Automatic Array Alignment in Parallel Matlab Scripts

Igor Z. Milosavljević and Marwan A. Jabri
Computer Engineering Laboratory
School of Electrical and Information Engineering
University of Sydney, NSW 2006, Australia
igor@sedal.usyd.edu.au

Abstract

We present the ParAL system which compiles Matlab scripts into C programs with calls to a parallel run-time library. The novel feature of the compiler is the optimisation of array alignment which reduces or eliminates unnecessary communication overheads. We have evaluated this technique on several Matlab codes. For comparison, the same applications were hand-coded using the PBLAS library. The aligned codes were on average 43% faster then the misaligned codes, with the speedup factor of almost 4 achieved in some cases. This optimisation technique enabled ordinary Matlab scripts to run at a similar speed as manually optimised PBLAS codes.

1. Introduction

Many practical applications can be implemented using matrix operations as building blocks. The need for software infrastructure which supports development of such applications has resulted in many array programming tools and environments. Libraries of subroutines such as LINPACK were among the first such tools, and the Matlab language which originated as a more convenient front-end to such libraries enjoys considerable popularity in the scientific and engineering communities. Languages such Matlab and APL are often described as *very high level array languages*, because they code high level array operations such as matrix product using the same syntax traditional programming languages use for scalar operations.

Very high level array languages are an attractive target for parallelisation as high level array expressions map cleanly onto parallel primitives of the run-time library, and since such libraries have already been implemented with success, e.g., Scalapack [4], the overall good performance should result.

Several projects targeting parallel Matlab exist. Most of them require special code annotations or explicitly parallel instructions [18] (see, [17] for other references) or act as interactive code restructurers [5]. Obtaining parallelism from ordinary scripts is more difficult but is preferable. Some existing systems allow this [2, 17]

However, parallel array primitives require array alignment to avoid *residual communication* overhead [1]. Several techniques for inference of good alignment based on static code analysis have been proposed, but none of the existing systems implement them. One of the reasons for slow acceptance of these techniques is the lack of experimental data with its effects on the performance of real programs.

In this paper we describe ParAL, a paralleliser of ordinary Matlab scripts, which implements alignment optimisation. The effect of alignment optimisation on actual running time of programs was tested on real applications. The parallelised Matlab scripts were compared with implementations hand-coded with PBLAS [3]. The immediate goal of this work is a demonstration of feasibility of automatic alignment with Matlab programs and its experimental evaluation. In addition, we believe that Matlab should be a language of choice for parallel scientific programming, and, this paper intends to give concrete support to this view.

In the rest of the paper, Section 2 gives an overview of the ParAL system, and Section 3 describes the approach to automatic alignment used. The test applications are described in Section 4. The results of performance measurements are presented in Section 5. Finally, the conclusions are presented in Section 6.

2. The ParAL system

The ParAL system takes ordinary Matlab scripts on input, and after several compilation phases produces an SPMD C program with calls to a run-time library of parallel matrix primitives.

In the first compilation stage, the ranks of the variables are determined by performing data-flow analysis of the Matlab script. This is in general a non-trivial task [5, 17], but

in our case it is simplified because we do not allow mutable array shapes after the first time they are fixed. This assumption is also necessary to correctly infer the alignment requirements during static code analysis. As a result, an annotated script is produced for further processing. Subsequently, compound expressions are decomposed into their primitive constituents and temporary variables are introduced. Optimisations are done in this phase, such as elimination of unnecessary temporaries and some peephole optimisations such as operation fusion. The code is not transformed into static single assignment form to avoid unnecessary array temporaries. Most importantly, using the gathered information about variable ranks and the operation semantics, the alignment requirements are recorded in the data-flow graph (DFG). This DFG is passed to the alignment optimisation algorithm which returns the layout parameters of all array variables, including the compiler generated temporaries. Finally an SPMD C program with calls to a run-time library is generated.

ParAL's back-end is a run-time library that implements a set of matrix operations roughly based on the BLAS level 1 and 2 specification. The implementation currently includes double precision general matrix and vector operations. The operation such as matrix-vector multiply and rank 1 update are implemented using standard 2-D grid blocked algorithm as used in PBLAS routines `pdgemv` and `pdger`. BLAS kernels are used for local block computation and BLACS routines [6] are used for communication.

