

Asynchronous I/O in HIRLAM

Ole Vignes, DNMI, May 2001

Abstract

In the past two different implementations of asynchronous I/O for HIRLAM has been developed. One of them by Jan Boerhout, Sun Microsystems, in conjunction with the benchmark for KNMI's latest acquisition of a new computer. Boerhout's approach is called HIRLAM Gribfile Server (HGS), and is described in HIRLAM Newsletter No. 39, October 2001 [1]. The second implementation is developed by Jussi Heikonen, CSC and Kalle Eerola, FMI. It is described in [2]. The present work has been an attempt to unify the two approaches, and to make a code suitable for the HIRLAM reference system.

1 Introduction

Although HIRLAM has successfully been run on a large number of processors, the scalability is severely limited by the fact that only one processor is doing input/output while the others are idle. In fact, not only I/O, but also encoding and decoding of GRIB fields is handled by one processor only. There are at least two possibilities for improving this situation. One is *parallel I/O*, in which more processors would be engaged in the GRIB encoding/decoding and also the reading and writing of files. Another possibility is *asynchronous I/O*, in which the elapsed time spent in GRIB and I/O operations is not reduced, but rather "hidden" behind the numerical computations. The idea is to use dedicated processors to do the I/O, and route I/O fields to/from the computational processors via memory, which is often two orders of magnitude faster than accessing disks.

In the following we give a brief description of the two implementations of asynchronous I/O developed for HIRLAM.

1.1 HIRLAM Gribfile Server, HGS

The HGS implementation is based on a Unix System V feature known as shared memory, which is again part of a set of features called interprocess communication (IPC). System V shared memory should not be confused with the SHMEM communication system for parallel programs, as the two systems are totally unrelated. The native interface to IPC is the C language, thus most of HGS is written in C.

The HGS I/O server is created in the main program by a simple fork, before SHMEM initialization. Communication between the HIRLAM master process and the HGS server is via an anonymous pipe, created before the HGS child is forked. Both input and output is handled asynchronously, but the actions of the HGS server was hardcoded in the KNMI benchmark version. It would have to be generalized to read input namelists in order to handle an arbitrary schedule.

The HGS code was designed to limit the code modifications of existing routines to a minimum, and to leave low level GRIB and I/O code unchanged. This is achieved by inserting an extra layer of subroutines HGS_GROPEN, HGS_GREAD and HGS_GRCLOS for input and HGS_GWOPEN2, HGS_GWRITE and HGS_GWCLOS2 for output at the places where their original (non-HGS_) counterparts were called. For example, in HGS_GREAD a test is made

if we are inside HGS or inside HIRLAM. If in HGS, then a grib-field is read, decoded and placed in a shared memory segment. If in HIRLAM, then every PE reads its part of the field directly from the shared memory segment. A similar logic is used for output. The pipe between HIRLAM PE 0 and the HGS server is used to send completion signals, so that HIRLAM does not try to read a segment before it is ready, and vice versa, that HGS does not begin to encode and write data from a shared memory segment before all HIRLAM PEs have put their data there.

Since the HGS server is created before SHMEM initialization, it does not have a PE number, and thus the existence of this extra process does not disturb the SHMEM synchronization and data exchange operations in the model. While this is a desirable feature, it is not easily ported to MPI. On most computers, MPI programs are started by something like `mpirun -np <n ranks>`, and the creation of `<n ranks>` processes happens immediately. A new fork after this, even if on rank 0 only, is likely to disturb subsequent MPI communication.

1.2 The CSC/FMI implementation

The CSC/FMI implementation is written in Fortran 90 and is adapted to the MPI message passing option of GC_COM, the general communication library used in HIRLAM. Suppose $N = \text{NPROCX} * \text{NPROCY}$. The CSC/FMI code would then be started by `mpirun -np N + 1`, where ranks $0, \dots, N - 1$ are computational processes and rank N is the I/O server. Or output server to be more precise, since only output is handled asynchronously in this implementation.

