High Performance Principal Component Analysis with ParAL

Igor Z. Milosavljević Matthew G. Partridge Rafael A. Calvo*
Marwan A. Jabri

SEDAL, Dept of Electrical Engineering, University of Sydney
{igor,mp,rafa,marwan}@sedal.usyd.edu.au

ABSTRACT

Principal Component Analysis (PCA) is a technique which can reduce the dimensionality of high dimensional data. The PCA algorithms must process high volumes of data quickly, and using parallel processing to obtain high computation speed is a natural choice. We present parallel implementations of two PCA algorithms written in ParAL — a high-level array programming language. ParAL provides a simple interface to parallel computation which is suitable for use by non-experts, and the tests show that the efficiency of the resulting code is encouraging. We have implemented and tested the SPCA and Hotelling's Power method algorithms on a cluster of workstations and a CM-5 supercomputer, contributing to high performance PCA.

1. Introduction

PCA has many applications including as a preprocessing step for machine learning methods such as neural networks. In some sense it also competes with neural networks in that both methods try to perform feature extraction. In very high dimensional data settings, feature extraction is crucial for both speed and accuracy of learning algorithms. However, it is in these settings that it is also important that PCA be performed quickly — its parallelisation is a natural approach. This paper seeks to extend the speed of PCA through parallelism, and to show how methods such as PCA may be simply and efficiently parallelised.

Parallel computing is a method of increasing the computing speed by dividing the computation over several processing nodes and running those portions in parallel. However, parallelism introduces an overhead of communication between processing nodes needed to synchronise and exchange data. This overhead is significant, and can even result in parallel speed being lower than the serial speed. Achieving good parallel performance requires a partitioning of computation such that unnecessary communication is avoided.

In general, there are two opposing approaches to obtain an efficient parallel program. A low-level approach is where the programmer takes the full control in managing the parallelism, for example, by using the programming interfaces provided by libraries such as PVM [14]. This method pays off in terms of performance, but it requires expertise, and is labour intensive. The other method is to write high-level code which does not address the parallelism directly but relies on the compiler and other tools to produce efficient low-level parallel code. The second method is preferable for the majority of users, but the compilers and tools necessary for its practical application are still unsatisfactory, and this area is a subject of intensive research.

Whilst general purpose parallelising compilers are not currently available [2], some classes of problems are more suitable for automatic parallelisation. One such group of problems are the algorithms which can be expressed in terms of array operations. Array operations are naturally parallel, so the compiler has a much simpler task in mapping a sequence of such operations into efficient low-level parallel code. The algorithms for PCA that we discuss here fall into this class, and we present their implementation using the Parallel Array Language (ParAL) — a programming framework for array algorithms

*Permanent address: Instituto de Fisica Rosario, Bvd. 27 de Febrero 210 Bis, 2000 Rosario, Argentina

suitable for non-expert users [11].

2. PCA

PCA has been developing since the beginning of this century as one of many dimensionality reduction techniques. In mathematical terms, n correlated random variables are transformed into a set of $d \leq n$ uncorrelated variables. These uncorrelated variables are linear combinations of the original variables and can be used to express the data in a reduced form. Traditionally the PCA of a data set was performed by estimating the covariance matrix of the data and diagonalising it. As a result, a set of eigenvalues and eigenvectors were obtained. The eigenvectors can be interpreted as the directions where the variance of the data set is maximised.

The techniques that perform an explicit estimation of the covariance matrix are grouped as *matrix methods*: these include Singular Value Decomposition (SVD) and Hotelling's Power method. Alternatively, *data methods* do not need an explicit computation of the covariance matrix [8], and include Hebbian neural networks and SPCA methods [13]. In an accompanying paper [3], different PCA techniques are described in more detail and compared.

The present paper considers only the bases of the two fastest methods for finding a few of the possible principal components, namely the Hotelling's Power method (for covariance matrices) and SPCA_2 (for direct data analysis).

