A PGAS-based implementation for the unstructured CFD solver TAU

Christian Simmendinger
T-Systems Solution for Research
Pfaffenwaldring 38-40
70569 Stuttgart, Germany
christian.simmendinger@t-systems.com

Jens Jägersküpper
German Aerospace Center(DLR)
Institute of Aerodynamics and Flow Technology
38108 Braunschweig, Germany
Jens.Jaegerskuepper@dlr.de

Rui Machado
Fraunhofer ITWM
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
rui.machado@itwm.fhg.de

Carsten Lojewski
Fraunhofer ITWM
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
lojewski@itwm.fhg.de

ABSTRACT

Whereas most applications in the realm of the partitioned global address space make use of PGAS languages we here demonstrate an implementation on top of a PGAS-API. In order to improve the scalability of the unstructured CFD solver TAU we have implemented an asynchronous communication strategy on top of the PGAS-API of GPI. We have replaced the bulk-synchronous two-sided MPI exchange with an asynchronous, RDMA-driven, one-sided communication pattern. We also have developed an asynchronous shared memory strategy for the TAU solver. We demonstrate that the corresponding implementation not only scales one order of magnitude higher than the original MPI implementation, but that it also outperforms the hybrid OpenMP/MPI programming model.

Keywords

Computational Fluid Dynamics, Communication libraries, Software for communication optimization, PGAS, GPI

1. INTRODUCTION

Over the last decade PGAS languages have emerged as an alternative programming model to MPI and promising candidates for better programmability and better efficiency ([12]). However, PGAS languages like Co-Array Fortran (CAF) [2] or Unified Parallel C (UPC) [12] require the re-implementation of large parts of the implemented algorithm. Additionally - in order to deliver superior scalability to MPI - languages like CAF or UPC require a specific data-layout, suitable for data-parallel access. PGAS languages also require an excellent compiler framework which is able to e.g. automatically schedule and trigger the overlap of communication and computation.

In the problem domain of CFD on unstructured grids, where data access is predominantly indirect and graph partitioning a science in itself [9], a compiler driven overlap of communication and computation poses quite a challenge. While waiting for a solution from the compiler folks we have opted for a more direct approach in the meantime.

In this work we present a parallel implementation of the CFD solver TAU [11] on top of a PGAS-API. Similar to MPI this PGAS-API comes in the form of a communication library. Our approach hence does not require a complete rewrite of applications. However – in order to achieve better scalability than MPI – the implementation requires a rethinking and re-formulation of the communication strategy. We have developed an asynchronous shared memory implementation for the TAU solver which scales very well and readily overlaps communication with computation. In addition we have replaced the bulk-synchronous two-sided MPI exchange with an asynchronous, one-sided communication pattern, which makes use of the PGAS space of the GPI-API [8]. We demonstrate that the corresponding implementation not only scales one order of magnitude higher than the original MPI implementation, but that it also outperforms the hybrid OpenMP/MPI programming model. For a production sized mesh of 40 million points we estimate that with this implementation the TAU solver will scale to about 28,800 x86 cores or correspondingly to 37 Double Precision Tflop/sec of sustained application performance.

2. THE TAU SOLVER

In this work we closely examine the strong scalability of the unstructured CFD solver TAU, which is developed by the German Aerospace Center(DLR) [6] and represents one of

the main building blocks for flight-physics within the European Aerospace eco-system. The TAU-Code is a finite-volume RANS solver which works on unstructured grids. The governing equations are solved on a dual grid, which, together with an edge-based data structure, allows to run the code on any type of cells.

The code is composed of independent modules: Grid partitioner, preprocessing module, flow solver and grid adaptation module. The grid partitioner and the preprocessing are decoupled from the solver in order to allow grid partitioning and calculation of metrics on a different platform from the one used by the solver.

