A Novel Shared-Memory Thread-Pool Implementation for Hybrid Parallel CFD Solvers

Jens Jägersküpper[1] and Christian Simmendinger[2]

[1] German Aerospace Center (DLR)
Institute of Aerodynamics and Flow Technology
Center of Computer Applications in Aerospace Science and Engineering ($C^2A^2S^2E$)
38108 Braunschweig, Germany
Jens.Jaegerskuepper@DLR.de
[2] T-Systems Solution for Research (SfR)
Pfaffenwaldring 38–40,
70569 Stuttgart, Germany

Abstract. The Computational Fluid Dynamics (CFD) solver TAU for unstructured grids is widely used in the European aerospace industry. TAU runs on High-Performance Computing (HPC) clusters with several thousands of cores using MPI-based domain decomposition. In order to make more efficient use of current multi-core CPUs and to prepare TAU for the many-core era, a shared-memory parallelization has been added to one of TAU's solver to obtain a hybrid parallelization: MPI-based domain decomposition plus multi-threaded processing of a domain.

For the edge-based solver considered, a simple loop-based approach via OpenMP FOR directives would – due to the Amdahl trap – not deliver the required speed-up. A more sophisticated, thread-pool-based shared-memory parallelization has been developed which allows for a relaxed thread synchronization with automatic and dynamic load balancing.

In this paper we describe the concept behind this shared-memory parallelization, we explain how the multi-threaded computation of a domain works. Some details of its implementation in TAU as well as some first performance results are presented. We emphasize that the concept is not TAU-specific. Actually, this design pattern appears to be very generic and may well be applied to other grid/mesh/graph-based codes.

1 Intro

The TAU code, which is developed at the Institute of Aerodynamics and Flow Technology of the German Aerospace Center (DLR), is widely used in the European aerospace industry for Computational Fluid Dynamics (CFD), c. f. e. g. [6]. The solver is designed for unstructured grids, yet it may also be used with (block) structured grids. MPI-based domain decomposition allows TAU to be run on HPC clusters withseveral thousands of cores. To make more efficient use of current multi-core CPUs and to prepare TAU for the many-core era a shared-memory parallelization has been implemented for one of the TAU solvers. That solver implements an explicit Runge/Kutta scheme with geometric multigrid acceleration for unstructured grids.

Several approaches for a shared-memory parallelization for TAU have been evaluated, and finally, a novel solution following the thread-pool model has been developed. This thread-pool model allows the concurrent processing of tasks subject to dependencies among the tasks. In addition to temporal data dependencies and the concept of mutual completion of processed data, this solution incorporates also mutual exclusion among tasks to prevent data races. The concept allows for a significantly relaxed thread synchronization compared to bulk-synchronous models. It features an automatic load balancing and allows to implement a straightforward overlap of communication and computation in TAU. The implementation shows a very good performance for the TAU solver. However, the used methodology is not specific to the TAU solver and should be applicable to a wide range of programs that work on unstructured or structured grids/meshes/graphs.

1.1 Motivation

Due to the electric capacity of a CPU chip, the higher the clock frequency, the higher the voltage needs to be – and the higher the thermal power to be dealt with. Since an effective solution of this problem is not available, clock rates no longer increase. Moore's law, however, is still valid: the number of transistors per chip doubles roughly every one and a half years. The CPU manufacturer's policy to further increase the theoretical peak performance of their chips: multi-core chips. Though we see an exponential growth in explicit parallelism, this change in the hardware is not at all reflected in HPC software: With a moderate number of cores per socket there has been simply no need to adapt the parallelization. Due to the large number of cores per socket in modern CPU designs, however, the picture is changing: One MPI process per core has become a problem. For CFD, the more MPI processes are used, the smaller the average computational load per MPI process and the more MPI communication is needed to synchronize the flow variables across the processes. The time for this synchronization is to be considered serial. Hence, simply by Amdahl's law, there is a maximum number of MPI processes (and hence cores) that can be reasonably used.

Shared-memory parallel (multi-threaded) computation of the domains suggests itself as a possible way to increase the scalability. When all cores of a CPU are used to process one domain, the number of domains drops from the number of cores to the corresponding number of sockets. Even though the idea of such a hybrid (2-level) parallelization is straightforward, this approach requires the shared-memory parallel computation of the domains to be sufficiently efficient. It turns out that this is quite a challenge for CFD on unstructured grids.

