Chapter 6

Parallel Integral Curves

David Pugmire
Oak Ridge National Laboratory

Tom Peterka
Argonne National Laboratory

Chistoph Garth
University of Kaiserslautern

Understanding vector fields resulting from large scientific simulations is an important and often difficult task. Integral curves, curves that are tangential to a vector field at each point, are a powerful visualization method in this context. Application of integral curve-based visualization to very large vector field data represents a significant challenge due to the non-local and data-dependent nature of integral curve computation, and requires careful balancing of computational demands placed on I/O, memory, communication, and processors. In this chapter we review several different parallelization approaches based on established parallelization paradigms (across particles, and data blocks) and present advanced techniques for achieving scalable, parallel performance on very large data sets.

6.1 Introduction

The visualization of vector fields is a challenging area of scientific visualization. For example, the analysis of fluid flow that governs natural phenomena on all scales from the smallest (e.g. Rayleigh-Taylor mixing of fluids) to the largest (e.g. supernovae explosions) relies on visualization to elucidate the patterns exhibited by flows and the dynamical structures driving them. Processes typically depicted by vector fields – such as transport, circulation and mixing – are prevalently non-local in nature. Due to this specific property, methods and techniques developed and proven successful for scalar data visualization are not readily generalized to the study of vector fields. Hence, while it is

technically feasible to apply such methods to directly derived scalar quantities such as vector magnitude, the resulting visualizations often fall short in explaining the mechanisms underlying the scientific problem.

A large majority of visualization approaches devised to visualize vector fields are based on the study of their integral curves. Naturally understood as trajectories of massless particles, such curves are ideal tools to study transport, mixing and other similar processes. These *integration-based methods* make use of the intuitive interpretation of integral curves as the trajectories of massless particles, and were originally developed to reproduce physical flow visualization experiments based on small, neutrally buoyant particles. Mathematically, an integral curve is tangential to the vector field at every point along curve, and individual curves are determined by selecting an initial condition or *seed point*, the location from which the curve begins, and an *integration time* over which the virtual particle is traced.

Integral curves are applicable in many different settings. For example, direct visualization of particles, and particle families and their trajectories gives rise to such visualization primitives as streamlines, pathlines, streak lines, or integral surfaces (e.g. [**?**]). To specifically study transport and mixing, integral curves also offer an interesting change of perspective. Instead of considering the so-called *Eulerian* perspective that describes evolution of quantities at fixed locations in space, the *Lagrangian* view examines the evolution from the point-of-view of an observer attached to a particle moving with the vector field, offering a natural and intuitive description of transport and mixing processes. Consider for example the case of combustion: fuel burns while it is advected by surrounding flow; hence the burning process and its governing equations are primarily Lagrangian in nature.

Further, Lagrangian methods concentrate on deriving structural analysis from the Lagrangian perspective. For example, *Finite-Time Lyapunov Exponents* (FTLE) empirically determine exponential separation rates among neighboring particles from a dense set of integral curves covering a domain of interest (e.g. [**?**]). From *ridges*, i.e. locally maximal lines and surfaces, in FTLE, on can identify so-called *Lagrangian Coherent Structures* that approximate hyperbolic material lines and represent the dynamic skeleton of a vector field that drives its structure and evolution. For an overview of modern flow visualization techniques, we refer the interested reader to [**?**].

Computationally, integral curves are approximated using numerical integration schemes (cf. [**?**]). These schemes construct a curve in successive pieces: starting from the seed point, the vector field is sampled in the vicinity of the current integration point, and an additional curve sequence is determined and appended to the existing curve. This is repeated until the curve has reached its desired length or leaves the vector field domain. To propagate an integral curve through a region of a given vector requires access to the source data. If the source data can remain in main memory, it will be much faster than a situation where the source data must be paged into main memory from a secondary location. In this setting, the non-local nature of particle

advection implies that the path taken by a particle largely determines which blocks of data must be loaded. This information is a priori unknown and depends on the vector field itself. Thus, general parallelization of integral curve computation is a difficult problem for large data sets on distributed memory architectures. Furthermore, to compute very large sets of integral curves, as mandated by modern visualization and analysis techniques, further requires distributed computation.