The flow solver is a three-dimensional finite volume scheme for solving the Unsteady Reynolds-Averaged Navier-Stokes equations. In this work we have used the multi-step Runge-Kutta scheme of TAU. The inviscid fluxes are calculated employing a central method with scalar dissipation. The viscous fluxes are discretized using central differences. We have used a 1-equation turbulence model (Spalart/Allmaras). In order to accelerate the convergence to steady state, local-time stepping and a multigrid technique based on agglomeration of the dual-grid volumes are employed.

2.1 Subdomains in TAU: colors

In order to achieve good node-local performance on cache architectures TAU perfoms a partitioning of every MPI domain (which in turn are obtained by means of a domain decomposition) into small subdomains (colors) [1]. These subdomains typically contain of order 100 mesh points. With a memory footprint of 1KByte per mesh point a color typically fits in the L2 cache. The points within these domains, as well as the connecting faces are renumbered in a preprocessing step in order to maximize spatial data locality. Temporal data locality is achieved by consecutively iterating over the colors until the entire MPI domain has been processed.

Due to this implemented coloring TAU typically runs with around $12-25\%$ of the double precision peak performance of a standard dual-socket x86 node – which is quite remarkable for the problem domain of CFD. However - the performance of TAU depends on the used solver type, the structure of the mesh and the specific use case.

In Table 1 we consider three different cases on a single node: M6, F6 and Eurolift. The smallest use case M6 with 100,000 points has a total memory footprint of 100MB. With this memory footprint the solver does not fit in the 12MB L3 cache of the used 2.93 Ghz dual socket 6-core Intel Westmere node. Nevertheless M6 shows the best performance of the three considered use cases. The F6 use case with 2 million points represents a mesh size per node, which is common for production runs. This use case delivers good performance as well. Lastly for the Eurolift 13.6 million points the performance degrades substantially, even though 13.15% of peak performance is still a good value for an unstructured CFD solver. However, we note that – mainly due to the correspondingly long run times – such large use cases are rarely used on a single node.

3. THE GPI API

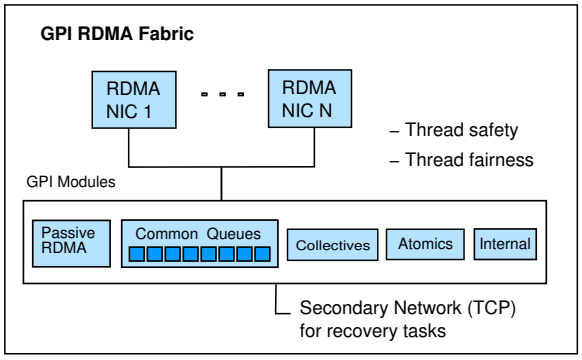

Figure 1: GPI modules

GPI (Global address space Programming Interface) is a PGAS API for C/C++ and Fortran applications. It focuses on asynchronous, one-sided communication as provided by modern, RDMA-enabled interconnects such as Infiniband.

The GPI API has two main objectives. Firstly, maximum communication performance by exploiting directly the network interconnect and minimizing the communication overhead by allowing a full communication and computation overlap (zero-copy). The second objective is to provide a simple API for the development of parallel applications based on more asynchrounous algorithms and an easy to learn programming model.

GPI is strictly a PGAS API with some similarities to (Open)Shmem. In contrast to the PGAS languages of CAF or UPC,tthe GPI API hence neither requires extensive modifications of the source code nor an efficient underlying compiler framework. What is however required, is a thorough rethinking of the communication patterns of the application and a reformulation of bulk-synchronous data-exchange towards an asynchronous model. The Global Arrays framework is not really applicable to unstructured meshes, so we have not considered it here.

3.1 Functionality of GPI

The figure 1 depicts the GPI modules organization. The depicted modules of GPI interact directly with the interconnect (one or more RDMA NICs) and expose the whole GPI functionality.

The GPI functionality is therefore resumed in five modules: passive communication (Passive RDMA), global communication through queues (Common Queues), collective operations (Collectives), atomic counters (Atomics) and utilities and internal functionality (Internal).

