The next-generation CFD solver *Flucs* – HPC aspects

Jens Jägersküpper  
German Aerospace Center  
Institute of Aerodynamics and Flow Technology  
Center for Computer Applications in AeroSpace Science and Engineering (C²A²S²E)

With contributions by several colleagues

T-Systems-SfR HPCN workshop, May 5th 2015
Towards the Virtual Aircraft
Motivation & Strategic Objectives

Need for DLR Virtual Aircraft Software Platform

- Support industrial and research activities in Germany
- Full knowledge of all Flight Physics aircraft properties relevant for design & certification
- Trade-off between disciplines for technology evaluation
- Identify future options for HPC based aircraft design
- Enhance and maintain aircraft design capabilities in DLR

DLR Guiding concept: Virtual Product
Towards the Virtual Aircraft
Challenges & Capability Needs

Simulation of Flight Envelope
- Separated flows
- Transition laminar/turbulent
- Unsteady effects
- Multidisciplinary simulations

Aircraft Optimization
- Link of preliminary and “detailed” simulation / optimization capabilities
- High-fidelity for relevant disciplines
- Large number of design parameters
- Identification of realistic & relevant load scenarios for structural lay-outs (metal, composites)
Long term goals

- Development of an integrated software platform for multidisciplinary analysis and optimization based on high-fidelity methods,
  - Integration of relevant disciplines (aerodynamics, structures, flight mechanics, flight control)
- Development of a next generation CFD solver with innovative solution algorithms and high parallel performance on future HPC architectures

Short term goals (6/2016)

- Realistic maneuver simulations
- Integrated aero/structural design
- Demonstration of new capabilities using industrial relevant configurations, best practice guidelines
- First release of next generation CFD solver “Flucs” for industrial use
HPC and CFD

How to use multiple CPUs to solve one CFD problem?

- 36 points
- 60 edges

Domain Decomposition:
- 4 domains
- 12 edges cut
- 20 points affected by cuts

4 domains (here congruent), each
- 9+6 points
- 15+3 edges
HPC and CFD
Keeping the domains synchronized using MPI

4 domains (here congruent), each with
- 9+6 points
- 15+3 edges

Overlap of domains, the “halos”
- 4 x 3 = 12 cut edges
- 12 faces doubled, i.e. added
- 12 x 2 = 24 “ghost points”

MPI communication volume is proportional to # ghost points, i.e. proportional to # edges cut
HPC and CFD

Limitations of (MPI-based) Domain Decomposition

- Due to the ghost points, which are necessary to compute the domains independently, domain decomposition increases the total problem size.
- The more domains, the more edges are cut by the domain decomposition.

Example: Consider a 3D cubic structured mesh with 100 x 100 x 100 points.

- Decomposition in $2^3 = 8$ subcubes, each with $50^3 + 3 \times 50^2 = 132,500$ points, i.e., 125,000 plus 6%.
- Decomposition in $8 \times 8 = 4^3 = 64$ subcubes, some with $25^3 + 6 \times 25^2 = 19,374$ points, i.e., 16,625 plus 16%.
- Decomposition in $8 \times 64 = 8^3 = 512$ subcubes, some with $13^3 + 6 \times 13^2 = 3,211$ points, i.e., 1,953.125 plus 64%.

Observation: The #ghost points increases significantly as the #domains is increased since more and more edges must be cut (and doubled).
Parallelization via Domain Decomposition using MPI

The more domains are used (for a given mesh) …

- the more halos to be kept in sync via MPI
- the larger the total halos’ size
- the larger the total MPI traffic
- the more MPI messages are necessary
- the smaller the (average) message size

**Strong scaling is affected**

- inherently by domain decomposition due to increasing total problem size
- MPI communication gets latency-bound
- various load imbalances
  - domains vary w. r. t. #points and #edges
  - varying halo sizes
  - #neighboring domains varies
HPC and CFD

Parallel Performance – Amdahl’s Law

\( p \) denotes the fraction of a program’s serial runtime spent in parts of the code that can be executed in parallel, i.e. concurrently. (Serial fraction is \( 1 - p \).)

