Parallel Line Integral Convolution

Malte Zöckler, Detlev Stalling, Hans-Christian Hege

Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
{zoeckler,stalling,hege}@zib.de

Abstract. Line integral convolution (LIC) is a powerful method for computing directional textures from vector data. LIC textures can be animated, yielding the effect of flowing motion. Both, static images and animation sequences are of great significance in scientific visualization. Although an efficient algorithm for computing static LIC textures is known, the generation of animation sequences still requires a considerable amount of computing time. In this paper we propose an algorithm for computing animation sequences on a massively parallel distributed memory computer. With this technique it becomes possible to utilize animated LIC for interactive vector field visualization. To take advantage of the strong temporal coherence between different frames, parallelization is performed in image space rather than in time. Image space coherence is exploited using a flexible update and communication scheme. In addition algorithmic improvements on LIC are proposed that can be applied to parallel and sequential algorithms as well.

1 Introduction

The visual representation of vector fields is a challenging problem in scientific visualization. When analyzing a vector field one is interested in its characteristics, e.g. sources and sinks, vortices, critical points and separation lines. Images of vector fields should faithfully represent these features. A common problem with standard visualization methods is limited spatial resolution. Neither arrows or icons nor field lines or field ribbons can be placed densely without producing visual spaghetti. For two-dimensional fields texture based methods therefore have become an attractive alternative. Spot noise [7,4] and line integral convolution (LIC) [2,3,6] are prominent examples.

Textures do not only provide static images. Animating them gives an intuitive opportunity for visualizing vector fields. The resulting image sequences bear resemblance to that of a real stationary fluid flow with interspersed moving particles. However, the generation of animated texture sequences is computationally quite expensive. Therefore the method has not yet been applied for

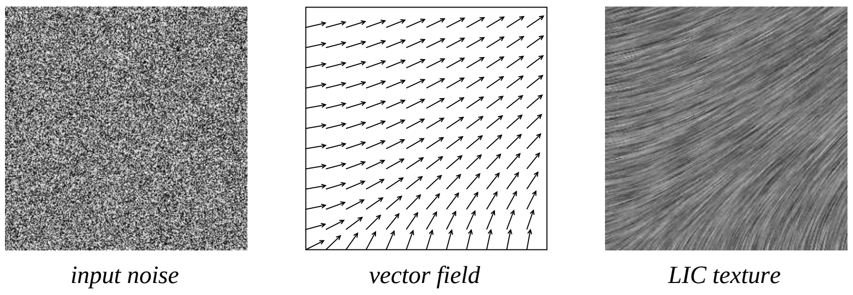

Fig. 1. In line integral convolution a random input noise is blurred along the field lines of a vector field.

interactive vector field visualization. Within an interactive visualization environment fast image synthesis is essential, even for the big data volumes that often arise in scientific computing. If data is produced on a parallel machine in a distributed simulation environment, it becomes favourable to synthesize images on the parallel computer, instead of transferring huge amounts of raw data over the network. For both reasons we have implemented a texture-based visualization method, namely line integral convolution, on a massively parallel computer (Cray T3D). Depending on the number of processors utilized, the system allows us to compute whole animations within a few seconds or less.

In a series of animated LIC textures there are strong spatial and temporal coherences. Exploiting these can speed up computation significantly. On a parallel computer this has some impact on the design of an efficient parallelization scheme. Before discussing the parallelization issues in detail, we will present in Section 2 a short summary of line integral convolution, focusing on the fast version proposed in [6]. Some new algorithmic improvements are presented in Section 3. In the next Section we describe our parallel LIC algorithm. Results of the implementation are discussed in Section 5, leading to concluding remarks in Section 6.

2 Line Integral Convolution

Given a vector field $\boldsymbol{f}:\mathbb{R}^2 \rightarrow \mathbb{R}^2$, the idea of line integral convolution is to blur a noise function T along the field lines or integral curves of $\boldsymbol{f}$. Blurring results in highly correlated pixel values along the field lines, whereas perpendicular to them almost no correlation appears. Therefore LIC images provide a clear visual impression of the directional structure of $\boldsymbol{f}$. This is illustrated in Figure 1. Mathematically the blurring is described by the convolution integral

$$I(\boldsymbol{x}) = \int\limits_{s_0-L_f}^{s_0+L_f} k(s-s_0)\, T(\boldsymbol{\sigma}(s))\, ds, \tag{1}$$

with $\boldsymbol{x} = \boldsymbol{\sigma}(s_0)$. In this equation k denotes a one-dimensional filter kernel of length $2L_f$, and $\boldsymbol{\sigma}(s)$ denotes a field line parametrized by arc-length s.