Althought GPI focuses on one-sided communication, the passive communication has a Send/Receive semantic that is, there is always a sender node with a matching receiver node. The only difference is that, and therefore the passive term, the receiver node does not specify the sender node for the receive operation and expects a message from any node. Moreover, the receiver waits for an incoming message in a passive form, not consuming CPU cycles. On the sender side, the more active side of the communication as it speci-

Table 1: Node-Local performance of TAU

Use Case	Mesh Points	Gflop/sec/node (Double Precision)	% of Peak (Double Precision)
M6	100,000	15.95	22.6
F6	2 million	12.39	17.6
Eurolift	13,6 million	9.25	13.15

fies the receiver, the call is nevertheless non-blocking. The sender node sends the message and can continue its work, while the message gets processed by the network card.

Global communication allows asynchrounous, one-sided communication primitives on the partitioned global address space. Two operations exist to read and write from global memory independent of whether it is a local or remote location. One important point is that each communication request is queued and those operations are non-blocking, allowing the program to continue its execution and hence a better use of CPU cycles while the interconnect independently processes the queues. There are several queues for requests which can be used for example, for different kinds of communication requests. If the application needs to make sure the data was transfered (read or write), it needs to call a wait operation on one queue that blocks until the transfer is finished and asserting that the data is usable.

In addition to the communication routines, GPI provides global atomic counters, i. e. integral types that can be manipulated through atomic functions. These atomic functions are guaranteed to execute from start to end without fear of preemption causing corruption. There are two basic operations on atomic counters: fetch and add and fetch, compare and swap. The counters can be used as global shared variables used to synchronize nodes or events.

Finally, the internal module includes utility functionality such as getting the rank of one node or the total number of nodes started by the application. Besides those API primivites, it implements internal functions such as the start-up of an application.

3.2 Programming and execution model

The GPI programming model implements a SPMD-like model where the same program is started by the nodes of a cluster system and where the application distinguishes nodes through the concept of ranks.

The application is started by the GPI daemon. The GPI daemon is a program that is running on all nodes of a cluster, waiting for requests to start an application. The GPI daemon interacts with the batch system to know which nodes are assigned to the parallel job and starts the application on those node. The GPI daemon also performs checks on the status of the cluster nodes such as Infiniband connectivity.

To start a GPI application, the user must define the size of the GPI global memory partition. If there are enough resources available and the user has permissions for that, the same size will be reserved (pinned) for the GPI application on all nodes. The global memory is afterwards available to the application through its lifetime and all threads on all nodes have direct access to this space via communication primitives.

The communication primitives act therefore directly on the GPI global memory, without any memory allocation or memory pinning. The communication primitives accept a tuple (offset, node) to steer the communication and control exactly the data locality.

The application should place its data on the GPI global memory space, making it globally available to all threads. Having the data readly available, applications should focus on algorithms that are more asynchronous and possibly leaving behing the implicit synchronization imposed by a two-sided communication model.

4. APPROACH AND IMPLEMENTATION

The parallelization of the flow solver is based on a domain decomposition of the computational grid. In the work presented we have used the CHACO toolkit ([7]). The CHACO toolkit manages to deliver well balanced partitions even for a very large number of nodes. We have used the multi-level Kernighan-Lin [5] methods of CHACO.

4.1 The asynchronous shared memory implementation

As briefly explained in the introduction TAU makes use of a pool of colors (subdomains in a MPI domain). The implemented asynchronous shared memory parallelization uses the available pool of colors as a work pool. A thread pool then loops over all colors until the entire MPI domain is complete [4].

Unfortunately, this methodology is not as trivial as it seems. About 50% of the solver loops run over the faces of the dual mesh (the edges of the original mesh) and update the attached mesh points. Updates to points attached to cross-faces (faces where the attached mesh points belong to different colors) then can lead to race conditions, if the attached points are updated by more than a single thread at the same time. We have solved this issue by introducing the concept of mutual exclusion and mutual completion of colors. In this concept a thread needs to mutually exclude all neighbouring colors from being simultaneously processed. To that end all threads loop over the entire set of colors and check their status. If a color needs to be processed, the respective thread attempts to lock the color. If successful, the thread checks the neighbouring color locks. If no neighbouring color is locked, the thread computes the locked color and subsequently releases the lock. The lock guarantess that no neighbouring color can be processed during that time frame, since all threads evaluate the locks of the respective neighboring colors prior to processing [4].

