Cray X1 ScaLAPACK Optimization

Adrian Tate

May 17th 2004

1 Introduction

ScaLAPACK [7] has become a popular, efficient and reliable parallel library, just like its serial counterpart LAPACK [1], with heavy use in the fields of electro-magnetics, solid-state physics, astrophysics, climate modelling and QCD. The library contains a comprehensive selection of methods for solving systems of equations, and for finding eigenvalues, all for various matrix types. The main competitors to ScaLAPACK are the NAG parallel library [5] and PLAPACK [4], though usage of either of these libraries is less documented than ScaLAPACK. Vendors are today supplying ScaLAPACK as a component in their numerical libraries. SGI, IBM and Intel provide ScaLAPACK as standard, though Cray Inc. is the first vendor to perform optimisations on ScaLAPACK.

In order to understand the importance of ScaLAPACK library optimisation in the near future, it is important to look at the trends and road-maps of modern architectures. Whilst there is a great deal of importance in ensuring software executes efficiently on todays supercomputers, this must be coupled with an understanding of the way that the software will perform in the mid and long term. This is especially relevant if the advances in technology could undermine any changes made to the current software. To justify a long term optimisation of ScaLAPACK, there must be a sustained interest and market in distributed memory supercomputers (or distributed-shared memory supercomputers), and the communications overhead on those new DM or DSM systems must remain significantly high for scalability of the library to be impeded.

If advances in interconnect network performance increased at a drastically higher rate than the corresponding processor performance, then there would be no need for communications optimisation of numerical libraries, since the computation time would dominate the communication time heavily. We would then see a notable boost in the scalability of applications.

Unfortunately, though there have been some impressive advances in interconnect technology in recent years, this scenario is at best unlikely. SGI's Altix systems use the same interconnect technology as the Origin series, though the processing speed has been doubled. IBM have no interconnect road-map beyond the Federation switch, though their processor speed is set to double within the next five years. Cray's X1e system will show improvements more heavily biased toward processing capabilities. This evidence leads us to believe that communication to computation ratios in existing numerical library codes are destined to become higher over the next 3 years, with the best case scenario being a continuation of the current level. There is certainly no indication that the balance will be so tipped toward the interconnect technology that ScaLAPACK optimisation is deemed unnecessary.

Cray Inc. have recognised the need to address ScaLAPACK optimisation in the short and long term, and as such have collaborated with the University of Manchester, who have a history of addressing ScaLAPACK performance for particular architectures [9]. There are two parallel fronts to this collaboration, the communications optimisation of ScaLAPACK for DSM machines such as the Cray X1/X1e and the long term optimisation of ScaLAPACK for all architectures.

1.1 Communication Programming on the X1

Most MPP style applications are traditionally written in MPI, since this is a fully portable and standardised communications medium. On the Cray T3E and T3D there were ways of writing MPP codes that usually out-performed these MPI codes, usually at the expense of portability. The shmem library [8] was developed by Cray and gives the user a lower latency alternative to MPI. A shmem library call makes a request for remote memory without the involvement of the process to whom that memory is associated, unlike MPI, where both the giving and receiving process are necessarily aware of and involved in the data exchange. The latency of the transfer is lower than MPI, there is no buffering of data, and significant algorithmic improvements can be made due to the 1-sided exchange. To be effective however, a very careful degree of synchronisation needs to be maintained, and if not careful, this can result in shmem codes being similar in performance to their MPI equivalents, since too heavy a synchronisation level cancels any advantageous performance effects. With careful programming however, certain algorithms can be vastly improved if shmem is used to replace severe communication bottlenecks [9, 2]. Vendor support for shmem now comes from Cray, SGI, Compaq, IBM and any cluster with a Quadrics interconnect, so

the portability trade-off that users would previously find is vastly lower.

