Practical Experience in the Dangers of Heterogeneous Computing

L. S. Blackford[*], A. Cleary[†], J. Demmel[‡], I. Dhillon[‡], J. Dongarra[§],
S. Hammarling[¶], A. Petitet[†], H. Ren[‡], K. Stanley[‡], R. C. Whaley[†]

Abstract

Special challenges exist in writing reliable numerical library software for heterogeneous computing environments. Although a lot of software for distributed memory parallel computers has been written, porting this software to a network of workstations requires careful consideration. The symptoms of heterogeneous computing failures can range from erroneous results without warning to deadlock. Some of the problems are straightforward to solve, but for others the solutions are not so obvious, or incur an unacceptable overhead. Making software robust on heterogeneous systems often requires additional communication.

This paper addresses the issue of writing reliable numerical software for networks of heterogeneous computers. We describe and illustrate the problems encountered during the development of ScaLAPACK and the NAG Numerical PVM Library. Where possible, we suggest solutions to avoid potential pitfalls, or if that is not possible, recommend that the software is not used on heterogeneous networks.

1 Introduction

There are special challenges associated with writing reliable numerical software on networks containing heterogeneous processors, that is processors which may do floating point arithmetic differently. This includes not just machines with completely different floating point formats and semantics, such as Cray vector computers running *Cray arithmetic* versus workstations running IEEE standard floating point arithmetic, but even supposedly identical machines running with different compilers, or even just different compiler options or runtime environments.

The basic problem occurs when making *data dependent branches* on different processors. The flow of an algorithm is usually data dependent and so slight variations in the data may lead to different processors executing completely different sections of code.

This paper represents the experience of the ScaLAPACK and NAG teams in developing numerical software for distributed memory message-passing systems, and the awareness

[*](formerly L. S. Ostrouchov) University of Tennessee, Knoxville, USA
[†]University of Tennessee, Knoxville, USA
[‡]University of California at Berkeley, USA
[§]University of Tennessee, Knoxville, and Oak Ridge National Laboratory, Oak Ridge, TN, USA
[¶]University of Tennessee, Knoxville, and Numerical Algorithms Group Ltd., Oxford, UK

that the software being developed may not be as robust on heterogeneous systems as on homogeneous systems. We briefly describe the work of these teams in Section 2, and Section 3 defines our use of the terms homogeneous and heterogeneous computing, and discusses the considerations leading to the definitions.

In Sections 4, 5 and 8 we look at three areas that require attention in developing software for heterogeneous networks: machine parameters, where we discuss what the values of machine parameters, such as machine precision should be; checking global arguments and communicating floating point values; and algorithmic integrity, that is, how can we ensure that algorithms perform correctly in a heterogeneous setting. The particular case of communicating floating point values on IEEE machines is briefly discussed in Section 6. Some additional considerations arising from what we regard as poor arithmetic, ranging from lack of full IEEE arithmetic support to unnecessary overflow in complex arithmetic, are discussed in Section 7.

This report is an updated version of [5], which takes into account problems encountered during the preparation of Version 1.2 of ScaLAPACK [2].

2 Motivation and Background

The challenges of heterogeneous computing discussed in this paper came to light during the development of ScaLAPACK and the NAG Numerical PVM Library ([17]).

ScaLAPACK is a library of high performance linear algebra routines for distributed memory MIMD machines. It is a continuation of the LAPACK project, which has designed and produced an efficient linear algebra library for workstations, vector supercomputers and shared memory parallel computers ([1]). Both libraries contain routines for the solution of systems of linear equations, linear least squares problems and eigenvalue problems. The goals of the LAPACK project, which continue into the ScaLAPACK project, include efficiency so that the computationally intensive routines execute as fast as possible; reliability, including the return of condition estimates and error bounds; portability across machines; flexibility so that users may construct new routines from well designed components; and ease of use. Towards this last goal the ScaLAPACK software has been designed to look as much like the LAPACK software as possible. ScaLAPACK is naturally also concerned with scalability as the problem size and number of processors grow.

