Tutorial for the NAG Parallel Library (PVM-based Version)

Contents

1   Using this Tutorial

The aim of this Tutorial is to help you to use the NAG Parallel Library. Please take your time to read this section of the manual. The NAG Parallel Library comes in two versions, depending on which software is used for the underlying message-passing mechanism. One version uses PVM, the other uses MPI. You will need to know which version you will be using.

This version of the Tutorial describes the PVM version of the Library; see Section 2.1 for more information about PVM. A separate version of the Tutorial describes the MPI version. The rest of the documentation applies to both the PVM and MPI versions.

The NAG Parallel Library is designed to simplify problem solving on parallel machines as far as possible, and is intended for three classes of users:

 (1)   those who want to call the Library routines as part of their larger parallel code;

 (2)   those who want to use the Library routines in isolation;

 (3)   those who have codes which utilize sequential libraries (such as the NAG Fortran Library) and wish to replace the sequential code sections with calls to the equivalent NAG Parallel Library routines.

In any of the above cases the following assumptions are made:

   you are familiar with the Fortran 77 language;

   you are working in a Unix environment;

   you have a copy of PVM 3.3 (or a later version) installed on your computer.

The Tutorial presents a series of simple example programs, illustrating the use of the Library. The numerical problems which have been chosen for the examples should be simple to understand. This Tutorial aims neither to instruct you in the numerical background of the chosen problems nor to instruct you in the techniques utilized in parallelization in general. Note that the examples are intended to be followed in order, one after the other.

You will need to refer to the documentation of the routines which are used in the examples.

If you run the example programs presented in this Tutorial using your own compiler and machine, you should not expect to get exactly the same results as those which are reproduced in this document, especially when numerical values are printed to high precision.

All the information presented in this Tutorial is also presented in a more condensed form in the **Essential Introduction**. You are definitely **not** expected to read the Essential Introduction before reading this Tutorial. But after working through the Tutorial, you should find the Essential Introduction helpful for reference purposes.

2   What are PVM and BLACS?

This version of the NAG Parallel Library is based on PVM and the BLACS communication mechanisms. This section will give you a short overview of these communication mechanisms and their relationship. If you are already familiar with these mechanisms, you may proceed to the next section.

2.1   PVM

PVM stands for Parallel Virtual Machine and is a public domain software system intended to simplify parallel programming on a network of UNIX machines, or on truly parallel machines. PVM is a 'message-passing' library. Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. The aim of PVM is to take machines that are already available, and enable them to be used collectively as a parallel processing environment. PVM consists of a server process, called the PVM daemon or **pvmd**, and the PVM Library, which contains user callable routines for message passing, process control and synchronization. Implementations of PVM are available for most common UNIX workstations and many parallel machines. In addition, some manufacturers provide efficient machine-specific implementations. You are referred to [2] for more information on this subject.

A public domain version of PVM may be obtained from `netlib`[1] .

---

[1] If you are not familiar with netlib, send the e-mail message `help` to `netlib@ornl.gov`, or look up the web-page `http://www.netlib.org`

2.2   BLACS

The BLACS (Basic Linear Algebra Communication Subprograms) are also a collection of routines that provide a mechanism for message passing on parallel machines. The BLACS are composed of a series of communication primitives, global operations and support routines. The NAG Parallel Library makes extensive calls to the BLACS. The BLACS constitute a linear algebra oriented message-passing interface that may be implemented efficiently and uniformly across a broad range of distributed memory platforms. The BLACS exist in order to make linear algebra applications both easier to program and more portable. It is for this reason that the BLACS are used as the communication layer of ScaLAPACK, a linear algebra package targeted at distributed memory machines. We will cover ScaLAPACK in greater detail in Section 10. One of the main strengths of the BLACS is that it shields the developer from the underlying message-passing system (PVM, MPI etc.) by providing a consistent interface to it and as a consequence increases the portability of applications.

3   How to Run Programs which Call Library Routines

It is important that you read and understand this and Section 4. They give you information about what you need to know before you start using the NAG Parallel Library.

3.1   About the Library

This version of the NAG Parallel Library is based on PVM and the BLACS communication mechanisms, and is intended primarily for distributed memory parallel machines, including networks and clusters of machines. It can also be used on shared memory systems which have an efficient implementation of PVM. The Library can also be run on single processor machines (provided PVM is installed).

The Library supports parallelism and memory scalability and has been designed to be portable across a wide range of parallel machines. But the Library cannot be expected to achieve near peak performance for all problems on all machines, and in particular algorithmic scalability is unlikely across the whole Library.

The routines in this Library fall into two categories:

Numerical routines – the routines which you call to carry out computation in the area(s) in which you are interested;

Utility routines – the routines which you call to help you to spawn processes on your machine, distribute data, exit PVM and/or the BLACS, input/output matrices, etc. These routines can be found in the F01, X04 and Z01 chapters of the Library.

Before going any further, here are some definitions of terminology which will be used throughout this Tutorial. These terminologies will be discussed in more detail in the appropriate sections.

Root processor	The logical processor started by the user (i.e., at the keyboard or from a script). In PVM terminology this is the process with no parents.
SPMD model	Stands for Single Program Multiple Data. In this model the same program is executed on all processors of a system, but the execution may follow different paths through the program on different processors.
Spawning	A term used in PVM when further processes are invoked from within the user program (either directly using PVMFSPAWN or by a call to one of the NAG Parallel Library utility routines). In the SPMD model of parallelism, these additional processes will be identical copies of the user program.
Processor Grid	A virtual machine consisting of logical processors arranged in a two-dimensional array.
Library Grid	Essentially a Processor Grid created by the mechanism in the NAG Parallel Library. This mechanism can be invoked by some utility routines in the Library.

In order to run a program which makes use of the NAG Parallel Library, you must be able to start PVM, configure a 'parallel virtual machine' and, on completion, halt PVM.

3.2   Starting PVM

You must start PVM before running a program which uses a NAG Parallel Library routine. Starting PVM involves executing either the `pvm` or `pvmd` commands. Without arguments these commands start up the PVM daemon on the local host. The two methods of running PVM are shown here.

(1)   Issue the command

   `% pvm`

(`%` denotes a shell prompt) to invoke the PVM console and produce the prompt

   `pvm$>$`

which then accepts commands from the standard input. At this point you may add further hosts (or nodes in a parallel machine), making them available to PVM. The command `help` gives you a summary of the available commands. You may then use 'quit' command to quit from the PVM console, leaving the PVM daemon running on the specified hosts (nodes).

(2)   Issue the command

   `% pvmd hostfile &`

to start PVM in the background where `hostfile` is a file specifying the hosts (or nodes) to be included in the virtual machine.

PVM may be started and stopped many times on any of the hosts on which it is running (although typically a user starts only one console). PVM may be stopped by returning to the PVM console (via the `pvm` command) and issuing a `halt` command or by killing the background `pvmd` process.

3.3   Executing a PVM Application Program

Suppose PVM is started by one of the methods described above. In order to execute a program, PVM looks for your executable in the default location `$HOME/pvm3/bin/PVM_ARCH`, where `$HOME` indicates your home directory and `PVM_ARCH` is the architecture of your host. For example, if your PVM application wants to create (spawn) a task called `myex` on a Silicon Graphics workstation (running operating system IRIX 5.x) called 'sgi11', then 'sgi11' must be able to access an executable file `$HOME/pvm3/bin/SGI5/myex`. However, if you wish, you can change this default to a different search path in the hostfile. See [2] to learn how this can be done.

3.4   Accessing the Library

In order to use the NAG Parallel Library, you will need to know how to compile a Fortran 77 program, link it to the appropriate Libraries provided in the NAG Parallel Library (e.g. BLACS, PBLAS) and the appropriate implementation of PVM, and execute it. These details can be found in the Users' Note, and may vary from one computing system to another, and in some respects from one installation to another. You also must refer to local information provided by your installation. We assume that you have this information available whenever you want to execute a program which calls the NAG Parallel Library routines.

In the rest of the Tutorial, the term **routine** is used to refer to both subroutines and functions.

4   Conventions Assumed by the Library and its Documentation

4.1   Finding the Documentation You Need

The Library is divided into **chapters**, each devoted to a branch of numerical analysis or statistics. Each chapter has a three-character name and a title, e.g.

   D01 – Quadrature

(The chapters and their names are based on the ACM modified SHARE classification index [3].) Most documented routines in the Library have seven-character names. Although seven-character names are not standard Fortran 77 [4] , they are a supported extension of any compiler capable of supporting PVM and the BLACS. These routines names always begin with the characters of the chapter name, e.g.

D01AUFP

Note that the second and third characters are **digits**, not letters; e.g. 0 is the digit zero, not the letter O. The last letters of each routine name are always 'FP'. The chapters occur in alphanumeric order.

Therefore, in order to use the routine D01AUFP, you must refer to Chapter D01 of the Library.

Chapter F07 (Linear Equations (ScaLAPACK)) and Chapter F08 (Least-squares Problems (ScaLAPACK)) contain routines derived from ScaLAPACK. These routines may be referenced either by their NAG-style names or their ScaLAPACK names, e.g. F07ADFP (PDGETRF). Details regarding these alternate names can be found in the relevant Chapter Introductions.

A few of the documented routines have six-character names and are identical to routines, with the same name, in the NAG Fortran Library (a comprehensive mathematical and statistical library in Fortran 77). The naming structure of these routines is as described above, except for the absence of the final 'P' in the routine name.

4.2   The 'Library Grid'

Before we go any further, we describe the way in which the NAG Parallel Library views the concept of a **process**. A process can be viewed as a thread of execution (which minimally includes a stack, registers and memory). In general, multiple processes may share a physical **processor**. Here, the term processor refers to the actual hardware.

In the NAG Parallel Library, each process is treated as if it were running on a separate physical processor. In this case, the process must exist for the lifetime of the NAG Parallel Library routines run, and its execution should only affect the execution of other processes through the message-passing calls. Furthermore, in the NAG Parallel Library, algorithms are presented in terms of processes, rather than physical processors. In general there may be several processes on one processor, in which case we assume that the run-time system handles the schedule of processes. Since the Library does not control the allocation of processes onto processors, it assumes one process for one processor. For this reason, in this Tutorial, we use the term process and processor interchangeably.

The NAG Parallel Library is designed to execute on a **two-dimensional (2-d) grid of logical processors** which we refer to as the **Library Grid** hereafter. This simply means that the Library views a parallel machine as a 2-d rectangular grid of processors. However, whilst the Library assumes this logical topology, no assumption is made about the actual physical topology of the system.

Before calling any Library routine (expect for a few) in the NAG Parallel Library, it is essential that you first set up the Library Grid. There are two ways in which a Library Grid can be created:

(1)   using the utility routine provided in the NAG Parallel Library;

(2)   using the BLACS or PVM directly; in this case we assume that you have some familiarity with the BLACS and/or PVM. This mechanism allows sophisticated application of the NAG Parallel Library, e.g. multigridding, a technique which enables the users to create and use many logical grids simultaneously on their parallel machines. See Section 12.

The above two grid-creation methods can both be invoked by calls to the NAG Parallel Library routine. If you want to use method (1) you must call Z01AAFP and if you want to use method (2) you must call Z01ADFP **and** Z01AAFP.

It is also important that you understand that all the Library routines use certain internal variables:

Communication internal variables – required for use by PVM and the BLACS communication mechanisms;

Library internal variables – required for use by the Library routines.

A call to Z01AAFP initializes the 'communication internal variables' and the 'library internal variables'. It also informs the other Library routines (subsequently called) that the Library Grid has been set up by a call to Z01AAFP, whereas a call to Z01ADFP followed by a call to Z01AAFP informs the Library routines to initialize only the 'communication internal variables' and informs the Library that there is no need to initialize the 'library internal variables' as the Library Grid has already been set up using the BLACS (or PVM).

Hence, both grid-creation methods (1) and (2) may be used to initialize the 'communication internal variables', but **only method (1) initializes** the 'library internal variables'.

It is also important that, once the use of the Library Grid has been completed, all the memory allocated by initializing the above internal variables be released and the Library Grid be un-defined and destroyed.

4.2.1 Defining a Library Grid using Z01AAFP

Z01AAFP sets up a 2-d grid of logical processors on which programs calling NAG Parallel Library routines will execute in parallel. This routine takes the available processes spawned by PVM and maps them onto the Library Grid. A call to Z01AAFP has the following specification:

```
CALL Z01AAFP( ICNTXT, MP, NP, IFAIL )
```

Z01AAFP has **two** explicit functions.

> It spawns the MP × NP processes and creates a 2-d MP by NP Library Grid; the user-supplied input arguments MP and NP specify the dimensions of the grid. MP represents the number of processor rows, and NP represents the number of processor columns.

> It also returns a **context** (ICNTXT) for the Library Grid. The idea of using a context was adopted from the BLACS (used extensively as one of our message-passing mechanisms). Simply, a context can be viewed as an identifier for that Library Grid which was set up by a call to Z01AAFP. This means that processors enclosed in the Library Grid using a particular context can safely communicate even if other (possibly overlapping) grids are also communicating with different contexts. Contexts are used so that individual routines using the BLACS for communication can, when required, safely operate without worrying if other distributed codes are being executed on the same machine. Most of the Library routines use this context to ensure that message passing in the NAG Parallel Library does not interact with any other processor grids created in your program. In summary, a context allows us to:

>> create arbitrary groups of processes upon which to execute (the Library Grid);

>> create an indeterminate number of overlapping or disjoint grids (multigridding);

>> isolate each grid so that grids do not interfere with each other (safe communication).

> **Note:** No attempt should be made to examine or modify the value of a context.

> (As you gain more familiarity with the NAG Parallel Library in this Tutorial, you will realize that almost all routines in the NAG Parallel Library have a context as their first argument, except those routines derived from ScaLAPACK routines which have a context as an element of an integer array.)

As with most of the Library routines, Z01AAFP also returns the integer value IFAIL which reports its success or failure. IFAIL is considered in Section 5.3 and Section 9.3.

4.2.2 Using BLACS and/or PVM for defining a Library Grid

The use of this facility assumes that you know how to initialize a 2-d grid using the BLACS or PVM. If you do not wish to make a direct call to Z01AAFP to set up the Library Grid and wish to use BLACS or PVM routines, then a call to Z01ADFP must be made prior to a call to Z01AAFP. This feature is particularly useful for sophisticated users who need to pass their own generated context to the NAG Parallel Library routines. The use of Z01ADFP also allows multigridding, a technique which allows the users to specify many logical processor grids (possibly overlapping) simultaneously in their program. Z01ADFP need be called only once and that call must be the **first** call to any Library routine. If that is not the case then an error condition is raised by subsequent calls to other Library routines.

4.2.3 Un-defining and Un-initialising a Library Grid: Z01ABFP

When you have finished using the current Library Grid (created by a call to Z01AAFP), you must call Z01ABFP to 'un-define' the grid. This informs the Library mechanism that the current Library Grid is no longer valid and may also optionally exit from the message-passing environment. If you choose to exit from the message-passing environment, you should set the value of CONT = 'N'; any subsequent call to a Library routine will result in a failure and an error condition. If you choose to declare a new Library Grid with a different shape or total number of processors (by varying MP and NP), you must call Z01ABFP with CONT = 'Y' and then call Z01AAFP to declare your new 2-d Library Grid. However, if Z01ADFP was used, it is your responsibility to make calls to PVM or BLACS routines to un-define and destroy the Library Grid.

The Library has a restriction that the number of new processors (MP×NP) must not exceed the total on the first entry to Z01AAFP. An example of grid reshaping will be given in Section 12.1.

This Tutorial mainly uses a Library Grid created by a call to Z01AAFP, and the use of Z01ADFP will only be considered in Section 12.

Hence, a typical call to a NAG Parallel Library routine such as D01AUFP must look like:

```
Call the NAG Parallel Library routine to initialize the Library Grid
      CALL Z01AAFP( ICNTXT,MP,NP,IFAIL )
      .
      .


Call the NAG Parallel Library routine(s) to solve your problem
      CALL D01AUFP( ICNTXT, ... )
      .
      .


Call the NAG Parallel Library to release the Library Grid
      CALL Z01ABFP( ICNTXT,CONT,IFAIL )
```

It is useful to think of Z01AAFP and Z01ABFP as left and right braces around calls to the Library routines.

Exceptions are those Library routines which have no argument, and the NAG Fortran Library routines (routines whose names do not end with 'P'). These routines do not use the Library Grid information set by Z01AAFP, so they may be called prior to a call to Z01AAFP.

In what follows, the Tutorial uses the symbol $\{i, j\}$ to denote the logical processor in row i and column j of the grid, with $\{0, 0\}$ as the processor at the top left corner of the grid. The Library also associates the parent process with the **root processor**, and this is usually processor $\{0, 0\}$.

4.3 Library Model

The Library assumes a Single Program Multiple Data (SPMD) model of parallelism in which a single instance of your program executes on each of the logical processors with (possibly) different data. In the current version of the Library, only one Library Grid is allowed to execute Library routines at any one time, but you may be able to define more than one grid yourself. This is called **multigridding**, and an example will be given in Section 12.2.

4.4 Input/Output in the Library

Most NAG Parallel Library routines perform no output to external files, except possibly to output error messages. **All error messages are written by a single processor (usually the root) to a logical error message unit**. The corresponding unit number (which is set by default to 6 in most implementations) can be changed by calling the Library routine X04AAF.

Some Library routines (e.g. A00AAFP) output messages, only by the root processor, to a logical **advisory message** unit. This unit number (which is also set by default to 6 in most implementations) can be changed by calling the Library routine X04ABF. Although the advisory message unit is logically distinct from the error message unit, in practice the two unit numbers may be the same.

All output from the Library is formatted.

