Parallelization of a sparse matrix-vector multiplication algorithm

About the efficiency of the I/O library of MPI-2 using a parallelized algorithm of a sparse matrix-vector multiplication

M.M.P. van Hulten

May 19, 2006

Contents

1 Introduction

The purpose of this research is to find out what is the best way to parallelize a program which processes so much data that I/O is unavoidable.

A not so smart solution to this problem is to just parallelize the program, don't mind the I/O in all processes and hope everything will go fine.

In the scope of this research two smarter solutions are suggested and compared. The first is to handle the I/O by one process and send the necessary data from this process to other processes. The second is to use an extra layer of software that handles I/O.

The hypothesis is that the second solution is more efficient, because in that case messages would be passed directly between the filesystem and a process, so no extra communication should be needed between processes. This however all depends on the implementation of this extra I/O handling layer.

2 Theory

For the parallelization of the calculation MPI, Message Passing Interface, is used. MPI I/O is used for the extra I/O layer, which is contained in the MPI-2 standard.

For a detailed background about MPI see [2] and for MPI-2 specific features like MPI-IO, see [3].

The matrices used in the benchmarks are sparse and therefore compressed. The format used to accomplish this is Compressed Row Storage (CRS) [1]. This format contains three different types of arrays. The first is a single array, containing the number of non-zero elements in each row. For each row there is an array containing the non-zero elements, as well as an array containing the column index of each of these non-zero elements.

3 Version background

Initially the parallelized version of the program would have been based on mod3a 4.2, but the structure of the program didn't suffice. There is a working version based on version 4.2, named 'mod3a 5.0pre4'. The job distribution is very inefficient. Therefore a rewrite is done, giving us 'mod3a 6.0rc2', the MPI I/O implementation, and 'mod3a 6.1rc3', the normal I/O version. These two versions are similar in structure and discussed in this paper. 'mod3a 6.0rc2' is working, but it possibly needs a bit of polishing. In the following I'll call this just 'version 6.0' or the MPI I/O version. The 'rc' in 'mod3a 6.1rc3' is rather pretentious, because it actually does not work (rc1 does, but is not efficient). In the following I'll just call this 'version 6.1' or the version without MPI I/O.
Both versions 6.0 and 6.1 are included as an appendix of this paper. Other versions are also available [5]. All versions are licensed under the GNU General Public License version 2 [4] and later versions of the GNU GPL.

4 Setup

The Fortran source code is shown in appendix B. See the following sections for an explanation of the relevant source.

4.1 Normal I/O (mod3a 6.1)

For the version without MPI I/O (appendix B.2), the data generation is done on the master process, as follows.

```
If ( myid .eq. 0 ) Then
    Do i = 1, n
        Call genraja( m, n, i, na, lura, luja, writim )
    End Do
End If
```

Here genraja() generates a row of data in Compressed Row Storage (CRS) format. The data is written to the file lura (containing the non-zero elements) and luja (containing the column indices of the non-zero elements).
After rewinding lura and luja and putting in a MPI_Barrier(), the calculation will start, but just 'jobsize' rows per run, to avoid too many MPI_Send() and MPI_Recv() calls at the same time. Within the loop the job is divided over all processes (except for process zero) al follows.

```
Do i = 1, nprocs-1
    If ( myid == 0 ) Then
        Allocate( ra( offset(i)+1:offset(i+1) ), stat=alloc_stat )
        Allocate( ja( offset(i)+1:offset(i+1) ), stat=alloc_stat )

        ! Read and distribute rows (ra and ja)
        Read( lura ) ra
        Read( luja ) ja
        Call MPI_Send( ra( offset(i)+1 : offset(i+1) ), &
                               offset(i+1) - offset(i), MPI_REAL8, i, 1, &
                               MPI_COMM_WORLD, ierr )
        Call MPI_Send( ja( offset(i)+1 : offset(i+1) ), &
                               offset(i+1) - offset(i), MPI_INTEGER, i, 2, &
                               MPI_COMM_WORLD, ierr )
        Deallocate( ra, stat=alloc_stat )
        Deallocate( ja, stat=alloc_stat )
```

So the data is read by the master process (zero) and sent to all other processes. Each process (not zero) will receive the data, see below. This data is then used to calculate the dot products.

```
    Else
        If ( myid == i ) Then
            Allocate( ra( offset(i)+1:offset(i+1) ), stat=alloc_stat )
            Allocate( ja( offset(i)+1:offset(i+1) ), stat=alloc_stat )

            ! Receive ra and ja (all rows for this process).
            Call MPI_Recv( ra( offset(myid)+1 : offset(myid+1) ), &
                      offset(myid+1) - offset(myid), MPI_REAL8, 0, 1, &
                      MPI_COMM_WORLD, istat, ierr )
            Call MPI_Recv( ja( offset(myid)+1 : offset(myid+1) ), &
                      offset(myid+1) - offset(myid), MPI_INTEGER, 0, 2, &
                      MPI_COMM_WORLD, istat, ierr )

            ! Calculate dot products.
            Do j = myrow_os + 1, myrow_os + mynrows
                Call smxv( m, n, irun*jobsize + j, ra, ja, b, c, na, &
                  myoffset, offset, lura, luja, readtim, myid, nprocs )
            End Do

            Deallocate( ra, stat=alloc_stat )
            Deallocate( ja, stat=alloc_stat )
        End If
    End If
    Call MPI_Barrier( MPI_COMM_WORLD, ierr )
End Do
```

To be sure everything goes well, a MPI_Barrier() is set at the end of the loop.
The result vector, 'c', is collected on the master node.

```
Call MPI_AllGatherV ( &
       c( irun*jobsize+myrow_os+1 : irun*jobsize+myrow_os+mynrows ), &
       mynrows, MPI_REAL8, c( irun*jobsize + 1 : (irun+1) * jobsize ), &
       nrows, row_os, MPI_REAL8, MPI_COMM_WORLD, ierr )
```

That's it.

4.2 MPI-IO (mod3a 6.0)

The data is generated per row.

```
Do i = myrow_os + 1, myrow_os + mynrows
    Call genraja( m, n, i, na, myoffset, lura, luja, writim )
End Do
```

Here 'myrow_os' and 'mynrows' are dependent on the process that is running, in such a way that the work is as equal as possible split over all processes.
The same strategy is used for the calculation.

```
Do i = myrow_os + 1, myrow_os + mynrows
    Call smxv( m, n, i, b, c, na, myoffset, lura, luja, readtim, nprocs )
End Do
```

In both genraja() and smxv() the special calls MPI_File_read_at() and MPI_File_write_at() are used for I/O.

4.3 Execution of the tests

The programs are executed on an IRIX64 machine with eight CPU's.

For each of the chosen matrix dimensions (m and n) performance tests will be done when running with one process, with two processes... upto eight or nine processes. Each of these tests are repeated five to ten times, giving us a decent standard deviation.

5 Results and discussion

All the time measurements are shown in Appendix A. Uninteresting raw data can be found on this site [5] (the gnumeric spread sheets).

Looking at the MPI-IO results for n=25000, there seems hardly any performance gain at all. Almost all results show an extreme decrease in execution time from seven to eight processes. A possible explanation would be that some regulary used variables, like b, the multiplication vector, are not flushed from de CPU cache above some number of processes, because of the decrease in jobsizes with the increase of the number of processes. However, fixing the number of columns and varying the number of rows, and vice versa, do not change this transition location in nprocs.

Another approach to this curiosity is to only count the measurements for one, two, three, maybe four, and eight (and higher) processes for real measurements, and ignore the rest. We can imagine, for most graphs, a smooth curve. It is possible that it goes wrong from four/five to seven processes, because the operating system is doing other stuff. From eight processes and higher the processes aren't locked anymore to specific processors and because of an intelligent dynamic distribution of processes over the processors by the operation system, the performance is better and more in line with our expectations.

When increasing the number of columns excessively, as done for the graphs in appendix A.3, it is shown that there is, in this specific usage of the program, performance gain when parallelizing. This is already visible when using 500000 columns and it is even more obvious with 20,000,000 columns, even though there is some kind of superlinear performance from three to four processes, for which an explanation would need more tests which is outside the scope of this research. That the parallelization works for this kind of row size was to be expected, because the data is processed per row, so the data chunks are large and the MPI overhead is relatively small.

Something else that got my attention is that the processes do not show much CPU usage, so my guess is that the I/O isn't going well and is the likely bottleneck. In future research this program should be tested on different machines and also compared with master-only I/O results. The latter is tried, as seen in the last plot, but the code (mod3a 6.1rc3) doesn't run flawless at all and even gives the wrong results. It could however very well be used as it is programmed rather clean and checked over and over, just like the MPI-IO version (mod3a 6.0rc2).

The last plot (appendix A.4 shows results around 0.6 seconds, while the first graph from A.3 (same matrix size) only shows results significantly above 0.6 seconds (1 second and higher). This hints strongly to a bad performance of the MPI-IO implementation.

6 Conclusion

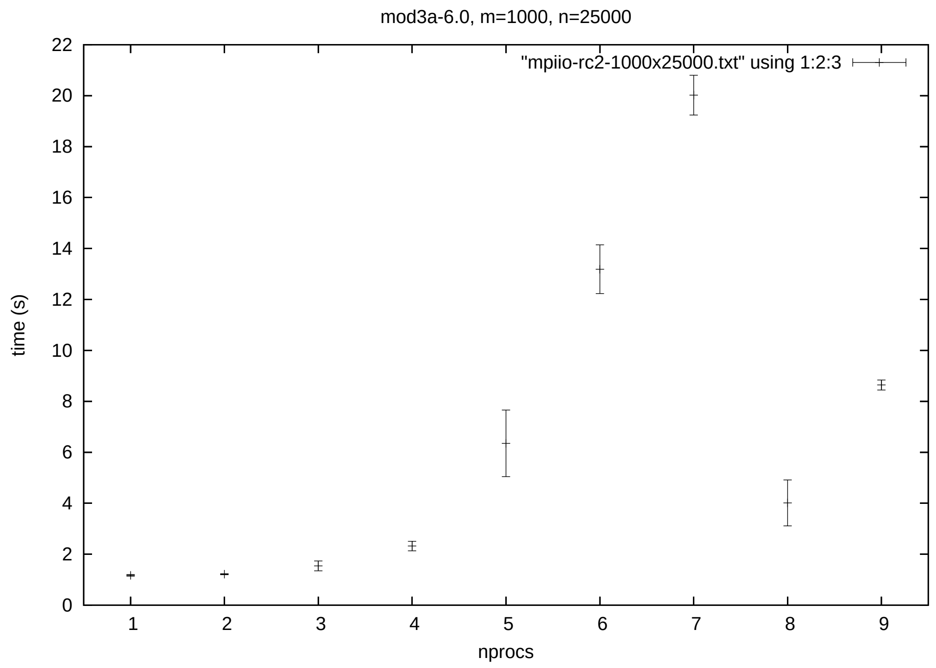
At least for a small number of columns (number of elements per row) the MPI-IO implementation desastrously fails in efficiently parallelizing the sparse matrix-vector multiplication.

