Toward Automatic Performance Tuning for Numerical Simulations in the SILC Matrix Computation Framework

(Extended abstract)

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

[1] CREST, Japan Science and Technology Agency, Saitama 332–0012, Japan
[2] The University of Tokyo, Tokyo 113–8656, Japan
[3] University of Tsukuba, Ibaraki 305–8550, Japan

1 Introduction

Numerical simulations are important techniques that reduce the costs of experimentation in various scientific and industrial fields. A common feature in many applications of numerical simulations is the appearance of partial differential equations (PDEs) as a mathematical model of phenomena to be simulated. Discretizing the PDEs results in systems of linear equations, which are solved by some linear solvers to obtain numerical solutions of the original PDEs. Some numerical simulations require only one linear system to be solved. Examples of such simulations are those that investigate the steady states of phenomena in linear problems. Other simulations require a number of linear systems to be solved one after another. Such simulations include steady-state simulations in nonlinear problems, as well as time-dependent simulations which figure out the non-steady states that evolve over time.

Solving linear systems constitutes a major part of the execution time of a numerical simulation, so that it is crucial to solve the linear systems efficiently. To make this possible, many linear solvers have been proposed. Some of the linear solvers are provided in the form of matrix computation libraries, which reduce the burden of writing application programs for numerical simulations on one hand. The availability of various libraries, on the other hand, makes it difficult to try out alternative libraries to find the best one. This situation is worsen by the diversity of computing environments from desktop PCs to high-performance computing environments, as well as interoperability issues due to mixture of different programming languages.

To relieve the difficulties associated with the use of matrix computation libraries and to facilitate the development of application programs for numerical simulations, the authors have been proposing a matrix computation framework named Simple Interface for Library Collections (SILC) [1, 2]. The SILC framework is a piece of middleware to be present between users' application programs and the matrix computation libraries to be used, allowing the users to

In Proceedings of the Second International Workshop on Automatic Performance Tuning (iWAPT2007), September 20-21, 2007, Koshiba Hall, University of Tokyo, Tokyo, Japan.

write their application programs independently of particular libraries, computing environments, and programming languages. SILC is currently implemented based on a client-server architecture, in which application programs of SILC are clients of a SILC server. There are three types of SILC servers available: sequential servers that run on uniprocessor machines, OpenMP-based parallel servers for shared-memory parallel machines, and MPI-based parallel servers for distributed parallel computing environments. SILC clients are either sequential or MPI-based parallel programs. In this paper, we focus on a sequential SILC client and OpenMP-based parallel (multithreaded) SILC server.

The purpose of the paper is to present a method for obtaining a performance model of a SILC client that conducts a numerical simulation in the SILC framework. The performance model to be obtained describes the execution time of a SILC client in terms of the number of threads on which a parallel SILC server runs (in this paper, we use the terms "thread" and "processor" interchangeably, assuming that at most one thread is assigned to a processor). The obtained performance model is then used to determine the optimal number of threads for the particular combination of the SILC client and server. Finally, based on this performance modeling method, we propose an automatic performance tuning mechanism for numerical simulations in SILC.

2 The SILC Matrix Computation Framework

SILC is a client-server system in which SILC clients (i.e., users' application programs) utilize matrix computation libraries through a SILC server. SILC clients first deposit data (such as matrices and vectors) into a SILC server together with names for later reference. Next, the clients make a series of requests for computation (such as solution of linear systems) by means of textural mathematical expressions. These computation requests are translated into calls for appropriate library functions and carried out on the server side. Finally, the clients retrieve the computation results (if necessary) from the server.

In SILC, solution of a linear system $A\boldsymbol{x} = \boldsymbol{b}$ can be requested by a textual mathematical expression like "`x = A \ b`", where A is a coefficient matrix, $\boldsymbol{b}$ is a right-hand side vector, and $\boldsymbol{x}$ is a solution vector. In the textual expression, `A` and `b` are previously defined names and new variable `x` is defined. Both dense and sparse linear systems can be handled by the same expression shown above. Selection of linear solvers is automatic in such a way that dense linear systems are solved by LU factorization, while iterative methods are employed for sparse linear systems. The selection of linear solvers are also configurable by users by means of the `prefer` statement. If there are alternative libraries having different linear solvers, users can choose a desirable library by an expression like "`prefer leq_lis`" for example, where `leq_lis` is an iterative solvers library available in SILC. In the requests that follow the `prefer` statement, linear systems are solved by means of the specified library. Besides the solution of linear systems, SILC provides a rich set of operators and built-in functions for expressing computation requests, as well as good support for various data types, matrix storage

formats, and arithmetic precisions. Even the entire application program of a numerical simulation can be implemented only by means of SILC's mathematical expressions.