To separate the output process from the rest a new MPI communicator is defined that includes only the computational processes. Then the original MPI_COMM_WORLD communicator is redefined (inside GC_COM) to the newly created one. This makes GC_COM behave as before, but on the subset of computational processes only.

The output process executes a separate subroutine from the very start of HIRLAM. It is controlled by HIRLAM rank 0, which is the process doing I/O originally. When HIRLAM rank 0 enters an output routine, it merely sends the necessary calling parameters to the output rank N , and then immediately returns to computation. The HIRLAM processes $0, \dots, N - 1$ now all send their data to the output rank N , instead of to rank 0.

To be able to proceed immediately with computation, HIRLAM processes use non-blocking communication calls when sending control parameters or data to the output process. For this to work, extra buffers are allocated to make sure the data sent are not modified before the corresponding receive is complete. This handling of extra buffers is implemented as dynamically allocated linked lists of user-defined buffer types, hence the need for Fortran 90.

2 The unified implementation

The original goal of this work was to integrate the HGS code in the HIRLAM reference system. However, while working with the code it was realized that it might not be as portable as desired. HGS works on SUN and SGI machines with shared memory, but it would not run on e.g. the Cray T3E, which does not support SysV shared memory, and it would not run on e.g. a Linux cluster, which does support SysV shared memory, but does not run a single instance of the operating system. At the same time, *met.no* had got access to the code developed at CSC/FMI based on MPI, and it was clear that this code would be portable to a larger number of computing systems than HGS. The decision was therefore made to develop a unified version with code from both the available implementations, with a common interface

and a common set of features as far as possible. The idea is that if you can run HGS you should probably do that, because it is believed to be faster. But if you cannot, you can still run an MPI version with almost the same functionality. To achieve this, some changes had to be made to both the previous versions.

The name HGS has been used also for the unified version, mostly because the call structure of original HGS was more general in that it included both input and output. It also had almost all of its modifications in library `grdy`, while the CSC/FMI version also had modifications in `grw1`, which appear to be unnecessary. A lot of the work therefore consisted in writing MPI counterparts of HGS routines based on ideas from the CSC/FMI code. So in the following, HGS is the name for the unified asynchronous I/O option in HIRLAM, but there are two underlying implementations. One based on Unix System V features and written mostly in C. This will be called the IPC implementation of HGS. The other implementation is written in Fortran 90, and is based on MPI message passing. It will for simplicity be called the MPI version of HGS. This is slightly misleading, because the IPC version can also be combined with MPI, not only SHMEM. The real question is whether the computing system supports SysV IPC or not.

Here are some of the features of the unified version:

- More than one PE or rank can be used as an I/O server if desired.
- No reading of namelists by the I/O server(s).
- Fallback to original, synchronous I/O without recompilation possible.
- Both input and output handled asynchronously.

Each of these require some further comments. As mentioned, the need in HGS to fork the I/O server before SHMEM initialization was felt to limit its portability. This has therefore been changed, so that the I/O server now has its own PE number or rank. This change implied that an anonymous pipe could no longer be used to communicate between HIRLAM PE 0 and the I/O server. It was therefore decided, since HGS already required SysV shared memory, to use another SysV feature for communication with the I/O server, namely a *message queue*. With this change, it also became apparent that the work in extending the functionality to allow for more than one I/O server was relatively straightforward. Testing on an SGI Origin 3800 with 220 CPUs also showed that the use of more than one I/O server was a highly relevant extension.

