# 計算環境に依存しない行列計算ライブラリインタフェース SILC

長谷川 秀彦[†1,†3] 須田 礼仁[†2,†3] 額田 彰[†3]
梶山 民人[†3] 中島 研吾[†4,†3] 高橋 大介[†5,†3]
小武守 恒[†3] 藤井 昭宏[†6,†3] 西田 晃[†2,†3]

本報告では、行列計算ライブラリの使い勝手を向上させるため、(1) データの受け渡しと演算処理の指定を分離する、(2) 演算処理の指定は文字列（数式）で行う、(3) 演算処理にはユーザプログラムのメモリ空間を使用しないという特徴を持つライブラリインタフェース SILC (Simple Interface for Library Collections) を提案する。開発中の SILC の行列計算向き命令記述、多様な計算環境（逐次、並列）での実装方式についても述べる。SILC を利用することで、ユーザはデータは必要最低限（作業領域はいらない）、計算環境が変わってもソースプログラムは変更不要、複数ライブラリの同時使用が可能（ライブラリの出所によらない）というメリットを享受できる。

# Computing Environment Independent Interface for Matrix Computation Library

Hidehiko Hasegawa,[†1,†3] Reiji Suda,[†2,†3] Akira Nukada,[†3]
Tamito Kajiyama,[†3] Kengo Nakajima,[†4,†3]
Daisuke Takahashi,[†5,†3] Hisashi Kotakemori,[†3] Akihiro Fujii[†6,†3]
and Akira Nishida[†2,†3]

We propose a new library interface SILC (Simple Interface for Library Collections) which (1) separates passing data and requesting computation at calling library program, (2) uses mathematical expression in text data for a computation request, and (3) uses no user program's memory for computation done by library programs. Then the SILC enables users to prepare no working storage in a user program for any library program, to run a user program in any computing environment with no program change, and to use many types of libraries in a same program easily. This advantage is practically important for users to use their program in seamless.

## 1. はじめに

数値シミュレーションの核となる行列計算（連立一次方程式の解法、固有値計算、 FFT など）には、様々な高性能数値計算ライブラリが提供されている。[1)] 実際は、それらがユーザの書こうとしているプログラムから手軽に使えるわけではない。ライブラリを使うには、ライブラリが定めた形式のデータを用意し、必要な作業用領域（多くは配列）を見積もって確保して、それらを決められた順序でサブルーチンに渡してやらなければならない。ソースプログラムで提供されたライブラリの場合は、あらかじめ使用するコンピュータでコンパイルしておく必要がある。うまくリンクができて、プログラムが実行できたとしても、原因不明の実行時エラー（アルゴリズムに依存する不可避なエラーやシステムに関係したエラーなど）によって、新たに頭をかかえることになるかもしれない。うまく実行できたとしても、他の計算環境（たとえばパソコンから PC クラスタ）に移ろうとすれば、プログラムの書き換えに再び労力を費やすことになる。

本報告では、(1) データは必要最低限（作業領域はいらない）、(2) 計算環境が変わってもソースプログラムは変更不要、(3) 複数ライブラリの同時使用が可能（ライブラリの出所によらない) という特徴を持つライブラリへのインタフェースとそれを実現するシステムについて述べる。

ライブラリ本体や性能に関しては現状と比べて大き

†1 筑波大学 図書館情報メディア研究科
Grad. Sch. of Lib., Info. & Media Stud., U. of Tsukuba
†2 東京大学 情報理工学系研究科
Grad. Sch. of Info. Sci. & Tech., the Univ. of Tokyo
†3 科学技術振興機構 戦略的創造研究推進事業
JST CREST
†4 東京大学 理学系研究科
Grad. Sch. of Science, the University of Tokyo
†5 筑波大学 システム情報工学研究科
Grad. Sch. of Sys. & Info. Eng., University of Tsukuba
†6 工学院大学
Kogakuin University