\( N \) is the number of processors (here: CPU cores) used to speed up the program’s execution – in the parallelized parts of the code.

**Amdahl’s Law:** If \( p < 1 \), the maximum attainable speed-up, i.e. the factor by which the execution can be speed up, is bounded above by

\[
\frac{1}{(1-p) + \frac{p}{N}} \quad \xrightarrow{N \to \infty} \quad \frac{1}{1-p}
\]

**Notes**

- This law is merely an upper bound since perfect speed-up of the parallel fraction of the code is assumed, cf. the term “\( p/N \)”
- A given problem, i.e. a fixed problem size, is assumed
- That is, the scenario considered is “strong scaling”
In case 1% of the code is serial and 99 processors are used, the speed-up is 50 at the best so that the parallel efficiency is limited by $\frac{50}{99} \approx 50\%$ a priori.
Domain Decomposition / Partitioning – TAU Performance

The graph shows the speedup of different partitioning methods as a function of the number of CPU cores. The methods compared include:

- 3v, TAU-private
- 3v, TAU-private + MPI rank optimization
- 3v, Zoltan-partition
- 3v, Zoltan-partition + MPI rank optimization
- 3v, Chaco

The graph-based partitioning method is indicated by the green line, which shows a higher speedup compared to other methods, especially at higher numbers of cores. The linear speedup compared to 12 cores is also shown as a reference line.

RCB is indicated by a specific marker on the graph.
Domain Decomposition / Partitioning – Neighbors
1024 domains

Message rate is a bottleneck
HPC Software – Parallelization / Programming

Common: MPI on a single compute node / workstation

For instance on a 4-socket quad-core node: 16 cores – 16 MPI processes

One MPI process/core is a bottleneck

The boon and bane of MPI:
For the last two decades no need to adapt the parallelization of codes
The Multi-Core Shift in HPC Hardware

Moore’s law:
“The number of transistors on a chip doubles roughly every 1.5 years.”

Observed until recently:
The performance of a single CPU core doubled roughly every 1.5 years.
(A common misinterpretation of Moore’s law!)

Observed since a few years:
Even though Moore’s law is still valid, today the performance of a single CPU core hardly increases. Instead, the number of cores per chip is increased so that the performance of a chip keeps increasing.

Conclusion: As computers are no longer getting faster, but “bigger”, software must in particular be able to efficiently utilize more and more cores to continue benefiting from the advance in HPC hardware.
The Manycore Shift in HPC Hardware – and Software

Multiple levels of explicit parallelism in cluster hardware
1. nodes – message passing (MPI) perfectly reflects interconnect!
2. sockets (multiple CPUs)
3. multi-core CPUs
4. symmetric multi-threading
5. SIMD (vector) units

Simple, but crucial conclusion:
To obtain maximum computational performance of such an HPC system, all levels of parallelism in its hardware need to be fully exploited.

→ Multi-level (“hybrid”) parallelization matching hardware’s characteristics
→ Asynchronous computation/communication to overlap/hide latencies and/or load imbalances
The “HI-CFD” Project

“HI-CFD” stands for
Highly efficient implementation of CFD codes on many-core architectures

Funding by the Federal Ministry of Education and Research (BMBF)

⇒ Program “ICT 2020 – Research for Innovations”
⇒ Call “HPC Software for Scalable Parallel Computation”
⇒ Duration: 3 years, 1/2009 – 12/2011

Goal:
Development of generic methods and tools for adaptation and optimization of CFD codes for many-core architectures.

Demonstration of these methods and tools by applying them to the CFD codes TAU (unstructured meshes) and TRACE (block-structured meshes).
The German HI-CFD Project – Main Topics

Utilizing SIMD capabilities of today’s and future processors

⇒ SIMD – Single Instruction Multiple Data: concurrent exec of same operation on different, independent sets of data
⇒ SSE on Intel / AMD x86_64 CPUs, IBM’s double hummer PPC, CUDA for Nvidia GPUs, ATI Stream (ATI GPUs and AMD CPUs), OpenCL, etc.