While the concept of mutual exclusion guarantees a race-condition free implementation, the concept of mutual com-

pletion provides the basis for an asynchronous operation. We have resolved the Amdahl Trap of the OpenMP fork-join model by complementing the above concept of mutual exclusion with a concept of mutual completion. Wether or not a color can be processed in this concept now not only depends on the locks of the neighboring colors, but it also depends on the local state as well as the state of the neighbouring colors. The threads carry a thread local stage counter which is incremented whenever all colors have been processed in one of the color loops of the code. The colors in turn carry a stage counter which is increased every time a color is processed. All loops in TAU have been reimplemented as loops over colors. This holds true not only for the face loops (which run over the faces of a the dual mesh), but also for the point loops. A color in a point loop which follows an face loop hence can only be processed if the color itself still needs to be processed and additionally all neighbouring colors to this color have been updated in the previous face loop. In this implementation there neither is a global synchronization point nor an aggregation of load imbalances at the synchronization point. Instead we always have local dependencies on states and locks of neighbouring colors. We are confident that this concept will scale to a very high core number. Wether it actually scales to e.g the 50 cores of the Intel MIC architecture remains to be seen.

This data-flow model is somewhat similar to the data-flow model of StarSs [10], however, in our case local data dependencies (neighbouring colors) are bound to the data structures and set up during an initalization phase. In contrast to StarSs a dedicated runtime environment is not required. Also StarSs does not feature mutual exclusion and hence would not be applicable here.

The threads in this implementation are started up once per iteration - there is a single OpenMP directive in the code.

4.2 Overlap of communication and computation

The asynchronous shared memory implementation of TAU has been extended with an overlap of communication and computation. The first thread, which reaches the communication phase, locks all colors which are attached to the MPI Halo (see Fig. 7), performes the halo exchange and subsequently unlocks the colors attached to the MPI Halo. All remaining threads compute (non-locked) non-halo colors until the halo colors are unlocked again by the first thread. Since the lock/relase mechanisms for the colors already were in place, the overlap of communication and computation was a logical extension of the principle of mutual exclusion of colors. In the benchmarks section we compare the performance of the hybrid OpenMP/MPI implementation against the hybrid OpenMP/GPI implementation. We emphasize that the only difference between these two implementations is the communication pattern of the next-neighbour halo-exchange. In the OpenMP/GPI Implementation the communication pattern is based on asynchronous one-sided RDMA writes, whereas the OpenMP/MPI is implemented as state-of-the-art MPI_Irecv, MPI_Isend over all neighbours with a subsequent single MPI_Waitall.

4.3 GPI implementation

In order to maintain the asynchronous methodology also on a global (inter-node) level, the GPI implementation targets the removal of the implizit synchronization due to the MPI handshake in the 2-sided MPI communication. In the GPI implementation of TAU all sends hence are 1-sided asynchronous communication calls and the sender immediately returns. Since the sender does not know anything about the receiver state, we have implemented a stage counter for the communication: In TAU all communication is performed in SPMD fashion. All processes on all nodes run through the same halo exchange routines in exactly the same sequence. By providing rotating receiver and sender frames and incrementing stage stage counters for send and corresponding receive we can guarantee a correct execution of the solver. A negative side effect of this asynchronous methodology is, that we can not write directly via RDMA into a remote receiver halo, since the sender does not know the state of the remote receiver.

While assembling the message from the unstructured mesh, we compute a checksum and write this checksum into the header of our message (together with the stage counter, sender id etc.). The receiver side evaluates the stage counter and sender ID. If these match, the receiving thread assembles the receive halo of the unstructured mesh and - while doing this - computes the checksum. If computed checksum and header checksum coincide and all outstanding receives are complete, the receiver returns. We note that assembling a complete receive halo at the earliest possible stage is of crucial importance for the scalability of the solver. The reason for this is, that for very small local mesh sizes, most colors will have at least one point in the halo. Threads which compute non-halo colors then quickly run out of work and are stalled until the communication thread (or threads) unlocks the halo colors.