な変化はないが、個々のライブラリプログラムはそのままでも、複数のライブラリから必要なライブラリプログラムが同時使用できれば、ユーザの選択肢は格段に広がる。また、逐次、共有並列、分散並列といった計算環境に依存しないインタフェースを提供することで、ユーザが作成したプログラムの稼働範囲を広げることもできる。重要なのは、ライブラリの利用者とライブラリプログラム作成者にとっての単純さである。

## 2. 数値計算ライブラリの問題点

ライブラリプログラムは共通に使われるプログラムの集まりで、ユーザはこのプログラムを利用することによって、内部の詳細を知ることなく目的の処理が実行できる。[2] C や Fortran プログラムから使われる数値計算のライブラリが数値計算ライブラリである。たとえば Fortran で連立一次方程式 $Ax = b$ の解を求めるプログラムは次のようになる。ここでは、ライブラリプログラム LU で係数行列 $A$ を LU 分解し、SOLVE で右辺ベクトル $b$ に対する前進後退代入を行う。

```
PARAMETER (LDA=101,N=100)
REAL *8 A(LDA,N), B(N)
INTEGER IP(N), STATUS

係数行列を配列Aに、右辺を配列Bに与える
CALL LU(LDA,N,A,IP,STATUS)
IF( STATUS.EQ.0 ) THEN
   CALL SOLVE(LDA,N,A,IP,B)
      /*  A の値は壊され、解が B に入る  */
END IF
END
```

LU と SOLVE には、ライブラリの仕様に定められた形式のデータを用意し、所定の順序で変数を記述しなければならない。同じ連立一次方程式を別のライブラリプログラムで解く場合は、データの格納方法、変数の並べ方などに変更がいる。どのプログラミング言語から使えるのかについても注意がいる。一般には、プログラムを書き換えず、同一のユーザプログラムで単精度計算と倍精度計算、４倍精度計算の結果を比較することは不可能である。

疎行列に対する反復解法のライブラリでは、計算アルゴリズムは同じでも、用いられるデータ格納形式が多数あるため、それらに対応したプログラムを用意していてはライブラリプログラムの数が膨大になってしまう。そこで、プリミティブな計算である行列・ベクトル積などのインタフェースの仕様を定めておき、この部分の計算プログラムをユーザに作成させる ( Reverse Communication[3] )。このようなやり方では、完成度の高いライブラリをユーザに提供できたことにはならない。

このような問題を解消するため、Common Component Architecture[4] ではオブジェクト指向言語によるラッパーを用意し、多くのライブラリをスムースにつなげて各種プログラミング言語から利用できるようにしている。プログラム間で CCA の規約に沿ったデータ受け渡しを行うためのラッパーは誰かが用意する必要がある。このような枠組みは、多くのデータ形式に対応できるが、新たなデータ形式を導入するのはたいへんな作業になる。

SPMD 形式の並列プログラムからライブラリを使うことを考えよう。ユーザプログラムのそれぞれが逐次ライブラリを呼び出したり、持っているデータ全体を渡して並列ライブラリを呼び出したりするのは既存の呼び出し法でもできる。しかし、 p 並列の並列プログラム全体が、 q 並列で構成された並列ライブラリを呼び出す場合は、ライブラリへのデータの渡し方、データ全体の情報の交換など、ユーザプログラムとライブラリプログラム間のインタフェースに難問が発生する。ライブラリプログラムがすべての場合の準備しておくのは不可能だし、並列計算プログラムとライブラリプログラム間のインタフェースに決定版はない。このためユーザプログラムと並列計算ライブラリ間のデータ受け渡しは、並列行列計算ライブラリ ScaLAPACK[5] のようにファイルを利用するのが一般的である。

これらの問題の原因は、

- 格納形式や処理の詳細を制御する引数など、低レベルの記述を用いていること
- 名前やインタフェースが、ライブラリ、精度、格納形式、並列処理手法などの実装ごとに異なること

