Two Different Data-Parallel Implementations of the BLAS *

Petter E. Bjørstad   Tor Sørevik

Abstract

Massively parallel computer systems, having thousands of identical processors operating in SIMD mode, hold the promise of delivering cost effective computing alternatives for many important problems in scientific computing. Computational linear algebra is of fundamental importance to a large class of compute intensive algorithms. This paper discusses the implementation and performance of the computational BLAS kernels in a data-parallel setting. Two different programming languages are compared and several compiler issues are discussed.

Keywords: Data-parallel, Fortran 90, SIMD, BLAS, LAPACK

1 Data-parallel programming

This paper discusses the data-parallel programming model applied to efficient implementation of basic computational kernels in linear algebra. This programming model has gained acceptance among computational scientists by way of the proposed array extensions to the Fortran language. These extensions constitute an important part of the new Fortran 90 standard (F90) [11]. The model received further attention and prominence through the pioneering work of Thinking Machines Corporation, leading to more widespread use of massively parallel computers operating in a data-parallel fashion.

Today, there is a broad consensus that this programming paradigm will be of importance on both SIMD and MIMD parallel computing platforms. The major computer vendors, in order to provide high productivity software tools, all have a data-parallel Fortran compiler as a top priority. The F90 standard and the recent work on the HPF (High Performance Fortran) definition [9], makes it likely that better and more portable software will emerge across different MPP platforms. At least one vendor (DEC), has

*Para//ab, Institutt for Informatikk, University of Bergen, N-5020 Bergen, Norway. This research was partially supported by NAVF grant D.01.08.054 and by NTNF grant IT2.28.28484

BLAS	Type of operation	Example	Data	Complexity
BLAS-1	vector-operations	daxpy	$O(n)$	$O(n)$
BLAS-2	matrix-vector op.	rank-1 update	$O(n^2)$	$O(n^2)$
BLAS-3	matrix-matrix op.	matrix multiply	$O(n^2)$	$O(n^3)$

Table 1: Functionality and Complexity of BLAS 1-2-3

announced that their data-parallel Fortran will support both SIMD and MIMD hardware platforms as well as sequential workstation development environments.

Parallel computers, in particular massively parallel processors (MPPs), claim a potential price / performance advantage as well as a desirable scalability over traditional high performance computers. This claim, unfortunately, depends heavily on the availability of software that is able to use the underlying hardware efficiently. The shortage of such software and the long development time for good compilers for this class of machines, have increased the importance of mathematical libraries that can be used by application programs. Modular software, efficient libraries, and a restructuring of many application codes to take advantage of data parallelism are necessary in order to foster widespread and diverse use of MPP computers. The best known and most important collection of computational kernels is the BLAS [7, 8, 13], proposed originally in 1973 [10]. The functionality and complexity of the 3 levels of BLAS kernels are summarized in Table 1.

The newly released linear algebra library, LAPACK, [1] is entirely built upon the BLAS. It is designed to run efficiently on a wide range of high performance computers, including vector machines and parallel, shared memory computers. Some preliminary figures on the performance of an LAPACK routine on a local memory MPP system are given here.

There is an increase in computational and programming complexity, as well as in the number of (numerically equivalent) ways to formulate a BLAS algorithm, when moving from level 1 BLAS to level 3 BLAS. This richness makes it a formidable task to assess the optimal implementation of the higher level BLAS routines. To minimize the programming effort, it is therefore desirable to program the BLAS in a high level language. This desire conflicts with the assumption made by higher level software; the BLAS should be machine specific and tuned for maximum performance. The wide range of parallel computer architectures and the relatively short lifespan of any computer today, coupled with the long development time of efficient compilers for high level languages, contribute to the slow market acceptance of the technology. The development of a truly mature compiler for this class of computers will take at least two to three hardware generations. One therefore seems forced to develop compute intensive kernels in a more machine specific language, in order to use such systems, while waiting for higher level compilers to achieve sufficient performance. A positive synergy from such an approach may come from the performance goals thus established for the compiler development.

In an attempt to reduce the implementation and tuning effort of the BLAS, Van Loan and Kågström suggested an implementation of the BLAS-3 kernels based on the matrix multiplication kernel DGEMM [12]. We have adopted a similar philosophy, implementing many level 3 BLAS using DGEMM, as well as leaving many of the more specialized BLAS-2 routines in F90. Our current implementation does not exploit symmetry, nor banded or triangular structure of the matrices, resulting in a potential loss of performance. A future improvement to our implementation may be to incorporate the techniques used by Van Loan and Kågström.

