Parallel Calculation of Accurate Path Lines using Multi-Block CFD Datasets with Changing Geometry

Andreas Gerndt, Marc Schirski, Torsten Kuhlen, Christian Bischof
Center for Computing and Communication
Aachen University (RWTH), Germany

Abstract *In many cases, an elliptical system of partial differential equations (PDEs) has to be solved in order to compute a given flow problem. For domain decomposition, mainly the multi-block grid approach is used. A variety of flows are unsteady, thus the calculation of path lines is a common way of exploring the flow field. However, computing path lines is more complicated if the underlying grid geometry changes over time. We make use of a fragmented multi-block dataset for a parallelization approach to compute path lines. We describe our enhancements of VTK, the used basic toolkit for scientific visualization, which neither supports multi-block nor time-dependent datasets. Our extensions include the handling of unsteady datasets as well as adaptive step-size control and time-position-interpolation. Finally, we depict the results of our efforts in order to speed-up Computational Fluid Dynamics (CFD) explorations in Virtual Environments.*

Keywords: Computational Fluid Dynamics, Multi Grid, Interpolation, Parallel Algorithms, High-Performance Computing, Virtual Reality

1 Introduction

Visualization techniques are applied in scientific computation to achieve understandable visual representations of numerical data in order to improve comprehension. Virtual Reality offers a faster access to complex spacial relationships but requires real-time interaction capability. Therefore, huge simulation data should not be handled directly by the visualization host. Likewise, expensive post-processings should be shifted onto a dedicated high-performance computer.

For Computational Fluid Dynamics (CFD), the whole process can be divided into three categories, and each of them is subdividable into further phases. The three large areas are grid generation, the simulation itself, and post-processing.

The post-processing prepares simulation results for the visualization. This includes computations in order to extract objects like streamlines, iso-surfaces, and cut-planes, which can subsequently be converted to drawable objects.

This paper concerns the parallel computation of time-dependent path lines. First, the advantage and the exploitation of multi-block topologies is depicted. Thereafter, the approach of VTK in order to calculate streamlines is described. Based on general functionalities of this toolkit, we have implemented accurate path line schemes, explained next. Finally, the parallelization framework shows the possibility to speed-up path line computations.

2 Data Structures

The type of generated grids is not only significant for the quality of the simulation but also for the extraction algorithms. Grid generation is exclusively geared towards an optimal converging simulation result. The extraction and filtering methods are obliged to make use of these grids.

Normally, structured grids are used to solve systems of partial differential equations (PDEs). To model complex flow fields, the computational domain is subdivided into multiple structured grids. Up to now, these multi-

block structured grids are one of the most frequently used kind of domain decomposition.

2.1 Topology

Multi-grid solvers based on finite volume methods are efficient and widely used. This approach, that solves Navier-Strokes equations with large number of unknowns, makes use of a multi-block grid, which has often more than hundreds blocks.

Multi-block grids can be classified in different types by considering the boundary connectivity. The most common type is the matching type. Each interface node has a corresponding node of the connected neighbor block. This guarantees the continuity of the physical field at the block interfaces. In the case of local refinement, a discontinuity of the solution might appear. In the most general case, the boundaries of neighboring blocks are displaced and the partitioning is locally refined, i.e. in general, there are no common points anymore.

2.2 The Engine Multi-Grid Dataset

We used a relatively simple full-matching multi-block dataset. However, it contains all properties needed to develop a software framework which manages arbitrary multi-block datasets. The dataset describes the inflow of fluid in a spark ignition engine with 4 valves [1]. The simulation includes the filling and the compression phase but not the ignition itself. The simulation domain is decomposed in 23 single blocks (Figure 1). Furthermore, the complete simulation process is divided into 63 discrete time levels.