Co-array Fortran (CAF) [6] is an extension to Fortran 95, and offers a medium for 1-sided data transfers whilst giving a complete syntactical framework in which to perform them. The simplicity of parallel codes written in CAF can be quite striking in relation to the MPI equivalents, but the performance on the Cray t3e was equivalent to shmem-based codes [2]. On the X1, use of CAF in part or all of an application can give rise to extremely efficient codes, for a number of reasons:

- Significantly lower latency than MPI
- one sided message passing can allow algorithmic improvements
- No buffering of data
- No library call

The absence of a library call can give an important performance gain, external library calls can lead to compiler optimisations not working as thoroughly. In the case of the X1, this can lead to less vectorisation, hence CAF X1 codes may give rise to readily vectorised applications.

The syntax for CAF is extraordinarily simple in relation to MPI. An inter process data transfer can be performed using the following statement

```
A(1:5) = B(1:5)[1]
```

Here, B should be declared as a co-array which involves including square brackets at declaration (e.g B[*]). This simple line of code results in the the first five elements of array B on process 1 being copied to the first five elements of the local array A.

Within ScaLAPACK, all meaningful computation is performed in BLAS (Basic Linear Algebra Subroutines), with PBLAS (Parallel Blas) providing the parallel partitioning. Since Cray LibSci BLAS routines are highly tuned, we assume that the performance of any ScaLAPACK routines is limited by its communication patterns at the PBLAS stage, the actual communications medium at the BLACS stage, or the interface to BLAS routines at the local level (i.e the passing of ideal parameters to the BLAS routines). The first and second of these limitations are not separate, since we are likely to make changes to communication patterns if we alter the communications medium, and are also assumed to far outweigh the effects of the final point. Hence it was decided that an initial optimisation of the ScaLAPACK library would be a review of the communications procedures and patterns, with the intent

of making some alterations that introduce some of the beneficial effects of CAF.

2 LibSci Optimisations

Early analysis of the performance of the public domain ScaLAPACK library on the X1 showed that there were very localised and specific bottlenecks that were creating performance problems for a small number of routines. In particular, the LU factorisation routine PxGETRF appeared to be performing poorly. An important Cray customer regularly runs applications that necessarily require good performance from PxGETRF, and hence there was an initial concern that this routine be optimised throughly. It was felt that after this routines had some initial performance measures looked at, the library as a whole would need some attention.

The policy decision taken was to enhance the functionality of the BLACS programs [3]. Currently, the public domain versions of these routines contain a limited set of subroutines that provide all the communications functionality required in ScaLAPACK. Previous work however has shown that specific communication bottlenecks can be better replaced by a less general replacement procedure. A sensible optimisation of the the BLACS would therefore be to extend the functionality wherever this seemed appropriate, governed by the concerns of the particular routines that are being prioritised.

Profiles and performance studies of double complex PZGETRF reveal that the poor performance is specific to the X1, since performance on the SGI Origin public domain version can be shown to scale well up to 16 processors, as shown in Figure 1. The most disturbing feature of X1 performance was the slow down that occurred between 2 and 4 processors. As the number of processors is increased, the execution times on the X1 become closer to the SGI times, despite beginning several times better on 1 and 2 processors.

Number of Processors	1	2	4	8	16
Cray X1 (MSP)	22.0	13.9	32.1	23.2	26.8
SGI Origin 3000	219.1	135.5	75.4	40.8	23.1

Figure 1: LU factorisation times for M=N=4000, MB=NB=63

In the analysis of the LU factorisation routine it became obvious that there was a serious bottleneck in the area of row pivoting. This row pivot is dealt with in a particular way with MPI. Since the data is non-contiguous in memory, the elements that are to be transferred are copied into a contiguous

vector (packed) before being transfered across the network to the destination processor, where it is unpacked. This packing and unpacking process was one of the expensive features in the execution, the others were general MPI latency, and broadcast times.