In the following sections we try to illustrate these issues with the MasPar MP-1 computer. The machine is in its third year of operation at the time of this report.

2 Two different data-parallel programming languages

This paper reports results and compares algorithms implemented in two different data-parallel languages. The first language, MPL, (MasPar Programming Language) is more machine specific than the partial F90 implementation that is also available. We proceed to describe the two alternatives in some more detail.

The MPL language is a straightforward extension of ANSI C. The extension introduces new parallel data types (ex. "`plural int k`"), a few primitives for permuting data (inter processor communication) and a few statements defining an active set of processors. With these tools, the programmer is responsible for coding his parallel algorithm such that it executes in parallel (i.e. on parallel data). Important issues, which are left to the programmer, include:

1. Which data should be declared as parallel data?
2. How should the parallel data map onto the grid of processors?
3. When and how to send data between processors?
4. Which (sub)set of processors should evaluate the current expression?

Using the data-parallel programming model these are the genuine parallel aspects of any algorithm. The programming model is intuitive and easy to grasp, the programmer writes a single ANSI C program, keeping in mind that all operations involving plural variables will be carried out by all processors that are in the active set. Of the issues listed above, the mapping of data to the processor grid and how to perform communication, causes complications for the programmer that are truly independent of the programming model and very machine specific. The other issues relate more closely to the design of algorithms and can often be addressed without specific knowledge of the target machine hardware.

In F90 the burden of taking these decisions is moved from the programmer to the compiler writer. This, of course, makes the construction of the compiler a much harder task. Below we list the the MasPar Fortran compiler's answers to our questions:

1. Data declared as arrays and used in F90-style is the only parallel data.
2. Cyclic mapping of one and two dimensional arrays whose size is greater than the processor array. For higher dimensional arrays, the next dimensions are mapped to local processor memory. This default can be affected by the programmer via the use of compiler directives.

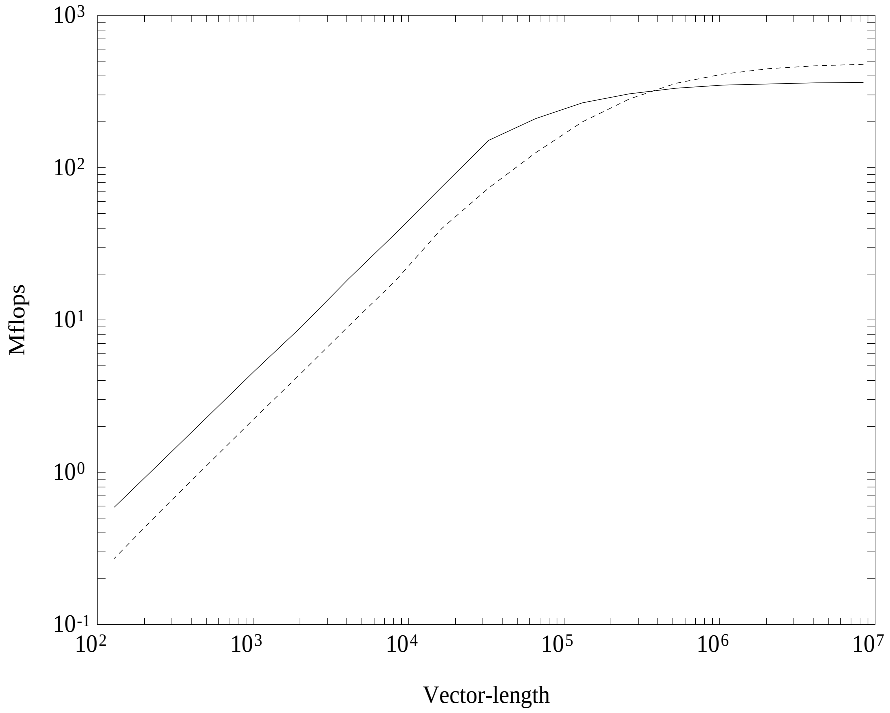

Figure 1: *Performance of DAXPY on an MP-1 with 16k PE. Solid line is F90 and dashed line is MPL implementation.*

3. Communication is implicitly defined when expressing array operations that affect elements stored on different processors, eg. `A(1:10)+A(2:11)`. Communication is also built into intrinsic functions. There are array construction and manipulation functions such as `RESHAPE`, `TRANSPOSE` and `CSHIFT`, which return with data in new allocations, and array evaluation functions like `SUM` and `MATMUL`, where communication is needed, but data is restored to their original locations before returning.

