A MATLAB-Based Code Generator for Parallel Sparse Matrix Computations Utilizing PSBLAS

Taiji SASAOKA†, Nonmember, Hideyuki KAWABATA†a, and Toshiaki KITAMURA†, Members

SUMMARY Parallel programs for distributed memory machines are not easy to create and maintain, especially when they involve sparse matrix computations. In this paper, we propose a program translation system for generating parallel sparse matrix computation codes utilizing PSBLAS. The purpose of the development of the system is to offer the user a convenient way to construct parallel sparse code based on PSBLAS. The system is build up on the idea of bridging the gap between the easy-to-read program representations and highly-tuned parallel executables based on existing parallel sparse matrix computation libraries. The system accepts a MATLAB program with annotations and generates subroutines for an SPMD-style parallel program which runs on distributed-memory machines. Experimental results on parallel machines show that the prototype of our system can generate fairly efficient PSBLAS codes for simple applications such as CG and Bi-CGSTAB programs.

key words: matrix computations, MATLAB, program translation, parallel libraries

1. Introduction

Parallel programs for distributed memory machines are not easy to create and maintain, especially when they involve sparse matrix computations. Sparse matrix computations frequently arise in scientific simulations. For example, when partial differential equations are to be solved numerically by some discrete approximation techniques such as finite-element or finite-difference methods, linear systems with large sparse coefficient matrices have to be solved. Since this kind of applications usually require a huge amount of computations, parallel processing for solving sparse linear systems to reduce the execution time is considered to be inevitable. However, it is quite tiresome to construct a program dealing with sparse matrices which runs even on a scalar machine. Parallelizing it to make it run on a distributed computer is a burden for the user indeed. To make matters worse, a parallel program which is tuned for a parallel machine might not run very fast on another machine because characteristics of a parallel machine, in general, greatly vary from one machine to another.

In order to alleviate the situation, we observe that we have to think of two aspects of programming environments. One is the encapsulation of details of things from the user, i.e., intensive use of libraries. Especially for numerical computation programs, the user is encouraged to use library routines for primitive operations such as a matrix-vector product because 1) the user can avoid the need to think of details of implementation of those algorithms, 2) library routines which survived the test of time are possibly correct, and 3) those routines can be expected to be much more efficient than naive implementations for the same algorithms by the user. Some libraries can even deal with parallel sparse computations. By using well-known libraries, the user can construct reliable, as well as portable in terms of performance, programs.

Another aspect we should think of is the interface between the user and the computer, i.e., the specification of the programming language. The user should be able to easily read, write and modify programs. Since application programmers might not be experts of computer systems, subtle differences of syntax, or even symbols used in the language, may affect the productivity of mathematical programs. In case of matrix computations, appropriate data structures and matrix operations compatible to ordinary mathematical notations should be available. The user may want to write s*A*v for sAv, where s, A, and v represent a scalar s, a matrix A, and a column-vector v, respectively.

Considering those two points, existing programming environments for parallel machines are, unfortunately, not successful enough for efficient large-scale parallel sparse matrix computations.

Library packages such as BLAS[4]–[6] and LAPACK[7] are known to be effective and vendor-tuned routines of them are usually accessible on most commercial computers. Parallel matrix computations are supported by ScaLAPACK[8] and sparsity of matrices are maintained by PSBLAS[9]. However, it could be laborious to use those library routines because many of subroutines of such libraries usually require a lot of parameters supplied by the user. In addition, languages which can be used to utilize such libraries are often limited to those similar to Fortran or C, in which matrix computations have to be expressed in pointwise manner.

There are several approaches to design parallel languages or extend existing languages to deal with parallelism. HPF[1] is an extension of Fortran which offers a standardized way to direct distribution patterns of arrays used in the program. OpenMP [2], [3] provides the user another way of directive-based language extension for Fortran or C. They

*Note that Fortran 90's matrix operations are not consistent with mathematical notations—you can not write A*B when both A and B are not scalar variables.
are useful for simple parallel programs but currently not suitable enough for sparse matrix computations because no special treatment for sparse matrices is considered.

Compared to C or Fortran, languages like MATLAB [10] may be more appropriate as a candidate for an ideal mathematical programming environment. Using MATLAB, the user can write mathematical programs as if he/she is writing mathematical algorithms on paper as shown in Fig. 1. Figure 1 clearly depicts the similarity of the syntax of both descriptions. MATLAB’s execution environment is interpreter-based and its performance is known to be problematic in some cases, but language translation techniques exist to make MATLAB programs run fast [11]–[14] and some of them support generating parallel codes. However, as for parallel and sparse computations, there is no programming environment available to the authors’ knowledge.

In this paper, we propose a program translation system for generating parallel sparse matrix computation codes utilizing PSBLAS. The purpose of the development of the system is to offer the user a convenient way to construct parallel sparse code based on PSBLAS. The system reads a MATLAB program, also known as an M-file, with just a few directives to denote attributes of input variables. The system generates subroutines for an SPMD-style parallel program which runs on machines where PSBLAS library is available, i.e., most parallel computers with MPI communication library. The functionality of generating PSBLAS code is being built into our matrix language compiler named CMC which translates MATLAB-based scripts into Fortran or C [14], [15]. Experimental results show that our prototype system can emit fairly efficient PSBLAS programs for simple linear sparse solvers.