The routines that were being used to perform the vector swap were the BLACS send receive pair xGESD2d and xGERV2d. This appeared to be a situation where a more specific communications routine may perform better. Since the send receive pair are designed for any point-to-point communications, in this particular instance generic point-to-point replacements would perform worse than a specific vector swap program. This is related to the amount of synchronisation involved in each. If each processor were to use a 1-sided point to point send and 1-sided point to point receive then we would need 3 barrier synchronisations, whereas a new subroutine that can perform the entire swap requires only 2 synchronisations. Hence new functionality was added to LibSci's BLACS library in the form a Vector swap routine xVecswap. The non contiguous data transfers in xVecswap are performed using the following (pseudo) expression:

```
call shmem_igetXX(source_array,target_array,source_stride,taget_stride,length,dest_pe)
```

This statement performs a strided, 1-sided direct memory transfer with starting at source_array(1) remotely and target_array(1) locally, with a remote stride of source_stride and a local stride of local_stride. XX= the number of bytes to be transferred.

Hence a block of data can be transferred using a loop:

```
do i = 1, ni

  call shmem_igetXX(source_array(1),target_array(1),remlda,lda,mi,dest)

end do

where ni = number of columns to be transferred
      mi = number of rows to be transferred
      remlda = remote leading dimension
      lda = leading dimension
      dest = destination processor
```

Symmetric memory, being memory that is allocated from the symmetric heap, is ordinarily necessary for data transfer using shmem, CAF, UPC or

MPI-2. This is because only one processor is involved in the exchange and hence cannot receive addresses from the other processor. In MPI, a buffer is defined by both parties and there is no ambiguity of where the data is located. If memory is allocated from the symmetric heap, both parties have the relevant data in the same location, and hence any processor naturally looks for remote data in the same location that he does locally. Co-arrays are allocated from the symmetric heap in a way that is hidden from the user, when using shmem the user must explicitly allocate his arrays from the symmetric heap using 'shpalloc '. Unfortunately, there are performance issues with using symmetric memory. Firstly, a call to shpalloc necessarily involves a global barrier because all processors need to be involved, this actually prevents shpalloc being used in a subroutine that may be called by a subset of processors, which is the case for all of the BLACS communication routines. Further, the use of symmetric memory can be a real burden to the programmer, because usually ones actual arguments are not symmetric. This means that a symmetric array must be allocated and the data copied into this symmetric array. As well as being problematic for the programmer, this is inefficient, as a parallel numerical routine may be called hundreds of thousands of times repeatedly, with an internal communication routine being called thousands of times for every call of the parallel routine, hence internal copies for the sake of data transaction are likely to become visible (at best) or dominate (at worst). The need for symmetric memory in the LibSci Vector swap routine was avoided using the following technique, where process 0 is to perform a direct memory transfer from process 1.

```
Subroutine nonsymtrans(A,m,n,iam,dest)
Real :: A(m,n), AC(m,n)
Pointer(aptr,AC)
Integer*8 :: flag
Integer   :: iam,dest
DATA flag /0_8/
SAVE flag


! if iam 0 then wait for location

If(iam==0)then
Call shmem_wait(flag,0)
aptr = flag
flag = 0
```

```
call shmem_fence()
call shmem_get(A,AC,m*n,dest)

else

! put location to 0


call shmem_put8(flag,loc(A),1,dest)

endif

end subroutine
```

In this example, process zero waits for some information from process 1. Process 1 passes the address of the array A to process 0, who then gives aptr that value. Since aptr is a Cray pointer to AC, AC now has the same location as the remote A. Hence when process 0 later makes a shmem_get call, it gives AC as the source array, when making the remote data transfer, process zero looks in the AC location, which is remotely array A's location. This process is in some ways a concession, since we have made the process equivalent to a hand-shake, but the performance improvements in doing so outweigh this concession, and any synchronisation can be performed in parallel (the shmem_wait allows process 0 to wait for data, but also acts as a point-to-point barrier). In co-array Fortran, one can achieve the same accessing of non-symmetric data using a derived-type that is a co-array.

```
subroutine cafp ( A, C, len , dest )
type caf
real, pointer, dimension ( : , : ) :: co
end type

real :: A(*),C(len)
type (caf) :: B[*]
integer :: len,dest


B%co => A(1 : len)


call sync_all()
```

```
B[dest]%co( 1 : len ) = C(1 : len)


end subroutine
```