である。このようなライブラリ呼び出し法は手軽かつ直感的で、逐次プログラム同士の場合、少数の限られたライブラリを使う場合、他のプログラムと独立な場合などにはよいが、並列プログラムの場合、多くのプログラムを合せて使う場合、複数の計算環境で実行する場合などでは問題となる。

## 3. SILC (Simple Interface for Library Collections) の考え方

ユーザとしては、多様な計算環境を意識せず、特に逐次か並列かを意識せず、同一のプログラムが変更なしに使えればうれしい。そこで、単純にデータを渡して、計算の一切を任せることを考える。具体的には、仮想的な計算環境にデータを預けて、預けたデータに対する演算処理を依頼し、必要になったときに仮想的な計算環境から結果を受け取るような仕組みである。ユーザは、仮想的な計算環境が逐次か並列か、どんなライブラリが動作しているかなどは意識しなくてよいし、作業に必要な領域を用意する必要もない。

われわれは上記の「夢」を実現するため、三つの特徴

(1) データの受け渡しと演算処理の指定を分離する
(2) 演算処理の指定は文字列（数式）で行う
(3) 演算処理にはユーザプログラムのメモリ空間を使用しない

を持つインタフェース SILC (Simple Interface for Library Collections) を提案する。演算処理の指定に文字列を使うのは、複数のプログラミング言語に対して統一的なインタフェースを提供するためである。

SILC では、２章で示した連立一次方程式 $Ax = b$ の解法を

```
PARAMETER (LDA=101,N=100)
REAL *8 A(LDA,N), B(N), SOL(N)
INTEGER IP(N), STATUS

係数行列を配列Aに、右辺を配列Bに与える
CALL PUT('A', LDA, N, A, STATUS)
CALL PUT('b', N, B, STATUS)
      /*  A, B を転送し A, b と名付ける */
CALL SILC(' x = A\b ')
      /*  連立一次方程式を解く */
CALL GET('x', N, SOL, STATUS)
      /*  解 x を配列 SOL に転送する */
END
```

のように書く。必要に応じて、どのライブラリプログラムを使うとか、演算精度はどうするかなどを指定する。

SILC システムでは、ライブラリプログラムとユーザプログラムが共通に利用する仮想的なメモリ空間を用意し、データはそこに存在させる。ユーザプログラムは計算に先立って、必要なデータ（行列、ベクトル）に名前をつけてその仮想的なメモリ空間におき、それから処理を依頼する。指示に応じて、ライブラリプログラムが仮想的なメモリ空間からデータを受け取って処理し、処理結果をその空間に戻す。処理結果は必要になったときに受けだす。仮想的なメモリ空間内のデータは、演算によって破壊されることなく、消滅を指示されるか仮想的なメモリ空間が消滅するまで存在する。ユーザプログラムとライブラリプログラムは仮想的なメモリ空間とデータのやりとりができればよく、それぞれの動作環境（逐次か並列かなど）は異なってもよいし、プログラミング言語の制約もない。データ形式の変換は SILC システムの担当で、ユーザプログラムのデータ構造に対応した PUT, GET ルーチン、既存のデータ形式との変換は、データ形式の数だけ必要になるが、それでも個々のデータ形式に対応した解法ルーチンを作るよりは少ない労力ですむだろう。`'X=A+B; Y=A-B'` のような演算指示から自動的に並列性を抽出すれば、逐次実行のプログラミング言語中に並列処理を記述したことになる。並列実行可能なリソースがあれば並列実行によって高速化できる。SILC の考え方は、プログラミング言語、使用するライブラリ、計算環境（単一マシン、複数マシン、並列環境、Grid 環境）に左右されない一般的な枠組みを提供することにある。SILC の仕組みを使えば、パソコン上のスクリプト言語 Python からクラスタで動作する ScaLAPACK を利用するようなシステム[6] も容易に構築できる。