The rest of the paper is organized as follows: In Sect. 2, we describe libraries for matrix computations and the structure of programs using PSBLAS. The design and implementation of our code generation system is presented in Sect. 3. Evaluation of the system over experimental results are shown in Sect. 4. Related work is summarized in Sect. 5. In Sect. 6, we summarize the proposal with some concluding remarks.

2. Preliminary: Libraries for Matrix Computations

2.1 Basic Mathematical Libraries for Matrix Computations

Frequently used basic numerical algorithms have been constructed as library packages. Representatives of them include Basic Linear Algebra Subprograms (BLAS) [4]–[6], which comprises many computational routines among matrices and vectors. Another package named Linear Algebra PACKage (LAPACK) [7], which consists of solvers of linear systems and eigenproblems, is constructed assuming the existence of BLAS. Library routines for sparse matrix computations are proposed as Sparse BLAS [18]. They are available on most commercial high-performance computers with supports by their vendors. By writing programs using those library routines, the user can keep his/her program’s portability of performance to be high.

LAPACK has been extended to manage parallel computation on distributed machines as ScaLAPACK [8]. ScaLAPACK is built on a set of communication subprograms for matrix computations named BLACS [19]. BLACS is implemented over general-purpose communication libraries such as MPI.

ScaLAPACK does not support sparse matrices; every matrix is manipulated as two-dimensional arrays. In order to incorporate sparse matrix computations in user programs, library support of data structures and computations for sparse matrices are inevitable (or else the user has to pay much effort to write sparse computations manually using BLACS or MPI directly). PSBLAS is one of the libraries that are designed for granting such kind of desire of the user.

2.2 PSBLAS: A Library for Parallel Sparse Matrix Computations

PSBLAS is a library package which supports sparse matrix computations on distributed-memory machines [9]. PSBLAS is designed to fit in the layer between application programs and BLACS as shown in Fig. 2. Parallel programs with PSBLAS are of Single Program and Multiple Data
(SPMD) scheme which runs basically in the data-parallel manner.

Figure 3 illustrates the general structure of a parallel program with PSBLAS. In a program using PSBLAS, each sparse matrix, which is referenced to in data-parallel matrix computations, have to be distributed among separate processors. Each distribution pattern is recorded in a structure of data called a descriptor. The user carries out each matrix computation invoking an appropriate computational routine passing pointers of distributed matrices and vectors, and a corresponding descriptor, to the routine. Each computational routine 1) checks which part of the data has to be transferred among separate processors obeying owner-computes rule, 2) transfers, i.e., sends and receives, data, and 3) does computations. All computational routines must be called by all processes because they employ global communications for error checking. Distribution patterns of matrices among processes are, although restricted to be row-wise, almost arbitrary. The user can control the detailed scheme by preparing external subroutines which define the scheme such that “the elements of row \(i\) goes to process number \(j\).” Those routines are called PARTS routines in [9].

A program with PSBLAS has to begin with initializing and constructing a process topology using BLACS routines. So the user is required to have some knowledge of BLACS to use PSBLAS. In addition, PSBLAS does not support every functionality of BLACS, e.g., PSBLAS assumes that the underlying process grid defined using BLACS is of one-dimensional. Although interfaces of computational routines of PSBLAS is similar to corresponding BLAS routines, they are not exactly the same. Taking those into account, it can be said that writing parallel programs with PSBLAS routines might not be very easy for the user.

3. PSBLAS Code Generation from MATLAB-Based Scripts

3.1 Overview of the System

We designed a MATLAB-based code generator for parallel sparse matrix computations utilizing PSBLAS and have been developing a prototype of it. The system

- reads a MATLAB M-file with a few annotations about input variables’ attributes,
- determines other variables’ attributes in the M-file and extract detailed information of each operation, and
- generates a set of routines utilizing PSBLAS and BLACS libraries, which are used to construct parallel executables for distributed memory machines.

Generated set of routines by the system consists of a computational routine and a gateway routine. The former corresponds to “Matrix computations” part in Fig. 3 and the latter deals with “Creation of descriptors,” “Preparation of matrices,” and “Deallocation” parts in the figure.

Sparse matrices used in the computation are supposed to be given by the user using Compressed Column Storage (CCS) or Compressed Row Storage (CRS) format [25]. Although the distribution scheme of matrices and vectors is limited to row-wise block-cyclic because of PSBLAS’s limitation, almost arbitrary pattern can be specified by the user by writing a PARTS routine.