The convolution integral has to be calculated for every pixel at least once – a prohibitively expensive task if no coherences can be exploited. In case of a constant filter kernel k a fast evaluation strategy can be established by calculating only the change of the integral while stepping along a field line. This exploits the coherence between neighbouring points on a single field line. Choosing a constant filter kernel represents no limitation, as explained in Ref. [6]. The convolution integral is approximated by $2N_f+1$ equidistant discrete samples

$$I(\boldsymbol{x}_0) = \frac{1}{2N_f+1} \sum_{i=-N_f}^{N_f} T(\boldsymbol{x}_i), \quad \boldsymbol{x}_i = \boldsymbol{\sigma}(s_0 + i\,\frac{L_f}{N_f}). \tag{2}$$

Typical values of L_f comprise 10 to 40 pixel lengths. Samples should be taken with an increment of half a pixel. We are using Runge-Kutta methods with adaptive step-size control and error monitoring for field line integration. After having computed $I(\boldsymbol{x}_0)$ it is easy to update intensity along a field line via

$$I(\boldsymbol{x}_{i\pm1}) = I(\boldsymbol{x}_i) - \frac{T(\boldsymbol{x}_{i\mp N_f})}{2N_f+1} + \frac{T(\boldsymbol{x}_{i\pm(N_f+1)})}{2N_f+1}\,. \tag{3}$$

We call this method *fast LIC*. The equation is evaluated for N_s samples in both directions. A strategy for choosing the value of N_s is discussed in Section 3.3. Usually a single pixel will be covered by multiple field lines. Therefore all samples falling into that pixel have to be averaged. During the calculation for each pixel the number of hits and accumulated convolution values have to be stored. In the following we call these variables *hit count* and *accumulation buffer*. For optimal performance we would like to compute as few field lines as possible to cover the whole image. In Section 3.2 we discuss how such an optimization can be achieved.

To obtain a motion effect filter kernels are shifted along the field lines. This kind of animation allows one to encode sign and magnitude of the vector field in an unambigious and visually attractive way. Letting the filter kernel move N_a samples in both directions, we obtain time-dependent intensities

$$I(\boldsymbol{x}_0, t) = \frac{1}{2N_f+1} \sum_{i=-N_f-\lfloor tN_a\rfloor}^{N_f-\lfloor tN_a\rfloor} T(\boldsymbol{x}_i), \quad t \in [-1,1]\,. \tag{4}$$

The negative sign in the index of $\boldsymbol{x}_{-\lfloor tN_a\rfloor}$ takes into account that a backward moving filter results in an apparent flow in forward direction. Obviously this flow is non-periodic. However, by applying a simple blending technique we are able to create a periodic animation sequence with half the number of frames, namely

$$I^*(\boldsymbol{x}_0, t) = w(t-1)\, I(\boldsymbol{x}_0, t-1) + w(t)\, I(\boldsymbol{x}_0, t), \quad t \in [0,1]\,, \tag{5}$$

where $w(t)$ denotes the hat function. Due to the blending, contrast of the resulting images varies over time. This can be compensated by rescaling intensity I^* properly. Details of this scaling process are described in [6].

3 Algorithmic Improvements

In this section we will describe some improvements on the fast LIC algorithm outlined in the previous section. These improvements have special impact on the parallel implementation described later in this paper and are advantageous in the sequential case, too.

3.1 Temporal Coherence

The main point in designing a fast algorithm for line integral convolution is to take advantage of the pixel correlation along a field line. Thereby performance gains of an order of magnitude can be achieved.

In a similar way temporal coherence of pixel intensities can be exploited to speed up the generation of animation sequences. From Eq. (4) it is obvious that $I(\boldsymbol{x}_0, t) = I(\boldsymbol{x}_{-tN_a}, 0)$. This means that pixel intensity at a given location in one frame is equal to pixel intensity at some other location in some other frame. Therefore, by keeping all frames in memory, repeated computation of equal field lines and convolution sums can be efficiently avoided. Evaluating Eq. (3) for samples $\boldsymbol{x}_{-N_s-N_a}$ to $\boldsymbol{x}_{N_s+N_a}$ provides all information necessary to obtain all time dependent intensities for samples $\boldsymbol{x}_{-N_s}$ to $\boldsymbol{x}_{N_s}$.

The obvious drawback of this method is, that a large amount of memory is needed to hold a separate accumulation buffer for each frame. For example when using an accumulation buffer of 4-byte integers, a sequence of 25 video frames at 768×576 pixels resolution would need about 85 MB of main memory (recall that due to the mandatory blending 50 frames are needed to create a periodic sequence of 25 frames). This amount can be halved by performing the blending operation on the fly. After $I(\boldsymbol{x}_0, t-1)$ and $I(\boldsymbol{x}_0, t)$ have been computed, both intensities are immediately composed according to Eq. (5). The result is assigned to an accumulation buffer which corresponds to the final frame. With this technique a blending operation has to be performed each time a pixel is being hit. On the other hand, there is no need any more for an additional pass over all accumulation buffers in a post-processing step. Actually, we found that this more than compensates the costs for the additional blendings. Usually each pixel is hit not more than 5-6 times.

3.2 Seed Point Selection

The selection of seed points for field line integration has strong impact on the total number of lines that have to be computed in order to completely cover the image. If seed points for two different field lines are too close to each other, the lines are likely to run together, lowering the efficiency significantly.