The multi-block grid geometry is changed to fit into the domain. During the compression simulation (from time level 35 onward) the inlet valves are closed and not considered, and therefore, block 18 to 23 are removed. Moreover, for each time level, single blocks can change their resolution. Thus, the whole multi-block grid consists of 114.998 nodes in time level 1 and can inflate to 245.588 nodes for time level 34. This leads to a total dataset of 365 MByte. In view of actual data sizes of current CFD simulations, which can reach some hundred Giga-Bytes, this engine dataset seems to be slightly small.

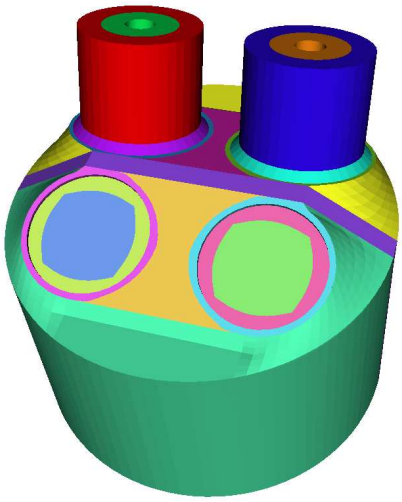

Figure 1: Single colored blocks.

With regard to restricted memory size, file server access time, and small network bandwidth, the requirement to exploit multi-block structures for speeding up calculations is obvious; i.e., it is essential to make use of topology information to get data on demand just needed for required computations.

2.3 Multi-Block Topology

During each particle integration step, it is possible that the particle leaves its current block. Then, the newly entered block and cell must be searched. This search is restricted to blocks of one common time level only. As soon as the integration time crosses a current time level, all currently used blocks must be replaced by blocks of the next time level. Because of the moving geometry, it cannot be guaranteed that the current position is still located in the same block number and cell.

Within structured grids, the neighboring cells are inherently determined. For the inter-block connectivity, our developed multi-block topology data structure (Figure 2) takes the place of this inherent topology. This multi-block topology is used to find neighboring blocks and cells rapidly.

Only blocks with connected cell sides are considered as neighbors. If only a cell edge or cell vertex is shared, this connection is not

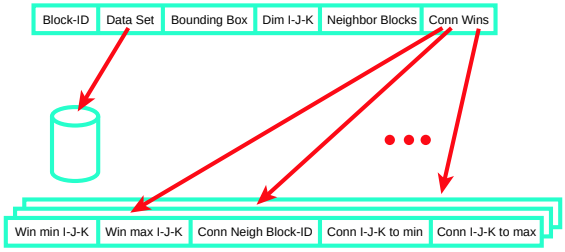

Figure 2: Data structure for multi-block topology information.

considered, although a particle might leave a current block and enters an edge or vertex connected block. Not each single cell connection is stored. In general, a group of cell sides of one block spawn a window to another group of cell side of a second block. These windows are stored and can now be used to determine the next block and block cell directly.

If the chosen block does not contain the new particle position, a search over all remaining blocks is started. In order to reduce time and memory, only block candidates should be loaded into memory. These are determined by a bounding box test, which checks the new particle position against pre-calculated bounding boxes. These bounding boxes consist of ranges of the block extension in all three dimensions of the physical space. They are sometimes rather rough assumptions of the comprised region.

If this test also fails, the particle has left the whole dataset. Normally, this can be recognized before the bounding box test, since each block boundary knows whether it is connected with a neighboring block or whether it is a boundary of the whole dataset. Moreover, information is stored about the kind of dataset boundary. Therefore, viscosity boundary conditions (slip and no-slip), inflow or outflow boundary conditions, and other conditions can be assigned. In our test case, the walls of the combustion chamber consist of no-slip boundaries (zero velocities are assigned to all except the bottom surface, which has got the velocity of the connected piston) and two inflow boundaries (at the valves). Therefore, particles can enter but never leave the engine again!

