Reference HIRLAM Scalability Optimization Proposal

Rev 1.2

Danish Meteorological Institute

NEC High Performance Computing Europe

Jan Boerhout

December 10, 2003

Abstract

The scalability of Reference HIRLAM is very good, as has been demonstrated on several shared memory platforms. When using a high number of processors, the SHMEM version continues to scale well, but the GC/MPI version in its current form is efficient only up to a relatively small processor count. Several experiments have been carried out on a NEC SX-6 cluster with different MPI communication methods. The best methods and their performance are presented in this report.

Modifications with respect to revision 1.1:

1. The performance table in section 7 contained a number of incorrect (suboptimal) entries (the performance graph was correct).
2. A missing reference number in section 3 has been added.
3. Some remarks have been added concerning recent HGS optimizations.

Table of Contents

1 Introduction

The measurements with the HIRLAM Reference model version 6.1.2 indicate that the parallel efficiency is open to improvement when using a relatively high number of processors. The portions of the time spent in the horizontal diffusion, the Helmholtz solver and the halo swap code sections increase with higher processor counts. The amount of communication between the sub grids appears to increase with the number of processors squared. The data redistribution routines *twod_to_fft*, *fft_to_tri*, *tri_to_fft* and *fft_to_twod* appear to be responsible for the limited parallel efficiency, together with the halo swap code sections in the routines *swap*, *swap_ps*, *slswap*, *swap_uvpqr*, *swap_ustag* and *swap_vstag*.

A joint DMI and NEC HIRLAM scalability project was launched to optimize these code sections. Several approaches are have been evaluated. (A separate DMI/NEC report discusses the source code and performance results for all other methods considered).

The methods yielding the best results are presented in the following sections.

2 Optimization approach

The initial plan was to use the MPI-2 Remote Memory Access (RMA), or one-sided message passing, because the SHMEM version is very efficient on a number of systems. The Cray SHMEM library is also a single-sided message passing library and it is straight-forward to port the SHMEM code sections to RMA.

Unfortunately, the SHMEM code sections are very fine-grain: many small data sections are exchanged between the program’s tasks. On a shared memory system with an efficient SHMEM implementation there is no difference between memory latency of local and remote memory (other than NUMA effects). On a combined shared and distributed memory system, such as a SX-6 cluster, the latency incurred in the communication between the nodes in the model’s fine-grain communication pattern results in noticeably lower efficiency. Though the IXS crossbar interconnect between the SX-6 nodes is very efficient, the vast number of messages makes the approach of replacing all SHMEM put and get calls by MPI put and get calls less suitable than anticipated.

The number of messages can be reduced dramatically by exploiting the HIRLAM sub grid structure in a different way. The direct use of non-blocking MPI routines results in a very efficient implementation, which supports runtime selection between point-to-point and all-to-all MPI-1 communication routines.

In the next sections the sub grid data structure is discussed, followed by an explanation of the new communication methods.

3 Sub grid definitions

In the sections below the grid point data distributions are visualized as blocks with the orientation shown in Fig. 3-1. All HIRLAM computations are done with the data organized in the **TWOD** data distribution, except for the horizontal diffusion and Helmholtz solver, which require the data in distributions referred to as **FFT** and **TRI** as shown in the next sections.

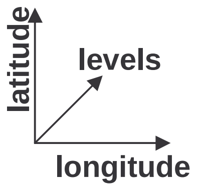

Fig. 3-1 Grid point data orientation in sub grid diagrams

3.1 TWOD data distribution

Fig. 3-2 below shows how the HIRLAM sub grids are defined in the TWOD data distribution and how these are assigned to processes. The numbers on the sub grid fronts represent the MPI process rank (or SHMEM PE) numbers.

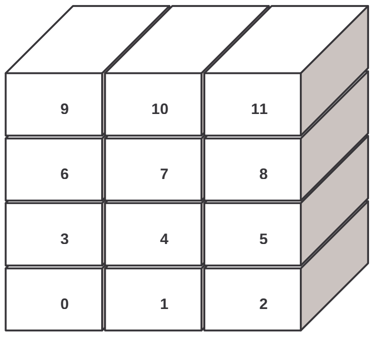

Fig. 3-2 HIRLAM sub grid definition in TWOD data distribution

The TWOD grid point storage order in the FORTRAN arrays is shown in Fig. 3-3. The axis system indicates that the first array index accesses grid points with increasing longitudes; the next index is used for increasing latitudes and the third for increasing levels.

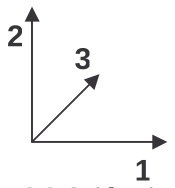

Fig. 3-3 Grid points storage order in TWOD

The processes are numbered 0 to nproc-1. A second numbering scheme is used, consisting of a pair of coordinates indicating the processors position in X and Y direction, where the first is in the range 0 to nprocx-1 and the second in 0 to nprocy-1. The pictures in this document are all drawn with nprocx=3 and nprocy=4. For example, in Fig. 3-2 the TWOD sub grid handled by process 7 has XY process coordinates (1, 2).

Each sub grid has dimensions (klon, klat, klev), where klon and klat vary per sub grid, depending on its location in X and Y.

3.2 FFT data distribution

In order to compute a Fourier transform along a complete latitude line consisting of klon_global grid points, the data is to be redistributed across the processors. The original HIRLAM code redistributes all sub grids' klat x klev full latitudes across all processors, while maintaining a list of *level slabs*: groups of latitudes with the same level.

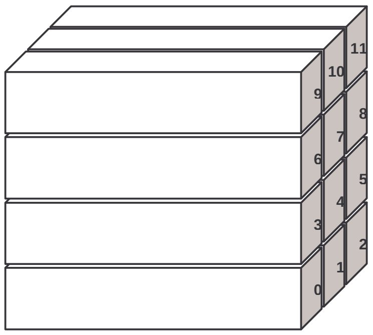

Fig. 3-4 HIRLAM sub grid definition in FFT data distribution

The modified version defines the FFT distribution as shown in Fig. 3-4. The processes in X direction in the FFT distribution hold klat complete latitudes for klev/nprocx levels.