This extension also suggested the solution to the problem of the hardcoded actions of the I/O server in original HGS. In the new implementation, all HGS servers are turned into pure “slaves”, in that they neither have a predetermined schedule nor read any namelists to get one. Instead, they take all their “orders” from HIRLAM PE 0 by reading the IPC message queue. The first server to read a particular message gets the task, and then the system automatically removes this message from the queue. At this point the MPI implementation differs from the IPC implementation. In the MPI implementation HIRLAM rank 0 has to direct messages to a particular I/O server, it cannot just place requests on a queue. Therefore, in the MPI implementation, once a particular I/O server has been assigned one logical unit, it will do all I/O on this unit for the rest of the run. In the IPC implementation one I/O server could take the output from e.g. unit 80 at one postprocessing timestep, while a different server could do the job the next time unit 80 is ready for output.

HIRLAM ordinarily reads its input from namelists. These namelists are read by process 0, and values are distributed to the other processes as needed. In the case of the number of I/O servers however, this value is read from an environment variable called `NPROC_HGS`. The reason for this deviation is the following: When HIRLAM process 0 communicates namelist parameters, this should be done to the computational processes only. But before the computational processes can set up the proper communicator, they have to know the number of I/O servers. Thus this variable is not suited for a namelist, unless all processes read this namelist. Instead, the choice has been made to use an environment variable. Note that this environment variable, `NPROC_HGS`, may be set to 0. What will happen in this case (and also if it is unset) is that asynchronous I/O is turned off, and HIRLAM will revert to its old I/O scheme. This flexibility comes in addition to the possibility of not compiling in asynchronous I/O support at all (in which case the `NPROC_HGS` variable does not matter, of course).

As indicated above, in the unified version both implementations handle input asynchronously. For this to be of any benefit, the I/O servers must get a signal to read an input file well in advance of HIRLAM actually needing these data, since they do not know anything about the sequence of I/O events. This has been solved by inserting HGS requests at a few places in subroutine GEMINI. The first two boundary files are requested right at the beginning, and later, as soon as data from a boundary file has been distributed, prereading of the next file is immediately requested, unless of course, the last one has already been read. The MPI implementation differs slightly from the IPC implementation in that it needs an extra request for input. On the first request, it will read the file from disk and place the fields in a linked list of buffers. But these buffers are allocated locally in the I/O server's memory, so it needs an additional signal in order to start the distribution of fields to the computational processes when they are actually needed. In the IPC version, each computational process goes straight to shared memory to read their data.

The IPC version of HGS has one feature that the MPI version does not have. Because of the global nature of SysV shared memory and message queues, it is possible to run the I/O server part of HGS as a standalone program. This means that the computational part is another standalone program, and this is the reason why such a split might increase efficiency. At least on the SGI Origin, synchronization between all processes in a parallel program is faster than synchronization among a subset only. The option is activated by setting environment variable `HGS_STANDALONE=yes`. In addition, two instances of `hlprog` should be started, one with `$NPROC_HGS` processes and another with `NPROCX*NPROCY` processes. The instance that runs on only `$NPROC_HGS` processes will automatically detect that it is a pure I/O server program. This makes the IPC version again look more like the original HGS, in that the HIRLAM synchronization is untouched.

2.1 Limitations

As mentioned above, the unified HGS version now opens up for the possibility to use multiple I/O servers, in case the workload for a single server is too high to keep up with the computational processes. It is however important to realize that extra I/O servers will not automatically increase performance in this situation. To understand why, we need to look at the underlying implementation in more detail.

First of all, at least in the current version, multiple I/O servers cannot cooperate in the reading or writing of a single file. This has to do with the way the underlying low-level I/O routines work. Therefore the potential for I/O workload sharing is in the distribution of

input/output files across the HGS processes.

In original HGS, i.e. the IPC version, each Fortran logical unit used for I/O is assigned its own shared memory segment. This segment is designed to hold the fields of one input or output file only. So when e.g. a segment is filled with output fields, this segment can not be reused by the model until its contents are processed and written to disk. It is therefore important, regardless of the number of I/O servers, to use different logical units if multiple files are input or output at the same timestep in the model. This applies in particular to the different post-processing streams defined in namelists namppp. If this rule is followed, then it will also be possible for multiple I/O servers to process different logical units at the same time.

