Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets

Andreas Gerndt[1], Samuel Sarholz[2], Marc Wolter[1], Dieter an Mey[2], Christian Bischof[2], Torsten Kuhlen[1]

[1]Virtual Reality Group, RWTH Aachen University, Germany
[2]Center for Computing and Communication, RWTH Aachen University, Germany
{gerndt, sarholz, wolter, anmey, bischof, kuhlen}@rz.rwth-aachen.de

Abstract

Extraction of complex data structures like vector field topologies in large-scale, unsteady flow field datasets for the interactive exploration in virtual environments cannot be carried out without parallelization strategies. We present an approach based on Nested OpenMP to find critical points, which are the essential parts of velocity field topologies. We evaluate our parallelization scheme on several multi-block datasets, and present the results for various thread counts and loop schedules on all parallelization levels. Our experience suggests that upcoming massively multi-threaded processor architectures can be very advantageously for large-scale feature extractions.

Keywords: Flow Field Topology, Nested Parallelization, Multi-threading, CFD Post-Processing, Virtual Reality

1. Introduction

Helman and Hesselink were one of the first who described approaches to visualize the topology of flow fields computed by Computational Fluid Dynamics (CFD) [1]. The most important step is to find all locations where the velocities vanish. These positions are also called critical points. Once detected, they can be classified, and separation lines may be computed to illustrate the division of the flow field into homogeneous segments. In the meantime, several authors presented a variety of improvements. A brief overview of common algorithms for vector field topology extraction is given by [2].

Due to insufficient depth perception and restricted interaction schemes, the topology analysis of 3-dimensional, turbulent flow fields on desktop graphics workstations is often time-consuming and error-prune. Virtual environments, however, may offer methods for a clearly improved explorative analysis [3].

One fundamental demand in virtual environments is real-time interaction. Therefore, the post-processing of large-scale and unsteady flow fields should be carried out by high-performance clusters in order to relieve the visualization front-end and to speed-up the feature extraction.

Complex flow fields are frequently decomposed into several grids. In the case of first-order critical point computation, the used algorithm is a cell-based approach, i.e. no additional information of neighboring cells is needed. Moreover, the bisection scheme introduced by Globus et al. [4] works on sub-cells which can also be processed locally. Therefore, the hypothesis of the work presented in this paper is that we might achieve significant speed-up by the use of nested parallelization on all depicted levels: time level, block level, cell level, and sub-cell level. The goal is to find the best distribution of processes to these levels for an optimum balancing and scaling.

The paper is structured as follows: In Section 2 the implemented iterative bisection algorithm for the identification of vector field topologies is described. Afterwards, datasets used to evaluate the critical point extraction are examined. Section 4 presents strategies for nested parallelization by means of OpenMP. Finally, experimental results are presented and evaluated in Section 5.

SC2006 November 2006, Tampa, Florida, USA

0-7695-2700-0/06 $20.00 ©2006 IEEE

2. Critical Points in Vector Fields

In this paper, only the bisection scheme published by Globus et al. is considered which merely handles first-order critical points. As this approach is based on structured grids, most of the following formulas are mainly valid when handling hexahedral cells. Higher-order approaches for 2D triangle meshes using the Clifford Algebra are presented for instance by Scheuermann et al. in [5]. As an extension, Mann and Rockwood describe a 3-dimensional method in [6]. Their algorithm is rather similar to the approach presented here except for the way to determine candidate cells.

2.1. Critical Point Position

Scalar and vector fields show topological structures defined by characteristic features. For scalar fields, these features are local minimum and maximum points. These extrema are also called critical points or singularity points of the scalar field (cf. [7]). Usually gradients are computed in order to determine the entire topological description, which again results in a vector field analysis.

For flow fields, one is particularly interested in the investigation of the velocity field, which is defined usually by vectors stored at the nodes of the grid. A critical point in a vector field is a position where the velocity vector shows zero length. In a first step, the position of a critical point must be determined.

Since the flow field is simulated on the basis of a discrete grid, only velocities at the vertices of a grid cell can be used to estimate whether a critical point lies within the cell or not. For structured datasets, the basic cell type is a hexahedron. In general, linearity is assumed between cell nodes so that tri-linear interpolation can be utilized in order to determine the velocity at an arbitrary position within a cell.

Furthermore, we have to transform a curved hexahedron from physical space ($\mathcal{P}$-space) into computational space (C-space). Thereafter, the next steps only work with unit cubes defined in natural coordinates. Let $x_{i,j,k}$ be the cell node with the lowest index in C-space and let $f(x_{i,j,k})$ be the velocity at that vertex, then the tri-linear interpolation function parameterized by α, β, γ within a range of [0,1] is defined as follows:

$$TL(\alpha,\beta,\gamma) = \sum_{I,J,K=0}^{1} f(x_{i+I,j+J,k+K}) \cdot \psi_I(\alpha)\psi_J(\beta)\psi_K(\gamma); \quad \psi_0(\alpha) = 1-\alpha;\ \psi_1(\alpha) = \alpha \tag{1}$$

A 3D critical point is found where the following equation holds:

$$TL(\alpha,\beta,\gamma) = (0,0,0)^T$$

However, finding roots analytically using Equation (1) is hardly practicable. Instead, the Newton-Raphson iteration is usually applied (cf. e.g. [8]). As vectors and matrices occur, the Newton-Raphson formula can be rearranged and than looks as follows:

$$x_{i+1} = x_i - \left(f'(x_i)\right)^{-1} f(x_i)$$

This approach is a predictor-corrector method and is applied until the distance between the position x_i and x_{i+1} is below a threshold. In order to find the roots of the tri-linear interpolation function, we also need its derivative:

$$TL'(\alpha,\beta,\gamma) = \begin{pmatrix} \partial TL/\partial\alpha \\ \partial TL/\partial\beta \\ \partial TL/\partial\gamma \end{pmatrix}^T \tag{2}$$

The partial derivatives may easily be determined analytically by using the polynomial form of Equation (1). Also the computation of its inverse is rather straightforward as TL' is non-singular. Generally, the inverse of a non-singular 3-dimensional square matrix A is:

$$A^{-1} = \frac{1}{|A|}\begin{pmatrix} a_{22}a_{33}-a_{23}a_{32} & a_{13}a_{32}-a_{12}a_{33} & a_{12}a_{23}-a_{13}a_{22} \\ a_{23}a_{31}-a_{21}a_{33} & a_{11}a_{33}-a_{13}a_{31} & a_{13}a_{21}-a_{11}a_{23} \\ a_{21}a_{32}-a_{22}a_{31} & a_{12}a_{31}-a_{11}a_{32} & a_{11}a_{22}-a_{12}a_{21} \end{pmatrix}$$

There are, however, some restrictions when using the Newton-Raphson iteration. First, we need a "good" starting position x_0. Otherwise, the iteration might never terminate. In our case, we always use the cell mid-point in C-space as an initial position and iterate only until a certain maximum number of steps is reached.