You must ensure that the relevant Fortran unit number is associated with the desired external file, either by an OPEN statement in your calling program, or by operating system commands.

If more than one grid is defined, then you **must** ensure that one of the processors in the Library Grid is attached to the terminal which performs input/output, otherwise error messages may be lost (unless your system supports parallel input/output which is currently beyond the scope of this Tutorial) .

5   Example 1: Displaying Library Information

In this section we consider the simplest task of displaying information about the Library as it is installed on your system (for example, the type of machine, the compiler, the precision, the release). This task is performed by the routine A00AAFP. The routine A00AAFP is trivial: it is a subroutine with no arguments which writes information to the standard output unit attached to the root processor. This routine is one of the few routines in the NAG Parallel Library which **may** be called either **with** calls to or **without** calls to Z01AAFP and Z01ABFP. We illustrate both cases here.

5.1   A First Program

Here is a complete program to call A00AAFP:

```
1:      PROGRAM EX1A
2:      CALL A00AAFP
3:      STOP
4:      END
```

If you try it, you should see something like this appear on the standard output unit (normally the screen):

```
** Start of NAG Parallel Library implementation details **

   Implementation title: Silicon Graphics IRIX 5
           Product Code: FDSG502DP
                Release: 2
              Precision: FORTRAN double precision
Message passing library: Public domain version of PVM

** End of NAG Parallel Library implementation details **
```

You will probably not see exactly the same text, but that is as it should be: the precise details depend on the system on which your program is running. It is enough for now to have verified that you have correctly called a Library routine.

5.2   Displaying Library Information Using Z01AAFP and Z01ABFP

Here is another complete program to call the routine A00AAFP which achieves the same result as Program EX1A:

```
 1:      PROGRAM EX1B
 2:      INTEGER MP,NP,IFAIL,ICNTXT
 3:      CHARACTER*1 CONT
 4:      MP = 2
 5:      NP = 2
 6:      IFAIL = 0
 7:      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
 8:      CALL A00AAFP
 9:      CONT = 'N'
10:      IFAIL = 0
11:      CALL Z01ABFP(ICNTXT,CONT,IFAIL)
12:      STOP
13:      END
```

Note: We reiterate that PVM looks for your executable in the default location `$HOME/pvm3/bin/PVM_ARCH`, where `PVM_ARCH` (= `SGI5` in this case) is the machine architecture name. This default can be changed to a different search path in the hostfile (see your site representative, or the person who installed PVM).

The output from Programs `EX1A` and `EX1B` is identical; however, the programs use different mechanisms to provide the same Library information. In fact Program `EX1B` is executed on a 2 by 2 Library Grid. It should be emphasized that nothing is gained by running A00AAFP in parallel because the Library information is output only on the root processor ($\{0, 0\}$ processor). Despite this fact, Program `EX1B` provides a useful and simple example for describing how Z01AAFP and Z01ABFP may be utilized in your program. Here is the description of Program `EX1B`:

`Lines 2-3:` declarative section of the program.

`Lines 4-5:` MP and NP indicate the number of rows and columns to be used in the Library Grid, respectively. In this example, we have set a 2 by 2 grid.

`Line 7:` the call to Z01AAFP initializes some internal variables and defines a 2 by 2 grid as well as a unique Library context (ICNTXT). This context, which is an argument to most of the Library routines, is used to create an environment for the underlying Library message-passing mechanism. We defer the discussion of how spawning is invoked by Z01AAFP to Section 7.4.

`Line 8:` the call to the Library routine, in this case A00AAFP.

`Lines 9-11:` On completion of calls to NAG Parallel Library routines, it is essential that we exit from the Library mechanism. Z01ABFP frees the internal variables and un-defines the grid context set in Z01AAFP. `Line 9` indicates that the program is not participating in any further message passing.

5.3 A First Look at Run-Time Errors

In this example, we illustrate what happens if you forget to call Z01AAFP in your program but not Z01ABFP. Consider the following example:

```
     *  This program will exit with an error condition
     *
 1:      PROGRAM EX1C
 2:      INTEGER MP,NP,IFAIL,ICNTXT
 3:      CHARACTER*1 CONT
 4:      MP = 2
 5:      NP = 2
 6:      CALL A00AAFP
 7:      CONT = 'N'
 8:      IFAIL = 0
 9:      CALL Z01ABFP(ICNTXT,CONT,IFAIL)
10:      STOP
11:      END
```

The above program is exactly the same as Program `EX1B`, except that the call to Z01AAFP is missing. If you try it, you should see the following output:

```
** Start of NAG Parallel Library implementation details **

   Implementation title:  Silicon Graphics IRIX 5
           Product Code: FDSG502DP
                Release: 2
              Precision: FORTRAN double precision
Message passing library: Public domain version of PVM


** End of NAG Parallel Library implementation details **
** ABNORMAL EXIT from NAG Parallel Library routine Z01ABFP: IFAIL =  -1000
** Z01ABFP has been called without calling Z01AAFP first.
** NAG hard failure - execution terminated
{-1,-1}, pnum=0, Contxt=-666, killed other procs, exiting with error #0.
```

A00AAFP generates the Library information, but the program exits with an error condition with the value of IFAIL = −1000. As the error message indicates, the error was detected in Z01ABFP.

Each error which can be detected by a Library routine is associated with a number assigned to the integer argument IFAIL (−1000 in this case). These numbers, with explanations of the errors, are listed in Section 5 (Error Indicators and Warnings) in the routine document. If you look in Section 5 of the document for Z01ABFP, the following description can be found:

IFAIL = −1000

> The logical processor grid and library mechanism (Library Grid) have not been correctly defined, see Z01AAFP.

This is indeed the case. Since A00AAFP is a special routine with no argument, it can be run independently of the routines Z01AAFP and Z01ABFP (as was illustrated in Program `EX1A`); hence the Library information appears. Remember that a call to Z01AAFP is required to initialize and define some internal variables, a logical grid and a Library context. The information generated by Z01AAFP is internally checked and utilized by most Library routines. In Program `EX1C`, Z01ABFP is the first routine in the program which requires the use of this information. Since this information was not available, Z01ABFP detected this, output an informative error message, and halted the program. The final line of the error message:

    {-1,-1}, pnum=0, Contxt=-666, killed other procs, exiting with error #0.

is output from a BLACS routine (BLACS_ABORT) which kills the processes associated with the Library Grid. We will come back to this later.

This is the cue to start explaining how run-time errors are handled by the Library. The error, failure or warning conditions considered here are those that can be detected by explicit coding in a Library routine. They should not be confused with run-time errors which may be detected by your system, e.g. detection of overflow or failure to assign an initial value to a variable.

In the rest of this Tutorial we use the word 'error' to cover all types of error, failure or warning conditions detected by a routine. They fall roughly into four classes:

  (i)   Essential initialization routines were not called. This means that it is not possible to begin computation.

  (ii) On entry to the routine the value of an argument is out of range or inconsistent with other processors. This means that it is not useful, or perhaps not even meaningful, to begin computation.

  (iii) During computation the routine decides that it cannot yield the desired results, and indicates a failure condition. For example, a Black Box routine for solving a system of linear equations will indicate a failure condition if it considers that the matrix is singular.

  (iv) Although the routine completes the computation and returns results, it cannot guarantee that the results are completely reliable; it therefore returns a warning. For example, an optimization routine may return a warning if it cannot guarantee that it has found a local minimum.

All four classes of errors are handled in the same way by the Library.

Most of the NAG Parallel Library routines which you can call directly have an integer argument called IFAIL. As you may already have guessed, this argument is concerned with the Library error trapping mechanism (and, for some routines, with controlling the output of error messages and advisory messages). You should have noticed this argument in the documentation of Z01AAFP and Z01ABFP. The specification of IFAIL is a standard text which appears in the documentation of most NAG Parallel Library routines. It reads:

IFAIL — INTEGER   *Global Input/Global Output*

On entry: IFAIL must be set to 0, −1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended values are:

IFAIL = 0, if multigridding is **not** employed;

IFAIL = −1, if multigridding is employed.

On exit: IFAIL = 0 unless the routine detects an error (see Section 5).

IFAIL has **two** purposes:

(i)   to allow the user to specify what action the Library routine should take if an error is detected;

(ii)   to inform the user of the outcome of the call of the routine.

For purpose (i), the user **must** assign a value to IFAIL before the call to the Library routine. Since IFAIL is reset by the routine for purpose (ii), the argument must be the name of a variable, **not** a literal or constant. For example,

```
CALL ZO1AAFP(ICNTXT,MP,NP,0)
```

will result in unpredictable behaviour. The correct way to use the argument IFAIL is:

```
IFAIL = 0
CALL  ZO1AAFP(ICNTXT,MP,NP,IFAIL)
```

The value assigned to IFAIL before entry should be either 0 (**hard fail** option), or 1 or −1 (**soft fail** option). If after completing its computation the routine has not detected an error, IFAIL is reset to 0 to indicate a **successful call** and control returns to the calling program in the normal way. If the routine does detect an error, its action depends on whether the hard or soft fail option was chosen. We shall leave it until Section 9.2 to explain how we can handle a soft fail option.

As its description suggests, IFAIL is of type integer, and is classified as *Global Input/Global Output*. Before continuing with further examples, it is helpful to understand the way in which routine arguments are classified in the NAG Parallel Library. This is described in the next section.

6   Arguments of Library Routines

Section 4 of each routine document contains the specification of the arguments, in the order of their appearance in the argument list.

Arguments in the NAG Parallel Library are classified as follows:

Input: you must assign values to these arguments before entry to the routine, and these values are unchanged on exit from the routine.

Output: you need not assign values to these arguments on or before entry to the routine; the routine assigns values to them.

Input/Output: you must assign values to these arguments before entry to the routine, and the routine may then change these values.

Workspace: array arguments which are used as workspace by the routine. You must supply arrays of the correct type and dimension. In general, you need not be concerned with their contents.

External Procedure: a subroutine or function which must be supplied (e.g. to evaluate an integrand or to print intermediate output). Usually it must be supplied as part of your calling program, in which case its specification includes full details of its argument list and specifications of its arguments (all enclosed in a box). Its arguments are classified in the same way as those of the Library routine, but because you must write the procedure rather than call it, the significance of the classification is different:

> *Input*: values may be supplied on entry, which your procedure **must not** change.
> *Output*: you should assign values to these arguments before exit from your procedure.
> *Input/Output*: values may be supplied on entry, and you should assign values to them before exit from your procedure.

User Workspace: array arguments which are passed by the Library routine to an external procedure argument. They are not used by the routine, but you may use them to pass information between your calling program and the external procedure.

In addition arguments are also classified as being either *Local* or *Global*.

Local: local input arguments may have different values on different logical processors on entry to the Library routine. Local output arguments may be assigned different values on different logical processors on exit from a Library routine. The significance of these values is given in the routine documentation.

Global: global input arguments **must** have the same value on each logical processor on entry to a Library routine. If this is not the case then the Library routine exits with an error condition. For global input array arguments this condition applies element-wise. Global output arguments will be assigned the same value on all processors on exit from a Library routine.

As an example the argument IFAIL is classified as *Global Input/Global Output.* This essentially means: on entry to those routines which have IFAIL as an argument, when IFAIL is set to 0, say, then all the participating processors in the Library Grid must also have the value of IFAIL set to 0, and on exit it will have the same value on all the processors.

7 Example 2: Generating Random Numbers

For our next example, we take another simple task, but one that does involve some numerical computation, namely to generate a sequence of random numbers from a uniform distribution over the interval (0,1).

7.1 Searching for Random Number Generating Routine

If you look up 'pseudo-random' in the **Keywords in Context (KWIC)** index, you will be referred to the routine G05AAFP, which generates random numbers. The name indicates that you should refer to Chapter G05 (as described in Section 4.1).

The chapter starts with a Chapter Introduction with the title 'Random Number Generators' which is followed by a Chapter Contents which lists the available routines in Chapter G05. If you search for G05AAFP in the Chapter Contents, you will find the routine name and a one-line description which confirms that G05AAFP generates pseudo-random real numbers from a uniform (0,1) distribution using a Wichmann–Hill generator. G05AAFP is another example of a Library routine which has no argument. Therefore, it may be used prior to a call to Z01AAFP, after a call to Z01ABFP, or without any calls to Z01AAFP and Z01ABFP in your program.

7.2 Functions in the Library

The specification section of the document for G05AAFP states that it is a double precision **function** with no argument. The 'Specification' section reads:

```
DOUBLE PRECISION FUNCTION G05AAFP()
```

Before going any further, we need to say something about the precision in which the Library is implemented. The NAG Parallel Library is implemented principally in 'double precision'. The information displayed by A00AAFP should confirm this. Hence, assuming that your implementation is double precision, you must make sure that, in your program, all floating-point arguments are declared of type 'double precision', either with an explicit `DOUBLE PRECISION` declaration, or by using a suitable `IMPLICIT` statement. Using double precision in all the programs, you can compile and run the following program to generate a sequence of four random numbers:

```
1:      PROGRAM  EX2A
2:      INTEGER          N
3:      PARAMETER        (N = 4)
4:      INTEGER          I
5:      DOUBLE PRECISION RAND(N),G05AAFP
6:      DO 10 I = 1, N
7:         RAND(I) = G05AAFP()
8:  10  CONTINUE
9:      WRITE (6,*) (RAND(I), I = 1, N)
```

```
10:       STOP
11:       END
```

If you run the above **sequential** program, you should see something similar to the the following on your
screen:

```
0.9760044636156003    0.3584819212790158    0.7684902391334956
0.9150123074167526
```

7.3   A First Parallel Numerical Program

Suppose that you need to generate identical sequences of random numbers on each processor in a Library
Grid. This can be done in a very simple manner! Before a call to G05AAFP you need to set up a Library
Grid. You can easily achieve this by first specifying the number of rows and columns in the grid, followed
by a call to Z01AAFP. The following program generates identical sequences of four random numbers on
each processor in a 2 by 2 Library Grid:

```
 1:       PROGRAM  EX2B
 2:       INTEGER          N
 3:       PARAMETER        (N = 4)
 4:       INTEGER          I,ICNTXT,IFAIL,MP,NP
 5:       DOUBLE PRECISION RAND(N),G05AAFP
 6:       MP = 2
 7:       NP = 2
 8:       IFAIL = 0
 9:       CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
10:       DO 10 I = 1, N
11:          RAND(I) = G05AAFP()
12:  10  CONTINUE
13:       IFAIL = 0
14:       CALL Z01ABFP(ICNTXT,'N',IFAIL)
15:       STOP
16:       END
```

It is also possible to generate different sequences of random numbers on different processors; this is
illustrated in Section 7.6.

If you try to run Program `EX2B`, no output is produced. The program generates four random numbers
on each processor and then stops. `Lines 6-7` specify the dimensions of a 2 by 2 Library Grid, which is
set up by a call to Z01AAFP at `line 9`. Z01AAFP also defines a Library context. Calls to G05AAFP
at `lines 10-12` generate the random numbers. Since no further action is required in the program after
generating the random numbers, Z01ABFP is called at `line 14` to un-define the Library Grid and the
associated context.

We now take a closer look at the way in which parallelism is invoked by Program `EX2B`.

7.4   A Closer Look at Z01AAFP

We remind you that the SPMD paradigm simply means that the same program is run on each processor.
Taking Program `EX2B` as our example, we will illustrate how the same program is mapped onto each
processor.

It is important to know that, in general, the spawning is invoked on the root processor and is achieved by
a call to Z01AAFP. When you start running Program `EX2B`, the root processor initially executes all the
instructions until a call to Z01AAFP is made. At this point Z01AAFP spawns a further three copies of
Program `EX2B` onto other available processors and waits until each program on the spawned processors
calls Z01AAFP. At this stage all the processors start to execute the same program independently.

Figure 1 gives a clear pictorial description of how Z01AAFP works in Program `EX2B`; the heavy arrows
indicate the spawning action and each box represents a processor.

```
      PROGRAM EX2B
      .
      .
      MP = 2
      NP = 2
      IFAIL = 0
      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
      DO 10 I = 1, N
         RAND(I) = G05AAFP()
 10   CONTINUE
      IFAIL = 0
      CALL Z01ABFP(ICNTXT,'N',IFAIL)
      STOP
      END
             Processor {0,1}
```

```
      PROGRAM EX2B
      .
      .
      MP = 2
      NP = 2
      IFAIL = 0
      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
      DO 10 I = 1, N
         RAND(I) = G05AAFP()
 10   CONTINUE
      IFAIL = 0
      CALL Z01ABFP(ICNTXT,'N',IFAIL)
      STOP
      END
      Processor {0,0}: the root
```

```
      PROGRAM EX2B
      .
      .
      MP = 2
      NP = 2
      IFAIL = 0
      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
      DO 10 I = 1, N
         RAND(I) = G05AAFP()
 10   CONTINUE
      IFAIL = 0
      CALL Z01ABFP(ICNTXT,'N',IFAIL)
      STOP
      END
             Processor {1,0}
```

```
      PROGRAM EX2B
      .
      .
      MP = 2
      NP = 2
      IFAIL = 0
      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
      DO 10 I = 1, N
         RAND(I) = G05AAFP()
 10   CONTINUE
      IFAIL = 0
      CALL Z01ABFP(ICNTXT,'N',IFAIL)
      STOP
      END
             Processor {1,1}
```

Figure 1
Library spawning mechanism with an SPMD model

Note: You can also view the call to Z01AAFP in your program as a synchronization point: **execution will not proceed beyond this point until all processors have reached it.** It is also possible to set the row and column numbers (MP and NP) of the Library Grid only on the root processor. In this case the call to Z01AAFP on the root processor will broadcast these values to the other processors. To do this, you must know how to identify the root processor in your Library Grid.