For constructing a parallel executable, the user is supposed to do a few preparation. First, the user has to insert instructions to initialize BLACS (with additional variable declarations), which is the “Initialization” part in Fig. 3, at the very beginning of the user’s program. Next, the automatically generated gateway routine should be called as just like a computational routine with an additional variable. Last, finalization code has to be inserted at the end of the program, which corresponds to “Finalization” part in Fig. 3. Since the parallel executable is supposed to be started by a command such as \texttt{mpirun} of MPI, code segments written by the user (including preparations of sparse matrices) should be guarded such that a single processor executes them exclusively.

After the gateway routine is called by the user routine (at all processors), sparse matrices and vectors are automatically distributed as specified by PARTS routines. The computational routine, which is a result of direct translation of the routine supplied as M-file, is invoked from the gateway routine. The result of computation is gathered in the gateway routine and passed back to the user’s program.

ootnote{A running parallel PSBLAS program consists of multiple processes. Multiple processes can exist on the same single processor although each process does not share any resources including memory area. In this paper, unless otherwise stated, we assume that each process is expected to run on a separate processor.}
The targets of our program translation system are environments where PSBLAS library is available: Regardless of the underlying operating system and the hardware configurations, translated programs would run on the system provided the software hierarchy on the machine is as shown in Fig. 2. In Sect. 4, experimental results not only on a distributed-memory machine but also on an SMP are shown.

There are some minor limitations on programming; current implementation of our prototype can not deal with elementwise modification of distributed sparse matrices. Besides, since matrix computations are carried out by calling PSBLAS routines, available matrix operations are restricted to those PSBLAS routines cover. Also, some features of the MATLAB language such as dynamic reshaping of matrices are not supported.

3.2 Analysis of MATLAB Scripts by CMC

Our prototype of MATLAB-to-PSBLAS translator is based on CMC which translates MATLAB-based scripts into Fortran [14], [15]. The flow of the translation by CMC is illustrated in Fig. 4. CMC reads annotated MATLAB scripts and generates Fortran subroutines, which can be linked with other user-supplied routines. By using the CMC’s facility of program analysis and optimization as a front end, we developed a code generator to deal with PSBLAS routines. In this section, CMC’s basic analysis functionality is summarized briefly.

In MATLAB, no variable declaration is used and types of variables, as well as the meaning of each operation, are determined dynamically. In order to translate a MATLAB program into languages like C or Fortran, the translator has to fix each variable’s attributes such as shape, size, and intrinsic type, statically. CMC does this by using a few annotations given by the user.

Intermediate representation of a program treated in CMC is an abstract syntax tree (AST) which retains the structure of control sequences and mathematical expressions in the program. Several analyses and optimizations, including dependence analysis, renaming of variables, variable attribute analysis, and reduction of the number of variables, are done over ASTs.

Figure 5(a) shows an example of a tree which represents an assignment expression $y = \alpha A x + \beta y$ where $\alpha$ and $\beta$ are scalars, $A$ is a full matrix, and $y$ is a column vector. Each node which constitutes the tree of Fig. 5(a) is either a leaf node, which corresponds to a variable reference, or an internal node, which is an operator of a matrix operation. In the course of iterative analysis, the attributes of leaves should be fixed first, and an operator’s attributes are determined (synthesized) looking at its operands’ attributes. Tables 1 and 2 are the rules to derive shape and structure, respectively, of an expression based on its subexpressions’ attributes. For the expression in Fig. 5(a), the shapes of nodes (except for an assignment operator) are determined as shown in the figure.

The meaning, or implementation, of the operator will be determined depending on its operands’ attributes. Note that the operator ‘*’ can be recognized as a matrix-vector product, a matrix-matrix product, or a multiple of a scalar value, depending on the shapes of its operands. Intrinsic type and sparse schemes of operands are also taken into account to select appropriate instruction sequences in the output language for the operation.

When the analyzing stage described above finished, a few classical optimizations such as dead code elimination,
### Table 1 Resulting shapes for binary operations between A and B.

<table>
<thead>
<tr>
<th>shape of A</th>
<th>shape of B</th>
<th>S</th>
<th>R</th>
<th>C</th>
<th>U</th>
<th>L</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>S</td>
<td>R</td>
<td>C</td>
<td>U</td>
<td>L</td>
<td>D</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>Row vec.</td>
<td>R</td>
<td>S</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column vec.</td>
<td>C</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upper tri. mat.</td>
<td>U</td>
<td>C</td>
<td>U</td>
<td>F</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower tri. mat.</td>
<td>L</td>
<td>C</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagonal mat.</td>
<td>D</td>
<td>C</td>
<td>U</td>
<td>L</td>
<td>D</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full mat.</td>
<td>F</td>
<td>C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) shape of $A^\times B$.

<table>
<thead>
<tr>
<th>shape of A</th>
<th>shape of B</th>
<th>S</th>
<th>R</th>
<th>C</th>
<th>U</th>
<th>L</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>S</td>
<td>R</td>
<td>C</td>
<td>U</td>
<td>L</td>
<td>D</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>Row vec.</td>
<td>R</td>
<td>S</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column vec.</td>
<td>C</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upper tri. mat.</td>
<td>U</td>
<td>C</td>
<td>U</td>
<td>F</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lower tri. mat.</td>
<td>L</td>
<td>C</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diagonal mat.</td>
<td>D</td>
<td>C</td>
<td>U</td>
<td>L</td>
<td>D</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full mat.</td>
<td>F</td>
<td>C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) shape of $A\pm B$, $A-B$, $A_+/B$, and $A^{-}\ast B$.