Hybrid Parallel SPMD-Programming (Single Program Multiple Data)

⇒ CFD codes – just as many other codes – often purely MPI-parallelized
⇒ For clusters, today’s multi-core chips and upcoming many-core architectures allow for shared memory parallelization per node (per socket)
⇒ How can CFD codes benefit from a shared-memory/MPI hybrid parallelization?
  Lesson learned from literature: Hybrid not necessarily better than pure MPI
Multi-Threaded Processing of a Domain
Asynchronous shared-memory parallelization

Example: Domain partitioned into 4 colors (without halos!)

Each color gets processed *en bloc* by one of the threads

Race conditions possible ONLY at marked “critical” points, which are touched by different colors

Updates of those points MUST be serialized in some way

**Mutual exclusion of neighboring colors to prevent data races**

“Color locking” – as long as a color is being processed, no neighbor color can be processed

Here, only BLACK and BLUE can be processed concurrently (no common point → no data race)
Hybrid-Parallel TAU: Explicit Runge/Kutta, 4W Multigrid
One domain per chip (MPI) + Multi-Threaded processing

TAU Performance (241 K points, RK+MG 4W, JST, scal. diss., GG grads, SAO)

Wallclock (Sec.)

Hybrid Parallel TAU + SCOUT
Plain Reference TAU
linear scaling

#Cores of HAL9000 (2 x 6 cores per node, Westmere + QDR IB)
TAU Prototype with PGAS-Based Halo Synchronization

- TAU-prototype using the PGAS library “GPI”, which becomes GASPI ref. impl
- 1-sided RDMA via Infiniband is used to update ghost points
- integrates nicely with the HICFD task-based shared-memory parallelization
BMBF Project “GASPI“

GASPI – “Global Address-Space Programming Interface”

- Duration: 3 years, 6/2011 – 5/2014
- 10 partners: Fraunhofer ITWM, Fraunhofer SCAI, T-Systems SfR (lead), scapos AG, Technische Universität Dresden (ZIH), Forschungszentrum Jülich, Karlsruhe Institut of Technologie, Deutscher Wetterdienst, DLR AT, DLR AS

- Project goals:
  - Define and establish a PGAS API (Partitioned Global Address Space) for a asynchronous 1-sided communication model
  - Reference implementation (GPI) and open-source implementation
  - Port some applications (at least “kernels”) from MPI to GASPI
Full exploitation of future HPC systems
Consolidation of current DLR CFD solvers
Flexible building blocks
Basis for innovative concepts & algorithms e.g. high-order-finite element discretization, adaptivity, …
Seamless integration into multidisciplinary simulation environment
State-of-the-art software engineering methods (C++11, templates)
Key Aspects and Design Drivers

- Multi-disciplinary (coupled) simulations → **FlowSimulator (FS)**
- Efficient Chimera technique, e.g. for moving/sliding meshes
- Time-accurate simulation of unsteady flows
- **Mixed-element unstructured meshes** with hanging nodes (1:2 for a start)

- Finite volumes (FV) discretization as well as higher-order discretization via finite elements (DG method)
- Cell-centered discretization with improved reconstruction (FV)
- Implicit solution algorithms
  - option: consistent derivatives via automatic differentiation (AD)

- Multi-level parallel implementation
- Reduce I/O to a minimum
- **Modular software design** (testing, maintenance, extensibility)
Next-Generation CFD Solver
Software Development Process

- Modernized Development Environment
  - Programming language: C++ with heavy use of templates
  - User-level functionality available through Python interface (FlowSim.)
  - Standard state-of-the-art development tools
    - Customized **eclipse** as IDE (Integrated Development Environment)
    - **git** version control (promotes branch-based development)
    - Continuous integration using **jenkins**
      - automatic build, running tests, check of programming style
    - Defined Web-based code-reviewing process using **mantis**, **gerrit**
      - Two reviewers need to approve changes to core functionality