Second, the approach will always find only one root although multiple roots within a cell are possible. Therefore, before applying Newton-Raphson iteration, a sub-division scheme called bisection is carried out dividing a hexahedron into eight sub-cells. The velocities at the sub-cell's vertices are again determined by tri-linear interpolation. Bisection allows not only the detection of multiple critical points but also improves the convergence of Newton-Raphson.

But not each hexahedral cell contains critical points. A heuristic exists which estimates whether a cell might include roots of the tri-linear function or not. These cells are called candidate cells and are exclusively considered for the subsequent bisection scheme. Other cells are discarded. The heuristic used here simply compares the *x*-, *y*-, and *z*-components of all velocity vectors stored at the cell vertices. If the signs of all components change in any comparison of these vectors, a candidate cell is found. For 2-D, the following figure depicts a candidate cell:

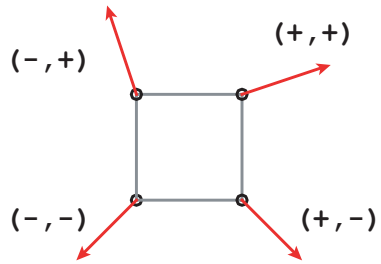

Figure 1. Sign test to determine 2-D candidate cells.

After one bisection step, the candidate cell test is again carried out. Sub-cells determined as candidates are used for further bisection steps. This iterative sub-division can now be applied until a certain bisection depth is reached or the velocities at all vertices of a sub-cell are below a threshold. After the bisection terminates, the actual critical point position can now be determined by the Newton-Raphson iteration.

2.2. *C*-Space Transformation

The algorithm described works only if the hexahedron cell is aligned along the coordinate axes and if all its edges have length 1. This is the reason to convert each cell first of all from $\mathcal{P}$-space to *C*-space. The transformation is defined by vertex positions of each cell and can be described by the Jacobian:

$$J_x = \nabla x = \begin{pmatrix} \partial x_x/\partial x & \partial x_x/\partial y & \partial x_x/\partial z \\ \partial x_y/\partial x & \partial x_y/\partial y & \partial x_y/\partial z \\ \partial x_z/\partial x & \partial x_z/\partial y & \partial x_z/\partial z \end{pmatrix}$$

The function x transforms an arbitrary point from $\mathcal{P}$-space to *C*-space. It is not a continuous function. Cell vertices are simply mapped onto the corresponding discrete *i,j,k*-coordinates. Within cells, $\mathcal{P}$-space positions, still determined by tri-linear interpolation, are mapped onto $i+\alpha$, $j+\beta$, $k+\gamma$. Therefore, the so-called continuous Jacobian (cf. [9]) is merely defined for one cell and consists of partial derivatives of Equation (1):

$$J_x(\alpha,\beta,\gamma) = \begin{pmatrix} \partial TL_x/\partial\alpha & \partial TL_x/\partial\beta & \partial TL_x/\partial\gamma \\ \partial TL_y/\partial\alpha & \partial TL_y/\partial\beta & \partial TL_y/\partial\gamma \\ \partial TL_z/\partial\alpha & \partial TL_z/\partial\beta & \partial TL_z/\partial\gamma \end{pmatrix}$$

This formula is equivalent to the derivation of the tri-linear function defined in Equation (2) but now vertex positions instead of velocities are used as input values. The Jacobian can be applied for *C*-space to $\mathcal{P}$-space transformation of velocities defined at cell vertices:

$$v = \nabla x \cdot u$$

Velocity vectors are denoted by v in $\mathcal{P}$-space and by u in *C*-space, respectively. We are, however, interested in the reverse transformation which is just the inverse Jacobian (cf. [10]):

$$u = \nabla x^{-1} \cdot v$$

2.3. Critical Point Classification

Critical points classify the flow field behavior in their immediate vicinity. For the assessment, the velocity gradient tensor is needed, which is defined as the Jacobian J_v of the velocity field.

$$J_v = \nabla v = \begin{pmatrix} \partial v_x/\partial x & \partial v_x/\partial y & \partial v_x/\partial z \\ \partial v_y/\partial x & \partial v_y/\partial y & \partial v_y/\partial z \\ \partial v_z/\partial x & \partial v_z/\partial y & \partial v_z/\partial z \end{pmatrix} \quad (3)$$

For trilinear hexahedral elements, eight shape functions N_i (one for each node) can be determined by means of Equation (1) [11]. Let v_i be the velocities at the nodes, than they can be directly applied to compute the interpolated velocity field within a cell:

$$v(\alpha,\beta,\gamma) = N \cdot v = \sum_{i=0}^{7} N_i(\alpha,\beta,\gamma) \cdot v_i$$

This equation can now be used to compute the velocity gradient tensor given in Equation (3) for a certain position.

The final step for the classification is to compute the eigensystem of the yielded Jacobian J_v. The result consists of three real eigenvalues and eigenvectors, or two complex conjugate and one real eigenvalues and eigenvectors, respectively. Exclusively real values identify nodes, otherwise spirals are detected [12].

For 2-D flow fields, streamlines along real eigenvectors describes the complete topology of the velocity field. For the 3D case, these so-called separatrices just give an impression of the topology (cf. Figure 2). Separating surfaces [13] or connectors [12] are more accurate. This paper, however, does not cover the computation of the entire topology structure.

