Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields

Boonthanome Nouanesengsy, *Student Member, IEEE*, Teng-Yok Lee, *Student Member, IEEE*, and Han-Wei Shen

Abstract—Because of the ever increasing size of output data from scientific simulations, supercomputers are increasingly relied upon to generate visualizations. One use of supercomputers is to generate field lines from large scale flow fields. When generating field lines in parallel, the vector field is generally decomposed into blocks, which are then assigned to processors. Since various regions of the vector field can have different flow complexity, processors will require varying amounts of computation time to trace their particles, causing load imbalance, and thus limiting the performance speedup. To achieve load-balanced streamline generation, we propose a workload-aware partitioning algorithm to decompose the vector field into partitions with near equal workloads. Since actual workloads are unknown beforehand, we propose a workload estimation algorithm to predict the workload in the local vector field. A graph-based representation of the vector field is employed to generate these estimates. Once the workloads have been estimated, our partitioning algorithm is hierarchically applied to distribute the workload to all partitions. We examine the performance of our workload estimation and workload-aware partitioning algorithm in several timings studies, which demonstrates that by employing these methods, better scalability can be achieved with little overhead.

Index Terms—Flow visualization, Parallel processing, 3D vector field visualization, Streamlines.

1 Introduction

Because of significant gains in computational power within the past decade, scientists are now able to perform simulations at unprecedented spatial and temporal resolutions. The sheer size of data generated from these simulations, however, becomes a great challenge for effective visualization and data analysis. Not only that, it is difficult to perform such data intensive tasks in machines far away from where the data are stored. In addition, the memory and storage space available to most workstations is hardly enough to host the entire dataset. As a result, it becomes necessary to shift the visualization computations as close as possible to where the data reside, in many cases to the supercomputers where the simulations are performed.

The focus of this paper is on parallel streamline generation for large scale vector fields using massively parallel computers. Visualizing streamlines is still one of the most common methods for analyzing three dimensional flow fields. While it is well understood how to compute streamlines sequentially, generating streamlines for large datasets in parallel presents unique challenges. For a very large dataset, it is necessary to partition the data into smaller data blocks and distribute them to the compute processes, since it is not possible for one process to hold the entire dataset in memory. Because the spatial distribution and the lengths of streamlines across the entire domain can vary, load balancing is often difficult to achieve.

While several algorithms have been proposed in the past for parallel streamline generation [2, 21, 24, 4], there is a lack of robust models to measure the cost of load imbalance and therefore make it difficult to develop an efficient algorithm with any kind of optimality guarantee. In this paper, we propose a novel mathematical model to solve the data distribution and load balancing problem for parallel streamline generation using principles of optimization. To minimize the communication cost, our algorithm partitions the data statically, that is, no data shuffling among the compute processes will take place at run time once the data are distributed. With a given set of seeds, the compute processes generate streamlines in their own blocks, and communicate the particle positions once the particles reach the block boundaries. The need to communicate particle positions at certain synchronization points divides the whole parallel streamline generation process into a sequence of *rounds*. To ensure maximum parallel speedup, our mathematical model minimizes the difference of workload in each round using quadratic programming. Given a sequence of data blocks and their corresponding workload, our optimization method produces a grouping, or partitioning, of the blocks. Each partition is given to one compute process. One feature of our algorithm is that a data block can be assigned to multiple partitions. The degree of data replication is calculated by our optimization algorithm to achieve optimal load balancing.

Besides the mathematical model that is used to solve the load balancing problem, another important ingredient of our algorithm is a graph model used to predict the workload for every data block in every round. Blocks are encoded as nodes in the graph, and adjacent blocks will have edges connecting their corresponding nodes. Using this graph, an estimate of the workload for each data block can be obtained. The workload estimates are then used as input to our partitioning algorithm.

The paper is organized as follows. After reviewing existing work for parallel field line generation in Section 2, we present the basic idea of our parallel streamline generation algorithm in Section 3. The mathematical model and the optimization algorithm are presented in Section 4, followed by the workload estimation algorithm in Section 5. We present the results of our algorithm in Section 6, address the limitations and future works in Section 7, and conclude this paper in Section 8.