This method of using co-arrays appears heavily in libsci's ScaLAPACK, we will refer to this method as the 'pointer method' within the scope of this document. The CAF method of using non-symmetric data is more powerful than the shmem version since the shmem version requires communication for the purpose of transferring data amongst one process in particular, whilst the co-array method allows all processors to access the non-symmetric memory

Using the two techniques described in this section, the performance of the new Vector swap routine improves the performance of the LU factorisation sufficiently to solve the more severe performance problems, though there is still a problem when moving from 4 to 8 processors, this problem is related to broadcast times and was dealt with by the production of new BLACS broadcasts.

Number of Processors	1	2	4	8	16
Cray X1 (MSP)	20.0	12.4	8.8	9.13	6.70
SGI Origin 3000	219.1	135.5	75.4	40.8	23.1

Figure 2: Optimised X1, and SGI O3K LU factorisation times for M=N=4000, MB=NB=63

The special feature of this algorithm is that we see the scaling characteristics to be much better. Comparing figures 3(b) and 3(c) helps us to deduce that the performance of the ring algorithm is proportional to number of processors, whilst the direct algorithm is only dependent on the number of processors for the barrier section. This can make a huge difference at large process counts.

Intelligent memory structures such as that exhibited by the Cray X1 can allow extremely efficient codes whilst simultaneously being very simple. In the case of a collective operation such as a broadcast, this idea can be extended to its maximum, since the 'simplest' broadcast algorithm can be used, which is described in figure 3 (c). Here, after synchronisation, each process simultaneously copies memory from the source element. The X1 memory system is robust enough to deal with these simultaneous copies and hence an algorithm of this type was incorporated into LibSci.

Using CAF, the algorithm shown in Figure 3 (c) can be very easily programmed.

