CFD Proxy

The CFD proxy implements and evaluates the overlap efficiency for halo exchanges in unstructured meshes. The application reads the multigrid mesh levels of a graph-partitioned and preprocessed F6 airplane mesh (netcdf format). The mesh then is subsequently distributed to a given number of threads. Every thread will exclusively update the points which it owns.

Threading model

In passing we note that unstructured mesh implementations typically require indirect read/write access via the edges of the mesh to the actual mesh data. In thread based environments this indirect read/write access can implicate potential data races: When updating the mesh data, two or more threads (via the cross cutting edges) can perform the update at the same time.

In order to resolve this problem, we have adopted a threading strategy which allows read access only to points belonging to other thread domains and read/write access to the points the thread actually owns. The CFD proxy hence implements 3 different edge types: edges which reside entirely within the thread domain (and which are allowed to update both attached data points, ftype 1), edges which only write to the left data point (ftype 2) and edges which only write to the right data point (ftype 3). Effectively this implementation doubles the number of cross cutting edges between all neighboring threads -- both in terms of computational overhead and in terms of data volume. Nevertheless the implementation features very good scalability.

Overlapping computation with communication

The edges/faces of the distributed mesh are re-sorted according to whether or not they belong to the inner/outer mesh halo layers and according to their edge type. Edges of e.g. ftype 2 which In addition belong to the halo layer will be processed first, halo edges of ftype 3 will be processed next, halo edges of ftype 1 next, etc.

The actual CFD kernel we here implement is a green gauss gradient calculation with a subsequent halo exchange. In order to maximize scalar efficiency we have split the thread domains into sub domains (colors) which fit in the L2 cache. We perform strip mining on these colors both with respect to initialized and finalized points (for details see gradients.c).

For an efficient overlap of computation with communication we want to trigger the communication as early as possible. When preprocessing the mesh we hence mark up all finalized points per color which belong to the mesh halo. The thread which completes the final update (for a specific communication partner) on these halo points then triggers the corresponding communication – either via MPI_Isend, MPI_Put or gaspi_write_notify. We note that while this method allows for a maximal overlap of communication and computation, it either requires a full MPI_THREAD_MULTIPLE or a MPI_THREAD_SERIALIZED MPI version. For the latter version we have encapsulated the actual MPI_Isend and MPI_Put in an OpenMP critical section.

Implementations

We have implemented various MPI versions(both one-sided and two sided) and GASPI versions of this kernel.

- MPI bulk synchronous.
  This implements non-blocking MPI_Irecv, MPI_Isend, MPI_Waitall. Bulk synchronous indicates that communication is performed sequentially after computation.
- MPI early-receive
  This implements MPI bulk synchronous communication, where the MPI_Irecv is posted before the actual gradient computation starts. In principle this should avoid performance issues due to late receivers.
- MPI asynchronous
  This implements MPI early receive, but in an asynchronous manner. The MPI_Isend is issued a soon the respective communication buffer is complete.
- GASPI bulk synchronous
- *This implements GASPI bulk synchronous communication and leverages one-sided GASPI communication via the GASPI notify mechanism. Avoids the problem of late receivers due to the GASPI one-sided communication.*
- GASPI asynchronous
  Similar to MPI asynchronous. Uses gaspi_write_notify instead of MPI_Isend. Features almost zero computational overhead. For small numbers of rank/cores this implementation features the same performance as a theoretical communication free version, i.e. the GASPI asynchronous version allows for a complete overlap of communication and computation. For large numbers of ranks/cores this implementation scales very high – of order 100 points per thread domain are possible.
- MPI fenced-put bulk synchronous
  Implements MPI one sided communications fenced put operation. Does not scale very well due to scalability issues in the collective fence operation.
- MPI fenced-put asynchronous
  Same as MPI fenced put but as an asynchronous implementation with MPI_Put immediately after a final halo update (per communication partner)
- MPI pscw bulk synchronous
  Implements the MPI post start complete wait cycle in a bulk synchronous manner.
- MPI pscw asynchronous
  Same as MPI pscw bulk synchronous but in an asynchronous version. Similar to MPI_async or GASPI_async. This version calls MPI_Win_Complete after the last MPI_Put in order to enforce the start of the MPI one-sided communication.
- MPI flat model bulk synchronous
  Implements MPI bulk synchronous as a flat MPI (single threaded) model.
  MPI flat model asynchronous
  Implements MPI asynchronous as a flat MPI (single threaded) model.
- *Comm free gradients*
  Implements a hypothetical zero communication model. Represents the theoretical limit. Does not scale linearly due to the additional computational effort for computing halo mesh data and due to the load imbalance on the coarse multigrid levels (caused by mesh aggregation) Assumption of ideal network interface with infinite bandwidth and zero latency managed by