This is illustrated in Figure 2. The images show pixel coverage after 120, 240 and 360 field lines have been computed. Grey levels encode the number of hits per pixel. The first algorithm chooses starting points in scanline order. The second algorithm divides the image into 4x4 blocks, taking a first pixel out of each block, then a second one, and so on. The third strategy uses Sobol quasi random sequences [1,5]. Quasi random sequences lead to point sets with a discrepancy smaller than that expected for truly random sets. A Sobol sequence is a one-to-one mapping

$$\boldsymbol{x} : i \leftrightarrow x_i \;, \qquad i, x_i \in \{1, \dots, n_x\}. \tag{6}$$

Such sequences can be generated by number-theoretic methods. In its original form the length of Sobol sequences is a power of two. Sequences of arbitrary length are simply obtained by discarding out-of-range numbers. Given two such sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ of equal length $n_x = n_y$, all pixels of a square image can be accessed in quasi-random order using

$$\begin{aligned} r_n = {} & (x_1, y_1), (x_2, y_2), \dots, (x_{n_x}, y_{n_y}), \\ & (x_1, y_2), (x_2, y_3), \dots, (x_{n_x}, y_1), \\ & \dots \\ & (x_1, y_n), (x_2, y_1), \dots, (x_{n_x}, y_{n_y - 1}) \end{aligned} \tag{7}$$

For non-square images with $n_x < n_y$ a similar numbering can be obtained by using an extended sequence $\bar{\boldsymbol{x}}$ with $\bar{x}_i = x_{i \bmod n_x}$, $i \in \{1, \dots, n_y\}$. If $n_x > n_y$ then $\boldsymbol{y}$ is extended instead of $\boldsymbol{x}$. For maximal performance both sequences may be tabulated.

Obviously the Sobol technique achieves much higher pixel coverage for a given number of field lines. The total number of samples usually is about 10-20% less compared to the block iteration method. This is discussed in more detail in the results section. However, accessing large blocks of memory in quasi-random order may cause severe cache problems on modern computer architectures. These cache insufficiencies may even completely outweigh the benefits of this method for seed point selection.

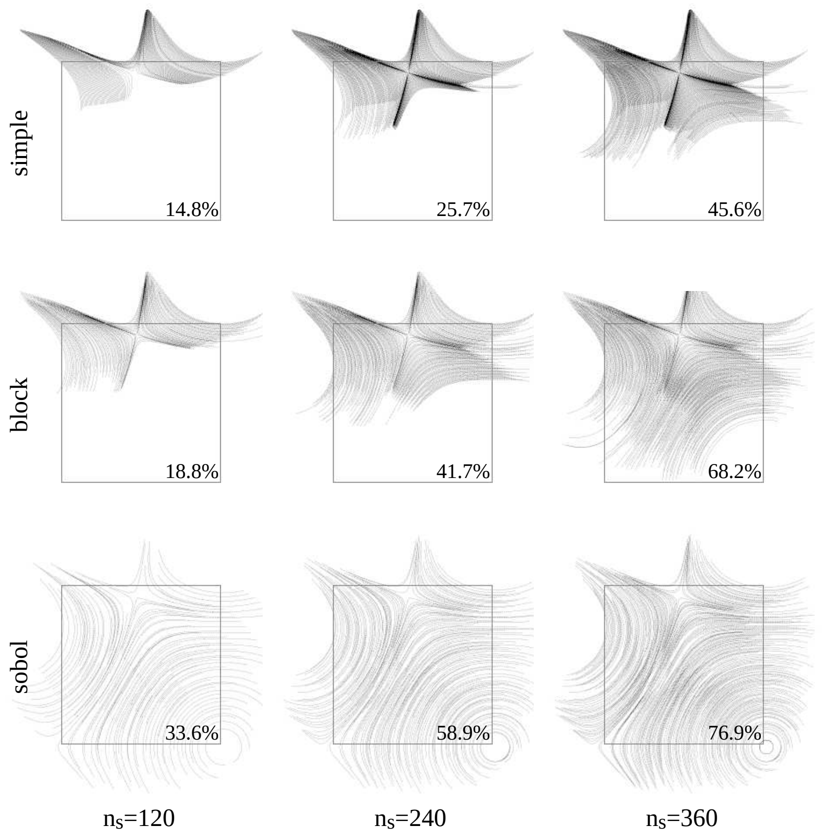

Fig. 2. Pixel coverage for three different seed point selection strategies. Grey levels encode the number of hits per pixel. The quasi-random Sobol sequence yields much higher pixel coverage for a given number of field lines.

3.3 Adaptive Field Line Length

To obtain the convolution integral for a seed point $I(\boldsymbol{x}_0)$ a field line of length $2L_f$ has to be computed, corresponding to the extent of the convolution filter kernel. To benefit from the fast update scheme described by Eq. (3), we continue the field line some distance L_s in both directions. At a first glance performance should be best when making L_s as large as possible. Unfortunately this is not true for most vector fields due to the following reasons: In fields containing sinks and sources many field lines will run toward these points, producing many hits for a small set of pixels until the integrator detects the singularity and field line integration is stopped. In fields containing vortices a field line could even turn around for infinitely long time without ever hitting new pixels. In consequence an optimal field line length exists, that depends on the vector field characteristics and the actual *coverage*. In the following we will outline a simple adaptive strategy to determine an approximation of the optimal value of L_s.