1.2 Outline

In the following sections we describe how the proposed shared-memory parallel computation of the (MPI-) domains works and give some implementation details and first performance results. We start with a brief introduction to the TAU code in Sec. 2. In particular, the original MPI parallelization of TAU via domain decomposition is explained. In Sec. 3 we describe the concept of the shared-memory implementation. As the tasks to be processed by the threads show not only temporal dependencies, but also data dependencies, a simple loop-based parallelization via OpenMP is precluded. The tasks hence are processed asynchronously using mutual exclusion to prevent data races (asynchronous subject to the dependencies between the tasks). To efficiently handle temporal dependencies we employ the concept of mutual completion.

The relaxed synchronization of the threads improves the scaling considerably: The accumulation of load imbalances, which occur at every synchronization point in the multi-threaded program flow, is drastically reduced since global synchronization is replaced by dynamic local synchronization.

We think that our approach would be applicable to many other numerical codes which use a multi-threaded task parallel approach. Details of how this concept has been actually implemented in the TAU code are given in Sec. 4 and first performance results are presented in Sec. 5. Finally, we conclude in Sec. 6 and give an outlook how to further improve the performance of the shared-memory parallelization presented.

2 The DLR TAU Code

The TAU code is a 3D-flow solver that simulates compressible external flows (steady or time-accurate) on unstructured grids using finite-volume discretization via the Reynolds-averaged Navier-Stokes equations (RANS). TAU features several turbulence models (Spalart/Allmaras, SST, RSM, etc.) as well as hybrid RANS/LES capabilities. It supports central spatial discretization (namely JST) as well as several upwind schemes. Moreover, the user may choose between cell-vertex and cell-centered metric. The most frequently used TAU solvers are Runge/Kutta and LU-SGS. Here we focus on the explicit Runge/Kutta solver with geometric multigrid, cell-vertex metric, central discretization, and a Spalart/Allmaras turbulence model, which requires one equation for the eddy viscosity in addition to the five equations for mass, impulse and energy. As a consequence, the number of degrees of freedom (DoF) is given by 6 times the number of grid points. Each edge in the grid corresponds one-to-one to a face in the so-called dual grid. This dual grid comprises control volumes around the grid points. The calculation of the fluxes between these control volumes, which are also called dual cells, is the main computational task in the scenario considered here. To integrate the fluxes for all dual cells, we could loop over all points and for each point we would loop over the faces of its surrounding dual cell. Recall however that each face in the dual corresponds one-to-one to an edge in the original grid. Thus, in this hypothetical implementation we would touch each edge twice: Once for each of the two end points of the edge.

For a more efficient access to main memory, the current implementation of TAU loops over the edges instead (c. f. [2]): For each edge, i.e., for each face in the dual grid, the respective value for the two points are updated. Using this "edge-based" scheme, the number of point-data loads is halved compared to a loop over the dual cells. Though TAU is edge-based, naturally, there are also point loops. The main computational load, however, is caused by edge loops.

MPI Parallelization via Domain Decomposition

TAU's parallelization is based on a domain decomposition: the grid is cut into several pieces (domains) by means of a partition of the point set, which may be obtained using a graph partitioning software like Chaco ([4]), Zoltan ([3]), (Par)Metis ([5]), etc. Each edge connecting two points in different domains has to be doubled, so that an edge as well as the two incident points exist in both domains (overlap). This adds a number of points to each domain, which are called "ghost points". Furthermore, the total number of edges in all domains equals the number of edges in the original grid plus the number of edges cut during domain decomposition. The flow values at a ghost point must be kept in sync with the corresponding data at the original point. This is done using message passing, namely MPI. MPI-based domain decomposition can be considered the standard parallelization concept for grid-based numerical codes that are run on modern HPC architectures, c. f. [8].

Obviously, the more edges are cut for domain decomposition, the more data has to be passed around via MPI to keep the domains, namely the evolving flow solution at the discretization points, synchronized. The time for this (usually bulk-synchronous) synchronization via MPI plus the inherent load imbalance due to an imperfect partitioning can be considered a serial part of the algorithm ($\rightarrow$Amdahl's law). Nevertheless, TAU's MPI parallelization scales well, c. f. [1].