3. Automatic alignment

Background. Automatic partitioning techniques have a large potential to simplify programming, improve portability, and are necessary to deal with compiler generated temporaries which are invisible to the programmer [1]. The problem of finding an optimal data partitioning is an NP-hard problem and development of heuristic solutions has been a subject of much research interest.

High level array languages expose the operation semantics which simplifies the task of code analysis for partitioning. Array alignment is a minimum requirement for locality, and a well designed alignment will reduce or entirely eliminate residual communication. Automatic array alignment has been intensively studied, and many heuristic were proposed (see [1, 13] for review). Implementation of these techniques in current array programming systems is not yet common. The literature mainly offers evaluation based on theoretical estimates and Monte Carlo simulations, so the impact on performance of real code is not well understood. Some experimental work on real code exists, but it focuses on lower-level constructs such as sequential or "forall" loops [7, 9, 15], and not on "pure array" languages.

The alignment problem is modelled as a mapping of data-flow graph nodes (variables) into the space of affine alignments [1]. The GPM algorithm (greedy potential minimisation) [13] is used in ParAL, because it operates on more general data-flow graphs than other solutions, such as compact dynamic programming (CDP) [1], and it was shown that GPM is in many cases more accurate than CDP based on static performance estimates [13]. This paper contributes measurements of run-time performance of real code.

Alignment process. In ParAL, the distributed arrays are partitioned on a 2-dimensional virtual process grid managed by BLACS routines [6]. We concentrate on axis alignment only. Many practical problems, including the ones we benchmark in this paper, have the axis alignment as the only non-trivial component.

On a 2-D grid there are two possible axis alignments: *normal orientation* and *transposed orientation*, denoted here as N and T. Using HPF decomposition directives [11], the possible alignments are as follows.

```
      REAL*8 A(M,N), B(M,N)
      REAL*8 y(M), x(N)
!HPF TEMPLATE TMat(M,N), TVecN(M,Q), TVecT(P,N)
!HPF ALIGN A(I,J) WITH TMat  (I,J) ! normal
!HPF ALIGN B(J,I) WITH TMat  (I,J) ! transposed
!HPF ALIGN y(I)   WITH TVecN (I,:) ! normal
!HPF ALIGN x(J)   WITH TVecT (:,J) ! transposed
```

The input of alignment process is the unannotated code, and on the output one of the above alignment specifications is assigned to each variable in the code.

Each parallel matrix primitive places constraints on the alignment of its arguments. We express these alignments as relationships between the alignment of the input arguments with the alignment of the output arguments. For dyadic operations there are two relationships, and for monadic there is one. Based on GPM formulation [13] these relationships for some of the most frequent primitives are presented in the following table.

Scalar	$y := \alpha x + y$	$\omega_x = \omega_y$	
Inner	$y := Ax$	$\omega_x = \omega'_A$	$\omega_y = \omega_A$
	$y := A'x$	$\omega_x = \omega_A$	$\omega_y = \omega'_A$
Outer	$A := A + xy'$	$\omega_x = \omega_A$	$\omega_y = \omega'_A$

Here, ω_a denotes the orientation of array a (matrix or vector), and ω'_a denotes the opposite of ω_a, that is, if $\omega_a = N$ then $\omega'_a = T$, and vice versa.

The above requirements are added to the data-flow graph (DFG) edges in the form of orientation preferences. Edges are weighted according to array size and frequency of execution (both statically estimated). The resulting annotated DFG, called *orientation preference graph* is passed to the GPM algorithm which returns a configuration of graph node orientations resulting in small cost of residual communication.

4. Test problems

The selected test problems fall into the class of regular dense matrix applications. They may be generally characterised as iterative procedures in which matrix-vector product (MVP) and rank 1 update (RU) are the most compute-intensive operations.

The test problems were implemented in two ways: as Matlab scripts and hand-coded using the PBLAS library. The Matlab scripts were compiled by ParAL wherein they were automatically aligned by the GPM algorithm. The PBLAS implementations were explicitly aligned by hand.

The PBLAS implementations closely follow the Matlab scripts, although compound expressions had to be manually broken down and temporaries introduced. Two PBLAS versions were made for each problem: one with unoptimised alignment (misaligned) and the other with optimised alignment (aligned). The alignment of optimised codes are equal to those chosen by the ParAL compiler for Matlab codes, after inspecting them by hand to make sure that no better solution is possible. The misaligned versions either have default alignment settings (all nodes in N orientation), or they were derived using some other alignment method which yields less efficient results. The following paragraphs give the test problem descriptions.