3. Simulation Environment

3.1. Overview

As a short introductory example, consider the following ParAL program which reads a vector, multiplies it with a randomly generated matrix, then adds it to the fourth column of the matrix.

```
operator $.(%inner, "&local_inner_mvp");
read m, n;
array A[m,n], x[n];
read x;
A = $rand(A);
y = A $. x;
z = A(:,3) + x; # indices start from 0
```

The '`operator`' statement in the first line declares '`$.`' to be of the inner matrix-vector operation *class*, and binds it to the function which does the matrix-vector product on local blocks of a distributed array. In other words, '`$.`' is declared as the matrix-vector product operator.

The function '`local_inner_mvp`' is included in the basic ParAL back-end, but '`$.`' is not defined by default, because the user will often want to use a specialised local operation to gain performance.

The ParAL language is rich enough to enable simple coding of PCA algorithms observed in this paper. The full SPCA code is presented in Appendix A.

3.2. Architecture of ParAL

The ParAL system consists of two layers: (1) the front-end, which translates the ParAL source into a series of ANSI C calls to a library of lower-level array primitives; and (2) the back-end, which is a library of parallel array primitives mapped onto a virtual grid of processors.

Front-end translator. The C code resulting from the front-end translation is a series of calls to routines from the back-end library of array primitives, generated from the parse tree of the input code. The translator performs compile-time checks of consistency, and some optimisations, such as array alignment and reduction of number of temporaries. Related work in this area was done for APL [4] and MATLAB [7] array languages.

Back-end library. The back-end primitive library is a general purpose library for parallel array computation built on top of the BLACS library [1]. Related approaches are the ScaLAPACK [5] and PLAPACK [17] projects.

The back-end is designed in an object-oriented fashion, and contains various primitives for array manipulation. The computational primitives are designed in a *skeletal* fashion [6], so the user can instantiate new primitives by passing pointers to serial routines that perform the local array manipulation (operations on array blocks stored locally on a computing node).

Because the back-end is built on top of BLACS, ParAL codes are portable to all architectures supported by BLACS, including PVM clusters, IBM SP and Intel Paragon.

The back-end implements two parallel mappings of matrix-vector operations: 2-D blocked (mesh) [9] and ring-systolic [10].

Fig. 1: Sample images from the ORL database.

4. Experiments

4.1. The Compression Problem

We have addressed the problem of image compression on the ORL database of face images [12]. A sample of images is given in Figure 1. As a test case, we extract the minimum number of eigenvectors that explain 80% of the total variance. The ORL database consists of 400 images of 92×112 pixels, which gives rise to 10304-dimensional data points. Using both Power method and SPCA with 20 iterations per principal component (eigenvector), it takes 44 components to explain 80% of the variance.

Note that the covariance matrix for the ORL database is of dimension 10304×10304, which precludes the execution of the Power method on a single workstation. In this case, parallelism is not only a matter of speedup, but is also necessary to provide the memory capacity.

4.2. Target Machines

We have run ParAL codes on two architectures: a COW (Cluster Of Workstations), and a CM-5 supercomputer.

The parallel computing on COWs has recently become a topic of wide interest. Considering the drop in price and increase in performance of commodity computing components, COWs provide very affordable access to parallel computing. The COW system that we used consists of 50 DEC Alpha stations at the University of Sydney Faculty of Engineering. This facility is primarily intended as for student use, not for running parallel applications.

The CM-5 machine is a 128 node installation at the University of Adelaide. The ParAL back-end addresses only a small fraction of the peak computing power of the CM-5 system, because it does not utilise the vector units, rather, the code is mapped onto the control processors only, which are the Sparc 2 architecture [15]. However, the CM-5 system has the advantage of ample memory resources required for using the Power method.

Table: 1: Speed of serial matrix-vector product.

Matrix Size	Speed [MFLOPS]	
	Alpha	CM-5
100×100	9.1	13.3
2000×2000	4.0	4.5

4.3. Performance Metrics

The basic performance metric that we have measured is the execution time. For the Alpha cluster, the execution times reported are real (wall clock) times. Each test was repeated 10 times, and the average is reported. The tests were done at the time when the load by other users was low, so the wall clock timings accurately reflect the performance. The standard deviation was under 2% for all of the measurements. The CM-5 timings were gathered using the `CMMD_node_timer_elapsed` function [16].