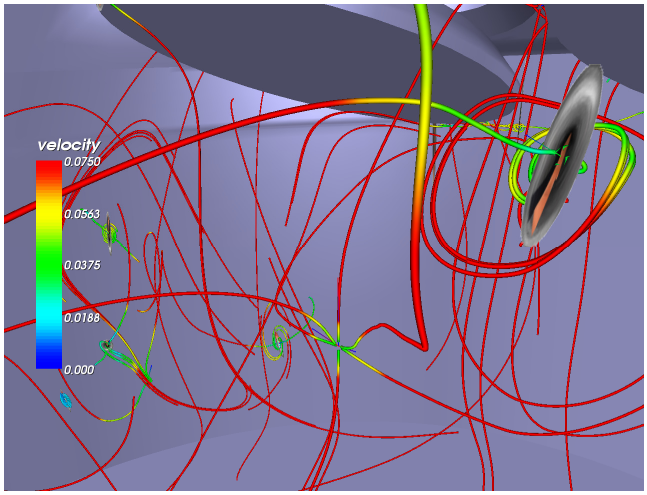

Figure 2. Engine, critical point symbols and separatrices.

3. Datasets

A variety of datasets was utilized in order to explore the behavior of the critical point computation algorithm. The specifications of the four presented datasets are depicted in Table 1.

The smallest but most interesting dataset is a spark ignition engine where only the inflow and compression phase were simulated. It is a multi-block (MB) dataset where the chamber and the inlet valves are decomposed into 23 structured grids (cf. Figure 3). The position and the number of vertices and cells may change between two successive time levels. From time level 35 onwards, the valves are closed and were not considered by the simulation anymore so that the dataset is reduced by the 6 concerned blocks to 17 blocks.

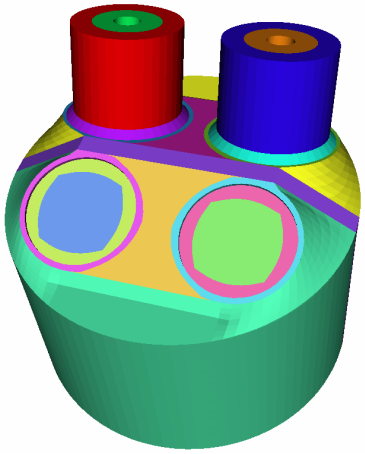

Figure 3. Differentially colored blocks of the Engine multi-block dataset.

For all datasets, the total amount of occurring critical points was computed. In Figure 4 the found critical points of the engine dataset are presented per time level.

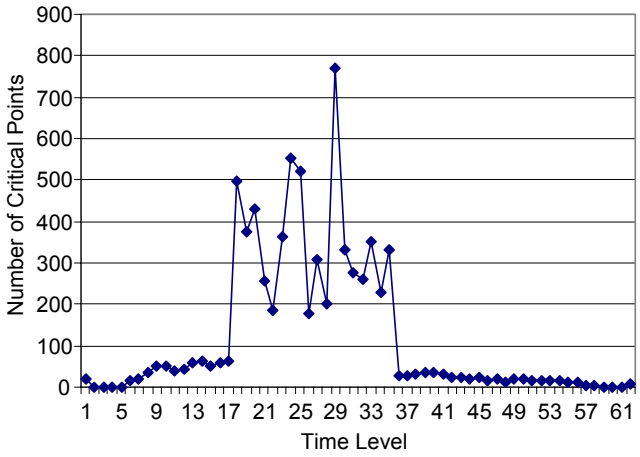

Figure 4. Engine, amount of critical points.

	Courtesy by	Time Levels (TL)	Blocks / TL	Cells / TL	Total Blocks	Total Size
Engine	AIA	62	23 / 17	37 170 - 191 960	1 318	365 MB
Propfan	DLR	50	144	2 373 600	7 200	5 137 MB
Dual Vortex	AIA	152	8	2 359 296	1 216	12.6 GB
Shock	AIA	919	1	1 901 592	919	73.4 GB

Table 1: Specification of used datasets

One can see that critical points are primarily detected during the inflow process (time level 1 to 34). To identify the cost of the bisection approach, the runtime for the basic load (cell traversal, *C*-space transformation and initial candidate cell test) as well as the total runtime were measured. The results for the engine are depicted in Figure 5. Whereas the initial load is independent of the selected bisection depth, the total runtime is not. The presented result was measured using a bisection depth of 6.

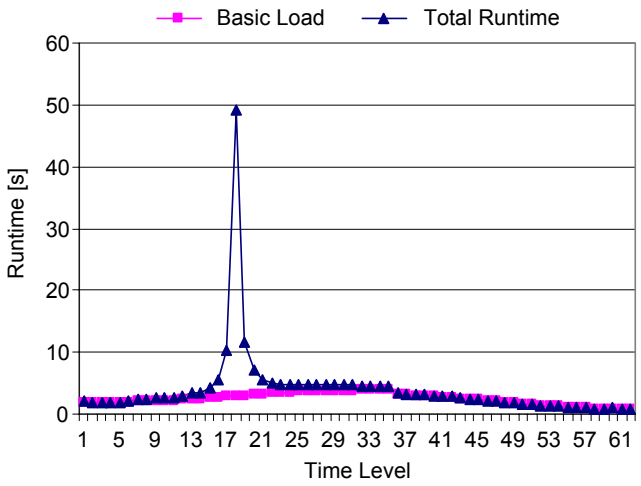

Figure 5. Engine, basic load and total computation load.

A high peak can be detected around time level 18. This is the situation where the valves stop and change their directions. Thereafter, they move against the inflow. For that situation, velocities close to the valve boundary are almost zero. Here, the bisection approach detects a large amount of sub-cells with opposite directions, which results in high bisection expense. The Newton-Raphson iteration needs also a lot of time as it does not converge quickly or not at all (cf. Figure 6).