We assume that the costs for obtaining pixel intensities for all samples of a

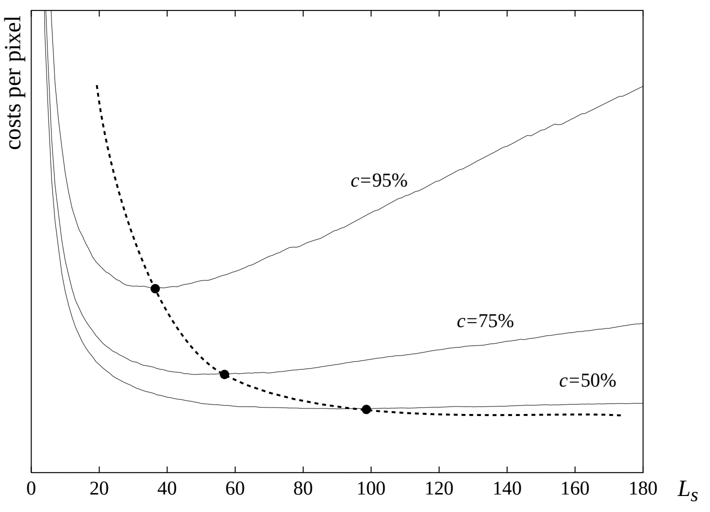

Fig. 3. Costs per pixel versus length L_s of the computed field line segments. Three curves corresponding to different degrees of coverage are shown.

field line segment of length $2L_s$ are proportional to the actual length of the field line segment which has to be computed to get these values. This length equals $2L_s$ plus an inital offset of $2L_f$ to hold the inital filter kernel. Therefore we have

$$C_{\text{field line}}(L_s) = \alpha(L_f + L_s)\ . \tag{8}$$

Small initialization costs for cache refill and for starting the integrator are ignored. Sampling a field line segment of length l will cause a certain number $P(l)$ of previously unhit pixels to be hit. As stated above $P(l)$ is *not* proportional to l, although it is of course monotonously increasing. Because we start our field lines always on pixels that are not yet touched, at least one pixel is hit, i.e. $P(0) = 1$.

The average costs per pixel for a particular field line length L_s are then

$$C_{\text{pixel}}(L_s) = \frac{\text{costs of field line}}{\text{number of new pixels}} = \frac{\alpha(L_f + L_s)}{P(L_s)}\ . \tag{9}$$

This function will in general have a minimum indicating the optimal value of L_s. To locate this minimum we have to measure $P(l)$ for a certain number of field lines (often enough to get reliable statistics). Then Eq. (9) is evaluated and the new field line size L_s is set to the optimal value. To measure $P(l)$ we simply have to keep track of the number of new hits obtained per evaluation of Eq. (3) at a particular distance.

Figure 3 shows the costs per pixel for three different degrees of coverage. The same data set as in Figure 2 has been used. As expected $C_{\text{pixel}}(L_s)$ increases as more and more pixels are being covered. At the same time the optimal field line length gets shorter. Comparing this method with a fixed size algorithm gave the following results for a 512×512 image:

L_s	N field lines	N hits	Total costs
150 (fixed)	5200	1,300,000 (100%)	100%
200 ... 10 (adaptive)	6670	901,000 (69.3%)	80.7%

Notice that the total number of field lines has increased due to a reduced average length. Nevertheless for the adaptive case the total costs estimated by

$$C_{\text{total}} \sim N_{\text{hits}} + N_f N_{\text{field lines}} \tag{10}$$

are only about 80% of the costs of the fixed length strategy due to the smaller number of hits.

4 Parallelization

There are several possibilities to parallelize line integral convolution. The aim is to find a parallelization strategy which allows one to exploit both spatial and temporal coherences.

4.1 Parallelization in Time

The simplest idea is to compute the frames of an animation sequence in parallel. No communication would be required and existing sequential code could be applied without any changes. However, the temporal coherence discussed in section 3.1 would not be exploited, resulting in a performance penalty of an order of magnitude and a corresponding waste of computing power. In addition the total number of processing elements (PEs) that can be employed would be limited by the number of frames. Periodic animation sequences of static vector fields typically do not exceed 30 frames. Therefore this method is not suitable for massively parallel architectures.

4.2 Parallelization in Image Space

An alternative approach is to distribute computation in image space. Concerning accumulation and hit count buffer there are two possibilities: use of shared global buffers which may be accessed by all PEs, or alternatively use of partitioned buffers which may be accessed by individual PEs only. Global buffers would require synchronization (and on distributed memory machines also communication) for every sample. This is obviously not a suitable approach for distributed memory machines. On shared memory systems a locking mechanism would have to guarantee that whenever a processor updates accumulation and hit count buffer, no other PE tries to modify the same memory location. However, on currently available shared memory architectures use of locking mechanisms at such a fine granularity is prohibitively slow.

To avoid the synchronization overhead caused by a shared accumulation buffer while still exploiting temporal coherence we suggest partitioning of image space. The image is subdivided into several subdomains which are assigned to different processors. Each processor is responsible for computing an animated LIC sequence in a particular subdomain. For calculating pixel values near the boundary of their subregions the PEs need to access vector field data from neighboured subregions. After replicating the static vector data every processor could in principle do its job without communication. However, if field line calculations were terminated at the boundaries of the subdomains, much of the potential benefits of the fast LIC algorithm would be lost. This is caused by the startup costs αL_f in Eq. (8). Computing fewer but longer field lines is in general much cheaper than computing many short ones. The more PEs are used, the smaller the subdomains are. With increasing number of PEs therefore the exploitation of spatial coherence decreases. In the limit of many PEs performance would converge against the original, unaccelerated LIC algorithm [2].