6.2 Challenges to Parallelization

In this chapter, we are aiming to both qualify and quantify the performance of three different parallelization strategies for integral curve computation. Before we provide a discussion of particular parallelization algorithms in Section **??**, we will first present the challenges particular to the parallelization of integral curve computation on distributed memory systems.

6.2.1 Problem Classification

The parallel integral curve problem is complex and challenging. To design an experimental methodology that provides robust coverage of different aspects of algorithmic behavior, some of which is dataset dependent, we must take into account the following factors that influence parallelization strategy and performance test design.

Data Set Size. If the dataset is *small*, in the sense that if fits entirely into the memory footprint of each task, then it makes most sense to distribute the integral curve computation workload. On the other, if the dataset is larger than will fit entirely in each tasks memory, then more complex approaches are required that will involve data distribution and possibly computation distribution.

Seed Set Size. If the problem at hand requires only the computation of a thousand streamlines, parallel computation takes secondary precedence to optimal data I/O and distribution; we refer to the corresponding seed set as *small*, such small seed sets are typically most often encountered in interactive exploration scenarios where relatively few integral curve's are interactively seeded by a user. A *large* seed set may consist of many thousands of seed points. Here, it will be desireable to distribute the integral curve computational workload, yet data distribution schemes need to allow for efficient parallel integral curve computation.

Seed Set Distribution. In the case where seed points are located *densely* within the spatial and temporal domain of definition of a vector field, it is likely that they will traverse a relatively small amount of the overall data. On the other hand, for some applications like streamline statistics, a *sparse*

seed point set covers the entire vector field evenly. This distribution results in integral curve's traversing the entire data set. Hence, the seed set distribution is a significant factor in determining if performance stands to gain most from parallel computation, data distribution, or both.

Vector Field Complexity. Depending on the choice of seed points, the structure of a vector field can have a strong influence on which parts of the data need to be taken into account in the integral curve computation process. Critical points or invariant manifolds of strongly attracting nature draw streamlines towards them, and the resulting integral curve's seeded in or traversing their vicinity remain closely localized. On the other hand, the opposite case of a nearly uniform vector field requires integral curve's to pass through large parts of the data.

6.3 Approaches to Parallelization

Particle advection places demands on almost all components of a computational system, including memory, processing, communication, and I/O. Because of the complex nature of vector fields, seeding scenarios, and the types of analyses, as outlined in Section **??**, there is no single scalable algorithm suitable for all situations.

Broadly speaking, algorithms can be parallelized in two primary ways. Parallelization across the seed points, where seeds are assigned to processors, and data blocks are loaded as needed, and parallelization across the data blocks, where processors are assigned a set of data blocks, and particles are communicated to the processor that owns the data block. In choosing between these two axes of parallelization, the differing assignments of particles and data blocks to processors will place differing demands on the computing, memory, communication, and I/O sub-systems of a cluster.

These two classes of algorithms have a tendency of performing poorly due to workload imbalance, either through an imbalance in particle to processor assignment, or through loading too many data blocks, and becoming I/O bound. Recent research has shown that hybrid parallelization approaches yield the most promising results.

In the sections that follow, we outline algorithms that parallelize with respect to particles, and data blocks, and discuss their performance characteristics. Next, we outline several different strategies for achieving scalable performance using hybrid parallelism. These strategies are aimed at keeping a balanced work load across the set of processors, the design of efficient data structures for handling integral curve computation, and efficiency in communication.

In the approaches outlined below, the problem mesh is decomposed into a number of spatially disjoint data blocks. Each block may or may not have

ghost cells for connectivity purposes. Each block has a time step associated with it, thus two blocks that occupy the same space at different times are considered independent.

6.3.1 Parallelization over Seed Points

The first of the traditional algorithms is one that parallelizes over the axis of particles. In this algorithm, the seed points are uniformly assigned across the set of processors. Thus, given n processors, $1/n$ of the seed points are assigned to each processor. To enhance spatial locality, the seed points are sorted spatially before being assigned to processors. As outlined in Algorithm **??**, each processor integrates the points assigned to it until termination, or until the point exists the spatial data block. As points move between blocks, each processor loads the appropriate block into memory. In order to minimize I/O, each processor integrates all particles to the edge of the loaded blocks, loading a block from disk only when there is no more work to be done on the in-memory blocks. Each particle is only ever owned by one processor, though blocks may be loaded by multiple processors. Each processor terminates independently when all of its streamlines have terminated.