Matrix-vector product (mvp). This baseline test consists of computing an MVP $y = Ax$ in a loop, and it compares the PBLAS `pdgemv` routine to the corresponding MVP routine from ParAL's back-end. The code is perfectly aligned if $\omega_x = T$.

Power method (power). Hotelling's power method is often used in principal component analysis (PCA) [8]. The code has alignment conflicts. All variables are set to position N in the aligned code, which is the solution found by GPM. A straight longest spanning tree solution [10] was used as an example of misaligned code.

Simple PCA (spca). The SPCA algorithm [14] is another method for principal component analysis. Perfect alignment is possible. All arrays are set to N in the misaligned code.

Biconjugate gradient (bicg). The biconjugate gradient algorithm is an iterative method for solution of systems of linear equations. Our implementation is based on the NRF90 routine `linbcg` [16]. This algorithm has alignment conflicts, and GPM solution sets all variables to position N, same as the default setting. As the misaligned case, we choose an example of local optimisation with poor results, obtained by aligning one of the MVP operations by placing its vector argument into position T.

Error back-propagation learning (mlpbpr). The error back-propagation algorithm and its many variants is the most frequently used method for "training" artificial neural networks (ANNs) [12]. This test implements the on-line variant of the algorithm. Perfect alignment is possible. All arrays are set to N in the misaligned code.

5. Experimental results

Environment. The test problems were run on two parallel systems: a 64-node CM-5 machine and a 12-node cluster of Alpha workstations (COW). The problems were run on the maximal number of processing elements with varying problem size. The processor grid aspect ratio is set to match the matrix aspect ratio. Most of the test problems run with square matrices except that the aspect ratio of **spca** is 1×16 and of **mlpbpr** is 1×4.

The running time of the programs is the measured variable. On the Alpha cluster the measurements are based on wall clock time, repeated and averaged until small variance was achieved. CM-5 has a system routine for measuring the time slice allocated to a process, which includes communication overhead.

Results and discussion. From the measured run time t we calculated the absolute speed as $f(n) = \frac{N_{op}}{t}$ flop/sec, where N_{op} is the operation count. For each problem, we distinguish three absolute speed for different implementations: f^*_{matlab} is the speed of an aligned Matlab program, f^*_{pblas} is the speed of an aligned PBLAS program and f_{pblas} is the speed of a misaligned PBLAS program.

The absolute speeds of Matlab scripts is given in Figure 1. These plots also show the theoretical estimate valid for a single MVP and RU operation with a square $n \times n$ matrix on a $P \times P$ process grid. The running time estimate is

$$\begin{aligned} t &= t_{\text{comp}} + t_{\text{comm}} \\ &= \frac{n^2}{P^2 F_{cpu}} + 2\left(T^v_{com}\frac{n}{P} + T^0_{com}\right)\lceil \log_2 P \rceil . \end{aligned}$$

The parameter F_{cpu} is the speed of serial BLAS `dgemv` subroutine on one machine node. The communication time is for the tree broadcast/combine implemented in BLACS, where T^v_{com} and T^0_{com} are constants, representing the variable and fixed components of message transfer time. These parameters are measured by timing the message transfer between two neighbouring nodes and performing a linear regression. The measured values of these parameters are as follows.

	F_{cpu}	T^v_{com}	T^0_{com}
CM-5	1.05 Mflops	1.35 μsec	89.20 μsec
Alpha	5.50 Mflops	13.70 μsec	1444.00 μsec