3 Particle Tracing

The Lagrangian formulation of flow fields depicts the position of material particles. This formulation is needed in order to compute path lines, also denoted as particle traces or as particle advection. A path line is the numerical solution of an ordinary differential equation (ODE) described as an initial value problem. The initial value is the seed point. While the integration makes use of preceding particle positions, the particle trace cannot be parallelized piece by piece. Computing the motion of particles through a flow field is governed by the following equation:

$$\frac{dx}{dt} = v'(x,t) \tag{1}$$

3.1 VTK's Streamline Approach

VTK does not support unsteady CFD post-processing. It manages single datasets only. For this, a variety of steady flow extraction algorithms, like the calculation of cut-planes, iso-surfaces, and streamlines are implemented. These basic extraction algorithms can now be used for the extension to the unsteady case. For instance, the class `vtkStreamLine` and its super-class `vtkStreamer` have been used as the basis of our path line computation. In this chapter, we first explain the behavior of the original VTK classes before we describe our extension.

After determination of all seed points, `vtkStreamer` calls `ThreadedIntegrate()`, the main integration function to compute streamlines. The most important parameter that can be set by the application is the integration step length. The determined integration step size, the particle position, and the particle velocity are passed to the particle integrator. VTK offers Runge-Kutta second order and fourth order to solve initial value problems. The Runge-Kutta integrator as well as the `vtkStreamer`-object make use of `vtkInterpolatedVelocityField`, which controls the access to the dataset. It is utilized whenever someone wants to acquire a velocity vector or scalar value at a certain position of

the flow field; i.e., it hides the discrete nature of the respective dataset.

Furthermore, `vtkInterpolatedVelocityField` addresses the fact that searching a cell within an arbitrary dataset is a time-consuming task, and the particle integration often stays in one cell for a long time. Therefore, it caches the current cell and assumes that this cell will be the current cell for the next integration step. This speeds up streamline calculations considerably. The current cell can also be set by the application, and the cell caching can be enabled and disabled.

The most important method to compute fourth-order Runge Kutta (Equation 2), which is offered by the class `vtkRungeKutta4`, is `ComputeNextStep()`. It expects the current particle position x_i and the velocity $v(x_i)$ at that position. Furthermore, it needs the step size h and the elapsed time the particle is travelled, which subsequently is ignored by VTK's streamline implementation.

$$
\begin{aligned}
k_1 &= v'(x_i, t_i) \\
k_2 &= v'(x_i + h/2 \cdot k_1, t_i + h/2) \\
k_3 &= v'(x_i + h/2 \cdot k_2, t_i + h/2) \\
k_4 &= v'(x_i + h \cdot k_3, t_i + h) = v'(x_i + h \cdot k_3, t_{i+1}) \\
\\
x_{i+1} &= x_i + h \cdot \left(\tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4\right)
\end{aligned}
\tag{2}
$$

3.2 Step Size Control

Generally, reducing the step size improves the solution by means of decreasing the truncation error; however, more iterations are needed and the accumulated rounding error increases. Therefore, one should not use more steps than needed for an accurate solution. For adequate ODE computations, the integration algorithm should apply an adaptive integration step size control. [2]

VTK applies an integration scheme similar to one described by David Lane [3]. VTK uses the current speed of the considered particle and the diameter of the cell where the particle is just located. In order to measure the diameter of the cell, the bounding box diagonal is computed. Furthermore, the step size is controlled by a user-defined constant, which defines the fraction of a cell size to be considered.

David Lane additionally proposes that smaller steps should be used for grid regions where the flow velocity varies rapidly. However, VTK does not make use of global information nor does it consider truncation errors. Other approaches estimate this error in order to achieve the next step size. A rather straightforward approach is step doubling [4]. The recent step size is used to compute a first estimation of the next particle position. Thereafter, for the same step, only half the size is used and applied twice. The distance from the first to this second obtained particle position can be used as an indication for the truncation error. If this difference is too large or too small, the step size must be adjusted.