In order to manage the handling of blocks, this algorithm makes use of caching of blocks in a LRU fashion. If there is insufficient memory for additional blocks when a new block must be loaded, the block that is purged is that which was least recently used. Clearly, having more main memory available decreases the need for I/O operations, and increases the performance of this algorithm.

Another method for handling of data blocks was introduced in [**?**] where an extended hierarchy of memory layers is used for data block management. Using local a local disk, either SSD or hard drive, careful management of data block movement was shown to increase overall performance by about 2X.

Algorithm 1 Parallelization over seed points algorithm

```
while not done do
    activeParticles = particles that reside in a loaded data block
    inactiveParticles = particles that do not reside in a loaded data block
    while activeParicles not empty do
        advect particle
    end while
    if inactiveParticles then
        Load a dataset from inactiveParticles
    else
        done = true
    end if
end while
```

Advantages:

This algorithm is straightforward to implement, requires no communication, and is ideal if I/O requirements are known to be minimal. For example, if the entire mesh will fit into main memory of a single process, or if the flow is known to be spatially local. In such cases, parallelization is trivial, and parallel scaling would be expected to be nearly ideal. Further, the risk of processor imbalance is minimized because of the uniform assignment of seeds to processors.

Disadvantages: Because data blocks are loaded on demand, it is possible for I/O to dominate this algorithm. For example, in a vector field with circular flow, and a data block cache not large enough enough blocks, data will be continuously loaded, purged in the LRU cache. Additionally, work load imbalance is possible if the computational requirements for particles differ significantly. For example, if some particles are advected significantly farther than others, the processors assigned to these particles will be doing all the work, while the others sit idle.

6.3.2 Parallelization over Data

The second of the traditional algorithms is one that parallelizes over the axis of data blocks. In this algorithm, the data blocks are statically assigned to processors. Thus, given n processors, $1/n$ of the data blocks are assigned to each processor. Each particle is integrated until it terminates, or leaves the blocks owned by the processor. As a particle crosses block boundaries, it is communicated to the processor that owns the data block. A globally maintained active particle count is maintained so that all processors may monitor how many particles have yet to terminate. Once the count goes to zero, all processors terminate.

Advantages: This algorithm performs the minimal amount of I/O, each data block is loaded once, and only once into memory. As I/O operations are orders of magnitude more expensive that computation, this is a significant advantage. In situations where the vector field is known to be uniform, and the seed points are known to be sparse, this algorithm is well suited.

Disadvantages: Because of the spatial parallelization, this algorithm is very sensitive to seeding, and vector field complexity. If the seeding is dense, only the processors where the seeds lie spatially will be doing any work. All other processors will be idle. Further in cases where the vector field contains critical points, or invariant manifolds, points will be drawn to these spatial regions, increasing the work load of some processors, while decreasing the work of the rest.

Comparisons: To compare the performance of these two algorithms, we use

Algorithm 2 Parallelization over data

```
activeParticleCount = N
while activeParticleCount > 0 do
    activeParticles = particles that reside in a loaded data block
    inactiveParticles = particles that do not reside in a loaded data block
    while activeParicles not empty do
        advect particle
        if particle terminates then
            activeParticleCount = activeParticleCount - 1
            Communicate activeParticleCount update to all
        end if
        if particle moves to another data block then
            Send particle to process that owns data block
        end if
    end while
    Receive incoming particles
    Receive incoming particle count updates
end while
```

