Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

An Interface for Halo Exchange Pattern

Mauro Bianco

Swiss National Supercomputing Centre (CSCS)

Abstract

Halo exchange patterns are very common in scientific computing, since the solution of PDEs often requires communication between neighbor points. Although this is a common pattern, implementations are often made by programmers from scratch, with an accompanying feeling of "reinventing the wheel". In this paper we describe GCL, a C++ generic library that implements a flexible and still efficient interface to specify halo-exchange/halo-update operations for regular grids. GCL allows to specify data layout, processor mapping, value types, and other parameters at compile time, while other parameters are specified at run-time. GCL is also GPU enabled and we show that, somewhat surprisingly, GPU-to-GPU communication can be faster than the traditional CPU-to-CPU communication, making accelerated platforms more appealing for large scale computations.

1. Introduction

Many applications in high performance computing use domain decomposition to distribute the work among different processing elements. To manage synchronization overheads, decomposed sub-domains logically overlap at the boundaries and are updated with neighbor values before the computation in the inner region proceeds. A subset of applications using domain decompositions are finite difference kernels on regular grids, which are also referred to as *stencil computations*, and the overlapping regions are called *ghost* or *halo* regions that need to be updated with data from neighbor regions during an operation often referred to as *halo update*. Typically, these applications use Cartesian grids and each process handles a set of regular multi-dimensional array of elements with halo elements. We call these arrays data fields to emphasize the role they have in application programmer view. Communication for the halo update, on large machines, is usually implemented using MPI. The communication portion of the halo update is called *neighbor exchange*, or *halo exchange*, which is a *communication pattern*.

The distinction between halo update and halo exchange is required since the communication aspect of the update operation can be seen as a collective communication operation that involves only data movement, while the "update" phase is related to the semantic of the algorithm. Indeed, even though the data exchange pattern (neighbor exchange) is clearly defined when the Cartesian computing grid is defined (either a mesh or a torus), at application level there are many parameters that can vary when we talk about halo update: one is the mapping between the coordinates of the elements in the domain and the directions of the coordinates of the computing grids; the data layout of the domains themselves; the type of the elements in the domain; the periodicities in the computing grids to manage the case in which certain dimensions "wrap around" or not. Additionally, when we want to deal with accelerators, which typically have their own address space, we need also to specify where the data is physically located (either host or accelerator memory).

While these parameters are application dependent, others are architecture/platform dependent. Other degrees of freedom are related to how to perform communication (e.g., asynchronous versus synchronous), what mechanism to use to gather and scatter data (halos) from the domains, etc. All these variables make the specification of a halo exchange collective operation quite complex. For this reason we have developed the

Generic Communication Layer (GCL) to provide a C++ library of communication patterns directly usable by application programmers and flexibly matching their requirements. GCL has been developed from requirements of climate application developers who needed to port their application to GPUs with reasonable effort. At the moment, GCL provides two rich halo exchange patterns, one used in case the structures of the halos are known when the pattern is instantiated but data can be allocated on the fly, and another in which the halo dimensions can be specified when the exchange is needed. GCL also provides a generic all-to-all exchange pattern that allows specifying arbitrary data exchange but it is not fully optimized for performance.

GCL has been designed as a multi-layer software. At bottom layer (subsequently dubbed "L3") the definition of a communication pattern involves only data exchange considerations. The only think the lower level is concerned about is the exchange of buffers of data from a process *i* to a process *j* at a given time. Above is a more application oriented pattern specification (subsequently dubbed "L2"), which deals with user data structures. For instance, at this level, the halo exchange communication pattern can process any 2D or 3D arrays (in what follows we focus on 3D for simplicity), of arbitrary element, types, with standard memory layout, and arbitrary halo widths and periodicities. To deal with less common data structures and applications, another layer had been devised (subsequently dubbed "L1") to use high order functions to collect data and communicate, as is the case for other projects [1,2,3]. This level has not been implemented yet and requires careful design of the interfaces that we would like to engage with other partners.

The rest of the paper is organized as follows: Section 2 presents the software architecture of GCL. Section 3 will present the main interfaces at the bottom level, which are not typically accessed by the application developer. Section 4 introduces the main interfaces available to the user for performing halo updates. Section 5 discusses some software engineering issues. Section 6 will present some performance results on different architectures. Finally Section 7 concludes the paper with future developments considerations.