4. By default all processors possessing the data in question take part in the computation. This default may be altered by special control structures such as `WHERE` and `FORALL`.

In the next few sections we will show the difference in implementation and performance of BLAS routines written in F90 and in MPL.

3 BLAS-1, vector - vector algorithms

The BLAS-1 routines are simple vector operations. They emerged in the mid 70's as a way to hide do-loops, structure code, identify vector operations for the then new vector computers, and use registers more efficiently (by way of unrolling) in the mini-computers [10, 13]. The F90 language almost eliminates the explicit need for this class, since these operations now can be expressed in one or at most a few statements. As an example we will consider DAXPY (Double A X Plus Y); scale a vector and add the result to a second vector

$$\mathbf{y} = a\mathbf{x} + \mathbf{y}.$$

In F90 this may be coded as:

```
.........
Y(YLOW:YHIGH:YSTRIDE)= Y(YLOW:YHIGH:YSTRIDE)
                     + A * X(XLOW:XHIGH:XSTRIDE)
.........
```

Ignoring the special code needed to handle short vectors and end effects, the corresponding MPL code may look like:

```
.........
blocks = (yhigh - ylow + 1)/nproc; /* virtualization */
for (i=0; i< blocks; i++)
    *y++ += a* *(x++);
.........
```

In Figure 1 we compare the performance of the two implementations. Note that all figures in this paper will be on a log-log scale, and plot computational rate (Mflops) as a function of problem size. The performance reported is based on elapsed time from invoking the subprogram in question from a main program written in F90, until returning with the result. The numbers therefore include overhead such as subroutine calls and testing of the input data. This overhead constitutes the sequential part of the programs. The difference between the two implementations is that the MPL version needs two extra subroutine calls in order to support a somewhat awkward interface between F90 and MPL.

Returning to Figure 1 we remark that this is an example of a computational kernel without processor communication. For vectors shorter than the number of processors (16384), each participating processor does only 2 arithmetic operations. Thus, for small vectors most of the time spent is in the sequential part of the program. With less overhead, the F90 code of course performs best in this range. For large problem size, the figure reflects local memory bandwidth and a near peak performance. In this range the MPL program achieves slightly better performance, most likely due to more efficient virtualization and better memory access patterns than the F90 generated code.

This figure is quite representative for BLAS-1 performance in general. The F90 implementation performs best for short vector lengths due to the overhead of calling MPL, while the MPL code achieves somewhat better peak performance. This picture was quite different only two years ago, reflecting the steady progress towards an efficient F90 compiler.

4 BLAS-2, matrix - vector algorithms

Many important algorithms in linear algebra can be formulated with a rank one update to a matrix as the inner loop. This operation is therefore one of the most important BLAS-2 kernels [8]. The BLAS-2 routine DGER takes as input one matrix and two vectors. The output is the $m \times n$ matrix, $\mathbf{A}$, updated with the following outer product:

$$\mathbf{A} = \mathbf{A} + \alpha \mathbf{x}\mathbf{y}^T.$$

In F90 this can be done by using the intrinsic function `SPREAD` which expands a one-dimensional array to two dimensions. We apply `SPREAD` to $\mathbf{x}$ and $\mathbf{y}$ along different dimensions, and then multiply the expanded vectors elementwise.

The following code is sufficient:

```
.........
A(1:M,1:N) = A(1:M,1:N) + alpha * SPREAD(X(1:M),DIM=2,NCOPIES=N)
                                * SPREAD(Y(1:N),DIM=1,NCOPIES=M)
.........
```

The corresponding MPL code is:

```
int m,n,i,j,ind,mbloc,nbloc;
double alpha;
plural int iproc,nxproc,nyproc;
plural double a,y,yspread,x,xspread;

mbloc = NBX(m); /* no.  of non-empty blocks of x */
nbloc = NBY(n); /* no.  of non-empty blocks of trans(y) */
.....
/* transpose y */
router[(iproc/nyproc) + (iproc%nyproc)*nxproc].y = y;
/* scaling and spreading out x-values at the DPU */
x *= alpha;
for (i=0; i<mbloc; i++)
    if (iyproc == i) xnetcS[nyproc].xspread[i] = x;
/* Loop over all blocks of A */
ind = 0;
for (j=0; j< nbloc; j++){
    if(ixproc == j) xnetcW[nxproc].yspread = y;
    for (i=0; i<mbloc; i++){
        a[ind] += xspread[i]*yspread;
        ind++;
    }
}
```