The derived metrics are: *absolute speed* in FLOPS, computed as number of multiply-accumulate operations divided by time, *speedup* $S(P) = T(P)/T(1)$, where $T(P)$ is parallel time on P nodes and $T(1)$ is serial time, and *utilisation* $U(P) = S(P)/P$.

Because the matrix-vector product is the operation in which the observed PCA algorithms spend most of the execution time, the speed of a serial matrix-vector product routine on a single machine is useful for comparison. These speeds are given in Table 1.

The ParAL back-end library (including the local matrix-vector product primitive) was compiled using the `gcc` compiler with '`-O4`' optimisation level.

Table: 2: Power method performance.

System:	Alpha COW	CM-5
Nodes	32	128
	(4×8)	(8×16)
Time [sec]	1347	1075
Speed [MFLOPS]	101	126
Utilisation	78%	82%
Speedup	25	105

Table: 3: SPCA performance on the Alpha cluster for the ORL problem using the mesh matrix-vector operation mapping.

Nodes P	Time $T(P)$	Speedup $S(P)$	Utilisation $U(P)$
1	1488 sec	1.0	100%
5	384 sec	3.9	78%
10	313 sec	4.8	48%
15	273 sec	5.5	37%
20	285 sec	5.2	26%

5. Results

5.1. Power Method Performance

The Power method was tested both on CM-5 and COW. Due to the largeness of the covariance matrix, the startup and execution times were rather long, therefore we have performed only a few tests. The results are summarised in Table 2.

5.2. SPCA Performance

The SPCA algorithm was run on the Alpha cluster only. Some representative measurements are reported in Table 3.

To study the scalability and the effect of the choice of the matrix-vector operation mapping (mesh or ring), we have conducted a series of tests with increased database size. In addition to the normal 400 images database, we have measured the performance for database sizes of 800 and 1600; the enlarged databases were obtained by duplicating the existing data points, and the test consisted of computing some fixed number of eigenvectors, ignoring their significance. The results are summarised in Figure 2.

The larger data sets give a more scalable execution, because the communication overhead is dominated by a fixed message start-up time, which is amortised when more computations are scheduled onto each node. Beyond a certain number of processors, the parallel speed starts to decrease. The reason is that when many hosts access the Ethernet LAN at the same time, the communication becomes slower due to access collisions.

It is interesting to note that with 400 data point problem, the ring mapping performs better than mesh mapping for certain processor array sizes. For larger data set sizes, the performance of both mappings is almost identical. Due to its systolic nature, the ring mapping may be able overlap some communication and computation, which is planned for future work.

6. Conclusions

This paper improves the speed and feasibility of the fastest known methods for performing PCA through their parallel implementation. Perhaps more importantly, however, it illustrates how serial array-based algorithms may be easily implemented using ParAL, a newly developed parallel programming system, yielding parallel code without using knowledge about parallelism.

A. ParAL SPCA Code

Due to space limitations, only the code for the SPCA algorithm is presented. The Power method code is very similar.