2. GCL Software architecture

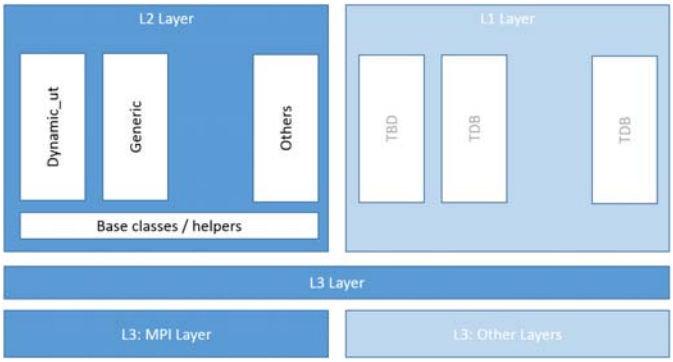

Figure 1: GCL software architecture. The light boxes are not actually implemented but planned and GCL is designed to be extended so. Users access L2 and L1 layers which use L3 layer to perform the communication pattern. The current available L3 level uses MPI, but other communication libraries are planned.

Even though GCL is designed to be a *library of communication patterns*, the main focus for the current users is the halo update operation at level L2. To deal with other, non performance-critical, communication, a generic all-to-all is also provided at level L2 and L3. At L2 it assumes 3D arrays are available (in C/C++ we assume a 3D array uses affine expressions for accessing data), on L3 buffers of data are assigned from/to any source/destination.

GCL is designed to be layered so to simplify extension. User accesses GCL through L2 and L1, which use L3. Certain L3 facilities might be used by the user, such as the `proc_grid`s, that describe Cartesian grids of processors, which is then used by the communication pattern to locate neighbors. These facilities, being at level L3, may be bound to specific communication mechanisms available on the available machine. For instance an MPI computing grid requires knowing about `MPI_communicator`s. One of the design goals of GCL was to use different low-level communication mechanisms, such as active messages, distributed global address spaces, etc. There are some challenges in doing so that will be discussed in a subsequent section.

At L2, users can access different interfaces depending on their needs, and we discuss later the available interfaces and the ones that will be developed next. At software architecture level, we can note that these interfaces rely on some common set of base classes and helper functions that the *library developer* (as distinct from the *application developer* that is the *user of GCL*) is strongly advised to re-use. However, the dependencies with the level below make this set of tools intrinsically quite complex.

Excluding a little runtime component providing initialization and finalization of the library, GCL is a C++ *template*/*header-only* library, thus the source code of the library is integral part of the library itself. This allows the compiled program to be very well optimized, and permits the debugging of the application through stepping into the program lower levels of abstraction. Even though this can be not a valuable feature for the novice user, it is useful to the advanced one to understand which paths the program goes. This can be seen in other template libraries, such as the STL.

3. Generic Halo Exchange: L3

At lower level, the halo exchange interface depends on the underlying communication mechanism chosen on the machine in hand. GCL, at current state, uses MPI and L3 provides two main interfaces for halo exchange communication pattern. The first one is `Halo_Exchange_3D` which is a class that accepts as template argument a `proc_grid` type, which is a class describing a Cartesian grid of processors. When constructed the actual processor grid object must be provided and subsequently the programmer can set it up providing information about location of the buffers for sending and receiving. The main interfaces are divided in two categories: the first sets up the communication pattern, the second execute the halo exchange.

```
register_send_to_buffer<0,1,1>(pointer, size); //or
register_send_to_buffer(pointer, size, 0, 1, 1);
```

Figure 2: GCL interface for halo registering buffers to exchange at L3 for a 3D pattern.

The first set includes:

- `register_send_to_buffer`, to tell the pattern about a sending buffer. The interfaces to do so is shown in Figure 2. To indicate the neighbor to which the buffer is intended to, the user uses relative coordinates in the (integral) range [-1:1], which are relative coordinates of the destination processor;
- `register_recv_from_buffer`, to tell the pattern about a receiving buffer. The interface is analogous to the previous one.

After all the buffers are set up, users can invoke functions to send and receive data. The main interfaces are quite simple:

- `exchange()`: This member function takes the data in the send buffers (pointers to accessible memory can be pointing to accelerator memory) and send it to the corresponding neighbors, receives data from the neighbors, and returns when the operation is locally complete (i.e., there is no global synchronization at the end, so other processes may still be executing it);
- `start_exchange()`: Prepares the reception of data and launch the commands for sending local data. This function returns as soon this operation completes, so the data in buffers are not assumed to be already sent off or received;
- `wait()`: Wait for the local data to be sent and incoming data to be received. As `exchange()`, this function returns when locally finished.