As a consequence, for any fixed size CFD problem, there is a maximum number of domains that can be effectively used to compute this problem. A further increase of the number of domains eventually results in the parallel efficiency to drop until increasing the number of domains no longer speeds up the calculation at all (limit of scalability). If the number of domains is increased even further, the wall-clock time to solve this problem actually starts to increase. The scalability limit does not only depend on the CFD problem, but also on the cluster used – and on the application, of course.

MPI-synchronized domains + multi-threaded processing of domains

To increase the number of usable compute cores without increasing the number of domains, each domain must be computed on multiple cores. For the shared-memory parallel computation of a single domain with multiple threads, however, data parallelism becomes an issue: If, in an edge loop, two edges incident to a common point are processed concurrently (to update their two points, respectively), the update of the shared point may result in a data race: One of the two updates for this point can get lost. We hence not only need an efficient multi-threaded processing of edges, but also the prevention of data races. We need mutual exclusion among the updates of the same point. In principle this can be achieved in several different ways as we will detail now.

3 The Shared-Memory Parallelization – Generic Concept

We consider a given grid/mesh/graph consisting of a number of points and edges. Each edge connects two points. The number of edges incident to a point may vary, i. e., the connectivity may be regular (like for structured grids), yet it may also be irregular (unstructured girds). Note that whether this is an original graph or one obtained by domain decomposition makes no difference. Usually, the graph is very sparse as there may be several millions of points, yet each point has a very limited number of neighbors, say in the range of tens. Let us consider an algorithm that contains a large number of loops over both, edges as well as points: When passing over the edges, data associated with each edge's two points may be read and also updated. As a consequence, data races are possible when two edges incident to a common point are processed concurrently. Such data races must be prevented to ensure a correct behavior of the program. There are several obvious approaches to do so.

A critical section per point to ensure mutual exclusion of access to data associated with a point. This requires one mutex/lock variable per point, which has to be aquired whenever the point's data is read or written. Pthreads, OpenMP, or system libraries provide adequate functionality. Intrinsic functions for locked memory accesses provided by compilers may be used for a custom implementation.

Atomic updates of point data so that each read-update-write sequence touching a value associated with a point is atomic. For x86 architectures, "lock cmpxchg8b" may be used for an atomic update of a double-precision floating-point value (as it is done by most compilers for "omp atomic" directives).

Obviously, both approaches result in a huge number of locked memory accesses. Nevertheless, these two simple approaches were prototypically implemented in TAU. As expected, the very frequent use of locked memory access turns out a severe performance problem. If all edges incident to a point are exclusively processed by the same thread, no data races are possible for this point. Consequently, one may consider the following approach

Partition of the edge set into as many subsets as threads are concurrently running. Each edge is mapped to a particular thread. For points exclusively touched by edges processed by a single thread, no data races are possible. So mutual exclusion of point data accesses must be provided only for points that are incident to edges processed by two or more different threads.

We call a point "critical" if it is incident to edges processed by different threads so that mutex is necessary. The partition of the edges should be such that the total number of critical points is minimized. In addition, the number of edges processed by each threads should be as balanced as possible to obtain a good load balance. Moreover, the number of critical points touched by each thread should be as balanced as possible. As one might notice, these are quite a number of constraints on the partition of the edge set. In addition to the imbalances due to the edge partition not being perfect, also varying waiting times for locked memory access result in a load imbalance among the n threads when they process the n edge sets in parallel. Despite these load-balancing issues, whenever a point is to be updated within a parallelized edge loop we must know whether this point is critical or not. A prototype implementation in TAU showed that, even though this additional information may be stored without additional memory space (for instance by using signed integers for the point indices and taking the sign bit as an indicator) so that the memory access pattern is not changed, the branching between critical and non-critical points leads to a considerable overhead.

Each edge set spans/induces a subgraph. Note that critical points belong to two or more subgraphs. Assume for a moment that two of the n edge sets are such that the corresponding subgraphs are disjoint. Then these two edge sets can be concurrently processed by two threads without the need to prevent data races. In general, no data races can occur as long as only disjoint subgraphs are concurrently processed. Then each point – in particular the formerly critical ones – is accessed by at most one thread at a time. We thus finally refine the above model towards the following approach:

Asynchronous dependencies-driven parallel processing of a large number of small subgraphs. This model enables a dynamic load-balancing among threads. Moreover, by ensuring mutual exclusion among subgraphs sharing (a) common point(s), no data races can occur. So there is no need to tell between critical points and non-critical ones, which drastically reduces the number of locked memory accesses as well as the overhead of branching.