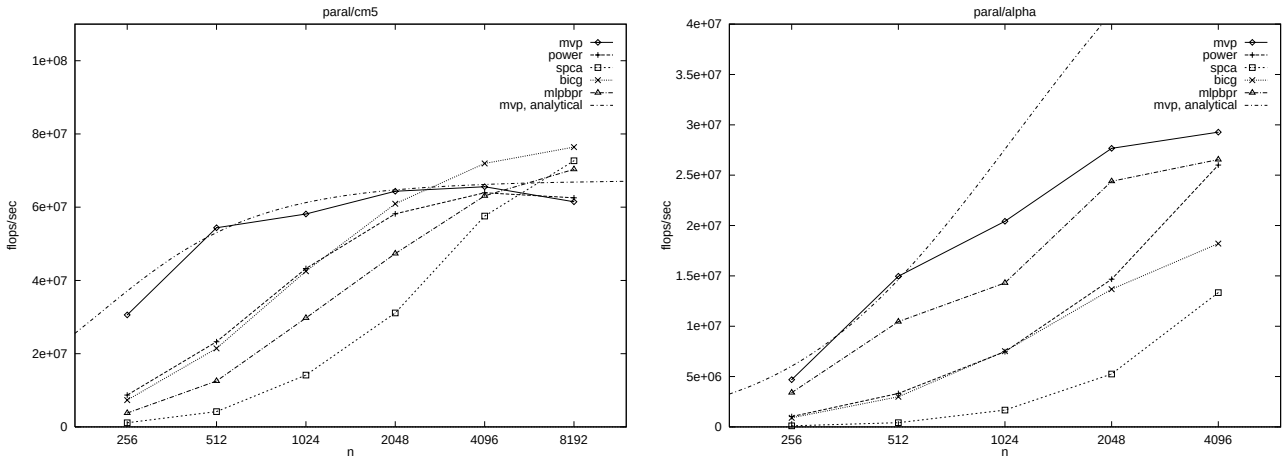

Figure 1. Absolute speed of Matlab scripts compiled by ParAL.

`dgemv` was compiled with Sun/Digital `f77 -O`. The correlation coefficient for the communication times was 0.99.

The performance gains resulting from optimised alignment are indicated by the relative speeds f^*_{matlab}/f_{pblas} and f^*_{pblas}/f_{pblas}, summarised in Table 1.

The ParAL performance on CM-5 shows good efficiency, compared to the theoretical estimate. This is expected for a machine with a high computation-to-communication speed ratio. The tests **bicg** and **mlpbpr** even exceed the theoretically predicted performance. The reason is that they contain operations of the form $y = A'x$ which are translated into calls to `dgemv` with parameter `trans='T'`, giving about 50% higher serial performance than F_{cpu} used in the theoretical curve, which was measured with `trans='N'`. This is due to the implementation of the Sun `f77` compiler.

The efficiency of ParAL on the Alpha cluster is significantly below the theoretical estimate. One of the reasons is that the point-to-point communication speed affected by the overall network traffic, and, as we did not have dedicated access to the cluster during the measurements, that peak local computation speed was rarely reached on all machines at the same time. However, the results are still meaningful as a comparison between different implementations run under same conditions.

The speed achievable with ParAL using ordinary Matlab code is in most cases very close to hand-coded PBLAS implementation, as shown by the similarity of figures for ParAL and PBLAS columns in Table 1. This is expected, since essentially the same parallel algorithms are used to implement the key operations MVP and RU.

6. Conclusions

We have presented the ParAL system for parallelising Matlab scripts that implements automatic alignment optimisation. The resulting code showed performance comparable to hand-coding using a state-of-the art parallel library. The overall speed increase due to alignment optimisation was measured to be 43% on average, with maximum speedup of 3.89. Average improvement on the Alpha cluster was 35% and 50% on the CM-5.

Alignment is often difficult to do by hand, even with medium-sized programs like some of the test problems addressed in this paper. Our Matlab compiler is in a preliminary development stage, allowing only relatively simple programs. We envisage more complex alignment relationships for larger applications in which case automatic alignment would be practically necessary.

Although currently implemented within ParAL, these results are applicable to other languages, especially to other parallel Matlab compilers.

Using Matlab instead of directly coding with a subroutine library has the advantage of a more convenient interface, and this is why Matlab is very popular on uniprocessor platforms. Translating Matlab scripts into subroutine calls can be done efficiently, and this is done successfully on uniprocessors. However, parallelisation requires additional optimisation techniques to preserve the efficiency. The alignment technique shown here as feasible and resulting in significant performance gains is one such technique which brings closer the use of Matlab to parallel computers.

The full sources of Matlab and PBLAS implementations and additional information can be found on the worldwide web, www.sedal.usyd.edu.au/˜igor/paral.

Table 1. Speed increase due to optimised alignment. Table shows relative speed of aligned ParAL and PBLAS code vs. misaligned PBLAS code.