2 Related Work

Several systems have been proposed in recent years for parallel field line generation. Yu *et al.* proposed a hierarchical approach, advecting particles within blocks in different resolutions in parallel. One drawback of this approach is that field lines cannot extend beyond block boundaries, thus limiting their lengths [24]. Pugmire *et al.* proposed another parallel streamline generation system that monitors load balance throughout the computation, and dynamically adjusts the system using different strategies, including loading blocks when necessary [21]. Recently this system was extended to take advantage of multicore architectures [2]. Dynamically loading data blocks during runtime, however, can cause extra I/O overhead. In contrast, Peterka *et al.* investigated another system that uses a static partitioning of the vector field [20], which was also used by a recent map-reduce like [6] programming framework *DStep* for parallel streamline computation [15]. The partitioning of the blocks was produced using a round-robin scheme, regardless of the flow complexity and the spatial distribution

• *The authors are with The Ohio State University, E-mail: {nouanese,leeten,hwshen}@cse.ohio-state.edu.*

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

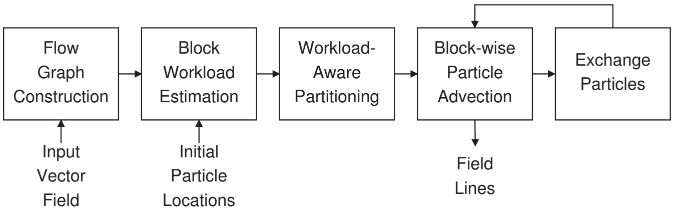

Flow Graph Construction
Block Workload Estimation
Workload-Aware Partitioning
Block-wise Particle Advection
Exchange Particles
Input Vector Field
Initial Particle Locations
Field Lines

Fig. 1. Overview of our parallel streamline generation system.

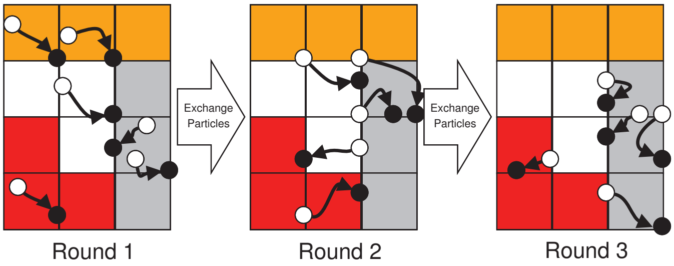


Fig. 2. Block-wise streamline generation. The vector field is decomposed into 4×3 blocks, which are separated into four partitions. Blocks are colored by their partition. In the beginning, six particles are placed in this vector field. The beginning and ending locations of particles are marked as white and black circles, respectively.

of particles. In this paper, we will demonstrate that by considering these pieces of information, our partitioning scheme can lead to better performance for streamline generation.

Graph-based representations have been utilized in other aspects for flow field visualization. Because graphs can provide an abstract representation of a flow field, Xu and Shen created a graph-based user interface as an overview of the vector field, with the goal of guiding the user to views with less occlusion [23]. Also, by representing the flow field as a graph, techniques for graph analysis can be applied for flow field analysis. One example is the flow field partitioning algorithm proposed by Chen and Fujishiro, which uses spectral clustering to partition a vector field to minimize communication overhead. [4].

The goal of graph partitioning is to decompose the graph into disjoint subsets with a minimum edge cost. While graph partitioning is an NP-hard problem [10], because of its applications in different areas, several techniques have been developed in the past, including quadratic programming [13], spectral clustering [4], and multi-level approaches [7, 14]. A detailed review of graph partitioning and its applications for scientific computation can be seen in the article by Scholegal *et al.* [22].