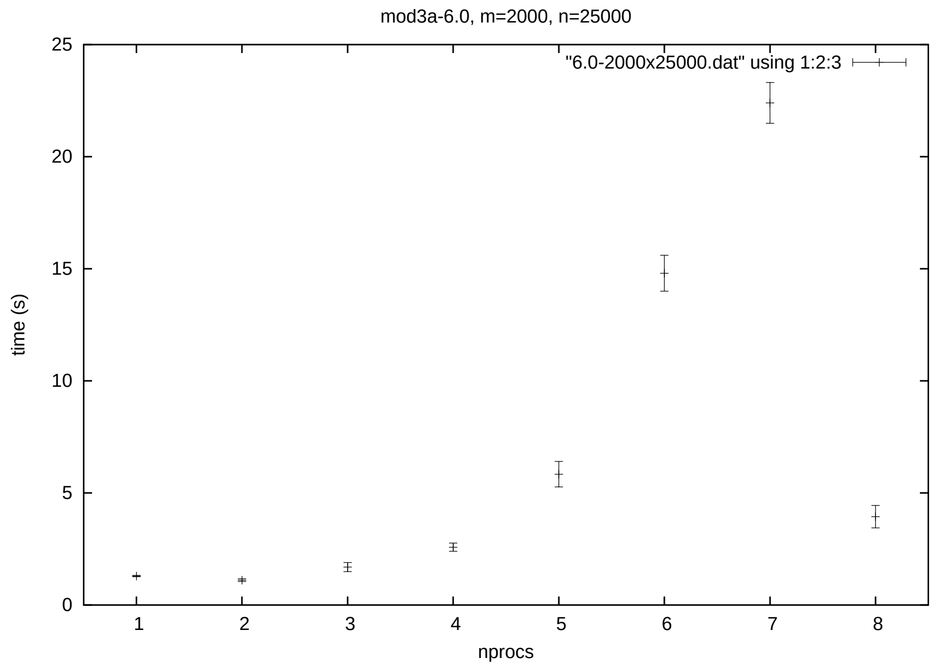
For a large number of columns there seems to be some performance gain, but also some awkward results (superlinear performance gain).

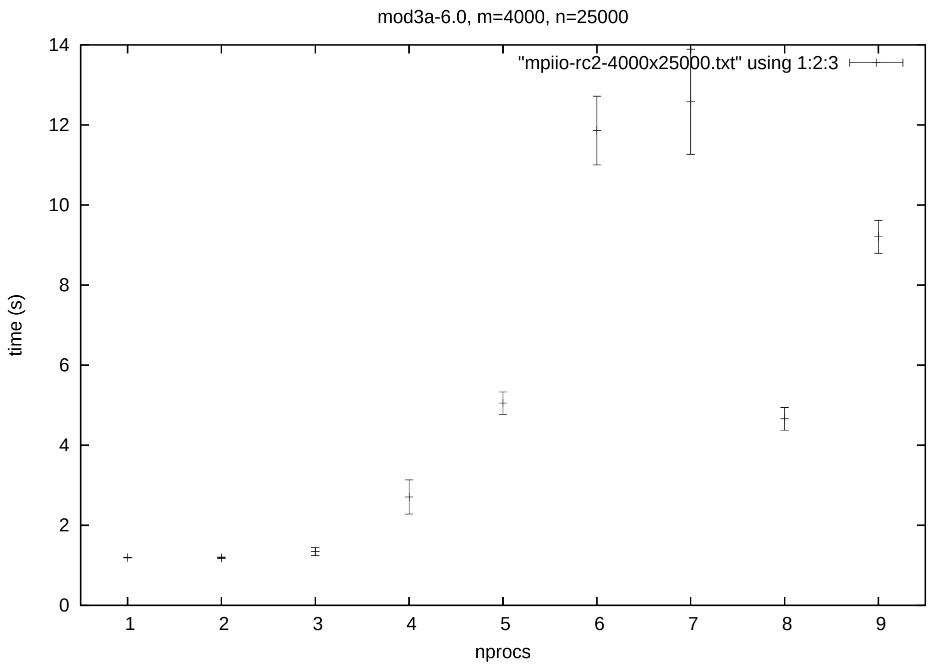
More research is needed before any conclusions can be made about MPI-IO. At the very least the timing results should be compared with timing results of a (working) master-I/O version, structured like mod3a 6.0, as well as runs on other machines.