Many of these goals have been attained by developing and promoting standards, especially specifications for basic computational and communication routines. Thus LAPACK relies on the BLAS ([16, 7, 6]), particularly the Level 2 and 3 BLAS for computational efficiency, and ScaLAPACK relies upon the BLACS ([8]) for efficiency of communication and uses a set of parallel BLAS, the PBLAS ([3]), which themselves call the BLAS and the BLACS. LAPACK and ScaLAPACK will run on any machines for which the BLAS and the BLACS are available. A PVM ([9]) version of the BLACS has been available for some time and the portability of the BLACS has recently been further increased by the development of a version that uses MPI ([18]).

As the BLACS are perhaps not so widely known as the BLAS and LAPACK, we now give a brief description. The BLACS, which stands for Basic Linear Algebra Communication Subprograms, form a message passing library, specifically designed for dense linear algebra, in which the computational model consists of a one or two dimensional grid of processes, where each process stores matrices and vectors. The BLACS include synchronous send and receive routines to send a matrix or submatrix from one process to another, to broadcast submatrices to many processes, or to compute global reductions such as sums, maxima and minima. There are also routines to set up, change, or query the process grid. The BLACS permit a process to be a member of more than one, possibly overlapping, grids, each one labeled by a **context**. Some message passing systems also include the idea of a context; in MPI it is termed a *communicator*. See [8] and [18] for further details.

The NAG Numerical PVM Library is a library of numerical routines, also for distributed memory MIMD machines, that contains routines for dense and sparse linear algebra, including ScaLAPACK routines, quadrature, optimization, random number generation and various utility routines for operations such as data distribution and error handling. This library owes much to the ScaLAPACK development, uses essentially the same model for distributed memory computing as ScaLAPACK and was developed with the same goals in mind ([11]). Since the development of an MPI version the NAG Library is now known generically as the NAG Parallel Library.

Both ScaLAPACK and the NAG Numerical PVM Library use the BLACS computational model and utilize the BLACS context. In addition they both use an SPMD programming model.

ScaLAPACK and the NAG Numerical PVM Library were developed with heterogeneous environments in mind, as well as standard homogeneous machines. But during development it was realized that we could not guarantee the safe behavior of all our routines in a heterogeneous environment and so, for the time being, both libraries are only fully supported on homogeneous machines. ScaLAPACK, though, is tested on networks of IEEE machines and is believed to work correctly in such environments, and it is intended to be able to fully support other heterogeneous environments in the near future. Any known heterogeneous failures are documented in the file `errata.scalapack` on Netlib[1]. It is intended that the NAG Parallel Libraries should also support heterogeneous environments in the future.

In this report we concentrate primarily on the ScaLAPACK experience.

3 Homogeneous and Heterogeneous Computing

The definition of a heterogeneous computing environment depends to some extent on the application. Here we attempt a definition that is relevant to numerical software. The three main issues determining the classification are the hardware, the communication layer, and the software (operating system, compiler, compiler options). Any differences in these areas can potentially affect the behavior of the application. Specifically, the following conditions must be satisfied before a system can be considered **homogeneous**:

1. The hardware of each processor guarantees the same storage representation and the same same results for operations on floating point numbers.

2. If a floating point number is communicated between processors, the communication layer guarantees the exact transmittal of the floating point value.

[1] `http://www.netlib.org/scalapack/index.html`

3. The software (operating system, compiler, compiler options) on each processor also guarantees the same storage representation and the same same results for operations on floating point numbers.

We regard a **homogeneous machine** as one which satisfies condition (1.); a **homogeneous network** as a collection of homogeneous machines which additionally satisfies condition (2.); and finally, a **homogeneous computing environment** as a homogeneous network which satisfies condition (3.). We can then make the obvious definition that a **heterogeneous computing environment** is one that is not homogeneous. The requirements for a homogeneous computing environment are quite stringent and are frequently not met in networks of workstations, or PCs, even when each computer in the network is the same model.