The advantages of the new definition are:

1. No extensive buffering needed to overcome the problem of excessive message count due to the fine-grained data exchange, saving both time and memory
2. The nprocy groups of nprocx processors communicate independently
3. Regular redistribution allows the use of advanced low-latency MPI routines
4. Simpler data administration

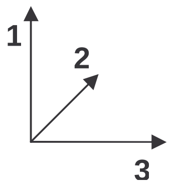

Fig. 3-5 Grid points storage order in FFT

Because in the TWOD distribution the third dimension is klev, it is straight-forward to redistribute the data for nprocx groups of klev/nprocx levels.

A second difference with the original code is the FFT array data storage order, as shown in Fig. 3-5.

This choice has two advantages:

1. In both the horizontal diffusion and the Helmholtz solver this is a more natural choice as vectorization (or software pipelining on scalar processors) is achieved along the grid points with latitudes and levels for a specific longitude.
2. The transposition to the TRI distribution is very efficient and very similar to the transposition from TWOD to FFT distribution.

3.3 TRI data distribution

In the third and last distribution TRI we have full latitude lines with klat_global grid points for filtering in Y direction and solving the tridiagonal system of equations in the Helmholtz solver. Each of the FFT sub grids as shown in Fig. 3-4 is divided in nprocy unique chunks of longitudes. These chunks are exchanged with the nprocy processes handling the sub grids oriented vertically such, that each TRI sub grid contains a set of full latitude lines.

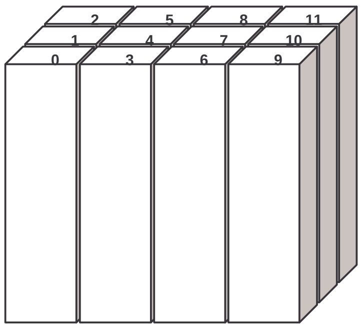

Fig. 3-6 HIRLAM sub grid definition in TRI data distribution

The resulting TRI distribution as presented in Fig. 3-6.

The TRI array storage order is shown in Fig. 3-7. A similar justification holds as for the FFT storage order: data access order and data block redistribution efficiency. Vectorization (or software pipelining on scalar systems) is achieved in the low-pass filter routine *impsub* and solver routines *hhsolv*.

Subroutine *impsub* calls the low-pass filter function lowpass, which even transposes the input data before and after the filtering process. With the new data orientation this extra transposition is no longer necessary.

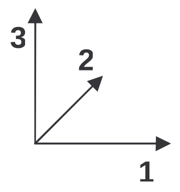

Fig. 3-7 Grid points storage order in FFT

Subroutine *hhsolv* calls subroutine *fft44*, which is already well-suited to handle the data directly in the new orientation. The array indices in the tridiagonal solver have been inverted.

The same structure can be used for the SHMEM version. The performance gain will probably not be as significant as with MPI, but the reduction of the high number of small messages is expected to improve the efficiency somewhat. The additional SHMEM code is quite small and the difference between the two versions minimal.

3.4 Data rearrangement

Before the data is exchanged by message passing (or before, depending on the transpose direction) the data is rearranged from source array to the destination array storage order. This is done by explicit local buffering. Explicit buffering is completely parallelizable with OpenMP, while having the full aggregate local memory bandwidth available.

4 Halo data swap

The evaluation of a sub grid's points requires access to a halo of adjacent points, which are located in the adjacent sub grids. Fig. 4-1 represents a sub grid, where the grid points evaluated are shown in gold. The grid points in the halo around the gold square are copied from the surrounding sub grids, whereas the points at the edge inside the gold square are required by those neighboring sub grids.

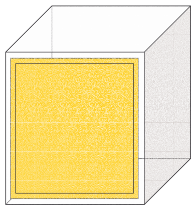

Fig. 4-1 – Grid points evaluated in a sub grid

The halo swap procedures send inner points to the halo zones of the adjacent sub grids and receive such points from those sub grids, as illustrated in Fig. 4-2. The process consists of two stages. First, the halo layers are exchanged with the sub grids at the north and the south. Once this exchange is complete, the procedure is repeated between the sub grids at the east and west. The north-south and east-west exchanges must be sequential; otherwise the grid point at the corners of the sub grid will not be updated correctly. It would be very well possible to execute all swaps simultaneously, but this approach involves explicit exchanges with the sub grids at the north-east, north-west, south-west and south-east. In addition to the need for 2×4 more messages, those messages would be much smaller (klev points each) than the others (roughly klon × klev or klat × klev points). Therefore, the current ordered north-south and east-west exchange is expected to be the most efficient approach.

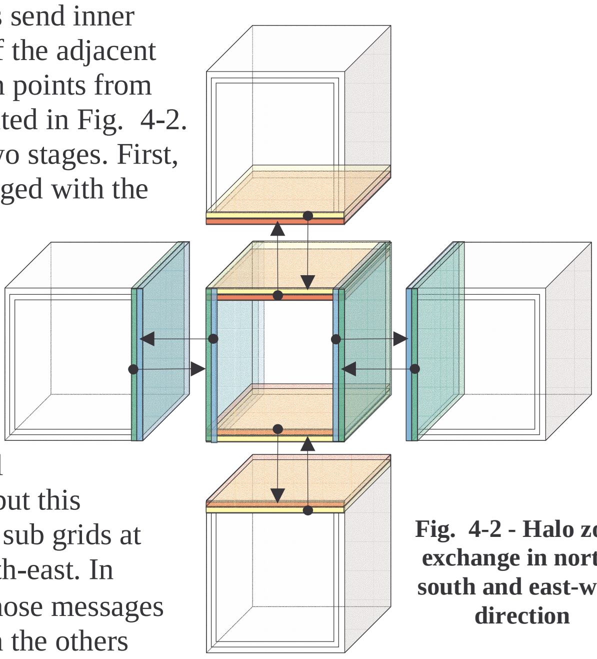

Fig. 4-2 - Halo zone exchange in north-south and east-west direction

There are swap subroutines for several different situations, such as:

- asymmetric halo zone in pressure field (swap_ps)
- single point wide halo zone (swap)
- multiple points wide halo zone (slswap)
- combinations of these two for multiple fields (e.g. swap4)
- halo exchange in one direction for wind fields (e.g. swap_ustag)

The optimizations applied are characterized as follows:

1. Same buffer used for all halo swap routine versions
2. Asynchronous message passing

5 Implementation

The GC library was a wonderful concept at the time when there was no clear winner in the message passing arena. Now that MPI has been well-accepted as the standard, there is no longer a compelling need for a layer in-between. The more powerful MPI features are not supported via the GC library. The SHMEM code is already placed outside of the GC library structure, actually for the same reason: to be able to make use of the efficient SHMEM functionality directly.