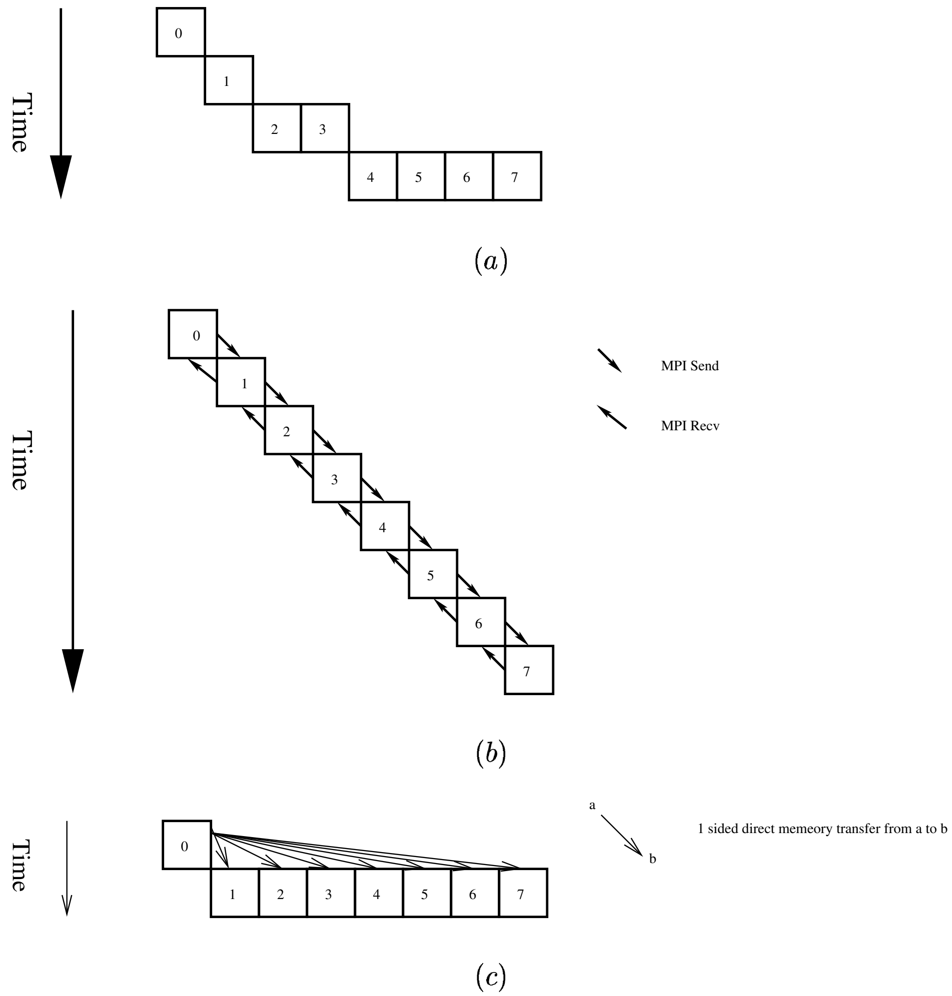

Figure 3: Approximate times for three broadcast algorithms a) a binary tree broadcast b) a ring broadcast and c) a 1-sided direct broadcast

```
temp(:,:) = transpose(a(:,:))
call sync_all
a(:,:)[target] = temp(:,:)
call sync_all
```

One feature that is important in the above example is the use of a co-array of derived type for the main data transfer. Since we do not have a co-array being passed into the subroutine, and we do not want the expensive copying in and out of a co-array, we can use a co-array of derived type that also has the target attribute in the following manner

Number of Processors	2	4	8	16	32
Old broadcasts	96.03	17.02	14.15	9.30	8.50
New Broadcasts	87.57	14.70	12.16	9.21	7.64

Figure 4: LU factorisation times for M=N=5000, MB=NB=63

Number of Processors	4	8	16	32
Old broadcasts	110.29	65.84	38.57	35.27
New Broadcasts	87.30	51.20	31.92	27.29

Figure 5: LU factorisation times for M=N=10000, MB=NB=63

Number of Processors	32	64	128
Old broadcasts	148.4	92.69	66.45
New Broadcasts	126.25	76.20	44.01

Figure 6: LU factorisation times for M=N=20000, MB=NB=63

3 Longer Term Alterations to ScaLAPACK

Whilst there have been significant improvements made in the LibSci ScaLAPACK implementation, there are several reasons why to continue with this

approach (i.e the optimisation of BLACS routines directly) will not help the performance of the library indefinitely. We explore these reasons here.

The use of CAF as a direct replacement for MPI has achieved good results as we saw in Section 2, mainly due to its superior handling of non-contiguous data and its lack of library calls. It is however, not entirely efficient to have a co-array that is defined inside a communications routine and which is used only in that particular routine. If a co-array was passed into the communications routine, then we would see much better performance. Ideally then, LibSci's ScaLAPACK would have co-arrays defined throughout the library. This may pose a problem for the caller of the new library, since we might expect that the user's main data all be held in co-arrays when the library is called. There are two solutions to this; a separate co-array LibSci could be made easily available for those users that wish to have co-arrays included in their application, whilst existing users could make use of a library that switched data to co-arrays at the topmost ScaLAPACK layer. Alternatively, the entire library could be programmed using the pointer method as described in section 2. A performance evaluation of using this method throughout ScaLAPACK is included in this section.