Darmofal and Haimes measure the angle between velocity vectors at successive particle positions [5]. If the direction changes over 15 degrees, the step size is halved, whereas the step size is doubled if it is less than 2 degrees. Kenwright and Lane [6] describe a similar scheme, which measures the angle between successive line segments on a path line (Equation 3). Through experiments they found out that their algorithm with 12 degrees upper and 3 degrees lower limits is more accurate than the step doubling scheme.

$$
\cos\theta_i = \frac{(x_{i-1} - x_i) \bullet (x_i - x_{i+1})}{|x_{i-1} - x_i|\,|x_i - x_{i+1}|} \tag{3}
$$

Friedrich Weller [2] suggests calculating a value q by using the already computed weights k_1 to k_3 of Runga-Kutta formulas:

$$
q = \left| \frac{k_3 - k_2}{k_2 - k_1} \right| \tag{4}
$$

Furthermore, he suggests to double the step size if $q < 0.005$ and halve the step size if $q > 0.04$. The preceding step is then recalculated with this new step size. For $0.005 < q < 0.04$ the step size remains unchanged and the next step can be computed. In some cases, k_1 and k_2 are equal, thus, $|k_2 - k_1|$ is directly set to 0.01.

Instead of step doubling as mentioned above, Press preferred the Runge-Kutta-Feldberg method [4]. It is a robust embedded Runge-

Kutta formula and roughly a factor of two more efficient than algorithms based on step doubling. Press uses fifth-order Runge-Kutta (Equation 5) and compares the result with the embedded fourth-order formula (Equation 6).

$$
\begin{aligned}
k_1 &= h \cdot v'(x_i, t_i) \\
k_2 &= h \cdot v'(x_i + b_{21}k_1,\ t_i + a_2 h) \\
k_3 &= h \cdot v'(x_i + b_{31}k_1 + b_{32}k_2,\ t_i + a_3 h) \\
k_4 &= h \cdot v'(x_i + b_{41}k_1 + b_{42}k_2 + b_{43}k_3,\ t_i + a_4 h) \\
k_5 &= h \cdot v'(x_i + b_{51}k_1 + b_{52}k_2 + b_{53}k_3 + b_{54}k_4, \\
&\quad t_i + a_5 h) \\
k_6 &= h \cdot v'(x_i + b_{61}k_1 + b_{62}k_2 + b_{63}k_3 + b_{64}k_4 + \\
&\quad b_{65}k_5,\ t_i + a_5 h)
\end{aligned}
$$

$$
x_{i+1} = x_i + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 \tag{5}
$$

$$
x^*_{i+1} = x_i + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 \tag{6}
$$

The used constants are found by Cash and Karp [4] and are slightly more efficient than Feldberg's original ones. The difference between both results is used as an error estimate and controls the modification of step sizes.

We have implemented the schemes by Kenwright and Lane, Weller, and Feldberg and compared the results with the original VTK results. The angle approach breaks off after some iteration because of the high dynamic of the engine dataset. Illegal deviations are recognized too late and dividing the step size confirms the deviations only. The other two approaches work well. The great advantage is that they merely need a few additional computations in order to verify the truncation error. Weller's approach is faster since it needs less correcting steps.

3.3 Extension to Pathlines

We inherit our path line classes from `vtkStreamline` and `vtkStreamer`. In order to enable multi-block topology, `vtkInterpolatedVelocityField` must be adapted. The adaptive Runge-Kutta schemes are based on `vtkRungeKutta4`. The main modifications are built into the integration method of `vtkStreamer`.

For every step, different computations need one or more flow field positions. Therefore, a discrete time level must be selected in which the particle should be located. Then, the current cell, that bounds the given position, is searched, and thereafter, the position is weighted against the cell nodes, where the discrete data is stored. These weights can now be used for tri-linear interpolation in order to obtain the needed values at a given position. VTK uses the Newton-Raphson method for the weight determination. Kenwright and Lane depict the numerical problem of straightforward tri-linear interpolation and recommend a more efficient approach to find cell and weights. Thus, the hexahedral cells of the grid are decomposed in tetrahedrons. Fortunately, this grid decomposition does not have to be carried out in advance but can be performed locally whenever a hexahedral cell is entered [6].