Two new MPI communicators are introduced, one which accesses nprocy groups of nprocx processes and one which accesses nprocx groups of nprocy processes. Only the processors in the groups interact in the MPI data transfer routines.

Two different MPI methods have been implemented in the data transposition routines, selectable at runtime by the environment variable MP_METHOD (see table Table 5-1).

Value	**MPI Routine(s) Used**
`ptp` or `point_to_point`	MPI_Isend and MPI_Irecv
`ata` or `all_to_all`	MPI_Alltoallv

Table 5-1 - MPI method selection environment variable MP_METHOD

5.1 Subroutines TWOD_TO_FFT and FFT_TO_TWOD

The purpose of subroutine *twod_to_fft* is to redistribute the input array div such, that each process holds a number of complete longitude lines of klon_global points. Because subroutine *fft_to_twod* performs the inverse of this operation and its code is almost identical (except for the order of the operations and the data transfer direction, of course) its code is considered self-explanatory.

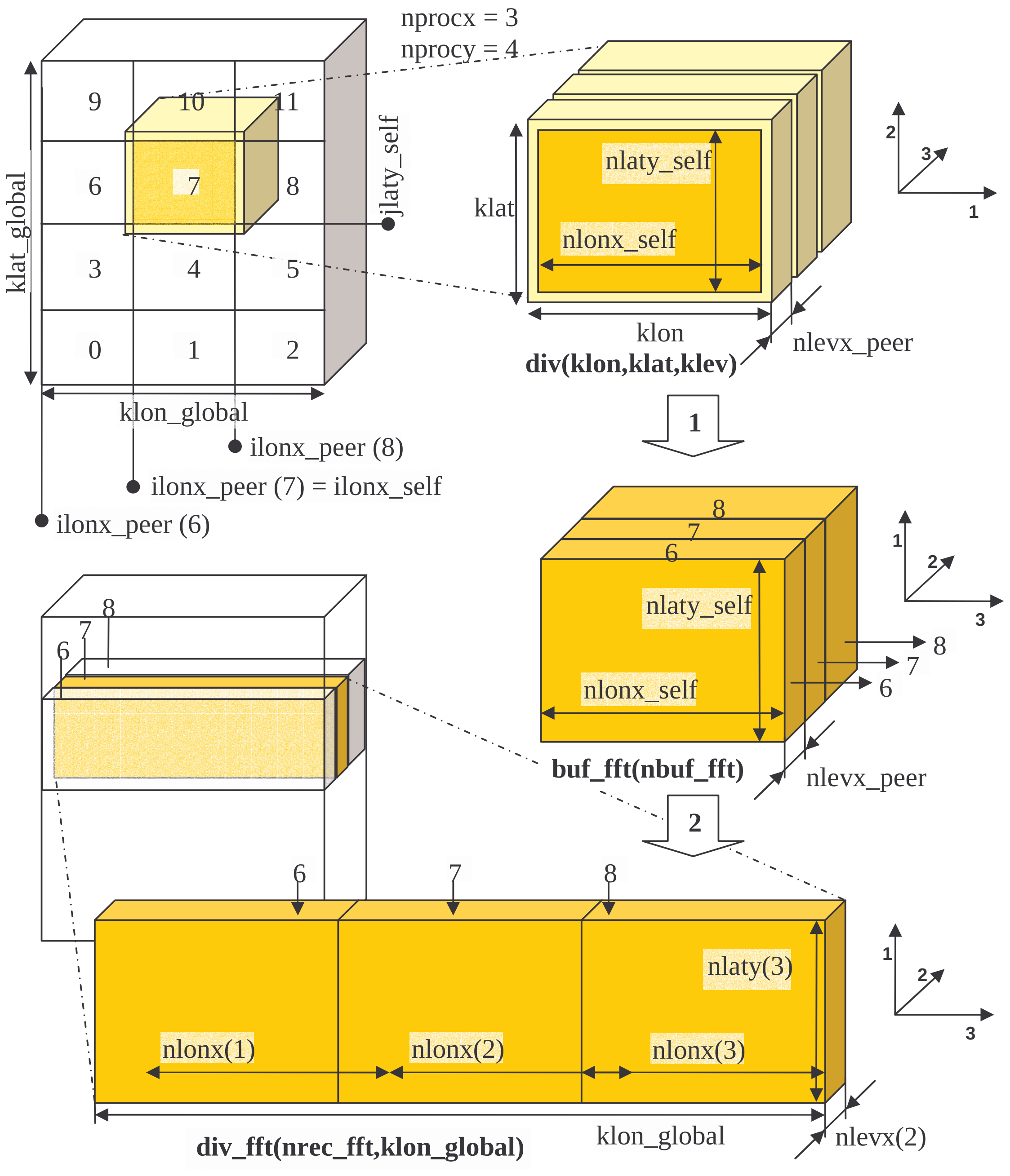

Fig. 5-1 Data flow in subroutine twod_to_fft

Fig. 5-1 gives an overview of the data flow from input array div (distributed as shown earlier in Fig. 3-2) via the intermediate array buf_fft to the output array div_fft (with distribution presented in Fig. 3-4). For practical reasons, a distribution of 3 x 4 sub grids has been chosen. The top left cube represents the global grid divided into the 12 sub grids. Each sub grid is handled by a separate process.

```
if( mp_buffered )then
    nlonx_self = nlonx(i_pe_grid+1)
    nlaty_self = nlaty(j_pe_grid+1)
    do ipe = 1, nprocx
       nlevx_peer = nlevx(ipe)
       imin = offs_fft_inp(ipe)
       jmin = offs_buf_fft(ipe)
       do i = 1, nlonx_self
         do k = 1, nlevx_peer
            jbuf = jmin + nlaty_self*( k-1 + (i-1)*nlevx_peer )
            do j = 1, nlaty_self
              buf_fft(jbuf+j) = div(imin+i,j,k)
            end do
         end do
       end do
    end do
 end if
```