3.1 CAF ScaLAPACK

As already discussed, the direct optimisation of BLACS routines, such as that already carried out in the LibSci collaboration can give some performance improvements for specialist areas of the library that are currently suffering but does not use co-arrays to their maximum potential. In addition, there are algorithmic reasons why we might need to re-think our approach. In particular, the nature of 1-sided communications procedures must be discussed in detail, since we find that changes to the BLACS layer results in some degree of serialisation of communications. If we consider a blocked matrix transpose as shown in figure 3.1 then we can begin to see how the inherent differences between 1-sided and 2-sided communications procedures can affect algorithms. Using 9 processors, distributed among a 3x3 process grid, we can make a simple code from either MPI or Co-array Fortran (though the CAF implementation is still notably more simple). Figure 3.1 shows the simplified overall communication costs for a two sided implementation. Each processor has to be involved in a 2-way operation that for both the remote sending of data and the receiving of data from a remote processor. The overall cost is equivalent to 4x an MPI send operation.

We can visualise the respective process involved in the two variations in Figure 3. We could of course use MPI non blocking send-receive pairs, or

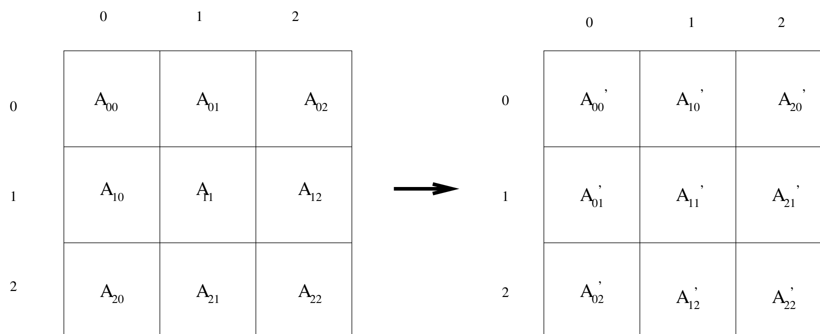

Figure 7: A block matrix transpose on a 3x3 process grid.

introduce other ways to make the MPI-BLACS version more efficient, but this would not match the way that the PBLAS are designed. It is clear from the diagram, that even though this is a contrived and very simple case, the CAF implementation is more efficient than the MPI BLACS, as long as the additional barrier layer remains negligible in relation to the four 2-sided MPI handshakes. This is likely in all but the most severely imbalanced problems.

We now must consider the situation where the internal MPI BLACS procedures are replaced with Co-array Fortran replacements. In this case, we need a direct replacement for the BLACS calls made within PBLAS, i.e we will replace BLACS send DSGES2D and BLACS receive DGERV2D with co array replacements, which are called directly from the PBLAS. Figure 3.1 shows how the process would proceed if these CAF replacements were introduced into existing PBLAS style interface to a distributed transpose. Remote puts require the additional synchronisation layer due to the dangerous possibility that remote data no longer exists or has not yet been declared. One might say that there are two consecutive synchronisation calls made, the second of which is unnecessary, but we are assuming that the barrier calls and the remote get are made within a *generic* replacement subroutine, which therefore requires necessarily that a safety barrier call is made on entry and exit. (This is why LibSci new vector swap routine replaced existing calls to BLACS send and BLACS receive). The two sided message passing call is likely to be slower than a remote put, but we can see when comparing figures 3(a) and 3.1 that the new replacement is unlikely to be much more efficient than the MPI BLACS original, if indeed it is more efficient at all. If the synchronisation costs become large, the new version may indeed be less efficient than the BLACS version.