上記のような便利さを実現するために引き換えにした部分もある。たとえば、

- 必要なメモリ量が増えること
- データ転送が必要になること
- メモリコピーが増えること
- すべてが動的であること

である。しかし、メモリ量の制約は緩和される方向だし、将来的にはデータ転送も高速化されるだろう。

さらに、行列計算は大量のデータを必要とし、それ以上に膨大な演算が必要なために高速化が強く要求されていることと、どんな大規模行列であっても演算指示を数式で書けば高々数文字で書けるという特徴がある。たとえば、$N \times N$ の密行列 $A$ の行列式の値 $\det(A)$ を求めるには、$N^2$ のデータを転送し、$O(N^3)$ の演算を行うが、結果は一つのスカラー量である。すなわち、いったんデータを仮想的なメモリ空間に転送してしまえば、数十バイトの文字列で膨大な演算量の処理を制御することができる。しかも、演算を実行するのはその計算環境に最適化された並列プログラムでありうる。したがって、現状でも転送に必要なコストを上回る性能向上の可能性があるだろう。

ライブラリにあるプログラムをリンクしてユーザプログラムに取り込むという静的な考え方と、計算を依頼した時点で最新のライブラリプログラムによって処理されるという動的な考え方には、大きなギャップがある。ライブラリとバージョンを指定して使えるようにすることも可能だろうだが、現時点では多様なライブラリの最新版を容易に使えるようにするのが目的である。したがって、古いバージョンのライブラリプログラムは将来に残らず（存在せず）、これまでの「ある時点でライブラリをリンクして凍結したユーザプログラム」とは発想を転換してもらう必要がある。

以上の議論から、 SILC のメリットをまとめると以下のようになる。

### 3.1 ユーザのメリット

(1) 計算環境（単一マシン、複数マシン、並列環境、Grid 環境）によってソースコードが変わらない。通信時間、演算時間は変わってもプログラムの全体像は変わらない
(2) 仮想的なメモリ空間を経由して複数のプログラムを容易に組み合わせられる。また、仮想的なメモリ空間をデータ変換に用いたり、プログラ

ムの分割に用いることもできる

(3) 同一インタフェースで複数のライブラリプログラムが利用できる。データが仮想的なメモリ空間に登録できれば、あとはどの「ライブラリの機能」が使いたいかを記述すればよい

(4) 並列実行が自然に導入される。ユーザプログラムとライブラリプログラムは非同期並列実行であり、リソースが許せば高速化される。演算処理の指定から並列性が自動抽出されて並列実行される可能性もある

### 3.2 ライブラリ開発者のメリット

(1) プログラムを最新に保てる。バグ入りのプログラムをリンクして残されることもなく、常に最新のプログラムを使ってもらえる

(2) ユーザプログラムのデータ構造を意識しなくてよい。仮想的なメモリ空間のデータ形式は自由なので、最適なデータ形式、精度、分散方法などを選択できるし、1種類だけのデータ格納方式に対応した解法でも（性能を意識しなければ）多様なデータ格納形式に対応できる可能性がある

(3) 作業領域の確保や分散方法の変更が、ユーザプログラムとは独立に行える。インタフェースを変更することなく、実行環境に合わせた最適化ができる

(4) 既存のアプローチと矛盾しない。既存の大多数のプログラムを SILC の枠組みに組み込める

## 4. SILC の仕様と実装方式

### 4.1 SILC の命令記述

SILC の命令記述言語は一種のプログラミング言語である。しかし SILC が線形代数を中心とする数値計算ライブラリに用いられることを目的としていることから、数学的な演算を簡易に記述できるように高度化された言語とする。しかし、ユーザが習得しやすいように、既存のプログラミング言語や数学システムとあまり異ならないように注意も払う。

**識別子：** SILC では数値や行列などに名前を付けてプログラミング言語の「変数」のように扱うことができる。しかし、型やサイズの宣言のようなことは行わない。識別子には任意の型、任意のサイズのデータをバインドすることができる。単純な代入文では、識別子にすでにどのような型のどのような値がバインドされていても無視され、新しい値がバインドされる。

