High-Performance Linear Algebra

P. R. Briddon, *University of Newcastle-Upon-Tyne, and*
Adrian P. Tate, *CSAR, University of Manchester*

ABSTRACT: *In a distributed memory parallel programming environment linear algebra operations are performed using libraries such as ScaLAPACK[1]. Here we will outline an example of a code where a ScaLAPACK eigensolver routine directly inhibits performance. We will detail the approach taken to replicate the functionality of ScaLAPACK routine PDSYTRD whilst increasing the performance significantly, and thus improving the scalability of the code in question. We will detail the mathematical and programming based techniques used to achieve the high performance and will outline the direction of future work in this area.*

1. Introduction

AIMPRO is a widely used physics code, used for first-principles simulation of material properties [2]. In solving the Schrödinger equation for the electronic structure of molecules or solids, the standard real symmetric matrix eigenproblem A x = λ x must first be solved. Here A is an n×n matrix. This equation occurs when solving a partial differential eigenvalue equation using a basis set expansion and occurs in many applications in science or engineering. The standard approach for dense matrices and when a substantial proportion (e.g. 10%) of the eigenvectors are needed is to reduce the matrix to tridiagonal form. The eigenvalues/eigenvectors of this matrix are then more easily found. This is done in (e.g.) LAPACK. Here we will discuss efficiency of tri-diagonalisation using the householder method. For an explanation of this method, and a description of tridiagonal matrices, see [3].

2. Serial Implementation

```
do i=n-1,1,-1
    !setup elementary reflector x(1:i),O(i) work

    y(1:i) = dsymv ( a(1:i,1:i), x(1:i) )
    ! symmetric matrix vector multiply, O(i^2) work

    !...O(i) work generating vectors u(:) and v(:)

    call dsyr2( u(1:i), v(1:i), a(1:i, 1:i) )
    !  a_ij = a_ij +u_i*v_j - O(i^2) work
enddo
```

Figure 1 LAPACK tridiagonalisation

The design philosophy of ScaLAPACK and LAPACK is to use level 3 BLAS wherever possible, but previous implementations such as EISPACK used level 1 BLAS. Figure 1 shows the structure of the LAPACK code *dsytd2*, focusing only on the main computational ingredients which are calls to level-2 BLAS routines *dsymv* and dsyr2. Here routines *dsymv* and *dsyr2* both contribute $2n^3/3$ flops giving a total operation count of $4n^3/3$.

Level 3 BLAS routines use matrix blocking to improve cache re-use, hence the LAPACK routine *dsytrd* replaces a number of calls to dsyr2 with a single call to the level 3 routine *dsyr2k*. Calls to *dsymv* remain. The resultant performance is therefore intermediate between level 2 and level 3 BLAS speeds (but closer to the level 2 performance).

3. Parallel Implementation

ScaLAPACK builds on the LAPACK design philosophy and parallelisation is achieved largely by using parallel versions of the underlying BLAS routines. The ScaLAPACK tridiagonalisation routine *pdsytrd* therefore utilises the parallel BLAS (PBLAS) routines *pdsymv* and psyr2k. Unfortunately routine *pdsytrd* performs badly. This is due (a) to the poor performance of pdsymv on one node, (b) a load imbalance occurs as the preparation of the matrix for routine pdsyr2k, itself scaling poorly, only takes place on a column of processes. Scaling of the routine drops off even on 16 processors (see later Figure 8). Results herein are taken from runs on a 512 processor SGI Origin 3800 system, upon which the majority of this work was carried out.

In some runs of AIMPRO, matrices of dimension of order 2000 are repeatedly diagonalised on a large number of processors, and ScaLAPACK is not sufficient or appropriate for this purpose.

4. Modified coding strategy

An alternative parallel tridiagonal solver could follow the approach shown in Figure 2, where the level 2 BLAS implementations *dsyr2* and *dsymv* are used instead of their level 3 counterparts, but *dsyr2* is removed from the end of one previous iteration and fused with *dsymv* at the start of the next. This delayed update of the matrix exposes more flops per element of matrix a(i,j) loaded into cache, enabling the serial performance to be boosted. The disadvantage is that a special "blas" routine must be handwritten to achieve this.

```
do i=n-1,1,-1

    if(i==n-1)then
    y(1:i) = dsymv ( a(1:i,1:i), x(1:i) )
    else
    call dsyr2( u(1:i), v(1:i), a(1:i, 1:i) )
    y(1:i) = dsymv ( a(1:i,1:i), x(1:i) )
    endif
    …
    do dsyr2 !BUT ONLY FOR COLUMN "i" of a
(remainder done in next iteration).

   enddo
   call dsyr2 (just for the last iteration)
```

Figure 2 Fusing BLAS for better cache re-use

The resultant routine gives a serial performance close to the LAPACK equivalent without the need for level 3 BLAS. Hence any parallelisation begins at the same initial performance. The efficiency of parallelisation will now depend firstly on how well the handwritten "blas" routine scales and also on how any additional communication operations perform.