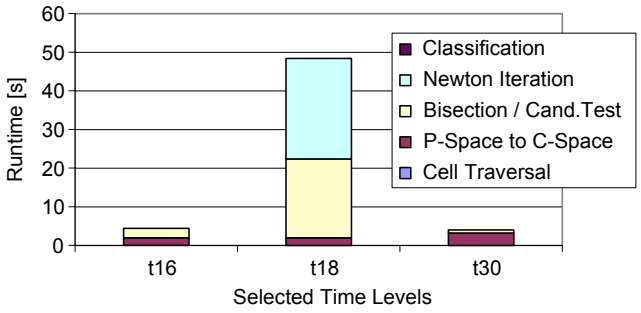

Figure 6. Engine, single processing steps for several time levels.

Finally, not only one location but a complete line of critical points is extracted. After time level 18, the next time levels (up to level 35) also identify critical point lines but because of the increasing velocity of the moving valves the algorithm is able to detect candidate cells more accurately. As a result, the runtime needed for bisection and Newton-Raphson iteration is considerably reduced.

Only some blocks (cf. Figure 7) are involved in the runtime peak around time level 18. This can cause balancing problems during parallel data extraction.

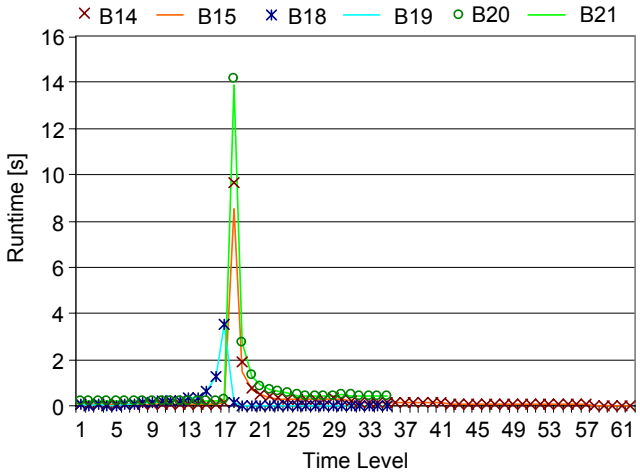

Figure 7. Blocks with highest load are depicted.

The second dataset is a counter-rotating propulsion turbine. Merely 12 blocks for one blade are used for the simulation (cf. Figure 8) as the flow field around each of the 12 blades is identical. For general post-processing, we completed the propfan dataset by replicating and rotating the original blocks, which yields the largest dataset per time level of the test suite with 2.4 million cells distributed among 144 block files (cf. Table 1).

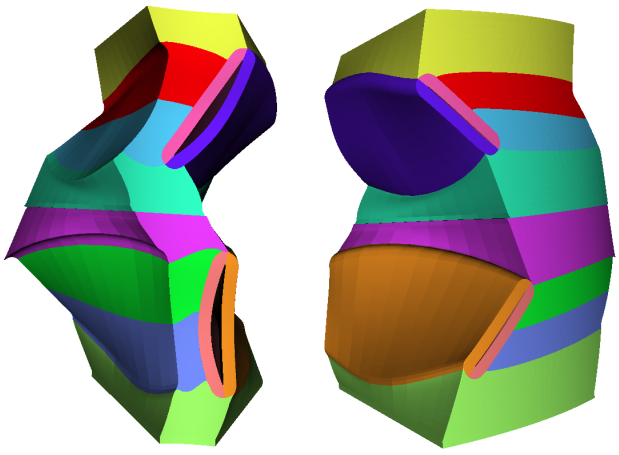

Figure 8. Single Blocks of one twelfth of the propfan dataset (left: inner view, right: outer view).

Again, the number of critical points (cf. Figure 9) as well as the basic load and the total runtime (cf. Figure 10) were

measured. But now, a bisection depth of 6 and 10 were used. Whereas the number of critical points and the basic load are not affected by the increasing depth, one can see that new small peaks for the total runtime appears at time level 11 and 37. The reason is similar to the case of the engine. The bisection does not follow just one subdivision path but searches in breadth. Instead of 5 bisections on average, now up to 2155 bisections are carried out.

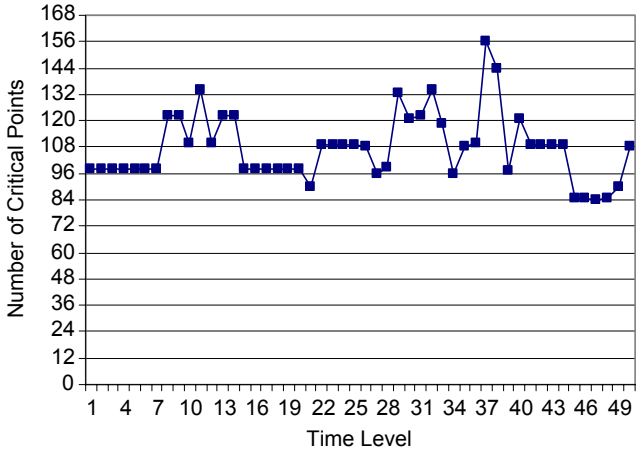

Figure 9. Propfan, amount of critical points.

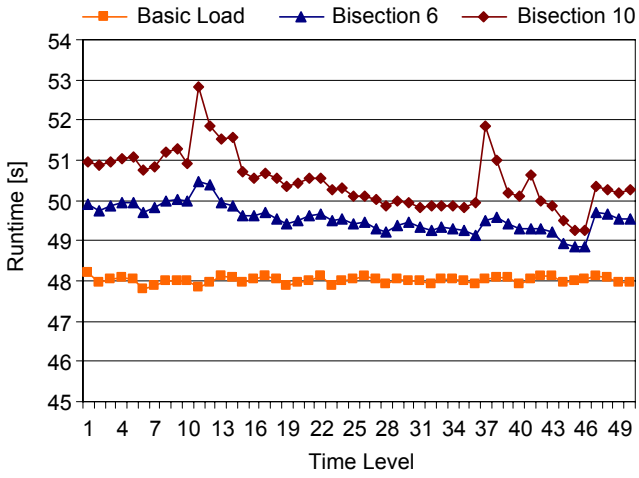

Figure 10. Propfan, basic load and computational load for bisection depth 6 and 10.