Another available interface at level L3 is `Halo_Exchange_3D_DT`, which accepts `MPI_Datatypes` instead of linear buffers. This allows L2 pattern to avoid running packing and unpacking procedures, and letting the job to the MPI library. This does not have impact on user code, but breaks encapsulation between L2 and L3. Since `MPI_datatype`s does not seem to provide sufficient performance benefits on accelerators, this support may be dropped in future versions of GCL, but since some architectures may support `MPI_datatypes` better, the support is maintained for evaluation purposes.

When calling the exchange functions the data is assumed to be already prepared in buffers of `MPI_Datatypes`.

4. Generic Halo Exchange: L2

At level L2 the interfaces are more complex and diverse since the L2 layer is closer to application developers who manage higher level data-structures. For this reason it is necessary to introduce some basic concepts before describing the interfaces. In what follows we refer to a *data field* as a multi-dimensional array seen from the point of view of the application programmer as the space where to apply a finite difference schema (or some other computation that needs the halo update operation). A data field has 1) an *inner*, or *core*, region, that is the

sub-array including the indices corresponding to the physical domain, and 2) a *halo region* in which the data will be updated from neighbors, and is typically accessed *read-only* to make the computation in the inner region.

One of the most important concepts is the `halo_descriptor`, which is a simple structure with 5 integers that describe a dimension of a multi-dimensional array, used as data field. Figure 3 depicts the meaning of these numbers, which are

1. `minus`: the number of halo elements in the direction of decreasing indices;
2. `plus`: the number of halo elements in the direction of increasing indices;
3. `begin`: the index of the beginning of the inner region of the data field relative to the 0 index of that dimension;
4. `end`: the end index of the inner region (inclusive);
5. `length`: the total length of the dimension of the array.

Specified in this way, halo descriptors allow taking into account padding. Then a list or array of three halo descriptors describes completely a 3D array in terms of halo exchange requirements.

Figure 3: Halo descriptor numbers. In this example `minus`=2, `plus`=3, `begin`=3, `end`=8, `length`=13.

Users do not have to conform to a specific data layout when using GCL, as it may be the case for other libraries. They can think of the coordinates of data fields as first coordinate, second coordinate, and third (or ijk, or xyz, etc.). When specifying to GCL to instantiate a particular halo exchange pattern, the user will indicate how the coordinates will be sorted in increasing stride order. To do so, the user uses `layout_map`s, so that `layout_map<2,0,1>` means: "The first dimension, in user coordinates, has the greatest stride (it is the third in increasing stride order, so it has index 2), the second has the smallest stride (that is 1), and the third is the intermediate one". The increasing stride order has been selected to conform to FORTRAN users that were the first users of GCL. In this case, `layout_map<0,1,2>` correspond to the layout of a FORTAN 3D array (considering i,j,k typical coordinates). A C array will be indicated as `layout_map<2,1,0>`.

In addition, the user can indicate how the *directions* in user coordinates map to the processor grids. For instance, to move along the first dimension (let us assume it is "i") the user can find the data along the first, the second, or the third dimension in the computing grid. To specify this, GCL uses again the `layout_map` concept: `layout_map<1,2,0>` means: "The first dimension in data maps to the second dimension in the processors grid, the second into the thirds, and the third into the first". It can be cumbersome to think how to map things right, but it has to be typically done once for a given application.

An additional parameter that is important for halo exchange is the periodicity of the computing grid. This is passed as runtime argument (since it was a requirement expressed by the users to easily deal with different physical representations without recompiling the code). GCL offers the `boollist` concept, that allows the programmer to indicate the periodicities along user dimensions (GCL arranges it to match it to the increasing stride representation).

Other customizations are possible. One concerns the packing/unpacking methods to be used to collect and redistribute the data from and to data fields. Currently three methods are available:

- `manual_packing`: data is collected and redistributed by for-loops on CPUs data and by kernels on GPUs;
- `mpi_pack`: `MPI_Pack` and `MPI_Unpack` functions are used in both CPU and GPU data;
- `mpi_datatype`: a single `MPI_Datatype` describes all data to be collected in all data fields to be exchanged in a single step.