1	0.000129	entry	10	0.000359	pdsyr2
2	0.004264	reflector generation	11	0.104754	summation
3	0.019923	vector copies	12	0.001907	entry
4	0.407934	bcast	13	0.208461	transpose
5	0.001052	barriers	14	0.605466	update
6	0.001217	update	15	0.001203	barriers
7	0.708108	bcast	16	0.240084	sum & bcast
8	0.002171	copies	17	0.736583	transpose
9	0.008711	update	18	0.000085	exit

Figure 3 Profiling the parallel routine Exposes parallel inefficiency

ScaLAPACK's block-scattered distribution means that apart from a very small number of diagonal blocks, we deal mainly with rectangular matrices. This makes the parallel version of the hand-written BLAS routine even simpler than the serial one. The resultant parallel routine performs 50% quicker than the ScaLAPACK equivalent and scales better. One advantage of this new code is that the parallel inefficiencies have been exposed rather than being hidden deep in library calls and we can now continue to improve scalability.

5. Optimisation of Communication

With full profiling, the main areas of parallel inefficiency can be pinpointed. Figure 3 shows the full profiling output for the parallel routine. Clearly sections 13, 16 and 17 are troublesome. Sections 4,7 and 14 appear to contribute heavily to the execution time also, though these sections scale well, whilst the designated sections actually see an increase in execution time with number of processors. These areas of the code correspond to a distributed matrix transposition (sections 13 and 16) and a vector summation and broadcast (section 17).

ScaLAPACK routine *pbdtrnv* transposes a vector held in block-cyclic distribution from a row process to a column process or vice-versa. Though a seemingly simple operation, routine *pbdtrnv* is complex and over-engineered, being 800 lines of code with calls to blacs and potentially MPI underneath. This implementation must have a significant latency. Figure 6 shows the individual performance of *pbdtrnv*. We cannot hope for speed-up here since the amount of work increases per-processor with the number of processors, but the significant increase in execution time must be improved upon if the parallel tridiagonal routine is to scale as a whole. Since the routine is communication heavy, it is likely that the best optimisations lay in decreasing the latency of data transfer. There are a number of techniques that can achieve this

- Switch to using 1-sided communications
- Tune to the requirements of program
- Remove barriers
- Exploit grid shape
- Look out for additional cache-reuse opportunity

Data transfer was performed using shmem put/get operations or using shmem_ptr, which gives the address of a remote data object that can then be used as a local object via a Cray pointer. The latency of a shmem_get is around 10% of an MPI_SEND for the data size in question and shmem_ptr can be even quicker, especially on a globally addressable system such as the SGI Origin for obvious reasons.

Though shmem or MPI-1sided operations can be much quicker there is an obvious concern in terms of synchronisation. Remote data objects must be accessible

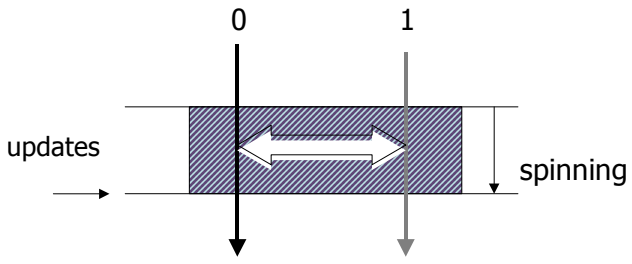

Figure 4 Synchronisation for Data Transfer

and unused by the process in question, similarly both processes must reside within the routine in question, since the data object may be changed to a later value in the code and could be accessed too early by the local process. Hence, despite the inherent performance gain, synchronisation is essential and can result in the performance being decreased. Global barrier routines scale very badly and can be

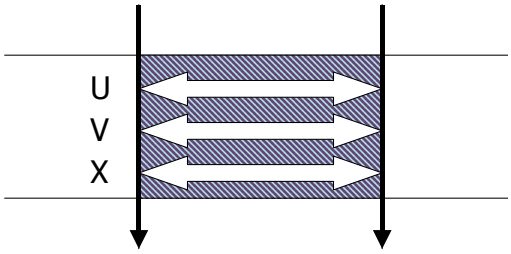

Figure 5 Synchronisation Sharing

unnecessary for point-to-point data-transfer. Instead, the target and host (remote and local) processors can synchronise exclusively with one another using the method shown in Figure 4. Here the remote processor spins on the value of a 'safety' integer, which is updated to a release value at the appropriate time by the local processor. This can be achieved by using the shmem_fence, shmem_wait or shmem_wait_until functions or through a shmem_ptr being located on a safety integer which is updated by the host process at an appropriate time.