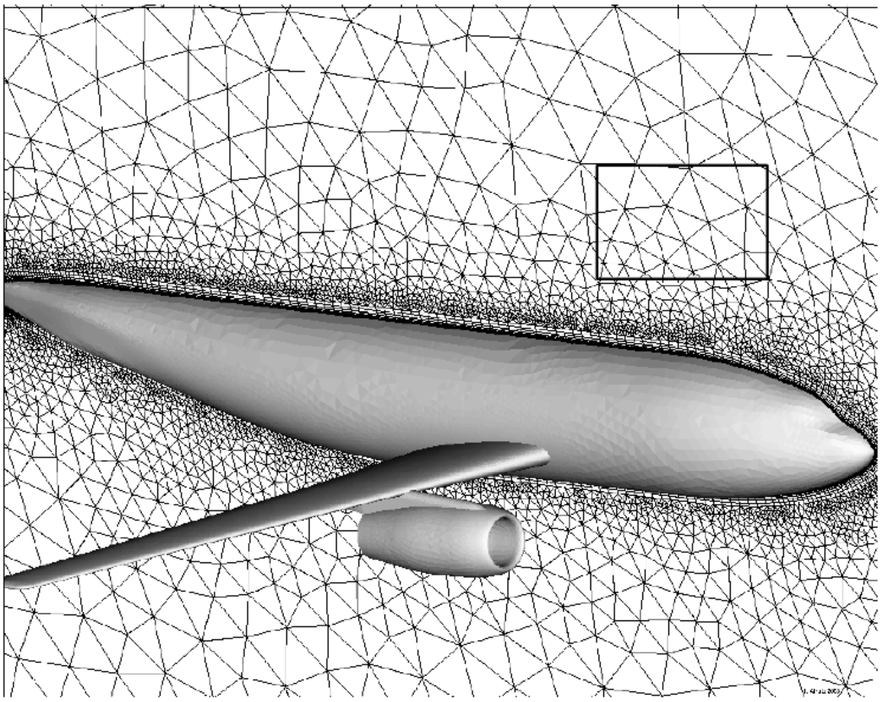
an ideal communication library producing zero overhead/interference from its runtime system. Used as baseline value in speedup plots (comm free gradients @ 144 cores = 1).

Results

We have evaluated the CFD proxy on a state of the art Infiniband FDR fat tree topology. The nodes are dual socket 12 core Ivy Bridge nodes. We have used all cores of a node in all implementations. For the flat MPI model we have used 24 ranks per node, for the hybrid OpenMP/MPI and OpenMP/GASPI implementation we have used 12 threads per rank and one rank per socket.

