



HIRLAM GRIBfile Server Optimization

October 15, 2004

NEC HPC Europe – Jan Boerhout

Version 1.1

Table of Contents

1. Introduction	3
2. HGS Rationale.....	3
3. Functionality	4
4. Implementation.....	7
4.1. <i>Brief history</i>	7
4.2. <i>Fortran 95.....</i>	7
4.3. <i>Source code organization.....</i>	7
4.3.1. <i>Module mod_hgs</i>	7
4.3.2. <i>Include files.....</i>	7
4.3.3. <i>Debug prints.....</i>	8
4.4. <i>Detailed functional description.....</i>	8
4.4.1. <i>Initialization</i>	8
4.4.2. <i>HGS command loop.....</i>	8
4.4.3. <i>Gemini time stepping loop.....</i>	9
4.4.4. <i>Model and HGS interaction</i>	9
4.4.4.1. <i>HGS pre-reads input data - hgs_preread_inputfile.....</i>	10
4.4.4.2. <i>Model reads input data - hgs_send_input_data</i>	11
4.4.4.3. <i>Model writes output data - hgs_collect_output.....</i>	13
4.4.4.4. <i>HGS stores output data - hgs_engrib_and_store</i>	15
4.4.5. <i>Multiple HGS tasks.....</i>	16
5. Performance.....	16
6. Known limitations	17
6.1. <i>Namelist input to be used for boundary file pre-read</i>	17
6.2. <i>HGS process memory.....</i>	17
7. Concluding remarks.....	18
8. References	19

1. Introduction

Some acronyms and concepts obvious to the insiders are explained to make the information in this document better accessible for readers outside the meteorological application area. Hopefully, this will not be too annoying for the experts.

The High Resolution Limited Area Model (HIRLAM) is a weather model used operationally by a number of countries united in the HIRLAM consortium (see <http://hirlam.knmi.nl> for more information).

The model uses boundary files produced by a global model at the European Centre for Medium range Weather Forecasts (ECMWF) and produces model output data files. Both input and output files contain the grid point data as fields in bit streams defined in the GRIB file format (WMO standard). The HIRLAM program suite adds a structure to these GRIB files to facilitate accessing the files as standard Fortran *direct access* files.

The HIRLAM forecast model has been parallelized for use on shared and distributed memory computer architectures. The implementation is based on the MPI standard. For more efficient I/O handling, the default MPI task group is split into a model group and an I/O group dedicated to GRIB input and output operations.

The I/O group is referred to as the HIRLAM GRIBfile Server or HGS.

The challenging I/O requirements of the Danish Meteorological Institute (DMI) demand a more efficient buffering scheme to reduce the time needed for the communication between model and HGS.

This document describes how the new HGS at DMI in HIRLAM version 6.2.1 works in terms of functionality and performance. For reasons of better readability, the full functionality is discussed, including the parts created by Ole Vignes, who wrote the MPI version.

2. HGS Rationale

The maximum time allowed for a new forecast is a given (in the order of 20 to 40 minutes). Over time, the use of multiple and faster processors has made it possible to increase the resolution and model complexity. Higher resolution means larger grid sizes and, consequently, more I/O. Additionally, the need for more data updates per time interval, in terms of both input and output, makes the I/O time more visible because of the available system bandwidth, which may be high but is always limited. Whenever the model requires input and/or produces output data, the wall clock time for a complete time step increases to a multiple of the pure computational time.

In principle, there are several possible solutions to this problem:

1. parallelization of the GRIB encode/decode subroutines
2. parallel raw data input and output (non-GRIB format)
3. asynchronous I/O
4. decoupling of model and I/O tasks

The first alternative (parallelization) would require a fine-grain distribution of the encoding and decoding operations. Given the bit-stream record structure, it would be cumbersome to split the input records for distribution across the MPI tasks and to join partial output records

from the MPI tasks. The actual input and output would also have to be parallelized, requiring some kind of parallel filesystem.

The second alternative (raw I/O) would require an additional preprocessing step for GRIB decoding and an additional post processing step for encoding. The disadvantage is a more complex setup, requiring file conversion before and after a forecast run, involving careful scheduling and more disk space.

The third option (asynchronous I/O, from an operating system point of view) solves only part of the problem: the real time-consuming work is the GRIB encoding/decoding part, not the actual I/O.

Handling these encoding/decoding operations asynchronously is what we really need: this is the basis of the fourth alternative, which involves a dedicated set of one or more MPI tasks for I/O – including the encoding and decoding operations.

3. Functionality

Several experiments have been run to produce results in such a form, that the functionality can be visualized using a special viewer. The experiment characteristics are: grid size of 610 x 568 x 40 points, 3 hours forecast, boundary input every hour, model output every hour, time steps of 360 seconds, 15 processors used.

An instructive way to describe how the HGS works is to show a picture of the relevant operations on a time line. The HIRLAM executable used has been instrumented with a facility sampling the complete subroutine call stacks at regular intervals. In Figure 3.1 below, the activity of 2 out of 15 HIRLAM MPI tasks are shown as a trace of call stack samples.