Fig. 5-2 – Data buffering in subroutine `twod_to_fft`

The sub grid handled in process 7 is enlarged to show the distribution across the nprocx(=3) processes 6, 7 and 8. The inner grid points of sub grid 7 are to be redistributed by level groups across these processes. The outline arrow with label 1 in Fig. 5-1 represents the code section in subroutine *twod_to_fft* as listed in Fig. 5-2, responsible for buffering the data before transferring to the peer processes. Array buf_fft is one-dimensional, but should be interpreted as drawn: a set of 3-dimensional arrays, one for each of the “horizontal” processes 6, 7 and 8. The data storage order is chosen such that after the subsequent transfer operation the data structure in div_fft is in the proper orientation. Arrays nlonx, nlaty, nlevx, offs_fft_inp and offs_buf_fft represent dimensions and offsets precalculated in subroutine *decompose_hh*. Array offs_fft_inp contains the data segment offsets in input array div. Array buf_fft consists of nprocx segments, one for each of the “horizontal” processes (6, 7 and 8 in our example), indexed by the offsets in array offs_buf_fft. The numbers of longitudes and latitudes to be sent are the same for each peer process (nlonx_self and nlaty_self), the number of levels (nlevx_peer) may be different if klev is not a multiple of nprocx.

The data from div is copied to buf_fft in the order of increasing latitudes, then increasing levels and finally increasing longitudes. Once the buffer is filled, the second and final operation is the data transfer. One of two code sections is compiled in, controlled by preprocessor macros MPI_SRC and SHMEM. Both sections have been coded in full, because the integral design minimizes the difference between the two versions and optimizes the memory requirements for both versions. (Please note that though the SHMEM code should be in working order, it has not been tested at the time of this report revision 1.1. Comments have been inserted at several locations to remind the user of the status *untested*.)

The MPI code supports two different transfer methods, as discussed in the introduction of this section. The buffer segment offsets and segment sizes are precalculated in subroutine *decompose_hh*.

The outline arrow with label 2 in Fig. 5-1 represents the code sections in subroutine *twod_to_fft* responsible for the data exchange with the peer processes in the group of nprocx “horizontal” processes.

The MPI process rank number, ranging from 0 to nprocx-1 is defined in the context of communicator hl_comm_x. All offsets and sizes are initialized in subroutine *decompose_hh*. (see section 5.4).

Fig. 5-3 shows the **point-to-point message passing** code. The immediate send/receive MPI calls request the data transfers, which will be completed at the time of the return from mpi_waitall.

```
    nreq = 0
    do ipe = 1, nprocx

       call mpi_isend(
+        buf_fft(offs_buf_fft(ipe)+1),size_buf_fft(ipe),rtype,
+        ipe-1, 1000, hl_comm_x, req(nreq+1), ierror )

       call mpi_irecv(
+        div_fft(offs_fft_self(ipe)+1,1),size_div_fft(ipe),
+        rtype,ipe-1,1000, hl_comm_x, req(nreq+2), ierror )

       nreq = nreq + 2
    end do

    call mpi_waitall( nreq, req, stat, ierror )
```

Fig. 5-3 – Point-to-Point Method in twod_to_fft

The **all-to-all message passing** method implemented is presented in Fig. 5-4. The offset and size arrays used for mpi_alltoallv are the same as used in the point-to-point transfer method.

```
    call mpi_alltoallv(
+           buf_fft, size_buf_fft, offs_buf_fft,  rtype,
+           div_fft, size_div_fft, offs_fft_self, rtype,
+           hl_comm_x, ierror )
```

Fig. 5-4 – All-to-All Method in twod_to_fft

The **SHMEM code** is listed in Fig. 5-5. Because SHMEM does not have a very flexible process group mechanism, the process numbers (of PE numbers) are evaluated explicitly. The SHMEM code in subroutine *decompose_hh* sets the buffered bit in mp_method_fft variable, so that buf_fft is constructed in the same way as for the MPI code version.

```
#ifdef SHMEM
      ! WARNING: untested code
      pebase = jpe_grid * nprocx - 1
      do ipe = 1, nprocx
         call shmem_put( div_fft(offs_fft_peer(ipe)+1,1),
     +                   buf_fft(offs_buf_fft (ipe)+1),
     +                   size_buf_fft(ipe), pebase + ipe )
      end do
      call shmem_barrier_all
#endif
```

Fig. 5-5 – SHMEM code in twod_to_fft

At the time of this report version (1.2) the new SHMEM code has been tested by Ole Vignes (NMI). Early results on a SGI Origin system indicate that the new SHMEM code is faster than the original version, but the new MPI code is even faster.

5.2 FFT_TO_TRI and TRI_TO_FFT

Fig. 5-6 presents the data flow for the FFT to TRI transposition for process rank 7. The data in array div_fft is stored in the order: latitude, level and longitude, as shown earlier in Fig. 3-4. The number of inactive points at the beginning and the end of the global longitude range is kpbpts+1. The remaining longitudes are divided across the 4 sub grids, resulting in a distribution (referred to as TRI) of full longitude lines across those sub grids as shown earlier in Fig. 3-6.