†: For operations $+$ and $-$, resulting shape is $F$ when the scalar value is not zero.

### Table 2 Resulting structure of a binary operation between A and B.

<table>
<thead>
<tr>
<th>shape of A</th>
<th>shape of B</th>
<th>dense</th>
<th>sparse</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>(scalar)</td>
<td>dense</td>
<td>sparse</td>
</tr>
<tr>
<td>dense</td>
<td>dense</td>
<td>dense</td>
<td>sparse</td>
</tr>
</tbody>
</table>

†: Resulting structure is dense when the operation is an addition or a subtraction.

code motion, and common subexpression elimination, and matrix-computation-specific ones like strength reduction of matrix operations are performed\[14\],[15]. In some cases, dynamic memory management mechanisms have to be incorporated. Although CMC has been extended to take care of those cases\[16\], current code generator for PSBLAS programs does not support dynamic reallocation of memory for matrices.

### 3.3 Translating the Computational Part: Incorporating BLAS-Like Computational Routines

The interface of the computational routines of PSBLAS has been designed keeping the similarity to BLAS routines.† Some BLAS and PSBLAS routines are shown in Table 3. Many of BLAS or PSBLAS routines can treat complicated computations, requiring several parameters given by the user. For example, DGEMV, which compute $y \leftarrow \alpha A x + \beta y$, can be used for computing $y \leftarrow A x$, or $y \leftarrow y - A x$, by giving the parameters $\alpha$ and $\beta$ appropriately.

The flexibility of those BLAS-like routines might cause difficulties when automatic translation is considered. For example, the computation of

$$ y \leftarrow \alpha A x + \beta y $$

could be carried out by the following inefficient way:

CALL DGEMV(\ldots, \alpha, A, \ldots, 0,0\ldots)\ldots
CALL DSCAL(\ldots, \beta, \ldots)
CALL DAXPY(\ldots, 1,0\ldots, \ldots)

which means a sequence of $t \leftarrow A x$, $y \leftarrow \beta x$, and $y \leftarrow t + y$, instead of a better way like

CALL DGEMV(\ldots, \alpha, A, \ldots, \beta, \ldots).

In order to rule out such kind of redundant expressions, we construct our prototype translation system as follows. Since an expression of matrix computations can be arbitrary long, each tree which represents a mathematical expression has to be shortened so that each expression can be rewritten instead of a better way like

CALL DGEMV(\ldots, \alpha, A, \ldots, \beta, \ldots).

Note that we give matrix operations (i.e., DGEMM or DAXPY) higher priority over non-matrix operations (i.e., DAXPY or DSCAL) in the course of translation of expressions. In addition, our system tries to use as less number of BLAS calls as possible when translating matrix expressions.

Our translation algorithm of a matrix expression is

†Each set of computational functions of PSBLAS and BLAS are not a subset of the other. Although PSBLAS routines have less variety than BLAS’es, some PSBLAS routines are more complex than BLAS’es corresponding ones. For example, F90_PSSPXB BY of PSBLAS can deal with a vector expression $y \leftarrow \alpha x + \beta y$, but DAXPY of BLAS can not.

### Table 3 Computational routines of BLAS and PSBLAS.

(a) BLAS routines.

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>xSCAL</td>
<td>$x \leftarrow \alpha x$</td>
</tr>
<tr>
<td></td>
<td>xCOPY</td>
<td>$y \leftarrow x$</td>
</tr>
<tr>
<td></td>
<td>xAXPY</td>
<td>$y \leftarrow \alpha x + y$</td>
</tr>
<tr>
<td></td>
<td>xDOT</td>
<td>$rot \leftarrow x^\top y$</td>
</tr>
<tr>
<td></td>
<td>xNRM2</td>
<td>$|x|_2$</td>
</tr>
<tr>
<td></td>
<td>xGEMV</td>
<td>$y \leftarrow \alpha A x + \beta y$ etc.</td>
</tr>
<tr>
<td>Level 2</td>
<td>xTRMV</td>
<td>$x \leftarrow \alpha A x$, where $A$ is triangular.</td>
</tr>
<tr>
<td></td>
<td>xTRSV</td>
<td>$x \leftarrow \alpha A^{-1} x$, where $A$ is triangular.</td>
</tr>
<tr>
<td>Level 3</td>
<td>xTRMM</td>
<td>$C \leftarrow \alpha A B + \beta C$, etc.</td>
</tr>
<tr>
<td></td>
<td>xTRSM</td>
<td>$C \leftarrow \alpha A^{-1} B$, etc., where $A$ is triangular.</td>
</tr>
</tbody>
</table>