The horizontal axis represents time (see time scale in seconds above the graph) and each

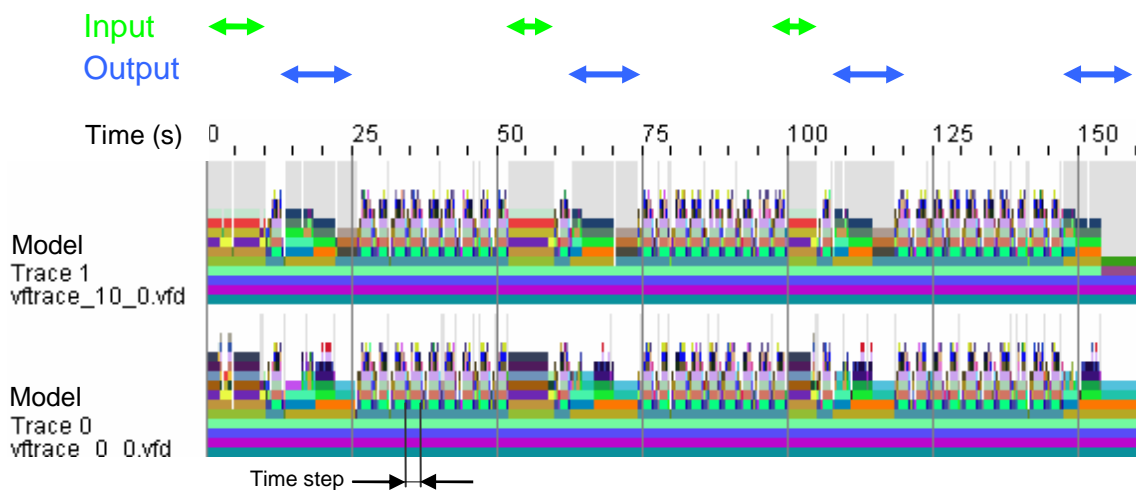


Figure 3.1 – Forecast run (3 hours) with sequential I/O

function called is presented as a colored bar at a vertical position related to the level on the call stack. The time intervals in which I/O is done are indicated by the green and blue arrows on top. In the sections between the I/O intervals the time steps are recognizable as a pattern of small block pairs, representing the dynamics and physics parts, on the 6th level.

It is obvious from this picture how much the performance is dominated by I/O. At the time of e.g. output, each model task sends its local fields to the model task of rank 0, which constructs the global fields, encodes the GRIB records and writes these to the output file. During the

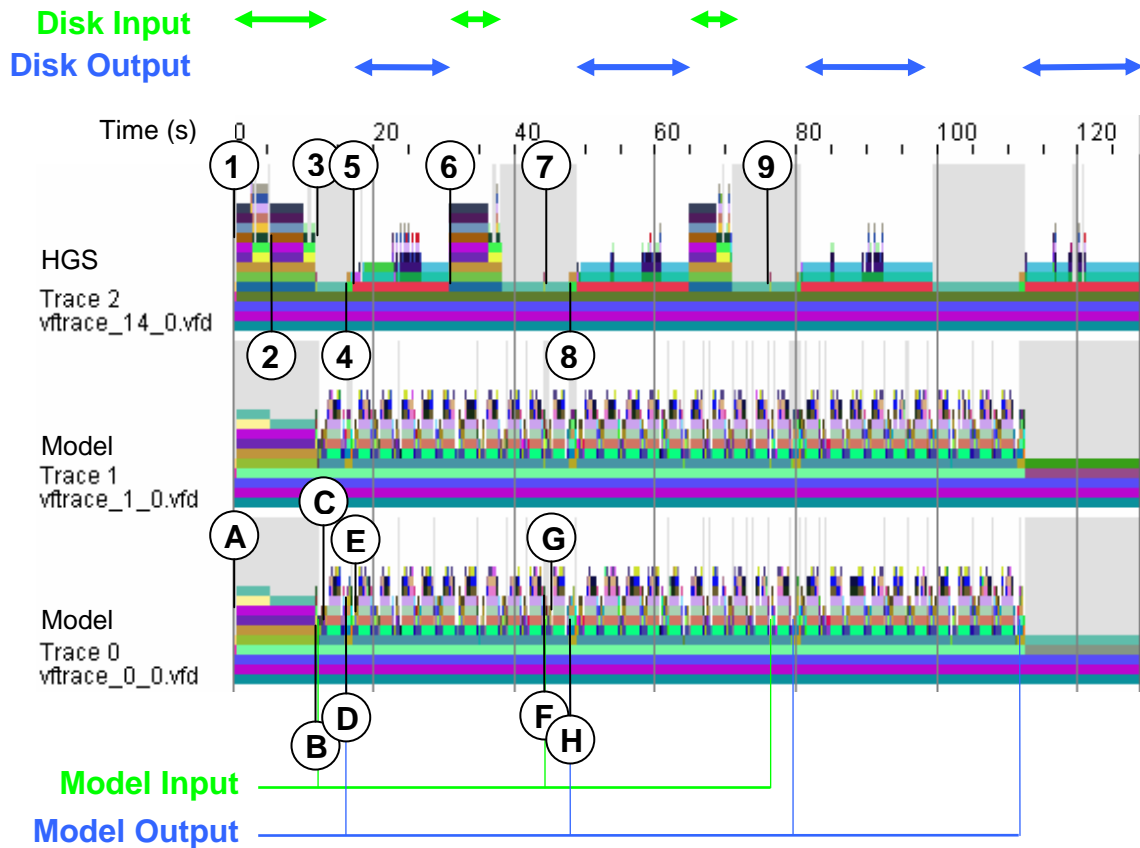


Figure 3.2 – Forecast run (3 hours) with asynchronous I/O

encoding and write operation, all other model tasks wait until the next local field can be sent to task 0.

Figure 3.2 shows how the HGS changes the order of the operations (displaying only 2 of the 14 model traces for practical reasons). The job was repeated on the same number of processors, but this time one of the 15 tasks serves as HGS, leaving 14 tasks for model execution. All other run characteristics are the same.

Referring to Figure 3.2, the time for disk I/O (green and blue arrows) is the same as in the non-HGS run, but the model does not have to wait for these operations to finish anymore. Once the first input files are available, the model starts to run and will no longer have to wait for input. At all times, the input will be available and the time to collect the sub grid input is very short. Likewise, the model saves all sub grid output data simultaneously in a very short transmission burst.

HIRLAM initializes the HGS in subroutine HLPORG if the environment variable NPROC_HGS specifying the number of HGS tasks is set to 1 or more. If not set or 0, HIRLAM processes the I/O sequentially in task 0.