7.5   Z01ACFP: A Routine for Identifying the Root Processor

In this section we introduce another useful utility routine, namely Z01ACFP, which identifies the root processor in your Library Grid. Since input and output normally occurs through the root processor (see Section 4.4), Z01ACFP is a very useful. Z01ACFP is a logical function and has no argument, so it can be called anywhere in your program, even before a call to Z01AAFP. When called by all processors in the Library Grid, Z01ACFP returns .TRUE. on the root processor and .FALSE. on the others.

7.6 Setting Processor Rows and Columns on the Root Processor

If you look up the 'Arguments' section of Z01AAFP, the specification of arguments MP and NP are as follows:

MP — INTEGER *Local Input/Global Output*

On entry: the number of rows, m_p, in the logical processor grid will be taken to be the value supplied on the root processor (usually the {0,0} processor and identifiable by a call to Z01ACFP).

On exit: the value supplied to this routine on the root processor will be returned on all other processors. The value is unchanged on the root processor.

NP — INTEGER *Local Input/Global Output*

On entry: the number of columns, n_p, in the logical processor grid will be taken to be the value supplied on the root processor (usually the {0,0} processor and identifiable by a call to Z01ACFP).

On exit: the value supplied to this routine on the root processor will be returned on all other processors. The value is unchanged on the root processor.

As the specifications suggest, MP and NP are classified as *Local Input/Global Output*. On entry to Z01AAFP, their values must be set on the root processor, but need not be the same on all other participating processors. However, on exit, all the processors will have the same values, namely the values of MP and NP on the root processor.

The following example has the same functionality as Program `EX2B`.

```
 1:      PROGRAM  EX2C
 2:      INTEGER         N
 3:      PARAMETER       (N = 4)
 4:      INTEGER         I,ICNTXT,IFAIL,MP,NP
 5:      DOUBLE PRECISION RAND(N),G05AAFP
 6:      LOGICAL Z01ACFP
 7:      IF (Z01ACFP()) THEN
 8:         MP = 2
 9:         NP = 2
10:      END IF
11:      IFAIL = 0
12:      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
13:      DO 10 I = 1, N
14:         RAND(I) = G05AAFP()
15:  10  CONTINUE
16:      IFAIL = 0
17:      CALL Z01ABFP(ICNTXT,'N',IFAIL)
18:      STOP
19:      END
```

A pictorial description of the execution path of this program, on different processors, is given in Figure 2. It is evident from `lines 7-10` of Program `EX2C` that MP and NP are initially set on the root processor only. Then, a call to Z01AAFP on `line 12` internally broadcasts these values to other processors in the Library Grid.

Before leaving this section, as a final example, we develop a more elaborate program which reads the values of MP and NP from the root processor (standard input) and writes the random numbers, generated by calls to G05AAFP, on the root processor (standard output):

```
 1:      PROGRAM  EX2D
 2:      INTEGER         N
 3:      PARAMETER       (N = 4)
 4:      INTEGER         I,ICNTXT,IFAIL,MP,NP
 5:      DOUBLE PRECISION RAND(N),G05AAFP
 6:      LOGICAL Z01ACFP
```

```
     PROGRAM EX2C
     .
     .
     IFAIL = 0
     CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
     DO 10 I = 1, N
        RAND(I) = G05AAFP()
  10 CONTINUE
     IFAIL = 0
     CALL Z01ABFP(ICNTXT,'N',IFAIL)
     STOP
     END
```
Processor {0,1}

```
     PROGRAM EX2C
     .
     .
     MP = 2
     NP = 2
     IFAIL = 0
     CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
     DO 10 I = 1, N
        RAND(I) = G05AAFP()
  10 CONTINUE
     IFAIL = 0
     CALL Z01ABFP(ICNTXT,'N',IFAIL)
     STOP
     END
```
Processor {0,0}: the root

```
     PROGRAM EX2C
     .
     .
     IFAIL = 0
     CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
     DO 10 I = 1, N
        RAND(I) = G05AAFP()
  10 CONTINUE
     IFAIL = 0
     CALL Z01ABFP(ICNTXT,'N',IFAIL)
     STOP
     END
```
Processor {1,0}

```
     PROGRAM EX2C
     .
     .
     IFAIL = 0
     CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
     DO 10 I = 1, N
        RAND(I) = G05AAFP()
  10 CONTINUE
     IFAIL = 0
     CALL Z01ABFP(ICNTXT,'N',IFAIL)
     STOP
     END
```
Processor {1,1}

Figure 2
Different execution paths in SPMD paradigm.

```
    7:      IF (Z01ACFP()) THEN
    8:         WRITE (6,*) 'Enter MP and NP'
    9:         READ (5,*) MP, NP
   10:      END IF
   11:      IFAIL = 0
   12:      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
   13:      DO 10 I = 1, N
   14:         RAND(I) = G05AAFP()
   15:  10  CONTINUE
   16:      IF (Z01ACFP())  WRITE (6,*) (RAND(I), I = 1, N)
   17:      IFAIL = 0
   18:      CALL Z01ABFP(ICNTXT,'N',IFAIL)
   19:      STOP
   20:      END
```

If you run the above program, you should see something like this on your screen:

```
    Enter MP and NP
    2
```

2

```
      0.9760044636156003       0.3584819212790158      0.7684902391334956
      0.9150123074167526
```

The program is quite self-explanatory, but a few words need to be said about the generated output. Since the `WRITE` statement, on `line 16`, is only executed on the root processor, then the printed values are the values of `RAND` generated on this processor. To output the values of `RAND` on the other processors to the standard output, you need to collect these values onto the root processor. To do this, you need to be familiar with calls to routines in the BLACS and/or the PVM Libraries.

7.7 A Second Look at Run-Time Errors: Invalid Context

In this section we consider one of the commonly occurring errors in the use of the NAG Parallel Library: invalid Library context. As an example, consider Program `EX2E`, which attempts to generate four random numbers on four different processors. The example calls G05ABFP, which selects a Wichmann–Hill random number generator from some available generators and initializes the seeds on a Library Grid of processors. It then calls G05AAFP to generate the distinct sequences of random numbers.

Here is the program:

```
    *  This program may exit with an error condition
    *
 1:      PROGRAM  EX2E
 2:      INTEGER          N
 3:      PARAMETER        (N = 4)
 4:      INTEGER          I,ICNTXT,IGEN,MPAVAL,NPAVAL,MYROW,MYCOL
       +                  MP,NP,ISEED(4)
 5:      DOUBLE PRECISION RAND(N),G05AAFP
 6:      MP = 2
 7:      NP = 2
 8:      IFAIL = 0
 9:      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
10:      CALL BLACS_GRIDINFO(ICNTXT,MPAVAL,NPAVAL,MYROW,MYCOL)
11:      DO 10 I = 1, 4
12:         ISEED(I) = (I*MYROW+(I+1)*MYCOL+ 5)*550008
13: 10   CONTINUE
14:      IGEN = NP*MYROW+MYCOL+1
15:      IFAIL = 0
16:      CALL G05ABFP(ICONTXT,ISEED,IGEN,IFAIL)
17:      DO 20 I = 1, N
18:         RAND(I) = G05AAFP()
19:  20   CONTINUE
20:      CALL Z01ABFP(ICNTXT,'N',IFAIL)
21:      STOP
22:      END
```

Note: This example assumes that your compiler does **not** set any un-initialised variables to zero. Otherwise, the program may produce the correct result.

At `line 10`, a call to the non-Library routine BLACS_GRIDINFO is made. As its name suggests, BLACS_GRIDINFO is a routine from the BLACS Library. It is such a useful routine that we choose to introduce it here. BLACS_GRIDINFO returns information about a grid associated with a particular context. Given the Library context ICNTXT (returned by Z01AAFP), BLACS_GRIDINFO returns the number of processor rows and columns available in the Library Grid (MPAVAL and NPAVAL which should be the same as MP and NP respectively) and the row and column grid coordinate {MYROW, MYCOL} of the calling processor.

The reason we used BLACS_GRIDINFO in Program `EX2E` is as follows. In order to generate a different sequence of random numbers on each processor, we must ensure that ISEED and/or IGEN are initialised to different values on different processors before using G05AAFP. One way of generating different ISEED and IGEN is to use the node coordinates as they are different on each processor. A call to BLACS_GRIDINFO provides this information. `lines 11-14` show how MYROW and MYCOL were used to set the local arguments ISEED and IGEN.

Now, if you run Program `EX2E`, you should see something like this:

```
    ** ABNORMAL EXIT from NAG Parallel Library routine G05ABFP: IFAIL = -2000
    ** The value of ICNTXT was not returned by a call to Z01AAFP.
    ** NAG hard failure - execution terminated
     Error messages from BLACS_ABORT may follow
   {0,0}, pnum=0, Contxt=0, killed other procs, exiting with error #0.
```

The message indicates that the error was detected in G05ABFP and it is due to an invalid value for ICNTXT. If you look at `line 9` of Program `EX2E`, a call to Z01AAFP defines the context ICNTXT. The failure occurred because the call to G05ABFP mistakenly used the un-initialized variable ICONTXT instead of ICNTXT (note the extra letter O).

The value -2000 is always reserved for this failure. A description associated with this IFAIL can be found in Section 5 of the document for G05ABFP (and any other Library routines which have ICNTXT as an argument). It reads:

IFAIL $= -2000$

   The routine has been called with an invalid value of ICNTXT on one or more processors.

Program `EX2E` should successfully generate different sequences of random numbers on different processors, as desired, if `line 16` is replaced by:

```
    CALL G05ABFP(ICNTXT,ISEED,IGEN,IFAIL)
```

8   Example 3: Solving Systems of Linear Equations

Chapter F04 and Chapter F07 of the NAG Parallel Library are concerned with solving systems of linear equations, for which the conventional mathematical notation is

$$Ax = b. \tag{1}$$

Chapter F07 and Chapter F08 contain routines derived from ScaLAPACK, a linear algebra package aimed at distributed memory machines. Because their interfaces are different to the other Library routines, the ScaLAPACK routines are considered in a greater detail in Section 10.

The routines in Chapter F04 are categorised as Black Box routines, in that they solve the linear equations in a single call with the matrix A and the right-hand sides b being supplied as data. These are the simplest routines to use for solving linear systems and are suitable for equations with a number of right-hand-side.

General Purpose routines require a previous call to a routine to factorize the matrix A. This factorization can then be used repeatedly to solve the equations for one or more right-hand sides which may be generated in the course of the computation. The Black Box routines simply call a factorization routine and then a General Purpose routine to solve the equations. There are currently no General Purpose routines in Chapter F04, but such routines are available in Chapter F07, and these routines also allow a little more flexibility with the distribution of the matrices.

The Chapter F04 contains a number of Black Box routines for solving real and complex linear systems and least-squares problems. The following two routines specifically are designed to solve real linear equations:

   F04EBFP: solves a general linear systems with a real matrix A;

   F04FBFP: solves linear systems with a real symmetric positive-definite matrix A.

In this Tutorial we will illustrate the use of F04EBFP, but if you have a real system with a symmetric positive-definite matrix, it is preferable to use F04FBFP. Complex versions of the above routines are also available.

People often wish to solve several systems with the same matrix A, but with different right-hand sides b_i and different solutions x_i:
$$Ax^{(i)} = b^{(i)} \ \text{ for } \ i = 1, \ldots, r,$$
which can be rewritten in matrix notation as

$$AX = B, \tag{2}$$

the columns of B being the right-hand side vectors $b^{(i)}$, and the columns of X being the solution vectors $x^{(i)}$. In the rest of this section we use the notation in (2) as vectors can be considered as a special case of matrices.

8.1 Arrays in the Library

For solving systems of the form (2), it is convenient to store the matrix B in a two-dimensional array B. Most array arguments in the NAG Parallel Library have dimensions which depend on the size of the problem. In Fortran terminology they have 'adjustable dimensions': the dimensions occurring in their declarations are integer variables which are also arguments of the Library routine. However, some Library routines are designed so that they can be called with an argument which may be set to 0 (in which case they would usually exit immediately without doing anything). If so, the declarations in the Library routine use the 'assumed size' array dimension.

If you look up the F04EBFP document, the specification of argument B takes the form:

B(LDB,∗) — DOUBLE PRECISION array *Local Input/Local Output*

Note: the second dimension of the array B must be at least max(1,numroc(NRHS,NB,p_c,0,n_p)).

On entry: the local part of the the the n by r right-hand side matrix B.

On exit: the n by r solution matrix X distributed in the same cyclic 2-d block distribution.

and the argument LDB is described as follows:

LDB — INTEGER *Local Input*

On entry: the first dimension of the array B as declared in the (sub)program from which F04EBFP is called.

Constraint: LDB $\geq$ max(1, numroc(N, NB, p_r, 0, m_p)).

B is an example of an 'assumed size' array and you **must** make sure that you supply the **first** dimension of the array B, as declared in your calling (sub)program, through the argument LDB.

8.2 Constraints

The specification of the argument LDB contains the text:

Constraint: LDB $\geq$ max(1, numroc(N, NB, p_r, 0, m_p)).

The word '*Constraint*' or '*Constraints*' in the specification of an *Input* argument precedes a statement of the range of valid values for that argument. If F04EBFP is called with LDB = 0, the routine will take an error exit, returning a non-zero value of IFAIL.

Constraints on arguments of type CHARACTER only list upper-case alphabetic characters. For example, in the F04EBFP document:

Constraint: TRANS = 'N', 'T' or 'C'.

In practice all routines with CHARACTER arguments will permit the use of lower-case characters.

In case you are wondering, we have not forgotten about 'numroc', 'p_r' and 'm_p' which appear in the above '*Constraint*' text. Section 3.1 of each routine document defines the terms and symbols used in describing the data distribution within the document. If you look up the F04EBFP document, you will find the definitions of these terms in Section 3.1 of the document for F04EBFP. However, the meaning of 'numroc' will become clearer in the later part of this section.

8.2.1   A Prototype Program

Let us go back to our original problem of solving a system of linear equations.

Suppose we wish to solve (2), where

$$A = \begin{pmatrix} 1.80 & 2.88 & 2.05 & -0.89 & 2.20 & 0.45 & 1.40 & 0.50 \\ 5.25 & -2.95 & -0.95 & -3.80 & 6.00 & 2.40 & 0.60 & -1.80 \\ 1.58 & -2.69 & -2.90 & -1.04 & 3.20 & -1.80 & 2.20 & -0.60 \\ -1.11 & -0.66 & -0.59 & 0.80 & 2.10 & 2.20 & 0.40 & -2.40 \\ 4.25 & 3.75 & 2.00 & -1.80 & 2.10 & 0.25 & -1.62 & 0.70 \\ 3.25 & 0.55 & -4.00 & 0.44 & -2.25 & 1.12 & 2.65 & -0.86 \\ -1.34 & 0.34 & -4.00 & 3.20 & 1.10 & -0.44 & 3.20 & 2.00 \\ -0.90 & 1.34 & -3.32 & 4.85 & 0.40 & 0.26 & -2.44 & 0.80 \end{pmatrix} \text{ and } B = \begin{pmatrix} 14.82 \\ 44.15 \\ 9.37 \\ 11.08 \\ 10.44 \\ -0.12 \\ -27.86 \\ -46.05 \end{pmatrix}. \quad (3)$$

Here is a program which attempts to solve (3) using F04EBFP on a single processor:

```
 1:        PROGRAM EX3A
 2:        INTEGER           N,LDA,LDB,NRHS,NB,NIN,NOUT,NIP,NL,NRHSL
 3:        PARAMETER         (N=8,NRHS=1,NB=2,NIN=1,NOUT=6)
 4:        PARAMETER         (LDA=N,LDB=LDA,NL=N,NRHSL=NRHS,NIP=NB+NL)
 5:        INTEGER           ICNTXT,IFAIL,MP,NP,I,J,IPIV(NIP)
 6:        CHARACTER*1       TRANS
 7:        DOUBLE PRECISION A(LDA,NL),B(LDB,NRHSL)
 8:        LOGICAL           Z01ACFP
 9:        MP = 1
10:        NP = 1
11:        IFAIL = 0
12:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
13:        OPEN (UNIT=NIN,FILE='./tutorial_ex3a.dat')
14:        TRANS = 'N'
15:        DO 10 I = 1, N
16:           READ(NIN,*) (A(I,J), J = 1, N)
17: 10     CONTINUE
18:        DO 30 J = 1, NRHS
19:           DO 20 I = 1, N
20:              READ(NIN,*) B(I,J)
21: 20        CONTINUE
22: 30     CONTINUE
23:        IFAIL = 0
24:        CALL F04EBFP(ICNTXT,TRANS,N,NB,A,LDA,NRHS,B,LDB,IPIV,IFAIL)
25:        IF (Z01ACFP()) THEN
26:           WRITE (NOUT,*) 'Solution(s)'
27:           DO 50 J = 1, NRHS
28:              DO 40 I = 1, NL
29:                 WRITE (NOUT,9999) B(I,J)
30: 40           CONTINUE
31: 50        CONTINUE
32:        END IF
33:        CLOSE (NIN)
34:        CALL Z01ABFP(ICNTXT,'N',IFAIL)
35:        STOP
36: 9999 FORMAT(1X,F8.4)
37:        END
```

Here is the data file `tutorial_ex3a.dat`:

```
1.80  2.88  2.05 -0.89  2.20  0.45  1.40  0.50
5.25 -2.95 -0.95 -3.80  6.00  2.40  0.60 -1.80
1.58 -2.69 -2.90 -1.04  3.20 -1.80  2.20 -0.60
```

```
    -1.11 -0.66 -0.59  0.80  2.10  2.20  0.40 -2.40
     4.25  3.75  2.00 -1.80  2.10  0.25 -1.62  0.70
     3.25  0.55 -4.00  0.44 -2.25  1.12  2.65 -0.86
    -1.34  0.34 -4.00  3.20  1.10 -0.44  3.20  2.00
    -0.90  1.34 -3.32  4.85  0.40  0.26 -2.44  0.80    : End of matrix A
     14.82
     44.15
      9.37
     11.08
     10.44
     -0.12
    -27.86
    -46.05      : End of rhs B
```

Note: The listing of the Program EX3A at `line 13` does not give a full pathname for the data file `tutorial_ex3a.dat`. You must make sure that the full pathname is given in this and any similar statement.