3 Parallel Streamline Generation

An overview diagram of our parallel streamline generation system is shown in Fig. 1. Our goal is to develop a high performance parallel streamline generation system for distributed-memory environments using MPI [11]. The inputs to our system are a vector field, and an initial set of user specified seed locations. The vector field is decomposed into blocks, and the resulting decomposition is encoded into a *flow graph*. A flow graph records how data blocks are related to each other according to the underlying flow directions. The flow graph is used in conjunction with the initial seeds to estimate the workload for each block during the course of streamline generation. Section 5 explains the flow graph and block workload estimation method in more detail. Once we have an estimate of workload for each block, our workload-aware partitioning algorithm is used to model the problem of assigning blocks and generating load balanced partitions as an optimization problem. Our partitioning algorithm is explained in detail in Section 4. Since a flow graph only depends on the vector field data, and can be reused for any subsequent runs, the flow graph construction is a preprocessing step. The block workload estimation and partitioning algorithms are performed at run time since they depends on the streamline seed locations.

Once the partitions have been calculated, all processes will load their assigned data blocks into memory. By loading all the data at once, MPI I/O can be employed to maximize bandwidth and I/O performance. Once loaded, blocks are kept in memory throughout the computation. The block assignments are static, meaning that block assignments will not change, and no further transfer of block data is performed.

Beginning with the initial streamline seeds, the particle tracing process is divided into a sequence of stages, or *rounds*. Fig. 2 illustrates a sequence of three rounds. For each round, each process advects every particle lying within its assigned blocks. A particle is traced until it travels outside the block it initially began in. Once all particles have been advected in this manner, the round is complete. Particles terminate either when they exit the vector field domain, when it is detected that a critical point is reached, or the user-specified maximum advection steps are exceeded. In between rounds, particles are exchanged among processes, with particles being sent to the processes that owns the blocks for the particles to continue the advection. Communication is done via point-to-point nonblocking sends and receives. Although non-blocking communication methods are used, the communication is coordinated synchronously, since a process must wait on its neighbors, who must wait on their neighbors, and so forth until the dependency extends to almost all processes. Even though communication is a crucial part of our system, here we focus on computational load imbalance, as it was found to be the main bottleneck to performance. Computation continues till all particles have terminated or the maximum number of rounds, a user specified parameter, is reached.

In our system, once a process finishes advecting its particles for one round, it must wait on all other processes to finish advection before particles can be exchanged. Load imbalance can occur if some processes require more computation time for particle tracing, forcing all other processes to wait. The amount of computation time a process needs for a round is the sum of the work of all its blocks for that round, so the assignment of blocks to processes is the most crucial element in affecting computational load balance. A load balanced partitioning should result in all processes having similar compute time, which will minimize waiting time. This condition should be satisfied for every round. The problem is further compounded by the fact that blocks may require different amounts of work for different rounds, so changing the assignment of a block to another process may improve the load balance of some rounds, but could also worsen the load imbalance of other rounds.

An important property of our system is that the workload for a block only depends on the particles within, but not to which partitions it belongs. For example, in Fig. 2, in the first round, the particle in the lower left block is advected, and stops when it crosses the block boundary. Even though the block that the particle traveled to is also owned by the same process, the particle does not continue advection until the second round. The result of this behavior is that blocks will require the same amount of work independent of which process owns it. This simplifies our optimization problem, since block workload amounts stay constant, and do not have to be adjusted based on whether any of the block's neighbors are present in the same partition.

If this were not the case, and advection could continue after a block boundary is reached, several issues exists. First, since the workload of a block is dependent on whether its neighbors are also in the same partition, the optimization problem becomes more complex to solve. Second, in our experience, it causes the computation to be more load imbalanced, since it is possible that some partitions may have much more work than others. For example, if a vortex spans two blocks and those two blocks are owned by the same process, then that process will take much longer to finish a round, as many of its seeds will be advected the maximum number of steps. Due to these challenges, existing techniques either analyze the flow field to ensure such a situation never happens, such as the algorithm proposed Yu *et al.* [24] by limiting the trace lengths, or allocates extra processes to monitor the