The more complicated look of the MPL routine is due to the explicit virtualization, which leads to a need for computing the number of blocks in the layout of $\mathbf{A}$ and looping over these blocks. The code becomes more complicated still when `m` or `n` exceeds the number of processors, so that `x` and `y` are also explicitly virtualized. This is all taken care of by the compiler in the F90 code. Unlike the BLAS-1, the data-parallel implementation of

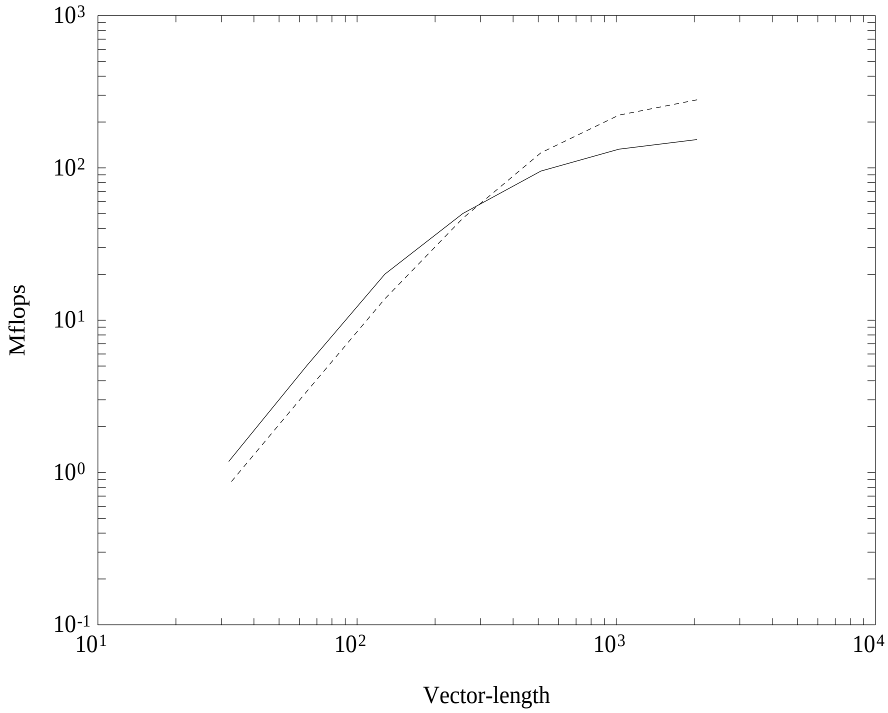

Figure 2: *Performance of DGER on an MP-1 with 16K PE. Solid line is F90 and dashed line is MPL implementation.*

higher level BLAS requires communication. In F90, this is all inside the intrinsic function SPREAD. Note, in particular, how the vector transpose is hidden. In MPL, as can be seen from the code segment, the programmer explicitly controls how the communication is organized. A global `router` is invoked to transpose the **y**-vector, while mesh-based communication primitives (`xnetcS, xnetcW`) are used to copy the vector across the processor mesh. In the MPL code, communication has also been removed from the innermost loop by broadcasting the pieces of the **x**-vector in advance and storing them in a temporary array. This is a typical example of a choice between minimizing the communication or the storage requirement. In MPL, this choice is explicitly under programmer control. In a higher level language this decision is implicitly taken by the construction of the language and/or by the compiler. The F90 code presented above, will normally require substantial temporary storage for the broadcast **y**-vector as well as for the **x**-vector. In fact, the additional memory requirement of F90 may often be of more concern than lack of efficiency, since this may prevent the solution of large scale problems. This class of problems is precisely where the MPP systems should have the best price / performance advantage over alternative technologies.

The performance of the two codes are reported in Figure 2. As for the BLAS-1 routines, the additional overhead makes the MPL version slower for small problems, but now the MPL breaks even for vectors of length a few hundred. The code is significantly faster on problems having a typical size in many applications. This example shows how a fairly simple algorithm already is nontrivial both for the compiler as well as for the MPL