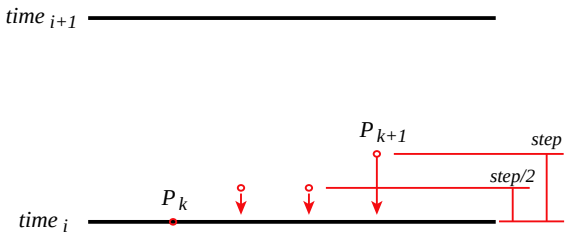

Figure 3: Time information is ignored and mapped to current time level.

In order to compute path lines in unsteady flows, one does not only need position interpolation but also interpolation over time. A straightforward approach just changes few VTK streamline computation code. VTK ignores the time information and maps all Runge-Kutta computation steps to the last discrete time level (Figure 3). In order to enhance this behavior to path lines, a second probe is applied using the last time level plus one. Thereafter, the real particle position is interpolated between the obtained two probes considering the last particle's integration time, which is a continuous time stamp. However, one must consider that both probes typically have different step sizes. Therefore, these values are also interpolated according to the last particle time stamp. Finally, VTK uses the dis-

tance between last and new particle and their speed to retrieve the real applied step size.

4 Parallelization

Our test case starts with 8 particles on a seed line below one of the outlet valves. The start time is at level 14 and all path lines are integrated to level 22. The two probe path line integration and the step-size control schemes listed above are applied. The results are matched visually and by distance comparison of the end positions of each path line. Furthermore, the complete computation time and the needed time steps are measured. One result is depicted in figure 4.

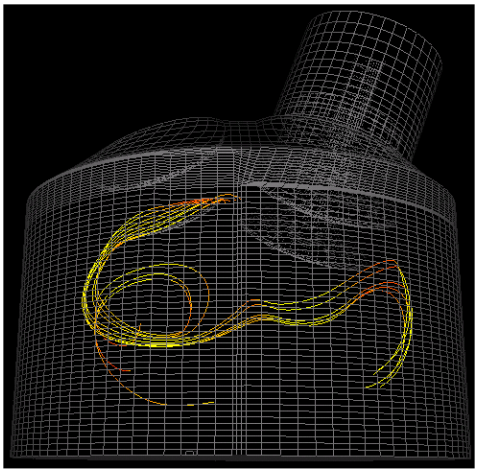

Figure 4: Path lines computed by Runge-Kutta-Feldberg scheme.

VTK already offers multi-threading capabilities for streamline calculation. For a more generalized approach, applicable in shared memory systems as well as in distributed memory systems, we integrated the path line extraction in the parallelization framework of our Virtual Reality toolkit ViSTA [7]. This framework is based on the Message Passing Interface MPI. In order to relieve the visualization system, the CFD extraction computation is completely shifted onto a high-performance multiprocessor system denoted as work host. The visualization host can now send a request to the work host in order to compute path lines. On the work host, each single path line is computed by one processor denoted as work node. The resulting path lines are sent back to the visualization host, where they can directly be displayed as polylines or moving particle traces.

Tests have been run on a high-performance Sun Fire 6800 SMP cluster from SUN Microsystems. It consists of 24 UltraSPARC-III 900 MHz processors connected to a shared memory. Each process of the Sun Fire 6800 can access up to 24 GByte shared main memory. The MPI implementation on the Sun Fire 6800 is optimized for SMPs, i.e. data packages are passed via shared memory.

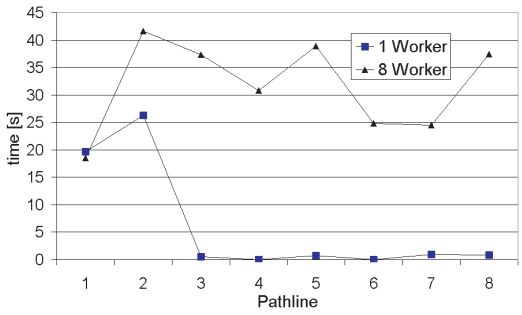

Figure 5: Loaded blocks using 1 processor (sequential load) and 8 processors (parallel load) to compute 8 pathlines applying the VTK scheme with a cell integration step length of 0.001.