The model and HGS actions are presented in sequence in the following lists. The letters and figures in parentheses refer to the time markers (T_x) as annotated in the trace graph in Figure 3.2. A “+” suffix means that the time of interest is slightly later than the annotated point.

In the runs presented, both input and output take place with the same frequency, but the HGS implementation supports independent schedules for input and output.

The following list gives an overview of the I/O process stages. A more detailed discussion of the major four HGS activities is provided in section 4.4.4 *Model and HGS interaction* on page 9.

Model	HGS
1. Model calls subroutine GEMINI (T_A).	1. HGS enters command loop (T_1).
2. GEMINI sends pre-read request twice to read the first two boundary input files (T_{A+}).	2. HGS receives and handles two pre-read requests: the input files are decoded, the global fields are split into local fields for each model sub grid and stored completely in memory buffers (T_1, T_2).
3. GETDAT is called to receive the first input file, involving several HGS requests in HIRLAM input routines to receive the local fields (T_{A+}). This takes a long time to complete, because the HGS is busy processing the pre-read of the first two files as a whole.	3. Input file requests received, which start the transmission of the input buffer content directly to the model tasks (T_3).
4. Repeat for second input file (T_B). This is very fast, because the HGS has already done all the input and decoding work at this time.	4. Repeat for second input file. Once both input files are read, HGS processes the field requests sent (T_{3+}).
5. Model executes first time step (T_C).	5. HGS waits for next request (T_{3+}).
6. PUTDAT is called, which sends a number of HGS output requests, each followed by the transmission of the output field parameters and data (T_D).	6. Receives output requests, each is processed to collect, merge, decode and store the output fields in a memory buffer (T_4).
7. A store request is sent to let HGS create the output files (T_E).	7. Receives store request to write the output buffers content to the output files (T_5).
8. If it is time for next input pre-read: a HGS request is sent to read the next boundary input file (T_{E+}).	8. Receives and processes a pre-read request (see 2) (T_6).
9. Model proceeds with next time steps (T_{E+}) until the next boundary input file is needed (T_F).	9. HGS waits for the next command.
10. Next boundary input file: model calls GETDAT (T_F)(see 3).	10. Input file open request received (see 3) (T_7).
11. Next time step (T_G).	11. HGS waits for next command.
12. Output due, repeat from 6 (T_H).	12. Receives output requests, repeat from 6 (T_8).

4. Implementation

4.1. Brief history

The first HGS implementation was based on shared memory architecture [3]. It was developed in the context of a benchmark and the time pressure did not permit the design of a more general approach, suitable for both shared and distributed memory architectures.

Around the same time a similar development took place at the Finnish IT Center for Science in cooperation with the Finnish Meteorological Institute, which was based on MPI, which supported asynchronous model output [1]. Boundary file input was still handled sequentially. The Norwegian Meteorological Institute merged the shared-memory HGS version and the CSC/FMI MPI-based approach [2], resulting in a much more portable version.

Recently, the MPI version has been optimized by DMI and NEC to be able to handle a demanding I/O schedule. The main goal was to reduce the communication time to the absolute minimum for both input and output. A second optimization goal was to use only one dedicated HGS task: given the high compute power of a single NEC SX-6 processor, it would be a waste to reserve more processors for the HGS than absolutely necessary.

4.2. Fortran 95

The HGS data collection and distribution mechanism requires the following features:

1. fast and flexible dynamic buffer allocation and deallocation
2. definition of derived types for buffer administration blocks
3. data entities of type *pointer* to create, destroy and link administration blocks and data buffers

These features are supported elegantly in the Fortran 95 standard [4]. Attempts to implement HGS without Fortran 95 features would have resulted in code that is awkward to write, read and maintain.

4.3. Source code organization

The following source files in library grdy contain the HGS code:

hgs.c	Fortran callable C function hgs_getenv_nproc_io_
hgs_ipc.c	IPC version of HGS for shared memory systems (not discussed)
HGS.f	Some common Fortran routines
HGS_MPI.f	MPI version of HGS (all major routines discussed in this section)

4.3.1. Module mod_hgs

All common HGS types and data are defined in module mod_hgs, the first code section in HGS_MPI.f.

4.3.2. Include files

HIRLAM include files are only included in a few special routines, where model data is needed. These routines are

<code>hgs_decomp_info</code>	to retrieve grid decomposition information
<code>hgs_get_ddr</code>	to get a file's data description record (DDR)
<code>hgs_set_ddr</code>	to set a file's DDR
<code>hgs_get_ddr_size</code>	to get the DDR size

4.3.3. Debug prints

Many code sections are instrumented with conditional debug prints, which report the code section's activity preceded by a time stamp with the format *hhmmss.sss*, which makes it possible to verify model and HGS synchronization. The debug messages also serve as code documentation. The debug level is read from environment variable `HGS_DEBUG` and is used to (de)activate specific messages:

- no debug prints (default if `HGS_DEBUG` not defined)
- all HGS subroutine entries and exits plus a few HGS internal activities, such as requests received in the command loop
- more HGS internal activities (buffer management, etc.)

4.4. Detailed functional description

4.4.1. Initialization

The HGS facility is initialized in subroutine *hgs_init*, called from HIRLAM main routine *hlprog*. All MPI tasks execute this call, but only the model tasks return to *hlprog* to participate in the forecast, whereas the remainder of the tasks forms the HGS.

In *hgs_init* the following actions are taken:

- Gets number of HGS processors from routine *hgs_getenv_nproc_io* from environment variable `HGS_NPROC` and decrements `NPROC` by this value, resulting in the number of model tasks.
- MPI communicator for the model *hl_comm* is redefined (originally set to `MPI_COMM_WORLD` in subroutine *gc_init*) for the new `NPROC` value.
- Global grid dimensions
- Various variable initializations.
- The model tasks return to *hlprog* and call HIRLAM routine *decompose* and HGS subroutine *hgs_decomp_info*.
- The HGS tasks also call the same *hgs_decomp_info*. Each HGS task collects the dimensions of all local grids and their positions in the global grid from each model task.
- The HGS tasks enter the command loop, waiting for requests and taking the appropriate action for each request received.