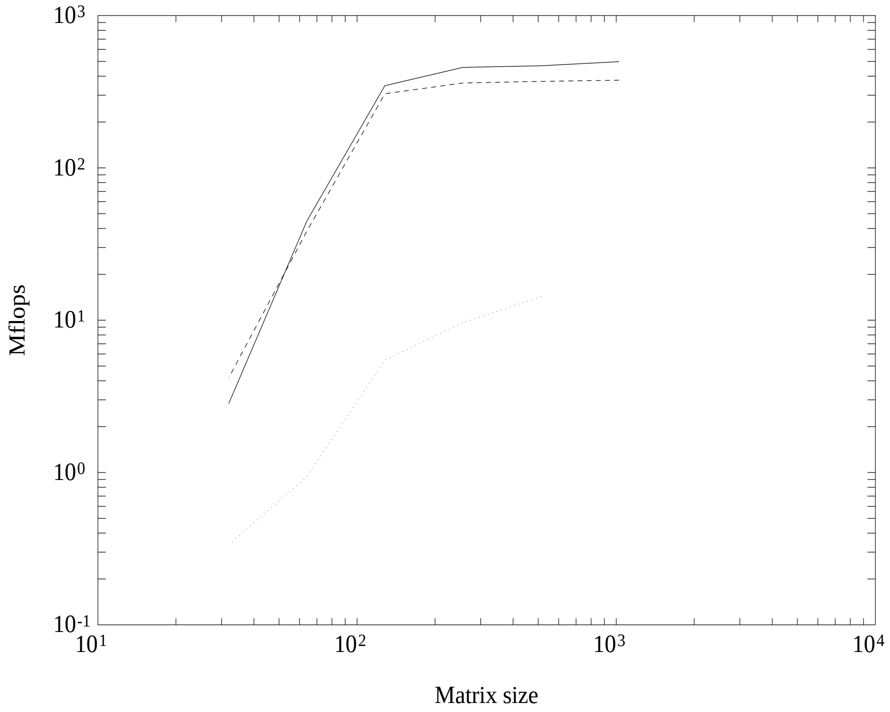

Figure 3: *Performance of DGEMM on an MP-1 with 16K PE. Solid line is F90 with* `MATMUL`*, dashed line is MPL implementation and dotted line is F90 with* `CSHIFT`*.*

programmer. The implementation involves optimal data movement and tradeoffs between communication and storage, questions that cannot be fully answered without knowledge of the invoking program.

5 BLAS-3, matrix - matrix algorithms

The BLAS-3 are the most compute intensive BLAS kernels, their most significant property being the 'volume to surface' ratio between computation and data [7]. Since all BLAS-3 kernels can be implemented using matrix multiplication as a workhorse, we focus on DGEMM (Double GEneral Matrix Multiplication). At the end of this section, we also consider the kernel for the solution of triangular linear systems with multiple right hand sides. This BLAS called DTRSM, is important in many block algorithms and exhibits a significant difference between the two data-parallel languages.

A simple and efficient algorithm for matrix multiplication on a data-parallel computer is the systolic algorithm due to Cannon [4]. This algorithm consists of two parts. In a preprocessing part, 'the preskewing', the data are properly set up for the systolic array. In the computational part, at each step we do one multiplication and one addition, and then shift the two participating matrices one step to the left or up. The following example shows the position of the matrix elements after preskewing for a 4×4 matrix. The A

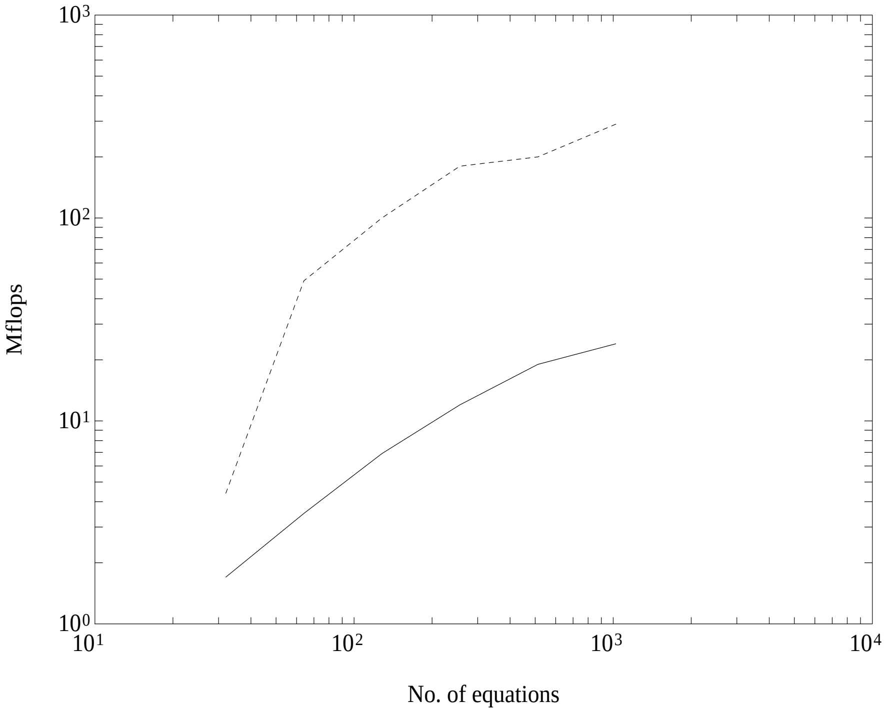

Figure 4: *Performance of DTRSM with 128 rigth hand sides on an MP-1 with 16K PE. Solid line is F90, dashed line is MPL implementation.*