A considerable time-consuming task is data loading from a file server. Because each work node reads its own data, a variety of blocks are read multiple times instead of being read once and shared by a data management system. Nevertheless, this is the unavoidable drawback in distributed systems like Linux clusters, which we have tested as well. The number of multiple loaded blocks depends on the seed points. In our measurements, the amount is rather high (see Figure 5).

Figure 6 shows the parallelization result. The speed-up is not entirely satisfying. The reason is the time needed to read single data blocks from file server. Without measuring the loading time (Figure 7), the result is the expected speed-up. Time needed to send data from one worker to another can nearly be ignored. Nevertheless, communication time slightly reduces the optimum efficiency.

Figure 6: Needed computing time subdivided in essential parts.

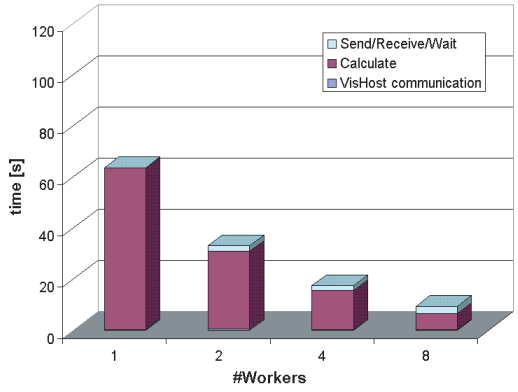

Figure 7: Needed computing time with prefetched data.

5 Conclusion

Our parallelization approach has the drawback that data must be read multiple times from hard disk. In our test case, this reduces the total speed-up considerably. Instead of managing CFD data access by each worker node, a central data manager could offer already loaded data blocks directly and would have load data files from a file server only once. Such centralized managing is the obvious approach for SMP clusters like the used Sun Fire cluster but more complicated for distributed clusters. Nevertheless, larger datasets and less main memory make it useful to develop an integrated data management in our parallelization framework. Technologies like parallel I/O and MPI-IO promise advantages even for distributed systems. Therefore, we will investigate and integrate a more sophisticated data management for parallel environments.

Considering topology information is the main strategy in order to accomplish a really considerable speed-up even in sequential path line computations. In our Virtual Environment, real-time interaction is achieved and the time to wait for CFD extraction results is acceptable. Additionally, using an adequate adaptive step-size control can avoid unnecessary attempts to find usable solutions and reduces the number of needed integration steps. Furthermore, the path lines are substantially more accurate than the VTK scheme because we use an approximately optimal number of steps for the numerical solution and additionally reduce rounding errors.

References

[1] Aschaf Abdelfattah. "Numerische Simulation von Strömungen in 2- und 4-Ventil-Motoren", PhD thesis, RWTH Aachen, 1998.

[2] Friedrich Weller. "Numerische Mathematik für Ingenieure und Naturwissenschaftler - Eine Einführung für Studium und Praxis". Vieweg, 1996.

[3] David A. Lane. "UFAT - A Particle Tracer for Time-Dependent Flow Fields", in: D. Bergeron, A. Kaufman (Eds.), Proceedings of Visualization, 1994.

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. "Numerical Recipes in C++", 2nd edition, Cambridge University Press, 2002.

[5] Dave L. Darmofal, Robert Haimes. "Visualization of 3-D Vector Fields: Variation on a Stream", Proceedings, AIAA 92-0074, 30. Aerospace Sciences Meeting and Exhibit, Reno, Nevada, January 1992.

[6] David N. Kenwright, David A. Lane. "Optimization of Time-Dependent Particle Tracing Using Tetrahedral Decomposition", Proceedings of Visualization, 1995.

[7] Andreas Gerndt, Thomas van Reimersdahl, Torsten Kuhlen, Christian Bischof. "A Parallel Approach for VR-based Visualization of CFD Data with PC Cluster". Proceedings, IMACS 2000, Lausanne, Swiss, 2000.