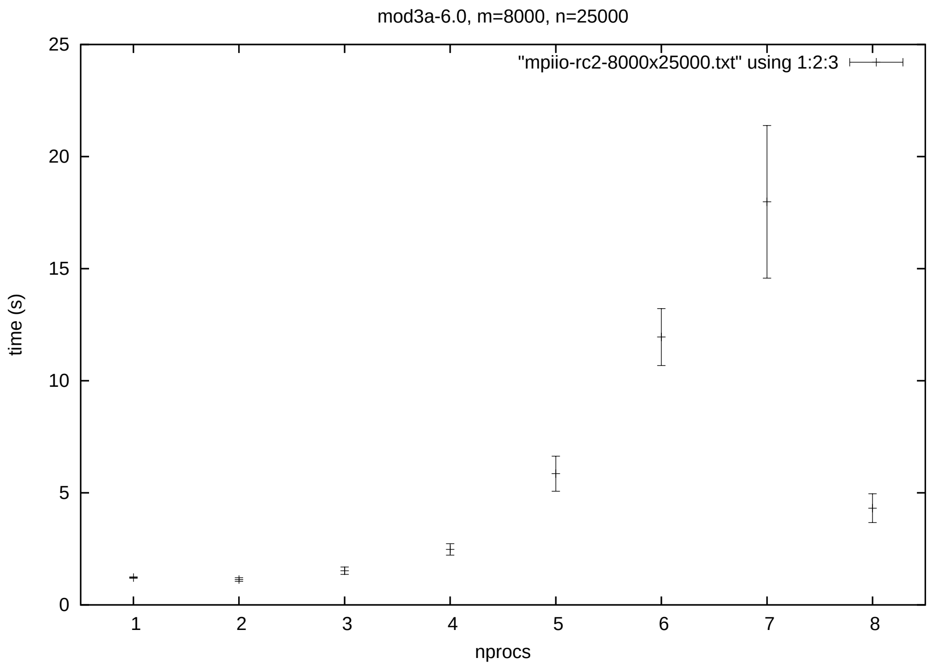
A Result plots

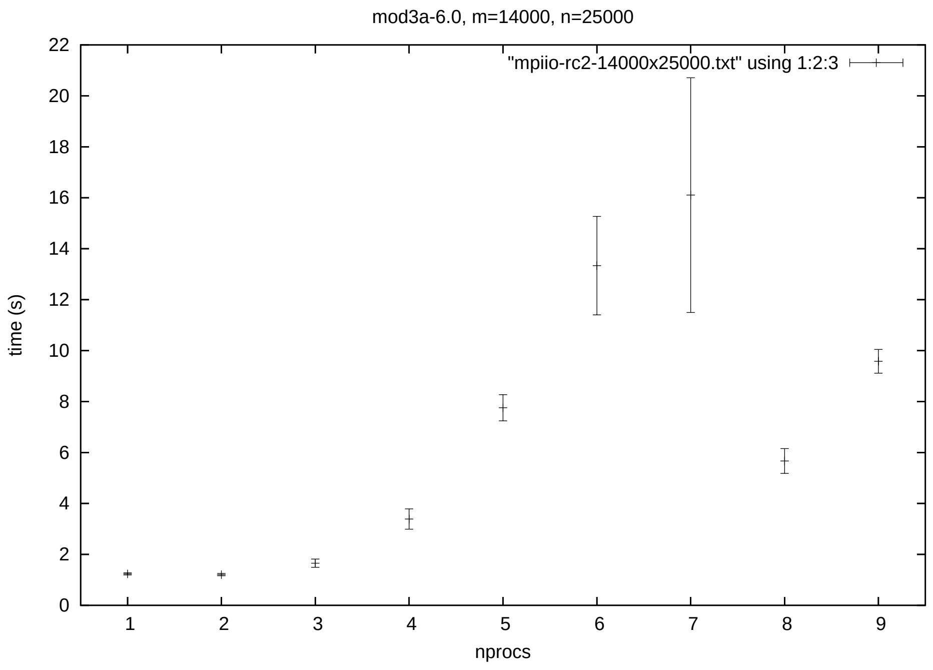
A.1 MPI I/O, n=25000

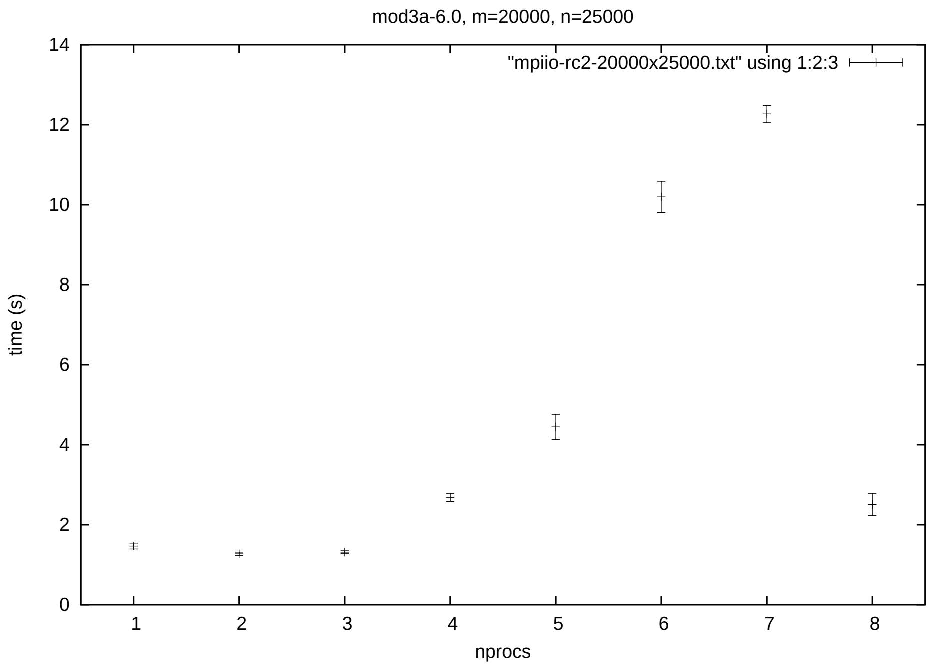


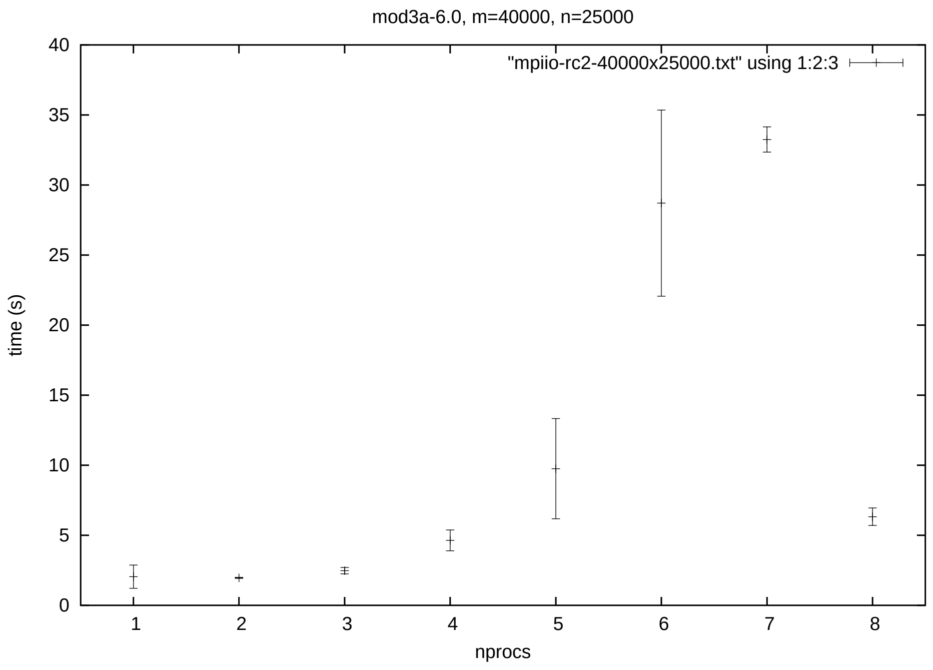


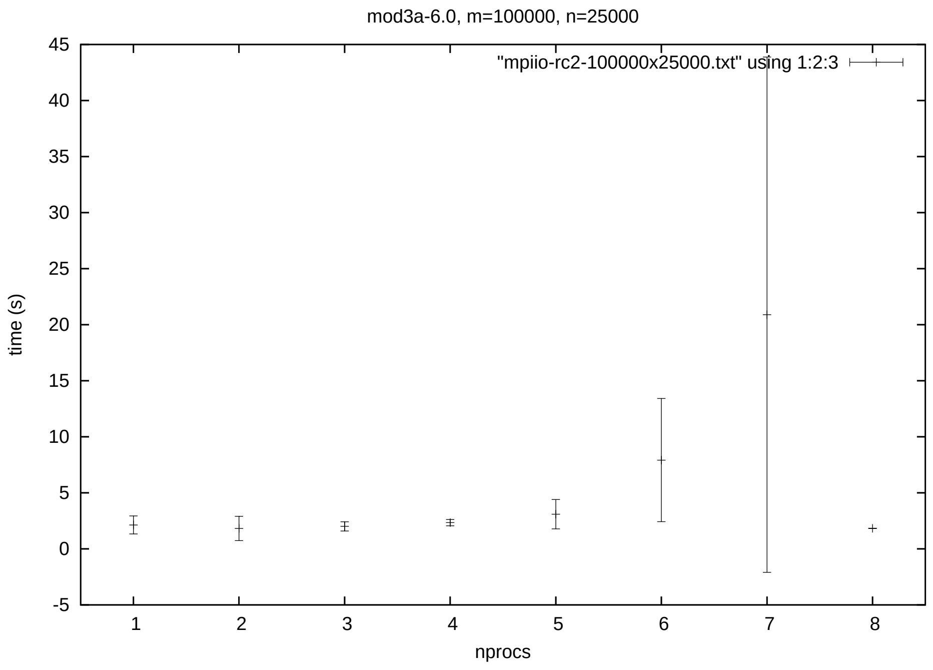

mod3a-6.0, m=4000, n=25000
"mpiio-rc2-4000x25000.txt" using 1:2:3
time (s)
nprocs
0
2
4
6
8
10
12
14
1
2
3
4
5
6
7
8
9


mod3a-6.0, m=8000, n=25000
"mpiio-rc2-8000x25000.txt" using 1:2:3
time (s)
nprocs
0
5
10
15
20
25
1
2
3
4
5
6
7
8

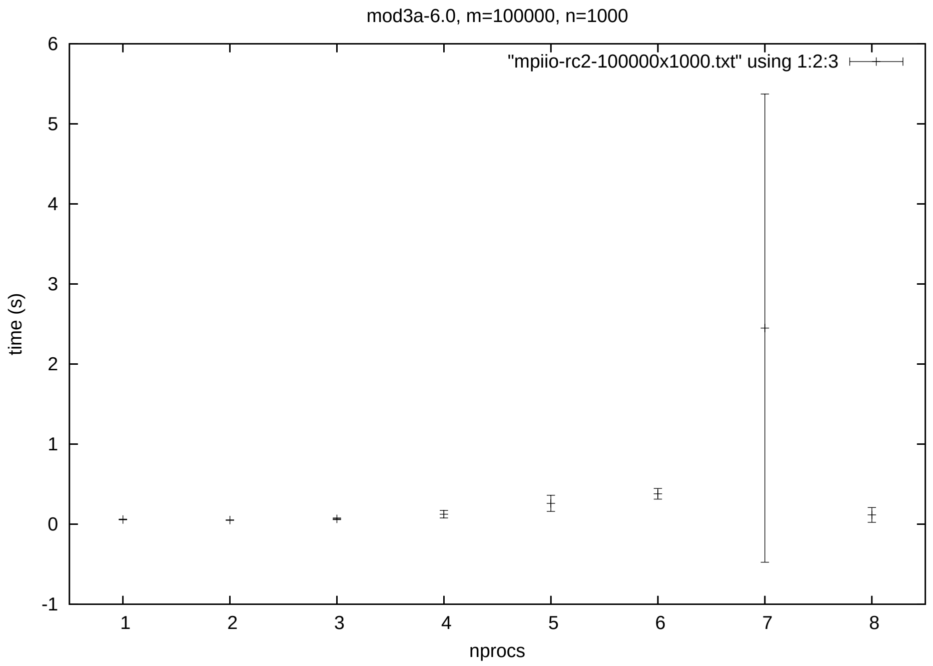

mod3a-6.0, m=14000, n=25000
"mpiio-rc2-14000x25000.txt" using 1:2:3
time (s)
nprocs
0
2
4
6
8
10
12
14
16
18
20
22
1
2
3
4
5
6
7
8
9

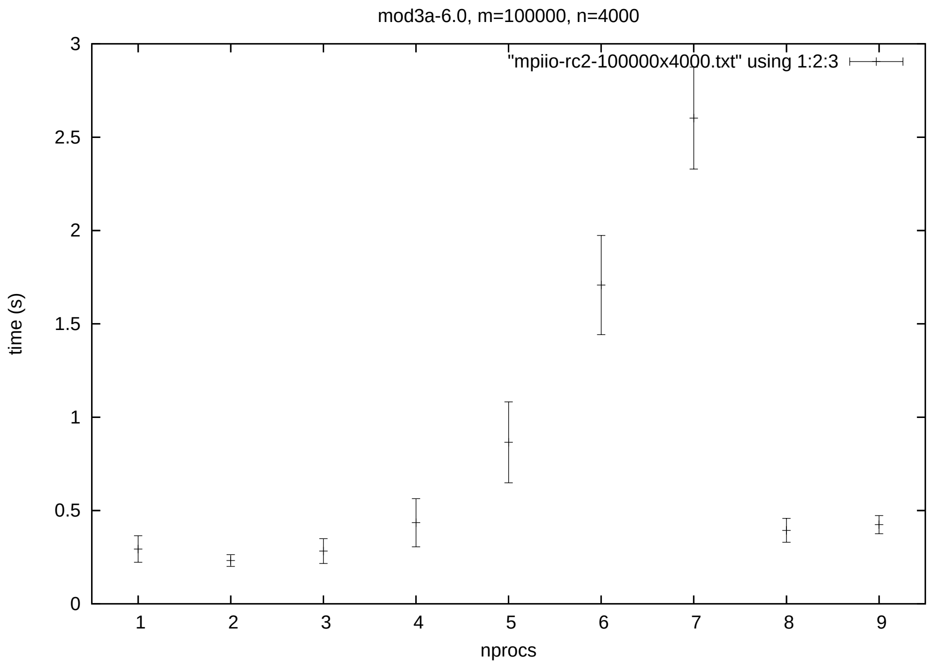

mod3a-6.0, m=20000, n=25000
"mpiio-rc2-20000x25000.txt" using 1:2:3
time (s)
nprocs
0
2
4
6
8
10
12
14
1
2
3
4
5
6
7
8

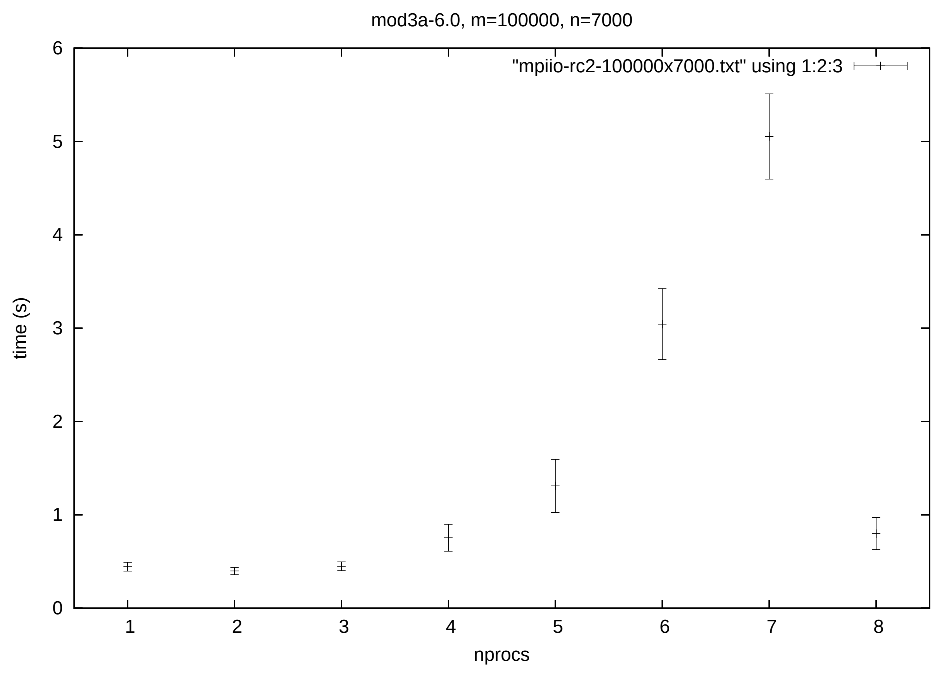

mod3a-6.0, m=40000, n=25000
"mpiio-rc2-40000x25000.txt" using 1:2:3
time (s)
nprocs
0
5
10
15
20
25
30
35
40
1
2
3
4
5
6
7
8