(b) PSBLAS Vesion 1.0 routines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F90_PSDOT</td>
<td>$\dot{rot} \leftarrow x^\top y$</td>
</tr>
<tr>
<td>F90_PSAXPBY</td>
<td>$y \leftarrow \alpha x + \beta y$</td>
</tr>
<tr>
<td>F90_PSNRM2</td>
<td>$|x|_2$</td>
</tr>
<tr>
<td>F90_PSSPMM</td>
<td>$y \leftarrow \alpha A x + \beta y$, etc.</td>
</tr>
</tbody>
</table>
| F90_PSSPSM   | $y \leftarrow \alpha A x + \beta y$, etc., where $A$ is triangular.
summarized as follows. As mentioned before, each syntax tree of a matrix expression consists of matrix operators and operands. The matching of BLAS routines, i.e., the search for a set of operators which is covered by a single BLAS call, and the restructuring of the syntax tree are repeated until when no change can be made further.

1. The tree is traversed in a pre-order manner to look for a multiplication or a division. If not found, goto step 4. If found, name the node \( n \) and examine the replaceability of \( n \) with a BLAS call:
   - If \( n \) is a matrix-matrix operation:
     - check if DTRMM, DTRSM, or DGEMM can be used for the operation by investigating the attributes of the operands. If not, continue the traversal.
   - If \( n \) is a matrix-vector operation:
     - check if DTRSV, DTRMV, or DGEMV can be used for the operation by investigating the attributes of the operands. If not, continue the traversal.
   - Otherwise:
     - check if DDOT, DNRM2, or DSCAL can be used for the operation. If not, continue the traversal.
2. Check operators adjacent to \( n \) if they are covered by the same BLAS call. If so, mark them. Try to increase the number of marked operators as many as possible.
3. Restructure the tree by replacing marked operators with an appropriate BLAS call and repeat the traversal of the tree (goto step 1).
4. The tree is traversed in a pre-order manner to look for an addition or a subtraction. If not found, stop. If found, replaceability of the node, named \( n \), with a BLAS operation is examined. name the node \( n \) and examine the replaceability of \( n \) with a BLAS call. Specifically, check if DAXPY can cope with \( n \). If not, continue the traversal.
5. Check operators adjacent to \( n \) if they are covered by the same BLAS call. If so, mark them. Try to increase the number of marked operators as many as possible.
6. Restructure the tree by replacing marked operators with an appropriate BLAS call and repeat the traversal of the tree (goto step 4).

For a simple assignment expression like \( A = B \), call for DCOPY is introduced. For PSBLAS programs, however, F90_PSAXPBY is used for a copy because routines like DCOPY are not available.

When the application of the above process to a matrix expression finishes, generation of the code corresponding to the resulting tree is straightforward. In some cases, however, modifications are needed. If operands for the BLAS operation is the result of another matrix operation, or, if “output variables” of BLAS routines appear in the right-hand-side, intermediate variables should be introduced, namely, creation of assignment expressions for them and replacement of corresponding subexpressions (or variable references) with the references to those newly introduced variables are performed.

In the case of the expression shown in Fig. 5(a), the matrix-vector multiplication, which results a value of column vector, is named \( n \) at step 1 of the above algorithm. After the decision of the replacement of the node with a call to xGEMV, all operators on the right-hand side are found, at step 2, to be covered by the same call to xGEMV. The tree shown in Fig. 5(b) is the result of the restructuring at step 3.

Figures 6 and 7 show another example of the program translation by our system. Figure 6 is a program of the CG method written in MATLAB and Fig. 7 shows the corresponding computational routine with PSBLAS, described:

```
function [x,i] = cg0s(A,x0,b,tol)
    r = b-A*x0;
    rn = norm(r);
    x = x0;
    p = r;
    i = 0;
    rho = r.' * r;
    while i
        i = i + 1;
        q = A * p;
        alpha = rho / (rho.' * q);
        x = x + alpha * p;
        r = r - alpha * q;
        rho_old = rho;
        rho = r.' * r;
        if sqrt(rho)/rn <= tol, break, end
        beta = rho / rho.old;
        p = r + beta * p;
    end
```

Fig. 6 The CG method.

```
subroutine cg0s(A, x0, b, tol, x, i, desc)
    use typesp
    use typedesc
    use f90psblas
    use f90tools
    type(d_spmat) :: A
    type(decomp.data.type) :: desc
    call blacs_gridinfo(...) 
    call f90_psaxpby(...,b,...,x,desc)
    call f90_psspm(...,A,x0,...,desc)
    r = sqrt(f90_psdot(r,r,desc))
    call f90_psaxpby(...,x0,...,x,desc)  ! x = x0
    call f90_psaxpby(...,r,...,p,desc)  ! p = r
    i = 0
    rho = f90_psdot(r,r,desc)
    do while (.true.)
        i = i + 1
        call f90_psspm(...,A,p,...,q,desc)
        alpha0 = f90_psdot(p,q,desc)
        alpha = rho/alpha0
        call f90_psaxpby(alpha,p,...,x,desc)
        call f90_psaxpby(-alpha,q,...,r,desc)
        rho_old = rho
        rho = f90_psdot(r,r,desc)
        if ((sqrt(rho)/rn).le.tol) then
            exit
        endif
        beta = rho/rho_old
        call f90_psaxpby(...,r,beta,p,desc)
    enddo
```

Fig. 7 The CG method using PSBLAS.