The first step is to transpose the data locally (see outline arrow 1 in Fig. 5-6), before the data exchange with the adjacent 4 (nprocy) sub grids. The storage order in the TRI distribution is

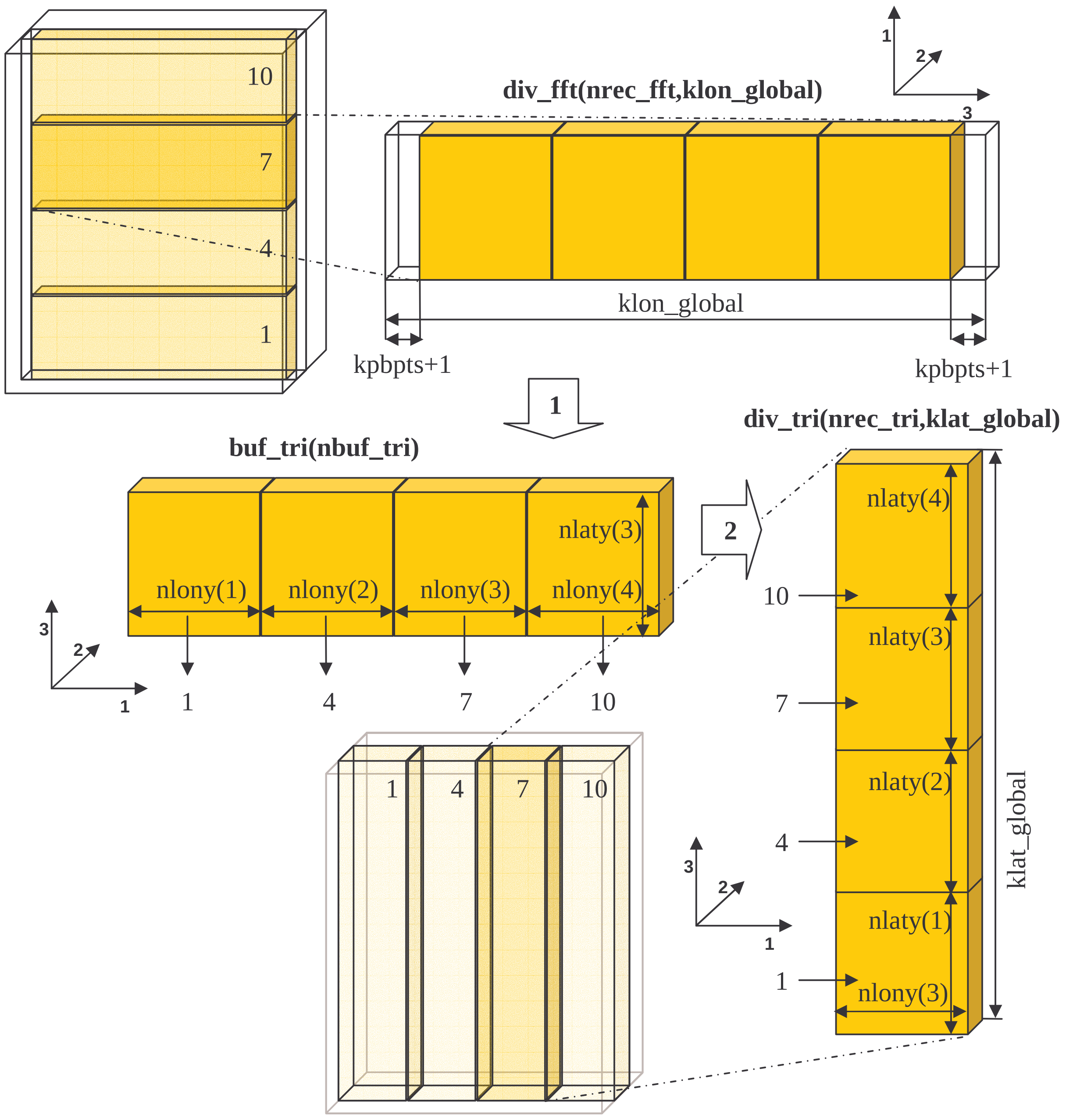

Fig. 5-6 Data flow in subroutine fft_to_tri

longitude, level and latitude. Array buf_tri is used for this purpose, consisting of nprocy segments, one per sub grid, each segment following the same storage order as in the TRI distribution in array div_tri.

The data buffering code as shown in Fig. 5-7 transposes the data along the input array columns. The dimensions, segment sizes and offsets for arrays div_fft and buf_tri are precomputed in subroutine *decompose_hh*. Once the buffer is filled, the segments are exchanged with the 4 “horizontal” sub grids 1, 4, 7 and 10. The buffer segments are received in the corresponding segments of array div_tri.

```
nlaty_self = nlaty(j_pe_grid+1)
nlevx_self = nlevx(i_pe_grid+1)
do jpe = 1, nprocy
   imin = offs_buf_tri(jpe)
   jmin = offs_tri_inp(jpe)
   nlony_peer = nlony(jpe)
   do k = 1, nlevx_self
      jfft = jmin + (k-1) * nlaty_self
      do j = 1, nlaty_self
         ibuf = imin + nlony_peer*( k-1 + (j-1)*nlevx_self )
         do i = 1, nlony_peer
            buf_tri(ibuf+i) = div_fft(jfft+j,i)
         end do
      end do
   end do
end do
```

Fig. 5-7 - Data buffering in subroutine fft_to_tri

The **point-to-point message passing** code (as shown in Fig. 5-8 is very similar to the code shown for subroutine *twod_to_fft*. The loop handles nprocy segments scattered to and gathered from nprocy processes in a group with communicator hl_comm_y. All communication is complete upon return from subroutine *mpi_waitall*.

```
 nreq = 0
 do jpe = 1, nprocy

    call mpi_isend(
+      buf_tri(offs_buf_tri(jpe)+1),size_buf_tri(jpe),rtype,
+      jpe-1, 1000, hl_comm_y, req(nreq+1), ierror )

    call mpi_irecv(
+     div_tri(offs_tri_self(jpe)+1,1),size_div_tri(jpe),rtype,
+     jpe-1,1000, hl_comm_y, req(nreq+2), ierror )

    nreq = nreq + 2

 end do

 call mpi_waitall( nreq, req, stat, ierror )
```

Fig. 5-8 - Point-to-Point Method in fft_to_tri

The **all-to-all message passing** code is listed in Fig. 5-9. Again, this is very similar to the code as shown for the all-to-all code in subroutine *twod_to_fft*.

```
 call mpi_alltoallv(
+        buf_tri, size_buf_tri, offs_buf_tri,  rtype,
+        div_tri, size_div_tri, offs_tri_self, rtype,
+        hl_comm_y, ierror )
```

Fig. 5-9 – All-to-All Method in tri_to_fft

The SHMEM version is shown in Fig. 5-10. Because SHMEM does not have a very flexible process group mechanism, the process numbers (of PE numbers) are evaluated explicitly. The buffer array buf_tri is filled in the same way as for the MPI code version.

```
#ifdef SHMEM
      ! WARNING: untested code

      pebase = ipe_grid
      do jpe = 1, nprocy
         call shmem_put( div_tri(offs_tri_peer(jpe)+1,1),
     +                   buf_tri(offs_buf_tri(jpe)+1),
     +                   size_buf_tri(jpe), pebase )
         pebase = pebase + nprocx
      end do
#endif
```

Fig. 5-10 – SHMEM code in tri_to_fft

Please note that at the time of this report version, the new SHMEM code has not been tested yet.