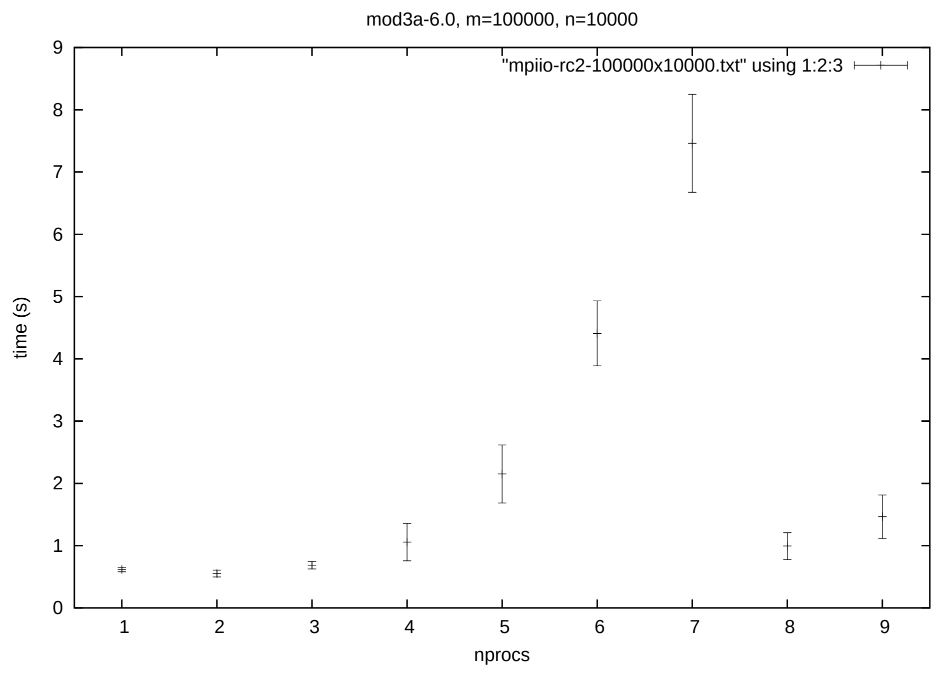

mod3a-6.0, m=100000, n=25000
"mpiio-rc2-100000x25000.txt" using 1:2:3
time (s)
nprocs
-5
0
5
10
15
20
25
30
35
40
45
1
2
3
4
5
6
7
8

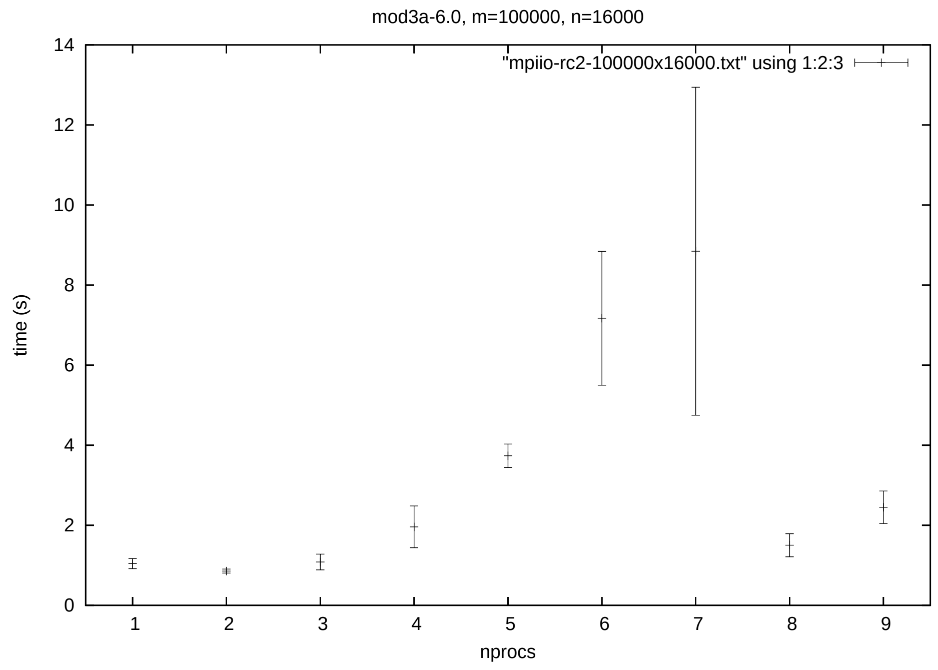
A.2 MPI I/O, m=100000

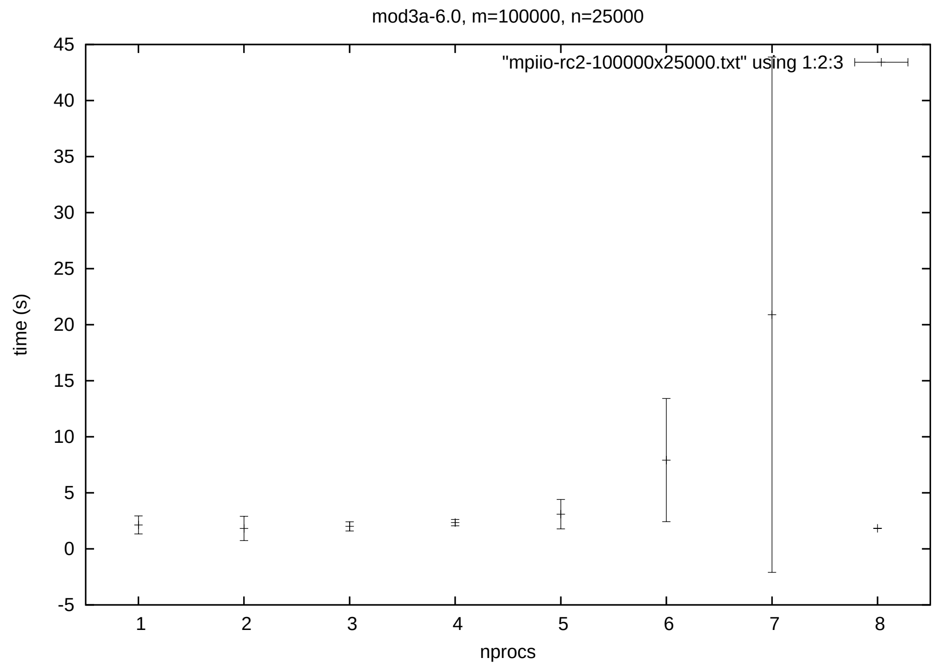




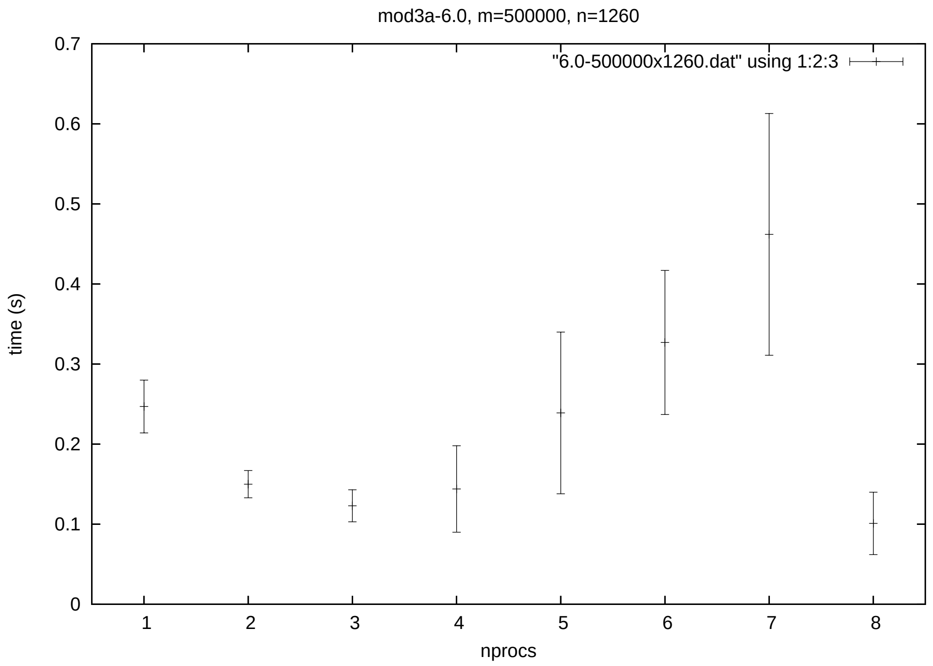

mod3a-6.0, m=100000, n=7000
"mpiio-rc2-100000x7000.txt" using 1:2:3
time (s)
nprocs
0
1
2
3
4
5
6
7
8

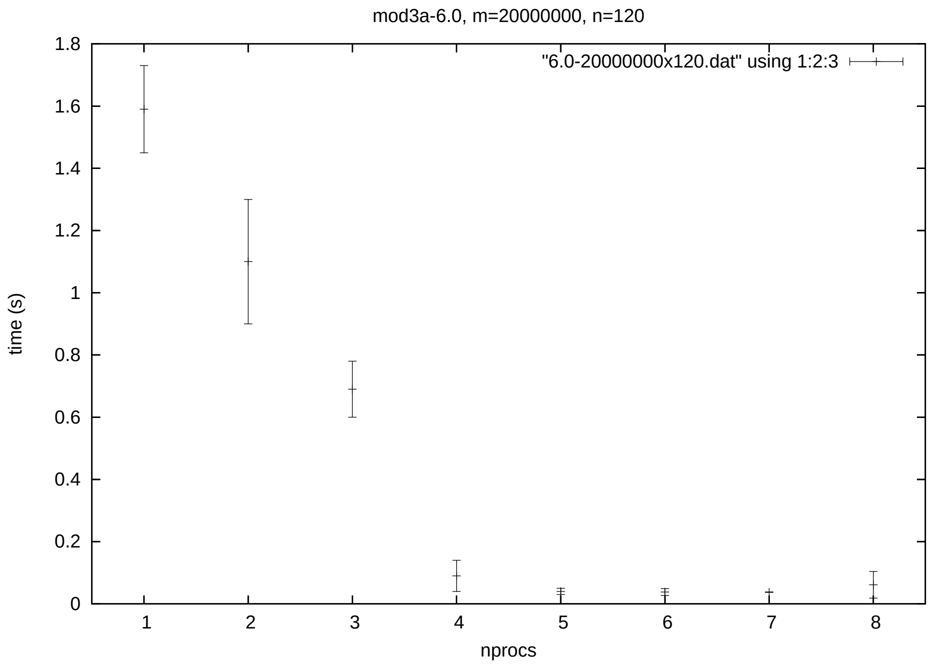

mod3a-6.0, m=100000, n=10000
"mpiio-rc2-100000x10000.txt" using 1:2:3
time (s)
nprocs
0
1
2
3
4
5
6
7
8
9


mod3a-6.0, m=100000, n=16000
"mpiio-rc2-100000x16000.txt" using 1:2:3
time (s)
nprocs
0
2
4
6
8
10
12
14
1
2
3
4
5
6
7
8
9