Here are the (correct) results:

```
    Solution(s)
      1.0000
     -1.0000
      3.0000
     -5.0000
      1.0000
      2.0000
      3.0000
     -4.0000
```

`Lines 9-11`, in Program EX3A, set up a 1 by 1 Library Grid and `lines 14-23` read and initialize input arguments which are needed by F04EBFP. A call to F04EBFP is made on `line 24`, and `lines 25-32` output the results. Finally, a call to Z01ABFP, on `line 34`, un-defines the Library Grid-logical and shuts down the message-passing environment.

8.2.2 Data Distribution

To solve (3) on four processors (a 2 by 2 Library Grid, say), you may be tempted to replace `lines 9-10` by:

```
    MP = 2
    NP = 2
```

The question is whether this change is sufficient to produce the correct result. The answer is simple: **no, it is not!** If you run this program, you will encounter an unpredictable, system dependent run-time error and the major factor behind this behaviour is related to **the way in which data (here our matrices) should be distributed before a call to F04EBFP**.

The underlying algorithm used in F04EBFP will perform correctly provided that, before a call to this routine, the matrices A and B are distributed in a special way over the Library Grid. Before describing the data distribution required by F04EBFP, it is useful to have a closer look at the specification of the array A in the F04EBFP document. It reads:

A(LDA,∗) — DOUBLE PRECISION array *Local Input/Local Output*

Note: the second dimension of the array A must be at least max(1,numroc(N,NB,p_c,0,n_p)).

On entry: the local part of the matrix A.

On exit: A is overwritten by the factors L and U distributed in the same cyclic 2-d block fashion; the unit diagonal elements of L are not stored.

As its specification states, array A is classified as *Local Input/Local Output*. Its description also specifies that on entry to the routine it should be set to 'the local part of the matrix A'. What does 'local part ... ' mean? Since the linear solver is designed to run on a parallel machine (with many processors), it is useful to store different portions of the matrix A on different processors rather than have the whole matrix stored on each processor. This will allow problems to be solved which may be beyond the memory capability of a single machine (particularly on networks of workstations or PCs).

Now, if you take a closer look at Program `EX3A`, the array A is treated as a global argument! On a single processor (1 by 1 Library Grid) this will not cause any problem because the communication and distribution of data are not an issue in this example. However, this will clearly cause a problem if you increase the number of processors in the program.

As a rule, the data distribution strategy used in a parallel algorithm should try to minimize data movement between processors and reduce the communication time. In particular, the way in which a matrix is distributed over the Library Grid has a major impact on the load balance and communication characteristic of the parallel algorithm. The F04 routines use what is termed as a **block-partitioned algorithm**. This means that the matrices are partitioned into rectangular blocks and at each major step of the algorithm a **block** of rows or columns is updated, and most of the computation is performed by matrix–matrix operations on these blocks. The block distribution strategy used in F04EBFP is called the **cyclic 2-d block distribution** and is parameterized by the number of rows and columns in a Library Grid and the **blocking factor** — that is, the number of rows and columns per block. The F04 routines use a **square** block and provide the argument NB which allows you to specify the block size used in the algorithm. In Program `EX3A` the block size NB was set to 2.

8.3 Cyclic 2-d Block Distribution

This distribution is also used by the ScaLAPACK routines in Chapter F07 and Chapter F08, and is such that row blocks of the matrix are distributed in a cyclic fashion to the associated rows of logical processors, and column blocks are distributed in cyclic fashion to the associated columns of logical processors. Here the terms row blocks and column blocks refer to one or more contiguous rows or columns of a matrix which are treated as a single entity from the algorithmic point of view.

	1	2	3	4	5	6	7	8	9	10	11	12
1	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
2	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
3	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
4	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
5	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
6	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
7	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
8	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
9	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
10	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}
11	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}	{0,0}	{0,1}	{0,2}
12	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}	{1,0}	{1,1}	{1,2}

Figure 3
Cyclic 2-d block distribution over a 2 by 3 logical grid of processors.

Figure 3 and Figure 4 illustrate the cyclic 2-d block distribution of a matrix consisting of 12 row and column blocks on a 2 by 3 grid. The block rows (and the block columns) of the matrix are numbered from 1 to 12, and the processors are represented by their coordinate indices.

Using Figure 3 and Figure 4 as our basis, and using a block of size 2, Figure 5 illustrates the way in which an 8 by 8 matrix A with elements $a_{i,j}$ must be distributed before entering F04EBFP. Figure 6 illustrates the explicit distribution of the matrices A and B in (3) as specified in the data file `tutorial_ex3a.dat`. Note the distribution of array B. The elements of B which were not set in Processors P_1 and P_3 are never accessed by F04EBFP.
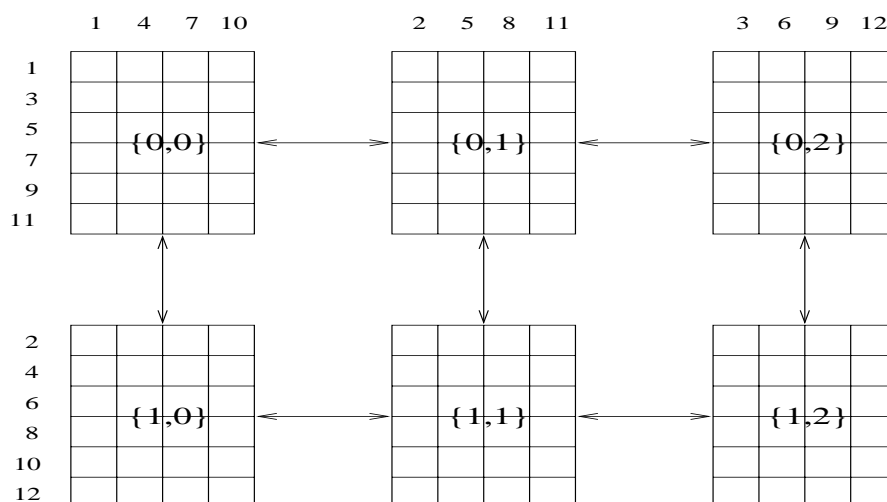
Figure 4
Cyclic 2-d block distribution from the processors' point of view.

We are now in a position to say what exactly 'numroc' means. 'numroc' is a function which gives the '**num**ber of **r**ows **o**r **c**olumns' of a distributed matrix owned by a specific processor. Obviously, the value of 'numroc' may differ from processor to processor (i.e., it is local) in the Library Grid as it depends on local information such as the row or column coordinate of the processor within the Library Grid. However, the Library provides the function Z01CAFP (NUMROC) for the evaluation of this value. For example, if you look up Figure 6, the value of 'numroc' (both row or column) of the matrix A on each processor is four. For the matrix B, the values of 'numroc' (both row and column) on processors $\{0,1\}$ and $\{1,1\}$ are zero. However, the number of rows on processors $\{0,0\}$ and $\{1,0\}$ is four and the number of columns on these processors is equal to one.

For F04EBFP (as with many other Library routines) the distribution of the matrix on to the Library Grid is the responsibility of the user, as is the collection of results if this is required. However, the NAG Parallel Library has minimized this burden by providing routines which aid you in the distribution and collection of data.

8.4   Reading and Printing General Matrices

In order to develop a program which solves (3) in parallel, we need to distribute matrix A in cyclic 2-d block form on our Library Grid. This task can easily be achieved by the routine X04BGFP, which reads in a matrix in its natural non-distributed form (as in the file `tutorial_ex3a.dat`) and distributes it into an array in a cyclic 2-d block form on a Library Grid.

Replacing `lines 9-10` in program `EX3A` by

```
        MP = 2
        NP = 2
```

and `lines 15-22` by

```
        IFAIL = 0
        CALL X04BGFP(ICNTXT,NIN,N,N,NB,A,LDA,IFAIL)
        IFAIL = 0
        CALL X04BGFP(ICNTXT,NIN,N,NRHS,NB,B,LDB,IFAIL)
```

and modifying the dimensions of the local arrays `A`, `B` and `IPIV` to use less storage on each processor (by changing the parameters in the second `PARAMETER` statement), here is a program to solve (3) in parallel on a 2 by 2 grid:

```
     *  This program does not produce the correct solution
     *
```

Processor{0,0}   **Processor{0,1}**

$$
\begin{array}{cccc}
a_{11} & a_{12} & a_{15} & a_{16} \\
a_{21} & a_{22} & a_{25} & a_{26} \\
a_{51} & a_{52} & a_{55} & a_{56} \\
a_{61} & a_{62} & a_{65} & a_{66}
\end{array}
\qquad
\begin{array}{cccc}
a_{13} & a_{14} & a_{17} & a_{18} \\
a_{23} & a_{24} & a_{27} & a_{28} \\
a_{53} & a_{54} & a_{57} & a_{58} \\
a_{63} & a_{64} & a_{67} & a_{68}
\end{array}
$$

$$
\begin{array}{cccc}
a_{31} & a_{32} & a_{35} & a_{36} \\
a_{41} & a_{42} & a_{45} & a_{46} \\
a_{71} & a_{72} & a_{75} & a_{76} \\
a_{81} & a_{82} & a_{85} & a_{86}
\end{array}
\qquad
\begin{array}{cccc}
a_{33} & a_{34} & a_{37} & a_{38} \\
a_{43} & a_{44} & a_{47} & a_{48} \\
a_{73} & a_{74} & a_{77} & a_{78} \\
a_{83} & a_{84} & a_{87} & a_{88}
\end{array}
$$

Processor{1,0}   **Processor{1,1}**
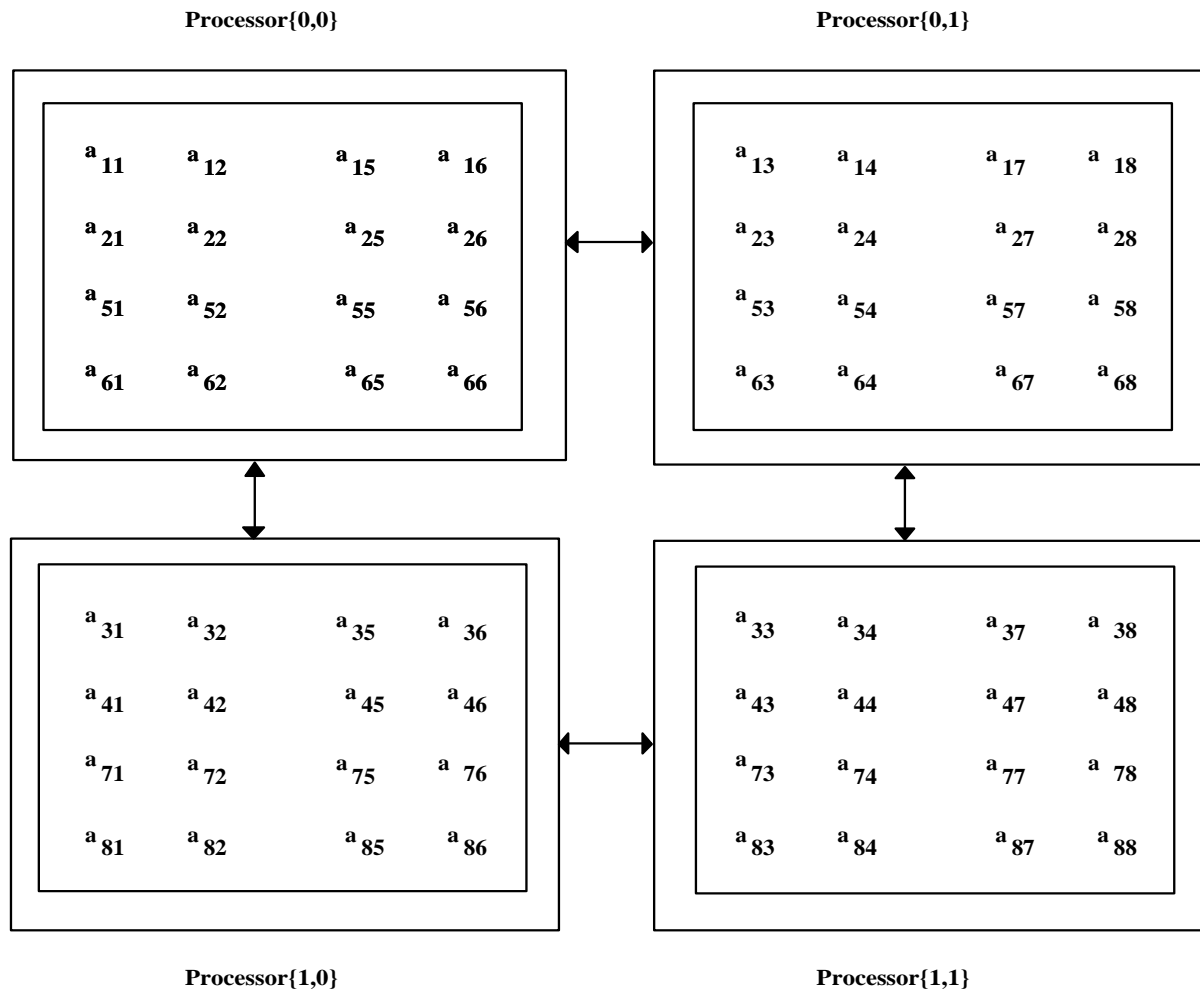
Figure 5
Cyclic 2-d block distribution of an 8 by 8 matrix A using a block of size 2.

```
 1:        PROGRAM EX3B
 2:        INTEGER           N,LDA,LDB,NRHS,NB,NIN,NOUT,NIP,NL,NRHSL
 3:        PARAMETER         (N=8,NRHS=1,NB=2,NIN=1,NOUT=6)
 4:        PARAMETER         (LDA=4,LDB=LDA,NL=4,NRHSL=NRHS,NIP=NB+NL)
 5:        INTEGER           ICNTXT,IFAIL,MP,NP,I,J,IPIV(NIP)
 6:        CHARACTER*1       TRANS
 7:        DOUBLE PRECISION A(LDA,NL),B(LDB,NRHSL)
 8:        LOGICAL           Z01ACFP
 9:        MP = 2
10:        NP = 2
11:        IFAIL = 0
12:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
13:        OPEN (UNIT=NIN,FILE='./tutorial_ex3a.dat')
14:        TRANS = 'N'
15:        IFAIL = 0
16:        CALL X04BGFP(ICNTXT,NIN,N,N,NB,A,LDA,IFAIL)
17:        IFAIL = 0
18:        CALL X04BGFP(ICNTXT,NIN,N,NRHS,NB,B,LDB,IFAIL)
19:        IFAIL = 0
20:        CALL F04EBFP(ICNTXT,TRANS,N,NB,A,LDA,NRHS,B,LDB,IPIV,IFAIL)
21:        IF (Z01ACFP()) THEN
22:            WRITE (NOUT,*) 'Solution(s)'
```