This important point suggests that, for a full treatment of ScaLAPACK,

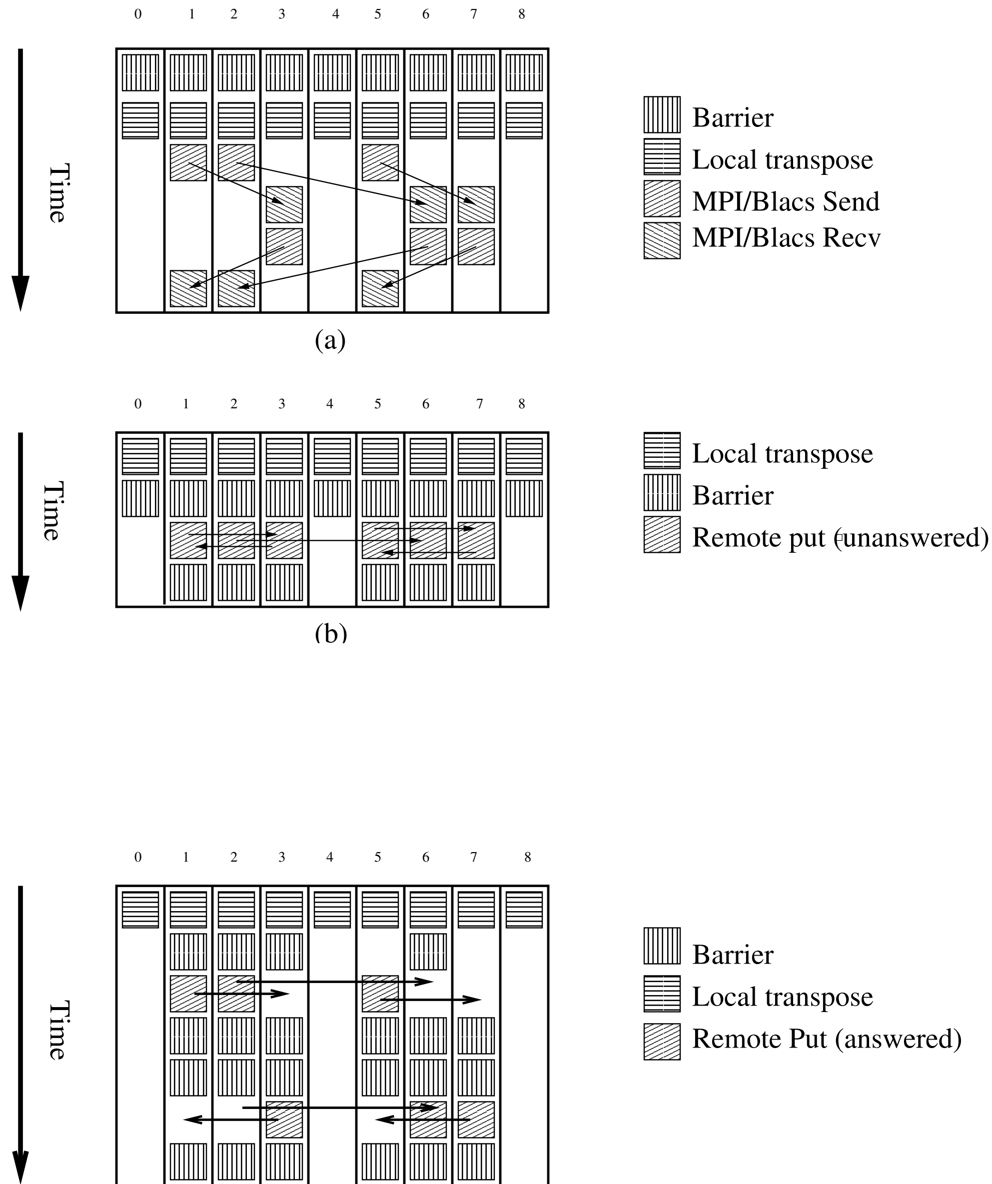

Figure 8: If direct CAF replacements were made to existing communications procedures and used directly within the PBLAS software

Array dimension	1	10	100	1000	10000	100000	1000000
Pointer method	369082	332822	308878	389721	1349011	11259883	110688810
Normal Method	306190	427384	313024	429069	1091233	9409761	77673916

Figure 9: Co-array Pointer Inefficiency

we would benefit greatly from adapting the PBLAS layer so that a more 1-sided friendly communications pattern was used.