For this reason we have separated the colors into send and receive halo colors in the preprocessing step. In the GPI implementation the last of the sender threads immediately unlocks the send halo, as soon as the sender frames have been assembled and sent. The sender threads then join the thread pool, which in turn then can process colors from the send halo in addition to non-halo colors.

Sends are usually mulithreaded, where the number of threads can be dynamic (e.g. depending on the current level of the multigrid). The receiving threads are assigned simultaneously and in principle also can be multithreaded. For the 6 cores/socket benchmarks however we have assigned 3 sender threads from the thread pool and a single receiver thread.

4.4 Modifications in GPI

Even though more than 95% of the communication in TAU is next-neighbour communication, the TAU solver requires a collective reduction operation for monitoring and the global residual (called once per iteration).

The collectives module of the GPI implements the global barrier and the Allreduce collective operations as this is one of the most frequently used collective operations. Although GPI aims at more asynchronous implementations and is diverging from a bulk-synchronous processing, some global synchronization points might be required and therefore the

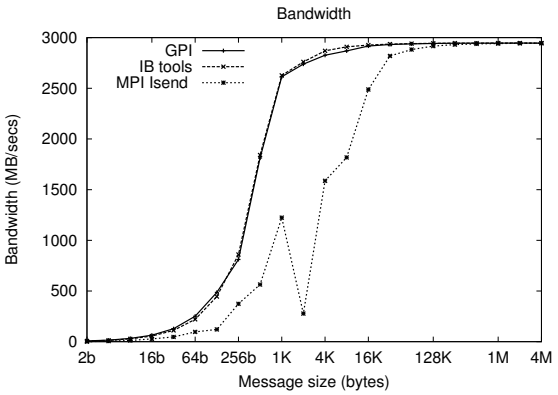

Figure 2: Bandwidth

need of a scalable barrier implementation.

The same justifies the implementation of the AllReduce collective operation, that supports typical reduce operations such as minimum, maximum and sum of standard types.

The Allreduce collective operation has been developed in the context of this work.

5. BENCHMARKS

The Allreduce micro benchmarks were produced with Mellanox ConnectX IB DDR and a central fully non-blocking fat-tree SUN Magnum switch. All other benchmarks were performed with the Mellanox MT26428 Infiniband adapter (ConnectX IB QDR, PCIe 2.0). The actual TAU application benchmarks were produced on the $C^2A^2S^2E$ system in Braunschweig [3], which has recently been upgraded to dual socket 2.93Ghz, 6-core Intel Westmere blades. The nodes feature the afore mentioned Mellanox MT26428 ConnectX DDR adapter and the central infiniband switch is a QDR switch from Voltaire with a blocking factor of two.

5.1 GPI micro benchmarks

In this section, we present some results on micro-benchmarks obtained with GPI when compared with the Infiniband performance tools and MPI.

The figure 2 presents the obtained bandwidth with GPI, the Infiniband performance tools (IB tools) and the MPI ISend operation.

The GPI performance follows the maximum obtained performance from the low level Infiniband performance tools and is very close to the maximum bandwidth already at 4KB message sizes. The MPI ISend operation also reaches maximum bandwidth but only at larger messages sizes. Although not noticeable on Figure 2, for small message sizes (up to 512 bytes), GPI always reaches a better bandwidth with over a factor of 2 when compared with the MPI ISend operation. These results aim at showing that GPI exploits directly the maximum performance (wire-speed) made available by the interconnect (Infiniband).