The primary cost in using SILC is the communication time required for data transfer between a client and a server. However, matrix computations have a characteristic that they tend to be time-consuming even with a small amount of input data. For instance, solving a dense linear system with n unknowns takes $O(n^3)$ time, while the size of input data is on the order of $O(n^2)$. Since the communication time is almost proportional to the data size, using a parallel SILC server leads to a significant reduction of computation time even at the cost of data transfer. Similar analysis can be made with regard to sparse linear systems. In many cases, the cost of data transfer in SILC can be canceled with a speedup by parallel computation in the parallel SILC server.

To obtain the maximum performance of parallel computation, on the other hand, how many threads should be used is not a trivial question. There are cases that using all of the available processors in a parallel machine results in poor performance. Therefore, we need to know the optimal number of threads that a parallel SILC server can achieve the best performance in order to maximize the speedup by means of parallel computation. The optimal number of threads depends on not only the computing environment and matrix computation libraries to be used, but also the problem to be solved (i.e., a particular application of numerical simulations), so that it is necessary for the SILC framework to have a mechanism for automatic performance tuning that allows a parallel SILC server to determine the optimal number of threads at run time according to the problem in hand.

Bringing automatic performance tuning to the SILC framework is beneficial from users' viewpoint. SILC is in fact an abstraction layer that hides the details of matrix computation libraries and underlying computing environments. Users can make computation requests without knowing exactly what is going on under the abstraction layer, which gives a SILC server a good deal of opportunity to do automatic performance tuning. The server can automatically perform various kinds of performance tuning independently of users' application programs. Users do not have to modify their application programs in order to tune performance. The SILC framework establishes an ideal foundation in which the automatic performance tuning technology can be put to good use.

3 Performance Modeling

The execution time of a SILC client is modeled as a function in terms of the number of threads p on which a parallel SILC server runs:

$$f(p) = a/p + bp + c \tag{1}$$

where a/p is part of computation time that is parallelized, bp is parallelization overhead that is proportional to the number of threads, and c includes time for

sequential, non-parallelizable computation as well as communication time for data transfer between the client and the server (a, b, $c > 0$).

Suppose that we have measured the execution time of a SILC client by means of a parallel SILC server running on n different numbers of threads. Let f_j be the measured execution time in the case of $p = p_j$ ($1 \leq j \leq n$). Based on these n samples of the measured execution time, the three unknown coefficients a, b, and c in Equation (1) can be determined by the least squares method [3].

Having the function f with the three coefficients determined, we estimate the optimal number of threads that leads to the minimum execution time. Since $p > 0$ and thus $f(p) > 0$, the necessary condition for the minimum $f(p)$ is

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

By solving the equation for p, we find the optimal number of threads as follows:

$$\phi_{opt} = \sqrt{\frac{a}{b}} \tag{2}$$

3.1 Examples

As a practical example of the performance modeling presented above, we take a SILC client that carries out a time-dependent numerical simulation of cloth motion [2]. The simulation employs a mass-spring model which represents the geometry of cloth as a mesh of particles connected by springs. The motion of the cloth (governed by Newton's law of motion) is computed by the backward implicit Euler method [4].

The algorithm of the cloth simulation is briefly described as follows. Let N be the number of particles and $\boldsymbol{x}_i \in \mathbf{R}^3$ be a position vector for particle i ($1 \leq i \leq N$). The geometry of the entire cloth is simply denoted by $\boldsymbol{x} \in \mathbf{R}^{3N}$. Similarly, $\boldsymbol{v}_i \in \mathbf{R}^3$ represents the velocity of particle i and those of all particles are simply denoted by $\boldsymbol{v} \in \mathbf{R}^{3N}$. Let $\Delta t > 0$ be a constant time interval and $\boldsymbol{x}_0$ and $\boldsymbol{v}_0$ be the position and velocity of the cloth at the end of the previous time step. The main loop over time steps consists of the following three steps:

Step 1. Compute force $\boldsymbol{f} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v})$ that acts on the cloth, and its derivatives $\partial \boldsymbol{f}/\partial \boldsymbol{x}$ and $\partial \boldsymbol{f}/\partial \boldsymbol{v}$. The force is calculated particle-wise as follows. Let P_i be a set of particles that are connected to particle i. Then, the force $\boldsymbol{f}_i$ that acts on particle i is defined as a sum of spring force $\boldsymbol{f}_{ij}$ and damping force $\boldsymbol{d}_{ij}$ between each pair of particles i and j that are connected by spring k as follows:

$$\begin{aligned}
\boldsymbol{f}_i &= \sum_{j \in P_i} (\boldsymbol{f}_{ij} + \boldsymbol{d}_{ij}) \\
\boldsymbol{f}_{ij} &= b_k(|\boldsymbol{x}_j - \boldsymbol{x}_i| - l_k) \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|} \\
\boldsymbol{d}_{ij} &= -h_k(\boldsymbol{v}_i - \boldsymbol{v}_j)
\end{aligned}$$

where b_k is a spring constant, h_k is a damping constant, and l_k is a rest length of spring k. The derivatives $\partial \boldsymbol{f}/\partial \boldsymbol{x}$ and $\partial \boldsymbol{f}/\partial \boldsymbol{v}$ are Jacobian matrices [5], each of which consists of N^2 submatrices as follows:

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

Off-diagonal submatrices are defined as follows:

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

where I is the 3×3 unit matrix. Diagonal submatrices are defined in terms of off-diagonal ones as follows:

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

Step 2. Solve a linear system $A\Delta \boldsymbol{v} = \boldsymbol{b}$ to find a change in velocity $\Delta \boldsymbol{v}$, where

$$A = M - \Delta t^2 \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} - \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}}$$
$$\boldsymbol{b} = \left\{ \boldsymbol{f} + \Delta t \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \boldsymbol{v}_0 \right\} \Delta t$$

and M is a $3N \times 3N$ diagonal matrix that represents the mass of particles. The linear system is solved by the Conjugate Gradient (CG) method [6] since A is sparse and symmetric positive definite.

Step 3. Update velocity $\boldsymbol{v}$ and position $\boldsymbol{x}$ as follows:

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

□

The SILC client is written in C and the entire simulation is implemented by means of SILC's mathematical expressions. For each time step, the SILC client makes a fixed number of requests for computation including a solution of a sparse linear system as well as sparse matrix additions and multiplications for creating the coefficient matrix and right-hand vector of the linear system.

Table 1 shows the measured execution time of the SILC client executed on a laptop PC (Intel Pentium M 1.10 GHz, 768 MB RAM, Windows XP SP2) together with a parallel SILC server running on SGI Altix 3700 (32 Intel Itanium 2 1.3 GHz processors, 32 GB RAM, Red Hat Linux AS 2.1). The client and server machines are in the same Fast Ethernet (100 Mbps) LAN. In addition, the CG method of the Lis iterative solvers library [7] was used to solve a linear system

Table 1. The measured execution time of a SILC client together with a parallel SILC server running on different numbers of threads.

Number of threads	1	2	4	8	16
Execution time (in seconds)	13.636	7.354	4.419	3.153	3.226

for each time step. The SILC client was build with MinGW (GCC 3.2.3), while the server and Lis were compiled using Intel C Itanium compiler version 9.1 with the `-O3` optimization option enabled. The number of particles (i.e., the problem size) is 1,024 and thus the dimension of the linear system is 3,072.