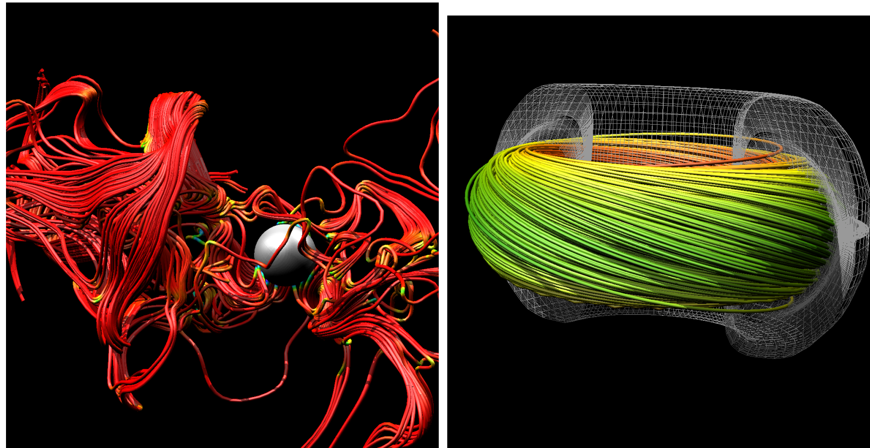

FIGURE 6.1: Particles advected through the magnetic field of a supernova simulation (A). Data courtesy of GenASiS. Particles advected through the magnetic field of a fusion simulation (B). Data courtesy of NIMROD.

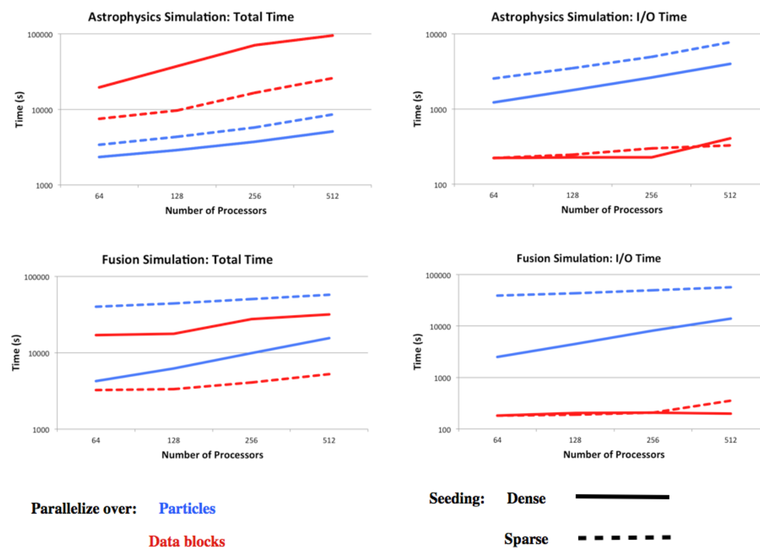

FIGURE 6.2: Log scale plots of total time to advect 20,000 particles through the astrophysics, and fusion datasets with both sparse and dense seeding scenarios.

20,000 particles, and use both sparse and dense seeding configurations on an astrophysics, and fusion simulations, shown in Figure **??**. The total times, and time spent doing IO operations for both simulations are shown in Figure **??**. Note that in the astrophysics simulation, parallelization across seeds exhibits the superior performance, despite the fact that significantly more I/O is performed. In this case, the additional I/O operations resulted in great processor utilization, and a higher overall efficiency. In the fusion simulation, where the flow circulates around the center of the simulation domain, the superior algorithm is not as clear. If the seeding is sparse, then parallelization over data blocks is superior, however, if the seeding is dense, then parallelization over seeds is most efficient.

6.3.3 Workload Monitoring and Balancing

As outlined previously, parallelization over seeds or data blocks are subject to workload imbalance. Because of the complex nature of vector field analysis, it is often very difficult or impossible to know a-priori which strategy would be most efficient. In general, where smooth flow, and complex flow can exist within the same data set, it is impossible to get scalable parallel performance using the traditional algorithms.

Research has shown that the most effective solution is a hybrid approach, where parallelization is performed over both seed points, and data blocks. In hybrid approaches, the resource requirements are monitoried, and reallocation of resources can be done dynamically based on the nature of a particular vector field, and seeding seeding scenario. By re-allocating as needed, resources can be dynamically used in the most efficient manner possible.

The algorithm introduced in [**?**] is a hybrid between parallelization across particles and data blocks discussed previously. In this hybrid algorithm, we dynamically parition the workload of both particles and data blocks to processors in an attempt to load balance on the fly based upon the processor workloads and the nature of the vector field. It attempts to keep all processors busy while also minimizing I/O by choosing either to communicate particles to other processors or to have processors load duplicate data blocks based on heuristics.

