mathrm{rel}} = 0$

Example #3

- Solve the following initial value problem using the Crank-Nicolson method

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad (t \ge 0,\ x,\ y \in (0, 1))$$

$$u(x, y, 0) = \begin{cases} 1 & \text{if } x, y \in (0.4, 0.6) \\ 0 & \text{otherwise} \end{cases}$$

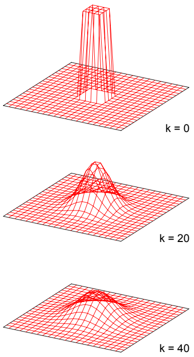
$$u(0, y, t) = u(1, y, t) = u(x, 0, t) = u(x, 1, t) = 0$$

- Let $t_0 = 0$ be the initial time and $\Delta t > 0$ be a time interval, and find $\boldsymbol{u}_k$ at $t_k = t_{k-1} + \Delta t$ by solving

$$A\boldsymbol{u}_k = C\boldsymbol{u}_{k-1}$$

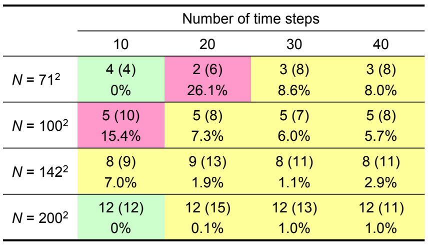
Outline of the simulation #3

```
silc_envelope_t A, C, u;

/* create matrices A, C and vector u0 */

SILC_PUT("A", &A);
SILC_PUT("C", &C);
SILC_PUT("u", &u); /* u0 */
for (k = 1; k <= n_steps; k++)
{
   SILC_EXEC("b = C * u");
   SILC_EXEC("u = A \\ b");
   SILC_GET(&u, "u"); /* uk */

   /* output solution uk at time tk */

}
```



Example #3: Results

- p_{opt} is accurate or $\varepsilon_{\mathrm{rel}}$ is small (< 10%) : OK
- $\varepsilon_{\mathrm{rel}}$ is large : NG (2 of 16)



	Number of time steps			
	10	20	30	40
$N = 71^2$	4 (4) 0%	2 (6) 26.1%	3 (8) 8.6%	3 (8) 8.0%
$N = 100^2$	5 (10) 15.4%	5 (8) 7.3%	5 (7) 6.0%	5 (8) 5.7%
$N = 142^2$	8 (9) 7.0%	9 (13) 1.9%	8 (11) 1.1%	8 (11) 2.9%
$N = 200^2$	12 (12) 0%	12 (15) 0.1%	12 (13) 1.0%	12 (11) 1.0%

Upper: p_{opt} (the true optimal number of threads in parentheses)
Lower: Relative error $\varepsilon_{\mathrm{rel}}$ in the execution time measured with p_{opt}

APT mechanism for SILC

- Outline
  1. A server collects n samples of execution time with different numbers of threads. For each number of threads, timing is done m times and the shortest is picked
  2. The server learns a performance model using the least squares method and predicts p_{opt}
  3. The server continues the simulation with the optimal number of threads
- Open issue
  - How to determine n and m

Summary

- Proposal of simple performance modeling for time-dependent simulations in SILC
  - Use of the least squares method
  - Accurate prediction of p_{opt}
- Outline of an APT mechanism for SILC
- Future work
  - Implementation of the APT mechanism
  - How to determine the two parameters n and m

Advertisement

- SILC v1.2 is freely available at

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

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