Assume, just as an example, that there are $10n$ subgraphs so that **on average** each of the n threads processes 10 **dynamically allocated** subgraphs per point/edge loop. Consider the processing of a subgraph s in a loop a task. Then we have task dependencies like "subgraph s must have been processed in the ith loop before s is processed in loop $i+1$". With this, we actually implemented a thread-pool pattern: In each loop the threads process tasks, i. e. subgraphs, that need to be processed at that stage of the program, i. e. for which the temporal data dependencies are met.[1] Furthermore, our task-dispatching logic incorporates the mutual exclusion of tasks for neighboring subgraphs to prevent data races.[2] In other words, the neighborhood structure of the subgraphs induce "mutex dependencies" among the tasks. No explicit thread synchronization is necessary since the threads synchronize automatically via the dispatching of

[1] This is somewhat similar to what the "SMP Superscalar" ("SMPSs") programming model/environment provides [7,9], yet in a bottom-up, loop-based approach, rather than the top-down function annotation in SMPSs.

[2] This is dissimilar to SMPSs, as (to our knowledge) SMPSs does not provide explicit locking of neighbouring data segments.

tasks subject to the temporal/mutex dependencies. Finally note that balanced sizes of the subgraphs are no longer that crucial. Instead, the processing of a subgraph should exclude as few as possible other subgraphs from being processed in parallel. So, the neighborhood structure of the subgraphs is most crucial here.

4 The Shared-Memory Parallelization – Implementation Details for TAU

In the TAU solver to be modified, a partitioning of the edge set, namely a **coloring of the edges,** already exists – yet for a different reason than described in the preceding section. Originally, the edge coloring was introduced to prevent data races when a long sequence of edges is processed concurrently on a vector processor. When vector-processor-based supercomputers were superseded by commodity clusters based on the omnipresent x86 architecture, this coloring was repurposed to maximize cache utilization. Even with the memory controller integrated into the CPU this cache optimization is absolutely critical to TAU's (serial) performance.

4.1 Cache Blocking in TAU

The TAU solver considered in this paper is memory-bound on current x86 architectures. To lower the number of loads from main memory per flop, the data layout is optimized with respect to cache utilization: The edges are sorted such that edges incident to the same point follow as closely as possible (temporal blocking of point-data access). Essentially we use space-filling Hilbert curves in this approach. In addition, the points are sorted such that, when accessing the points indirectly while looping over the edges, data associated with subsequently accessed points are close in the memory (spatial blocking). The temporal blocking is supposed to minimize the number of loads from main memory, whereas the spatial blocking is supposed to make use of the prefetching mechanisms of the hardware's memory/cache subsystem. With this strategy, TAU shows very good cache utilization. TAU is still memory-bound, though.

To enable a TAU programmer to split up a large edge loop into several routines (to improve code structure, readability, maintenance, expandability), so-called "colors" exist in TAU. Each color consists of a number of subsequent edges (in optimized order, c. f. the temporal blocking above) such that all the point data touched by these edges (which are spatially blocked, c. f. above) fit into the L2 cache. Note that this coloring is actually a partition of the edge set. A loop over the edges is equivalent to a nested loop over the colors followed by a loop over each color's edges. Code in the body of an edge loop may be split up into several routines on a per-color basis. Then only in the first routine the point data touched by the current color are loaded from main memory. For subsequent routines called for the color, this data is already cached. For commonplace x86 systems, the cache coloring in TAU enables L2-cache-local processing of an edge loop despite the code being split across several routines, possibly in different compilation units.

4.2 Modification of the Colors in TAU to Suite the Hybrid Parallelization Concept

Recall that the coloring in TAU is based on space-filling curves. Unfortunately, this results in a bad neighborhood structure among the subgraphs induced by the colors: there are too many colors with too many neighbors. As our concept for the shared-memory parallel processing of subgraphs described in Sec. 3 requires mutual exclusion among neighboring subgraphs, the coloring needs to be adjusted appropriately – without changing cache performance to the worse.

Recall that the subgraphs should be such that each subgraph touches as few as possible other subgraphs. Furthermore, the subgraphs should be balanced w. r. t. the number of points, whereas the number of edges per subgraph is of minor importance. These requirements perfectly fit graph-partitioning algorithms; for instance Sandia's "Chaco" (cf. [4]) may be used to obtain the colors.