The third dataset used in this paper consists of less but larger blocks per time level. The dual vortex dataset shows two moving vortex tubes, which burst approximately from time level 100 onwards. The number of critical points (cf. Figure 11) does not increase when the vortices burst, but since finding the exact location is now more complicated, runtime is higher (cf. Figure 12).

Instead of the others, the shock dataset is not a multi-block dataset but a rectilinear one. It contains more time levels than the other datasets. Furthermore, the number of critical points increases over time (cf. Figure 13). Because of

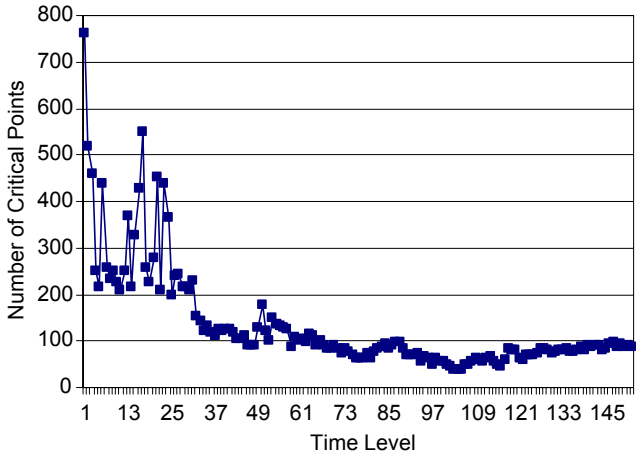

Figure 11. Dual Vortex, amount of critical points.

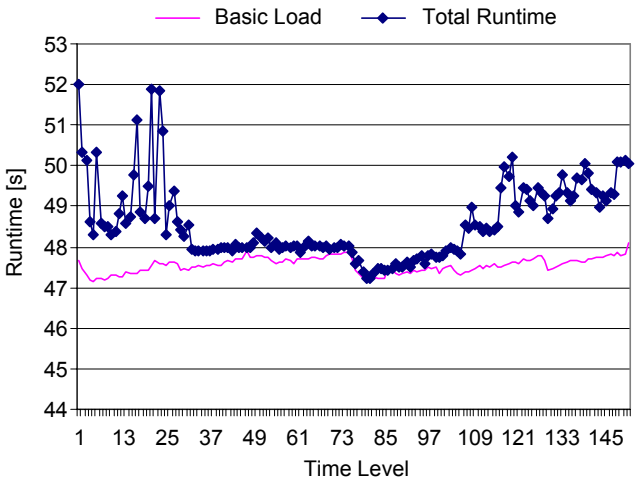

Figure 12. Dual Vortex, basic and total load.

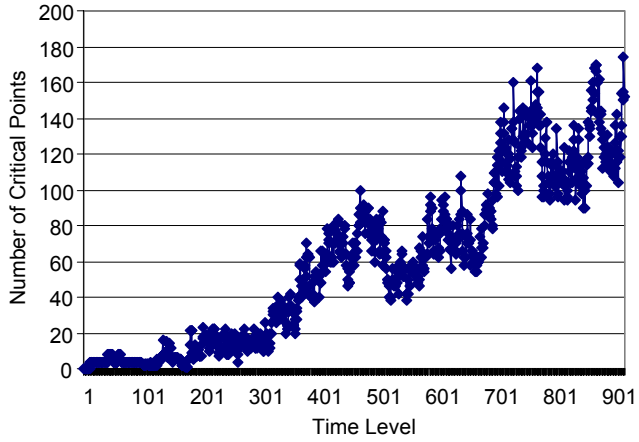

Figure 13. Shock, amount of critical points.

dominant flow direction and the rectangular shape of the cells, the bisection approach finds critical points accurately and fast (cf. Figure 14).

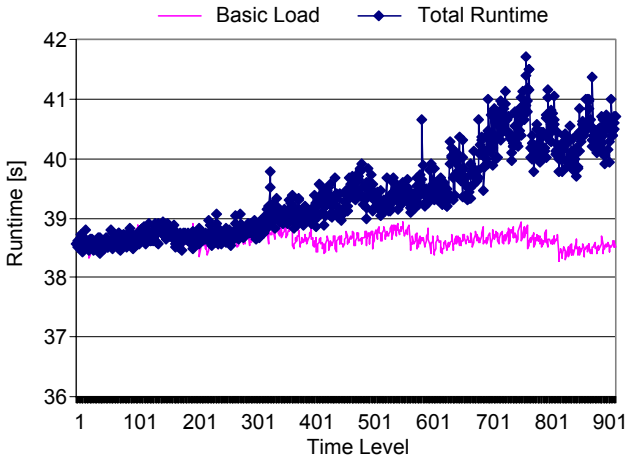

Figure 14. Shock, basic and total load.

Due to their various features, we thus consider these datasets good candidates for evaluating the parallel strategy presented in the following sections.

4. Parallelization

The following measurements were carried out on a Sun Fire E25K equipped with 72 UltraSparc IV dual-core processors, which is part of the supercomputer installed at the Center for Computing and Communication (CCC) of the RWTH Aachen University. The entire SMP cluster consists of 4 Sun Fire E25K nodes, 16 Sun Fire E6900 nodes, and 8 Sun Fire E2900 nodes.

A Linux cluster equipped with commodity graphics cards by NVidia operates a 5-side CAVE-like Virtual Reality display system [14]. For CFD post-processing purposes, a distributed software system has been developed. One part called ViSTA FlowLib [15] is responsible for the efficient, real-time rendering. The other part is the parallelization framework Viracocha [16], which runs at the HPC cluster and is applied for time-consuming CFD post-processing.

Viracocha is organized in several software layers. The lowest layer controls the communication among involved distributed components and the protocols, e.g. MPI, used. The middle layer manages incoming computation requests and running post-processing jobs. The top-most layer contains algorithm specific objects like the critical point computation parallelized with Nested OpenMP discussed in this paper.

Viracocha is available as a set of 32-bit libraries. As we are just interested in measuring the runtime of the parallel feature extraction, we always preloaded the entire dataset into the main memory. For handling large datasets, the available address space of a 32-bit framework, however, would be too restrictive which is why we took the algorithm module of Viracocha, compiled it as a 64-bit library version, and invoked the parallelized critical point computation from within a test suite. Consequently, the MPI layer of Viracocha was not available so that only OpenMP could be considered.