Figure 3 depicts the time needed (in micro-seconds) by the

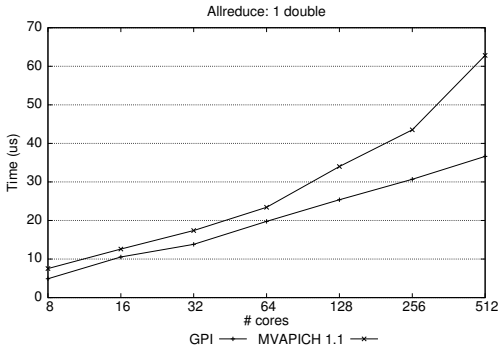

Figure 3: Allreduce operation on a double

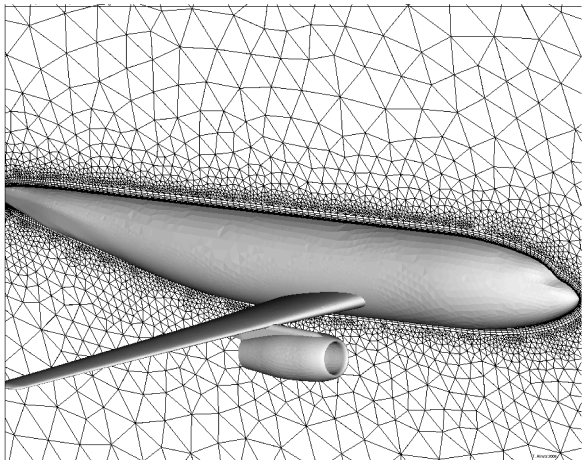

Figure 4: F6 use case

Allreduce operation and its scalability as we increase the number of nodes. The GPI operation is compared with the MVAPICH 1.1 MPI implementation.

The performance of the GPI Allreduce operation scales very well up to the maximum number of cores whereas the MPI version starts decreasing its scalability after 64 cores. The results demonstrate that the GPI operation yields a better scaling behaviour with an increasing number of cores.

5.2 TAU benchmarks

All presented results have been produced in the same run with an identical node set. In order to eliminate system jitter, which can arise due to the blocking factor of two, we have taken the minimum time for a single iteration out of 1000 iterations. For a 4W multigrid-cycle and our settings, this single iteration requires 140 next-neighbour halo exchanges with 15-20 local neighbours and a single Allreduce, which in turn implies that the TAU solver features close to perfect weak scalability.

The figures in 5 and 6 sum up the main results of this work. The three different lines represent the original MPI implementation, the OpenMP/MPI implementation, (run on a per-socket basis) and the OpenMP/GPI version, also run on a per-socket basis. We have also evaluated both OpenMP/MPI and OpenMP/GPI with the node as a basis