Actually, there are different types of colors since there are three types of points in TAU: points that lie in the physical boundaries of the computational domain, ghost points (forming the halo of a domain obtained by domain decomposition for MPI parallelization), and the rest of the points, which we call "inner points". Correspondingly, there are four color types: inner colors, boundary-touching colors, halo-touching colors, and boundary+halo-touching colors. This enables us to overlap the processing of physical boundaries and the synchronization of ghost points via MPI with the processing of inner colors – subject to mutex and temporal data dependencies, of course. This integrates nicely with the thread-pool model since the processing of domain/physical boundaries at a particular stage in the program can be considered tasks just as well.

4.3 Minimally Invasive Implementation of the Task Dispatching

As TAU is a production code (validated by/for its customers), the numerics cannot be easily changed just to better suite TAU's parallelization. The shared-memory parallelization that has been added (cf. above) is designed such that it would yield exactly the same results as the originally serial code if floating-point arithmetic was exact. With the thread-pool-based parallel processing of the colors, the order in which the edges/points are processed may change from loop to loop. As a consequence, the limited precision of floating-point operations can – at least in principle – result in numerical differences. For the TAU solver considered, however, merely negligible differences are observed, if any.

Besides the consistency of the numerical behavior, the following aspect of software development/engineering has been very important: The shared-memory parallelization was supposed to change the code as little as possible, preferably transparent to the programmers. The proposed model indeed allows for an almost transparent implementation: As a loop over the edges was already split, namely done by a nested loop over the colors and the color's edges, respectively, the task dispatching was easily integrated (using C-code-like syntax): In

```
for(color=colorhead; color != NULL; color=color->next)
  for(eidx=color->start; eidx < color->stop; eidx++)
    {  /* process edge with index eidx */  };
```

merely the first line must be changed into

```
for(color=get_color(grid); color!=NULL; color=get_color(grid))
```

The task dispatching logic is completely encapsulated in the newly introduced `get_color()` function, which returns an appropriate color as long as there are colors left to be processed at that stage, else NULL. Note that `get_color()` may block if, at a given stage, for all colors left at that stage a neighboring color is being processed. To loop over the points, an additional while loop is introduced:

```
for(pidx=0; pidx < grid->npoints; pidx++)
  {  /* process point with index pidx */  };
```

must be replaced by

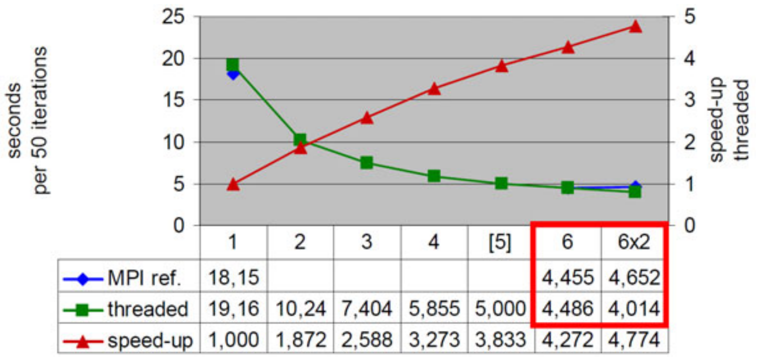
```
while(get_point_range(grid, &pidx, &pstop))
  for( ; pidx < pstop; pidx++)
    {  /* process point with index pidx */  };
```

The newly introduced `get_point_range()` function simply uses `get_color()` to obtain a color to be processed and with it a point range (each point is associated with exactly one color). The really nice thing with this implementation is that there is no need to change the bodies of point/edge loops – at least in principle. Naturally, when porting a loop, care has to be taken that no data is unintentionally shared. Thread-local storage (TLS) may be necessary. When, for instance, a maximum of a given value at the points is to be computed, TLS is only the half way: The maximum is to be determined as the maximum of the threads' local maxima, necessitating an explicit change in the code. Nonetheless, with this implementation of the concept, only a limited number of changes in the code are necessary to enable multi-threading.

In order to overlap the MPI communication (to synchronize the ghost points) with computation, `this_thread_syncs_halo()` is introduced. This function returns TRUE for exactly one of the threads. In case TRUE is returned, this function may block until no halo-touching color is processed and then keeps tasks for halo-touching colors from being dispatched until the domains' halos are synchronized. Meanwhile the other threads continue to process tasks for non-halo-touching colors. A similar mechanism is used to overlap the single-threaded processing of physical boundaries with the processing of inner colors.