- Modularized Software Design
  - Goal: improved maintainability and extensibility
    - Making adding new models/algorithms easier (less error-prone)
      - traded against complexity in framework functionality
  - Modules vary in complexity (w.r.t. programming)
Next-Generation CFD Solver
Software modules with varying complexity and access level

no access
restricted access
accessible

Framework / Infrastructure
- FSDM interfacing
- I/O
- Coupling
- Data Containers
- Loops
- Parallelization

Discrete Equation
- Definition of PDEs
- Closures (e.g. Gas Models)
- Models (e.g. SA-neg)

Spatial Discretization
- Finite Volume
- Discontinuous Galerkin

Time Integration
- Runge-Kutta
- Explicit
- Implicit
- Acceleration Techniques
- Multigrid

Python Interface
- Control of
- Program Flow
- Data Flow
- FS compatible
- Simulation scripting

core developer
model / algorithm developer
user
Multi-Level Parallelization: 2-Level Domain Decomposition
Message Aggregation

- Domain 0
- Subdomain 0,0
- Subdomain 0,1
- Domain 1
- Subdomain 1,0
- Subdomain 1,1

- add (ghost) points
Multi-Level Parallelization of \textit{Flucs}

Classical Domain Decomposition (1\textsuperscript{st} level)
\begin{itemize}
  \item One ghost-points communication buffer per domain and neighbor domain
  \item Ghost-point data grouped by donor domain \Rightarrow no scattering for receives
  \item But: \textcolor{red}{Overlap communication} with computation on domain-local data
\end{itemize}

Sub-Decomposition (2\textsuperscript{nd} level)
\begin{itemize}
  \item Static one-to-one mapping of a sub-domain to a (logical) core (thread)
    \begin{itemize}
      \item Each element is and stays located r/w in one core’s cache hierarchy
      \item \textcolor{red}{Data locality}
    \end{itemize}
  \item Cross-subdomain faces
    \begin{itemize}
      \item Data left and right of the face are read by both subdomains’ threads
      \item Writes are only on the owned elements
      \item Computations for such a face is done twice
      \item Minimum number of arithmetic operations traded for \textcolor{red}{data parallelism}
    \end{itemize}
\end{itemize}
First Performance and Scalability Results

Solver Settings
- Compressible Euler equations
- 2nd order FV discretization
- Explicit Euler time integration

Configuration
- 12-core IVB dual-socket
- FDR IB (Mellanox)
- IntelMPI 5.0.1 (shm:ofa)
- GASPI implementation
  GPI-2 1.1.0
Flucs – Single Node Scalability (2 x 12-Core IVB CPU)

Flucs @ CASE Cluster, 03/2015, F6 mesh with 4.9e6 cells (2e6 points)

Flucs 168b38 @ E5-2695v2 (2.4GHz)  
one domain per socket (IMPI 5.0)  

linear ——- dual socket

- 0.855
- 1.0 (100%)
- 3.834 (96%)
- 6.81 (85%)
- 8.96 (75%)
- 12.58
- 17.13 (71%) @ 100K elems/thread