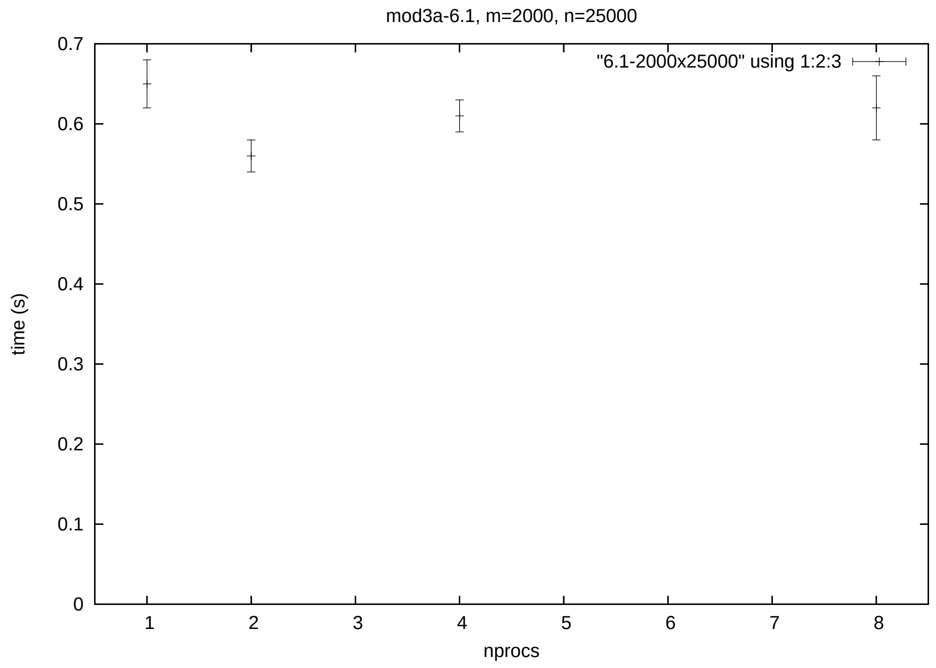

mod3a-6.0, m=100000, n=25000
"mpiio-rc2-100000x25000.txt" using 1:2:3
time (s)
nprocs
-5
0
5
10
15
20
25
30
35
40
45
1
2
3
4
5
6
7
8