4.4.2. HGS command loop

All further HGS actions are initiated by submission of a specific request for a specific *logical unit number (lun)*. Subroutine *hgs_request*, called by the model tasks, finds the HGS task

associated with the *lun* and sends the request. If the *lun* is not associated with a HGS task yet, one is assigned. The HGS tasks are assigned to *luns* in a round robin fashion.

The requests listed below are issued by the model:

- REQ_GETGRB Request to read a boundary input file, served by subroutine *hgs_preread_inputfile*. There is no interaction with the model except for a short message tagged TAG_INPUT_AVAIL sent when the file has been read, confirming the availability of the input data in HGS memory.
- REQ_GROPEN Request to send the boundary input data pre-read earlier, served by subroutine *hgs_send_input_data*, which starts a short interval of intensive communication with the model to send the input data.
- REQ_PUTGRB Request to collect model output data, served by subroutine *hgs_collect_output*, which starts a short interval of intensive communication with the model to receive the output data. All data is kept in memory until a REQ_STORE is received (see below).
- REQ_STORE Request to write the output data stored in the HGS buffers to disk, served by subroutine *hgs_engrib_and_store*. There is no interaction with the model at all.

One special request is issued by other HGS tasks, if more than one is used:

- REQ_SIGNAL Special request from other HGS task to HGS task rank 0: write a message to the file associated with *lun* 98 (request submitted by subroutine *proces* in library *grw1*), served by subroutine *hgs_write_flush*.

4.4.3. Gemini time stepping loop

The following four I/O calls in the time stepping loop of subroutine *gemini* (in this order, subject to the input and output schedule) interact with the HGS:

- getdat* Read boundary input data.
- putdat* Write model output data.
- hgs_request*(REQ_STORE, ...) Multiple REQ_STORE requests for each NPPSTR post-processing file (*luns* LUNPPF(1:NPPSTR) and the model (*lun* LUHIFI) output file. The HGS task(s) associated with these *luns* create the output files and free the data buffers.
- hgs_request*(REQ_GETGRB, ...) Request issued to pre-read the next boundary file.

The locations of the *hgs_request* calls have been chosen such that implied but unintended synchronization points are avoided in the interaction with the HGS tasks.

4.4.4. Model and HGS interaction

The table below shows the relationships between the events in the *gemini* time stepping loop and the request server routines in the HGS:

Purpose	Model call or request	HGS server routine
HGS pre-reads input data	REQ_GETGRB	<i>hgs_preread_inputfile</i>

Model reads input data	<i>getdat</i>	<i>hgs_send_input_data</i>
Model writes output data	<i>putdat</i>	<i>hgs_collect_output</i>
HGS stores output data	REQ_STORE	<i>hgs_engrib_and_store</i>

In the next four sections discuss the model - HGS interaction for these actions in detail.

4.4.4.1. HGS pre-reads input data - *hgs_preread_inputfile*

A boundary file pre-read request REQ_GETGRB is sent by model subroutine *gemini*. The exact location is important and has been chosen such that inadvertent synchronization with the HGS is avoided. It is important to send the REQ_GETGRB *after* the REQ_STORE request (see section *HGS stores output data - hgs_engrib_and_store*), otherwise the model would stall for the duration of the pre-read.

The HGS handles the pre-read request autonomously in subroutine *hgs_preread_inputfile*.

Subroutine *hgs_getgrb* is called, which in turn calls model subroutine *getgrb* in library *grdy*. This routine is also called via subroutine *getdat* from model control routine *gemini*. The file input routines called by *getgrb* for open, read and close are *groploc*, *grrdloc* and *grclloc*, which in turn call *hgs_gropen*, *hgs_gread* and *hgs_grclos*. The functionality of the latter three depends on the context: model or HGS task. Because this section describes the HGS side of reading the input file, only the HGS functionality is discussed. The model side of the routines *hgs_gropen*, *hgs_gread* and *hgs_grclos* is described in the next section *Model reads input data - hgs_send_input_data*.

Subroutine *hgs_gropen* calls subroutine *gropen* and copies the input file's administration record or Data Description Record (DDR) to the file's administration. The first input buffer is allocated and subroutine *hgs_init_input_buffer* is called to initialize the input buffer's field array for the first INPUT_BLKSIZE (=100) fields.

Subroutine *hgs_gread* (called repeatedly by subroutine *getgrb*) takes care of the following:

1. if the input buffer is full, the next one is allocated and linked to the previous buffer;
2. subroutine *gread* is called, which reads and decodes the global field from the input GRIB file;
3. the field parameters (type, parameter number, level and error code) are saved in the input buffer's header block (up to INPUT_BLKSIZE headers);
4. the global field is interpolated from GRIB to model grid;
5. the global field is decomposed into NPROC local fields, which are stored in the input buffer, sorted by sub grid number up to INPUT_BLKSIZE fields per buffer.

Subroutine *hgs_grclos* calls *grclos* to close the input file and marks the file as available.

Subroutine *hgs_preread_inputfile* calls *hgs_confirm_input_available*, which sends a message to the model, and returns to the HGS command loop. The whole input file is now stored in a number of input buffers, organized in a linked list, each buffer containing INPUT_BLKSIZE fields. The data is sent to the model upon request REQ_GROPEN, as described in the next section.