Processor {0, 0}   **Processor** {0, 1}

A				B
1.80	2.88	2.20	0.45	14.82
5.25	-2.95	6.00	2.40	44.15
4.25	3.75	2.10	0.25	10.44
3.25	0.55	-2.25	1.12	-0.12

A				B
2.05	-0.89	1.40	0.50	
-0.95	-3.80	0.60	-1.80	
2.00	-1.80	-1.62	0.70	
-4.00	0.44	2.65	-0.86	

Processor {1, 0}   **Processor** {1, 1}

A				B
1.58	-2.69	3.20	-1.80	9.37
-1.11	-0.66	2.10	2.20	11.08
-1.34	0.34	1.10	-0.44	-27.86
-0.90	1.34	0.40	0.26	-46.05

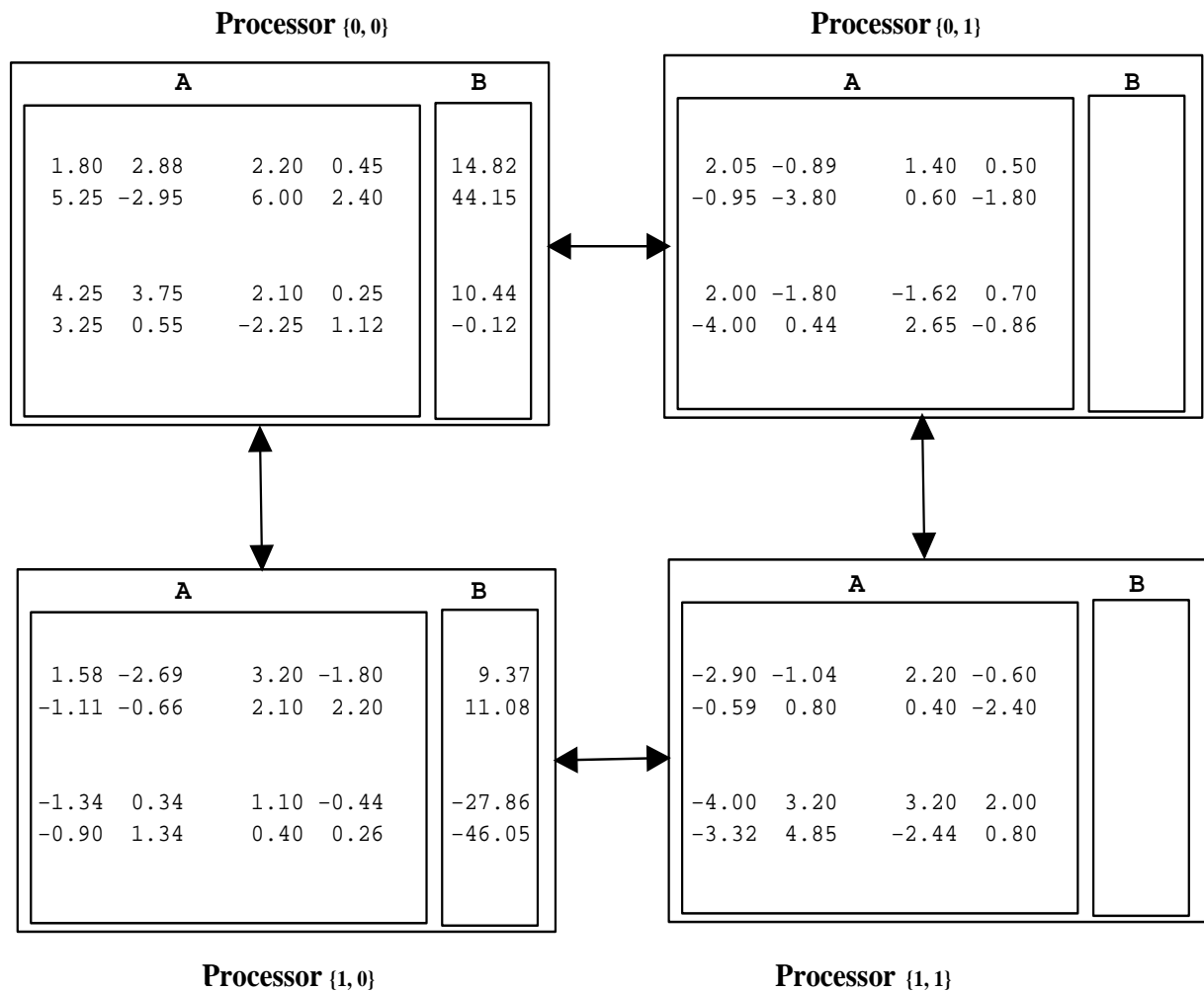| A				B
-2.90	-1.04	2.20	-0.60	
-0.59	0.80	0.40	-2.40	
-4.00	3.20	3.20	2.00	
-3.32	4.85	-2.44	0.80	

Figure 6

Cyclic 2-d block distribution of matrices A and B in (3) using a block of size 2.

```
23:          DO 50 J = 1, NRHS
24:             DO 40 I = 1, LDB
25:                WRITE (NOUT,9999) B(I,J)
26: 40          CONTINUE
27: 50       CONTINUE
28:       END IF
29:       CLOSE (NIN)
30:       CALL Z01ABFP(ICNTXT,'N',IFAIL)
31:       STOP
32: 9999 FORMAT(1X,F8.4)
33:       END
```

Using the data file `tutorial_ex3a.dat`, program `EX3B` gives the results:

```
Solution(s)
  1.0000
 -1.0000
  1.0000
  2.0000
```

If you compare this result with the correct result provided in the previous section, you see that the first two elements of the solution are correct, but the third and the fourth elements are the fifth and sixth

elements of the correct solution vector. The reason behind this behaviour is quite simple; a look at the specification of the argument B will help you to understand this better. The description of the argument B specifies that B is not only classified as *Local Input*, but also as *Local Output*. Its description says *On exit*: 'solution matrix X distributed in the same cyclic 2-d block distribution' which clearly emphasizes that the solution not only overwrites the array B, but is also stored in cyclic 2-d block distribution. After a call to F04EBFP, `lines 21-28` of Program EX3B attempt to output the solution on the root processor which clearly does not possess the solution in its non-distributed form.

To output the correct solution computed by F04EBFP, you need to collect the result onto the root processor in its natural order. The Library provides the routine X04BHFP which outputs a general real matrix A stored in cyclic 2-d block distribution over the Library Grid in its natural and non-distributed form. X04BHFP can be viewed as having the reverse functionality to X04BGFP.

To produce our final and correct program, you simply need to replace `lines 21-28` by

```
IF (Z01ACFP()) WRITE (NOUT,*) 'Solution(s)'
FORMAT = '(F9.4)'
IFAIL = 0
CALL X04BHFP(ICNTXT,NOUT,N,NRHS,NB,B,LDB,FORMAT,WORK,IFAIL)
```

Note that X04BHFP requires a double precision workspace argument WORK of size N and a global input argument FORMAT of type CHARACTER (note this makes FORMAT 9999 redundant and so it can be deleted). Incorporating these arguments in the declarative section of Program EX3B, we have:

```
 1:      PROGRAM EX3C
 2:      INTEGER           N,LDA,LDB,NRHS,NB,NIN,NOUT,NIP,NL,NRHSL
 3:      PARAMETER         (N=8,NRHS=1,NB=2,NIN=1,NOUT=6)
 4:      PARAMETER         (LDA=4,LDB=LDA,NL=4,NRHSL=NRHS,NIP=NB+NL)
 5:      INTEGER           ICNTXT,IFAIL,MP,NP,I,J,IPIV(NIP)
 6:      CHARACTER*1       TRANS
 7:      CHARACTER*80      FORMAT
 8:      DOUBLE PRECISION A(LDA,NL),B(LDB,NRHSL),WORK(N)
 9:      LOGICAL           Z01ACFP
10:      MP = 2
11:      NP = 2
12:      IFAIL = 0
13:      CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
14:      OPEN (UNIT=NIN,FILE='./tutorial_ex3a.dat')
15:      TRANS = 'N'
16:      IFAIL = 0
17:      CALL X04BGFP(ICNTXT,NIN,N,N,NB,A,LDA,IFAIL)
18:      IFAIL = 0
19:      CALL X04BGFP(ICNTXT,NIN,N,NRHS,NB,B,LDB,IFAIL)
20:      IFAIL = 0
21:      CALL F04EBFP(ICNTXT,TRANS,N,NB,A,LDA,NRHS,B,LDB,IPIV,IFAIL)
22:      IF (Z01ACFP()) WRITE (NOUT,*) 'Solution(s)'
23:      FORMAT = '(F9.4)'
24:      IFAIL = 0
25:      CALL X04BHFP(ICNTXT,NOUT,N,NRHS,NB,B,LDB,FORMAT,WORK,IFAIL)
26:      CLOSE (NIN)
27:      CALL Z01ABFP(ICNTXT,'N',IFAIL)
28:      STOP
29:      END
```

If you run this with the data file `tutorial_ex3a.dat`, you should get the correct results:

```
Solution(s)
  1.0000
 -1.0000
```

```
   3.0000
  -5.0000
   1.0000
   2.0000
   3.0000
  -4.0000
```

8.5   Another Look at Run-Time Errors: Inconsistent Global Argument

All global input arguments in Library routines must have the same values on all processors in the Library Grid. If you provide different values to an argument which is classified as *Global Input* or *Global Input/Global Output*, the Library detects this error.

As our example, we consider the argument NB in Program EX3C in the following program:

```
      *   This program will exit with an error condition
      *
 1:        PROGRAM EX3D
 2:        INTEGER           N,LDA,LDB,NRHS,NB,NIN,NOUT,NIP,NL,NRHSL
 3:        PARAMETER         (N=8,NRHS=1,NIN=1,NOUT=6)
 4:        PARAMETER         (LDA=4,LDB=LDA,NL=4,NRHSL=NRHS,NIP=2+NL)
 5:        INTEGER           ICNTXT,IFAIL,MP,NP,I,J,IPIV(NIP)
 6:        CHARACTER*1       TRANS
 7:        CHARACTER*80      FORMAT
 8:        DOUBLE PRECISION  A(LDA,NL),B(LDB,NRHSL),WORK(N)
 9:        LOGICAL           Z01ACFP
10:        MP = 2
11:        NP = 2
12:        IFAIL = 0
13:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
14:        NB = 2
15:        OPEN (UNIT=NIN,FILE='./tutorial_ex3a.dat')
16:        TRANS = 'N'
17:        IFAIL = 0
18:        IF (Z01ACFP()) NB = 1
19:        CALL X04BGFP(ICNTXT,NIN,N,N,NB,A,LDA,IFAIL)
           .
           .

28:        END
```

If you look at Section 3.2 of the document for X04BGFP of X04GBFP document, it starts with: 'The arguments M, N, NB and IFAIL are all global, and ...'. Obviously, this indicates that NB is a global argument. And, if you look at its specification:

NB — INTEGER   *Global Input*

it is confirmed that NB is classified as *Global Input*. Inserting the line

```
        IF (Z01ACFP()) NB = 1
```

before `line 17` of Program EX3C results in NB having the value 1 on the root processor and 2 on the other three processors (we also defined NB as a variable in Program EX3D rather than a constant). If you run this modified program, you should get something like this:

```
    ** ABNORMAL EXIT from NAG Parallel Library routine X04BGFP: IFAIL =    -5
     Global argument NB did not have the same value on all processors
    ** NAG hard failure - execution terminated
     Error messages from BLACS_ABORT may follow
    {0,0}, pnum=0, Contxt=0, killed other procs, exiting with error #0.
```

The error message indicates that it occurred in X04BGFP because 'Global argument NB did not have the same value on all processors'. Furthermore, NB occupies the fifth position in the argument list of X04BGFP, and it is no coincidence that IFAIL was set to -5 on exit. The following description in the 'Errors and Warning' section of the X04BGFP document makes it clear why IFAIL was set to this negative value:

IFAIL $= -i$

On entry, the ith argument had an invalid value. This error occurred either because a global argument did not the same value on all logical processors, or because its value on one or more processors was incorrect. An explanatory message distinguishes between these two cases.

In fact, negative error conditions occur in all routines which have global input arguments.

9 Example 4: Evaluation of Integrals

9.1 Routines with Subroutine Arguments

The problem considered in this section is to compute an approximation to the integral

$$\int_0^1 \frac{dx}{\sqrt{|x^2 + 2x - 2|}}. \tag{4}$$

The routine D01ATFP in Chapter D01 handles numerical integration of a function of one variable, over a finite interval, and is particularly designed to handle general integrands.

This routine requires a user-supplied subroutine to evaluate the integrand at an array of different points. In fact all the one-dimensional routines in Chapter D01 require a user-supplied subroutine for the evaluation of the integrand.

It is documented as follows:

F — SUBROUTINE, supplied by the user. *Global External Procedure*

F must return the values of the integrand f at a set of points.

Its specification is:

```
SUBROUTINE       F(X, FV, N)
INTEGER          N
DOUBLE PRECISION X(N), FV(N)
```

X(N) — DOUBLE PRECISION array *Local Input*
On entry: the points at which the integrand f must be evaluated.

FV(N) — DOUBLE PRECISION array *Local Output*
On exit: FV(j) must contain the value of f at the point X(j), for $j = 1, 2, ..., N$.

N — INTEGER *Global Input*
On entry: the number of points at which the integrand is to be evaluated. The actual value of N is always 21 which is the number of points in the Kronrod rule being used.

F must be declared as EXTERNAL in the (sub)program from which is called. Arguments denoted as *Input* must **not** be changed by this procedure.

It is also important to emphasize that all routine arguments are classified as global (i.e., the same procedure must be supplied to each processor), whereas **their** argument(s) may be either local or global.

Here is a program to evaluate the integral (4) on a 2 by 2 Library Grid:

```
 1:        PROGRAM EX4A
 2:        INTEGER          MAXSUB,LIW,LW
 3:        PARAMETER        (MAXSUB=100,LIW=400,LW=2000)
 4:        INTEGER          ICNTXT,IFAIL,NFUN,MP,NP,IWORK(LIW)
 5:        DOUBLE PRECISION A,ABSERR,B,EPSABS,EPSREL,RESULT,WORK(LW)
 6:        EXTERNAL F
 7:        LOGICAL Z01ACFP
 8:        MP = 2
 9:        NP = 2
10:        IFAIL = 0
11:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
12:        A = 0.0D0
13:        B = 1.0D0
14:        EPSREL = 0.0D0
15:        EPSABS = 1.0D-4
16:        IFAIL = 0
17:        CALL D01ATFP(ICNTXT,F,A,B,EPSABS,EPSREL,RESULT,ABSERR,NFUN,
18:       +             WORK,LW,IWORK,LIW,IFAIL)
19:        IF (Z01ACFP()) THEN
20:           WRITE (*,9998) 'RESULT =',RESULT, ' ABSERR =',ABSERR
21:           WRITE (*,9999) 'No. of function evaluations =',NFUN
22:        END IF
23:        CALL Z01ABFP(ICNTXT,'N',IFAIL)
24:        STOP
25: 9999 FORMAT (1X,A,I6)
26: 9998 FORMAT (1X,A,F12.8,A,E12.2)
27:        END

28:        SUBROUTINE F(X,FV,N)
29:        INTEGER N,I
30:        DOUBLE PRECISION FV(N),X(N)
31:        DO 20 I = 1,N
32:           FV(I) = 1.D0/SQRT(ABS(X(I)**2+2.D0*X(I)-2.D0))
33: 20    CONTINUE
34:        RETURN
35:        END
```

This gives the results:

```
RESULT =  1.50460652  ABSERR =    0.35E-04
No. of function evaluations =  1113
```

9.2 Handling Failure Exits from the Library

Now suppose that we wish to investigate the effect of specifying different values for the requested accuracy in Program EX4A. Since the value of the integral is close to 1, absolute and relative accuracy have roughly the same meaning. We will set the relative accuracy EPSREL to 0, which means that only absolute accuracy is operative.

The following program requests a progressively more accurate result, setting EPSABS to 10^{-3}, 10^{-4}, ... :

```
     *  This program will eventually exit with an error condition
     *
 1:        PROGRAM EX4B
 2:        INTEGER          MAXSUB,LIW,LW
 3:        PARAMETER        (MAXSUB=100,LIW=400,LW=2000)
 4:        INTEGER          ICNTXT,IFAIL,NFUN,MP,NP,IWORK(LIW)
 5:        DOUBLE PRECISION A,ABSERR,B,EPSABS,EPSREL,RESULT,WORK(LW)
```

```
 6:       EXTERNAL F
 7:       LOGICAL Z01ACFP
 8:       MP = 2
 9:       NP = 2
10:       IFAIL = 0
11:       CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
12:       A = 0.0D0
13:       B = 1.0D0
14:       EPSREL = 0.0D0
15:       IF (Z01ACFP()) THEN
16:          WRITE (*,*) ' Requested    Result    Estimated    Number of'
17:          WRITE (*,*) '  accuracy               accuracy   evaluations'
18:          WRITE (*,*) '  ...........................................'
19:       END IF
20:       DO 10 I = 3, 10
21:          EPSABS = 10.0**(-I)
22:          IFAIL = 0
23:          CALL D01ATFP(ICNTXT,F,A,B,EPSABS,EPSREL,RESULT,
24:      +                ABSERR,NFUN,WORK,LW,IWORK,LIW,IFAIL)
25:          IF (Z01ACFP()) WRITE (*,9999) EPSABS, RESULT, ABSERR,NFUN
26: 10    CONTINUE
27:       CALL Z01ABFP(ICNTXT,'No',IFAIL)
28:       STOP
29: 9999 FORMAT (1X,E12.2,F12.8,E12.2,I10)
30:       END

31:       SUBROUTINE F(X,FV,N)
32:       INTEGER N,I
33:       DOUBLE PRECISION FV(N),X(N)
34:       DO 20 I = 1,N
35:          FV(I) = 1.D0/SQRT(ABS(X(I)**2+2.D0*X(I)-2.D0))
36: 20    CONTINUE
37:       RETURN
38:       END
```

The results are:

```
    Requested    Result    Estimated    Number of
    accuracy                accuracy   evaluations

    ...........................................
    0.10E-02  1.50446645    0.98E-04       903
    0.10E-03  1.50460652    0.35E-04      1113
    0.10E-04  1.50460652    0.11E-05      1197
    0.10E-05  1.50462235    0.77E-07      1785
  ** ABNORMAL EXIT from NAG Library routine D01ATFP: IFAIL =      3
  ** Extremely bad integrand behaviour occurs around one or more subintervals.
  ** NAG hard failure - execution terminated
   Error messages from BLACS_ABORT may follow
  {0,0}, pnum=0, Contxt=0, killed other procs, exiting with error #0.
```

When the requested accuracy is 10^{-7}, the routine takes an error exit, returning a value IFAIL = 3, which is a computational **failure**: the routine cannot satisfy the requested accuracy. More extensive advice is provided in Section 5 of the routine document.

However, you may want to examine the value of the estimated error ABSERR, and depending on its value you may decide to accept the value of the integral returned in RESULT. But we cannot examine ABSERR unless we enable the program to continue execution on return from D01ATFP after the computational failure.

Before showing you how this can be done, we need to have another look at the argument IFAIL.

9.3   Another Look at IFAIL

9.3.1   Hard Fail Option

So far, we have been using the hard fail option in all our example programs. If the value of IFAIL is set to 0 on all processors (associated with the Library Grid) before calling the Library routine, execution of the program will terminate if the routine detects an error, and the IFAIL argument will have the same value on all processors. This option **aborts** not only the Library Grid, but also other grids which have been created in your program using BLACS. Before the program is stopped, this error message is output to the root processor:

```
** ABNORMAL EXIT from NAG Parallel Library routine XXXXXXX: IFAIL =     n
** NAG hard failure - execution terminated
```

where XXXXXXX is the routine name, and **n** is the number associated with the detected error. An explanation of error number **n** is given in Section 5 of the routine document XXXXXXX. (Error messages may be sent to an external file by calling the support routine X04AAF to change the default unit number for error messages.)