Some areas of distinction are quite obvious, such as a difference in the architecture of two machines, or the type of communication layer implemented. Communication issues are discussed in more detail in Section 6. Some hardware and software issues, however, can potentially affect the behavior of the application and be difficult to diagnose. For example, the determination of machine parameters such as machine precision, overflow, and underflow; or the implementation of complex arithmetic such as complex division; or the handling of NaNs and subnormal numbers could differ. Some of these subtleties may only become apparent when the arithmetic operations occur on the edge of the range of representable numbers. Section 4 discusses arithmetic issues in more detail.

The difficult question that remains unanswered for developers of library software is: when can we *guarantee* that heterogeneous computing is safe? There is also the question of just how much additional programming effort we should expend to gain the additional robustness. Unless we can incorporate a reliable test for homogeneity, we are also in danger of imposing a considerable additional performance penalty on homogeneous systems in order to perform safely on heterogeneous systems.

To illustrate the potential problems consider the iterative solution of a system of linear equations where the stopping criterion depends upon the value of some function, f, of the relative machine precision[2], ϵ. The test for convergence might well include a test of the form:

$$\texttt{If}\quad \|e_r\|_2/\|x_r\|_2 < f(\epsilon)\quad \texttt{then converged}$$

In a heterogeneous setting the value of f may be different on different processors and e_r and x_r may depend upon data of different accuracies, and thus one or more processes may converge in a fewer number of iterations. Indeed the stopping criterion used by the most accurate processor may never be satisfied if it depends on data computed less accurately by other processors. If the code contains communication between processors within an iteration, it may not complete if one processor converges before the others. In a heterogeneous environment, the only way to guarantee termination is to have one processor make the convergence decision and broadcast that decision.

This is a strategy we shall see again in later sections.

[2] A common definition of the relative machine precision, or unit roundoff, is the smallest positive floating point value, ϵ, such that $\mathrm{fl}(1+\epsilon) > 1$, where $\mathrm{fl}(x)$ is the floating point representation of x. See [12, Chapter 2] for further details.

4 Machine Parameters

Machine parameters such as the relative machine precision, the underflow and overflow thresholds, and the smallest value which can be safely reciprocated (which in LAPACK is called `sfmin`), are frequently used in numerical linear algebra computations, as well as in many other numerical computations. Without due care, variations in these values between processors can cause problems, such as those mentioned above.

Many such problems can be eliminated by using the *largest* machine precision among all participating processors. In LAPACK routine `DLAMCH` returns the (double precision) machine precision (as well as other machine parameters). In ScaLAPACK this is replaced by `PDLAMCH` which returns the largest value over all the processors, replacing the uniprocessor value returned by `DLAMCH`. Similarly, one should use the smallest overflow threshold and largest underflow threshold over the processors being used. The ScaLAPACK routine `PDLAMCH` runs the LAPACK routine `DLAMCH` on each process in the context and communicates the relevant maximum or minimum value. We refer to these machine parameters as the **multiprocessor machine parameters**. `DLAMCH` can also return the base, b, which nowadays is invariably $b = 2$, but what we would do for `PDLAMCH` if we ever had a mixture of binary and decimal machines in a network we leave as an open question!

Note that since `PDLAMCH` requires communication to each process in the context, it suffers from the weakness that it cannot be called by a subset of the processes (as might for example happen when a conditional statement such as an `IF` statement is being executed), because processes will be waiting for a communication which will never take place. There are many examples in ScaLAPACK codes, however, where only a subset of nodes (for instance one column or one row of the process grid) is performing a given computation, such as pivot selection. ScaLAPACK has to avoid calling `PDLAMCH` from such computations. Section 8 contains a specific example of this case.

For this reason, it is expected that the next release of the BLACS will support caching based on the BLACS context. We will then be able to perform the communication just once for each context and cache the values on the context. Subsequent `PDLAMCH` calls within the context will then access strictly local data, so will be more efficient, and thus may be safely called from code performing computations on grid subsets.