Since detailed knowledge of flow is often unpredictable or unknown, this algorithm was designed to adapt during the computation to concentrate resources where they are needed, distributing particles where needed, and/or duplicating blocks when needed. Through workload monitoring and balancing, we achieve parallelization across data blocks and particles simultaneously, and are able to adapt to the challenges posed by the varying characteristics of integration-based problems.

In this hybrid algorithm, the processors are divided up into a set of work groups. Each work group consists of a single master process, and a group of slave processes. Each master is responsible for monitoring the work load among the group of slaves, and making work assignments that optimize re-

source utilization. The master makes initial assignments to the slaves based on the initial seed point placement. As work progresses, the master monitors the length of each slave's work queue and the blocks that are loaded and reassigns particle computation to balance both slave overload and slave starvation. When the master determines that all streamlines have terminated, it instructs all slaves to terminate.

For scalable performance, work groups will coordinate between themselves, and work can be moved from one work group to another, as needed. This allows for work groups to focus resources on different portions of the problem as dynamically needed.

The design of the slave process is outlined in Algorithm **??**. Each slave continuously advances particles that reside in data blocks that are loaded. Similarly to the parallelize over seed points algorithm, the data blocks are held in a LRU cache to the extent permitted by memory. When the slave can advance no more particles or is out of work, it sends a status message to the master and waits for further instruction. In order to hide latency, the slave sends the status message right before it advances the last particle.

Algorithm 3 Slave process algorithm

```
while not done do
    if new command from master then
        process command
    end if
    while active particles do
        process particles
    end while
    if state changed then
        send state to master
    end if
end while
```

The master process is significantly more complex, and outlined at a high level in Algorithm **??**. The master process is responsible for maintaining information on each slave and managing their workloads. At regular intervals, each slave sends a status message to the master so that it may keep accurate global information. This status message includes the set of particles owned by each slave, which data blocks those particles currently intersect, which data blocks are currently loaded into memory on that slave, and how many particles are currently being processed. Load balancing is achieved by observing the work load of the group of slaves, and deciding when particles should be re-assigned to slaves that already have the required data blocks loaded, or whether the slave should load the data block, and then process the particles. All communication is performed using non-blocking send and receive commands.

The master process uses a set of four rules to manage the work in each group.

Algorithm 4 Master process algorithm

```
get initial particles
while not done do
    if New status from any slave then
        command ← most efficient next action
        Send command to appropriate slave(s)
    end if
    if New status from another master then
        command ← most efficient next action
        Send command to appropriate slave(s)
    end if
    if Work group status changed then
        Send status to other master processes
    end if
    if All work groups done then
        done = true
    end if
end while
```

1. **Assign loaded block:** The master sends a particle to a slave that has the required data block already loaded into memory. The slave will add this particle to the active particle list, and process the particle.

2. **Assign unloaded block:** The master sends a particle to a slave that does not have the required data block loaded into memory. The slave will load the data block into memory, add the particle to its active particle list, and process the particle. The master process uses this rule to assign initial work to slaves when particle advection begins.

3. **Load block:** The master sends a message to a slave instructing it to load a data block into memory. When the slave receives this message, the data block is loaded, and all particles requiring this data block are moved into the active particle list.

4. **Send particle:** The master sends a message to a slave instructing it to send particles to another slave. When the slave receives this command, the particle is sent to the receiving slave. The receiving slave will then place this particle in the active particle list.

The master in each work group monitors the load of the set of slaves. Using a set of heuristics, detailed in [**?**], the master will use the four commands above to dynamically balance work across the set of slaves. The commands allow particle, and data block assignments for each processor to be changed over the course of the integration calculation.

To compare the performance of these three algorithms, we use 20,000 particles, and use both sparse and dense seeding. Results of this hybrid algorithm,

FIGURE 6.3: Log scale plots of total time to advect 20,000 particles through the astrophysics, and fusion datasets with both sparse and dense seeding scenarios.

and the algorithms parallelizing over particles and data blocks using both sparse and dense seeding scenarios are shown for the astrophysics and fusion simulations in Figure **??**. Note that in the astrophysics simulation, the hybrid algorithm performs at a near optimal level, while performing significantly less I/O. In the fusion simulation, where parallelization across data blocks is clearly ideal, the hybrid algorithm performs well for sparse seeding, and out performs dense seeding, while at the same time, performing significantly less I/O than parallelization across seeds.