4.4.4.2. Model reads input data - *hgs_send_input_data*

Model and HGS interact strongly during the input phase. The model in subroutine *gemini* calls subroutine *getdat*, which in turn calls model subroutine *getgrb* in library *grdy*. The file input routines called by *getgrb* for open, read and close are *groploc*, *grrdloc* and *grclloc*, which in turn call *hgs_gropen*, *hgs_gread* and *hgs_grclos*. The functionality of the latter three depends on the context: model or HGS task. The previous section describes the use of these routines when called in the context of a HGS task.

The model reads the input file data, which has been read by the HGS earlier (see previous section). The HGS job is to send these data to the model, while deallocating the buffers from which all data have been sent.

Time marker 9 in Figure 3.2 indicates one of the intervals, in which this input procedure is executed. Because the interval is very short, the interval is presented on a more suitable time scale in Figure 4.1. The model trace shows two activity blocks, associated with receiving input data blocks, starting at time marker A.

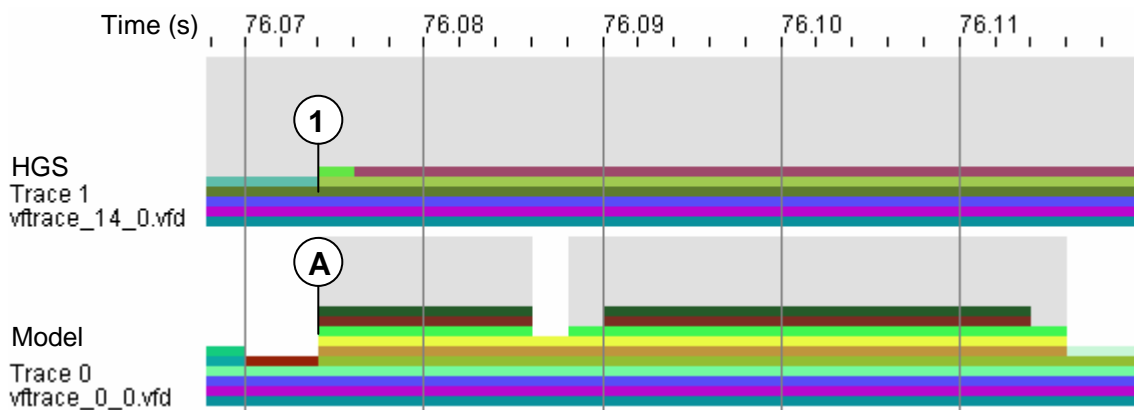


Figure 4.1 – Model receives boundary file input - overview

But the time scale is still too coarse to follow the sequence of events during the process of boundary data input. The most interesting part is the initial phase, where the model opens the input file and reads the first data. This phase is presented in more detail in Figure 4.2.

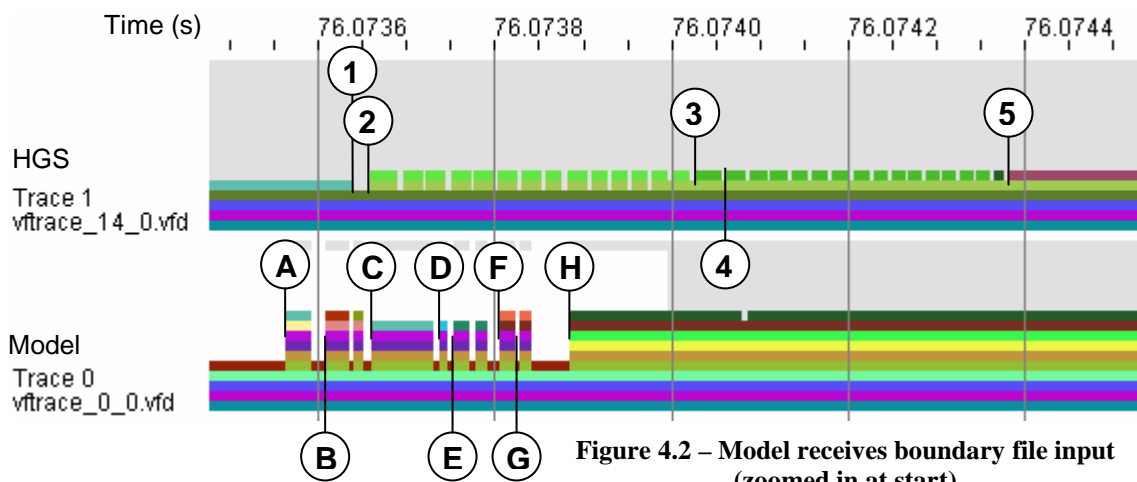


Figure 4.2 – Model receives boundary file input (zoomed in at start)

The model input process stages are listed below in sequence, referring to the time markers in Figure 4.1 and Figure 4.2.

Model	HGS
1. Subroutine <i>getgrb</i> calls <i>groploc</i> , which calls <i>hgs_gropen</i> (T_A).	1. The HGS remains in the command loop input at this stage.
2. <i>hgs_gropen</i> <ol style="list-style-type: none"> a. calls <i>hgs_check_input_available</i>, which does a <i>mpi_recv()</i> to get the confirmation (T_A). b. sends an open request to the HGS (T_B). c. receives DDR (T_C) and copies it to the right location (T_D). d. creates two input buffers to receive the input data (T_E). e. calls <i>hgs_get_next_input_block</i>, which receives <ol style="list-style-type: none"> 1. a header block (non-blocking, model task rank 0 only) (T_F); 2. a field block (non-blocking) (T_G). 	2. HGS response: <ol style="list-style-type: none"> a. No action: the HGS had already sent the confirmation at the end of an earlier pre-read. b. REQ_GROPEN received by HGS, served by subroutine <i>hgs_send_inputdata</i> (T_1). c. <i>hgs_send_inputdata</i> sends the file's DDR to all model tasks (T_2). d. No related HGS action. e. <i>hgs_send_inputdata</i> actions: <ol style="list-style-type: none"> 1. sends next header block to model rank 0 (T_3); 2. sends next field block to all model tasks (T_4) and deallocates the input buffer after it has been sent to the model.
3. <i>getgrb</i> calls <i>grrdloc</i>	3. No related HGS action.
4. <i>grrdloc</i> calls <i>hgs_gread</i> repeatedly for each field (T_H), actions: <ol style="list-style-type: none"> a. if no more fields in buffer: waits for the next input block and calls <i>hgs_get_next_input_block</i> (see action 2.e) b. copies field from buffer, verifies field parameters (rank 0 only) and returns 	4. Waits until next input block requested (T_5). <ol style="list-style-type: none"> a. Repeat action 2.e b. No related HGS action
5. Repeats action 4 until all fields read.	5. Repeats action 4 while there are buffers to be read.
6. Subroutine <i>getgrb</i> calls <i>grclloc</i> , which calls <i>hgs_grclos</i> , which is a dummy in the model.	6. Returns to the command loop.