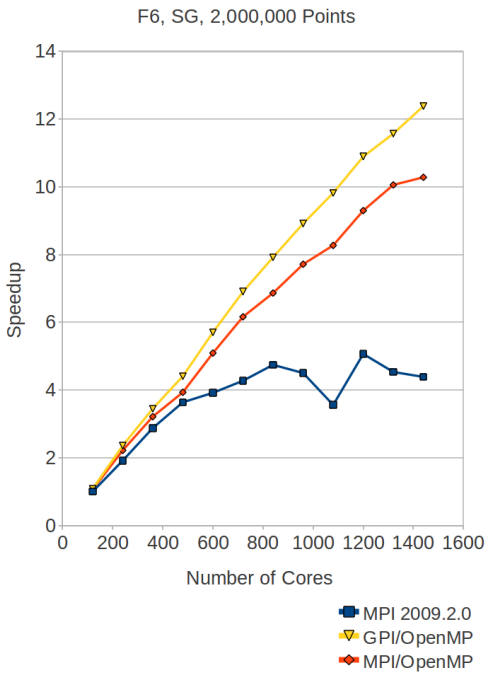

Figure 5: F6, Single Grid, 2 million Points

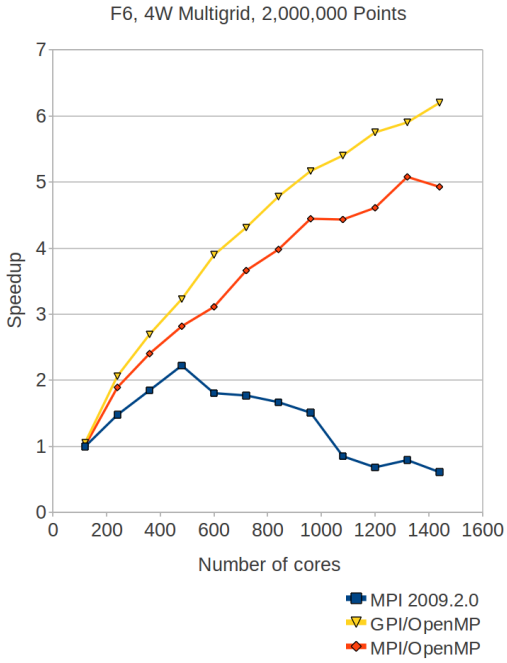

Figure 6: F6, 4W Multigrid, 2 million Points

for the shared memory implementaion. This version however suffers severly from cc-numa issues and we have refrained from using it. Since we expect future hardware developments to increase the number of cores per die rather than the number of sockets per node, we currently consider this a minor issue.

The depicted speedup is relativ to the MPI results for 10 nodes (120 cores). Until this point OpenMP/MPI as well as OpenMP/GPI scale almost linearly. At 240 cores (i.e. 20 nodes), the Tau Code has a local mesh size of around 100,000 points. Since both the node-local mesh size as well as the used solver type coincides with the M6 use case, we estimate the performance at this point to be about 20 nodes x 15.95 Gflop/sec = 319 Gflop/sec.

In order to test the strong scalability of our implementation, we have chosen a very small benchmark mesh with just 2 million meshpoints. With close to linear weak scaling, the strong scaling result implies that for a typical production mesh with 40-50 million mesh points, TAU should scale to around 28,800 x86 cores (or correspondingly to 37 double precision Tflop/sec of sustained application performance).

From this point on the observed curves are the aggregation of four effects: The increase of the workload due to the replication of send and receive halo, the node local load imbalance, the global load imbalance, and last but not least cache effects. We will briefly examine these four effects in a bit more detail:

5.2.1 Increased workload

Whenever we split a domain, we introduce additional faces and points (see Fig. 7).

In our case, due to the employed multigrid this effect can become quite dramatic: For 1440 cores we have 240 domains (1 domain per 6-core westmere). This corresponds to 2 million points / 240 = 8333 points per domain, which still seems reasonable. However - due to the agglomeration of the multigrid there are only about 60 points in a coarse level domain - and an additonal 120 points in the receive halo of this domain. The amount of work on the coarse multigrid levels thus increases substantially with an increasing socket count.

5.2.2 Node local load imbalance

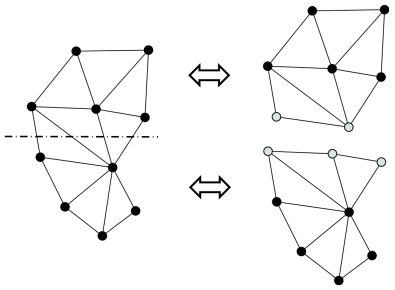

Figure 7: Splitting a domain: The introduced gray points and the correspondingly attached faces imply additional work for the solver. The gray points correspond to the recv halo, whereas the points connected to them correspond to the send halo.

While the asynchronous shared memory implementation is able to handle load imbalance extremely well (compared to e.g a loop parallelization with OpenMP), there are still some limits: In the above case of 60 inner points per domain and 120 points in the receive halo, all inner points probably will belong to the send halo. There is thus no possible overlap of communication and computation. All threads are stalled until the send frames have been assembled and again are stalled until the receive halo has been build.

5.2.3 Cache effects

The TAU solver already runs at good scalar efficiency at 240 cores with a node-local mesh size of 100,000 points. However - this mesh still does not fit into the cache entirely. There is a pronounced superlinear speedup at 480 cores in the curve, where large parts of the solver start to operate entirely in L3 cache.