6.3.4 Hybrid Data Structure and Communication Algorithm

Large seed sets and large time-varying vector fields pose additional challenges for parallel integral curve algorithms. The first is the decomposition of 4D time-varying flow data in space and time, and we will discuss a hybrid data structure that combines 3D and 4D blocks for computing time-varying integral curves efficiently. The second challenge arises in both steady and transient flows but is magnified in the latter case: this is the problem of performing nearest-neighbor communication iteratively while minimizing synchronization points during the execution. We will present a communication algorithm for the exchange of information among neighboring blocks that allows tuning the amount of synchronization desired. Together, these techniques have been used to compute 1/4 million steady-state and time-varying integral curves on vector fields over 1/2 terabyte in size [**?**].

Peterka et al. [**?**] introduced a novel 3D/4D hybrid data structure which is shown in Figure **??**. It is called hybrid because it allows 4D space-time to be viewed as both a unified 4D structure and a 3D space $\times$ 1D time structure. The top row of Figure **??** illustrates this idea for individual blocks, and the bottom row shows the same idea for neighborhoods of blocks.

The same data structure is used for both steady-state and time-varying flows by considering steady-state as a single time-step of the time-varying general case. A particle is a 4D (x, y, z, t) entity. In the left column of Figure **??**, a block is also 4D, with minimum and maximum extents in all four dimensions. Each 4D block sits in the center of a 4D neighborhood, surrounded on all sides and corners by other 4D blocks. A single neighborhood consists of $3^4 = 81$ blocks: the central block and adjacent blocks in both directions in the x, y, z, and t dimensions.

If the data structure were strictly 4D and not hybrid, the center and right columns of Figure **??** would be unnecessary; but then all time-steps of a time-varying flow field would be needed at the same time in order to compute an integral curve. Modern CFD simulations can produce hundreds or even thousands of time-steps, and the requirement to simultaneously load the entire 4D dataset into main memory would exceed the memory capacity of even the largest HPC systems. At any given time in the particle advection, however, it is only necessary to load data blocks that contain vector fields at the current time and perhaps one time-step in the future. On the other hand, loading multiple

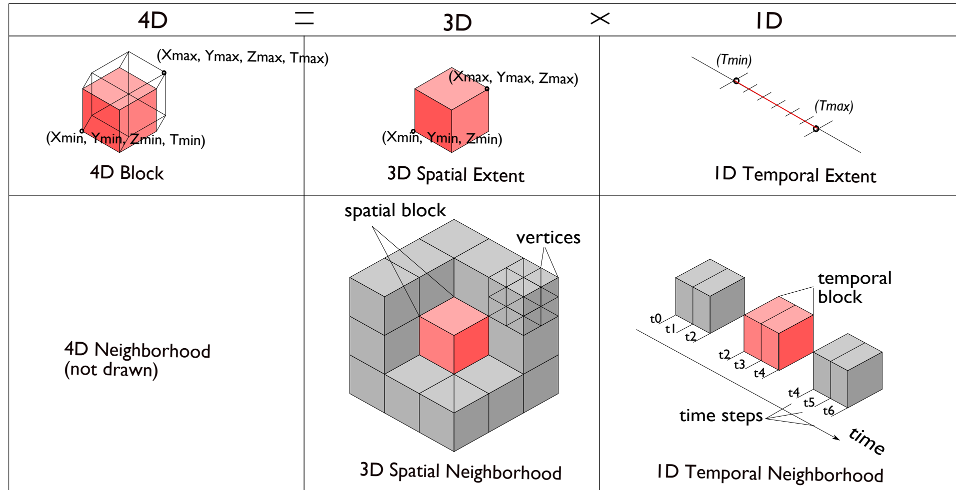

FIGURE 6.4: The data structure is a hybrid of 3D and 4D time-space decomposition. Blocks and neighborhoods of blocks are actually 4D objects. At times, such as when iterating over a sliding time window (time block) consisting of one or more time steps, it is more convenient to separate 4D into 3D space and 1D time.

time-steps simultaneously often results in a more efficient data access pattern from storage systems and reduced I/O time. Thus, the ability to step through time in configurable-sized chunks, or time windows, results in a flexible and robust algorithm that can run on various architectures with different memory capacities.

