LAPACK efficiency improvement through redesign in c/c++ style

Summary
Two new versions of banded linear equations solver have been developed with extensive usage of new mechanisms available in C/C++ language. They include dynamic memory allocation and pointers. Both solvers have been compared on Intel – Solaris10 platform with DGBSV solver from high-quality LAPACK package. New solvers reduce processing time by 15%-30% in the case of doubly dynamic memory management or even by 50% for singly dynamic management. Loop unrolling has been investigated and no stable performance improvement has been observed.

Keywords: LAPACK, banded linear equations, C language style

Streszczenie
Opracowano dwie nowe wersje programu rozwiązywania pasmowych układów równań liniowych z wykorzystaniem nowych mechanizmów dostępnych w języku C++. Obejmują one dynamiczną alokację pamięci i wskaźniki. Na platformie Intel-Solaris10 porównano oba programy z programem DGDSV ze znanej świetnej biblioteki LAPACK. Przedstawione tu nowe programy redukują czas przetwarzania o 15%-30% w przypadku podwójnie dynamicznej alokacji pamięci, a nawet o 50% w przypadku pojedynczej dynamiczności. Zbadano rozwijanie pętli, ale nie uzyskano stabilnej zadowalającej poprawy przetwarzania.

Keywords: pakiet LAPACK, pasmo we układy równań liniowych, język C/C++
1. Linear algebra package LAPACK and C/C++ language

Linear algebra package LAPACK, introduced around 1992, has gained reputation of the best performer in its field [1, 2, 7]. LAPACK efficiency is supported by BLAS and ATLAS libraries. Originally Fortran language has been used for LAPACK development. However other versions, using C and Java language, not to mention C++ interface, are currently available. The C version, dubbed CLAPACK, has been developed through conversion using f2c application.

It seems that there is ample room for LAPACK efficiency improvement. Specifically dynamic memory allocation and pointers may help to achieve this goal. This thesis will be proved here basing on the example of banded linear equation system solution. The solution using DGBSV program from LAPACK package will be compared with alternative program exploiting fully new facilities of C/C++ language [5, 6].

Also comparison with GNU Scientific Library [4], developed originally in C, has been considered. However this library is inferior to LAPACK with respect to efficiency. Moreover there is no program for banded linear equations solution.

2. Basic data structures in DGBSV program and the dynamic counterparts in C/C++ style

Program DGBSV from LAPACK package stores the nonzero band of coefficient matrix in the variable dimension (parametrized) array [1]. No dynamic memory allocation is used. Partial pivoting results in fill-in and consequently in bandwidth growth. In order to accommodate this fill-in, the array is over-dimensioned. The over-dimensioning of this basic array in DGBSV may be illustrated with small example [4] of the banded matrix with one full diagonal on the right side of main diagonal and two non-zero diagonals on the left. The structure of sample basic matrix is shown below in (1):

\[
\begin{bmatrix}
  * & * & * & + & + & + \\
  * & * & + & + & + & + \\
  * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\
  a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\
  a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\
  a_{31} & a_{42} & a_{53} & a_{64} & * & * \\
\end{bmatrix}
\] (1)
Unused elements are marked with an asterisk “*” and “+” character signals eventual fill-in. It is worth to note at this point, that interchange of matrix rows is implemented through physical copying of respective matrix elements.

First alternative variant of the program, developed from the scratch in C language, stores elements of individual rows in following structure:

```c
typedef struct
{
    int length;
    int capacity;
    int ColStart;
    int ColEnd;
    double *Values;
} ONEROW;
```

The functions and usage of structure elements is quite self-explanatory. The `length` component stores actual row length, `capacity` – actual dynamic memory segment size for row storing (capacity > length), `ColStart` and `ColEnd` – column indexes (starting and trailing) in current row, and `Values` – the pointer to dynamic memory with current row coefficients. These structures are allocated dynamically (depending on matrix size) with the aid of dynamic pointer array

```c
ONEROW *AmatrixBand;
```

The pivoting results in fill-in and enlarging of some non-zero rows. The enlargement is met by dynamic reallocation of memory pointed at by `Values`. Physical interchange of pivoted rows is avoided; simple pointer interchange in `AmatrixBand` array suffices. Doubly dynamic memory allocation imposes no limits on row interchange and cuts the processing time.