Hence the question arises how spatial and temporal coherence can be exploited *simultaneously.* The key observation is that the update of accumulation and hit count buffer can be relaxed since there is no need to represent the global state accurately at all times. If each PE would have an additional buffer to store the values calculated in some *overlap region* outside its subdomain, these values can *later* be combined with the results of other PEs. The resulting image remains correct. However, the computational expense may increase if more field lines than necessary are calculated due to missing information about global hit counts. The more frequently communication happens, the more exactly the global state is represented by values in the local buffers and the less redundant computations have to be performed. The limiting case corresponds to the use of a single shared accumulation buffer where spatial coherence is exploited as efficiently as in a sequential algorithm, but at the price of very high communication costs.

4.3 Pixel Coverage

We will now analyze how much work we could expect to save by updating the local hit count buffer at certain intervals. Let us denote the pixel coverage in the subdomain of a processor (i.e. the fraction of pixels hit at least once) by a number c in the range $[0,1]$. Let n be the number of hits. Of course not every hit leads to an increase in coverage because a pixel might already have been touched before. First we neglect any loss due to samples falling outside

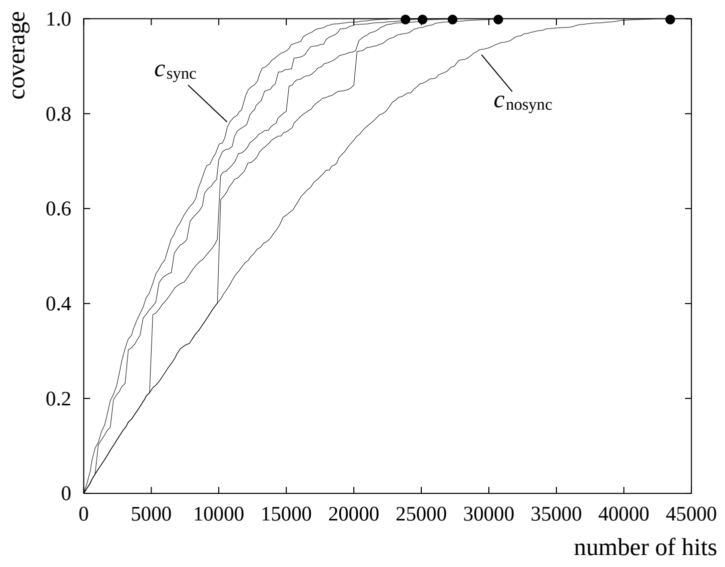

Fig. 4. Experimental coverage curves for different numbers of hit count synchronization. Data is taken every 200 hits. The steps correspond to buffer updates.

the subdomain. This corresponds to a perfectly synchronized hit count buffer, because in general lost pixels will be compensated by neighbouring nodes which will compute pixels outside *their* subdomain. Assuming that samples are distributed randomly like in a Poisson process, the increase in coverage is proportional to the uncovered area, i.e.

$$\frac{dc_{\text{sync}}}{dn} = a\,(1 - c_{\text{sync}}). \tag{11}$$

The constant factor a depends on the sampling width and the size of the subdomain. Integrating the above equation yields $c_{\text{sync}} = 1 - e^{-an}$. Obviously the model doesn't hold for large n. Otherwise coverage would never reach 100%. Now consider the case that the hit count buffer is never synchronized. Lost pixels are not compensated by neighbouring PEs any more. Therefore the factor a in Eq. (11) is somewhat smaller. Nevertheless the functional form of c_{nosync} is the same as c_{sync}. For an intermediate case where the buffers are being updated at discrete times we'll get a coverage curve somewhere between both extremes.

In Figure 4 some experimental data is shown. The synchronized curve approaches the limit of 100% coverage much faster than the unsynchronized curve. It also turns out that the limiting curve is approximated quite closely with relatively few buffer updates. The jumps in the coverage curves correspond to updates.

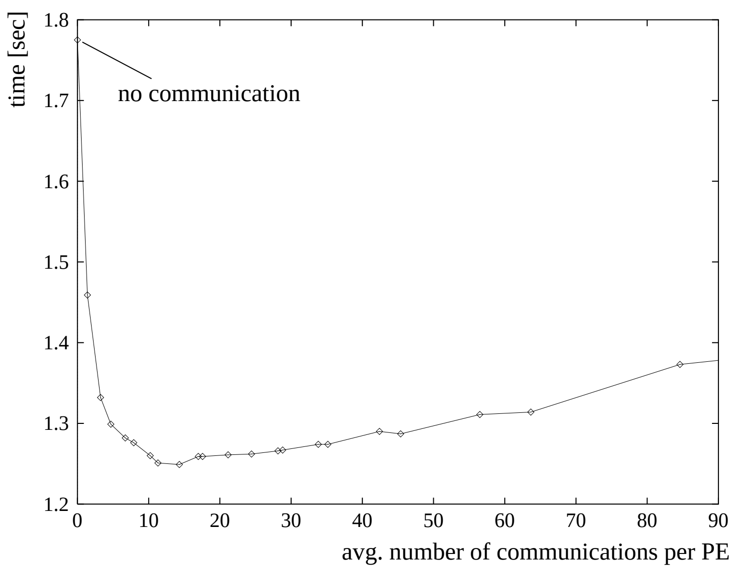

Fig. 5. Time for computing an animation sequence versus the average number of communications, i.e. non-local buffer updates. The vector field used for this plot is shown in Figure 6 (20 frames, 800 × 800 pixels each, 32 PEs of a Cray T3D).