5 Global Arguments and Floating Point Values

In a homogeneous environment we think of a *global* variable as having the same value on each process, but of course this may not be true of floating point values in a heterogeneous environment.

Where possible, the high level routines in the ScaLAPACK and NAG Libraries check arguments supplied by users for their validity in order to aid users and provide as much reliability as possibility. In particular, global arguments are checked. When these global arguments are floating point values they may of course, for the reasons previously discussed, have different values on different processors.

This raises the question of how, and even whether, such arguments should be checked, and what action should be taken when a failure occurs. If we compare the values, they may not be the same on each process, so we need to allow a tolerance based upon the multiprocessor machine precision. Alternatively, we can check a global argument on just one process and then, if the value is valid, broadcast that value to all the other processes. Of course this alternative approach has extra overhead, but it may be the most numerically sound solution, since the library routine has algorithmic control, and puts slightly less burden on the user.

Similar issues occur whenever we communicate a floating point value from one processor to another. Unless we have special knowledge, and one such case will be discussed in the next section, we should not assume that the target processor will have exactly the same value as the sending processor and we must write the code accordingly.

6 Communicating Floating Point Values on IEEE Machines

The IEEE standard for binary floating point arithmetic ([13]) specifies how machines conforming to the standard should represent floating point values. We refer to machines conforming to this standard as **IEEE machines**[3]. Thus, when we communicate floating point numbers between IEEE machines we might hope that each processor has the same value. This is a reasonable hope and will often be realized.

For example, XDR (External Data Representation, [19]) uses the IEEE representation for floating point numbers and so a message passing system that uses XDR will communicate floating point numbers without change[4]. PVM is an example of a system that uses XDR. MPI suggests the use of XDR, but does not mandate its use ([18, Section 2.3.3]). Unless we have additional information about the implementation we cannot assume that floating point numbers will be communicated without change on IEEE machines when using MPI.

Note that there is also an IEEE standard concerned with standardizing data formats to aid data conversion between processors ([15]).

7 Considerations Due to Poor Arithmetic

As we expand the ScaLAPACK test suite to encompass more rigorous testing, particularly for floating point values close to the edge of representable numbers (as is present in the LAPACK test suite), we are reminded of additional dangers which must be avoided in floating point arithmetic. For example, it is a sad reflection that some compilers still do not implement complex arithmetic carefully. In particular, unscaled complex division still occurs on certain architectures, leading to unnecessary overflow[5]. To handle this difficulty ScaLAPACK, as LAPACK, restricts the range of representable numbers by a call to routine `PDLABAD` (in double precision), the equivalent of the LAPACK routine `DLABAD`, which, if desired, takes the square root of the smallest and largest representable numbers for the computation to protect from unnecessary underflow or overflow. `PDLABAD` calls `DLABAD` locally on each process and then communicates the minimum and maximum value respectively. Arguably we should have separate routines for real and complex arithmetic, but since we

[3]It should be noted that there is also a radix independent standard ([14]).

[4]It is not clear whether or not this can be assumed for subnormal (denormalized) numbers.

[5]At the time of testing ScaLAPACK version 1.2, the HP9000 exhibited this behavior

hope that the need for DLABAD will eventually disappear we have so far resisted taking that step.

This is particularly irritating if one machine in a network is causing us to impose unnecessary restrictions on all the machines in the network, but without this, catastrophic results can occur during computations near the overflow or underflow thresholds.

Another problem that we have encountered during testing is in the way that subnormal (denormalized) numbers are handled on certain (near) IEEE architectures. By default, some architectures flush subnormal numbers to zero[6]. Thus, if the computation involves numbers near underflow and a subnormal number is communicated to such a machine, the computational results may be invalid and the subsequent behavior unpredictable. Often such machines have a compiler switch to allow the handling of subnormal numbers, but it can be non-obvious and we cannot guarantee that users will use such a switch.