matrix is preskewed row-wise, while the B matrix is preskewed column-wise.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{11} & a_{12} & a_{13} & a_{10} \\ a_{22} & a_{23} & a_{20} & a_{21} \\ a_{33} & a_{30} & a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{00} & b_{11} & b_{22} & b_{33} \\ b_{10} & b_{21} & b_{32} & b_{03} \\ b_{20} & b_{31} & b_{02} & b_{13} \\ b_{30} & b_{01} & b_{12} & b_{23} \end{pmatrix}$$

The computation is then accomplished by the following simple code:

```
C = C + A * B
DO I = 2,N
   A = CSHIFT (A,SHIFT=1,DIM=2)
   B = CSHIFT (B,SHIFT=1,DIM=1)
   C = C + A * B
ENDDO
```

or in MPL:

```
/* NB: No virtualization is included in this figure !  */
c += a*b;
for (i = 1; i < nxproc; i++){
    xnetW[1].a = a;
    xnetN[1].b = b;
    c += a*b;
}
```

In [2] we describe 3 different ways to accomplish the preskewing on the MasPar MP-1. We found it fastest to use the router network. In F90 this might have been accomplished by the following statement:

```
FORALL(I=0:N-1,J=0:N-1)A(I,J) = A(I,MOD(I+J,N))
FORALL(I=0:N-1,J=0:N-1)B(I,J) = B(MOD(I+J,N),J)
```

However, the present version of the compiler is unable to execute this statement in parallel, due to the appearance of the `FORALL` indices in the `MOD` function. Two other preskewing algorithms in can also be programmed in F90 using `CSHIFT` [2]. Only one of them will be executed in parallel by the current compiler, and this unfortunately, turns out to be the slowest one. They therefore all perform poorly.

The F90 version of DGEMM can be coded using the intrinsic function `MATMUL`. The functionality of `MATMUL` does not match that of DGEMM. `MATMUL` computes $C = AB$ while DGEMM computes $C = C + \alpha op(A) op(B)$, where *op* denotes the option whether or not to take the matrix transpose. Explicit calls to another intrinsic, `TRANSPOSE`, are therefore necessary. This forces extra storage as well as potentially more communication. A good implementation of DGEMM in MPL can avoid much of the extra storage as well as the cost of transposition, by combining transposition and preskewing as well as controlling the data flow in Cannon's algorithm. (If both A and B are transposed then no transposition is needed in order to compute C.)

In Figure 3 we report on the performance for the 3 different versions of DGEMM. The main reason for the poor performance of the F90 implementation of the systolic algorithm is the cost of using the intrinsic `CSHIFT` to perform all communication. In MPL, a nearest neighbor communication takes less time than a floating point addition, provided that variables can be kept in registers. This example shows a dramatic performance gap as of this writing, but there is reason to believe that a more advanced release of the compiler should be able to show much better performance, since the algorithm can be well expressed in the F90 language, (extended with the `FORALL` statement). The figure also shows that the F90 intrinsic function `MATMUL` has been highly optimized for performance.

Our last example considers the solution of triangular linear systems of equations with multiple right hand sides, as specified in the BLAS-3 kernel DTRSM. Consider a lower triangular matrix L, having M rows and a right hand side matrix B with M rows and N columns. In F90, the task of solving $LX = B$, with the solution matrix X overwriting B, can be expressed as:

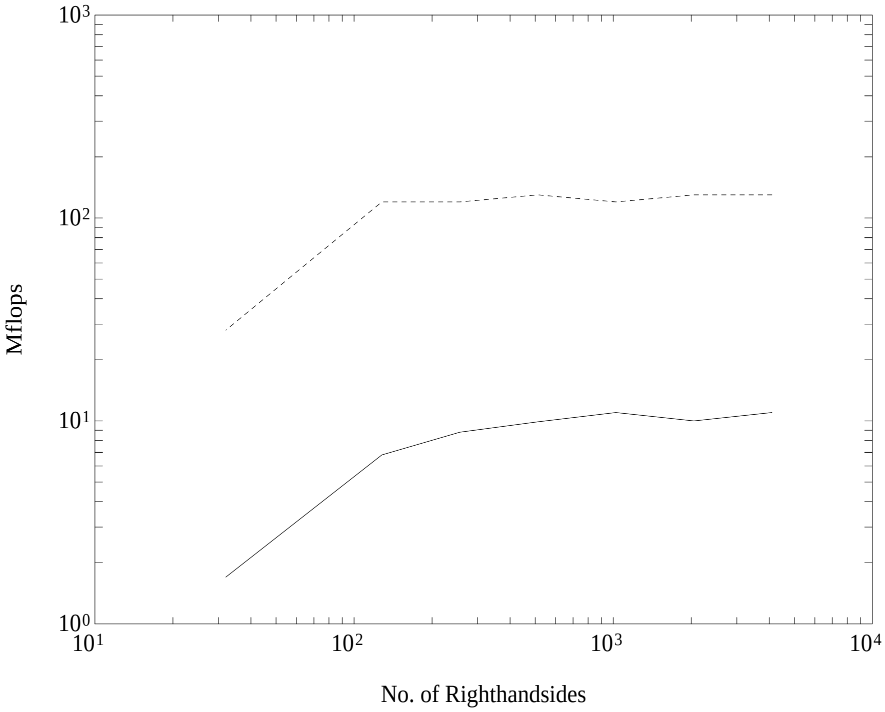

Figure 5: *Performance of DTRSM with 128 equations on an MP-1 with 16K PE. Solid line is F90, dashed line is MPL implementation*

```
DO I = 1,M
   B(I,:)  = B(I,:)  / L(I,I)
   B(I+1:M,1:N) = B(I+1:M,1:N) - SPREAD(L(I+1:M,I),DIM=2,NCOPIES=N)
                               * SPREAD(B(I,1:N),DIM=1,NCOPIES=M-I)
ENDDO
```

We recognize the rank-1 update (DGER BLAS-2 from the previous section), inside the loop. Unfortunately, we do not gain very much performance by this inlining. The rank-1 update involves the last part of the i'th column of L and the i'th row of B. Here, the F90 compiler needs considerably more sophistication in order to minimize communication. More likely, the row in B (corresponding to y^T in the rank-1 formula), will be transposed twice by the compiler. Calling the BLAS DGER is certainly no better, the array section $B(I, 1 : N)$ will be transposed in the call interface, and then transposed a second time, (back to its original shape) explicitly by the DGER routine. However, if the MPL DGER routine is inlined in an MPL DTRSM kernel, then unnecessary communication can be eliminated and a quite efficient implementation results. The performance of DTRSM is shown in Figure 4 where the number of right hand sides N is kept constant at 128, while the dimension of the system M increases. This case can take advantage of a block algorithm, with performance approaching that of matrix multiplication. The MPL routine is implemented this way, explaining the substantial difference from the F90 routine where blocking is not used. In Figure 5 we keep the system size fixed at $M = 128$, and consider

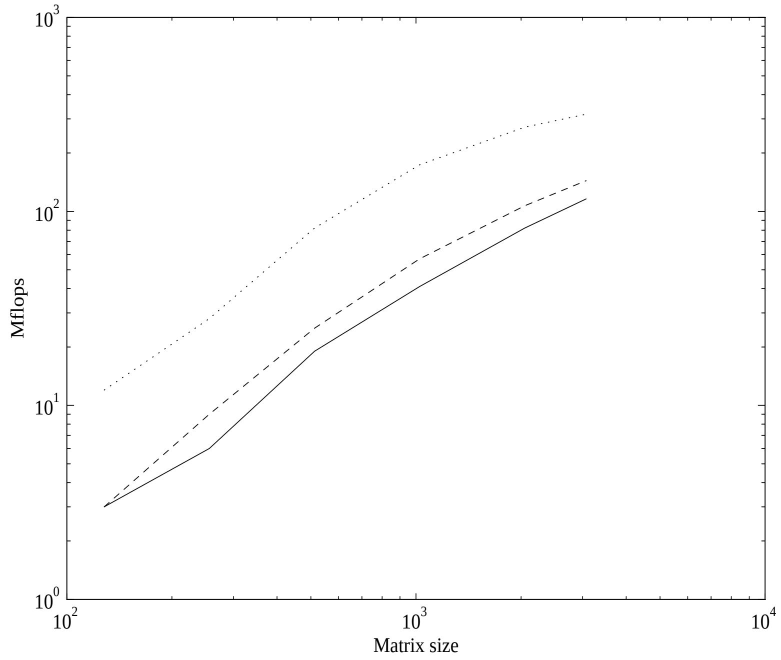

Figure 6: *Performance of DLUBR with 128 equations on an MP-1 with 16K PE. Solid line is F90, dashed line is MPL implementation, dotted line is a hand coded MPL LU-factorization routine for comparison.*

the performance as a function of the number of right hand sides. Again we see a considerable performance advantage for the MPL routine. In this case the performance gap can only be closed with a more advanced F90 compiler.