Even though blocks and neighborhoods are internally unified in space-time, the hybrid data structure enables the user to decompose space-time into the product of space × time. The lower right corner of Figure **??** shows this decomposition for a temporal block consisting of a number of individual time-steps. Varying the size of a temporal block–the number of time-steps contained in it–creates an adjustable sliding time window that is also a convenient way to trade the amount of in-core parallelism with out-of-core serialization. Algorithm **??** shows this idea in pseudocode. A decomposition of the domain into 4D blocks is computed given user-provided parameters specifying the number of blocks in each dimension. The outermost loop then iterates over 1D temporal blocks, while the work in the inner loop is done in the context of 4D spatio-temporal blocks. The hybrid data structure just described enables this type of combined in-core / out-of-core execution, and the ability to configure block sizes in all four dimensions is how Algorithm **??** can be flexibly tuned to various problem and machine sizes.

All parallel integral curve algorithms for large data and seed sets have one thing in common: the inevitability of interprocess communication. Whether

Algorithm 5 Main Loop

decompose entire domain into 4D blocks
for all *1D temporal* blocks assigned to my process **do**
  read corresponding *4D spatio-temporal* data blocks into memory
  for all *4D spatio-temporal* blocks assigned to my process **do**
    advect particles
  end for
end for

exchanging data blocks or particles, nearest-neighbor communication is unavoidable and limits performance and scalability. Hence, developing an efficient nearest-neighbor communication algorithm is crucial. The difficulty stems from the fact that neighborhoods, or groups in which communication occurs, overlap. In other words, blocks are members of more than one neighborhood: a block at the edge of one neighborhood is also at the center of another, and so forth. Hence, a delay in one neighborhood will propagate to another, and so on, until it affects the entire system. Reducing the amount of synchronous communication can absorb some of these delays and improve overall performance, and one way to relax such timing requirements is to use nonblocking message passing.

In message-passing systems like MPI, nonblocking communication can be confusing. Users often have the misconception that communication automatically happens in the background, but in fact, messages are not guaranteed to make progress without periodic testing of their status. After nonblocking communication is initiated, control flow is returned to the caller and *hopefully* some communication occurs while the caller executes other code, but to ensure this, the caller must periodically check on the communication and perhaps wait for it to complete.

The question then becomes how frequently to check back on nonblocking messages, and how long to wait during each return visit. An efficient commu-

Algorithm 6 Asynchronous communication algorithm

for all processes in my neighborhood **do**
  pack message of block IDs and particle counts
  post nonblocking send
  pack message of particles
  post nonblocking send
  post nonblocking receive for IDs and counts
end for
wait for enough IDs and counts to arrive
for all IDs and counts that arrived **do**
  post blocking receive for particles
end for

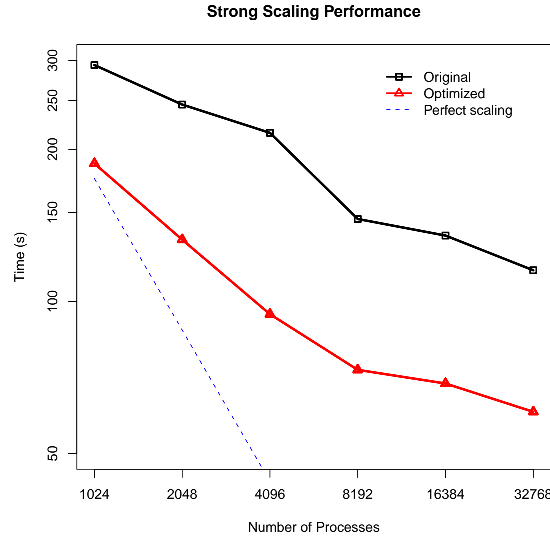

FIGURE 6.5: The cumulative effect on end-to-end execution time of a hybrid data structure and adjustable communication algorithm is shown for a benchmark test of tracing 256 K particles in a vector field that is $2048 \times 2048 \times 2048$.

nication algorithm that answers these questions was introduced by Peterka et al.[**?**] and is outlined in Algorithm **??**. It works on the assumption that nearest-neighbor communication occurs in the context of alternating rounds of advection and communication of particles. In each round, new communication is posted, and previously posted messages are checked for progress. Each communique consists of two messages: a header message containing block identification and particle counts, and a payload message containing the actual particle positions.