This behavior occurred during the heterogeneous testing of the linear least squares routines when the input test matrix was a full-rank matrix scaled near underflow. During the course of the computation a subnormal number was communicated, this value was unrecognized on receipt, and a floating point exception was flagged. The execution on the processor was killed, subsequently causing the execution on the other processors to hang. As we expand the test suite we expect to discover such behavior in other parts of ScaLAPACK, since we do not believe that there was anything special about the least squares routines.

A solution would be to replace subnormal numbers either with zero, or with the nearest normal number, but we are somewhat reluctant to implement this within ScaLAPACK, since this does not seem to be the right software level at which to do this.

A simple example program to illustrate this problem is given in Appendix A.

8 Algorithmic Integrity

The suggestions we have made so far certainly do not solve all of the problems. We are still left with major concerns for problems associated with varying floating point representations and arithmetic operations between different processors, different compilers and different compiler options. We have given one example at the end of Section 3 and we now illustrate the difficulties with three further examples from ScaLAPACK, the second example giving rather more severe difficulties than the first and third.

Many routines in LAPACK and hence also in ScaLAPACK, scale vectors and matrices. The scaling is done to equilibrate or balance a matrix in order to improve its condition, or to avoid harmful underflow, or overflow, or even to improve accuracy by scaling away from subnormal numbers. When scaling occurs we naturally have to ensure that all processes containing elements of the vector or matrix to be scaled, take part in the scaling. Consider the case of a four element vector

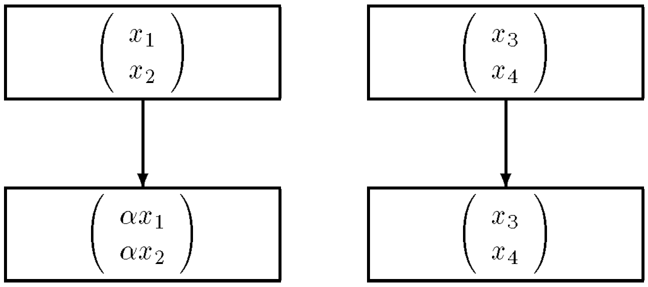
$$x^T = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix}$$

distributed over two processors, with the following test for scaling:

$$\texttt{if}\ \ \|x\|_2 < \delta\ \ \texttt{then}\ \ x \leftarrow \alpha x$$

[6]The DEC Alpha, at the time of writing, is an example.

As illustrated below, if we let each processor make the decision independently then we risk the danger of one processor scaling, while the other does not.



If this situation occurred the computation would now proceed with the meaningless vector

$$x^T = \begin{pmatrix} \alpha x_1 & \alpha x_2 & x_3 & x_4 \end{pmatrix}.$$

One way to ensure correct computation is to put one process in control of whether or not scaling should take place, and for that process to communicate the decision to the other processes. Having a **controlling process** is a common way to solve such problems on heterogeneous networks.

An example of a routine that scales to improve accuracy is the LAPACK routine `DLARFG`, which computes an elementary reflector (Householder transformation matrix) H such that

$$Hx = \beta e_1,$$

where β is a scalar, x is an n element vector and e_1 is the first column of the unit matrix. H is represented in the form

$$H = I - \tau v v^T,$$

where τ is a scalar and v is an n element vector. Since H is orthogonal we see that

$$|\beta| = \|x\|_2.$$

If $|\beta|$ is very small (subnormal or close to being subnormal), `DLARFG` scales x and recomputes $\|x\|_2$. This computation is at the heart of the LAPACK QR, and other, factorizations (see for example [10]).

In the case of the equivalent ScaLAPACK routine `PDLARFG`, x will typically be distributed over several processors, each of which participates in the computation of $\|x\|_2$ and, if necessary, scales its portion of the vector x and recomputes $\|x\|_2$. From the previous discussion we can see that we clearly need to take care here, or else, in close cases, some processors may attempt to recompute $\|x\|_2$, while others do not, leading to completely erroneous results, or even deadlock. This care will be exercised when ScaLAPACK is able to call the version of the BLACS that support caching, as discussed at the end of Section

4. The hope is that this will occur for Version 2.0 of ScaLAPACK. We could of course solve the problem now by using the idea mentioned above of a controlling process, but this would involve a rather heavy communication burden, and we prefer to wait until we can use the more efficient solution based upon the BLACS. Although failure is very unlikely and indeed we have not yet been able to find an example that fails without artificially altering the `PDLARFG` code, the possibility of failure exists.