Using this method, point to point transfer can be orchestrated at a significantly lower latency than through using global barriers. To achieve the lowest latency possible features of the code must be grouped together to enable as much synchronisation sharing as possible. In this case, three consecutive transposes are performed on vectors named u, v and x. Since the transpositions will occur with the same communication pattern each time, the

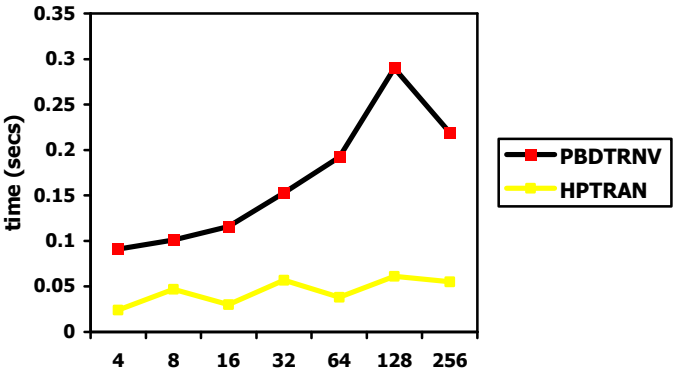

Figure 6 Perfromance of transpose section of tridiagonal routine and ScaLAPACK equivalent

synchronisation overhead is unnecessary for each transposition. Instead, multiple data transfer can be performed at the same point within the transposition routine, as shown in Figure 6. This decreases the latency and increases the performance of the tridiagonal routine significantly.

AIMPRO uses a square BLACS grid wherever possible and encourages the user to do so. This can be exploited by considering the nature of block-cyclic distribution. For a description of block-cyclic distribution, see [4]. In the case of a vector transpose, on square grids the resultant transpose will involve a data transfer to only 1 target processor, and the transpose equates to a regular-distribution transpose. Clearly this can be exploited, since the routine can knowingly perform this simple operation efficiently due to low communication costs. Routine *hptran*, the developed replacement for *pbdtrnv,* recognises a square grid and performs this very simple operation, resulting in a high perfrormance gain for square grids. Additionally, the routine performs the transposition of several identical vectors held simultaneously in the blacs grid. The resultant operation is a series of NP point to point operations for square grids or 2NP operations for n x 2n grids, 3NP for n x 3n etc. Figure 6 shows the overall performance for *hptran* in comparison to ScaLAPACK routine *pbdtrnv*, with the extreme performance gain of square grids being evident.

As revealed in Figure 3, sections 16 and 17 comprised an additional performance overhead and scaling impediment. The series of operations in this section begins with a vector summation of V along a process row, i.e.

> *If a series of column Vectors V_t ,t=0,M-1 of vector length p are distributed in columns of a blacs process grid of length N and width M, then the vector V_{nm} represents the local vector held on the process with BLACS grid row co-ordinate n and column co-ordinate m, and Vc_r is the summation over the r^{th} BLACS process row ,where*

$$Vc_r = \sum_{t=0}^{M-1} V_{rt}(i) \qquad i = 1, p \quad \text{,r=0,N-1} \qquad (1.1)$$

Similarly, row vector Y must be summed over the process column

> *If a series of row Vectors Y_t ,t=1,N of vector length p are distributed in columns of a blacs process grid of length N and width M, then the vector Y_{nm} represents the local vector held on the process with BLACS grid row co-ordinate n and column co-ordinate m, and Yr_r is the summation over the r^{th} BLACS process column ,where*

$$Yr_r = \sum_{t=0}^{N-1} Y_{rt}(i) \qquad i = 1, p \quad \text{,r=0,M-1} \qquad (1.2)$$

Since the vector summation of Yr and Vc is required, yr must first be transposed.