In addition, all the routines output explanatory error messages immediately before the standard termination message shown above.

The hard fail option should be selected if you are in any doubt about continuing the execution of the program after an unsuccessful call to a NAG Parallel Library routine.

In the case of hard failure, all processes created by BLACS will stop, including all those which are not in the Library context. Users familiar with the BLACS might like to note that the routine BLACS_ABORT is used for this purpose. That is why the error message

```
 Error messages from BLACS_ABORT may follow
{0,0}, pnum=0, Contxt=0, killed other procs, exiting with error #0.
```

is output.

For this reason, this option is not recommended for multigridding, a technique which enables the users to have many logical grids, including only one Library Grid, simultaneously.

9.3.2   Soft Fail Option

To select this option, you must set IFAIL to 1 or -1 on all processors (associated with the Library Grid) before calling the Library routine.

If the routine detects an error, IFAIL is reset to the associated error number on all processors; further computation within the routine is suspended and control returns to the calling program.

If you set IFAIL to 1, then no error message is output on the root processor (**silent exit**).

If you set IFAIL to -1 (**noisy exit**), then before control is returned to the calling program, the following error message is output on the root processor:

```
** ABNORMAL EXIT from NAG Parallel Library routine XXXXXXX: IFAIL =     n
** NAG soft failure - control returned
```

In addition, all routines output explanatory error messages immediately before the above standard message.

It is essential to test the value of IFAIL on exit if the soft fail option is selected.

A non-zero exit value of IFAIL implies that the call was not successful, so it is imperative that your program be coded to take appropriate action. That action may simply be to print IFAIL with an explanatory caption and then terminate the program. Some of the example programs in Section 8 of the routine documents have tests of this form. In the more ambitious case, where you wish your program to continue, it is essential that the program can branch to a point at which it is **sensible** to resume computation.

The soft fail option puts the onus on you to handle any errors detected by the Library routine. With the proviso that you are able to implement it **properly**, it is clearly more flexible than the hard fail option since it allows computation to continue in the case of errors. In particular there are at least three cases where its flexibility is useful:

(i) where additional information about the error or the progress of computation is returned via some of the other arguments;

(ii) in some routines, 'partial' success can be achieved, e.g. a probable solution found but not all conditions fully satisfied, so the routine returns a warning. On the basis of the advice in Section 5 and elsewhere in the routine document, you may decide that this partially successful call is adequate for certain purposes;

(iii) on a parallel machine, actions taken on discovering an error may be complicated by the fact that you may have other processes executing on grids other than the Library Grid. For example it may be unreasonable to simply stop the Library Grid when other grids in your program may be expecting results from it.

Let us return to our one-dimensional quadrature problem. Here is a program which prints information about the integral if on exit $IFAIL = 0$ or $ABSERR \le 10^{-7}$:

```
 1:       PROGRAM EX4C
 2:       INTEGER           MAXSUB,LIW,LW
 3:       PARAMETER         (MAXSUB=100,LIW=400,LW=2000)
 4:       INTEGER           ICNTXT,IFAIL,NFUN,MP,NP,IWORK(LIW)
 5:       DOUBLE PRECISION A,ABSERR,B,EPSABS,EPSREL,RESULT,WORK(LW)
 6:       EXTERNAL F
 7:       LOGICAL Z01ACFP
 8:       MP = 2
 9:       NP = 2
10:       IFAIL = 0
11:       CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
12:       A = 0.0D0
13:       B = 1.0D0
14:       EPSREL = 0.0D0
15:       IF (Z01ACFP()) THEN
16:          WRITE (*,*) ' Requested    Result    Estimated    Number of'
17:          WRITE (*,*) '  accuracy                accuracy    evaluations'
18:          WRITE (*,*) '  ...........................................'
19:       END IF
20:       DO 10 I = 3, 10
21:          EPSABS = 10.0**(-I)
22:          IFAIL = 1
23:          CALL D01ATFP(ICNTXT,F,A,B,EPSABS,EPSREL,RESULT,
24:      +               ABSERR,NFUN,WORK,LW,IWORK,LIW,IFAIL)
25:          IF (IFAIL.EQ.0 .OR. ABSERR.LE.1.0D-7) THEN
26:             IF (Z01ACFP()) WRITE (*,9999) EPSABS, RESULT, ABSERR,NFUN
27:          END IF
28: 10    CONTINUE
29:       CALL Z01ABFP(ICNTXT,'No',IFAIL)
30:       STOP
31: 9999 FORMAT (1X,E12.2,F12.8,E12.2,I10)
32:       END

33:       SUBROUTINE F(X,FV,N)
34:       INTEGER N,I
35:       DOUBLE PRECISION FV(N),X(N)
36:       DO 20 I = 1,N
37:          FV(I) = 1.D0/SQRT(ABS(X(I)**2+2*X(I)-2))
```

```
38: 20   CONTINUE
39:      RETURN
40:      END
```

And here are the results:

```
     Requested    Result    Estimated   Number of
     accuracy               accuracy    evaluations

     .................................................
       0.10E-02  1.50446645   0.98E-04       903
       0.10E-03  1.50460652   0.35E-04      1113
       0.10E-04  1.50460652   0.11E-05      1197
       0.10E-05  1.50462235   0.77E-07      1785
       0.10E-06  1.50462235   0.77E-07      2079
       0.10E-07  1.50462235   0.77E-07      2163
       0.10E-08  1.50462235   0.77E-07      2415
       0.10E-09  1.50462235   0.77E-07      2835
```

As IFAIL was set to 1 on `line 22` no error message is output when the estimated accuracy does not satisfy the required absolute accuracy. However, for completeness we present the results when IFAIL is set to -1 in Program `EX4C` on `line 22`:

```
     Requested    Result    Estimated   Number of
     accuracy               accuracy    evaluations

     .................................................
       0.10E-02  1.50446645   0.98E-04       903
       0.10E-03  1.50460652   0.35E-04      1113
       0.10E-04  1.50460652   0.11E-05      1197
       0.10E-05  1.50462235   0.77E-07      1785
 ** ABNORMAL EXIT from NAG Parallel Library routine D01ATFP: IFAIL =    3
 ** Extremely bad integrand behaviour occurs around one or more subintervals.
 ** NAG soft failure - control returned
       0.10E-06  1.50462235   0.77E-07      2079
 ** ABNORMAL EXIT from NAG Parallel Library routine D01ATFP: IFAIL =    3
 ** Extremely bad integrand behaviour occurs around one or more subintervals.
 ** NAG soft failure - control returned
       0.10E-07  1.50462235   0.77E-07      2163
 ** ABNORMAL EXIT from NAG Parallel Library routine D01ATFP: IFAIL =    3
 ** Extremely bad integrand behaviour occurs around one or more subintervals.
 ** NAG soft failure - control returned
       0.10E-08  1.50462235   0.77E-07      2415
 ** ABNORMAL EXIT from NAG Parallel Library routine D01ATFP: IFAIL =    3
 ** Extremely bad integrand behaviour occurs around one or more subintervals.
 ** NAG soft failure - control returned
       0.10E-09  1.50462235   0.77E-07      2835
```

10   Example 5: Using ScaLAPACK Routines

Chapter F07 and Chapter F08 of the NAG Parallel Library contain routines derived from ScaLAPACK, a package designed for the solution of dense or banded linear algebra problems on parallel distributed memory machines. These chapters provide routines for:

Real and complex matrix factorization: LU, Cholesky, QR

Solution of real and complex linear equations: general, symmetric positive-definite

Real and complex linear least-squares and associated problems

Estimating (real) triangular matrix condition numbers

Real symmetric eigenproblems

The routines in Chapter F07 and Chapter F08 utilize the same Library mechanism as the rest of the Library, but their interfaces are different. As with other Library routines, Z01AAFP and Z01ABFP may be used in conjunction with these routines to set up and un-define the Library Grid respectively.

10.1   F07 and F08 Routines and Other Library Routines

Before we proceed, here is a typical call to a ScaLAPACK routine:

```
         .
      CALL Z01AAFP( ICNTXT, MP, NP,IFAIL )
         .
      CALL F07ADFP( M, N, A, IA, JA, IDESCA, IPIV, INFO )
         .
      CALL Z01ABFP( ICNTXT, CONT, IFAIL )
         .
```

F07ADFP performs *LU* factorization of a real matrix. In addition to their NAG names (beginning with F07 or F08), ScaLAPACK routines can be referenced by their original names. So the call to F07ADFP in the above code fragment may be replaced by

```
      CALL PDGETRF( M, N, A, IA, JA, IDESCA, IPIV, INFO )
```

The equivalent ScaLAPACK name for an F07 or F08 routine can be found in the relevant routine document or the Chapter Introduction.

Essentially, there are **two** major differences between ScaLAPACK routines and the rest of the Library routines.

(i) **Context:** unlike other Library routines, the argument lists in the F07 and F08 routines **do not** start with the argument ICNTXT. (As we will see later, the value of ICNTXT must be contained in an element of the integer array IDESCA.)

(ii) **Error Mechanism:** the routines in Chapter F07 and Chapter F08 do not use the usual Library error-handling mechanism, involving the argument IFAIL. Instead they have a diagnostic parameter INFO, which allows us to preserve complete compatibility with the ScaLAPACK routines.

INFO is classified as *Global Output* and need not be set on entry. If the routine detects an error, INFO is reset to the associated error number on all processors; further computation within the routine is suspended and control returns to the calling program (this corresponds to a **soft failure** in terms of the error-handling terminology used in the rest of the Library). INFO indicates the success or failure of the computation, as follows:

INFO $= 0$, successful termination;

INFO $= -(i \times 100 + j)$, error in the jth component of the ith argument (for example, a component of an array descriptor);

INFO $= -i$, error in the ith argument;

INFO > 0 , error detected during execution, control is returned to the calling program.

It is **essential** to test INFO on exit from the routine.

10.2   Treatment of Matrices by ScaLAPACK

In common with Chapter F04, the routines in Chapter F07 and Chapter F08 use the **cyclic 2-d block distribution** strategy for all matrices and vectors (see Section 8.3 for a full description and example). However, the specification of the problem dimensions are not as straightforward as for other Library routines. We need to elaborate further.

The ScaLAPACK routines offer greater flexibility in dealing with matrices than other linear algebra routines in the Library (e.g., those in Chapters F02, F04 and F11).

Suppose you have an m_A by n_A matrix A. Before a call to an F07 or F08 routine, you must first make sure that A is distributed in a cyclic 2-d block form across the Library Grid. Then, ScaLAPACK allows you to reference and perform operations (such as factorization) on an m by n submatrix A_s. If we denote the starting row and column indices of the submatrix A_s by i_A and j_A respectively, then

$$A_s(1:m, 1:n) \equiv A(i_A : i_A + m - 1, j_A : j_A + n - 1).$$

Note that if $i_A = j_A = 1$, $m = m_A$ and $n = n_A$, then $A_s = A$.

On entry, most routines in ScaLAPACK require information about the matrix A and its submatrix A_s. The information about A_s can be easily provided by specifying its order and its starting row and column coordinate within the matrix A. One issue which is central in understanding ScaLAPACK is: **although it is the matrix A which must be distributed on all the processors across the Library Grid, the operation is only performed on the submatrix A_s.**

As an example, if you look up the specification of F07ADFP, it reads:

```
SUBROUTINE F07ADFP(M, N, A, IA, JA, IDESCA, IPIV, INFO)
ENTRY      PDGETRF(M, N, A, IA, JA, IDESCA, IPIV, INFO)
INTEGER          M, N, IA, JA, IDESCA(9), IPIV(*), INFO
DOUBLE PRECISION  A(*)
```

The ENTRY statement enables the routine to be called by its ScaLAPACK name.

The matrix A is stored in array A in a cyclic 2-d block distribution. Associated with this array are three arguments IA, JA and the descriptor array IDESCA. **In fact all the ScaLAPACK routines which have array argument(s) representing a matrix have three similar arguments for each array**.

On entry to F07ADFP, IA and JA must be set to the row and column coordinate at which the first element of the matrix A_s is stored in matrix A. IDESCA, which is referred to as the **I**nteger array **DESC**riptor for array **A**, should contain information about the array A and some other information.

10.2.1 Array Descriptors

An 'array descriptor' is associated with each array representing a matrix in the argument list of a ScaLAPACK routine. The array stores the information required to establish the mapping between each array entry and its corresponding process and memory location. Array descriptors are provided for

   dense matrices (which are distributed in a cyclic 2-d block form)

   band and tridiagonal matrices (which are distributed in a column block form);

   out-of-core matrices (which use a variation of the above distribution);

In Release 2 of the NAG Parallel Library provision is only made for dense matrices.

The first element of this array indicates the matrix and the distribution types used in the algorithm. Since this Release of the Library only includes ScaLAPACK routines for dense matrices, we merely consider the descriptor array associated with this type of matrices. Here is the definition of all its elements.

IDESCA(1)   the descriptor type: for dense matrices using cyclic 2-d block distribution this must be set to 1;

IDESCA(2)   the BLACS context (ICNTXT) for the Library Grid;

IDESCA(3) m_A, the number of rows of A;

IDESCA(4) n_A, the number of columns of A;

IDESCA(5) M_b, the blocking factor used to distribute the rows of A, i.e., the number of rows stored in a block;

IDESCA(6) N_b, the blocking factor used to distribute the columns of A, i.e., the number of columns stored in a block;

IDESCA(7)   the processor row index over which the first row of A is distributed;

IDESCA(8)   the processor column index over which the first column of A is distributed;

IDESCA(9)   the leading dimension (LDA) of the array A storing the local blocks of A.
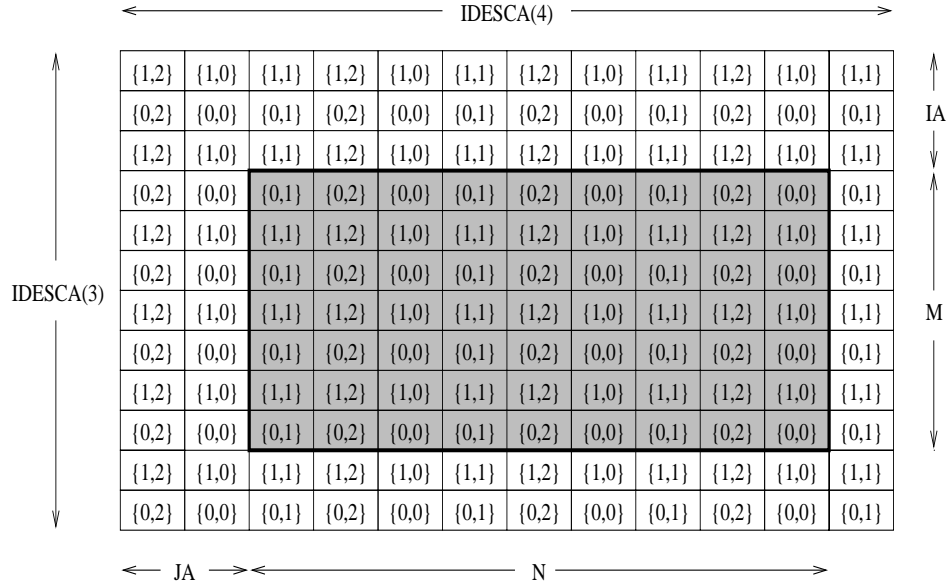
Figure 7
Referencing a submatrix

Note: An array descriptor **does not change**, in general, throughout the life cycle of the matrix with which it is associated.

Figure 7 illustrates graphically the meaning and use of the parameters M, N, IA and JA and of some of the entries of the array descriptor IDESCA. The case depicted shows a matrix distributed over a 2 by 3 logical grid. The first row of the matrix is stored in the second row of the grid (IDESCA(7) = 1), and the first column is stored in the third column of the grid (IDESCA(8) = 2). The indices within each block refers to the processor row and column which stores that block.

As an example, let each small box represent a 4 by 4 block (i.e., IDESCA(5) = IDESCA(6) = 4). Since there are twelve blocks, then A is a 48 by 48 matrix (i.e., IDESCA(3) = IDESCA(4) = 48). Similarly, the shaded area which is occupied by the submatrix A_s has seven row blocks and nine column blocks. Hence, A_s is a 28 by 36 matrix (i.e., M = 28 and N = 36). The row and column indices in which the submatrix A_s starts in the matrix A are 13 and 9 respectively (i.e., IA = 13 and JA = 9).

It is essential that you note the following.

> The second element of the array descriptor for the array A should be set to the context value ICNTXT returned by a call to Z01AAFP.
>
> **Note:** Any error condition due to invalid IDESCA(2) (ICNTXT) will set the value of INFO to $-N02$, where N is the position of IDESCA in the argument list of the ScaLAPACK routine, not -1000 or -2000 as is the convention with IFAIL.

> In all routines in this chapter the local elements of a matrix are stored in a 1-d array. For example, the local elements of the m_A by n_A matrix A can be stored in the real array A($*$). However, it is more convenient to consider A as a 2-d array of dimension (LDA,$*$), where LDA must be greater than or equal to the number of rows of A stored in the specific row of the Library Grid and the array A must have a number of columns greater than or equal to the number of columns of A stored in the specific column of the Library Grid.

> For the default Library mechanism, you may assume that the first row and column of the matrix A is always stored on the $\{0, 0\}$ processor. Hence, IDESCA(7) = IDESCA(8) = 0.

> If the **whole** matrix is referenced, then, IA = 1, JA = 1, M = IDESCA(3) = m_A and N = IDESCA(4) = n_A.

> In general, submatrices must start on the boundary between blocks. Exceptions are the right-hand side matrices for the solution of linear equations which can start in any column of the overall matrices.

We now proceed with an example.