Though some modifications are clearly needed at the PBLAS layer, it is not clear as to whether a CAF ScaLAPACK layer is essential. For this to be the case, the way those BLACS modifications have already been implemented (i.e. Co-array existing in the inner-most communications routines only) would have to be inefficient enough to warrant a new approach. Any inefficiency would be related to the inherent inefficiency of using the pointer method described in section 2. This inefficiency is the extra time taken to make a memory reference using the pointer method compared to a reference of a co-array dummy argument. Figure 3.1 shows the performance of a test code that compares these two times for varying array sizes (Here 'clock pulses' is just the relative time. The difference in performance increases as the number of passed elements increases, as one would expect if the time taken to reference an element is increased. Further tests, which were limited to a fixed number of array references inside the subroutine whilst the number of passed elements was varied showed a constant level difference between the two programs.

The amount of inefficiency that one is likely to experience when using this method is therefore proportional to the number of array references that one makes within the subroutine in question. In the new BLACS, we are unlikely to make extremely large amounts of memory references, since we are going to perform communication only, which should require one array reference for every time a Co-array reference is made. This will usually be equal to the number of columns of data to be transferred, which in turn is related to the blocking factor. We conclude that for block sizes that are typical (say 32 to 256), we will make an insufficient number of expensive references to make the new BLACS routines be deemed inefficient. Hence, the BLACS replacements that are within LibSci at present could be considered to be efficient, in the respect that they are not suffering performance problems related to the embedded co-array Fortran within them.

The fact that the BLACS may exist with the same software structure that is currently being employed seems at odds with the earlier point; that

the PBLAS layer is not conducive to good performance from CAF. We can however, keep the BLACS that contain embedded CAF, and still make changes to the PBLAS that will enable more effective use of 1-sided communications. For example, we can review PBLAS for situations where there are send/receive pairs that would be more efficient if replaced with a vector swap.

Since function calls are very expensive on the X1, BLACS routines may cause problems related to the MPI function calls. these calls, if contained within a loop may cause loops to not be vectorized, and hence can cause real problems. We propose that the communication based MPI calls within ScaLAPACK be replaced in their entirety, thus reducing the possibility of such a problem.

3.2 Summary

We can summarise the conclusions of this study as follows

- PBLAS layer will be modified to support 1-sided communications better.
- The functionality of the BLACS will be increased.
- The pointer method of using co-arrays is not inefficient for the relevant data sizes.
- Therefore, we can continue to create BLACS with embedded CAF.
- Because of expense of function calls, all MPI calls should be removed.

References

[1] J. Dongarra E.C. Anderson. Performance of lapack: A portable library of numerical linear algebra routines. *Proceedings of the IEEE*, 81:1094–1101, 1993.

[2] Paul Burton, Bob Carruthers, Gregory Fischer, Brian Johnson, and Robert Numrich. Converting the halo-update subroutine in the met office unified model to co-array fortran. 2001.

[3] Walker DW Choi J, Dongarra JJ. Software libraries for linear algebra computations on high-performance computers. *SIAM Review*, 2(37), 1995.

[4] R.A. Van de Geijn. *Using PLAPACK*. MIT Press, 1997.

[5] M Derkashan and A Krommer. The nag parallel library and the pineapl project. *Lecture NOtes in COmputer Science*, (1497), 1998.

[6] Robert Numrich and John Reid. Writing a multi-grid solver using co-array fortran. *Lecture Notes in Computer Science*, 1541(390-399), 2002.

[7] Choi J Demmel J Dhillon I Dongarra J Ostrouchov S Petitet A Stanley K Walker D Whaley RC. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. *COMPUTER PHYSICS COMMUNICATIONS*, 97(1-2), 1996.

[8] Cray Research. Shmem library.

[9] Adrian Tate and Patrick Briddon. High performance linear algebra. *CUG proceedings*, 2002.