A.3 MPI I/O, miscellaneous





A.4 MPI, I/O on master



B Fortran code

B.1 mod3a-6.0 specific source code

```
Program mod3a
! *****************************************************************
! *** This program is part of the EuroBen Benchmark          ***
! *** Copyright: EuroBen Group p/o                           ***
! ***            Utrecht University, Physics Department,     ***
! ***            High Performance Computing Group            ***
! ***            P.O. Box 80.000                             ***
! ***            3508 TA Utrecht                             ***
! ***            The Netherlands                             ***
! ***                                                        ***
! *** Author of the original program: Aad van der Steen      ***
! *** Rewrite by: Marco van Hulten                           ***
! *** Date       January 1995, bug fix May 1997, Fortran 90 version ***
! ***            Spring 1999 (Aad), MPI-2 version Febr 2006 (Marco) ***
! *****************************************************************
!  Version 6.0rc2 -- MPI implementation (with MPI-IO)
!
! ---------------------------------------------------------------------
! MOD3a tests a version of a condensed matrix-vector multiplication.
! The main program drives the subroutine 'smxv' which does the actual
! work. It does a vector update c(n) = A(n,m)*b(m) + c(n).
! A is an (n,m) matrix in condensed form: For each of the 'n' rows the
! number of elements /= 0.0 is held in array na(n) which resides in
! core.
! The column numbers for entries /= 0.0 of A are held in array 'ja' and
! the entries proper in array 'ra'. Both are on disk.
! 'b' is an (m)-vector which is held in core.
! 'c' is an (n)-vector which is held in core.
! ---------------------------------------------------------------------

Use         numerics
Use         mpi
Implicit    None

! Logical units for files
Integer                 :: luin, lura, luja
Character*12            :: filein="mod3a.in"

! Arrays dependent of input parameters
Real(l_), Allocatable :: b(:), c(:), ra(:)
Integer, Allocatable  :: na(:), ja(:)
Integer                 :: m, n, alloc_stat

! Row location and number of rows per process
Integer, Allocatable  :: row_os(:), nrows(:)
Integer                 :: myrow_os, mynrows, i

! Element offset and total number of elements per process
Integer, Allocatable  :: offset(:)
Integer                 :: myoffset, na_max, j

! Needed for checking when EOF luin
Integer( kind=MPI_OFFSET_KIND ) :: iteration, sizeluin
Integer, Parameter              :: ilen = 4

! Variables used in the MPI function calls
Integer              :: myid, nprocs, ierr, stat( MPI_STATUS_SIZE )

! Timing variables
Real(l_)             :: readtim, writim
Real(l_)             :: time_gen, time_cal
```

```
! Function for generating random numbers
Real(l_)           :: dran1

! Other variables for correctness and performance checks
Real(l_)           :: ioread, iowrit, mflops, var
Integer            :: idum, nfill, nflops
Logical            :: allok

Real(l_), Parameter :: zero = 0.0_l_, one = 1.0_l_, two = 2.0_l_, &
                       twenp = 0.2_l_, half = 0.5_l_, micro = 1.0e-6_l_, &
                       nano = 1.0e-9_l_

Integer( kind=MPI_OFFSET_KIND ), Parameter :: nul = 0


! ----------------------------------------------------------------------
! Initialize variables.
! ---
writim = zero
readtim = zero
allok = .TRUE.

! ----------------------------------------------------------------------
! Initialize MPI environment.
! ---
Call MPI_INIT( ierr )
Call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
Call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

Allocate( row_os( nprocs ), STAT=alloc_stat )
Call ErrorCheck( "row_os", alloc_stat )

Allocate( nrows( nprocs ), STAT=alloc_stat )
Call ErrorCheck( "nrows ", alloc_stat )

Allocate( offset( nprocs ), STAT=alloc_stat )
Call ErrorCheck( "offset", alloc_stat )

! ----------------------------------------------------------------------
! Call identification routine for this program and print start of
! output table.
! ---
If ( myid .eq. 0 ) Then
   Call state( 'mod3a' )
   Print 1000
End If

! ----------------------------------------------------------------------
! Open files for input and to hold 'ja' and 'ra' (the matrix).
! ---
Call Input( luin, filein, sizeluin, myid )

Call MPI_File_open( MPI_COMM_WORLD, 'data-ra', MPI_MODE_RDWR + MPI_MODE_CREATE, &
                    MPI_INFO_NULL, lura, ierr )
Call MPI_File_set_view( lura, nul, MPI_REAL8, MPI_REAL8, "native", &
                        MPI_INFO_NULL, ierr )
Call MPI_File_open( MPI_COMM_WORLD, 'data-ja', MPI_MODE_RDWR + MPI_MODE_CREATE, &
                    MPI_INFO_NULL, luja, ierr )
Call MPI_File_set_view( luja, nul, MPI_INTEGER, MPI_INTEGER, "native", &
                        MPI_INFO_NULL, ierr )

! ----------------------------------------------------------------------
! Iterate program, until end of file luin.
! ---
Do iteration = 1, sizeluin, 2*ilen
   ! ----------------------------------------------------------------------
   ! Read input parameters from luin and allocate memory for arrays.
   ! −−−
   Call MPI_File_read_all( luin, m, 1, MPI_INTEGER, stat, ierr )
   Call MPI_File_read_all( luin, n, 1, MPI_INTEGER, stat, ierr )

   Allocate( b(m), stat=alloc_stat )
   If ( alloc_stat .ne. 0 ) Then
      Print*, "Allocation of b failed. Errorcode =", alloc_stat, "; m =", m
      allok = .False.
      Exit
   End If

   Allocate( c(n), stat=alloc_stat )
   If ( alloc_stat .ne. 0 ) Then
      Print*, "Allocation of c failed. Errorcode =", alloc_stat, "; n =", n
      allok = .False.
      Exit
   End If

   Allocate( na(n), stat=alloc_stat )
   If ( alloc_stat .ne. 0 ) Then
      Print*, "Allocation of na failed. Errorcode =", alloc_stat, "; n =", n
      allok = .False.
      Exit
   End If

   ! −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
   ! Now generate for each row the number indicating the columns that
   ! are /= 0.0. The array 'na' holding these numbers is entirely in core
   ! (since version 6.0).
   ! 'na_max' is the size of the biggest row of the matrix.
   ! The filling of the matrix with elements /= 0.0 is about 0.1%
   ! and we choose a variation in the number of row entries of
   ! about 20%.
   ! We count the total number of row entries as 2*Sum(na(i)) is the
   ! number of flops performed in the program.
   ! No advantages of MPI are used here.
   ! This is a dependency for the initialisation of the job distribution
   ! variables (below).
   ! −−−
   nfill   = m/1000
   nflops = 0
   var     = twenp*Real( nfill, l_ )
   idum  = −666
   na_max = 0

   Do i = 1, n
      na(i) = nfill + Int( var*( dran1( idum ) − half ) )
      na_max = Max( na_max, na(i) )
      nflops = nflops + na(i)
   End Do

   Allocate( ra(na_max), stat=alloc_stat )
   Allocate( ja(na_max), stat=alloc_stat )

   nflops = 2*nflops

   ! −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
   ! Initialize job distribution variables.
   ! −−−
   !
   ! Calculate row offsets for all processes, and the number of rows
   ! to do for all processes (first few processes get a row more).
   ! And calculate element offsets for calculation.
   Do i = 1, nprocs
      If ( i <= Mod(n, nprocs) ) Then
         nrows(i) = n/nprocs + 1
      Else
```

```
        nrows(i) = n/nprocs
    End If
End Do
mynrows = nrows(myid+1)

row_os(1) = 0
Do i = 2, nprocs
    row_os(i) = row_os(i−1) + nrows(i−1)
End Do
myrow_os = row_os(myid+1)

offset(1) = 0
Do j = 2, nprocs
    offset(j) = offset(j−1)
    Do i = row_os(j−1) + 1, row_os(j)
        offset(j) = offset(j) + na(i)
    End Do
End Do

! −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
! Generate data for 'b', 'c', 'ja' and 'ra'.
! −−−
!
! Define multiplication vector b. This is done on all processes.
Do i = 1, m
    b(i) = one
End Do

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_gen = MPI_Wtime()

! Generate 'ja' and 'ra'. These arrays are never entirely in core
! and are written per 'mynrows' rows.
idum = −1993 − myid
myoffset = offset(myid+1)
Do i = myrow_os + 1, myrow_os + mynrows
    Call genraja( m, n, i, na, na_max, ra, ja, myoffset, lura, luja, &
                  idum, writim )
End Do

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_gen = MPI_Wtime() − time_gen

! −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
! End of data generation. We now time the matrix multiplication.
! The actual calculation is done in smxv().
! −−−
Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_cal = MPI_Wtime()

! Calculate dot products and send to master node.
myoffset = offset(myid+1)
Do i = myrow_os + 1, myrow_os + mynrows
    Call smxv( m, n, i, b, c, na, na_max, ra, ja, myoffset, lura, luja, &
               readtim )
End Do

Call MPI_AllGatherV( c(myrow_os+1:myrow_os+mynrows), mynrows, MPI_REAL8, &
                     c, nrows, row_os, MPI_REAL8, MPI_COMM_WORLD, ierr )

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_cal = MPI_Wtime() − time_cal

mflops = micro * Max( Real( nflops, l_ )/time_cal, nano )
Print 1010, n, m, time_cal, mflops, ioread, iowrit

! insert correctness check here...(set 'allok' to false if not ok) FIXME!
```

```
    Deallocate( b, stat=alloc_stat )
    if ( alloc_stat .ne. 0 ) then
        Print*, "Deallocation of b failed. Errorcode =", alloc_stat
        allok = .False.
        Stop
    end if

    Deallocate( c, stat=alloc_stat )
    if ( alloc_stat .ne. 0 ) then
        Print*, "Deallocation of c failed. Errorcode =", alloc_stat
        allok = .False.
        Stop
    end if

    Deallocate( na, stat=alloc_stat )
    if ( alloc_stat .ne. 0 ) then
        Print*, "Deallocation of na failed. Errorcode =", alloc_stat
        allok = .False.
        Stop
    end if

END DO

Call MPI_Barrier( MPI_COMM_WORLD, ierr )

If ( myid .eq. 0 ) Then
    Print 1020
    If ( allok ) Print 1040    ! FIXME (all proc's)
End If

! ---------------------------------------------------------------
! Close files and exit MPI environment.
! ---
call MPI_File_close( luin, ierr )
call MPI_File_close( lura, ierr )
call MPI_File_close( luja, ierr )
call MPI_Finalize( ierr )

! ---------------------------------------------------------------
! Formats.
! ---
 1000 Format ( /, " ", 48('-'), &
             '----------------------------', &
             /,' Mod3a: Out-of-core Matrix-vector ', &
             'multiplication',/ &
             74('-'),/ &
             '  Row | Column | Exec. time | Mflop rate |', &
             '  Read rate | Write rate |',/ &
             '  (n)   |    (m) |    (sec)      |   (Mflop/s) |', &
             '    (MB/s) |    (MB/s) |',/ &
             '--------+---------+-------------+-------------+', &
                           '-------------+-------------+' )
 1010 Format ( I7, ' |', I7, ' |', G13.5, '|', G13.5, '|', G13.5, &
             '|', G13.5, '|' )
 1020 Format ( 74('-') )
 1030 Format ( 'Deviation in row ', I7, ' = ', G13.5 )
 1040 Format ( //,' >>> All results were within error bounds <<<' )
! ---------------------------------------------------------------
End Program mod3a
```

```
Subroutine genraja( m, n, i, na, na_max, ra, ja, myoffset, lura, luja, &
                  idum, writim )
! ---------------------------------------------------------------
! Routine 'genraja' generates the relevant parts of the arrays 'ra' and 'ja'.
! The relevant parts of these arrays are written per row to unit 'lura' and
! 'luja', respectively.
! Note that ra and ja are actually na_max long, but only na(i) is used.
! ---

Use        numerics
Implicit   None

Integer :: na_max
Real(l_) :: ra(na_max)
Integer :: ja(na_max)

Integer :: m, n, i, myoffset, lura, luja
Integer :: na(n), idum
Real(l_) :: writim

! Local constants and variables.
Real(l_), External     :: dran1
Real(l_), Parameter    :: one = 1.0_l_
Integer                :: j

! Generate data.
Do j = 1, na(i)
   ra(j) = one
   ja(j) = Min( m, Int( m*dran1( idum ) ) + 1 )
End Do

! Write data to lura and luja.
Call Write_raja( lura, luja, myoffset, ra, ja, na(i), writim )

! Calculate new offset.
myoffset = myoffset + na(i)

End Subroutine genraja
```

```
Subroutine Write_raja( lura, luja, offset, ra, ja, count, writim )

Use    Numerics
Use    mpi
Implicit None

Integer     :: offset, count
Integer     :: lua, ja(count), lura, luja
Real(l_)    :: ra(count), writim

! Local variables and parameters
Integer     :: ierr, stat( MPI_STATUS_SIZE )
Real        :: t0

Integer, Parameter :: dlen = 8, ilen = 4, alen = dlen + ilen
Integer (kind = MPI_OFFSET_KIND) :: foffset_ra, foffset_ja

foffset_ra = offset*dlen
foffset_ja = offset*ilen

t0 = MPI_Wtime()
Call MPI_File_write_at( lura, foffset_ra, ra, count, MPI_REAL8, &
                        stat, ierr )
Call MPI_File_write_at( luja, foffset_ja, ja, count, MPI_INTEGER, &
                        stat, ierr )
writim = writim + MPI_Wtime() - t0

if ( ierr .ne. 0 ) then
   Print*, "Writing to file lua failed! ierr =", ierr
   Stop
end if
```

```fortran
End Subroutine
```

```fortran
Subroutine smxv( m, n, i, b, c, na, na_max, ra, ja, myoffset, lura, luja, readtim )
! ---------------------------------------------------------------------------
! Calculates a chunk of the product c(n) = A(n,m)*b(m) + c(n).
! The length of this part is at most 'lamax' long.
! Routine 'smxv' should be called about n/nprocs times per process
! to cover all 'n' entries of vector 'c'.
! Matrix A is in 'lsqr-format'. 'c' is in core. The number of column
! entries per row is stored in array 'na' which is also in core.
! 'ra' and 'ja' are read from units lura and luja and contain the
! non-zero matrix entries and the column indices where they are
! stored, respectively.
! ---------------------------------------------------------------------------
Use       numerics
Implicit  None

Integer   :: na_max
Real(l_)  :: ra(na_max)
Integer   :: ja(na_max)

Integer   :: m, n, i, lura, luja
Integer   :: myoffset, na(n)
Real(l_)  :: b(m), c(n)
Real(l_)  :: readtim

! Local variables and constants.
Integer               :: j
Real(l_), Parameter   :: zero = 0.0_l_

! Read data from lura and luja.
Call Read_raja( lura, luja, myoffset, ra, ja, na(i), readtim )

! Calculate dot product.
c(i) = zero
Do j = 1, na(i)
   c(i) = c(i) + ra(j)*b(ja(j))
End Do

! Calculate new offset.
myoffset = myoffset + na(i)

End Subroutine smxv
```

```fortran
Subroutine Read_raja( lura, luja, offset, ra, ja, count, readtim )

Use     Numerics
Use     mpi
Implicit None

Integer   :: offset, count
Integer   :: lura, luja, ja(count)
Real(l_)  :: ra(count), readtim

! Local variables and parameters
Integer   :: ierr, stat( MPI_STATUS_SIZE )
Real(l_)  :: t0
Integer, Parameter :: dlen = 8, ilen = 4, alen = dlen + ilen
Integer (kind=MPI_OFFSET_KIND) :: foffset_ra, foffset_ja

foffset_ra = offset*dlen
foffset_ja = offset*ilen

t0 = MPI_Wtime()
```

```
Call MPI_File_read_at( lura, foffset_ra, ra, count, MPI_REAL8, &
                       stat, ierr )
if ( ierr .ne. 0 ) then
   Print*, "Reading from file lura failed! ierr =", ierr
   STOP
end if

Call MPI_File_read_at( luja, foffset_ja, ja, count, MPI_INTEGER, &
                       stat, ierr )
if ( ierr .ne. 0 ) then
   Print*, "Reading from file luja failed! ierr =", ierr
   STOP
end if

readtim = readtim + MPI_Wtime() − t0

End Subroutine
```