Using the least squares method and the 5 samples of the measured execution time in Table 1, we determine the three unknowns in (1) as follows. Let q be the sum of squared differences between f_j and $f(p_j)$:

$$q = \sum_{j=1}^{n} (f_j - a/p_j - bp_j - c)^2$$

We want to minimize q to obtain the best possible performance model of the SILC client. Since q is a convex function, the necessary conditions for the minimum q are as follows:

$$\begin{aligned}
\frac{\partial q}{\partial a} &= -2 \sum p_j^{-1} (f_j - a p_j^{-1} - b p_j - c) = 0 \\
\frac{\partial q}{\partial b} &= -2 \sum p_j (f_j - a p_j^{-1} - b p_j - c) = 0 \\
\frac{\partial q}{\partial c} &= -2 \sum (f_j - a p_j^{-1} - b p_j - c) = 0
\end{aligned}$$

Each summation is computed from 1 to n. By separating each summation into four terms and moving terms including the unknowns to the left-hand side, we have

$$\begin{cases}
a \sum p_j^{-2} + bn + c \sum p_j^{-1} = \sum f_j p_j^{-1} \\
an + b \sum p_j^2 + c \sum p_j = \sum f_j p_j \\
a \sum p_j^{-1} + b \sum p_j + cn = \sum f_j
\end{cases} \tag{3}$$

Based on the data shown in Table 1, we also have

$$\begin{aligned}
n &= 5 \\
\sum p_j^{-2} &= 1.33203125 \\
\sum p_j^{-1} &= 1.9375 \\
\sum p_j^2 &= 341 \\
\sum p_j &= 31 \\
\sum f_j p_j^{-1} &= 19.01295319
\end{aligned}$$

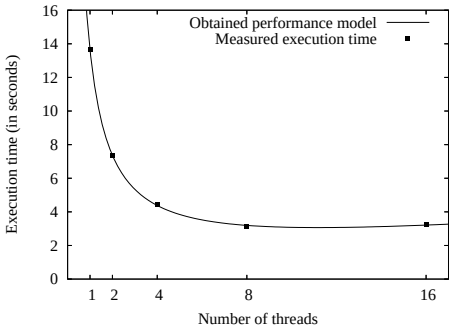

Fig. 1. The obtained performance model in (4), plotted together with the measured execution time shown in Table 1.

$$\sum f_j p_j = 122.856456$$
$$\sum f_j = 31.787137$$

Substituting them into (3) yields a linear system with regard to the three unknowns. By solving this linear system, we have $a = 12.751$, $b = 0.102$, and $c = 0.784$. Thus, the execution time of the SILC client is modeled as follows:

$$f(p) = 12.751/p + 0.102p + 0.784 \tag{4}$$

We also have $q = 3.400 \times 10^{-3}$. Figure 1 shows a plot of the equation together with the data in Table 1.

Equation (2) gives the optimal number of threads $\phi_{opt} = 11.176$ and thus $f(11.176) = 3.06542$. In reality, however, the number of threads must be integer, so that we determine which of $\lfloor \phi_{opt} \rfloor = 11$ and $\lceil \phi_{opt} \rceil = 12$ leads to the shorter execution time.

$$f(11) = 3.06571$$
$$f(12) = 3.07119$$

Therefore, the optimal number of threads in this example is expected to be $p_{opt} = 11$. Table 2 shows the measured execution time of the SILC client in the case of $p = p_{opt} \pm 2$. These supplementary experimental results confirm that the estimated number of threads is certainly optimal. The relative error of $f(11)$ is 3.010×10^{-3}, which shows that the obtained performance model is accurate.

We also examined the performance modeling method with different samples of the measured execution time of the same SILC client. We changed the client machines from the laptop machine to a desktop PC (Intel Pentium 4 3.4 GHz, 1 GB RAM), which has a Gigabit Ethernet (1 Gbps) LAN connection to the server machine (SGI Altix 3700). Based on 5 samples of the measured execution

Table 2. Supplementary experimental results for confirming the optimal number of threads $p_{opt} = 11$.

Number of threads	9	10	11	12	13
Execution time (in seconds)	3.216	3.123	3.057	3.081	3.158

time with different numbers of threads from 1 to 16, we obtained a performance model of the SILC client as follows:

$$f(p) = 12.540/p + 0.103p + 0.838$$

We also have $q = 1.117 \times 10^{-2}$. Since $\phi_{opt} = \sqrt{12.540/0.103} = 11.040$ and $f(11) = 3.11011 < f(12) = 3.11799$, we estimated the optimal number of threads is $p_{opt} = 11$. Based on supplementary experimental results in the case of $p = p_{opt} \pm 2$, we confirmed that the estimate was accurate. The relative error in $f(11)$ is 9.369×10^{-3}, which shows that the estimate is accurate.