The algorithm takes an input parameter that controls the fraction of previously posted messages for which to wait in each round. In this way, the desired amount of synchrony / asynchrony can be adjusted and allows the "dialing down" of synchronization to the minimum needed to make progress. In practice, we have found that waiting for only 10% of pending messages to arrive in each round is the best setting. This way, each iteration makes a guaranteed minimum amount of communication progress without imposing unnecessary synchronization. Reducing communication synchronization accelerates the overall particle advection performance and is an important technique for communicating across large-scale machines where global code synchronization becomes more costly as the number of processes increases.

The collective effect of these improvements is a $2\times$ speedup in overall execution time compared to earlier algorithms. This improvement is demon-

TABLE 6.1: Performance Benchmarks

Thermal Hydraulics	Hydrodynamics	Combustion
Strong scaling, 2048 x 2048 x 2048 x 1 time-step = 98 GB data, 128K particles	Weak scaling, 2304 x 4096 x 4096 x 1 time-step = 432 GB data, 16K to 128K particles	Weak scaling, 1408 x 1080 x 1100 x 32 time-steps = 608 GB data, 512 to 16K particles

strated in Figure **??** with a benchmark test of tracing 1/4 million particles in a steady state thermal hydraulics flow field that is 2048^3 in size. Peterka et al.[**?**] benchmarked this latest fully-optimized algorithm using scientific datasets from problems in thermal hydraulics, hydrodynamics (Rayleigh-Taylor instability), and combustion in a cross-flow. Table **??** shows these results on both strong and weak scaling tests of steady and unsteady flow fields. The left-hand column of Table **??** shows good strong scaling (where the problem size remains constant) for the steady-state case, seen by the steep downward slope of the scaling curve. The center column of Table **??** also shows good weak scaling (where the problem size increases with process count), with a flat overall shape of the curve. The right-hand column of Table **??** shows a weak scaling curve that slopes upward for the time-varying case, and it demonstrates that even after optimizing the communication, the I/O time to read each time block from storage remains a bottleneck.

Advances in hybrid data structures and efficient communication algorithms can enable scientists to trace particles of time-varying flows during their simulations of CFD vector fields at concurrency that is comparable with their parallel computations. Such highly parallel algorithms are particularly im-

portant for peta- and exascale computing, where more analyses will need to execute at simulation run-time in order to avoid data movement and avail full-resolution data at every time step. Access to high-frequency data that are only available in situ is necessary for accurate integral curve computation; typically simulation checkpoints are not saved frequently enough for accurate time-varying flow visualization.

Besides the need to visualize large data sizes, the ability to trace a large number of particles is also a valuable asset. While hundreds of thousands or millions of particles may be too many for human viewing, a very dense field of streamlines or pathlines is necessary for accurate follow-on analysis. The accuracy of techniques such as identifying Lagrangian coherent structures and querying geometric features of field lines relies on a dense field of particle traces.

6.4 Conclusions

The analysis and visualization of vector fields is a challenging, and critical capability for use in understanding complex phenomenae in scientific simulations. Integral curves in vector fields provide a powerful framework for the analysis of vector fields using a variety of techniques, such as streamlines, pathlines, streak lines, streamsurfaces, as well advanced Lagrangian techniques, such as FTLE, or Lagrangian Coherent Structures.

Due to the complexity of working with general vector fields outlined in Section **??**, stresses on the entire computational system are likely, including computation, memory, communication and I/O. In order to obtain scalable performance on large data sets, and large numbers of particles, care must be taken to design algorithms that are adaptable to the dynamic nature of particle advection. We have outlined several strategies for dealing with this complexity. These include dynamic work load monitoring of both particles and data blocks and re-balancing of work among the available processors, efficient data structures, and techniques for maximizing the efficiency of communication between processors.

Acknowledgment

We gratefully acknowledge the use of the resources of the Argonne and Oak Ridge Leadership Computing Facilities at Argonne and Oak Ridge National Laboratories. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Work is also supported by DOE with agree- ment No. DE-FC02-06ER25777.