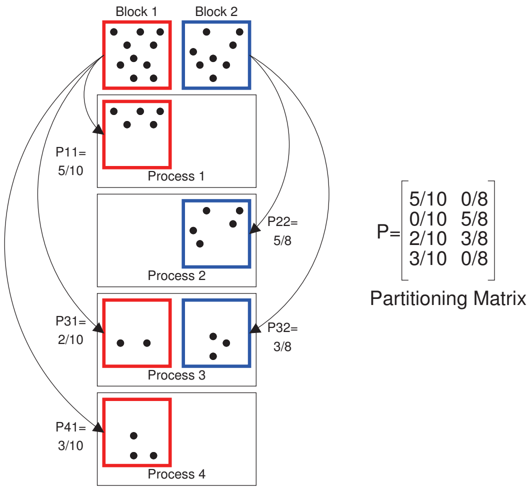


Fig. 3. Left: An example of a partitioning assignment composed of two blocks and four processes. The arcs from each block indicate which processes are responsible for tracing particles within this block, and the value P_{ki} near the arcs indicates the ratio of particles assigned to the processes. Right: The corresponding partitioning matrix, composed of the ratios, where rows represent processes and columns represent blocks.

workload and dynamically reassigns particles among the processes, the method used by Pugmire *et al.* [21]. While stopping particles after a block boundary is reached might lead to poor utilization of resources, Section 6 demonstrates that since our method creates a balanced workload, the idle time between consecutive rounds is minimized.

4 Workload-aware Data Partitioning

How the partitioning problem is modeled as an optimization problem is detailed in Section 4.1, and the solution to the optimization problem is shown in Section 4.2. For now, it is assumed that the exact workload for each block in every round is known *a priori*, and the discussion of how the block workload is estimated is deferred to Section 5.

4.1 Mathematical Model

4.1.1 Partitioning and Process Workload

The purpose of our partitioning method is to distribute the data blocks to the compute processes. The desired outcome of partitioning is to minimize the sum of workload differences among the processes, in all rounds. To help ensure a more even workload distribution, we do not limit blocks to being assigned to only one process. Instead, a bock can be assigned to multiple processes, if necessary. In this case, weights are assigned to each copy of the block, designating the percentage of particles to be processed for this copy. These weights are determined by the solution of the optimization problem.

Throughout this section, we assume that the number of data blocks is m, the maximum number of rounds is n, and the number of processes is p. The goal of our partitioning algorithm is to solve for the *workload ratio*, P_{ki}, which represents the percentage of the total amount of work from block i that process k is responsible for. We note that P_{ki} remains the same for all rounds, which means both the data and the workload assignment is fixed throughout the computation. This is done in order to minimize the communication and I/O cost. In the case that P_{ki} can only be a binary value of 0 or 1, each block is assigned to one process. On the other hand, if the value of P_{ki} is in $[0,1]$, then a block can be assigned to multiple processes, each of which is responsible for advecting a portion of the particles in the block determined by the value P_{ki}. Being able to have multiple copies of a data block at once is part of the parallel streamline system employed by Pugmire *et al.* [21]. Fig. 3 shows an example of how the workload of two blocks can be divided among four processes.

While we require that the workload ratio P_{ki} be the same for all rounds, the workload of a block will change from round to round. This is because particles will advect throughout the domain and hence a block may have different numbers of particles in different rounds. In our model, we maintain a different workload value for each block at every round. We denote the workload of block i in round j as L_{ij}. Given all workload values for every block in round j, and their workload ratios, P_{kj}, the total workload assigned to process k in round j, C_{kj}, is the sum of the workload from all blocks in round j multiplied by their workload ratios, or more specifically:

$$C_{kj} = \sum_{i=1}^{m} P_{ki} \times L_{ij},\ j = 1 \ldots n,\ k = 1 \ldots p \tag{1}$$