Whilst we could not find an example that failed without altering the code, we were able to experimentally simulate such a heterogeneous failure, using the current version of ScaLAPACK[7], by performing the QR factorization of a 6 by 6 matrix A such that

$$A = \delta \begin{pmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{pmatrix}, \quad \delta \text{ small}$$

We took $\delta = \texttt{sfmin}$, which is $\approx 10^{-38}$ on an IEEE machine. The value of `sfmin` is used in `PDLARFG` to determine whether or not to scale the vector, and we artificially adjusted the value so that $\texttt{sfmin} \leftarrow 2 * \texttt{sfmin}$ on one of the processes involved in the scaling decision. As expected, the execution of the factorization hung.

As the second, and somewhat harder problem consider the method of bisection for finding the eigenvalues of symmetric matrices performed by the ScaLAPACK routine `PDSYEVX`. In this algorithm, the real axis is broken into disjoint intervals to be searched by different processes for the eigenvalues contained in each interval. Disjoint intervals are searched in parallel. The algorithm depends on a function, say `count(a,b)`, that counts the number of eigenvalues in the half open interval [`a`, `b`). Using `count`, intervals can be subdivided into smaller intervals containing eigenvalues until the intervals are narrow enough to declare the eigenvalues they contain as being found. The problem here is that two processors may not agree on the boundary between their intervals. This could result in multiple copies of eigenvalues if intervals overlap, or missing eigenvalues if there are gaps between intervals. Furthermore, the `count` function may count differently on different processors, so an interval [`a`, `b`) may be considered to contain 1 eigenvalue by processor A, but 0 eigenvalues by processor B, which has been given the interval by processor A during load balancing. This can happen even if processors A and B are identical in hardware terms, but if the compilers on each one generate slightly different code sequences for `count`. In this example we have not yet decided what to do about all these problems, so we currently only guarantee correctness of `PDSYEVX` for networks of processors with identical floating point formats (but slightly different floating point operations turn out to be acceptable). See [4] for further discussion. Assigning the work by index rather than by range and sorting all the eigenvalues at the end may give the desired result with modest overhead. Of course, if floating point formats differ across processors, sorting is a problem in itself. This requires further investigation.

The symmetric eigensolvers, `PDSYEVX` and `PZHEEVX`, may also have trouble on heterogeneous networks when a subset of eigenvalues is chosen by value (i.e. `RANGE='V'`) and one of the limits of that range (`VL` or `VU`) is within a couple of units in the last place (ulps) of an actual eigenvalue. The two processors may then disagree on the number of eigenvalues specified by the range `VL` and `VU` and the code breaks with each process returning $\texttt{INFO} \neq 0$

[7] Version 1.2

(which is the LAPACK and ScaLAPACK failure indicator). This situation can happen when running the test code and should again be corrected in the next release. In every case that we have seen, the answer is correct despite the spurious error message. This is not a problem on homogeneous systems.

The third example is based upon the idea that some algorithms can perform redundant work in order to gain parallelism. While redundant work on different processors is intended to yield identical results, this may not be the case in a heterogeneous environment. For instance, one approach for parallelizing the symmetric eigenproblem is to perform the tridiagonal QR algorithm to reduce the tridiagonal matrix to diagonal form redundantly on all processors, save the plane rotations, and then accumulate the resulting Givens rotations in parallel into the relevant columns of the unit matrix. This results in $O(n^2)$ redundant work, but $O(n^3)$ parallel work, and requires no communication. Since the QR algorithm is not in general forward stable, slight differences in the underlying arithmetic can lead to completely different rotations and hence the danger of obtaining quite inconsistent eigenvectors. This problem can be solved by having a controlling process that runs the QR algorithm and then broadcasts the plane rotations to the other processes, but the communication cost is substantial: $O(n^2)$.