†For the sake of simplicity of presentation, translation of real-valued computation using BLAS routines only is shown. Complex-valued expressions and PSBLAS computational routines can be treated similarly.
In Sect. 3.1. With a few annotations on attributes of input parameters, i.e., annotations on variables $A$, $x0$, $b$, and $tol$, our prototype constructs a parallelized version of the code.

4. Evaluation

In this section, we present experimental results on the performance of PSBLAS programs generated by our prototype translation system. The purpose of this section is to show the efficiency of the translated PSBLAS programs on parallel machines; distributed parallel machines and SMPs. We used two sparse matrix computation programs implemented in MATLAB, that use the CG and Bi-CGSTAB algorithms. Both are solvers of sparse linear equations, which are used, in [9], by the developers of PSBLAS to show the performance of the library.

In order to estimate the overhead due to the use of PSBLAS library, we prepared programs written manually using MPI and compared the execution speed. Programs with MPI were coded such that the amount of communications was reduced as much as possible.

4.1 Experimental Environments

Used machines and software systems are as follows:

**Machine P1** An SMP cluster with 8 nodes.
- OS: Linux 2.4.20.
- Node:
  - CPU: Intel Xeon 2.8 GHz × 2
  - 2nd Cache: 512 MB (on chip)
  - Memory: Dual Channel DDR 1 GB
  - System bus clock frequency: 400 MHz
- Interconnect: Myrinet-2000 (2 Gbps bidirectional)
- Communication library: MPICH v1.2.5 (GM 2.0.5)
- Fortran compiler: Intel Compiler ifc v7.0 with -O3 -static
- BLAS: vender-supplied.
- PSBLAS: v1.0 compiled manually with ifc.

**Machine P2** Fujitsu HPC2500. A large-scale SMP cluster.
- 32 CPUs in an SMP node are used.
- CPU: SPARC64V 2.08 GHz, Memory: 512 GB, OS: Solaris 8.
- Fortran compiler: frt v5.6 with -Kfast, GP2=3
- MATLAB: v7.0.1 (R14) SP1.
- Communication library: MPI 2.0.
- BLAS: vender-supplied.
- PSBLAS: v1.0 compiled manually with frt.

**Machine S** A scalar machine. CPU: Pentium 4 2.4 GHz, Memory: 1 GB, OS: Linux 2.4.31. MATLAB: v7.0.1 (R14) SP1. Fortran compiler: g95 of GCC v4.0.1 with -O3. BLAS: compiled manually by g95.

Although Machine P1 is an SMP cluster with 16 CPUs (each SMP node owns two CPUs), experiments are done using up to 8 processes and no more than one process was executed on each node. Machine P2 was used to see the effectiveness of our approach on an SMP machine. Although Machine P2 offers the user a shared-memory programming environment, we used the system as a distributed environment through the use of MPI and PSBLAS. Experiments on Machine S were carried out in order to compare sequential executions of PSBLAS codes with executions of M-files on the MATLAB interpreter.

4.2 CG

The problem solved by the CG program is a finite-difference approximation of two-dimensional Laplace’s equation

$$\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} = 0$$

on a square domain using a square grid. The five-point difference scheme with natural ordering was used to construct corresponding coefficient matrices. Dirichlet condition was posed on the boundary of the domain. For every execution of the CG method, initial guess was set to zero and the CG iteration was stopped when the 2-norm of the residual vector became less than $10^{-6}$.

Figure 8 shows the results of the CG program on Machine P1. The sizes of solved problems, i.e., the number of unknowns, were 12,100 and 62,500. For the problem of each size, executions using 1, 2, 4, and 8 processors were timed. Figure 8 shows the speed-up ratio of parallelized CG programs against the sequential execution of them. Sequential executions were carried out using exactly the same code as the parallelized ones and invoked by typing like `mpirun -np 1 prog`.

Table 4 shows the detailed information of the results.

![Fig. 8 Speed-up ratio of the CG method on Machine P1.](image)
including execution times of parallel executions using 8 processors.

We can see several aspects that Fig. 8 and Table 4 imply. First, parallelized programs by our prototype can run efficiently when the size of solved problem is large. Second, generated PSBLAS programs by our prototype can compete manually-tuned MPI programs when the size of solved problems is large.

In Fig. 8, the curve labeled “MPI (12100)” shows super-linear speed-ups for small numbers of participating processors. This is not surprising; when the communication overhead is small compared to computation time, the increased cache hit rate by the distribution of data may cause such results. Unfortunately, PSBLAS programs did not exhibit such phenomena for this simple experiment.