B.2 mod3a-6.1 specific source code

```
Program mod3a
! ********************************************************************
! *** This program is part of the EuroBen Benchmark          ***
! *** Copyright: EuroBen Group p/o                           ***
! ***            Utrecht University, Physics Department,     ***
! ***            High Performance Computing Group            ***
! ***            P.O. Box 80.000                             ***
! ***            3508 TA Utrecht                             ***
! ***            The Netherlands                             ***
! ***                                                        ***
! *** Author of the original program: Aad van der Steen      ***
! *** Rewrite by: Marco van Hulten                           ***
! *** Date       January 1995, bug fix May 1997, Fortran 90 version ***
! ***            Spring 1999 (Aad), MPI−2 version Febr 2006 (Marco) ***
! ********************************************************************
!  Version 6.1rc3 −− MPI implementation (without MPI−IO)
!
! −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
! MOD3a tests a version of a condensed matrix−vector multiplication.
! The main program drives the subroutine 'smxv' which does the actual
! work. It does a vector update c(n) = A(n,m)*b(m) + c(n).
! A is an (n,m) matrix in condensed form: For each of the 'n' rows the
! number of elements /= 0.0 is held in array na(n) which resides in
! core.
! The column numbers for entries /= 0.0 of A are held in array 'ja' and
! the entries proper in array 'ra'. Both are on disk.
! 'b' is an (m)−vector which is held in core.
! 'c' is an (n)−vector which is held in core.
! −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

Use       numerics
Use       mpi
Implicit  None

! Logical units for files
Integer, Parameter   :: luin=2, lura=8, luja=10
Character*12         :: filein="mod3a.in"

! Arrays dependent of input parameters
Real(l_), Allocatable :: b(:), c(:), ra(:)
Integer, Allocatable  :: na(:), ja(:)
Integer               :: m, n, alloc_stat

! Row location and number of rows per process
```

```
Integer, Allocatable  :: row_os(:), nrows(:)
Integer               :: myrow_os, mynrows, i

! Element offset and total number of elements per process
Integer, Allocatable  :: offset(:)
Integer               :: myoffset, na_max

! Job divizing variables, only in version 6.1.
Integer             :: jobsize, jobrest
Integer             :: ndb, irun, main_os

! Variables used in the MPI function calls
Integer             :: myid, nprocs, ierr, istat( MPI_STATUS_SIZE ), j

! Timing variables
Real(l_)            :: readtim, writim
Real(l_)            :: time_gen, time_cal

! Function for generating random numbers
Real(l_)            :: dran1

! Other variables for correctness and performance checks
Real(l_)            :: ioread, iowrit, mflops, var
Integer             :: idum, nfill, nflops
Logical             :: allok

Real(l_), Parameter :: zero = 0.0_l_, one = 1.0_l_, two = 2.0_l_, &
                       twenp = 0.2_l_, half = 0.5_l_, micro = 1.0e-6_l_, &
                       nano = 1.0e-9_l_

Integer( kind=MPI_OFFSET_KIND ), Parameter :: nul = 0


! ------------------------------------------------------------------------
! Initialize variables.
! ---
jobsize = 480
writim = zero
readtim = zero
allok = .TRUE.

! ------------------------------------------------------------------------
! Initialize MPI environment.
! ---
Call MPI_INIT( ierr )
Call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
Call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )

If (nprocs == 1) Then
   Print*, "Run this program with at least two processes!"
   allok = .False.
   GoTo 610
End If

Allocate( row_os( nprocs ), STAT=alloc_stat )
Call ErrorCheck( "row_os", alloc_stat )

Allocate( nrows( nprocs ), STAT=alloc_stat )
Call ErrorCheck( "nrows ", alloc_stat )

Allocate( offset( nprocs ), STAT=alloc_stat )
Call ErrorCheck( "offset", alloc_stat )

! ------------------------------------------------------------------------
! Call identification routine for this program and print start of
! output table.
! ---
```

```
If ( myid .eq. 0 ) Then
   Call state( 'mod3a' )
   Print 1000
End If

! ----------------------------------------------------------------------
! Open files for input and to hold 'ja' and 'ra' (the matrix).
! ---
Open( luin, File = filein )
Open( lura, File='data-ra', Form='unformatted', Status='scratch' )
Open( luja, File='data-ja', Form='unformatted', Status='scratch' )

! ----------------------------------------------------------------------
! Iterate program, until end of file luin.
! ---
Do
   ! ----------------------------------------------------------------------
   ! Read input parameters from luin and allocate memory for arrays.
   ! ---
   Read( 2, *, End = 610 ) m, n

   Allocate( b(m), stat=alloc_stat )
   If ( alloc_stat .ne. 0 ) Then
      Print*, "Allocation of b failed. Errorcode =", alloc_stat, "; m =", m
      allok = .False.
      Exit
   End If

   Allocate( c(n), stat=alloc_stat )
   If ( alloc_stat .ne. 0 ) Then
      Print*, "Allocation of c failed. Errorcode =", alloc_stat, "; n =", n
      allok = .False.
      Exit
   End If

   Allocate( na(n), stat=alloc_stat )
   If ( alloc_stat .ne. 0 ) Then
      Print*, "Allocation of na failed. Errorcode =", alloc_stat, "; n =", n
      allok = .False.
      Exit
   End If

   ! ----------------------------------------------------------------------
   ! Now generate for each row the number indicating the columns that
   ! are /= 0.0. The array 'na' holding these numbers is entirely in core
   ! (since version 6.0).
   ! 'na_max' is the size of the biggest row of the matrix.
   ! The filling of the matrix with elements /= 0.0 is about 0.1%
   ! and we choose a variation in the number of row entries of
   ! about 20%.
   ! We count the total number of row entries as 2*Sum(na(i)) is the
   ! number of flops performed in the program.
   ! No advantages of MPI are used here.
   ! This is a dependency for the initialisation of the job distribution
   ! variables (below).
   ! ---
   nfill  = m/1000
   nflops = 0
   var    = twenp*Real( nfill, l_ )
   idum = -666
   na_max = 0

   Do i = 1, n
      na(i) = nfill + Int( var*( dran1( idum ) - half ) )
      na_max = Max( na_max, na(i) )
      nflops = nflops + na(i)
   End Do
```

```
Allocate( ra(na_max), stat=alloc_stat )
Allocate( ja(na_max), stat=alloc_stat )

nflops = 2*nflops

! ---------------------------------------------------------------------
! Generate data for 'b', 'c', 'ja' and 'ra'.
! ---
!
! Define multiplication vector b. This is done on all processes.
Do i = 1, m
   b(i) = one
End Do

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_gen = MPI_Wtime()

! Generate 'ja' and 'ra'. These arrays are never entirely in core
! and are written row by row.
! This is done by the master, for simplicity.
If ( myid .eq. 0 ) Then
   idum = -1993
   Do i = 1, n
      Call genraja( m, n, i, na, na_max, ra, ja, lura, luja, &
                    idum, writim )
   End Do
End If

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_gen = MPI_Wtime() - time_gen

! ---------------------------------------------------------------------
! End of data generation. We now time the matrix multiplication.
! The actual calculation is done in smxv().
! ---
Rewind( lura )
Rewind( luja )

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_cal = MPI_Wtime()

! Divide job in jobsizes of 'jobsize' rows. 'ndb' is the number
! of data blocks.
! This devision is done because otherwise there are problems with
! the number of 'simultaneous' MPI_Send() and MPI_Recv() calls.
ndb = n / jobsize
jobrest = Mod( n, jobsize )

! Outer loop calculation
Do irun = 0, ndb
   If ( irun .eq. ndb ) Then
      If ( jobrest .eq. 0 ) Then
         Exit
      Else
         jobsize = jobrest
      End If
   End If

   ! Initialize job distribution variables for these 'jobsize' rows.
   Do i = 2, nprocs
      If ( i <= Mod( jobsize, nprocs-1 ) ) Then
         nrows(i) = jobsize/(nprocs-1) + 1
      Else
         nrows(i) = jobsize/(nprocs-1)
      End If
   End Do
```

```
    mynrows = nrows(myid+1)

    row_os(2) = 0
    Do i = 3, nprocs
        row_os(i) = row_os(i−1) + nrows(i−1)
    End Do
    myrow_os = row_os(myid+1)

    offset(2) = 0
    Do j = 3, nprocs
        offset(j) = offset(j−1)
        Do i = row_os(j−1) + 1, row_os(j)
            offset(j) = offset(j) + na(i)
        End Do
    End Do
    myoffset = offset(myid+1)

    ! Note that the master process is not used for the calculation.
    !
    ! 'j' will be the process counter, meaning myid will be checked to
    ! this variable. This way a clean Send/Recv structure can be used.
    ! 'i' will be the row counter with respect to to the outer loop
    ! offset (main_os), which gives us row index main_os + i.
    !
    Do j = 1, nprocs−1
        Do i = row_os(j+1) + 1, row_os(j+1) + nrows(j+1)
            If ( myid == 0 ) Then

                ! Read and distribute rows (ra and ja)
                Read( lura ) ra( 1 : na(main_os + i) )
                Read( luja ) ja( 1 : na(main_os + i) )
                Call MPI_Send( ra( 1 : na(main_os + i) ), &
                               na(main_os + i), MPI_REAL8, j, 1, &
                               MPI_COMM_WORLD, ierr )
                Call MPI_Send( ja( 1 : na(main_os + i) ), &
                               na(main_os + i), MPI_INTEGER, j, 2, &
                               MPI_COMM_WORLD, ierr )

            Else If ( myid == j ) Then

                ! Receive ra and ja (all rows for this process).
                Call MPI_Recv( ra( 1 : na(main_os + i) ), &
                               na(main_os + i), MPI_REAL8, 0, 1, &
                               MPI_COMM_WORLD, istat, ierr )
                Call MPI_Recv( ja( 1 : na(main_os + i) ), &
                               na(main_os + i), MPI_INTEGER, 0, 2, &
                               MPI_COMM_WORLD, istat, ierr )

                ! Calculate dot products.
                Call smxv( m, n, main_os + i, b, c, na, na_max, &
                      ra, ja, myoffset, lura, luja, readtim, myid, nprocs )
            End If

        End Do
        Call MPI_Barrier( MPI_COMM_WORLD, ierr )
    End Do

    ! FIXME! −check_bounds geeft hier een probleem.
    Call MPI_AllGatherV ( &
          c( main_os+myrow_os+1 : main_os+myrow_os+mynrows ), &
          mynrows, MPI_REAL8, c( main_os + 1 : main_os + jobsize ), &
          nrows, row_os, MPI_REAL8, MPI_COMM_WORLD, ierr )
End Do

Call MPI_Barrier( MPI_COMM_WORLD, ierr )
time_cal = MPI_Wtime() − time_cal
```

```
    mflops = micro * Max( Real( nflops, l_ )/time_cal, nano )
    Print 1010, n, m, time_cal, mflops, ioread, iowrit

    ! insert correctness check here...(set 'allok' to false if not ok) FIXME!

    Deallocate( b, stat=alloc_stat )
    if ( alloc_stat .ne. 0 ) then
       Print*, "Deallocation of b failed. Errorcode =", alloc_stat
       allok = .False.
       Stop
    end if

    Deallocate( c, stat=alloc_stat )
    if ( alloc_stat .ne. 0 ) then
       Print*, "Deallocation of c failed. Errorcode =", alloc_stat
       allok = .False.
       Stop
    end if

    Deallocate( na, stat=alloc_stat )
    if ( alloc_stat .ne. 0 ) then
       Print*, "Deallocation of na failed. Errorcode =", alloc_stat
       allok = .False.
       Stop
    end if

END DO
610 CONTINUE

Call MPI_Barrier( MPI_COMM_WORLD, ierr )

If ( myid .eq. 0 ) Then
   Print 1020
   If ( allok ) Print 1040   ! FIXME (all proc's)
End If

! ------------------------------------------------------------------
! Close files and exit MPI environment.
! ---
Close( luin )
Close( lura )
Close( luja )
call MPI_Finalize( ierr )

! ------------------------------------------------------------------
! Formats.
! ---
 1000 Format ( /, " ", 48('−'), &
           '−−−−−−−−−−−−−−−−−−−−−−−−−−', &
           /,' Mod3a: Out−of−core Matrix−vector ', &
           'multiplication',/ &
           74('−'),/ &
           '  Row | Column | Exec. time | Mflop rate |', &
           '  Read rate |  Write rate |',/ &
           '  (n)   |    (m) |    (sec)      |    (Mflop/s) |', &
           '     (MB/s) |     (MB/s) |',/ &
           '−−−−−−−−+−−−−−−−−−+−−−−−−−−−−−−−−+−−−−−−−−−−−−−−+', &
                          '−−−−−−−−−−−−−−+−−−−−−−−−−−−−−+' )
 1010 Format ( I7, ' |', I7, ' |', G13.5, '|', G13.5, '|', G13.5, &
           '|', G13.5, '|' )
 1020 Format ( 74('−') )
 1030 Format ( 'Deviation in row ', I7, ' = ', G13.5 )
 1040 Format ( //,' >>> All results were within error bounds <<<' )
! ------------------------------------------------------------------
End Program mod3a
```

```
Subroutine genraja( m, n, i, na, na_max, ra, ja, lura, luja, &
                    idum, writim )
! ---------------------------------------------------------------------
! Routine 'genraja' generates the relevant parts of the arrays 'ra' and 'ja'.
! The relevant parts of these arrays are written per row to unit 'lura' and
! 'luja', respectively.
! Note that ra and ja are actually na_max long, but only na(i) is used.
! ---

Use        numerics
Use        mpi
Implicit   None

Integer :: na_max
Real(l_) :: ra(na_max)
Integer :: ja(na_max)

Integer :: m, n, i, lura, luja
Integer :: na(n), idum
Real(l_) :: writim

! Local constants and variables.
Real(l_), External     :: dran1
Real(l_), Parameter    :: one = 1.0_l_
Integer                :: j, alloc_stat, t0

! Generate data.
Do j = 1, na(i)
    ra(j) = one
    ja(j) = Min( m, Int( m*dran1( idum ) ) + 1 )
End Do

! Write data to lura and luja.
t0 = MPI_Wtime()

Write( lura ) ra(1 : na(i))
Write( luja ) ja(1 : na(i))

writim = writim + MPI_Wtime() − t0

End Subroutine genraja
```

```
Subroutine smxv( m, n, i, b, c, na, na_max, ra, ja, myoffset, lura, luja, &
                 readtim, myid, nprocs )
Use        numerics
Use        mpi
Implicit   None

Integer    :: na_max
Real(l_)   :: ra( na_max )
Integer    :: ja( na_max )

Integer    :: m, n, i, lura, luja
Integer    :: myoffset, na(n)
Integer    :: myid, nprocs
Real(l_)   :: b(m), c(n)
Real(l_)   :: readtim

! Local variables and constants.
Integer                :: j, alloc_stat, ierr, istat( MPI_STATUS_SIZE )
Real(l_), Parameter    :: zero = 0.0_l_

! Calculate dot product.
c(i) = zero
Do j = 1, na(i)
    c(i) = c(i) + ra(j)*b(ja(j))
```

```
End Do

myoffset = myoffset + na(i)

End Subroutine smxv
```