9 Closing Remarks

We have tried to illustrate some of the potential difficulties concerned with floating point computations on heterogeneous networks. Some of these difficulties are straightforward to address, while others require considerably more thought. All of them require some additional level of defensive programming to ensure the usual standards of reliability that users have come to expect from packages such as LAPACK and the NAG Libraries.

We have suggested reasonably straightforward solutions to the problems associated with floating point machine parameters and global values, and have suggested the use of a controlling process to solve some of the difficulties of algorithmic integrity. This can probably be used to solve most of these problems, but in some cases at the expense of considerable additional overhead, usually in terms of additional communication, which is also imposed on a homogeneous network unless we have separate code for the homogeneous case. Unless we can devise a satisfactory test for homogeneity and hence have separate paths within the code, separate code would defeat the aim of portability.

A topic that we have not discussed is that of the additional testing necessary to give confidence in heterogeneous environments. The testing strategies that are needed are similar to those already employed in reputable software packages such as LAPACK, but it may be very hard to produce actual test examples that would detect incorrect implementations of the algorithms because, as we have seen, the failures are likely to be very sensitive to the computing environment, and in addition may be non-deterministic.

The LAPACK and ScaLAPACK software is available from Netlib[8], as is the documentation and the LAPACK Working Notes. A number of the other references in the bibliography can also be found via Netlib, particularly [1], [9] and [18].

[8] `http://www.netlib.org/`

10 Acknowledgments

We wish to thank all of our ScaLAPACK and NAG colleagues for a number of useful discussions on heterogeneous computing and their valuable input to this paper.

Appendix A – Example Program

The following code is intended to illustrate possible failure when a processor receives a subnormal number, but may not itself (by default) handle such numbers.

The example constructs a one by two grid with process identifiers (0,0) and (0,1), and assumes that process (0,0) is running on a processor that generates IEEE subnormal numbers. For (possible) failure to occur process (0,1) should be running on a processor that does not support subnormal numbers.

We have observed failure when (0,0) is running on a Sun4 (which handles subnormal numbers correctly), and process (0,1) is running on a DEC Alpha under Unix, which by default flushes subnormal numbers to zero. (The non-default compiler flag -fpe1 will trap to software emulation.)