4.4.4.3. Model writes output data - *hgs_collect_output*

Model and HGS interact tightly in the output phase, moving the data from model to HGS in the shortest possible time. Model subroutine *gemini* calls subroutine *putdat*, which in turn calls (via other subroutines) the routines *postpp* and *getgrb* in library *grdy*. The file output routines called further down the call tree for open, write and close are *hgs_gwopen*, *hgs_gwrite* (through *gwwrloc*) and *hgs_gwclose*. The functionality of the latter three depends on the context: model or HGS task. The next section describes the use of these routines when called in the context of a HGS task.

The model sends its output data and forgets about these. The HGS job is to collect these data from the model, while allocating the buffers to store the output data.

Time marker 8 in Figure 3.2 indicates one of the intervals, in which this output procedure is executed. Because the interval is very short, the interval is presented on a more suitable time scale in Figure 4.3. The model trace shows a number of activity blocks associated with sending output data blocks, following time marker A. (Actually, in this interval one of three output GRIB files, the model history file, is output; the other files are created similarly.)

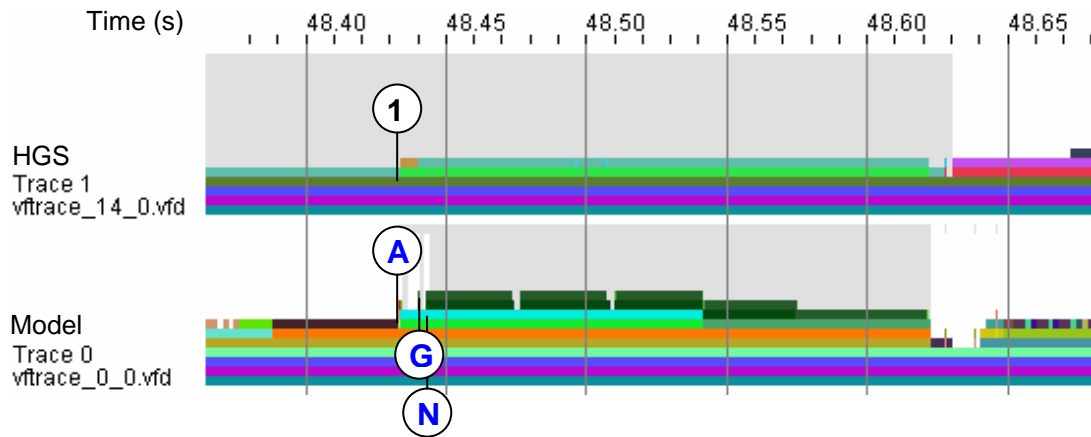


Figure 4.3 – Model sends output data - overview

The most interesting parts are the file open (after marker A), buffer swap (G) and buffer flush (N), which are presented in more detail in Figure 4.4.

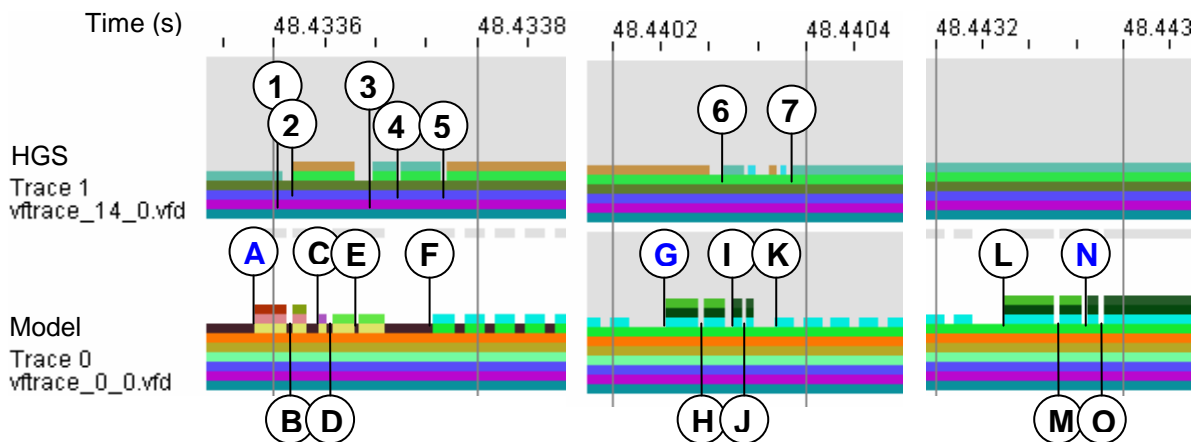


Figure 4.4 – Model sends output data - overview

The model output process stages are listed below in sequence with reference to the time markers in Figure 4.3 and Figure 4.4