Depending on the platform certain methods can be faster than others. For instance, the last two methods are not very efficient in the current implementation for GPUs on MVAPICH2. We are currently working on improving the performance of this implementation. Certain MPI implementations do not support `MPI_Datatype`s at all on GPUs.

Finally the user can specify at compile time if the data is on host memory or GPU memory (other accelerators could be added, but GCL now only supports CUDA enabled GPUs).

As can be seen, the number of configuration possibilities at level L2 is much higher than level L3. Thanks to increasing stride representation and C++ templates, it is possible to limit the amount of code necessary to provide all the combinations.

The most mature interfaces available in GCL are

- `halo_exchange_dynamic_ut`: This is a class in which the data field to be exchanged are not known before the invocation point, but they all share the same halo-structure and element type ("`ut`" stands for *uniform type*). With this pattern, stack allocated data fields, whose address is known only when the stack-frame is allocated, can be used;
- `halo_exchange_generic`: This is the most general class. Each data-field is associated to a *descriptor* that contains the pointer to the data and the corresponding halo-structure, and is instantiated with the pointer to the data, the type of the data elements. Using this pattern we have complete flexibility in performing any kind of halo-exchange.

Figure 4 provides an example of instantiating and executing a halo-update operation on three data fields on GPU memory and halo widths equal in all directions. The first template argument to the `halo_exchange_dynamic_ut` is the data layout, then the mapping between data coordinates and process grid coordinates. Additionally we specify the type of data to be exchanged and the number of dimensions of the data fields. Those are program specific information. The next two arguments, which are not mandatory, specify platform properties, such as where the data resides and how the packing should be performed. A general feature of a C++ library is that it allows for simple experimentation of different combinations, so that the user can easily figure out what are the parameters that lead to better performance. The rest of the code is divided in two parts. The first pass to the pattern object the halo descriptors of the data fields that will be updated, followed by a call to `setup()`, which sets up the L3 pattern used inside. After the setup is done, a list of pointers to data fields are passed and exchanged. The exchange can happen in a split-phase by calling `start_exchange()` followed by a `wait()`. `pack(…)` and `unpack(…)` come in two flavors: they can accept a vector of pointers to data fields, or a comma separated list of pointers to data fields. Pack must be called before start sending, while unpack must be called after the wait;

`halo_exchange_generic`, on the other hand, is more flexible and allows the user to specify arbitrary data fields (value types, data layouts, and sizes) to be exchanged together. To do so GCL provide a `field_on_the_fly` class that is a descriptor of a specific data field, including value types, sizes (array of halo descriptors, and pointer to the data). The only parameters needed by the pattern object are the mapping of dimensions to process grid, the placement of the data, the number of dimensions, and packing/unpacking method. Figure 5 shows an example of `halo_exchange_generic` of three data fields. The pattern requires much less template arguments, but performance-wise there may be more overhead for not knowing sufficient information at instantiation time, as we discuss in the experimental results section.

```
typedef GCL::halo_exchange_dynamic_ut<GCL::layout_map<2,1,0>,
                                      GCL::layout_map<0,1,2>,
                                      double, 3,
                                      GCL::gcl_gpu,
                                      GCL::version_manual > pattern_type;

pattern_type he(pattern_type::grid_type::period_type(true, false, false), CartComm);

he.add_halo<0>(H, H, H, DIM1+H-1, DIM1+2*H); // Order of user coordinates
he.add_halo<1>(H, H, H, DIM2+H-1, DIM2+2*H);
he.add_halo<2>(H, H, H, DIM3+H-1, DIM3+2*H);

he.setup(3); // Maximum # of fields

he.pack(A_ptr_on_gpu, B_ptr_on_gpu, C_ptr_on_gpu);

he.exchange();

he.unpack(A_ptr_on_gpu, B_ptr_on_gpu, C_ptr_on_gpu);
```