Both solvers, the DGBSV from LAPACK package and the new one, have been tested on Intel Core2 platform with 2.4 GHz clock frequency. The hardware was controlled by Solaris10 operating system. GNU compiler version 3.4.3 has been used for software compilation. The system of 32768 banded equations has been solved for several values of the non-zero band width.

Comparison of both solvers (fig. 1) shows superiority, in the sense of processing time, of the true C solver with dynamic memory allocation and pointer addressing. The speed-up of true C/C++ solver is in the range of 15% to 30%, depending on actual width of non-zero band.
3. Dynamic array as an alternative for doubly dynamic memory allocation

The speed-up, achieved through introduction of new C/C++ programming constructs is quite distinct but far from spectacular. In the search for better results, simplifying of data structures has been investigated. It is commonly known that elegant and versatile data structures are very helpful in the software design, debugging and maintenance. However, the structure versatility is almost always handicapped by longer processing time.

In order to simplify data structures in new C/C++ solver, doubly dynamic memory allocation has been abandoned. It is worth to note that dynamic memory reallocation (Values pointer) on most operating system and compiler platforms, must be accompanied by explicit initialization of additional memory segment. Moreover, there has been investigated the idea of loop unrolling during vector (one-dimensional array) processing. It has been patterned after DSCAL function (J. Dongarra) from LAPACK package.

The implementation of these ideas started with static (i.e. constant size) allocation of memory for individual matrix rows. Whole set of matrix rows is described by the array of pointers

```c
double *Abmatrix;
```
Information on individual rows is stored in simpler, than in first solver version, structure, i.e.

```c
typedef struct
{
    int length;
    int ColStart;
    int ColEnd;
    // double *Values;
} ROWINFO;
```

It should be reminded that memory segments storing non-zero elements of individual rows, are not reallocated. It means that at the very outset, they are dimensioned with appropriate redundancy for fill-in accommodation, similarly as basic array in DGBSV function from LAPACK package [1].

![Graph showing execution time vs. half-bandwidth](image)

**Fig. 2:** Banded linear equation system processing time as the function of half-bandwidth: LapackBand – DGBSV solver, Roll – singly dynamic C/C++ solver without loop unrolling. Unroll – C/C++ solver with loop unrolling factor = 5 (see below)

The tests (fig. 2) performed on the same Intel-Solaris10 platform have shown that this solver version reduces processing time down to 40%-50% of the time of LAPACK solver. Quite surprising is the adverse influence of loop unrolling. Only for three specific band width values loop unrolling has led to some insignificant improvement.
4. Efficiency of loop unrolling

The surprising lack of stable improvement for loop unrolling has been investigated in more details. As the test case the system with half-band width of 512 and the number of equations equal to 32768 has been used. Almost 20 values of loop unrolling factor have been checked. The loop unrolling factor is defined here as the number of elementary operations in single loop run. Loop unrolling factor of 1 is equivalent to unrolling absence.

The test runs (fig. 3) have shown that loop unrolling does not influence the processing time in unequivocal and monotonic way. Only for few specific values of unrolling factor, equal to 3 and 8 in the test series, the processing time has been reduced to approximately 40% of standard value.

![Graph showing influence of loop unrolling factor](image)

**Fig. 3:** Influence of loop unrolling factor in C/C++ solver for the system of 32768 equations with half-bandwidth of 512

It is worth to remind the results of unrolling, for the factor of 5 (as in DSCAL from LAPACK). In this series of tests, the loop unrolling reduced markedly the processing time for several “singular” values of band-width half equal to 704, 1024 and 1344. It seems that these “singularities” may be attributed to cache memory management. Quite interesting is the hypothesis that they appear for integer coincidence of cache memory size multiple and the size of basic data structure in the solver. Detailed investigation of these singularities (if it is really worth while) is an open problem.
5. Conclusions and final remarks

It has been demonstrated, as the result of these investigations and numerical experiments, that there exists real possibility to develop an alternative software with spectacularly better efficiency than LAPACK counterpart. The experiments and development have been performed in the field of banded linear equations system solvers. The speed-up by the factor of 2, has been achieved through the usage of pointer mechanism of C/C++ language. It is quite probable that similar redesign of other components of linear algebra package can result in further advances in respective areas.

References