Model	HGS
<ol style="list-style-type: none"> 1. Subroutine <i>putgrb</i> calls <i>hgs_gwopen2</i> (T_A), which <ol style="list-style-type: none"> a. sends a request (T_B); b. sends the file parameters (date, time, etc.) with TAG_OPEN (T_C); c. sends the DDR (T_D); d. creates an output buffer and a send buffer, then returns (T_D,T_E). 2. Routine <i>putgrb</i> calls <i>gwwrloc</i> repeatedly, which calls <i>hgs_gwrite</i>, which copies the field parameters to the field headers block and the field to the fields block in the output buffer (T_F). 3. When the output buffer is full, <i>hgs_gwrite_flush</i> is called (T_G), which <ol style="list-style-type: none"> a. swaps output and send buffer b. transmits (non-blocking) the send buffer in two parts: task 0 sends headers block (T_G,T_L), all tasks send the fields block (T_H,T_M) c. waits until the output buffer is ready (completion of previous sends: T_I, T_J and T_N,T_O); the trace at time T_O shows that the model task waits until the send buffer transmission is complete. 4. Repeat of 2 and 3 until <i>putgrb</i> has sent all fields. 5. Subroutine <i>putgrb</i> call <i>hgs_gwclos</i>, which sends a command message with TAG_CLOSE (not shown). 	<ol style="list-style-type: none"> 1. From the command loop receiving requests: <ol style="list-style-type: none"> a. receives REQ_PUTGRB request, served by <i>hgs_collect_output</i> (T₁), which enters a loop checking for messages from the model via <i>mpi_probe</i> (T₂); b. receives the file parameters (T₃). c. receives the DDR (T₄); d. no HGS involvement, waits for next output command via <i>mpi_probe</i> (T₅). 2. No HGS involvement until the buffer is full. 3. <i>mpi_probe()</i> returns with tag TAG_PARAM, <i>hgs_collect_output</i> <ol style="list-style-type: none"> a. allocates new output buffer and links it to the previous buffer; b. receives headers block from model task 0 (T₆) and fields blocks from all model tasks in a first come, first serve fashion (T₇). c. cycles receive loop and calls <i>mpi_probe</i> again. 4. Repeat of 3. 5. <i>mpi_probe()</i> returns with TAG_CLOSE (no further action needed for output file) to command loop in <i>hgs_init</i> for next HGS request.

4.4.4.4. HGS stores output data - *hgs_engrib_and_store*

After the model has “written” its output files, the data are still in the output buffers of one or more HGS tasks. Before the model continues with the next time step *gemini* sends the request REQ_STORE for each of the NPPSTR post-processing files on luns LUNPPF(1:NPPSTR) and the model output file on *lun* LUHIFI. The HGS task(s) associated with these *luns* create the output files and free the data buffers.

It is important to send the REQ_STORE requests *before* the REQ_GETGRB pre-read request, because a pre-read request before the store would have two unwanted side effects:

- the HGS would need more memory to contain input and output files at the same time;
- while the HGS is in the process of reading an input file it cannot serve output requests, stalling the model until pre-read completion.

The HGS handles the store requests autonomously in subroutine *hgs_engrib_and_store* as described below.

The DDR is copied from the HGS to the model file administration by *hgs_set_ddr*.

Subroutine *gwopen2* is called to open the GRIB output file.

Subsequently, each output record is retrieved from the record blocks in a linked list of output buffers. The records contain the field parameters (type, number, level and grid size and offset) and the field data of all sub grids. The global field is constructed from the sub grid fields and, if necessary, extracted according to the grid size and offset specified for this field. Subroutine *gwrite* is called to encode and write the record to the GRIB file.

Once all records have been written and all buffers have been deallocated, subroutine *gwclos2* is called to close the file. The return from *hgs_engrib_and_store* takes the HGS back to the command loop in *hgs_init* to wait for the next request.

4.4.5. Multiple HGS tasks

The use of multiple HGS tasks is supported: several input and/or output files can be processed in parallel. Figure 4.5 presents the trace graph of a 15 processor run with 3 processors assigned to the HGS (only one of the 12 model task traces is shown). The I/O operations are distributed across the HGS tasks in a round robin way, binding a HGS task to a Fortran logical unit number (lun). This feature is especially useful when the number of processors assigned to the model is large, reducing the pure model computation time to less than the time needed for I/O.