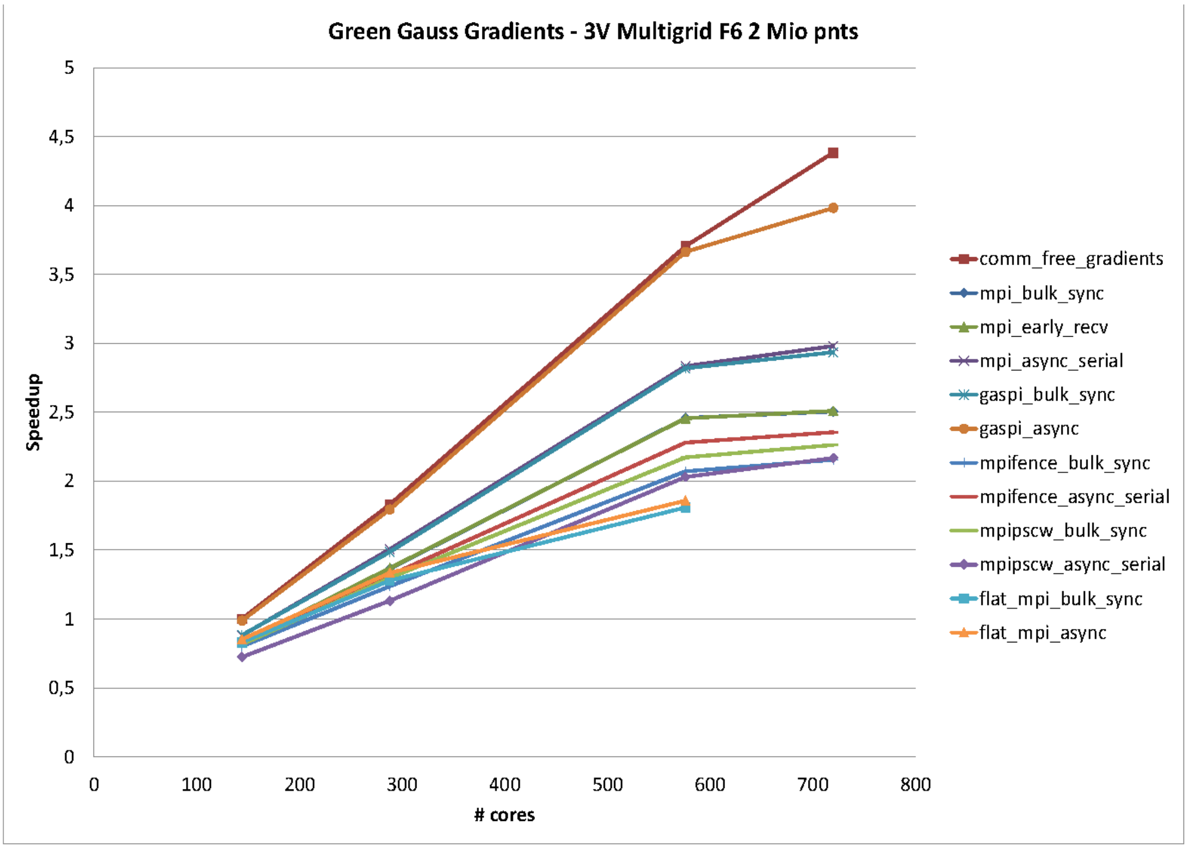
The CFD proxy in principle allows to test arbitrary multgrid cycles. We here consider a 3V multgrid cycle with 3 smoother steps per mesh level (2nd lvl smoothing both up/down) and present corresponding results for all implementations in Fig.1/2.

On average the number of mesh points shrinks by a factor of around 8 per multigrid mesh level. For the F6 mesh with 2 million points and 2304 compute cores we hence have around 868 mesh points per thread for the first (finest level) of the mesh, around 110 points per thread for the next coarser second level of the mesh and around 150 points per rank (12 points per thread) for the third level of the mesh.



DLR F6 mesh

The subsequent plots both show the same data, however the first plot exhibits the almost perfect overlap of gaspi_async for a low number of cores, whereas the second figure exhibts a substantially improved scalability of gaspi_async in comparison to the various MPI implementations. Speedup is measured vs comm_free_gradients @ 144 cores.