5.3 Swap Routines

The only difference between the subroutines *swap* and *slswap* is the number of halo points, which is fixed (single point) for *swap* and variable (argument) for *slswap*. Subroutine *swap* and the multi-field versions (*swap2*, etc.) have been replaced by simple wrappers calling the *slswap* versions with the number of halo points set to 1.

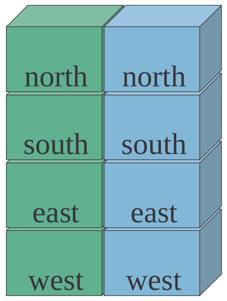

Fig. 5-11 - Swap buffer for send and receive

A common swap buffer structure is defined, large enough for all swap routine variants. The buffer is accessed via an offset array (initialized in *decompose_hh*), which divides the swap buffer in 2x4 segments for sending (green) and receiving (blue) data.

All swap variants pack the data into the swap buffer (see Fig. 5-11) and call the common data exchange routine *swap_buffer*, which is called for both north-south and east-west swap operations (see section 4).

Routine *swap_buffer* supports two MPI methods (RMA and asynchronous Point-to-Point) and SHMEM. All other swap routines are identical, except for some minor details.

The multi-field swap routines *swap[1-6]*, *slswap[1-6]* and *swap_uvpqr[1-3]* are constructed from a template in source file SLSWAP.inc, which is expanded several times in source file SWAP.f for increasing values of preprocessor macro NFIELD. This method keeps the amount of source code to a minimum. In the optimized code more multiple-field variants are defined and used compared to the original code.

Fig. 5-12 lists the buffer fill code section in slswap3 (expanded code generated by the preprocessor). The buffer SEND sections are filled with data to be send to the sub grids at the north and the south.

```
 subroutine slswap3( a, b, c,
+                    klon, klat, klev, khalo )
 …

! North-south data exchange

 if (.not. attop) then
    ndx = offs_swap(SEND,PNORTH)
    do h=klat-1,klat-khalo,-1
       do k=1,klev
          do i=1,klon
             buf_swap(ndx  ) = a(i,h,k)
             buf_swap(ndx+1) = b(i,h,k)
             buf_swap(ndx+2) = c(i,h,k)
             ndx = ndx + 3
          enddo
       enddo
    enddo
 endif

 if (.not. atbase) then
    ndx = offs_swap(SEND,PSOUTH)
    do h=2,1+khalo
       [  same code block as i-k loop above  ]
    enddo
 endif
```

Fig. 5-12 - Send buffer fill code in *slswap3*

Once the north and south buffer sections have been filled, subroutine *swap_buffer* is called, which takes care of the actual message passing work (see Fig. 5-13).

```
len_ns = khalo*klon*klev*3

 call swap_buffer(
$ PNORTH, PSOUTH, attop, atbase, pe_top, pe_base,
$ 1100, 1110, offs_swap, len_ns, len_ns,
$ nreq, request )
```

Fig. 5-13 – Swap buffer call in *slswap3*

Upon return from swap_buffer, the messages may still be underway. The *mpi_waitany* routine is used to wait for the first message transfer to complete. Depending on the request ID, which is related to the message source (in this case north or south), the message is copied from the swap buffer to the top or bottom halo layer (see Fig. 5-14). Cases 2 and 4 represent the receive operations. The send operations (cases 1 and 3) must be waited for, but don't require further action.

In this example, only the MPI version is shown; the only difference in the SHMEM code version is the omission of the *mpi_waitany* call.

```
   do ireq = 1, nreq
      call mpi_waitany( 4, request, idx, status, info )
      select case( idx )
      case( 2 )
         ndx = offs_swap(RECV,PNORTH)
         do h=klat,klat+khalo-1
            do k=1,klev
               do i=1,klon
                  a(i,h,k) = buf_swap(ndx  )
                  b(i,h,k) = buf_swap(ndx+1)
                  c(i,h,k) = buf_swap(ndx+2)
                  ndx = ndx + 3
               enddo
            enddo
         enddo
      case( 4 )
         ndx = offs_swap(RECV,PSOUTH)
         do h=1,2-khalo,-1
            [  same code block as i-k loop above  ]
         enddo
       end select
    end do
```

Fig. 5-14 - Receive buffer usage in *slswap3*

Fig. 5-15 lists the message passing code of *swap_buffer* (MPI version, exchange with south omitted). The buffer offset array has been setup such, that it can be used as both array index and as RMA target offset.

```
  if (.not. atnorth )then
     call mpi_isend(
 $      buf_swap(offset(SEND,dnorth)),len_n,rtype,
 $      pe_north,tag1,hl_comm,request(1), info )
     call mpi_irecv(
 $      buf_swap(offset(RECV,dnorth)),len_s,rtype,
 $      pe_north,tag2,hl_comm,request(2), info )
     nreq = nreq + 2
  end if
```

Fig. 5-15 – Subroutine *swap_buffer* (part, MPI)

Subroutines *swap_ustag* and *swap_vstag* have been modified to handle multiple levels, in line with all other multilevel swap routines. Subroutine *calpqr* has been changed to use the new versions (two loop splits).

5.4 DECOMPOSE_HH

Several array offsets, segment sizes, buffers and MPI derived types are required, depending on the transfer method. These are all precalculated in subroutine *decompose_hh*, based on the data structures shown in Fig. 5-1 and Fig. 5-6. The variable and array names have been chosen carefully to express the nature of the dimensions, offsets and sizes. E.g. the purpose of dimension variables nlonx_self and nlaty_peer is more understandable than of variables named ni and nj.