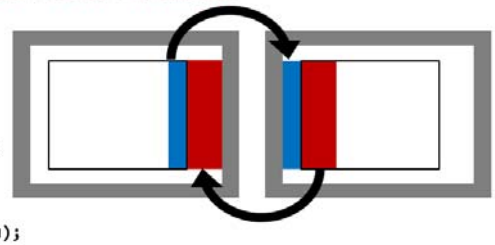

Figure 4: Example of `halo_exchange_dyinamic` exchanging three 3D fields, which is periodic in the first dimension and data reside on GPU memory, and packing in manual.

```
typedef halo_exchange_generic
            <layout_map<0,1,2>, 3, arch_type, packing_type >
            pattern_type;

pattern_type he(period_type(per0, per1, per2), CartComm);

field_on_the_fly<type1, layout1, pattern_type::traits> field1(ptr1, halo_dsc1);

field_on_the_fly<type2, layout2, pattern_type::traits> field1(ptr2, halo_dsc2);

field_on_the_fly<type3, layout3, pattern_type::traits> field1(ptr3, halo_dsc3);

he.pack(field1, field2, field3);

he.exchange();

he.unpack(field1, field2, field3);
```

Figure 5: Example of `halo_exchange_generic` of three data fields of potentially different data types, layout, and sizes.

5. Maintainability issues

By being a library of communication patterns, it follows that a single application may use only a small portion of the available features of GCL. This happens in many other cases, for instance the Boost C++ libraries are never used completely by a single user, or the MPI library, etc. This is a version of the "feature creep" problem in software development. The degree of reliability of a software product depends on the use of the features it provides (and the reactivity of the developing team). If some features are never used it is very likely that they are not robust, if not buggy, or inefficient. A typical example is provided by `MPI_Datatype`s in MPI, which are very powerful concepts but often lack an efficient implementation. So the reliability of the GCL depends on the users using it. Right now the two patterns described above are highly tested by multiple users, and it is reported to be reliable and fast.

We are evaluating what to develop next based on user requirements, since we do not want to produce features that would not be heavily tested *on the field*. It is important to note the, in the context of HPC, performance is part of correctness. The impact of a unique, generic and flexile interface, which would be appreciated in other application area, may be not welcome in HPC.

There is an issue in *code expansion*, that is, the number of specializations/versions of a concept/class to deal with certain specific parameters, like CPU and GPU implementations. This can lead easily to maintainability issues that may affect code readability and thus maintainability. Testing in a parallel environment is notoriously difficult since the number of combinations to be tested grow very quickly. A development team needs to be active on GCL and code reviews should be strictly performed, together with strict coding convention compliance. This is the main reason why GCL, although used by few users already, needs to be open-source and with more than one contributor. The GCL group is growing (three programmers at the moment) and we hope it will keep improving to provide better and better communication pattern implementations. We are anyway open to new contributors to keep GCL improve and be reliable.

GCL is almost completely header-only library, but supporting different communication mechanisms other than MPI could require increasing the runtime support needed by the library. This is also an issue in maintainability of the library, since this implies dealing with substantial amount of platform specific code.

6. Performance results

In this section we show some performance results on different architectures. It is important to note that in all the machines we run the same "user" code, while the implementation may differ for target specific options. We test GCL on the following architectures (in parentheses the dubbed names we use later to refer to the machines):

- CRAY XK7 (XK7): the machine uses CRAY Gemini interconnect to connect nodes with one AMD 2.1GHz Interlagos chip with 16 cores and one nVidia Kepler K20x GPU. GPU and host are connected through a PCI-X Gen2. We use here CRAY MPI version 6.0.0 which is G2G enabled, so that MPI messages can be exchanged directly from one GPU into another in another node.
- CRAY XE6 (XE6): the machine uses CRAY Gemini interconnect to connect nodes with two AMD 2.1GHz Interlagos chips to sum up 32 cores per node. We use here CRAY MPI version 5.6.0.
- CRAY XC30 (XC30): the machine uses Dragonfly network topology from the new CRAY CASCADE architecture. The nodes are dual socket 2.6GHz Intel Sandybrige and MPI version is CRAY MPI 5.6.5.

- IBM iDataPlex (iDP): a Infiniband QDR cluster whose nodes are dual 2.6GHz Intel Westmere chips (24 cores per node) plus two nVidia Fermi 2090 GPUs. We use here MVAPICH2/1.9 which is G2G enabled.
- Sandybridge+K20c cluster (K20c): a small Infiniband FDR cluster with 4 nodes made of dual 2.6GHz Intel Sandybridge chips (16 cores per node) and two nVidia Kepler K20c cards connected through PCI-X Gen3. We use here MVAPICH2/1.9 which is G2G enabled.