5 First Performance Results

The TAU RANS-solver considered is an explicit 3-stage Runge/Kutta scheme with multigrid acceleration; cell-vertex metric, central discretization (JST), scalar dissipation, Spalart/Allmaras turbulence model are used. We get right to the point: How does the pure shared-memory parallelization face against the pure MPI parallelization. This test was run on a single-CPU machine running SLES 11 with an Intel 6-core Westmere EP X5670 with 2-way SMT enabled. Unfortunately, no Intel compiler was available, so gcc 4.3.3 with full optimization was



	1	2	3	4	[5]	6	6x2
◆ MPI ref.	18,15					4,455	4,652
■ threaded	19,16	10,24	7,404	5,855	5,000	4,486	4,014
▲ speed-up	1,000	1,872	2,588	3,273	3,833	4,272	4,774

Fig. 1. Pure shared-memory parallelization (green) vs. pure MPI parallelization (blue) for an Intel Westmere 6-core CPU with 2-way SMT (X5670). Wall-clock time for 50 iterations vs. #threads or #domains, respectively; "speed-up" referes to "threaded".

used. Pinning of threads/MPI-processes to physical (logical) cores was applied when running 1 to 6 (12) threads/MPI-processes. The grid has 100,592 points (95,344 tetras and 163,625 prisms), i. e. about 17,000 points per physical core.

As Fig. 1 shows, the modified coloring slightly affects the serial performance (no multigrid). The wall-clock time increases by 5–6 % when comparing the shared-memory version (actually, the hybrid one) running single-threaded vs. the base-line/MPI-only TAU with a single domain. Performance measurements using LIKWID [11] indicate that cache utilization might be the reason. When comparing 6 threads for one domain vs. 6 domains (MPI processes), the shared-memory version is neck and neck with the original MPI version, it is off by less than 1%. As expected, the MPI version fails to utilize the 2-way SMT when running with 12 domains. For the shared-memory version, however, a speed-up of 1.117 is observed when using 12 threads (pinned to the 12 logical cores, respectively). The reason might be a better pipeline utilization and latency-hiding effects. Thus, the shared-memory version outperforms the base-line MPI version by about 10% for this particular setting. This clearly demonstrates the potential of the concept proposed (as well as of its implementation in TAU). According to measurements using LIKWID, the shared-memory parallelized TAU using 12 hardware threads obtains a sustained performance of 8.5 GFlop/s (double precision) for the X5670 (recall that CFD on unstructured grids is considered).

The main reason to add shared-memory parallel processing of domains to TAU, however, was to extend TAU's scalability, i. e., to use more cores more effectively. And indeed, the shared-memory parallelization enables us to effectively utilize more cores. The test depicted in Fig. 2 was run on the $C^2A^2S^2E$ cluster located at the Braunschweig site of the German Aerospace Center, which comprises 648 compute nodes, each with two Intel X5670, connected via QDR Infiniband. The grid used has 13 mio points. Unfortunately, the 2-way SMT of the CPUs is disabled, affecting the performance of the multi-threading, cf. above. The hybrid version of TAU uses one domain (MPI process) per socket, whereas

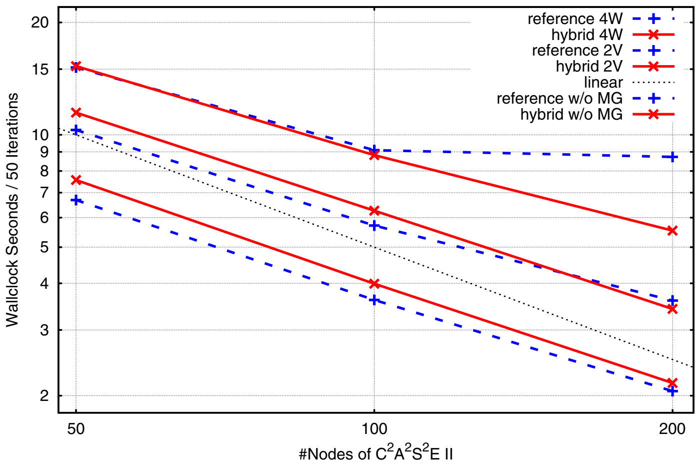


Fig. 2. Strong scaling of TAU. Reference, i. e. MPI only, in blue dashes vs. hybrid in solid red. Grid with 13 mio points on a 2-socket 6-core Westmere (X5670, 2-way SMT disabled) system with QDR Infiniband interconnect (50% blocking). Wall-clock time for 50 iterations vs. #nodes used. 3-stage explict Runge/Kutta with 4W multigrid (top), 2V (middle), and no multigrid (bottom).