**制御構文：** SILC では反復を記述する構文を意図的に排除している。SILC は他のプログラミング言語からライブラリとして呼び出されるため、反復が必要であれば呼び出し側のプログラミング言語の反復構文を用いれば十分である。もしそのような使い方が著しく非効率なのであれば、実行時に命令が供給される SILC の構文で反復を記述してもやはり非効率であろうから、そのような操作はライブラリレベルで実現するべきである。また、現在予定している言語では、条件分岐構文もない。従って、SILC の命令は本質的にすべてが単純な実行文である。しかし、条件分岐の一種であるマスク演算 (Fortran 90 以降にある where 文のようなもの) は今後追加を検討する価値があると考えている。

**エラーの防止および意味の明確化：** SILC では、行列やベクトルなどの要素・範囲アクセスの際には添え字範囲のチェックを行う。また、記述された命令の意味が曖昧にならないように文法が設計されており、実行時のチェックを行うことができる。まず、代入は「文」として扱い、C のように一つの文に複数の代入が行えるようにはしない。もし代入を式として扱うと、

```
A = f(X) + (X = Y)
```

のような命令では `f` の引数として渡されるのが `X` の代入前の値なのか、代入後の値なのか明確でなくなってしまう。同様の理由で、（値を返す）関数の引数は値を変更するものを許さない。例えば

```
A = f(X) + g(X)
```

のような命令で、`f` と `g` が引数を変更してしまうようであれば、渡される引数の値が不明確になるからである。そこで、引数を変更できるのは値を返さない「手続き」に限ることとする。さらに、一つの手続きに値を変更できる引数が複数ある場合、引数として渡されるものに重複がある「エイリアス」が生じないよう実行時にチェックを行う。例えば `p` という手続きの最初の 2 つの引数は値が変更され得、最後の引数はそうでない場合、

```
p(X[0:9], X[10:19], X[0:19])
```

のように呼び出せば可である（最後の引数はコピーが作られて渡される）が、

```
p(X[0:9], X[5:14], Y)
```

では最初の 2 つの引数が干渉してエイリアスとなっており、実行時エラーとなる。しかしエイリアスチェックに時間がかかり性能に影響するような場合には、エイリアスのチェックを行わないオプションの導入も検討する必要があると思われる。

**演算子など：** 詳細は紙数の都合で記述できないが、概略としては以下のようになる。

単項演算子：配列参照 `A[1:2]`, 関数呼び出し `f(x)`, 共役転置 `A'`, 複素共役 `A~`, 複素行列に対する単純転置 `A'~` または `A~'`。なお、配列の参照では添え字として整数ベクトルを与えることができ、順序置換や gather/scatter などの演算が記述できる。

二項演算子：四則演算、剰余 `a % b`, 連立一次方程式の求解 `A \ b` (`Ax = b` の解 `x` を求める)

その他、ベクトル・行列を生成する構文などが予定されている。また、各種の定数・関数・手続きも必要に応じて実装される予定である。

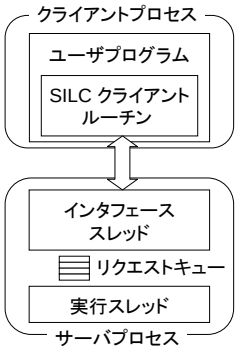


**図 1** クライアント 1 プロセス、サーバ 1 プロセスの構成

**ライブラリ呼び出し：** ユーザプログラムから呼ばれる SILC ルーチン（クライアントルーチン）と、SILC サーバにおけるライブラリの実行とは、基本的に非同期である。すなわち、クライアントルーチンがサーバに命令を送信すると、サーバは構文チェックのみを行ってクライアントに ack を返し、（データの読み出し命令など必要な場合を除いて）クライアントルーチンはユーザプログラムに制御を戻す。従って、SILC サーバがライブラリを実行している間にユーザプログラムは別の処理を行うことが可能である。

ユーザプログラムが並列の場合も、演算実行命令は一つのユーザプロセスが依頼すれば十分である。複数のユーザプロセスから命令が送られる場合の実行順序制御のための SILC 命令がいくつか提供される予定であるが、詳細は現在検討中である。