6 Conclusions

We conclude by showing the performance of the LU-factorization routine DLUBR from LAPACK, using the BLAS routines described in the previous sections. Figure 6 compares the performance of a hand coded LU routine (in MPL) with two versions based on the BLAS modules coded in MPL and F90. The figure shows significant performance gaps between all three versions. This is mainly caused by the same effect as we observed in Section 5 when trying to call a rank-1 update inside a level-3 type routine, but with a different requirement for data layout. DLUBR is a BLAS-3 based routine, meaning that most of the arithmetic is performed by the level 3 BLAS DGEMM. The fraction of work outside DGEMM scales as M/N where M is the block size. This fraction should be compared to the computational speed ratio between DGEMM and the remaining computations. To ensure high performance for a problem of size N (N is restricted by the available memory), the routines DTRSM and the task of doing an LU-factorization on a block column of the matrix must therefore also run efficiently. These two operations can both be formulated in terms of the BLAS-2 kernel DGER. One of the tasks can be written efficiently since it has a separate BLAS definition (DTRSM). There is however, no BLAS kernel that captures the LU-factorization. We discuss LU-factorization on a data-parallel computer using the LAPACK algorithmic approach in more detail in [3]. That work suggests that data-parallel BLAS can form the basis for achieving acceptable performance from libraries like LAPACK, with limited need for redesign of the higher level code (A single block column LU is needed in this case).

The work reported in this paper, has produced two sets of BLAS modules, one written in MPL, the other in F90. This software is available from `netlib@nac.no`. It is hoped that the modules will improve their performance over time, partly by using more sophisticated implementations, but perhaps more from future advances in compiler technology for data-parallel machines. We have tried to focus on a few simple, but important examples to review the current state of art on a widely used MPP computer. We have pointed to several important compiler optimization issues, important work discussing these issues in detail is beginning to emerge. The work [5, 6] considers the problems associated with optimal data layout and evaluation of the corresponding array expressions. It is our belief that this area will see rapid progress in the near future, leading to better compilers and considerable improvements in the performance of data-parallel code generated from a high level language such as F90.

Acknowledgment

The first author thanks Dr. Robert Schreiber for stimulating discussions.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, 1992.

[2] P. E. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic, *Efficient matrix multiplication on simd computers*, SIAM Journal on Matrix Analysis and Applications, (1992).

[3] P. E. Bjørstad and T. Sørevik, *Data-parallel blas as a basis for lapack on massively parallel computers*, in Linear Algebra for Large Scale and Real-Time Applications, 1993. To appear.

[4] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*, PhD thesis, Montana State Univ., 1969.

[5] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng, *Optimal evaluation of array expressions on massively parallel machines*, tech. report, RIACS TR 92-17, 1992.

[6] ——, *Automatic array alignment in data-parallel programs*, in Proceedings of the twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, 1993. To appear.

[7] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms: Model implementation and test programs*, ACM Trans. Math. Soft., 16 (1990).

[8] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An extended set of fortran basic linear algebra subprograms*, ACM Trans. Math. Soft., 14 (1988).

[9] H. P. F. Forum, *High Performance Fortran language specification version 0.4. Draft, Nov. 1992*, Rice University, 1992.

[10] R. Hanson, F. Krogh, and C. Lawson, *A proposal for standard linear algebra subprograms*, SIGNUM, 8 (1973).

[11] A. N. S. Institute., *Fortran 90: X3J3 internal document S8.118 Submitted as Text for ANSI X3.198-1991, and ISO/IEC JTC1/SC22/WG5 internal document N692 Submitted as Text for ISO/IEC 1539:1991, May 1991.*, ANSI, 1991.

[12] B. Kågström, P. Ling, and C. V. Loan, *High performance gemm-based level-3 blas: sample routines for double precision real data*, in High Performance Computing II, M. Durand and F. E. Dabaghi, eds., North-Holland, 1991, pp. 269 – 281.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for fortran usage*, ACM Trans. Math. Soft., 5 (1979).