We note that since processes are assigned a percentage of the total workload for a block, the sum of the workload ratios for a specific block is always 1, that is:

$$\sum_{k=1}^{p} P_{ki} = 1,\ i = 1 \ldots m \tag{2}$$

Based on Equation 1, we can write the workload of all the processors in one round as a column vector. For p processes in the j-th round, $C_j = [C_{1j}, \ldots, C_{pj}]^T$ is:

$$C_j = \begin{bmatrix} C_{1j} \\ \vdots \\ C_{pj} \end{bmatrix} = \underbrace{\begin{bmatrix} P_{11} & \ldots & P_{1m} \\ \vdots & \ddots & \vdots \\ P_{p1} & \ldots & P_{pm} \end{bmatrix}}_{P} \begin{bmatrix} L_{1j} \\ \vdots \\ L_{mj} \end{bmatrix} \tag{3}$$

In Equation 3, the $p \times m$ matrix P contains all the workload ratios. We refer to this as the *partitioning matrix*. Fig. 3 illustrates a 4×2 partitioning matrix. Given the block workloads, L_{ij}, for all blocks in all rounds, the goal of our partitioning algorithm is to solve the partitioning matrix to achieve a load balanced streamline computation.

4.1.2 Partitioning Cost

In order to obtain a partitioning matrix that can give us the best load distribution, since we are modeling the partitioning as an optimization problem, we need to define a cost function to minimize. As stated previously, it is important to minimize the difference between the workloads of all processes in each round to reduce process idle time. To model the degree of load imbalance in each round, we can use the variance of the workload of each process. When all processes have similar amounts of workload, the variance will be small; otherwise, a larger variance will be observed if any process ends up having a much larger workload than the average case. Since our goal is to find a partitioning matrix that can lead to balanced workload in all rounds, we define the cost function as the variance of the processor workloads, summed over all rounds.

According to its definition, the variance of a set of elements $x_1, \ldots, x_p$ is the average of the square minus the square of the average, denoted as $\frac{1}{p}\sum_{k=1}^{p} x_k^2 - (\frac{1}{p}\sum_{k=1}^{p} x_k)^2$. As shown below, this allows us to derive the cost function, $f(P)$, as a quadratic form of the unknown partitioning matrix P. Because P is a matrix of real numbers, both first and second derivatives of $f(P)$ can be analytically computed. As a result, searching for such a matrix can be done by conventional nonlinear programming algorithms such as gradient descent [1].

Here we show the derivation of the cost function, $f(P)$, as a quadratic form of P. By denoting $\mathbf{VAR}(C_{1j},\ \ldots, C_{pj})$ as the variance of process workloads, C_{kj}, in the j-th round, the cost function, $f(P)$, as the sum of variances from all the n rounds can be expressed by Equation 4 below. By substituting the vector C_j as $C_j = P[L_{1j},\ \ldots,\ L_{mj}]^T$ from Equation 3, we obtain the cost function $f(P)$:

$$\begin{aligned} f(P) &= \sum_{j=1}^{n} \mathbf{VAR}(C_{1j},\ \ldots,\ C_{pj}) \\ &= \sum_{j=1}^{n} (\frac{1}{p} \sum_{k=1}^{p} C_{kj}^2 - (\frac{1}{p} \sum_{k=1}^{p} C_{kj})^2) \end{aligned}$$

$$= \sum_{j=1}^{n} C_j^T \left(\frac{1}{p} \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} - \frac{1}{p^2} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix} \right) C_j \quad (4)$$

4.2 Solving Partitioning Matrices

In solving the partitioning matrix to minimize our cost function, we allow the value P_{ki} to be any value in $[0,1]$, as mentioned earlier. That is, the workload for each block can be shared by multiple processes and thus replication of data blocks is allowed. Empirically, we found that disallowing the workload of a block to be divided often results in worse load balancing. A simple example is to partition two data blocks that have 100 and 1000 units of workload respectively to two processes. Without data replication, each block can only be assigned to one process, and thus the workload difference will be 900 units regardless of the block assignment. On the other hand, we can achieve a better load balancing if we give 45% of the workload from the second block and 100% of the workload from the first block to one process, and the remaining work to the other. The disadvantage of allowing block replication is an increase of the memory requirement for the program, which is discussed in Section 6.2.