### 4.2 SILC の実装

**クライアント・サーバ構成：** 図 1 は最も簡単な構成である、クライアント 1 プロセス・サーバ 1 プロセスの場合の実装方式の概念図である。クライアントとサーバが同一のプロセッサ上にあってもよいし、ネットワークなどで結合された異なるプロセッサ上にあってもよい。

ユーザプログラムは SILC のクライアントルーチンを呼び出し、クライアントルーチンは基本的に引数をそのまま SILC サーバに転送する。

SILC サーバは大きく分けるとインタフェーススレッド、実行スレッドと、その間をつなぐリクエストキューからなる。クライアントルーチンからの命令はインタフェーススレッドが受信し、構文チェックが行われてリクエストキューに追加された後、ack が返される。実行スレッドはリクエストキューから順次命令を取り出し、命令に対応するライブラリを実行する。

クライアントとサーバが共に並列で同数のプロセスからなる場合には、クライアントごとに異なるサーバが接続する図 2 の構成が自然である。各プロセッサにクライアントとサーバが一つづつ乗ることにより、従来型の並列ライブラリと同じ構図となる。この場合もクライアントルーチンは命令をサーバのインタフェーススレッドに単純に転送し、サーバのインタフェーススレッドはそれをリクエストキューに追加する。各サーバのリクエストキューに格納された命令列の実行順序の制御、ライブラリ演算で必要となるサーバプロセス間のデータ通信などは、すべて実行スレッドが行う。

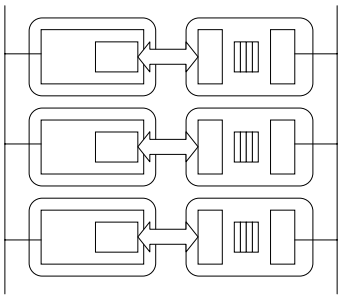

**図 2** クライアント・サーバが同数の並列プロセスからなる構成

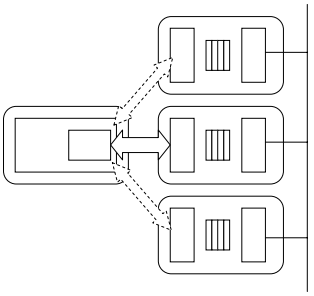

**図 3** 並列サーバに逐次クライアントが接続する場合の構成

クライアントが逐次でサーバが並列の場合、図 3 実線のようにクライアントがサーバプロセスの一つだけに接続する方式が考えられる。この場合、実行ルーチンが必要に応じて命令やデータを他のサーバプロセッサに転送し、並列実行を行う。一方、図 3 破線で示すように単一のクライアントが複数のサーバプロセスに接続する方式も可能である。こうすることにより、データ転送などの処理の一部が効率化できる可能性がある。

複数のクライアントから単一のサーバに接続する図 4 のような構成も考えている。この構成は単純にクライアントが並列プログラムである場合も含むが、演算結果を別プロセスで可視化したり、連成問題を複数のプログラムの疎結合で解いたりするような場合にも適用できる。この場合にはインタフェーススレッドが複数立ち上がり、それぞれのクライアントとのやり取りを制御する。リクエストキューは各クライアントに対して作られ、複数のクライアントからの命令の実行順序制御は実行スレッドが行う。