OpenMP [17] was designed for a straightforward parallelization of sequential codes for shared-memory systems by means of `pragma` directives. Loops are the primary target for distributing the computational work to multiple threads. The OpenMP parallel loop directive currently offers three different schedule kinds, which permit careful control of the distribution of work to threads. This is particularly useful if the loop iterations differ considerably in their computational costs. Such strategies are not offered, for instance, by MPI and it would be tedious to implement them. However, as will be explained later, we encountered load imbalances on all levels of parallelization that have been exploited for the critical point computation.

The simplest loop schedule is `static`, which causes an equal distribution of loop iterations to all threads. There is only a slight overhead involved but if the loop iterations heavily vary in their costs, such a schedule might lead to load imbalance. A chunk size parameter allows the bundling of iterations into chunks of the specified size as they are assigned to the threads. For `static`, chunks are distributed in a Round Robin manner.

On the other hand, OpenMP offers the `dynamic` schedule, which assigns iterations to the threads after they are done with their previous work. This eliminates any load imbalances as far as possible, but it also involves a higher scheduling overhead that primarily depends on the number of chunks.

The third schedule kind is the `guided` schedule, which is the second dynamic strategy to distribute work to threads. Whereas the `static` and the `dynamic` schedule always assign chunks of the given size – except for the last chunk of a loop, which may have fewer iterations – the `guided` schedule starts with larger chunks and then gradually reduces their size towards the end of the loop iteration space. The size of each chunk is proportional to the number of still unassigned iterations divided by the number of threads and is calculated with Equation (4). In this case, the chunk size parameter just specifies the minimum chunk size.

$$chunk\_size = \frac{\#unassigned\_iterations}{c \cdot \#threads} \qquad (4)$$

Sun's OpenMP implementation allows adjusting the weight parameter *c* in Equation (4) and uses 2 as a default value. A large weight parameter leads to a schedule that is similar to `dynamic`. A smaller *c* reduces the overhead but might increase the load imbalance.

In our algorithm, we heavily profit from nested parallelization which is well supported by the Sun OpenMP compiler. The mayor sources of parallelism are the three outermost loops processing time steps, blocks, and cells. Furthermore, we investigated the parallelization of the bisection algorithm on the sub-cell level. The latter aspect is only discussed in more detail for the engine dataset in Section 5.1.

Above all, we varied the number of threads and the loop scheduling on the three outer levels. The `dynamic` schedule seemed to be an adequate choice for higher parallelization levels as scheduling overhead should play a less dominant role. Furthermore, best load balancing may be achieved with a selected chunk size of 1. For the cell level, however, we expected that the `guided` schedule leads to an optimum compromise between scheduling overhead and load balancing. Therefore, we applied the following scheduling and chunk size parameters for our experiments:

```
#pragma omp parallel for num_threads \
    (nTimeThreads) schedule(dynamic,1)
for (curT=1; curT<=maxT; ++curT) {
#pragma omp parallel for num_threads \
    (nBlockThreads) schedule(dynamic,1)
  for (curB=1; curB<=maxB; ++curB) {
#pragma omp parallel for num_threads \
      (nCellThreads) schedule(guided)
    for (curC=1; curC<=maxC; ++curC) {
      FindCriticalPoints (
              curT, curB, curC);
} } }
```

Listing 1: Used scheduling for Nested OpenMP.

5. Results

Let *n* be the number of threads involved. We used up to 128 processors and measured all powers of two combinations of thread distributions to the parallelization levels. The amount of threads for the time level is denoted as ***ti*** and for block level as **b*j***. The number of threads assigned to the cell level denoted as ***ck*** is $n / (i \cdot j)$. For all but the engine dataset, we selected a maximal bisection depth of 10. The engine is only divided up to a depth of 6. The speed-up results obtained by the nested parallel critical point computation utilizing the engine dataset are depicted in Figure 15.

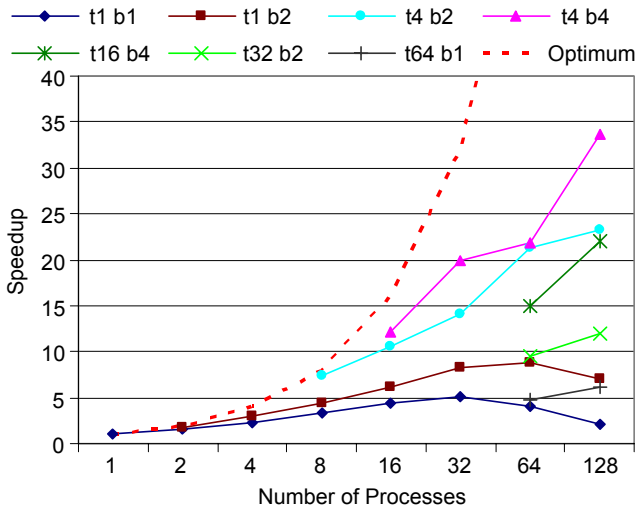

Figure 15. Engine, speed-up of selected probes.

Employing more than 4 threads on each of the two outer levels is not profitable because of the severe load imbalance caused by the peak in time level 18, block 20 and 21 (cf. Figure 7). On the other hand, scalability of the cell-based parallelization alone (cf. [t1,b1]) is limited as well. The speed-up drops when using more than 32 threads. Best results can be obtained by the combination of the three levels.

Looking at the propfan results (cf. Figure 16), the trend is similar but the speed-up is much better than for the previous dataset in all cases. A maximum is now reached at [t4,b16] and [t8,b16], respectively, and is close to the optimum speed-up. Due to a better load balance between time levels and due to a higher number of blocks, it is profitable to use more threads on the higher parallelization levels.