To find the partitioning matrix P with block replication, conventional optimization algorithms can be applied to minimize the partitioning cost, although we need to address two issues. First, for m blocks and p processors, this optimization problem contains $m \times p$ unknowns, which can be slow to solve for data with thousands of blocks, and supercomputers with hundreds to thousands of processors. Second, the solution should satisfy the constraint that workload ratios are bounded in $[0,1]$, and the sum of workload ratios of all processes for any given block should be 1. It can be costly to solve the optimization problem with these constraints. To accelerate the optimization process, rather than directly solving the partitioning matrix P that minimizes Equation 4, we decompose the partitioning problem that involves more than two processes into a sequence of subproblems that involve two processes only. Hereafter we refer to partitioning data blocks to more than two processes as *Multi-way Partitioning* and partitioning that involves only two processes as *2-way Partitioning*. Solving 2-way partitioning involves fewer unknowns and simpler constraints versus the original cost function, which results in a problem that is much more computationally tractable.

4.2.1 2-Way Partitioning

To understand why a 2-way partitioning matrix involves fewer unknowns, it is to be noted that in a 2-way partitioning matrix P, any element in the second row P_{2i} is one minus the corresponding element in the first row, namely, $P_{2i} = 1 - P_{1i}$. In other words, in solving the 2-way partitioning matrix P we only need to consider the first row. The cost function, $f(P)$, for 2-way partitioning is still in a quadratic form, involving the first row of P, which is shown in Equation 6. This quadratic form for the cost function allows us to solve the optimization problem using conventional optimization algorithms, while the cost function contains fewer unknowns. In addition, the only constraint to be satisfied is that all elements in P are to be in the [0, 1] range. This makes the optimization problem much easier to solve.

First, we require a new derivation of the cost function for 2-way partitioning. The column vector C_j contains the workload C_{1j} and C_{2j} of two processes in the j-th round. From Equation 3, we can rewrite the column vector C_j as a linear function of the first row:

$$C_j = \begin{bmatrix} C_{1j} \\ C_{2j} \end{bmatrix} = \begin{bmatrix} P_{11} & \dots & P_{1m} \\ P_{21} & \dots & P_{2m} \end{bmatrix} \begin{bmatrix} L_{1j} \\ \vdots \\ L_{mj} \end{bmatrix}$$

$$= \begin{bmatrix} P_{11} & \dots & P_{1m} \\ 1-P_{11} & \dots & 1-P_{1m} \end{bmatrix} \begin{bmatrix} L_{1j} \\ \vdots \\ L_{mj} \end{bmatrix}$$

$$= \begin{bmatrix} +L_{1j} & \dots & +L_{mj} \\ -L_{1j} & \dots & -L_{mj} \end{bmatrix} \begin{bmatrix} P_{11} \\ \vdots \\ P_{1m} \end{bmatrix} + \begin{bmatrix} 0 \\ \sum_{i=1}^{m} L_{ij} \end{bmatrix} \quad (5)$$

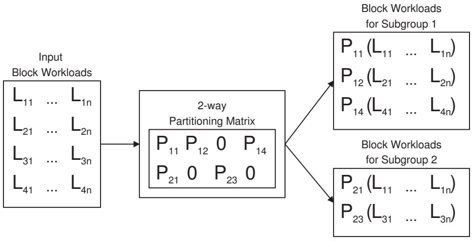


Fig. 4. An example of hierarchical partitioning. An initial workload matrix is used to generate a partitioning matrix, where each row represents a partition. Then the workload matrix is weighted by the resulting partition weights to generate two new workload matrices. More partitions are generated recursively.