pic source: Intel

# logical cores (hyper threads)
## Efficiency of Generated Machine Code

### Single threaded

<table>
<thead>
<tr>
<th>Metric</th>
<th>FlucS</th>
<th>TAU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI</td>
<td>0.663</td>
<td>0.705</td>
</tr>
<tr>
<td>Load to Store ratio</td>
<td>2.02</td>
<td>1.99</td>
</tr>
<tr>
<td>L1 DTLB miss rate</td>
<td>5.035e-05</td>
<td>5.031e-05</td>
</tr>
<tr>
<td>Branch rate</td>
<td>0.0546</td>
<td>0.0374</td>
</tr>
<tr>
<td>Branch misprediction rate</td>
<td>0.00014</td>
<td>0.00036</td>
</tr>
<tr>
<td>Branch misprediction ratio</td>
<td>0.0026</td>
<td>0.0097</td>
</tr>
<tr>
<td>Instructions per branch</td>
<td>18.32</td>
<td>26.73</td>
</tr>
<tr>
<td>L2 request rate</td>
<td>0.0342</td>
<td>0.0819</td>
</tr>
<tr>
<td>L2 miss rate</td>
<td>0.0091</td>
<td>0.0166</td>
</tr>
<tr>
<td>L2 miss ratio</td>
<td>0.2649</td>
<td>0.2024</td>
</tr>
<tr>
<td>L2 data volume [GBytes]</td>
<td>37.9</td>
<td>128.7</td>
</tr>
</tbody>
</table>

*Compiler: gnu 4.9.2*
### Efficiency of Generated Machine Code

12-threaded (6 x 2 HT)

<table>
<thead>
<tr>
<th></th>
<th>Flucs</th>
<th>TAU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>5.05</td>
<td>4.8</td>
</tr>
<tr>
<td>CPI</td>
<td>1.36</td>
<td>1.53</td>
</tr>
<tr>
<td>Load to Store ratio</td>
<td>2.14</td>
<td>1.96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Flucs</th>
<th>TAU</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 DTLB miss rate</td>
<td>8.526e-05</td>
<td>0.00023</td>
</tr>
<tr>
<td>Branch rate</td>
<td>0.0673</td>
<td>0.04047</td>
</tr>
<tr>
<td>Branch misprediction rate</td>
<td>0.0002</td>
<td>0.00036</td>
</tr>
<tr>
<td>Branch misprediction ratio</td>
<td>0.003</td>
<td>0.0088</td>
</tr>
<tr>
<td>Instructions per branch</td>
<td>17.7</td>
<td>25.0</td>
</tr>
<tr>
<td>L2 request rate</td>
<td>0.0381</td>
<td>0.1131</td>
</tr>
<tr>
<td>L2 miss rate</td>
<td>0.0093</td>
<td>0.0256</td>
</tr>
<tr>
<td>L2 miss ratio</td>
<td>0.28</td>
<td>0.25</td>
</tr>
<tr>
<td>L2 data volume [GBytes]</td>
<td>208</td>
<td>749</td>
</tr>
</tbody>
</table>
FlucS Runtime – Strong Scaling Scenario
1.9 Mio Elements @ C²A²S²E Cluster
Flucs Speed-up – Strong Scaling Scenario
1.9 Mio Elements @ C²A²S²E Cluster
Flucs Parallel Efficiency – Strong Scaling Scenario
1.9 Mio Elements @ C²A²S²E Cluster
FlucS Parallel Efficiency – Strong Scaling Scenario
1.9 Mio Elements @ C²A²S²E Cluster
**Flucs Parallel Efficiency – Strong Scaling Scenario**

1.9 Mio Elements @ C²A²S²E Cluster

1,900,000 elements on 200 nodes
- 9,500 elements per node
- 4750 elements per socket (not counting halo elements)
- ~ 400 elements per physical core
- ~ 200 elements per logical core, i.e. per thread

Who in the world would ever want to compute on just 2 mio. elements running roughly 10,000 threads?
Expected Exascale-System Architecture Characteristics

From “The Int’l Exascale Software Project Roadmap” (www.exascale.org)

- $10^8$ – $10^9$ cores
- Clock rates of 1 to 2 GHz
- Multithreaded, fine-grained concurrency of 10- to 100-way concurrency per core, i.e.,
- Approx. $10^{10}$-way concurrency for simultaneous operation and latency hiding
- Global address space without cache coherence; extensions to PGAS
- Explicitly managed high-speed buffer caches; part of deep memory hierarchy

Conclusion

What might seem “ridiculous” or “overengineered” to some today, might become necessary to efficiently use (high performance) computing hardware in less than 10 years. And the life cycle of *Flucs* has just begun…