10.3   Solution of Linear Equations Revisited

In this section we consider again the solution of equation (3) as presented in Section 8.2, but using routines from Chapter F07.

If you look up Chapter F07, you will find two routines which can be used in conjunction with each other to solve the equation

$$AX = B.$$

They are:

> **F07ADFP** – performs the LU factorization of a real m by n matrix.

> **F07AEFP** – solves an n by n real system of linear equations with multiple right hand-sides, previously factorized by a call to F07ADFP.

The aim of our next example is to reproduce the solution of equation (3) in Section 8.3.

When using a ScaLAPACK routine, it is always a good practice to set up the array descriptor(s) associated with an array first. We do this in a step by step fashion for the matrix A:

> **Step 1**: set IDESCA(1) = 1;

> **Step 2**: set IDESCA(2). IDESCA(2) determines on which grid the matrix A was distributed. Since we are using the Library Grid, IDESCA(2) may be set to the value of context which was returned by a call to Z01AAFP. Hence, IDESCA(2) = ICNTXT

> **Step 3**: A is an 8 by 8 matrix, so we must set IDESCA(3) = 8 and IDESCA(4) = 8;

> **Step 4**: a square block of size 2 was used in Program **EX3C**, so IDESCA(5) = 2 and IDESCA(6) = 2;

> **Step 5**: the first row and column of the matrix A was distributed on the first row and the first column of the Library Grid, so IDESCA(7) = 0 and IDESCA(8) = 0.

> **Step 6**: the leading dimension of the array A was set to 4 in Program **EX3C**, so IDESCA(9) = 4.

The next stage is to set up the arguments associated with the submatrix A_s. The matrix which appears in the linear system (3) and needs to be factorized is the matrix A itself. In other words, $A_s = A$. In this case, we simply set IA = JA = 1.

Here is a program which incorporates the above settings of IDESCA, IA and JA to compute the solution of the linear system (3):

```
 1:        PROGRAM EX5A
 2:        INTEGER          N,LDA,LDB,NRHS,NB,NIN,NOUT,NIP,NL,NRHSL,DT
 3:        PARAMETER        (N=8,NRHS=1,NB=2,NIN=1,NOUT=6,DT=1)
 4:        PARAMETER        (LDA=4,LDB=LDA,NL=4,NRHSL=NRHS,NIP=NB+NL)
 5:        INTEGER          ICNTXT,IFAIL,MP,NP,IA,JA,IB,JB
 6:        INTEGER          IDESCA(9),IDESCB(9),IPIV(NIP)
 7:        CHARACTER*1      TRANS
 8:        CHARACTER*80     FORMAT
 9:        DOUBLE PRECISION A(LDA,NL),B(LDB,NRHSL),WORK(N)
10         LOGICAL          Z01ACFP
11:        MP = 2
12:        NP = 2
13:        IFAIL = 0
14:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
15:        OPEN (UNIT=NIN,.'/tutorial_ex3a.dat')
16:        IDESCA(1) = DT
17:        IDESCA(2) = ICNTXT
18:        IDESCA(3) = N
19:        IDESCA(4) = N
20:        IDESCA(5) = NB
21:        IDESCA(6) = NB
22:        IDESCA(7) = 0
```

```
23:        IDESCA(8) = 0
24:        IDESCA(9) = LDA
25:        IA = 1
26:        JA = 1
27:        CALL X04BCFP(NIN,N,N,A,1,1,IDESCA,IFAIL)
28:        CALL F07ADFP(N,N,A,IA,JA,IDESCA,IPIV,INFO)
29:        IF (INFO .NE. 0) GO TO 10
30:        IDESCB(1) = DT
31:        IDESCB(2) = IDESCA(2)
32:        IDESCB(3) = N
33:        IDESCB(4) = NRHS
34:        IDESCB(5) = NB
35:        IDESCB(6) = NB
36:        IDESCB(7) = 0
37:        IDESCB(8) = 0
38:        IDESCB(9) = LDB
39:        IB = 1
40:        JB = 1
41:        IFAIL = 0
42:        CALL X04BCFP(NIN,N,NRHS,B,1,1,IDESCB,IFAIL)
43:        TRANS = 'N'
44:        CALL F07AEFP(TRANS,N,NRHS,A,IA,JA,IDESCA,IPIV,B,IB,JB,IDESCB,INFO)
45:        IF (Z01ACFP()) WRITE (NOUT,FMT=*) 'Solution(s)'
46:        FORMAT = '(F9.4)'
47:        CALL X04BDFP(NOUT,N,NRHS,B,1,1,IDESCB,FORMAT,WORK,IFAIL)
48: 10     CONTINUE
49:        CLOSE (NIN)
50:        CALL Z01ABFP(ICNTXT,'N',IFAIL)
51:        STOP
52:        END
```

Using the data file `tutorial_ex3a.dat`, we obtain:

```
Solution(s)
  1.0000
 -1.0000
  3.0000
 -5.0000
  1.0000
  2.0000
  3.0000
 -4.0000
```

which is the result obtained in Section 8.3.

Calls to X04BCFP at `lines 27` and `42` read and distribute the matrices A and B in cyclic 2-d block form across the Library Grid. The subroutine X04BCFP has the same functionality as X04BGFP, except that the user interface is designed to be compatible with the ScaLAPACK routines. A call to F07ADFP at `line 28` computes the LU factorization of the submatrix A_s $(= A)$.

Before a call to F07AEFP, we need to set up the array descriptor IDESCB associated with the array B. This is set at `lines 30-38` in exactly the same manner as IDESCA. Since the arrays A and B are participating on the same Library Grid, it is important that you also set IDESCB(2) to the context ICNTXT generated by Z01AAFP (as indicated by `line 31`).

After calling F07AEFP, a call to X04BDFP at `line 47` outputs the solution. Again, X04BDFP has the same functionality as X04BHFP, except that the user interface is designed to be compatible with the ScaLAPACK routines.

Finally, a call to Z01ABFP on `line 50` un-defines the Library Grid.

10.4 Solving Linear Equations Involving submatrices

Suppose that the matrices A and B in (3) (Section 8) were partitioned in the following manner:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} \tag{5}$$

where each A_i for $i = 1, 2, 3, 4$ is a 4 by 4 matrix and each B_i for $i = 1, 2$ is a 4 by 1 matrix.

Using the data file `tutorial_ex3a.dat`, we wish to show how you can solve $A_4 X = B_2$, where

$$A_4 = \begin{pmatrix} 2.10 & 0.25 & -1.62 & 0.70 \\ -2.25 & 1.12 & 2.65 & -0.86 \\ 1.10 & -0.44 & 3.20 & 2.00 \\ 0.40 & 0.26 & -2.44 & 0.80 \end{pmatrix} \quad \text{and} \quad B_2 = \begin{pmatrix} 10.44 \\ -0.12 \\ -27.86 \\ -46.05 \end{pmatrix}. \tag{6}$$

However, a simple way of solving (6) is to use Program **EX5A** by setting the values of N = 4, LDA = 2 and NL = 2 with the following data file:

```
 2.10  0.25 -1.62  0.70
-2.25  1.12  2.65 -0.86
 1.10 -0.44  3.20  2.00
 0.40  0.26 -2.44  0.80     : End of matrix A
 10.44
 -0.12
-27.86
-46.05     : End of rhs B
```

If you do that, you should get the following results:

```
Solution(s)
 26.3758
 -1.3300
  8.9438
-43.0394
```

The disadvantage of the above approach is the situation when you already have the matrices A and B distributed in cyclic 2-d block distribution and you are only interested in solving equations which involve submatrices of A and B.

To solve $A_4 X = B_2$ with A and B already distributed, we can use the approach adopted in Section 10.3 with a few minor modifications:

(1) N must be set to 4;

(2) the array elements IDESCA(3), IDESCA(4) and IDESCB(3) should be set to 8, which signifies the number of rows and columns of the matrices A and B. The values of input arguments to X04 routines should also be changed accordingly.

(3) the matrix that we want to factorize is the submatrix A_4, which is a 4 by 4 matrix and has its first element at the fifth row and the fifth column of the matrix A. In this case, we simply need to set IA = JA = 5.

(4) Likewise, for submatrix B_2, we need to set IB = 5 and JB = 1.

Here is the program which incorporates the above values of IA, JA, IB and JB:

```
1:      PROGRAM EX5B
2:      INTEGER         N,LDA,LDB,NRHS,NB,NIN,NOUT,NIP,NL,NRHSL,DT,NA
3:      PARAMETER       (N=4,NRHS=1,NB=2,NIN=1,NOUT=6,DT=1,NA=8)
4:      PARAMETER       (LDA=4,LDB=LDA,NL=4,NRHSL=NRHS,NIP=NB+NL)
5:      INTEGER         ICNTXT,IFAIL,MP,NP,IA,JA,IB,JB
6:      INTEGER         IDESCA(9),IDESCB(9),IPIV(NIP)
```

```
 7:        CHARACTER*1      TRANS
 8:        CHARACTER*80     FORMAT
 9:        DOUBLE PRECISION A(LDA,NA),B(LDB,NRHSL),WORK(NA)
10        LOGICAL           Z01ACFP
11:        MP = 2
12:        NP = 2
13:        IFAIL = 0
14:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
15:        OPEN (UNIT=NIN,./tutorial_ex3a.dat')
16:        IDESCA(1) = DT
17:        IDESCA(2) = ICNTXT
18:        IDESCA(3) = NA
19:        IDESCA(4) = NA
20:        IDESCA(5) = NB
21:        IDESCA(6) = NB
22:        IDESCA(7) = 0
23:        IDESCA(8) = 0
24:        IDESCA(9) = LDA
25:        IA = 5
26:        JA = 5
27:        CALL X04BCFP(NIN,NA,NA,A,1,1,IDESCA,IFAIL)
28:        CALL F07ADFP(N,N,A,IA,JA,IDESCA,IPIV,INFO)
29:        IF (INFO .NE. 0) GO TO 10
30:        IDESCB(1) = DT
31:        IDESCB(2) = IDESCA(2)
32:        IDESCB(3) = NA
33:        IDESCB(4) = NRHS
34:        IDESCB(5) = NB
35:        IDESCB(6) = NB
36:        IDESCB(7) = 0
37:        IDESCB(8) = 0
38:        IDESCB(9) = LDB
39:        IB = 5
40:        JB = 1
41:        IFAIL = 0
42:        CALL X04BCFP(NIN,NA,NRHS,B,1,1,IDESCB,IFAIL)
43:        TRANS = 'N'
44:        CALL F07AEFP(TRANS,N,NRHS,A,IA,JA,IDESCA,IPIV,B,IB,JB,IDESCB,INFO)
45:        IF (Z01ACFP()) WRITE (NOUT,FMT=*) 'Solution(s)'
46:        FORMAT = '(F9.4)'
47:        CALL X04BDFP(NOUT,N,NRHS,B,IB,JB,IDESCB,FORMAT,WORK,IFAIL)
48: 10     CONTINUE
49:        CLOSE (NIN)
50:        CALL Z01ABFP(ICNTXT,'N',IFAIL)
51:        STOP
52:        END
```

Using the data file `tutorial_ex3a.dat` we obtain the correct results:

```
    Solution(s)
     26.3758
     -1.3300
      8.9438
    -43.0394
```

Note that calls to X04BCFP at **lines 27** and **42** distribute the whole matrices A and B, **not** the submatrices A_4 and B_2.

11 Example 6: Using the NAG Parallel Library with the NAG Fortran Library

One of NAG's most successful products is its Fortran 77 Library, a library which contains over 1000 well-documented user-callable routines covering areas such as linear algebra, differential equations, quadrature, Fast Fourier Transform (FFT), optimization and statistics.

The NAG Parallel Library is distinct from the Fortran 77 Library, but you may call routines from both Libraries within the same program unit, provided that implementations of both Libraries are available for your system and compiler. In fact, a small number of routines in the NAG Parallel Library, whose names do not end with a 'P', were taken from the Fortran 77 Library.

Clearly the key application areas for which these Libraries may be used together are those areas in which the application of a numerical process generates independent tasks, each of which can be executed simultaneously on different processors. As an example, the equation which is concerned with the model of air flow over hills and small mountains is governed by a singular integral equation with a *convolution* kernel function on an infinite domain. This equation can be solved by splitting the Fourier transform of the solution into a singular and a non-singular part. The non-singular part can be computed by an appropriate FFT routine, whereas the singular part can be evaluated by an adaptive quadrature routine suitable for integrals on a semi-infinite interval. To speed up the process of solving this equation, it is conceivable to perform the task of computing the Fourier transforms and the evaluation of the integrals simultaneously on different processors. This can be achieved by setting up a Library Grid by a call to Z01AAFP, followed by calls to the quadrature and FFT routines in the NAG Fortran 77 Library (which provides extensive coverage in these areas). You have to note that issues such as load-balancing, synchronization and communication become your responsibilities. However, continuing the spirit of the Tutorial, we do not address the solution of the above integral equation here, but present a much simpler example which involves the computation of discrete Fourier transforms (DFTs).

In many application areas, it is often required to compute DFTs of several sequences of the same length. The aim of this section is to show you how this can be done in parallel using routines from the NAG Fortran 77 and the NAG Parallel Libraries.

The problem to be tackled in this section is to compute the DFT of m sequences, each containing n real data values. The DFT of n real data values is a particular type of complex sequence, called a **Hermitian** sequence which can be represented by only n, rather than $2n$, independent real values. For other properties of the Fourier transform, you should consult a standard textbook on the subject.

Chapter C06 of the NAG Fortran 77 Library is concerned with the computation of the discrete Fourier transform of a sequence of real or complex data values, as well as the computation of inverse Laplace transform, the direct summation of orthogonal series, etc. C06EAF is a NAG Fortran 77 Library routine which computes the discrete Fourier transform of a sequence of n real data values. Its specification is:

```
SUBROUTINE C06EAF(X,N,IFAIL)
INTEGER        N, IFAIL
DOUBLE PRECISION  X(N)
```

On entry, the array X must contain the sequence to be transformed, and on exit, X is over-written by the discrete Fourier transform stored in Hermitian form.

By setting $m = 4$ and $n = 6$, our aim is to use the mechanism used in the NAG Parallel Library to set up a 2 by 2 Library Grid and compute the discrete Fourier transform of the four sequences independently, one on each processor. Using the data file `tutorial_ex6a.dat`:

```
4 6
0.3490 0.3854 0.5417 0.9172
0.5489 0.6772 0.2983 0.0644
0.7478 0.1138 0.1181 0.6037
0.9446 0.6751 0.7255 0.6430
1.1385 0.6362 0.8638 0.0428
1.3285 0.1424 0.8723 0.4815
```

the following simple program reads in the four sequences on each processor and computes the discrete Fourier transform of the first sequence on $\{0, 0\}$ processor, the second sequence on $\{0, 1\}$ processor and so on. Here is the program:

```
 1:        PROGRAM EX6A
 2:        INTEGER          LDX,NIN, NOUT
 3:        PARAMETER        (NIN=1,NOUT=6,LDX=20)
 4:        INTEGER          IFAIL,I,J,ICNTXT,MP,NP,MPR,NPR,MYROW,MYCOL
 5:        DOUBLE PRECISION X(LDX,4)
 6:        MP = 2
 7:        NP = 2
 8:        IFAIL = 0
 9:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
10:        OPEN (UNIT=NIN,FILE='./tutorial_ex6a.dat')
11:        READ(NIN,*) M, N
12:        DO 10 I = 1, N
13:           READ(NIN,*) (X(I,J), J = 1, M)
14: 10    CONTINUE
15:        CALL BLACS_GRIDINFO(ICNTXT,MPR,NPR,MYROW,MYCOL)
16:        INDX =  NP*MYROW+MYCOL+1
17:        IFAIL = 0
18:        CALL C06EAF(X(1,INDX),N,IFAIL)
19:        CLOSE(NIN)
20:        CALL Z01ABFP(ICNTXT,'N',IFAIL)
21:        STOP
22:        END
```

This simple program does not generate any output. `line 9` sets up the 2 by 2 Library Grid and `lines 10-14` opens the data file `tutorial_ex6a.dat` and reads its contents. `line 16` sets the processor position INDX (= 1, 2, 3 or 4) using the row and column grid coordinates of the calling processor which were returned by a call to the BLACS routine BLACS_GRIDINFO. A description of BLACS_GRIDINFO was given in Section 7.7.