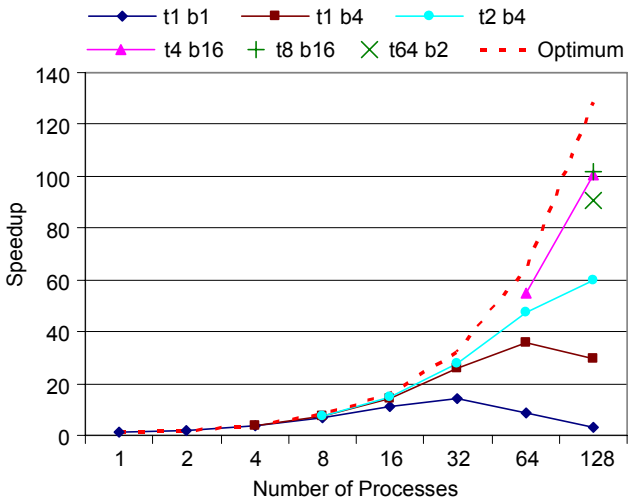

Figure 16. Propfan, speed-up of selected probes.

The speed-up is even better for the dual vortex dataset (cf. Figure 17) and the shock dataset (cf. Figure 18). The [t8,b2] and the [t32,b1] measurement, respectively, show almost perfect scalability. As the shock dataset only contains a single block, only two levels of parallelism are available.

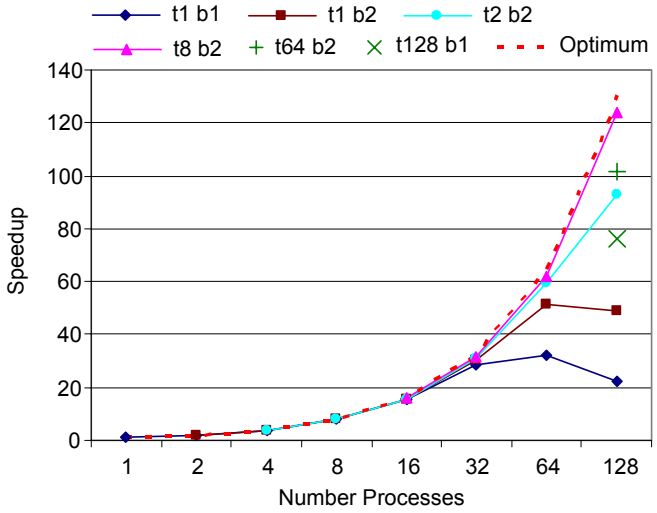

Figure 17. Dual Vortex, speed-up of selected probes.

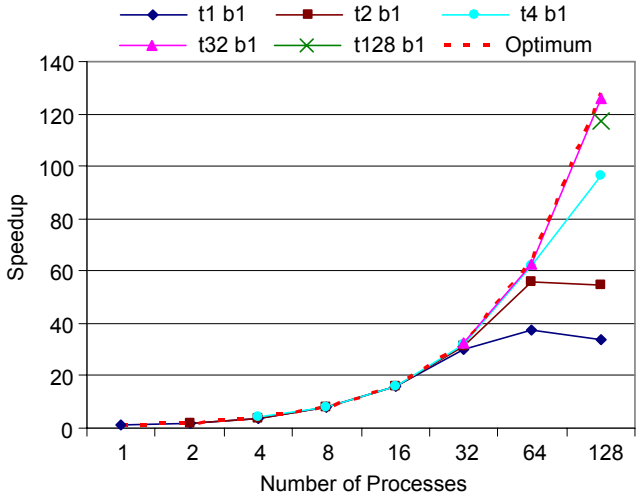

Figure 18. Shock, speed-up of selected probes.

5.1. Variation of Loop Schedules

In order to assess the measured runtimes, we also modified scheduling parameters and chunk sizes. If, for example, the simple `static` schedule is used on all three loop levels, the achieved speed-up drops to 30% (engine: 10.38) and 92% (shock: 116.19), respectively.

The engine dataset suffers from the heavy load imbalance introduced by block 20 and 21 of time level 18 (cf. Figure 7). In the following, we only consider this sub-problem. We deactivated the both uppermost parallelization levels and just looked at the cell and the sub-cell levels. As 4 threads could be profitably employed for the higher levels, as has been discussed before, only 8 threads are available on these lower levels. Running the sub-cell level with these 8 threads only leads to a speed-up of 2.36 compared to a performance gain of 4.16 when parallelizing the cell level.

Alternatively, we also modified the guided weight parameter (cf. Equation (4)). Figure 19 shows the result of parameter study when varying the number of threads and the weight factor of the `guided` schedule applied to the cell level loop. Using just 8 threads, a weight of 20 shows the best speed-up. This modification led to an improvement of the speed-up from 4.16 to 6.37. Furthermore, applying the weight factor 20 for the entire engine dataset, the overall speed-up using 128 processors ([t4,b4,c8]) could be improved from 33.67 (cf. Figure 15) to 55.18.

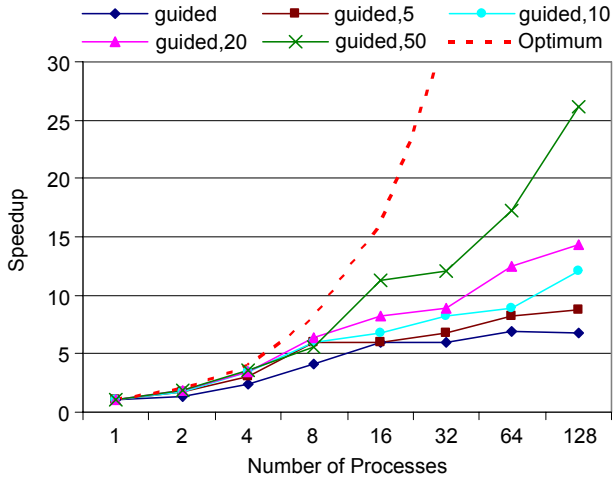

Figure 19. Engine, varying weights for `guided` processing block 20 and 21 at time level 18.

6. Conclusions and Future Work

Multi-level parallelization of the critical point computation using OpenMP was very successful for all investigated datasets. Using 128 threads, the worst case led to a speed-up of 55.18. For all the other datasets, measurements indicate that even larger shared-memory systems than the 144 core Sun Fire E25K would have been beneficial.