The same system with one linked list of buffers for each logical unit is also used in the MPI version. The original CSC/FMI code was more flexible in the buffer handling, but in order to simplify the extension to asynchronous input and multiple servers the MPI code now has a similar handling of logical units as the IPC version, thus the above rules apply also in this case.

2.2 Peculiarities of the IPC version

As mentioned previously, System V IPC shared memory segments and message queues are global resources that can be accessed by any process running on the same computer. It is therefore important that they are given unique identifiers or *keys*. These keys are unsigned integers of (at least) 32 bits. The scheme used to produce unique keys in HGS is as follows. The leftmost 16 bits are the numerical user-id (UID) of the user running HIRLAM. Then comes 4 bits where an experiment number in the range 0–15 may be set (by setting environment variable HGS_EXPNUM), and finally 12 bits for the logical unit (the message queue has “logical unit” 1). This is illustrated in Figure 1. With this scheme it should in principle

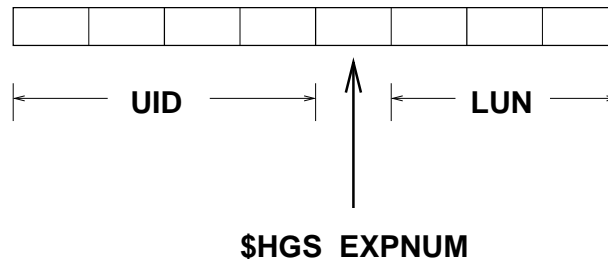


Figure 1: IPC key construction

be possible for any number of users to each run up to 16 different instances of HIRLAM with asynchronous I/O. In practice this will of course not be very relevant. Asynchronous I/O is probably only beneficial when running on a relatively large number of processors, and for obvious reasons not many such jobs may coexist on a single machine.

Another possible problem with IPC resources is that they may well outlive the process that created them. This can be a problem if a program crashes and is restarted. If the message queue is not empty on restart, it is likely that the run will hang on the next attempt. To avoid this a signal handler is installed that will catch as many of the UNIX signals that it can. When invoked this handler will remove the message queue and all still existing shared

memory segments created during the run. However, this is not guaranteed to always work, so it might still be necessary to remove IPC resources manually via the commands `ipcs` and `ipcrm`.

3 Results and discussion

The IPC implementation of HGS has been tested on a case where HIRLAM was run on 196 (`NPROCX=NPROCY=14`) processors on an SGI Origin 3800. The model had $468 \times 378 \times 40$ gridpoints and was run for 48 hours. Input boundary files were read every third hour, and amounted to around 400Mb total. Total output was around 1600Mb, with most of the fields output every third hour, plus a few surface fields every hour.

Five different cases were run, a reference run without asynchronous I/O, and the four possible combinations of 1 or 2 I/O servers with or without HGS running as a standalone program. Unfortunately the results were obtained in a period where the test machine was quite heavily loaded, so the timings do not look completely logical. To get an indication of the load, the time used in a single I/O-free timestep is also listed in addition to the CPU-time used for the complete 48 hour forecast. The results are summarized in Table 1.

NPROC_HGS	HGS_STANDALONE	Non-I/O step	CPU sec.
0	—	1.40	833
1	no	1.68	563
2	no	1.67	565
1	yes	1.79	572
2	yes	1.37	457

Table 1: CPU times for the 5 runs

The results were obtained with version 5.0.2 of HIRLAM, which has the old implementation of the digital filter initialization. In this implementation a lot of redundant I/O is done between the backward and forward steps of DFI, which greatly favors the HGS runs. Therefore, in order to be more realistic comparisons to the newest version with recoded DFI, the times used in the initialization part is *not* included in any of the total run times above. Nevertheless, we see that in all the runs with asynchronous I/O, the run time is reduced with from 32 to 46 percent compared to the normal synchronous I/O scheme. Jan Boerhout reported a 17 percent speedup on 64 processors on a SUN Enterprise 10000, and it is only logical that when going to even more processors the advantage of asynchronous I/O gets even bigger. This is illustrated in Figure 2, where the time spent in the various timesteps is plotted for cases 1 and 5 in the table. While timesteps with I/O can take nearly 20 seconds in the case with synchronous I/O, this is reduced to around 3 seconds in the asynchronous I/O case.