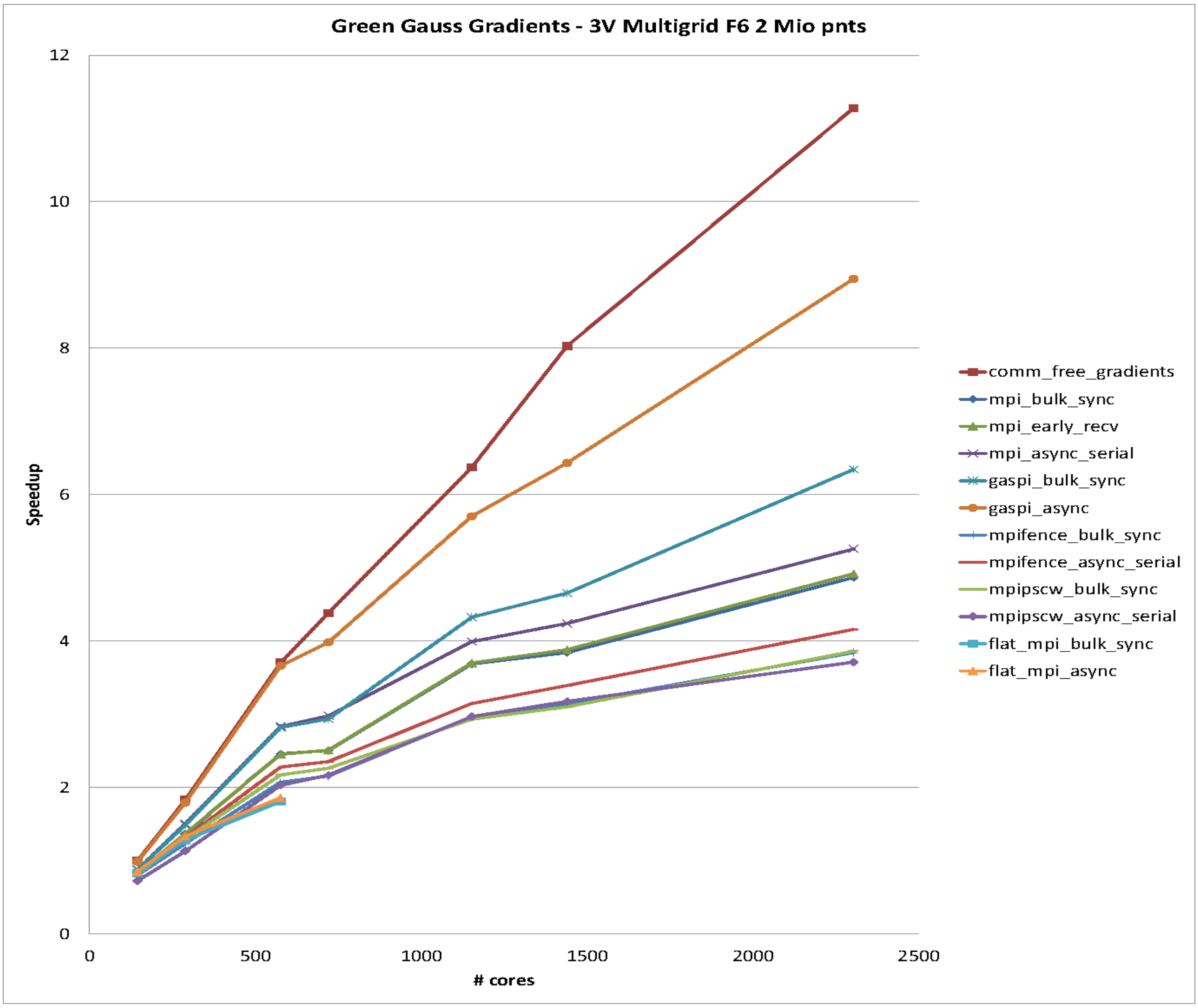
We note that the results vary slightly with different MPI implementations and setups (openmpi-1.6.5 vs intel mpi 5.0.1 vs mvapich2-2.1a vs thread_serialized vs. thread_multiple vs progress threads with or without dedicated communication cores).

We also acknowledge that it is quite possible that the Message Passinge Interface API and/or any other Communication API and/or Programming language potentially can yield better/more scalable results than what has been exhibited here – e.g. with other/better communication patterns. (MPI-3 passive target communication for example).

We encourage the readers of this document to improve this benchmark. You may change threading modell and communication pattern, data structures, point or face reordering, however you like, single threaded, multithreaded, in any programming language or API – as long as your algorithm / implementation performs the green gauss gradient reconstruction and a subsequent ghost cell exchange of the gradients everything is fine.

In any case we hope and suggest that you do some benchmarking on your own HPC infrastructure.

Ghost cell/halo exchanges are a very common use case in HPC. Providing implementations / recipes / blueprints which make optimal use of both the network capabilities and the corresponding API hence appear as a very worthwhile effort, especially since the potential implementation space has become rather large.



Community involvement

We encourage the HPC community to provide new patterns or improve existing ones. No restrictions apply for either communication API or programming languages used in these improved patterns. Bonus points are granted for highly scalable implementations which also feature better programmability.

We are happy to include all those patterns/implementations in this CFD Proxy Release.

Related Documentation

MPI

http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (MPI API)
http://www.open-mpi.org/ (MPI Release)
http://mvapich.cse.ohio-state.edu/news/https://computing.llnl.gov/tutorials/mpi/ (Tutorial)

GASPI

http://www.gpi-site.com/gpi2/gaspi/ (GASPI API)

https://github.com/cc-hpc-itwm/GPI-2 (GPI2 release, implements GASPI)

https://github.com/cc-hpc-itwm/GPI-2/tree/next/tutorial (Tutorial)