The program utilizes the BLACS. See [8] for further details on the BLACS.

```
      PROGRAM SUBNRM
*
*     .. Local Scalars ..
      INTEGER            IAM, ICNTXT, MYCOL, MYROW, NPCOL, NPROCS, NPROW
      REAL               TWO
*     .. Local Arrays ..
      REAL               X( 1 )
*     .. External Subroutines ..
      EXTERNAL           BLACS_EXIT, BLACS_GET, BLACS_GRIDINFO,
     $                   BLACS_GRIDINIT, BLACS_PINFO, BLACS_SETUP,
     $                   SGERV2D, SGESD2D
*     ..
*
*     Determine my process number and the number of processes in
*     machine
*
*     .. Executable Statements ..
      CALL BLACS_PINFO( IAM, NPROCS )
*
*     If underlying system needs additional setup, do it now
*
      IF( NPROCS.LT.1 ) THEN
         IF( IAM.EQ.0 ) THEN
            NPROCS = 2
         END IF
         CALL BLACS_SETUP( IAM, NPROCS )
```

```
      END IF
*
*     Set up a 1 by 2 process grid
*
      NPROW = 1
      NPCOL = 2
*
*     Get default system context, and initialize the grid
*
      CALL BLACS_GET( 0, 0, ICNTXT )
      CALL BLACS_GRIDINIT( ICNTXT, 'Row-major', NPROW, NPCOL )
      CALL BLACS_GRIDINFO( ICNTXT, NPROW, NPCOL, MYROW, MYCOL )
*
*     If I am in the grid perform some computation
*
      IF( MYROW.GE.0 .AND. MYROW.LT.NPROW ) THEN
*
         TWO = 2.0E+0
         IF( MYROW.EQ.0 .AND. MYCOL.EQ.0 ) THEN
            X( 1 ) = 7.52316390E-37
            X( 1 ) = X( 1 ) / 128.0E+0
*        X(1) = 0.58774718E-38, which is subnormal on IEEE machines
*
*        This call to SGESD2D sends X(1) to process (0,1)
            CALL SGESD2D( ICNTXT, 1, 1, X, 1, 0, 1 )
            WRITE( *, FMT = '(A,E16.8)' )'X00 = ', X( 1 )
            X( 1 ) = X( 1 ) / TWO
            WRITE( *, FMT = '(A,E16.8)' )'X00 / 2 = ', X( 1 )
*
         ELSE IF( MYROW.EQ.0 .AND. MYCOL.EQ.1 ) THEN
*
*        This call to SGERV2D receives X(1) from process (0,0)
            CALL SGERV2D( ICNTXT, 1, 1, X, 1, 0, 0 )
            WRITE( *, FMT = '(A,E16.8)' )'X01 = ', X( 1 )
            X( 1 ) = X( 1 ) / TWO
            WRITE( *, FMT = '(A,E16.8)' )'X01 / 2 = ', X( 1 )
*
         END IF
      END IF
*
*     Exit the BLACS cleanly
*
      CALL BLACS_EXIT( 0 )
*
      STOP
```

```
      END
```

References

[1] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 2nd edition, 1995. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[2] J. Choi, J. Demmel, I. Dhillon, J. J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. In J. J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 95–106. Springer-Verlag, Berlin, Germany, 1995. (Proceedings of the Second International Workshop, PARA '95, Lyngby, Denmark. See also LAPACK Working Note No.95).

[3] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. In J. J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 107–114. Springer-Verlag, Berlin, Germany, 1995. (Proceedings of the Second International Workshop, PARA '95, Lyngby, Denmark. See also LAPACK Working Note No.100).

[4] J. Demmel, I. Dhillon, and H. Ren. On the correctness of parallel bisection in floating point. *ETNA*, 3:116–149, 1995. (See also LAPACK Working Note No.70).

[5] J. Demmel, J. J. Dongarra, S. Hammarling, S. Ostrouchov, and K. Stanley. The dangers of heterogeneous network computing: Heterogenous networks considered harmful. In *Proceedings Heterogeneous Computing Workshop '96*, pages 64–71. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[6] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990. (Algorithm 679).

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–32, 399, 1988. (Algorithm 656).

[8] J. J. Dongarra and R. C. Whaley. A users' guide to the BLACS v1.0. LAPACK Working Note No.94. Technical Report CS-95-281, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA, 1995.

[9] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[10] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, 2nd edition, 1989.

[11] S. Hammarling. Parallel library work at NAG. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 172–182. SIAM, Philadelphia, PA, USA, 1994. (Proceedings of the Second Workshop, Townsend, TN, USA).

[12] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA, USA, 1996.

[13] IEEE. *ANSI/IEEE Standard for Binary Floating Point Arithmetic: Std 754-1985.* IEEE Press, New York, NY, USA, 1985.

[14] IEEE. *ANSI/IEEE Standard for Radix Independent Floating Point Arithmetic: Std 854-1987.* IEEE Press, New York, NY, USA, 1987.

[15] IEEE. *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors: Std 1596.5-1993.* IEEE Press, New York, NY, USA, 1994.

[16] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[17] K. McDonald. The NAG numerical PVM library. In J. J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 419–428. Springer-Verlag, Berlin, Germany, 1995. (Proceedings of the Second International Workshop, PARA '95, Lyngby, Denmark).

[18] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1996.

[19] SunSoft. *The XDR Protocol Specification. Appendix A of "Network Interfaces Programmer's Guide".* SunSoft, 1993.