For instance, the elapsed time of the critical point computation for the dual vortex dataset could be reduced from 2.5 hours down to 73 seconds. The calculation related to the shock dataset, which took 12 hours and 18 minutes sequentially before, was reduced to about 6 minutes, a tremendous reduction in runtime. For data exploration in virtual environments, usually not all but a subset of time

levels is applied. Therefore, a stride of 10 time levels for the shock dataset results in a highly satisfying processing time of only 35 seconds. This is fairly acceptable for large-scale data exploration with real-time systems.

Nested parallelization offers several degrees of freedom in choosing the number of threads and the loop schedule on each level. Based on numerous experiments, it is possible to develop first heuristics to determine a reasonable thread count for all three levels based on the number of time steps, blocks, and cells. These, however, do not allow complete elimination of all kinds of load imbalances as described for the engine dataset. Therefore, we are currently working on strategies for automatic adjustment of the available parallelization parameters.

Another promising approach might be the usage of the task queuing work-sharing construct, which is particularly suitable for the parallelization of the bisection algorithm on the sub-cell level. Task queuing is currently implemented in the Intel compiler [18] and a similar mechanism is expected to be accepted for the upcoming OpenMP 3.0 specification.

In this paper, the underlying parallelization framework Viracocha was not the focus. However, we additionally benefit from multiple shared-memory nodes by exploiting its hybrid parallelization scheme. In this paper, we also ignored loading data. Nevertheless, the influence of file system performance has to be assessed as well. A prerequisite is a complete 64-bit port of Viracocha.

7. Acknowledge

We would like to thank the German Research Foundation (DFG), who funded some of the methodical work. We are also grateful to the Institute of Aerodynamics, RWTH Aachen University, and to the German Aerospace Center (DLR), Institute of Propulsion Technology, Cologne, who kindly made available datasets for evaluation purposes.

8. References

[1] J. Helman, L. Hesselink, “Visualizing Vector Field Topology in Fluid Flows”, IEEE Computer Graphics and Applications, Vol. 11, Num. 3, pp. 36 – 46, 1991.

[2] T. Weinkauf, H. Theisel, H.-C. Hege, H.-P. Seidel, “Topological Construction and Visualization of Higher Order 3D Vector Fields”, Proceedings, Eurographics, Grenoble, France, 2004.

[3] T. Kuhlen, A. Gerndt, I. Assenmacher, B. Hentschel, M. Schirski, M. Wolter, C. Bischof, “Analysis of Flow Phenomena in Virtual Environments – Benefits, Challenges, and Solutions”, Proceedings, 11th International Conference on Human Computer Interaction, HCII 2005, Las Vegas, NV, 2005.

[4] A. Globus, C. Levit, T. Lasinki, “A Tool for Visualization the Topology of Three-Dimensional Vector Fields”, Proceedings, IEEE Visualization, San Jose, CA, pp. 33 – 40, 1991.

[5] G. Scheuermann, H. Krüger, M. Menzel, A. P. Rockwood, “Visualizing Nonlinear Vector Field Topology”, IEEE Transactions on Visualization and Computer Graphics, Vol. 4, Num. 2, pp. 109 – 116, 1998.

[6] S. Mann, A. Rockwood, “Computing Singularities of 3D Vector Fields with Geometric Algebra”, Proceedings, IEEE Visualization, Boston, MA, pp. 283 – 289, 2002.

[7] C. L. Bajaj, V. Pascucci, D. R. Schikore, “Visualization of Scalar Topology for Structural Enhancement”, Proceedings, IEEE Visualization, Research Triangle Park, NC, pp. 51 – 58, 1998.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, “Numerical Recipes in C++”, Cambridge University Press, 2002.

[9] I. A. Sadarjoen, T. van Walsum, A. J. S. Hin, F. H. Post, “Particle Tracing Algorithms for 3D Curvilinear Grids”, in: G. M. Nielson, H. Hagen, H. Müller, “Scientific Visualization - Overview, Methodologies, Techniques”, IEEE Computer Society Press, pp. 311 – 335, 1997.

[10] D. A. Lane, “Scientific Visualization of Large-Scale Unsteady Fluid Flows”, in: G. M. Nielson, H. Hagen, H. Müller, “Scientific Visualization - Overview, Methodologies, Techniques”, IEEE Computer Society Press, pp. 125 – 145, 1997.

[11] T. J. R. Hughes, “The Finite Element Method - Linear Static and Dynamic Finite Element Analysis”, Prentice-Hall International, Inc., 1987.

[12] H. Theisel, T. Weinkauf, H.-C. Hege, H.-P. Seidel, “Saddle Connectors - An Approach to Visualizing the Topological Skeleton of Complex 3D Vector Fields”, Proceedings of IEEE Visualization, Seattle, WA, pp. 225 – 232, 2003.

[13] K. Mahrous, J. Bennett, G. Scheuermann, B. Hamann, K. I. Joy, “Topological Segmentation in Three-Dimensional Vector Fields”, IEEE Transactions on Visualization and Computer Graphics, Vol. 10, Num. 2, pp. 198 – 205, 2004.

[14] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, “Surround-Screen Projection-Based Virtual Reality: The Design and the Implementation of the CAVE”, Proceedings, ACM Siggraph, Anaheim, CA, ACM Press, pp. 135 – 142, 1993.

[15] M. Schirski, A. Gerndt, T. van Reimersdahl, T. Kuhlen, P. Adomeit, O. Lang, S. Pischinger, C. Bischof, "ViSTA FlowLib – A Framework for Interactive Visualization and Exploration of Unsteady Flows in Virtual Environments", Proceedings, 7th International Immersive Projection Technologies Workshop, and 9th Eurographics Workshop on Virtual Environments, Zurich, Switzerland, ACM Siggraph, pp. 77 – 85, 2003.

[16] A. Gerndt, B. Hentschel, M. Wolter, T. Kuhlen, C. Bischof, "Viracocha: An Efficient Parallelization Framework for Large-Scale CFD Post-Processing in Virtual Environments", Proceedings, The International Conference for High Performance Computing and Communications, SC2004, Pittsburgh, PA, 2004.

[17] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, "Parallel Programming in OpenMP", Morgan Kaufmann, 1998.

[18] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, P. Petersen, "Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures", Proceedings, EWOMP, Rome, Italy, 2002.