B.3 mod3a-6.x generic source code

```
      Subroutine state(prgnam)
!-----------------------------------------------------------------------
!- This subroutine prints some information about the testing
!  circumstances and the name of the calling module.
!
!  Parameters
!  ----------
!
!  modnam - Character string that represents the name of the calling
!           module.
!
!  Authors: Aad van der Steen
!  Date   : September 1997.
!-----------------------------------------------------------------------
!
      Implicit None

      Character :: prgnam*8, machin*48, memory*48, compil*48, option*48,
     &             os*48, runby*48, comins*48, prec*48, date*8, time*10

!- Please insert the correct data for the current testing circumstances:

!                   123456789 123456789 123456789 123456789 12345678

      Data machin / 'IP27 mips                                       '/
      Data memory / '826 MiB                                         '/
      Data compil / 'MIPSpro Compilers: Version 7.30 (f90)          '/
      Data option / '-O3 -lmpi                                       '/
      Data os     / 'IRIX64                                          '/
      Data prec   / '\64-bits precision                             '/
      Data runby  / 'M.M.P. van Hulten                               '/
      Data comins / 'Utrecht University                              '/
!-----------------------------------------------------------------------
! --- Number of bits in floating-point representation.
      Write( prec(1:3), '(i3)' ) 8*8
      Print 9010, prgnam, machin, memory, compil, option, os,
     &             prec, runby, comins

! --- Report Date and time of calling.

      Call date_and_time( date, time )
      Print 9020, date(7:8), date(5:6), date(1:4),
     &             time(1:2), time(3:4), time(5:10)
!-----------------------------------------------------------------------
 9010 Format( ' EuroBen single-CPU benchmark V4.2, program ',A8/
     &        1X, 75('-')/
     &        ' Testing circumstances:'/
     &        ' Computer                 ', A48/
     &        ' Memory size              ', A48/
     &        ' Compiler version         ', A48/
     &        ' Compiler options         ', A48/
     &        ' Operating System version ', A48/
     &        ' Working precision        ', A48/
     &        ' Run by                   ', A48/
     &        ' Company/Institute        ', A48/ )
 9020 Format( ' Day: ', A2,
     &       3X, 'Month: ', A3,
     &       3X, 'Year: ', A4/
     &       ' It is now ', A2, ' hours, ', A2, ' minutes and ', A2,
     &       ' seconds'/
     &       1X, 75('-') )
!-----------------------------------------------------------------------
      End Subroutine state
```

```
Subroutine Input( lu, filename, size, myid )

Use mpi
Implicit None

Integer                          :: lu, myid
Integer( kind=MPI_OFFSET_KIND ) :: size
Character*11                     :: filename

! Local variables
Integer     :: m, n, ierr, stat( MPI_STATUS_SIZE )
Logical     :: existing, putin

INQUIRE( FILE = filename, EXIST = existing )

if ( .not. (existing) ) then
   if ( myid .eq. 0 ) then
      Print*, "Input file does not exist!"
      putin = .True.
      Do While ( putin )
         Print*, "Please enter dimensions of the matrix (m n)."
         Read( *, * ) m, n
         Print*, "Do you want to enter more input (T, F)?"
         Read( *, * ) putin
      End Do
   end if
   Call MPI_File_open( MPI_COMM_WORLD, 'mod3a.in', &
               MPI_MODE_RDWR + MPI_MODE_CREATE, MPI_INFO_NULL, lu, ierr )
   if ( myid .eq. 0 ) then
      Call MPI_File_write( lu, m, 1, MPI_INTEGER, stat, ierr )
      Call MPI_File_write( lu, n, 1, MPI_INTEGER, stat, ierr )
   end if
   Call MPI_File_close( lu, ierr )
else
   Print*, "Input file exists, continuing."
end if

Call MPI_File_open( MPI_COMM_WORLD, "mod3a.in", MPI_MODE_RDONLY, &
               MPI_INFO_NULL, lu, ierr )

Call MPI_Barrier( lu, ierr )
Call MPI_File_get_size( lu, size, ierr )

End Subroutine Input
```

```
Subroutine ErrorCheck( varname, errcode )

Character*6 :: varname
Integer     :: errcode

If ( errcode .ne. 0 ) Then
   Print*, "Allocation of ", varname, " failed. Errorcode =", errcode
   Call MPI_Finalize( ierr )
   If ( ierr .ne. 0 ) Then
      Print*, "MPI_Finalize error:", ierr
```

```
    End If
    Stop
End If

End Subroutine ErrorCheck
```

```
      Function dran1( idum )                    Result( ran )
      Use        numerics
      Implicit None

      Integer :: idum
! ----------------------------------------------------------------------
! --- dran1 returns a uniform deviate in (0,1).
!
! --- The algorithm is taken from Press & Teukolsky et.al. and
!     based on the linear congruential method with choices for
!     M, IA, and IC that are given by D. Knuth in "Semi-numerical
!     algorithms.
!
! --- Input-parameters:
!     Integer - idum. When idum < 0 the sequence of random values
!                    When idum >= 0, DRAN1 returns the next value
!                    in the sequence. When DRAN1 is called for
!                    the first time it is also initialised.
!
! --- Output-parameters:
!     Integer - idum. Next value of seed as produced by DRAN1.
!     Real(l_) - ran. Uniform deviate in (0,1)
! ----------------------------------------------------------------------
!
      Real(l_)          :: ran, r(97)
      Integer           :: iff, ix1, ix2, ix3, j
! ----------------------------------------------------------------------
! --- Definitions of the three linear congruences used in generating
!     the random number.

      Integer, Parameter :: m1 = 259200, ia1 = 7141, ic1 = 54773,
     &                      m2 = 134456, ia2 = 8121, ic2 = 28411,
     &                      m3 = 243000, ia3 = 4561, ic3 = 51349
      Real(l_), Parameter :: one = 1.0_l_, rm1 = one/m1,
     &                      rm2 = one/m2
!
      Save             iff, r, ix1, ix2, ix3
      Data             iff/0/
! ----------------------------------------------------------------------
! --- (Re)initialise if required.

      If( idum < 0 .OR. iff == 0 ) Then
        iff = 1
! ----------------------------------------------------------------------
! --- Seed first generator.

        ix1 = Mod( ic1 - idum, m1 )
        ix1 = Mod( ia1*ix1 + ic1, m1 )
! ----------------------------------------------------------------------
! --- Use it to seed the second generator.

        ix2 = Mod( ix1, m2 )
        ix1 = Mod( ia1*ix1 + ic1, m1 )
! ----------------------------------------------------------------------
! --- Use generator 1 again to seed generator 3.

        ix3 = Mod( ia1*ix1, m3 )
! ----------------------------------------------------------------------
! --- Now fill array with random values, using gen. 2 for the high
!     order bits and gen. 1 for the low order bits.
```

```
        Do j = 1,97
           ix1 = Mod( ia1*ix1 + ic1, m1 )
           ix2 = Mod( ia2*ix2 + ic2, m2 )
           r(j) = ( Real( ix1, l_ ) + Real( ix2, l_ )*rm2 )*rm1
        End Do
        idum = 1
     End If
! ----------------------------------------------------------------------
! --- This section is only reached when no (re)initialisation takes
!     place. A new random number is generated to fill the place of
!     a randomly picked element from array R (the selection of the
!     index is done by gen. 3).

     ix1 = Mod( ia1*ix1 + ic1, m1 )
     ix2 = Mod( ia2*ix2 + ic2, m2 )
     ix3 = Mod( ia3*ix3 + ic3, m3 )
     j    = 1 + (97 + ix3)/m3
     ran = r(j)
     r(j) = ( Real( ix1, l_ ) + Real( ix2, l_ )*rm2 )*rm1
! ----------------------------------------------------------------------
     End Function dran1
```

```
Module numerics
! ----------------------------------------------------------------------
! We define a Real type that presumably has the characteristics
! of 4 and 8-byte IEEE 754 floating-point types.
! (We assume the Integer type to be 'large enough').
!
Integer, Parameter :: s_ = Selected_Real_Kind(6,37)
Integer, Parameter :: l_ = Selected_Real_Kind(15,307)

End Module numerics
```

References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods.

[2] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with the Message Passing Interface*. Massachusetts Instutute of Technology, second edition, 1999.

[3] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 - Advanced Features of the Message Passing Interface*. Massachusetts Institute of Technology, second edition, 1999.

[4] http://www.gnu.org/licenses/gpl.txt.

[5] http://www.phys.uu.nl/∼hulten/mod3a/.