Please note that the offset and segment size calculations are quite subtle, because one has to be constantly aware of the possibility that the dimensions differ at source and destination of the transfer operation. The essential variables and arrays initialized in subroutine decompose_hh are listed below.

nlonx(nprocx)	number of longitudes inside halo for each of the nprocx sub grids

nlony(nprocy)	number of longitudes for each of the sub grids: klon_global redistributed across nprocy processes
nlaty(nprocy)	number of latitudes inside halo for each of the nprocy sub grids
nlevx(nprocx)	number of levels for each of the sub grids: klev distributed across nprocx processes
nlonx_self	number of longitudes to be handled by this process in a distribution across nprocx processes
nlony_self	number of longitudes to be handled by this process in a distribution across nprocy processes
nlevx_self	number of levels to be handled by this process in a distribution across nprocx processes
nlaty_self	number of latitudes to be handled by this process in a distribution across nprocy processes
****_peer	similar to ****_self, but for peer process
nrec_fft	number of FFT records (“lines” of klon_global longitudes)
nrec_tri	number of TRI records (“lines” of klat_global latitudes)
ilonx_self	longitude index in global grid corresponding with the leftmost non-halo sub grid longitude
jlaty_self	latitude index in global grid corresponding with the bottom non-halo sub grid latitude
buf_fft(nbuf_fft,nprocx)	data transfer buffer for FFT records
buf_tri(nbuf_tri,nprocy)	data transfer buffer for TRI records
buf_swap(nbuf_swap)	data transfer buffer for halo swaps
offs_buf_fft(nprocx)	segment offsets in buf_fft for nprocx processes; first offset is zero, distances between offsets are the same as the block sizes in size_buf_fft
offs_buf_tri(nprocy)	segment offsets in buf_tri for nprocy processes; first offset is zero, distances between offsets are the same as the segment sizes in size_buf_tri
offs_fft_inp(nprocx)	segment offsets in array div (input for twod_to_fft and output from fft_to_twod); the offset of the first segment is the number of halo points before the first non-halo point, distances between the offsets are klon * klat * nlevx(1:nprocx)
offs_tri_inp(nprocy)	segment offsets in input array div_fft (input for fft_to_tri and output from tri_to_fft); first offset is (kpbpts+1)*nrec_fft,

	distances between the offsets are the same as the segment sizes in size_buf_tri
offs_fft_peer(nprocx)	segment offsets in remote array div_fft (one-sided message passing only); offset is ilonx_self * nlaty_self * nlevx_peer
offs_fft_self(nprocx)	segment offsets in local array div_fft; offset is ilonx_peer * nlaty_self * nlevx_self
offs_tri_peer(nprocy)	segment offsets in remote array div_tri (one-sided message passing only); offset is nlony_peer * jlaty_self * nlevx_self
offs_tri_self(nprocy)	segment offsets in local array div_tri; distances between the offsets are the same as the segment sizes in size_div_tri
size_buf_fft(nprocx)	size of buf_fft segments: nlonx_self * nlaty_self * nlevx_peer
size_buf_tri(nprocy)	size of buf_tri segments: nlony_peer * nlaty_self * nlevx_self
size_div_fft(nprocx)	size of div_fft segments: nlonx_peer * nlaty_self * nlevx_self
size_div_tri(nprocy)	size of div_tri segments: nlony_self * nlaty_peer * nlevx_self
hl_comm_x	communicator associated with MPI process group handling a row of sub grids; color is j_pe_grid and key is i_pe_grid
hl_comm_y	communicator associated with MPI process group handling a column of sub grids; color is i_pe_grid and key is j_pe_grid
imin_tri	longitude index in global grid of first longitude index in array div_tri
imax_tri	longitude index in global grid of last longitude index in array div_tri
jlev_tri	level offset in global grid of first level index in arrays div_fft and div_tri
lon_active	active longitudes as required in subroutine impsub
lat_active	active latitudes as required in subroutine impsub
len_ns_max	maximum halo swap buffer segment sizes for north-south halo layer exchange
len_ew_max	maximum halo swap buffer segment sizes for east-west halo layer exchange
offs_swap(2,4)	swap buffer for halo exchanges in subroutines swap, swap_ps, slswap, swap_uvpqr, swap_ustag and swap_vstag
mp_method	message passing method for all transpositions

5.5 Other subroutines

Some minor modifications were needed in other routines, see Table 5-2.

Subroutine	**Modifications**
hhsolv	1. Level slab loop structure replaced by regular loop structure 2. Calls to *fft44* with new array dimensions and reversed index order 3. Reversed index order for arrays div_fft, div_tri and hhdia 4. Array zhhdia renamed to hhdia; allocated dynamically and initialized in *decompose_hh*.
impsub	Reversed index order for arrays div_fft and div_tri *Reminder:* check OpenMP code + directives (seem missing)
difhini	1. Level slab loop structure around array copies replaced by direct assignments 2. Dimensions on CCX and CCY dynamic
sl2tim	1. Sequences of swap routine calls are replaced by the multiple field versions *swap2*, etc. 2. Single-level multiple type arrays (such as tsip and swip) are treated as multilevel arrays with ktyp as the number of levels. This allows the use of multiple field swap versions for these arrays (as mentioned in point 1 above)
bixint, verint	In the original code the vector length is decreased with increasing number of processors; the introduction of an index array preserves the average vector length, resulting in much better performance when many processors are used.
calpqr	Loop splits for new multilevel versions of subroutines *swap_ustag* and *swap_vstag*

Table 5-2 - Modifications in remaining subroutines

5.6 Memory requirements

Though performance is the most important issue, a nice secondary benefit of the new code is the reduction in memory requirements. In the MPI version the data buffers are allocated dynamically.

```
#ifdef SHMEM

! WARNING: untested code

! DECOMPOSE_HH works out the Cray pointers to the individual
! buffers and verifies if dim_shmem_scale is large enough.
!
! Alternative approach: switch to dynamic allocation of symmetric
! memory, but this is not supported by all SHMEM look-alikes.

      integer, parameter ::
     + dim_shmem_scale = 1000,  ! This may be a little too big
     + dim_bufshmem    = klon_global_max * klat_global_max *
     +                   klev_global_max * dim_shmem_scale

      real :: bufshmem( dim_bufshmem )
      common/comshmem/bufshmem

#endif
```

Fig. 5-16 – SHMEM symmetric memory declaration

In the new SHMEM code one big array bufshmem is defined in common block /comshmem/, which should be big enough to cater for all buffer requirements. The size of this array is defined proportional to the global grid dimensions (see Fig. 5-16). Subroutine *decompose_hh* computes the amount of buffer size needed, prints available and required buffer sizes and verifies that array bufshmem is big enough. If not, the program aborts.