4 Summary and conclusions

A unified version of asynchronous I/O has been prepared for a coming HIRLAM reference version (> 5.1.4). There are two underlying implementations, one based on Unix System V interprocess communication and mostly written in C, and another one based on MPI message

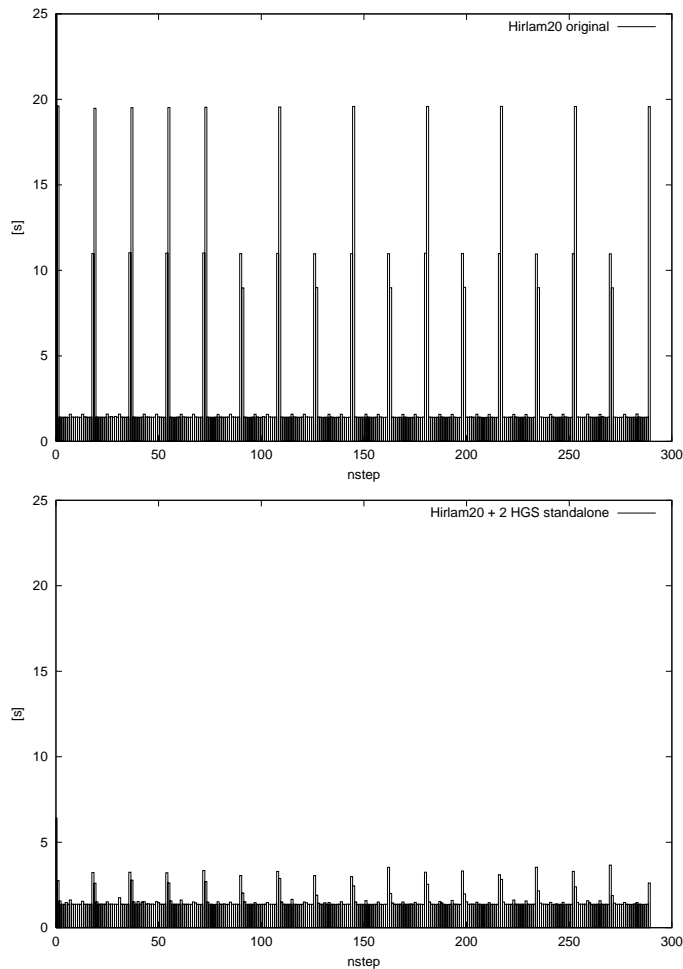


Figure 2: Seconds per timestep without and with asynchronous I/O

passing and written in Fortran 90. The name HGS (HIRLAM Gribfile Server) has been retained from Jan Boerhout's version.

The unified version has extended the functionality of both of the previous implementations. It is now possible to use more than one I/O server, and both input and output is always handled asynchronously.

Tests on an SGI Origin 3800 where HIRLAM was run on 196 processors showed a reduction in CPU time between 32 and 46 percent for the IPC version of asynchronous I/O compared to the original I/O scheme. Time has not yet allowed extensive testing of the MPI version.

References

- [1] Boerhout, J.: Improving HIRLAM Scalability by Asynchronous GRIB File Handling, HIRLAM Newsletter No. 39, October 2001, pp. 41–48.
- [2] Heikonen, J. and Eerola, K.: Improving Load Balance in a Weather Code: Asynchronous Output in HIRLAM with MPI, preprint 2002.