	CM-5						Alpha cluster					
	ParAL, f^*_{matlab}/f_{pblas}			PBLAS, f^*_{pblas}/f_{pblas}			ParAL, f^*_{matlab}/f_{pblas}			PBLAS, f^*_{pblas}/f_{pblas}		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
mvp	0.99	1.42	1.13	**0.98**	1.19	1.06	1.16	**3.11**	1.97	1.12	**3.89**	**1.95**
power	**0.97**	1.11	1.03	1.00	1.11	1.04	**0.81**	1.33	1.09	1.05	1.52	1.21
spca	2.21	**3.32**	**2.75**	2.02	**3.15**	**2.55**	0.95	1.80	1.19	0.93	1.68	1.11
bicg	1.04	1.26	1.16	1.03	1.20	1.09	0.94	1.29	1.06	**0.81**	1.31	1.17
mlpbpr	1.15	1.96	1.61	1.38	1.75	1.63	1.00	1.90	**1.42**	1.16	1.76	1.34
all	0.97	3.32	1.54	0.98	3.15	1.47	0.81	3.11	1.35	0.81	3.89	1.35

7. Acknowledgements

This work was sponsored by Australian Postgraduate Awards and the Department of Electrical Engineering of University of Sydney. The authors thank Matthew Partridge and Rafael Calvo for help with implementing the PCA algorithms and the New South Wales Centre for Parallel Computing for granting time on its facilities to support this work.

References

[1] S. Chatterjee et al. Automatic array alignment in data-parallel programs. In *ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 16–28, Jan. 1993.

[2] W.-M. Ching and A. Katz. An experimental APL compiler for a distributed memory parallel machine. In *Proc. Supercomputing*, Washington, D.C., Nov. 1994.

[3] J. Choi et al. A proposal for a set of parallel basic linear algebra subprograms. Lapack Working Note 100, U. of Tennessee and ORNL, May 1995.

[4] J. Choi et al. ScaLAPACK: A portable linear algebra library for distributed memory computers – design issues and performance. Lapack Working Note 95, U. of Tennessee, ORNL and UCB, 1997.

[5] L. DeRose et al. FALCON: a MATLAB interactive restructuring compiler. In *Proc. 8th Int'l Workshop Languages and Compilers for Parallel Computing*, 1995.

[6] J. J. Dongarra and R. C. Whaley. A user's guide to the BLACS v1.1. Lapack Working Note 94, Dept. of Computer Sciences, Univ. of TN, Knoxville, TN, May 1997. Available at www.netlib.org.

[7] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 3(2):179–193, Mar. 1992.

[8] I. T. Jolliffe. *Principal Component Analysis.* Springer Series in Statistics. Springer-Verlag, New York, 1986.

[9] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. Supercomputing*, San Diego, CA, Dec. 1995.

[10] K. Knobe, J. D. Lukas, and G. L. Steele, Jr. Data optimization: allocation of arrays to reduce communication on SIMD machines. *J. of Parallel and Distributed Computing*, 8:102–108, 1990.

[11] C. Koelbel et al. *High Performance Fortran HandBook*. MIT Press, Cambridge, MA, 1994.

[12] R. P. Lippmann. Pattern classification using neural networks. *IEEE Communications Mag.*, pages 47–64, Nov. 1989.

[13] I. Z. Milosavljević. An improved algorithm for array alignment in data-parallel programs. *J. of Parallel and Distributed Computing*, 1998. Accepted for publication.

[14] M. G. Partridge and R. A. Calvo. Fast dimensionality reduction and simple PCA. *Intelligent Data Analysis*, 2(3), 1998.

[15] M. Philippsen. Automatic alignment of array data and processes to reduce comunication time on DMPPs. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 156–165, Santa Barbara, CA, July 1995.

[16] W. H. Press et al. *Numerical Recipes in Fortran 90.* Cambridge University Press, second edition, 1996.

[17] M. J. Quinn, A. Malishevsky, and N. Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *Proc. 7th Int'l Symp. High Perf. Distrib. Computing (HPDC)*, Chicago, IL, July 1998.

[18] S. Ramaswamy, E. W. Hodges IV, and P. Banerjee. Compiling Matlab programs to ScaLAPACK: Exploiting task and data parallelism. In *Proc. 10th Int'l Parallel Processing Symp. (IPPS)*, Honolulu, HI, Apr. 1996.