4.4 Buffer Updates and Communication

To design an efficient update scheme we introduced a flag matrix that marks each pixel which has been hit. Although this information is also contained in the hit count buffer, the additional matrix is needed to facilitate asychronous communication. While the hit count buffer contains only local information of a subregion, the flag matrix indicates if a pixel has been hit by *any* PE. Each PE can get flag matrices from its neighbours at any time and combine them with its own one using fast logical operations. The amount of data transferred per communication step is reduced to a 1/32 compared to the transfer of a full 4 byte hit count buffer. The exact number of hits per pixel is evaluated only once at the end of the LIC algorithm.

In Section 4.3 it has been shown that an almost optimal coverage can be achieved with a relatively small number of communications. The resulting curve of computation time versus number of communications is shown in Figure 5.

The exact position of the minimum depends on the vector field characteristics as well as the number of PEs. However, the number of communications is not a very sensitive parameter, because the curve has a rather small slope to the right of its minimum. The basic parallel algorithm can be described by the following pseudo code:

```
set $nFieldlines = 0$
for each pixel $p$ do
```

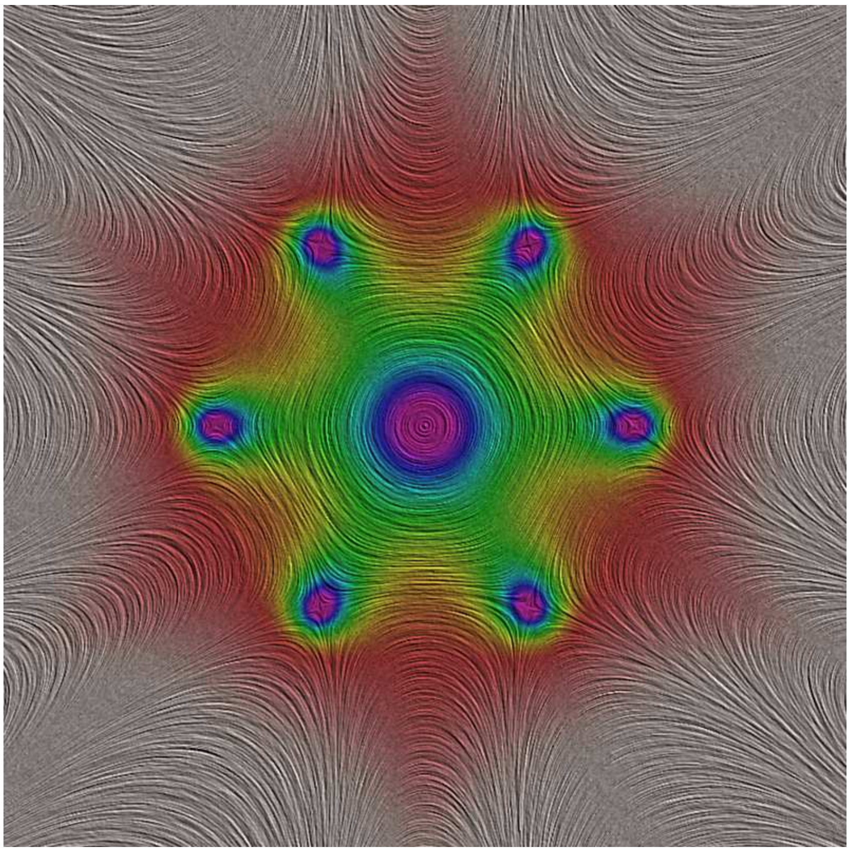

Fig. 6. Vector field visualization using line integral convolution combined with color coding and relief filtering.

```
        if (numHits($p$) == 0) then
            initiate field line computation with $\boldsymbol{x}_0$ = center of $p$
            compute convolution $I(\boldsymbol{x}_0)$
            add result to pixel $p$
            set $i = 1$
            while $i < N_S$ do
                update convolution to obtain $I(\boldsymbol{x}_i)$ and $I(\boldsymbol{x}_{-i})$
                add results to pixels containing $\boldsymbol{x}_i$ and $\boldsymbol{x}_{-i}$
                set $i = i + 1$
            end while
            set $nFieldlines = nFieldlines + 1$
            if ($nFieldlines$ mod $comInterval$ == 0) then
                for each neighbour PE $pe$ do
                    combine flag matrix from $pe$ with own flag matrix
                end for
            end if
        end if
    end for
    wait for other PEs.
```

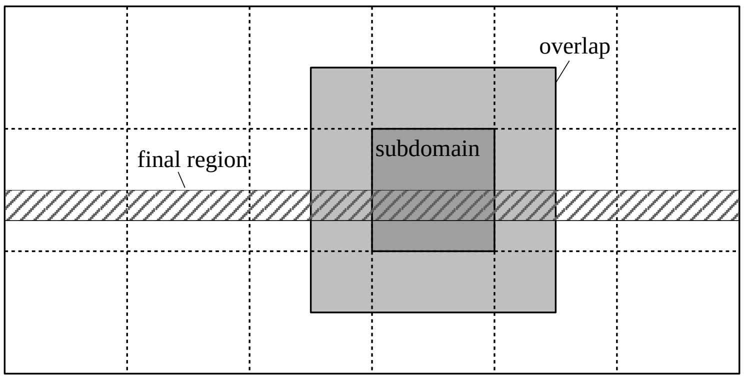

Fig. 7. Domain decomposition during LIC phase (rectangular blocks) and data collection phase (stripes).