**実行スレッド構成：** 図 5 は現在想定しているサーバ内の実行スレッドの構成方式を示している。

まず、リクエストキューには構文解析が完了した命令が蓄えられている。リクエストキューが複数ある場合、「実行順序制御」ユニットはそこから次に実行すべ

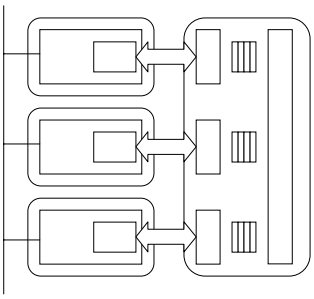

**図 4** 複数クライアントが単一サーバに接続する場合の構成

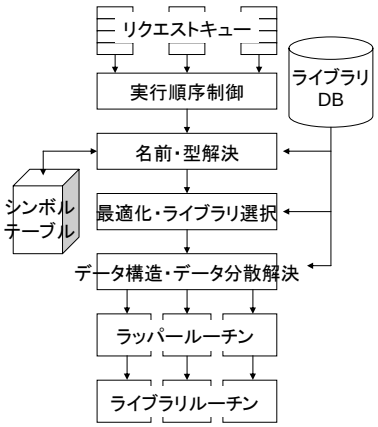


**図 5** SILC サーバの実行スレッドの構成方式

き命令を一つ選び出す。次の「名前・型解決」ユニットは、まずライブラリデータベースから関数や手続きの引数に関する情報を参照する。さらにシンボルテーブルを参照し、識別子にバインドされたデータを取り出し、型やサイズを確定する。次に「最適化・ライブラリ選択」ユニットがライブラリデータベースを参照し、演算の内容や順序の最適化および適切なライブラリの選択を行う。SILC は複合的なライブラリコレクションを統合することを想定しているため、ライブラリごとに想定しているデータ構造やデータ分散が異なると考えられる。そこで「データ構造・データ分散解決」ユニットが使用するライブラリルーチンに適するように必要に応じてデータ構造やデータ分散を調整・変換する。そしてライブラリルーチンは、インタフェースを統一するために各ルーチンにかぶせられたラッパールーチンを経由して呼び出される。

多くのライブラリを容易に取り込めるよう、ライブラリルーチンにかぶせるラッパールーチンはごく簡単なもので済むように設計する。何らかの記述言語を準備して、ラッパールーチンとライブラリデータベースのための情報を半自動的に生成することも検討している。これらのインタフェースは公開して、プロジェクト外部で SILC 対応のライブラリを提供したり、ユーザが独自のルーチンを追加したりすることもできるようにする計画である。

## 5. おわりに

ライブラリはリンクして使うという固定観念を捨て、(1) データを預け、(2) 計算を依頼し、(3) 必要なときに結果を取り寄せるというアプローチをとることで、多様な計算環境で多数のライブラリを手軽に使える可能性を示した。またそのようなシステムが、逐次環境（１台のマシンあるいは２台のマシン）、並列環境（共有、分散）で実装できることを示した。ユーザプログラムからみた相手先は仮想化されているので、 Grid 環境であっても理論的には問題ないはずである。

これまで、ライブラリはアルゴリズムの詳細を隠してユーザに提供するとか、限られた環境でユーザプログラムに対する影響を最小限にしつつ高速性を提供してきたが、現在では１種類のライブラリで済むでなく、一人のユーザが利用する計算環境も１台の PC からクラスタまで多岐にわたる。しかもコンピュータハードウェアの命は短く、プログラムの寿命は長くなっていく。個々の環境に対するオーバーヘッドによるリソースや性能のロスを許容し、 SILC のような仕組みを活用して労力を別の活動に向けるべきではなかろうか？

**謝辞** 本研究は，科学技術振興機構 (JST) 戦略的創造研究推進事業 (CREST) 「大規模シミュレーション向け基盤ソフトウェアの開発」プロジェクトの一部として実施した.

## 参 考 文 献

1) 村田健郎，小国力，三好俊郎，小柳義夫編著: 工学における数値シミュレーション，丸善 (1988).
2) 山本喜一, 榊原進, 野寺隆志, 長谷川秀彦: これだけは知っておきたい数学ツール, 共立出版 (1999).
3) 長谷川里美他訳: 反復法 Templates, 朝倉書店 (1996).
4) http://www.cca-forum.org/
5) Blackford, L. et al., *ScaLAPACK Users' Guide*, SIAM, Philadelphia, Pennsylvania(1997).
6) Piotr Luszczek and Jack Dongarra: *Design of Interactive Environment for Numerically Intensive Parallel Linear Algebra Calculations*, ICCS 2004, Krakow Poland.