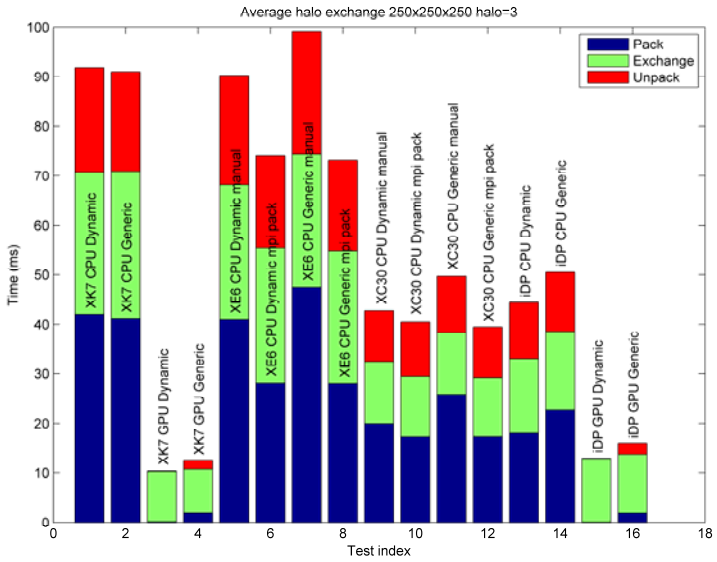

Figure 6: Comparison between XK7, XE6, XC30, and iDataPlex machines exchanging halos on three data fields of size 250^3 and 3 elements halos. We have here 16 MPI processes, 1 per node. As can be seen, GPU-to-GPU communication in modern MPI implementations leads to much better performance than regular MPI applications.

We ran weak-scaling benchmarks in which each MPI process owns three 250x250x250 3D data fields of doubles with halos of 3 elements all around and full periodicity on the axes. This stresses the communication infrastructure since each process sends 6 messages of 4.5MB along the faces, 12 messages of 55KB along the edges, and 8 messages of 648 Bytes along the corners.

Figure 8 shows the average execution times of such a pattern when running 16 processes, organized in a 4x2x2 computing grid, one per node. We ran experiments on both CPUs and GPUs when the latter is available. It is apparent that GPU versions are much faster that CPU ones. There are few reasons for this. The first is that packing and unpacking times are very small on GPUs, the second is the ability of modern MPI implementations to employ pipelining to perform data exchange from GPU memory to a remote GPU memory. The pipeline works intuitively as follows: a message is split into chunks and while sending a chunk to a remote node through the network interface, a new chunk is fetched from the GPU memory. Since GPU memory is pinned, the network interface driver can possibly use DMAs to perform these operations, thus minimizing the runtime overhead and virtually without losing bandwidth.

On non-GPU machines, namely XE6 and XC30, we show the impact of packing method. All the other uses manual, since it's the one that provide the best performance for GPU codes and we can then use the same user code to run these experiments on all GPU machines. On the other hand, we can see that MPI_Datatypes can potentially lead to better performance, even though the impact is not dramatic. The new network and the newer node architecture of XC30 leads to speed up communication of almost 2X with respect to XE6.

It is not trivial to extend these results to application performance, since this depends on how many patterns there are, how they can be interleaved, and if the MPI implementation can hide latency by effectively splitting communication or not. Anyway, we typically experience better or equal performance with respect to original applications in non-accelerated applications. It is difficult to measure the impact of accelerated applications since the ones using GCL were never available on GPUs before. As mentioned in the introduction, GCL was actually a requirement to port these applications to GPUs. To provide a intuition of how GCL can improve application development and performance, consider that there are 6 CUDA kernel to pack and 6 to unpack data and they are overall few hundreds of lines of code. This code is quite difficult to write and test, so having it embedded in the library is a great advantage to the application developer. Additionally, the use of a GPU-enabled MPI would

improve the application available bandwidth of about 3x because of the pipeline (with respect to copying the data out of the GPU into the host memory, then to the network, and then to the remote GPU memory).