4.5 Data Collection

After the convolution integral has been computed for each pixel some additional operations are necessary to generate the final images. First the accumulation buffers and hit counts in the overlap regions have to be combined. Then the accumulation buffers have to be divided by the hit counts and finally a contrast adjustment has to be performed.

These steps are also executed in a distributed way, although we choose a different domain decomposition: The image is divided into n equal sized horizontal stripes, where n is the number of PEs, cf. Figure 7. This allows fast output in scanline order without any further sorting steps. In other cases, e.g. where further image processing has to be performed in parallel, a different distribution might be chosen.

Each PE now retrieves those parts of other hit count and accumulation buffers that overlap with its stripe. Note that this can be done with a single shared memory get command per buffer if these are organized in scanline order. This was the reason for us to redistribute the data.

5 Results

We have compared the total work according to Eq. (10) as well as the total computing time for the Sobol and block seed point selection scheme in case of a single processor. For different computer architectures we got the following results:

	Cray T3D (1 processor)		SGI Indigo^2 R4400 150 MHz		SGI Indy R4600PC 100 MHz	
	work	computing time	work	computing time	work	computing time
block	100%	11.42 sec	100%	4.83 sec	100%	7.30 sec
sobol	88%	10.87 sec (95%)	88%	5.59 sec (115%)	88%	6.95 sec (95%)

The obvious conclusion is, that the Sobol scheme should only be used on machines without a second level cache (T3D, Indy R4600PC). Otherwise the

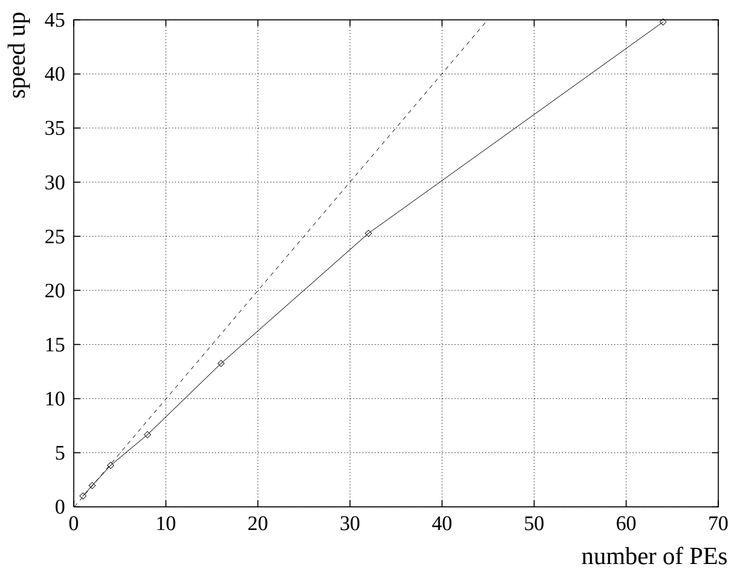

Fig. 8. Scalability of the parallel algorithm on a Cray T3D (pure computation time without I/O).

cache effects mentioned in Section 3.2 outweigh the gain in work and the program may even run slower (Indigo^2).

In Figure 8 the scalability of our parallel implementation on the Cray T3D is depicted. We achieve good speed-ups even with a large number of PEs. The deviations from the ideal linear behaviour are due to a number of reasons. On the one hand local differences in the vector field become more important as the subdomains get smaller. This increases load imbalances. On the other hand more and more hits are generated in the overlap region of each PE, which inevitably increases the amount of redundant computations. In addition start-up costs become more important.

While on a workstation (SGI Indigo^2 150 Mhz) it takes about 20 seconds to compute an animation sequence consisting of 20 frames 512×512 pixels each, we are able to create such a sequence in less than a second on a 64 PE partition of a Cray T3D. Such rates offer new opportunities for real time visualization of vector fields. Among others this is an interesting feature for controlling and steering large numerical simulations.

Figure 9 shows snapshots taken from an application we have developed to allow interactive exploration of velocity fields obtained from a CFD simulation. While the animated field is displayed on a local workstation, the user may enlarge interesting regions to examine details of the vector field. As the new image sequences are computed on the T3D, the user gets nearly immediate response. It is also possible to interact with the simulation (like changing boundary conditions) and to view the results without a significant latency.

6 Conclusion

We have presented algorithmic improvements to sequential fast LIC algorithms for the synthesis of static images and image sequences. An efficient way to parallelize these algorithms, retaining all benefits of the sequential case, has been proposed. Our implementation on a Cray T3D achieved a good speed-up even for 64 and more PEs.

The parallel implementation allows to use animated LIC for real time visualization. This offers a whole bunch of interesting opportunities in visualization and computer graphics.

Acknowledgement

This work was supported in part by BMBF/DFN (Berlin Regional Testbed for Broadband Networks). We thank Roland Wunderling for many helpful discussions about parallelization strategies as well as Johannes Schmidt-Ehrenberg for supporting us with the parallel implementation on the Cray T3D.

References

[1] P. Bratley, B.L. Fox *Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, vol. 14, pp. 88-100.