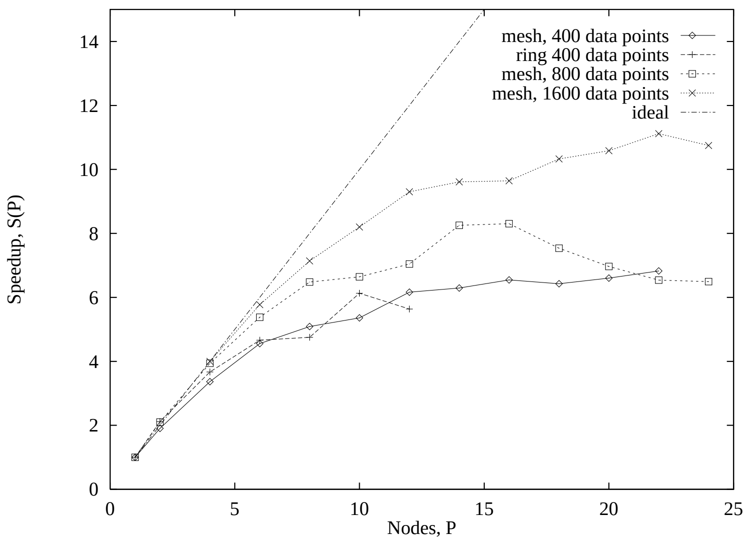

Fig. 2: Scalability of SPCA on the Alpha cluster. The data dimensionality is the same as in the ORL database. Only the results for the mesh mapping are shown where differences between mesh and ring are small.

.

```
# Read data set and parameters
array D[0,0];
M = 0; N = 0;

inlinec 'read_data_set("db_orl/files", &D, &M, &N)';
read max_iter_count, var_thresh;

array row[N], column[M]; # auxilliary vectors

# Mean-correct the data
column = $ones(column);
mean~ = $oneover(M) * column~ $. D;
print "mean=", $l2norm(mean), "\n";
D = -column $r2u mean~;

# Compute total variance
row = $ones(row);
tot_var = $sum($square(D) $. row);

# start SPCA iterations

exp_var = 0.0; # explained variance
evnumber = 0;  # number of eigenvectors computed

array a[N];

while (exp_var < var_thresh * tot_var) {
    a = $rand(a);
    for (i = 0; i < max_iter_count; i = i + 1) {
newa = D~ $. (D $. a);
lam = $l2norm(newa);
newa = newa * $oneover(lam);
a = newa;
    }
    evnumber = evnumber + 1;

    # Update explained variance
    exp_var = exp_var + lam;

    print "vector: ", evnumber,
          " exp_var = ", exp_var, "\n";

    # Deflate data set
    D = -(D $. a) $r2u a~;
}
```

Notes. The '`$.`' operator is the matrix-vector product. The '`$r1u`' operator is the rank 1 update: '`A = x $r1u y~`' means $A \leftarrow A + xy'$. Note that '`~`' denotes matrix transpose. There are no primitives that perform reduction on matrices, so this must be done by multiplying a matrix with a vector of ones, such as with computation of *mean* and *tot_var*.

References

[1] BLACS: Basic Linear Algebra Communication Subprograms. URL http://www.netlib.org/-blacs/index.html (retrieval date: Dec. 1997), Dec. 1997.

[2] W. Blume et al. Automatic detection of parallelism – a grand challenge for high-performance computing. *IEEE Parallel & Distributed Technology*, 2(3):37–47, Fall 1994.

[3] R. A. Calvo, M. G. Partridge, and M. A. Jabri. A comparative study of principal component analysis techniques. In *Proc. Ninth Australian Conf. on Neural Networks*, Brisbane, QLD, Feb. 1998.

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

[5] 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.

[6] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, MA, 1989.

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

[8] K. I. Diamantaras and S. Y. Kung. *Principal Component Neural Networks*. Wiley, New York, 1996.

[9] G. C. Fox et al. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

[10] J.-N. Hwang and S.-Y. Kung. Parallel algorithms/architectures for neural networks. *Journal of VLSI Signal Processing*, 1(1):221–251, 1989.

[11] I. Z. Milosavljević. ParAL: A parallel array language. Technical report, SEDAL, Dept. Elect. Eng., University of Sydney, 1998. In preparation.

[12] The ORL Database of Faces. URL http://www.orl.co.uk/facedatabase.html (retrieval date: Dec. 1997), Apr. 1994.

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

[14] Parallel Virtual Machine. URL http://www.epm.ornl.gov/pvm/-pvm_home.html (retrieval date: 26 May 1997), May 1997.

[15] Thinking Machines Corp. *The Connection Machine CM-5 Technical Summary*, Jan. 1992.

[16] Thinking Machines Corp. *CMMD Reference Manual*, May 1993.

[17] R. van de Geijn et al. PLAPACK: Parallel linear algebra package. URL http://www.cs.utexas.edu/users/plapack/ (retrieval date: 14 Nov. 1997), Nov. 1997.