Table 5 shows the results on Machine S as well as the results of sequential executions on Machine P1. MPI and PSBLAS were not available on Machine S but MATLAB was. In order to compare the performance of MATLAB and PSBLAS codes, following cases were timed:

1. the execution of the CG program on Machine S using the MATLAB interpreter
2. the execution of translated programs (with calls to BLACS and PSBLAS routines removed) on Machine S
3. the execution of the same code as 2. on Machine P1 using one processor
4. the execution of translated programs using BLACS and PSBLAS on Machine P1 using one processor

In Table 5, results of cases 2. and 3. are listed in columns labeled “BLAS.” Looking at those columns, we can see that a CPU on Machine P1 is about 1.8 times as fast as the processor of Machine S for the same CG program. Table 5 shows that 1) translated sequential programs run about twice as fast as executions using the MATLAB interpreter (on Machine S), and 2) translated PSBLAS programs do not require much overhead (on Machine P1). Accordingly, we may conclude that our system gives MATLAB programs the ability to make use of parallel processing facility effectively, and run much faster than sequential executions by MATLAB interpreter.

Table 6 shows the execution times on Machine P2. Although MATLAB interpreter can be used on the machine, no parallel execution was available. From the table, we see that PSBLAS programs run fairly efficiently on SMP machines, too. For example, the PSBLAS code ran 8.5 times as fast as MATLAB using 8 CPUs. The difference of the speed on 1 CPU between MATLAB and the others are larger than that on Machine S shown in Table 5. In the authors’ opinion, the reason of this is the poor performance of the Fortran compiler used on Machine P2 to compile the translated codes. At any rate, parallel execution of programs by PSBLAS outperforms the sequential execution by MATLAB interpreter.

4.3 Bi-CGSTAB

The problem solved by the Bi-CGSTAB program is a finite-difference approximation of two-dimensional elliptic equation

\[-\frac{\partial^2 Q}{\partial x^2} - \frac{\partial^2 Q}{\partial y^2} + \alpha \partial Q / \partial x = 0\]

on a square domain using a square grid. The five-point difference scheme with natural ordering was used to construct corresponding coefficient matrices. Neumann condition was posed at one side of the boundary of the domain and Dirichlet condition was posed on the other sides. The parameter \(\alpha\) was set such that \(\alpha h = 0.1\) where \(h\) is the mesh interval. Since resulting coefficient matrices of linear systems are asymmetric, the CG method can not be used to solve this type of problems. Initial guesses and stopping criteria were the same as the CG case.

Figure 9 shows the results of the Bi-CGSTAB program on Machine P1. The numbers of unknowns for solved problems were 12,100 and 62,500. For problems of each size, executions using 1, 2, 4, and 8 processors were timed. Figure 9 shows the speed-up ratio of parallelized Bi-CGSTAB programs against the sequential execution of them.

\*In fact, Sun’s Fortran compiler on Machine P2 generates faster executables often (especially when large-scale computations are involved). However, unfortunately, the compiler was not able to be used for generating parallel executables on Machine P2 using MPI.
Table 7 Execution times of the Bi-CGSTAB method in seconds on Machine P1 using 8 CPUs. The numbers of iterations for each execution are placed in parentheses.

<table>
<thead>
<tr>
<th>Number of unknowns</th>
<th>PSBLAS</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>12100</td>
<td>0.150 (170)</td>
<td>0.0549 (166)</td>
</tr>
<tr>
<td>22500</td>
<td>0.261 (247)</td>
<td>0.107 (224)</td>
</tr>
<tr>
<td>44100</td>
<td>0.554 (340)</td>
<td>0.365 (330)</td>
</tr>
<tr>
<td>62500</td>
<td>0.762 (361)</td>
<td>0.557 (336)</td>
</tr>
</tbody>
</table>

Table 8 Sequential execution times of the Bi-CGSTAB method in seconds on Machine S. Results of sequential executions on Machine M are also shown. The numbers of iterations for each execution are placed in parentheses.

<table>
<thead>
<tr>
<th>Number of unknowns</th>
<th>Machine S (scalar)</th>
<th>Machine P1 (1CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12100</td>
<td>0.923(171) 0.519(170)</td>
<td>0.305(169) 0.326(172)</td>
</tr>
<tr>
<td>22500</td>
<td>3.38(242) 1.60(244)</td>
<td>0.932(246) 0.854(226)</td>
</tr>
<tr>
<td>44100</td>
<td>9.94(336) 5.06(331)</td>
<td>2.72(318) 3.08(343)</td>
</tr>
<tr>
<td>62500</td>
<td>13.6(372) 9.27(409)</td>
<td>5.16(396) 5.10(395)</td>
</tr>
</tbody>
</table>

Table 7 shows the details of the result of parallel executions using 8 processors. Different from the CG case, the numbers of iterations are shown in parenthesis for each execution since iteration counts of PSBLAS and MPI programs were not exactly the same. The speed-up ratio illustrated in Fig. 9 is calculated based on the time required for each iteration.

Table 8 shows the results on Machine S as well as the results of sequential executions on Machine P1. The meaning of each column in Table 8 is the same as Table 5 for the CG case, except that the numbers of iterations are shown in parenthesis for each execution.