[2] B. Cabral, L. Leedom, *Imaging vector fields using line integral convolution*, Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics* 27, 1993, ACM SIGGRAPH, pp. 263-272.

[3] L.K. Forsell, *Visualizing Flow over Curvilinear Grid Surfaces unsing Line Integral Convolution*, Proceedings of Visualization '94, Bergeron and Kaufman, Eds., IEEE Computer Society Press, 1994, pp. 240-247.

[4] W.C. de Leeuw, J.J. van Wijk, *Enhanced Spot Noise for Vector Field Visualization*, Proceedings of Visualization '95, Nielson and Silver, Eds., IEEE Computer Society Press, 1995, pp. 233-239.

[5] I.M. Sobol, *On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals*, USSR Computational Mathematics and Mathematical Physics, vol. 7, no. 4, pp. 86-112.

[6] D. Stalling, H.C. Hege, *Fast and Resolution Independent Line Integral Convolution*, Proceedings of SIGGRAPH '95 (Los Angeles, California, August 6-11, 1995). In *Computer Graphics* Annual Conference Series, 1995, ACM SIGGRAPH, pp. 249-256.

[7] J.J. van Wijk, *Spot noise-texture synthesis for data visualization*, Proceedings of SIGGRAPH '91 (Las Vegas, Nevada, 28 July - 2 August, 1991). In *Computer Graphics*, 25, pp. 309-318.

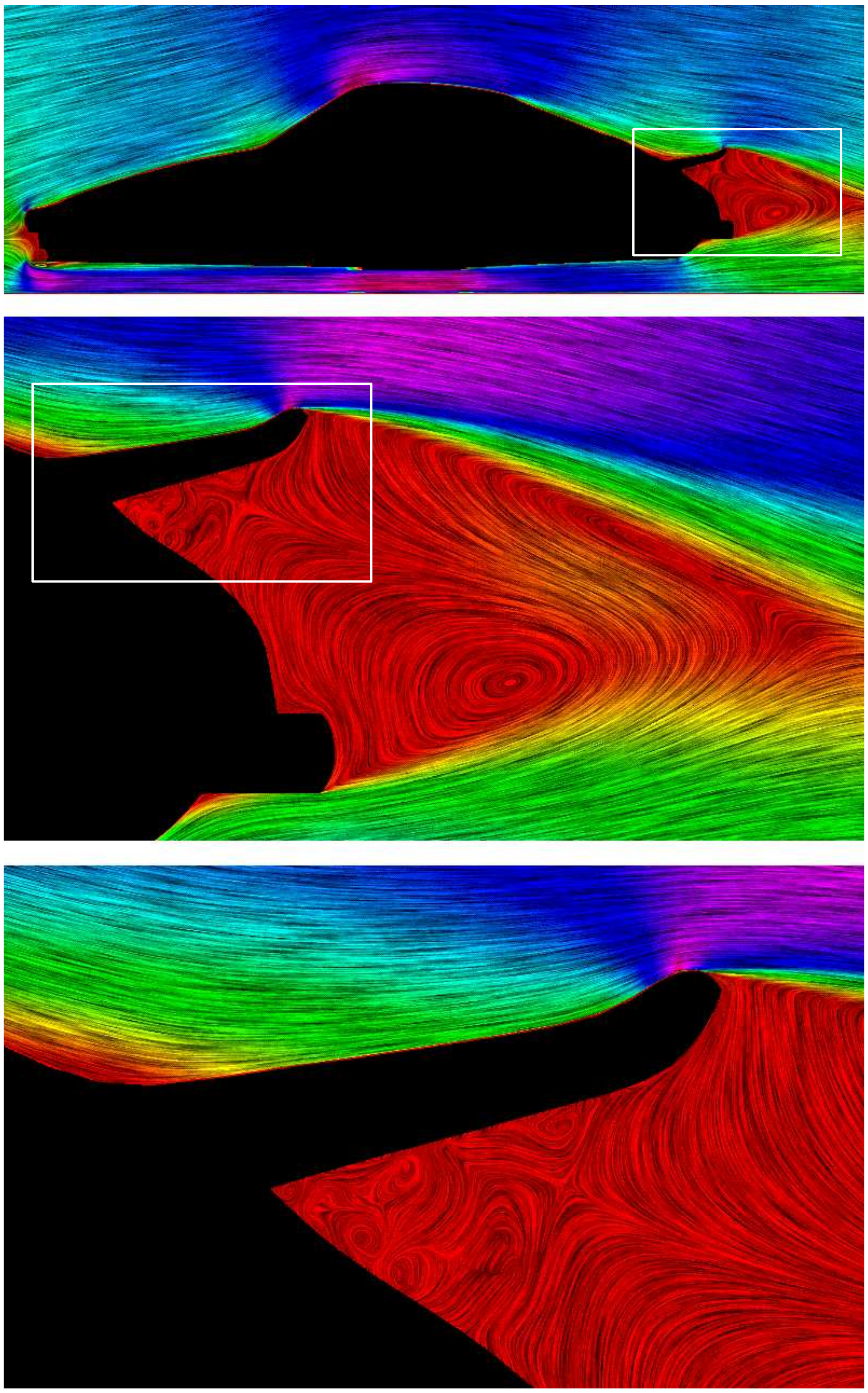
Fig. 9. Snapshots from an interactive visualization application.