If Yr_t represents the partial accumalation of Yr on column t then

$$(Yr^T)_t = (Yr_t)^T \quad \text{t=0,M-1} \tag{1.3}$$

Vectors Yr^T and Yc can then be added together

$$Xc_t = (Yr)_t^T + Vc_t \quad \text{t=0,N-1} \tag{1.4}$$

And the resultant vector broadcast over all process columns

$$Xc_{tr} = Xc_{t1} \quad \text{r=0,M-1} \quad \text{t=0,N-1} \tag{1.5}$$

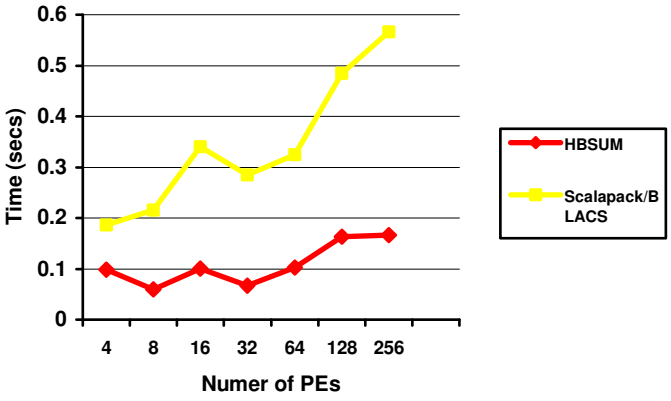

Figure 7 Comparison of the sum/broadcast area of ScaLAPACK and optimised codes

Alternative routines using 1-sided operations can be readily constructed for each of the steps. Ideally, sections of this series of mathematical operations could be fused together within one routine to reduce overhead, but the nature of the operations prevents this; where we have a summation over a BLACS process row followed by a summation of a separate vector over a process column the two operations cannot possibly be synchronised together, since the communication patterns are 'opposite' and the degree of synchronisation would be at least as aggressive as a global barrier. Hence each section must remain distinct and the synchronisation overheads accumulate, with the resultant shmem routines being no more efficient than the MPI-BLACS equivalent. However, if the order of mathematics is re-considered, synchronisation sharing may be possible. Consider the following re-ordering of Equations 1.1 to 1.3.

The series of Row Vectors Y_t ,t=0,N-1 are transposed, giving a series of column vectors $Y^T{}_t$,t=0,M-1

Simultaneously the summations of Y^T and V are calculated,i.e

$$Vc_r = \sum_{t=0}^{M-1} V_{rt}(i) \quad Y_r^T = \sum_{t=0}^{N-1} Y_{rt}(i) \quad i = 1, p \quad \text{,r=0,N-1} \tag{1.6}$$

Additionally, the tailored routine can now create the summation from equation 1.4 within the same subroutine by adding elements as they arrive, and the broadcast of equation 1.5, can to an extent be included in this same synchronisation overhead. Hence we have a series of 5 operations, now optimised using 1-sided communications with minimal synchronisation overhead. Figure 7 shows the significant performance gain this optimal routine has over the blacs-mpi equivalent.

When applied to the tridiagonalisation replacement for ScaLAPACK, the poor scaling has been improved tremendously. The speed-up drop off of pdsytrd on even 8 processors proves its inadequacy for HPC work, whilst the hand-made equivalent continues to show speed up from 128 to 256 processors. Figure 8 shows the overall speed-up figures for the original ScaLAPACK routine, the replacement and the best times after optimisation of communication. The obvious scaling advantage and the

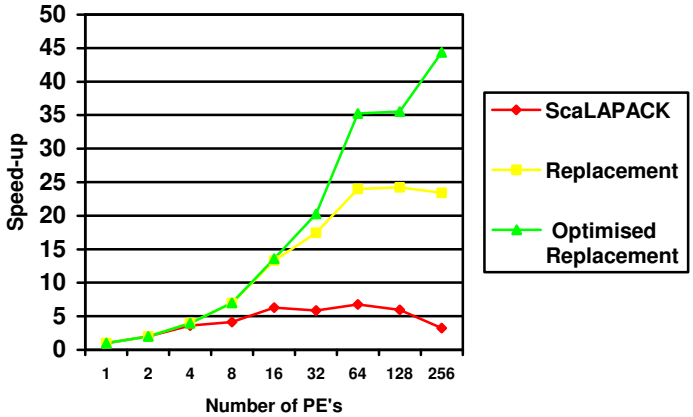

Figure 8 Relative Speed-up of pdsyrtd and the hand made equivalent routine, plus the hand made routine with optimised routines HPTRAN and HPBSUM included

significant performance gain allow AIMPRO to compute eigenvalues much more efficiently and will contribute to the move towards producing a capability-enabled AIMPRO code.

6 Conclusions

The excellent functionality of ScaLAPACK solvers are not being questioned, though it has been proven that their

performance, at least in one case, can be bettered in the limit where a smaller matrix problem must be solved on a larger number of processes than was envisaged by the developers of the library. More robust scaling behaviour will become increasingly important both with the concentration of funding into high-capability systems and also, at the lower end of the market, with the proliferation of less tightly coupled "commodity" Beowulf systems. It is hoped that future work by the authors will lead to improvements in linear algebra, and will contribute in both of these limits.

References

[1] ScaLAPACK www.nelib.org/scalapack
[2] AIMPRO Consortium http://aimpro.ncl.ac.uk/
[3] Higham, NJ *Accuracy and Stability of Numerical Algorithms, SIAM, 1996*
[4] Demmel et al, *ScaLAPACK users guide*