for the base-line TAU we have one domain per core, i. e. 12 domains per node. As Fig. 2 shows, the hybrid parallelization significantly increases TAU's scalability – at least when using 4W multigrid, which is preferable practice to do. (The improved scalability can be noticed already for 2V.) For the base-line TAU, for 4W multigrid scalability flattens off at 100 nodes (1200 cores): Using 200 nodes (2400 cores) instead does not result in a noticeable speed-up. In contrast, the hybrid parallel version scales well: The speed-up obtained when using 200 nodes instead of 100 nodes is almost as large as for 50 nodes to 100 nodes. In short words, the hybrid can run 50 iterations in 5.5 seconds, whereas the MPI-only version needs 8.9 seconds – a speed-up of over 1.6 in wall-clock time. Unfortunately, 2-way SMT is disabled, and besides that, more than 200 nodes of the HPC cluster could not be acquired to max out the speed-up attainable for this real-world CFD problem. Nevertheless, this test clearly proves that the hybrid-parallel TAU scales significantly better than its base-line using MPI only.

6 Conclusion and Outlook

As shown by the first, preliminary results described in the preceding section, the concept proposed for shared-memory parallelization of grid-based CFD solvers works well – at least its implementation in the TAU solver considered. Naturally, further tests are needed, in particular for other CPUs. We plan tests on AMD's Magny Cours and on IBM's Power7. It will be interesting to see how these relate to Intel's Westmere considered in the tests presented here.

Even though the implementation here was done for the TAU CFD solver, the concept is not TAU-specific, but rather can be applied to a large number of applications. For example for a stencil based code $A[i] = B[i-1]+B[i+1]$, the concept of dependencies among subgraphs translates to dependencies of thread chunks in OpenMP-parallel FOR loops. Whether or not $A[i]$ of the thread chunk can be calculated, for example, simply depends on whether or not the neighbouring index elements $B[i-1]$ and $B[i+1]$ have already been computed (temporal data dependency). We are currently porting this thread-pool-based parallelization for a stencil-based block-structured CFD turbo machinery code, c. f. [10].

References

1. Alrutz, T.: Investigation of the parallel performance of the unstructured DLR-TAU-code on distributed computing systems. In: Deane, E. (ed.) Parallel Computational Fluid Dynamics, pp. 509–516. Elsevier, Amsterdam (2005)
2. Alrutz, T., Simmendinger, C., Gerhold, T.: Efficiency enhancement of an unstructured CFD-code on distributed computing systems. In: Proc. ParCFD (2009)
3. Devine, K., Boman, E., Riesen, L., Catalyurek, U., Chevalier, C.: Getting started with Zoltan: A short tutorial. In: Proc. Dagstuhl Seminar Combinatorial Scientific Computing, Also Sandia National Labs Tech Report SAND2009-0578C (2009)
4. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Proc. 1995 ACM/IEEE Conference on Supercomputing (CDROM), ACM, New York (1995)
5. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Comp. 20, 359–392 (1998)
6. Kroll, N., Fassbender, J.K. (eds.): MEGAFLOW — Numerical Flow Simulation for Aircraft Design Results of the second phase of the German CFD initiative MEGAFLOW presented during its closing symposium at DLR, Braunschweig, Germany, December 10-11. Notes on Numerical Fluid Mechanics and Multidisciplinary Design, vol. 89. Springer, Heidelberg (2005)
7. Marjanović, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: Proc. 24th ACM Int'l Conference on Supercomputing, pp. 5–16 (2010)
8. Mavripilis, D.: Parallel performance investigation of an ustructured mesh Navier-Stokes solver. The Int'l Journal of High Performance Comp. 2(16), 395–407 (2002)
9. Planas, J., Badia, R., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. Int. J. High Perform. Comput. Appl. 23, 284–299 (2009)
10. Simmendinger, C., Kügeler, E.: Hybrid parallelization of a turbomachinery CFD code: performance enhancements on multicore architectures. In: Proc. ECCOMAS-CFD (2010)
11. Treibig, J., Hager, G., Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. CoRR, abs/1004.4431 (2010)