With a similar spirit, Figure 9 shows a smaller configuration of 8 MPI processes to be able to compare with the small K20c cluster. In this case we run 2 MPI processes per node on iDP and K20c, while we compare XK7 with 1 process per node, since it has 1 GPU per node. We can notice that iDP is slower here with respect the 16 processes in Figure 8. This is due to increased congestion of the two processes per node. We can also observe that the increased performance of the node in K20c with respect to iDP leads to important performance benefits on the former with respect to the latter. XK7, even though using double the nodes is anyway very competitive when using GPUs (it has less congestion but may cost more when considering accounting for resources). In both Figure 8 and 9 we can see that generic pattern on GPUs exhibits higher packing/unpacking times. This is because the `field_on_the_fly` object must be copied into GPU memory before performing these operations, and this creates some dependencies. We should additionally note that the bars for GPUs can hide some packing/unpacking times, since there is no explicit synchronization between these operations and the exchange phase. Introducing these synchronizations would show cleaner and more stable times, but would also pollute the overall execution time, which in the plots is the accurate wall-clock times measured after the data have be safely placed at destination location.

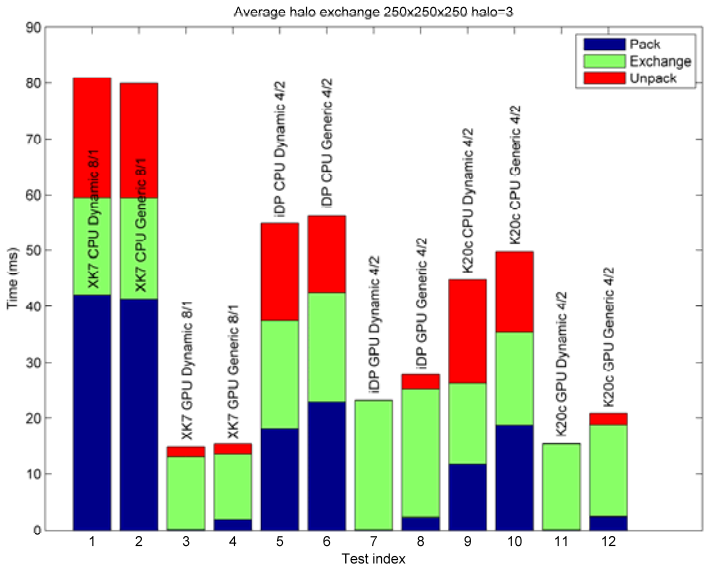

Figure 7: Comparison between XK7, iDataPlex, and a small cluster with K20c nVidia cards and FDR Infiniband network. The exchange is on three data fields of size 250^3 and 3 elements halos. We have here 8 MPI processes, 2 per node on iDataPlex and K20c, and 1 on XK7.

7. Conclusion and Future Work

In this paper we presented a generic C++ library for halo-update/halo-exchange patterns. This library is used in real applications and demonstrated to be useful in real cases. There are remaining challenges, such as maintainability issues and feature creeps that may arise in a project like this.

Although the performance of GCL is very good in benchmarks, the impact in real application depends on the structure of the application itself, so, while we always observe benefits by using GCL, it turns out that the available interfaces may be improved. We mentioned at the end of the last section that there are memory copies to GPU memory in case of the generic pattern. This could be avoided by having the pattern to be associated with a list of data `field_on_the_fly`s. This would lead to another interface, which is less generic than the `halo_exchange_generic`, and we are actually working on it. Another interface we are planning to develop is `halo_exchange_ut`, (“`ut`”stands for *uniform type*) in which sizes and pointers to data are known at pattern instantiation time and then can provide the best performance. As mentioned in Section 5, the importance of performance in HPC, makes the design of a single interface, most likely the generic one, unfeasible, since it would lead to sub-optimal performance that will lead application developers to implement their own solutions. It is generally true that the earlier information is available the better the performance could be. Generic pattern is, from this respect, intrinsically less efficient than the dynamic one. Nevertheless, the generic interface is required

in the effort of porting existing code. A new developed application should be designed to avoid late decisions as much as possible, but coping with legacy code is also an important feature to be considered.

More into the future we plan of implementing more L2 patterns, like broadcasts, scatters, gathers, and so on, the L1 layer for user defined data structures, and L3 layers based on PGAS runtime systems. The development of these will be dependent on the interest of the users in order to minimize the “feature creep” effect.

References

[1] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar. Modernizing the C++ interface to MPI. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, LNCS 4192, pages 266–274. Springer Berlin / Heidelberg, 2006.

[2] J. G. Siek, L. Lee, A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2001.

[3] Harshvardhan, A. Fidel, N. Amato, L. Rauchwerger. The stapl parallel graph library. In H. Kasahara and K. Kimura, editors, Languages and Compilers for Parallel Computing, LNCS 7760, pages 46-60. Springer Berlin Heidelberg, 2013.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.