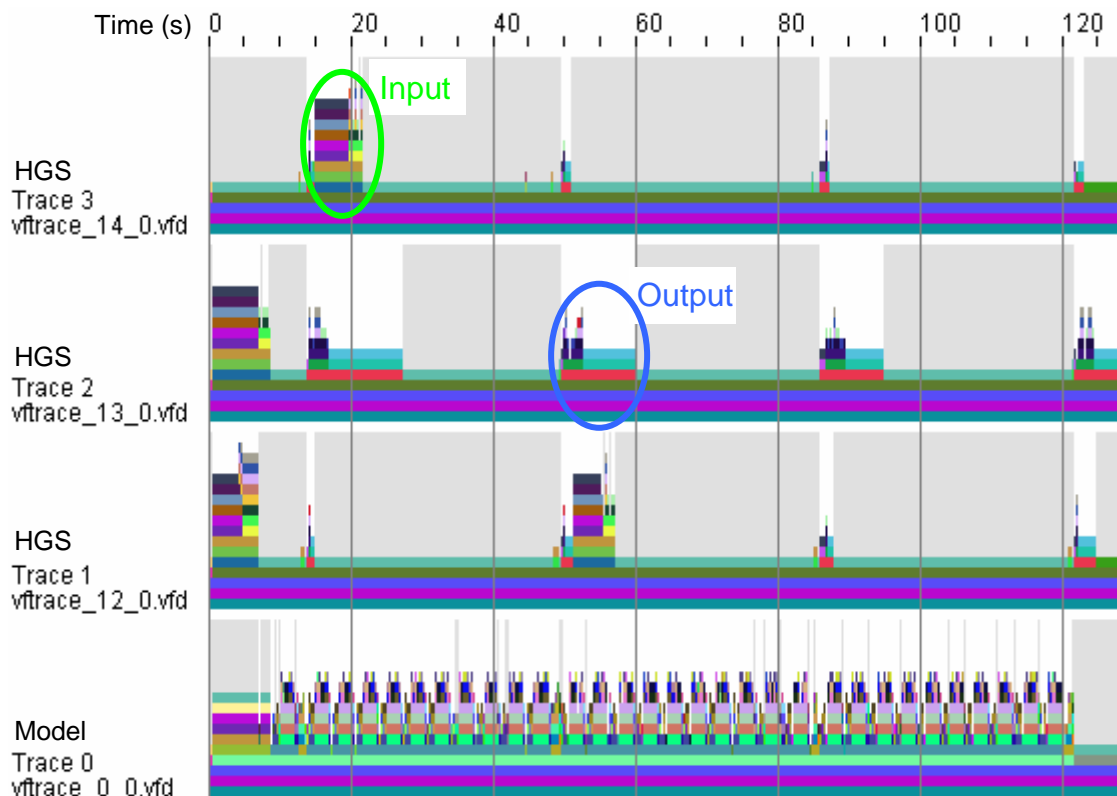


Figure 4.5 – Multiple HGS tasks: 3 HGS, 12 model (1 shown)

5. Performance

The performance gain depends on several parameters, including:

- the amount of I/O demanded (forecast hours between boundary input and model output);
- the number of processors used;
- the performance of the filesystem used;
- the amount of memory required to buffer the files.

When comparing runs with and without HGS task on a fixed number of processors, there are several effects, which have a positive or negative effect on the runtime:

Negative: one processor less to do the computations.

- Positive or negative: the grid decomposition is different with one processor less. For the demonstration runs presented in section 3, 15 processors were used in total. The sub grids are arranged as 3x5 without and 2x7 with HGS. The latter is less efficient, because the asymmetry is higher, so in this particular case the effect is negative.
- Positive: the GRIB encode/decode and I/O time is removed from the model time (except for the first boundary and last output files). Obviously, this is the main effect.
- Negative: the time needed to communicate with the HGS is short, but not zero.

The simple example runs discussed in section 3 show a runtime reduction from 160 to 129 seconds, or almost 20%. Because the forecast length is only 3 hours, the amount of waiting time for the first boundary files and the last output files is relatively large, the improvement for a real forecast run would even be higher.

6. Known limitations

There are a few issues, which may have to be addressed when applying the HGS functionality in a specific operational environment.

6.1. *Namelist input to be used for boundary file pre-read*

When the HGS reads the boundary file, it calls the HIRLAM routine `getgrb`. The functionality of this routine depends on specific HIRLAM parameters, input by means of a HIRLAM namelist file, which should be communicated explicitly from model to HGS. In the current setup for DMI, a pragmatic solution was chosen: hard-coding the desired parameters in subroutine `hgs_getgrb`.

6.2. *HGS process memory*

The model I/O has to be completed in the shortest possible time, requiring the HGS to accept or provide the data immediately when the model task provides or requests these data. The HGS is able to meet this demand by keeping the data in memory between input from disk and model request for boundary data, or model request and output to disk for output data.

When using the HGS on a system with an insufficient amount of memory per processor, it may be necessary to introduce a temporary file on a fast, local filesystem to store the raw field blocks. During the communication with the model, data would be written to or read from this temporary file. To obtain a high enough level of performance, it may be necessary to make use of the system's asynchronous I/O features when reading/writing this file.

7. Concluding remarks

The HIRLAM GRIBfile Server (HGS) version as described is the result of an optimization effort by DMI and NEC to obtain optimal performance on DMI’s HPC system consisting of 8 NEC SX-6 nodes of 8 processors each.

The benefit of the HGS can be demonstrated in a simple picture, showing the activity trace graph of two 3 hours forecast runs, one without and one with the HGS enabled (Figure 7.1).

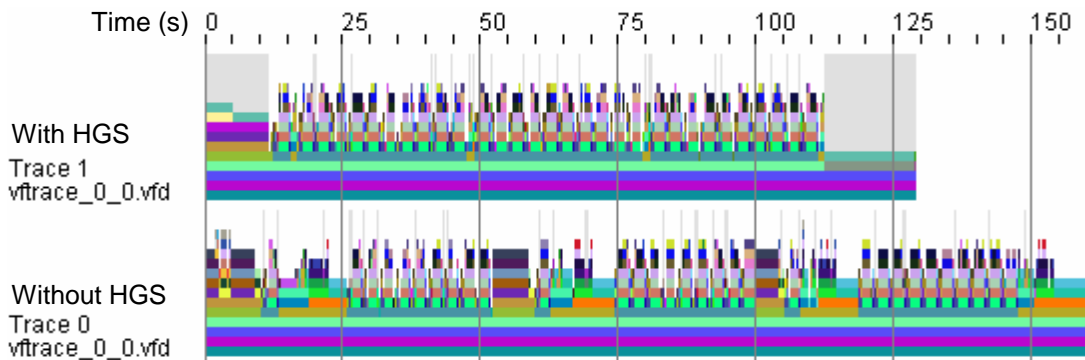


Figure 7.1 – Forecast runs with and without HGS

To ease the port to other platforms a debugging feature has been added in the form of runtime controlled debug prints with time stamps, which make it possible to monitor the model and HGS interaction in detail.

8. References

1. Improving Load Balance in a Weather Code: Asynchronous Output in HIRLAM with MPI by Jussi Heikonen (CSC) and Kalle Eerola (FMI), preprint 2002.
2. *Asynchronous I/O in HIRLAM* by Ole Vignes (DNMI), HIRLAM Newsletter 41, July 2002.
3. *Improving HIRLAM scalability by asynchronous GRIB file handling* by Jan Boerhout (NEC), HIRLAM Newsletter 39, October 2001.
4. FORTRAN 95 Language Definition in document ISO/IEC 1539-1:1997(E).