Cray pointers are used to define the location of the various message passing buffers. In a SHMEM environment, the use of Cray pointers should not be an issue, because when using SHMEM a Cray environment is implied anyway.

6 Functional Tests

The purpose of the functional tests is to verify the correctness of all new code sections for all decompositions possible on DMI's SX-6 configuration.

A relatively small G40 grid has been used for this purpose (see Table6-1). The tests for the different message passing methods are listed in Table 6-2.

Parameter	Value	Notes
KLON	202	
KLAT	190	
KLEV	40	
NSTOP	6	steps
Initialization	none	
Time step	240	seconds

Table6-1 – Model and execution parameters

Version	2D-FFT	FFT-TRI	Swap
original			
optimized	ptp	ptp	ptp
	ata	ata	ptp

Table 6-2 - Functional tests executed

The asynchronous I/O feature of HIRLAM (HGS) has also been tested with the optimized code version. The functionality has been verified, but performance measurements were not part of the tests.

7 Performance

7.1 Scalability

The runtimes in seconds are presented in Table7-1 for a set of runs, executed on the DMI SX-6 cluster with seven 8-processor nodes and one 4-processor node.

The model and execution details are given in Table 7-2.

Parameter	Value	Notes
KLON	602	
KLAT	568	
KLEV	60	
NSTOP	40	steps
Initialization	none	
Time step	180	seconds

Table 7-2 – Model and execution parameters

				Original			*Optimized*		
CPUs	**XxY**		**ideal**	**Time**	**Ratio**	**Eff**	**Time**	**Ratio**	**Eff**
8	1	8	1,00	254,91	1,00	100%	241,81	1,00	100%
16	2	8	2,00	171,55	1,49	74%	129,69	1,86	93%
24	3	8	3,00	136,49	1,87	62%	91,61	2,64	88%
32	4	8	4,00	105,67	2,41	60%	70,90	3,41	85%
40	5	8	5,00	83,40	3,06	61%	58,09	4,16	83%
48	6	8	6,00	82,46	3,09	52%	49,63	4,87	81%
56	7	8	7,00	83,43	3,06	44%	43,47	5,56	79%
60	6	10	7,50	75,01	3,40	45%	41,23	5,86	78%

Table7-1 – Performance results of original and modified code

No output files are generated. The best message passing method has been used: point-to-point, buffered. The runtimes represent the time spent in steps 0- 40 (`FORECAST TOOK` time subtracted by `PREPARATIONS TOOK` time).

The parallel speedup numbers are presented graphically in Fig. 7-1.

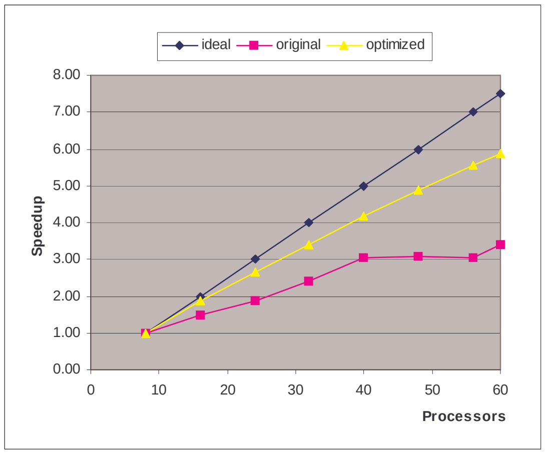

Fig. 7-1 - Parallel speedup improvement

Some observations:

1. The optimized code continues to scale up to the full 60-processor configuration.
2. Straight-forward non-blocking point-to-point message passing gives the best results.

7.2 Full forecast run with initialization and output

One more performance run has been executed with the following characteristics:

1. Grid details as given in Table 7-2
2. 48 hours forecast range
3. Initialization enabled
4. Boundary input every 6 hours
5. Output generated every 3 hours
6. No asynchronous I/O
7. All 60 processors used

This run completed within 27 minutes.

In practice, the model shall produce a 60 hours forecast and output every one hour. The asynchronous I/O feature (HGS) improves the performance further. Another area which has not been explored extensively yet is single-processor optimization, but this has been outside the scope of the assignment (with the exception of *bixint* and *verint*).

Meanwhile, HGS has been optimized further to reduce I/O time to a minimum, allowing hourly boundary input and results output both at a frequency of once per forecast hour, while using only one NEC SX-6 processor as HGS task. Performance measurements with shorter runs indicate that on 56+1 processors a full 60 hours forecast with hourly I/O, initialization enabled and a grid as specified in Table7-1 is expected to run in 24 minutes (based on a 24 hours performance measurement).

8 Summary

A new approach to data transposition yields higher parallel performance with simpler code and more efficient memory use.

The performance improvement measured on 60 SX-6 processors is around 40% relative to the original code, based on the compute-intensive time stepping part of the model (no I/O).

Several different message passing versions have been evaluated. The performance differences are not very significant (1-2%). Non-blocking point-to-point and all-to-all message passing methods give the best performance.

The asynchronous I/O feature of HIRLAM (HGS) works very efficiently with the optimized code version, as recent performance measurements have shown (not discussed in this document).

The optimized HIRLAM program includes a complete SHMEM version, because in the new design it was fairly easy and cleaner to integrate this variety in the new setup. However, verification of the new SHMEM code is not considered part of the DMI-NEC scalability improvement project. Ole Vignes (NMI) has ported this code to the SGI Origin system and early performance measurements show that both the new SHMEM and MPI codes are more efficient than the original SHMEM code.