4 Automatic Performance Tuning for Numerical Simulations in SILC

Based on the performance modeling presented in the previous section, we plan to implement a simple mechanism for automatic performance tuning in the SILC framework. We consider a SILC client that conducts a time-dependent numerical simulation by means of a parallel SILC server. Performance tuning is automatically carried out on the server side independently of the SILC client as follows:

1. For the first mn time steps of the simulation, the SILC server automatically collects n samples of the measured execution time with different numbers of threads (e.g., 1, 2, 4, ..., 2^{n-1}). For each number of threads, the server tries to measure the execution time of a single time step m times and pick the shortest execution time as the sample with regard to the number of threads.
2. The server obtains the performance model of the SILC client based on the measured execution time and estimates the optimal number of threads for the simulation.
3. The server carries out the rest of the simulation on the optimal number of threads.

Implementation of the performance tuning mechanism is part of our future work. We also need to devise a method for automatically choosing reasonable values for the number of trials m and the number of samples n, since lots of trials for many samples with non-optimal numbers of threads may significantly increase the cost of automatic performance tuning.

5 Related Work

Accurate prediction of execution time based on the performance modeling of application programs is a major research topic in the area of automatic performance tuning. To achieve the optimal performance tuning, many approaches have been proposed based on collection of performance information prior to the execution of application programs, extensive analysis of application codes, employment of detailed model parameters, and so on. For example, the Performance Analysis and Characterization Environment (PACE) [8] conducts the performance modeling of a user program based on semi-automated analysis of task dependency and communication patterns in the user program. The results of the code analysis, combined with predefined hardware models, are then used to control the user program during its execution so that the optimal performance is achieved.

In contrast, our performance modeling method is quite simple. Although additional experiments would be needed for more validation of performance models to be obtained, the initial experimental results in the previous section have suggested that our method is promising. Moreover, the primary objective of the SILC framework is to achieve a high degree of independence from particular libraries, computing environments, and programming languages. SILC is meant to allow users to write user programs in any programming language without caring about the libraries and underlying computing environments in use. That is why we have taken the present approach to automatic performance tuning, i.e. based on a simple performance model that does not require detailed performance parameters with regard to user programs, together with a performance tuning mechanism that can be largely implemented on the server side.

6 Concluding Remarks

In this paper, we presented a method for performance modeling for numerical simulations in the SILC framework. Experimental results with a SILC client for cloth simulation showed that our performance modeling yields an accurate performance model of the SILC application. We also outlined a performance tuning mechanism for time-dependent numerical simulations in SILC.

In the presentation in the workshop, we will further examine the accuracy and usefulness of our performance modeling method with other examples of numerical simulations in SILC.

Acknowledgment. This research was supported by a grant-in-aid project [9] in the Core Research for Evolutional Science and Technology (CREST) program of the Japan Science and Technology Agency.

References

1. Kajiyama, T., Nukada, A., Hasegawa, H., Suda, R., Nishida, A.: SILC: A flexible and environment independent interface for matrix computation libraries. In: Proc. PPAM 2005, LNCS 3911. (2006) 928–935

2. Kajiyama, T., Nukada, A., Suda, R., Hasegawa, H., Nishida, A.: Cloth simulation in the SILC matrix computation framework: A case study. In: Proc. PPAM 2007, LNCS, Springer. (to appear)
3. Kreyszig, E.: Advanced Engineering Mathematics, 8th Edition. Wiley (1999)
4. Baraff, D., Witkin, A.: Large steps in cloth simulation. In: Proc. ACM SIGGRAPH '98. (1998) 43–54
5. Lang, S.: Calculus of Several Variables, 2nd Edition. Addison-Wesley (1979)
6. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards **49** (1952) 409–436
7. SSI Project: User's Manual for Lis 1.0.2. (2006) http://ssi.is.s.u-tokyo.ac.jp/lis/.
8. Kerbyson, D., Papaefstathiou, E., Nudd, G.: Application execution steering using on-the-fly performance prediction. In: Proc. HPCN'98, LNCS 1401. (1998) 718–727
9. Nishida, A., Kotakemori, H., Kajiyama, T., Nukada, A.: Scalable software infrastructure project. In: Proc. SC06, poster. (2006) http://ssi.is.s.u-tokyo.ac.jp/.