From Fig. 9, Tables 7 and 8, we can see practically the same characteristics of PSBLAS programs generated by our prototype as the CG case. The PSBLAS code exhibited slightly better performance in Fig. 9 than in Fig. 8 for the number of unknowns of 62,500. This may be attributed to the fact that the Bi-CGSTAB algorithm involves more local computations such as addition of vectors than the CG method.

Table 9 shows the execution times on Machine P2.

<table>
<thead>
<tr>
<th>Number of CPUs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>10.5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(1.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>14.6</td>
<td>5.96</td>
<td>1.44</td>
<td>0.602</td>
<td>0.420</td>
<td>0.346</td>
</tr>
<tr>
<td>(0.72)</td>
<td>(1.8)</td>
<td>(7.3)</td>
<td>(17)</td>
<td>(25)</td>
<td>(30)</td>
<td></td>
</tr>
<tr>
<td>PSBLAS</td>
<td>19.3</td>
<td>11.4</td>
<td>4.35</td>
<td>0.989</td>
<td>0.594</td>
<td>0.525</td>
</tr>
<tr>
<td>(0.54)</td>
<td>(0.92)</td>
<td>(2.4)</td>
<td>(11)</td>
<td>(18)</td>
<td>(20)</td>
<td></td>
</tr>
</tbody>
</table>

While the performance of MATLAB interpreter on 1 CPU is superior to the others on 1 CPU, parallel execution of programs by PSBLAS outperforms the sequential execution by MATLAB interpreter.

5. Related Work

MATLAB-based parallel code generators utilizing ScALAPACK exist [12], [13]. Although any sparse problem could be solved using only dense structures, i.e., using two-dimensional arrays, it is obviously not efficient in both terms of required storage space and computational load. For sparse problems, especially the number of non-zero elements in each sparse matrix is small, the usage of sparse structures is observed to be essential for problems such as CG and SOR [15].

Aztec [22] is a package of iterative sparse solvers implemented in C. Although parallel computation is available in the package, programming using the package might be much complicated compared to writing programs in MATLAB.

PETSc [20] is a library package that enables the user to write parallel sparse matrix computations by fairly simple expressions. In PETSc, the syntax of programming languages are not modified; sparse computations and parallel processing are supported at the level of library interface. Each routine’s interface are simple compared to PSBLAS. The user can write parallel programs without knowing details of data distribution and communication underlying its parallel execution. The PETSc project is still active and ongoing [21]. Detailed comparison of performance between PSBLAS and PETSc are listed in our future work.

MathWorks, the producer of MATLAB, has started to deliver Distributed Computing Toolbox (DCT) which enables parallel execution of tasks on the MATLAB interpreter [23]. Using DCT, the user can invoke mutually independent subtasks and let them run on separate processors. Unfortunately, communication among subtasks is not available. Applications which make the most use of DCT’s functionality seem to be very limited.

6. Conclusions

In this paper, we proposed a code translation system for generating parallel sparse matrix computation code with PS-
BLAS. The input for the system is a MATLAB program with annotations, which is easy to read, write and modify by the user. We have shown the structure of the system and the process of translating matrix expressions in order to use BLAS-like routines. Using the system, a parallel sparse matrix computation program can be easily constructed and maintained. Experimental results on parallel machines show that the prototype of our system can generate fairly efficient PSBLAS codes for simple applications such as CG and Bi-CGSTAB programs.

We are planning to automate the process of parallel executable construction as much as possible. Our future work also include an intensive evaluation of the system using more complicated applications, supporting for the version 2 of PSBLAS which has been released recently, and an extension of the system for enabling more sophisticated treatment for distributed data such as remapping.

Acknowledgments

This work was supported in part by the HCU Grant for Special Academic Research (General Studies) under Grant No.4111, and by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), No.4111, and by the Ministry of Education, Science, Special Academic Research (General Studies) under Grant

References


Taiji Sasaoka received his B.E. degree in Computers Engineering from Hiroshima City University in 2006. Currently he is a student of the Graduate School of Information Sciences at Hiroshima City University. His research interests include parallel computing. He is a student member of IPSJ.
**Hideyuki Kawabata** received his B.E., M.E., and Ph.D. degrees in Computers Engineering from Kyoto University in 1992, 1994, and 2004, respectively. Currently he is an Assistant Professor in the Department of Computer Engineering at Hiroshima City University. His research interests include programming systems for high-performance computing. He is a member of ACM, IEEE-CS, IPSJ, and JSSST.

**Toshiaki Kitamura** received the B.E., M.E. and Ph.D. degrees in Computers Engineering from Kyoto University in 1978, 1980, and 1996, respectively. He is currently a Professor in the Department of Computer Engineering at Hiroshima City University. Previously, he was a computer architect at Computer and System Architecture Department, FUJITSU Limited in 1983–2000. From 2000 to 2002, he was an Associate Professor in the Center for Information and Multimedia Studies, Kyoto University. His research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE, ACM, and IPSJ.