In order to show that Program `EX6A` indeed produces the correct result, we need to print the DFT of the sequences, and compute and print their inverse transforms to show that the original sequences are restored. Here is the program:

```
 1:        PROGRAM EX6B
 2:        INTEGER          LDX,NIN, NOUT
 3:        PARAMETER        (NIN=1,NOUT=6,LDX=20)
 4:        INTEGER          IFAIL,I,J,ICNTXT,MP,NP,MPR,NPR,MYROW,MYCOL
 5:        DOUBLE PRECISION X(LDX,4)
 6:        LOGICAL Z01ACFP
 7:        MP = 2
 8:        NP = 2
 9:        IFAIL = 0
10:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
11:        OPEN (UNIT=NIN,FILE='./tutorial_ex6a.dat')
12:        READ(NIN,*) M, N
13:        DO 10 I = 1, N
14:           READ(NIN,*) (X(I,J), J = 1, M)
15: 10    CONTINUE
16:        CALL BLACS_GRIDINFO(ICNTXT,MPR,NPR,MYROW,MYCOL)
17:        INDX =  NP*MYROW+MYCOL+1
18:        IFAIL = 0
19:        CALL C06EAF(X(1,INDX),N,IFAIL)
20:        IF (Z01ACFP())
21:     +     WRITE (NOUT,*) 'Discrete Fourier transform in Hermitian form'
22:        CALL SNDRCV(ICNTXT,X,MP,NP,M,N,LDX,INDX,NOUT)
23:        CALL C06GBF(X(1,INDX),N,IFAIL)
24:        CALL C06EBF(X(1,INDX),N,IFAIL)
25:        IF (Z01ACFP()) WRITE (NOUT,*)
26:     +     'Original data as restored by inverse transform'
```

```
27:       CALL SNDRCV(ICNTXT,X,MP,NP,M,N,LDX,INDX,NOUT)
28:       CALL Z01ABFP(ICNTXT,'N',IFAIL)
29:       STOP
30:       END
31:
32:       SUBROUTINE SNDRCV(ICNTXT,X,MP,NP,MYROW,MYCOL,M,N,LDX,INDX,NOUT)
33:       INTEGER           ICNTXT,MP,NP,N,INDX,I,J,INDXR,LDX,NOUT
34:       DOUBLE PRECISION X(LDX,M)
35:       LOGICAL           Z01ACFP
36:       IF (.NOT. Z01ACFP()) THEN
37:          CALL IGESD2D(ICNTXT,1,1,INDX,1,0,0)
38:          CALL DGESD2D(ICNTXT,N,1,X(1,INDX),LDX,0,0)
39:       ELSE
40:          DO 20 J = 0,NP - 1
41:             DO 10 I = 0,MP - 1
42:                IF (I.NE.0 .OR. J.NE.0) THEN
43:                   CALL IGERV2D(ICNTXT,1,1,INDXR,1,I,J)
44:                   CALL DGERV2D(ICNTXT,N,1,X(1,INDXR),LDX,I,J)
45:                END IF
46:    10       CONTINUE
47:    20    CONTINUE
48:          DO 30 I = 1, N
49:             WRITE (NOUT,9999) (X(I,J), J = 1, M)
50: 30       CONTINUE
51:          WRITE (NOUT,*)
52:       END IF
53:       RETURN
54: 9999 FORMAT(1X,4F9.4)
55:       END
```

Using the data file `tutorial_ex6a.dat` once more, Program EX6B reads in four real sequences of length 6 on each processor and prints their DFTs in Hermitian form. It then performs the inverse transforms and prints the original sequences. Here are the results:

```
Discrete Fourier transform in Hermitian form
   2.0646    1.0737    1.3961    1.1237
  -0.2450   -0.1041   -0.0365    0.0914
  -0.2402    0.1126    0.0780    0.3936
  -0.2395   -0.1467   -0.1521    0.1530
   0.1375   -0.3738   -0.0607    0.3458
   0.4138   -0.0044    0.4666   -0.0508


Original data as restored by inverse transform
   0.3490    0.3854    0.5417    0.9172
   0.5489    0.6772    0.2983    0.0644
   0.7478    0.1138    0.1181    0.6037
   0.9446    0.6751    0.7255    0.6430
   1.1385    0.6362    0.8638    0.0428
   1.3285    0.1424    0.8723    0.4815
```

Up to `line 20`, Program EX6B does exactly the same as Program EX6A, i.e., it evaluates the four DFTs independently on different processors. But before going any further, a few words need to be said about the evaluation of the inverse transform.

To compute the inverse transform of a Hermitian sequence (DFT of a real sequence), you first need to form its complex conjugate and then compute the DFT of the conjugated sequence. Calls to C06GBF and C06EBF respectively at `lines 23-24` perform these tasks.

The collection and printing of the results on the root processor are performed by the subroutine SNDRCV in `lines 32-55`. SNDRCV uses four BLACS routines: IGESD2D, IGERV2D, DGESD2D and

DGERV2D. To print the data, it is necessary to collect them on the root processor. The `IF` statement at `line 36` checks whether the calling processor is the root processor or not. If it is not, the data is sent (`lines 37-38`) by the BLACS routines IGESD2D (sends integer data) and DGESD2D (sends real data) to the root processor. The root processor in turn (`lines 40-47`) receives the integer data and the real data by calls to IGERV2D and DGERV2D respectively. Finally, once all the data have been received by the root processor the results are printed (as indicated on `lines 48-50`).

12 Example 7: Advanced Features

12.1 Reshaping the Grid

This example shows how to un-define a Library Grid and declare a grid of a different size and shape. We use an optimization problem for this purpose.

The problem that we want to tackle is to minimize Rosenbrock's function:

$$F(x_1, x_2) = f_1^2 + f_2^2, \quad \text{where} \ \ f_1 = 10(x_2 - x_1^2), \ \ \text{and} \ \ f_2 = (1 - x_1). \tag{7}$$

Chapter E04 of the NAG Parallel Library contains the routine E04FDFP which can solve (7). The function (7) must be defined in a user-supplied subroutine.

The program declares a 2 by 2 grid and uses it to minimize Rosenbrock's function. Then it changes the grid size to 2 by 1 and solves the same problem.

Here is the program which uses the starting point $(-1.2, 1.0)$ to minimize (7) on two different grid sizes; the minimum of the function occurs at $(1.0, 1.0)$ and has a value 0.0:

```
 1:        PROGRAM EX7A
 2:        INTEGER           NOUT,LW,LIW,M,N,NB
 3:        PARAMETER         (NOUT=6,LW=200,LIW=200,M=2,N=2,NB=1)
 4:        DOUBLE PRECISION  FVAL,TAU
 5:        INTEGER           ICNTXT,IFAIL,ITERS,IW,K,MP,NP
 6:        DOUBLE PRECISION  W(LW),X(N)
 7:        LOGICAL           Z01ACFP
 8:        EXTERNAL          FUNC
 9:        MP = 2
10:        NP = 2
11:        IFAIL = 0
12:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
13:        X(1) = -1.2D0
14:        X(2) = 1.0D0
15:        TAU = 1.0D-4
16:        ITERS = 20
17:        IF (Z01ACFP()) WRITE (NOUT,*) 'Testing with ',NP*MP,' processors:'
18:        CALL E04FDFP(ICNTXT,M,N,NB,ITERS,TAU,X,FVAL,FUNC,W,LW,IW,LIW,
19:       +             IFAIL)
20:        IF (Z01ACFP()) WRITE (NOUT,9999) FVAL, (X(K),K=1,2)
21:        CALL Z01ABFP(ICNTXT,'Y',IFAIL)
22:        MP = 2
23:        NP = 1
24:        IFAIL = 0
25:        CALL Z01AAFP(ICNTXT,MP,NP,IFAIL)
26:        X(1) = -1.2D0
27:        X(2) = 1.0D0
28:        TAU = 1.0D-4
29:        ITERS = 20
30:        IF (Z01ACFP()) THEN
31:           WRITE (NOUT,*)
32:           WRITE (NOUT,*) 'Testing with ',NP*MP,' processors:'
33:        END IF
34:        CALL E04FDFP(ICNTXT,M,N,NB,ITERS,TAU,X,FVAL,FUNC,W,LW,IW,LIW,
```

```
35:      +                  IFAIL)
36:         IF (Z01ACFP()) WRITE (NOUT,9999) FVAL, (X(K),K=1,2)
37:         CALL Z01ABFP(ICNTXT,'N',IFAIL)
38:         STOP
39: 9999 FORMAT (1X,'Minimum value ',F9.4,'  found at ',2F9.4)
40:         END
41:
42:         SUBROUTINE FUNC(M,N,F,X)
43:         INTEGER M,N
44:         DOUBLE PRECISION F(M),X(N)
45:         F(1) = 10.0D0* (X(2)-X(1)*X(1))
46:         F(2) = 1.0D0 - X(1)
47:         RETURN
48:         END
```

If you run this program, you should get the following results:

```
Testing with            4 processors:
Minimum value    0.0000  found at    1.0000    1.0000


Testing with            2 processors:
Minimum value    0.0000  found at    1.0000    1.0000
```

`Lines 9-12` declare the initial 2 by 2 grid in the same way as previous examples. This sets the maximum number of processors that can be used in further Library Grids to 4. `Lines 13-16` initialize the input data for the problem. The array X is an initial guess at the minimization point, TAU is a nominal accuracy requirement, and ITERS specifies the maximum number of iterations that are allowed before the routine must give up trying to find the minimum (if all goes well it will reach a solution before this number of iterations).

`Line 18` is the call to the optimization routine E04FDFP and `line 20` prints the solution. On `line 21` a call to Z01ABFP is made with the continuation argument CONT set to 'Y' to inform the Library mechanism that a further grid will be declared.

`Lines 22-25` initialize a 2 by 1 Library Grid and `lines 26-29` set up the input arguments to be the same as the first call to E04FDFP. Another call to E04FDFP is made on `line 34` and the solution is printed once again on `line 36` so as to ensure that E04FDFP produces the same result on different Library Grids.

A call to Z01ABFP on `line 37` un-defines the grid and indicates that no further Library calls will be made by setting CONT = 'N'.

On `lines 42-48`, the subroutine argument FUNC defines Rosenbrock's function.

12.2   Multigridding

In this section we consider an example of multigridding – using more than one logical grid of processors to achieve different tasks in parallel. Two separate logical grids of processors are created:

A 2 by 2 Library Grid – this grid will be used to run E04FDFP to minimize Rosenbrock's function (7). Naturally, Z01AAFP is used to create this grid with the context LCNTXT.

A 2 by 1 user-defined grid – this will be used to evaluate the global sum of an integer value and make the result available to all the processors in this grid. The user-defined grid will be created by using the BLACS routine BLACS_GRIDMAP with the context UCNTXT.

Explicit calls to PVM and BLACS routines are made in the example in this section and we assume that you are familiar with PVM and BLACS and the terminology associated with them.

In order to create different grids, you should take the following steps.

(1) Spawn the total number of processors (in our case 6; 4 for the Library Grid and 2 for the user-defined grid) which you require to participate in your multigridding environment. To do this, you should use the PVM spawning routine PVMFSPAWN which also generates a 'task identifier' associated with each processor.

(2) After the spawning process, you should register each task identifier with BLACS. This can be done by a call to BLACS routine SETPVMTIDS.

(3) It is essential to let the Library mechanism know that you have already spawned processes without resorting to the default mechanism (i.e., not by a call to Z01AAFP). To do this you must make sure that you call the utility routine Z01ADFP prior to a call to Z01AAFP.

(4) Create your Library Grid by a call to Z01AAFP.

(5) Create the user-defined grid by a call to the BLACS routine BLACS_GRIDMAP.

You must make sure that the two grids have different contexts.

Here is the program which achieves the above tasks:

```
 1:       PROGRAM EX7B
 2:       INCLUDE 'fpvm3.h'
 3:       INTEGER          NOUT,LW,LIW,M,N,NB,LDU
 4:       PARAMETER        (NOUT=6,LW=2000,LIW=2000,M=2,N=2,NB=1,LDU=2)
 5:       INTEGER          BUFID,IFAIL,INFO,ITERS,K,LCNTXT,MP,MPU,MYTID,NP,NPCOL,
 6:      +                 NPROW,NPU,NTASKS,NUMT,PARTID,UCNTXT,UCOL,UROW
 7:       INTEGER          ISUM(1),IW(LIW),TIDS(10),USRMAP(LDU,1)
 8:       DOUBLE PRECISION FVAL,TAU,W(LW),X(N)
 9:       LOGICAL          Z01ACFP
10:       EXTERNAL         FUNC
11:       CALL PVMFMYTID(MYTID)
12:       CALL PVMFPARENT(PARTID)
13:       NTASKS = 5
14:       IF (PARTID.EQ.PVMNOPARENT) THEN
15:          CALL PVMFSPAWN ('ex7_b',PVMTASKDEFAULT,'*',NTASKS,TIDS(2),
16:      +                   NUMT)
17:          TIDS(1) = MYTID
18:          CALL PVMFINITSEND(PVMRAW,INFO)
19:          CALL PVMFPACK(INTEGER4,TIDS,NTASKS+1,1,INFO)
20:          CALL PVMFMCAST(NTASKS,TIDS(2),0,INFO)
21:       ELSE
22:          CALL PVMFRECV(-1,0,BUFID)
23:          CALL PVMFUNPACK(INTEGER4,TIDS,NTASKS+1,1,INFO)
24:       END IF
25:       CALL SETPVMTIDS(NTASKS+1,TIDS)
26:       CALL Z01ADFP
27:       CALL BLACS_GET(0,0,LCNTXT)
28:       CALL BLACS_GET(0,0,UCNTXT)
29:       MP = 2
30:       NP = 2
31:       IFAIL = 0
32:       CALL Z01AAFP(LCNTXT,MP,NP,IFAIL)
33:       MPU = 2
34:       NPU = 1
35:       USRMAP(1,1) = TIDS(5)
36:       USRMAP(2,1) = TIDS(6)
37:       CALL BLACS_GRIDMAP(UCNTXT,USRMAP,LDU,MPU,NPU)
38:       X(1) = -1.2D0
39:       X(2) = 1.0D0
40:       TAU = 1.0D-4
41:       ITERS = 20
42:       CALL E04FDFP(LCNTXT,M,N,NB,ITERS,TAU,X,FVAL,FUNC,W,LW,IW,LIW,
43:      +             IFAIL)
44:       CALL BLACS_GRIDINFO(UCNTXT,NPROW,NPCOL,UROW,UCOL)
45:       IF (UROW.LT.NPROW .AND. UCOL.LT.NPCOL) THEN
```

```
46:          ISUM(1) = 1
47:          CALL IGSUM2D(UCNTXT,'ALL',' ',1,1,ISUM,1,-1,-1)
48:          IF (UROW.EQ.0 .AND. UCOL.EQ.0) THEN
49:             OPEN (NOUT,FILE='ex7_b_res')
50:             WRITE (NOUT,*) ISUM(1)
51:             CLOSE (NOUT)
52:          ENDIF
53:          CALL BLACS_GRIDEXIT(UCNTXT)
54:       END IF
55:       IF (Z01ACFP()) WRITE (NOUT,9999) FVAL, (X(K),K=1,2)
56:       CALL Z01ABFP(LCNTXT,'Y',IFAIL)
57:       CALL PVMFEXIT(INFO)
58:       STOP
59: 9999 FORMAT (1X,'Minimum value ',F9.4,'  found at ',2F9.4)
60:       END
61:
62:       SUBROUTINE FUNC(M,N,F,X)
63:       INTEGER M,N
64:       DOUBLE PRECISION F(M),X(N)
65:       F(1) = 10.0D0* (X(2)-X(1)*X(1))
66:       F(2) = 1.0D0 - X(1)
67:       RETURN
68:       END
```

If you run this you should get

```
    Minimum value    0.0000  found at    1.0000    1.0000
```

and an output file `ex7_b_res` in your home directory which should contain the value 2.

`Lines 11-12` enrol each program (task) with PVM and return the task identifier of the parent (root) process. The parent process is responsible for spawning the other processes that will form the Library Grid. `line 14` identifies the parent process, and a call to PVMFSPAWN on `line 15` executes five identical copies of this program (with a fixed name `ex7_b`). The PVM task identifiers for all of the processes (parent and child) must be broadcast to all of the processes which will be needed later by BLACS routine SETPVMTIDS. The broadcast is achieved by `lines 18-23`.

All processors then register their task identifiers with the BLACS through a call to SETPVMTIDS on `line 25`. This routine lets the BLACS know which processes it can claim to map onto a specific logical processor grid.

`line 26` is a call to Z01ADFP which lets the Library mechanism know that the user has already spawned processes without resorting to the default Library mechanism.

`Lines 29-32` declare to 2 by 2 processor grid for use by the Library. This claims the first four processes in the task identifiers array TIDS for use by the Library. Two remaining tasks are available for exclusive use as part of another processor grid.

`Lines 33-37` make use of a BLACS utility, BLACS_GRIDMAP, to map the tasks represented by TIDS(5) and TIDS(6) onto a 2 by 1 grid of logical processors. At this point two processor grids exist, one associated with the context LCNTXT for use by Library routines (the Library Grid) and another associated with the context UCNTXT (the user's defined grid).

The Library Grid is used to solve the same optimization problem shown in Section 12.1. A call to E04FDFP is made on `line 42` with the context LCNTXT. All NAG Parallel Library routines immediately return if they are not part of the Library processor grid, making the calling processor available for other uses.

The call to BLACS_GRIDINFO on `line 44` returns information about the calling processors' membership of the processor grid associated with the context UCNTXT. The processors which are 'in context' are identified by the condition at `line 45`. Processors 'in context' compute a global sum of integers held in a vector ISUM. ISUM is initialised on `line 46` and a call to the BLACS routine IGSUM2D on `line 47`

computes the global sum and makes the result available to all processors 'in context'. One processor (the processor {0,0} in the user defined grid) is designated to output the result to an external file. The condition on `line 48` selects this processor and ISUM(1) is output to the external file `ex7_b_res`. `line 53` un-defines the user-defined grid by a call to BLACS_GRIDEXIT.

The results of the optimization calculation are output on `Line 55`. A call to Z01ABFP on `lines 56` un-defines the Library Grid and lets the Library mechanism know that no further grids will be declared.

Finally, the program de-registers with PVM by a call to PVMFEXIT on `line 57`.

13   Concluding Remarks

The aim of this Tutorial has been to help you to become familiar with the NAG Parallel Library and to develop confidence in using it.

If there are topics which are not yet covered, but which you would find helpful, please let us know, so that we can consider them for a future revision.

14   References

[**1**]   Dongarra J J and Whaley R C (1995) A users' guide to the BLACS v1.0. *LAPACK Working Note 94 (Technical Report CS-95-281)* Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA.
URL: http://www.netlib.org/lapack/lawns/lawn94.ps

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

[**3**]   (1960–1976) Collected algorithms from ACM index by subject to algorithms

[**4**]   (1978) American National Standard Fortran *Publication X3.9* American National Standards Institute