To simplify the notation, we denote this linear function as $C_j = B_j p_1 + c_j$ where p_1 is a column vector equal to the transpose of the first row vector in the partitioning matrix P. By substituting this linear function for C_j in the original cost function in Equation 4, we obtain the cost of a 2-way partitioning matrix P in Equation 6 shown below, which is a quadratic form of the vector p_1:

$$f(P) = \sum_{j=1}^{n} (B_j p_1 + c_j)^T \left(\frac{1}{2} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \frac{1}{4} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right) (B_j p_1 + c_j) \quad (6)$$

4.2.2 Penalty for Data Replication

When solving the partitioning matrix with block replication allowed, a trivial but undesired solution is to copy each data block to all processes, that is, all elements of matrix P have an equal weight. While this partitioning certainly achieves perfect load balancing, it means that each process is required to hold in memory a complete copy of the data, an undesired solution for large datasets. To avoid this trivial solution, we want to distribute the workload of a block to only a few processes to minimize memory and I/O overhead.

In order to achieve this goal, we add extra terms to the cost function to penalize the cases where a block is distributed to too many processes. Too much duplication is unwanted because of limited available system memory and to limit I/O overhead. In the case of 2-way partitioning, the penalty is high for a partitioning matrix where all elements have a value of 0.5; otherwise, if most elements are 0 or 1, the penalty is low. This can be realized by using the penalty term shown below in Equation 7, which involves the first row, $[P_{11}, \dots, P_{1m}]$, of the partitioning matrix P. The penalty is the sum of all elements in the first row, each of which is weighted by one minus its value. It can be shown that if all elements are either 0 or 1, this weighted sum will be zero. Otherwise, since the values of the elements in P are in [0, 1], the penalty reaches a maximum when all elements are 0.5.

$$g(P) = \sum_{i=1}^{m} (1 - P_{1i}) \times P_{1i} \quad (7)$$

By adding the penalty term, the final cost function for partitioning, $h(P)$, becomes:

$$h(P) = f(P) + \lambda g(P) \quad (8)$$

where λ is used as a trade-off factor between the partitioning cost defined in Equation 4 and the penalty defined in Equation 7. More details about the influence of this parameter is discussed in Section 6.2.

4.2.3 Solving 2-way Partitioning via Nonlinear Programming

To find the matrix P that minimizes $h(P)$, the selection of which optimization algorithm to use is crucial for its performance. Since the second derivative of $h(P)$ can contain negative eigenvalues, $h(P)$ can be non-convex. Under these circumstances, there is no analytical form to compute the global minimum. Therefore, we need to iteratively search for the local minimum.

Since the second derivative, i.e., Hessian matrix, of $h(P)$ exists, using Newton's method to minimize $h(P)$ should ideally take only a few iterations to converge. Newton's method, however, requires the Hessian matrix of $h(P)$, whose size is $m \times m$ for m blocks. Since analytically computing the $m \times m$ Hessian matrix can be time consuming, we choose to utilize the quasi-Newton method, which approximates the Hessian matrix locally. We use an implementation of the quasi-Newton method in the nonlinear programming solver OPT++ [18], where the Hessian matrix is approximated by the limited-memory version of Broyden–Fletcher–Goldfarb–Shanno (L–BFGS) algorithm [19]. In our tests, the quasi-Newton method obtained a comparable optimization result with those by Newton's method, but the quasi-Newton method took considerably less time, since the computation of the Hessian matrix is avoided.

4.2.4 Hierarchical Multiway Partitioning

In order to obtain a multi-way partitioning, the 2-way partitioning is hierarchically applied in order to generate the correct number of partitions. In the beginning, our algorithm takes the original workload values of all blocks over all rounds, and computes a 2-way partitioning matrix, where each of the two rows represents a partition. In order to continue subdividing, new workload values must be computed for each new partition. For each partition, if it has a non-zero workload ratio for a block, the workload of this block in all rounds are weighted by the workload ratio. The weighted workloads from the assigned blocks form the new workload values for that partition. Once new workload values are computed for each partition, 2-way partitioning is applied, and the process continues until the desired number of partitions is reached. Because the number of partitions after splitting is always two, currently our method only supports creating partitions that are a power of two.

Fig. 4 is an example of how the results of 2-way partitioning is used to generate new workload values. Here the block workloads are organized as a matrix in which each row represents the workload of all rounds of a block. For the i-th row, if the workload ratio P_{ki} is nonzero, this row is weighted by the workload ratio to become a row in the new workload matrix for the new partition.