5.2.4 Global load imbalance

While the CHACO toolkit does an excellent job in providing a load balanced mesh, small load imbalances are showing up on the coarser levels of the multigrid. In the asynchronous GPI implementation we can hide most of that imbalance: If a specific process is slower in a computational part A between two halo exchanges, it might be faster in a following part B. An asynchronous implementation is able to hide this load imbalance, provided the total runtime of A+B is equal to the corresponding runtimes A+B at other processes. We emphasize, that hiding this global load imbalance is only possible with an implementation which both can overlap the halo exchange between part A and B with computation and which also provides an asynchronous halo exchange. The hidden load imbalance shows up in the relative smoothness of the speedup curve in the OpenMP/GPI results.

6. CONCLUSION AND FUTURE WORK

We have presented a highly efficient and highly scalable implementation of a CFD solver on top of a PGAS API. The combination of both asynchronous shared memory implementation and global asynchronous communication leads to excellent scalability - even for a 4w Multigrid with 2 million mesh points / 1440 cores = 1388 mesh points/core. We were able to use key elements of the highly efficient scalar implementation (the colors) and reshape this concept towards an equally efficient but also highly scalable shared memory implementation.

We have extended the node-local asynchronous methodology towards a global asynchronous operation and thus were able to hide most of the global load imbalance arising from the domain decomposition. The GPI implementation features a dynamic assignment of the number of send and receive threads. We believe that this will be of strong relevance for future multi- or manycore architectures like the Intel MIC, since it will dramatically reduce both polling overhead for outstanding receives (on the fine mesh multigrid level) as well as a multithreaded simultaneous 1-sided send to all neighbours in the domain decomposition (on the coarse multigrid levels).

6.1 Future work

In a Partitioned Global Address Space every thread can access the entire global memory of the application at any given point in time. We believe that this programming model not only provides opportunities for better programmability, but that it also has a great potential to deliver higher scalability than is available in the scope of message passing. For the future we envision a new standard for a PGAS-APl which should enable application programmers to reshape and reimplement their bulk-synchronous communication patterns towards an asynchronous model with a very high degree of overlap of communication and computation.

7. ACKNOWLEDGMENTS

This work has been funded partly by the german Federal Ministry of Education and Research within the national research project HI-CFD.

8. REFERENCES

[1] T. Alrutz, C. Simmendinger, and T. Gerhold. Efficiency enhancement of an unstructured CFD-Code on distributed computing systems. In *Parallel Computational Fluid Dynamics 2009*, 2009.

[2] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. *Languages and Compilers for Parallel Computing*, pages 177–193, 2004.

[3] DLR Institute of Aerodynamics and Flow Technology. Center of Computer Applications in Aerospace Science and Engineering, 2007.

[4] J. Jägersküpper and C. Simmendinger. A Novel Shared-Memory Thread-Pool Implementation for Hybrid Parallel CFD Solvers. In *Euro-Par 2011, to appear*, 2011.

[5] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[6] N. Kroll, T. Gerhold, S. Melber, R. Heinrich, T. Schwarz, and B. Schöning. Parallel Large Scale Computations for Aerodynamic Aircraft Design with the German CFD System MEGAFLOW. In *Proceedings of Parallel CFD 2001*, 2001.

[7] R. Leland and B. Hendrickson. A Multilevel Algorithm for Partitioning Graphs. *Proc. Supercomputing '95.*, 1995.

[8] C. Lojewski and R. Machado. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science - Research and Development*, 23(3-4):125–132, 2009.

[9] J. Paz. Evaluation of Parallel Domain Decomposition Algorithms. In *1.st national computer science encounter , workshop of distributed and parallel systems.* Citeseer, 1997.

[10] J. Planas, R. M. Badia, E. . Ayguadé, and J. Labarta. Hierarchical task based programming with StarSs. *International Journal of High Performance Computing Applications*, 23:284 – 299, 2009.

[11] D. Schwamborn, T. Gerhold, and R. Heinrich. The DLR TAU-code: Recent applications in research and industry. In *European conference on computational fluid dynamics, ECCOMAS CFD*, pages 1–25. Citeseer, 2006.

[12] K. Yelick. Beyond UPC. *Proceedings of the Third Conference on Partitioned Global Address Spaces*, 2009.