5 Flow Graph and Block Workload Estimation

So far it has been assumed that the exact workload for each block at every round is known beforehand. Unfortunately, this knowledge cannot be obtained unless the actual particle advection calculations are performed. Because of this, we propose a block workload estimation algorithm designed to efficiently estimate the workload of each block. In order to estimate the block workload, first an abstract representation of the vector field, or flow graph, is computed, then initial seed locations are used in conjunction with the flow graph to estimate the workload of each block.

A flow graph is an abstract representation of the vector field encoded as a directed graph. Each block in the domain is represented as a node. For every pair of adjacent blocks, their corresponding graph nodes are connected by two weighted, directed edges. The weight of an edge from node A to node B represents the probability that a particle located in A will travel to B. Adjacent nodes will have two edges in opposite directions because the probability of a particle traveling from A to B may be different than the probability going from B to A.

In order to calculate the edge weights, particles are placed uniformly within each block, and are advected until they exit the block. The weight of an edge going to a neighbor is the ratio between the number of particles that reach that neighbor block and the total number of particles. The sum of all edge weights originating from one node is always 1 or less. The sum can be less than 1 if particles travel outside the vector field domain, or if particles encounter a critical point within the block.

Once the flow graph is computed in the preprocessing step, the workload for each block can be estimated at run time for a given set of streamline seeds. The input is a flow graph and a seed set. The output is an estimated number of particles in each block per round. The initial seed locations are examined to see how many seeds each block will begin with. These values are used to initialize the number of particles for a block's corresponding node. For each round, every edge in the flow graph is traversed, and the number of current particles in the source node is multiplied by the edge weight, and then added to the number of particles of the destination block for the next round. This continues until the maximum number of rounds is reached.

At this point, the estimated number of particles for each round can be used directly as the workload of a block, but in practice the number of particles do not correlate perfectly with computation time. This problem arises because of the difference in flow field characteristics for each block. For some blocks, particles may tend to travel only a few steps before exiting. On the other hand, particles in other blocks may have a tendency to require several steps. For example, a block containing a vortex will require more time to trace its particles versus a block without any vortices, assuming they both contain the same number of particles. Therefore, a more accurate measure of the computation time for a block is the number of advection steps generated. In order to estimate this, the average number of advection steps per particle is needed for each block. This value is calculated in the flow graph generation step. As particles are being advected in order to calculate edge weights, the total number of advection steps calculated is kept, and the average can be calculated from there. The final value for the estimated workload for a block is then the estimated number of particles multiplied by the average number of advection steps per particle. These estimated values are then used as input to the previously described workload-aware partitioning algorithm discussed in Section 4.

Table 1. Datasets used for strong scaling and weak scaling tests. The size of each dataset, as well as the number of seeds used in each of our tests are listed.

Dataset	Size (GB)	# Particles (Strong)	# Particles per Processor (Weak)
Nek	12	256K	64
Plume	5.81	256K	64
Ocean	3.86	256K	64
Flame	18.69	128K	32

6 Results

To test the performance of our algorithm, strong scaling and weak scaling tests were performed. Four datasets were used in our experiments. The first dataset is a thermal hydraulics simulation generated by the Nek5000 solver, which we refer to as *Nek*. The dataset was created from a simulation of the MAX experiment [17], based on the large-eddy simulation of Navier-Stokes equations, and resampled into a regular grid of $2048 \times 2048 \times 2048$. The dataset *Ocean*, with a resolution of $3600 \times 2400 \times 40$, is the result of an eddy resolving simulation with 1/10 horizontal spacing at the equator [16]. The third dataset, *Flame*, is a simulation of fuel jet combustion [8, 12], based on S3D, a solver for fully compressible Navier-Stokes flow [3]. The spatial resolution is $1408 \times 1080 \times 1100$. The fourth dataset, *Plume*, was produced by a simulation of the thermal downflow plumes on the surface of the sun, and has a spatial resolution of $504 \times 504 \times 2048$. The size of all four test datasets are summarized in Table 1, and an image of each dataset is shown in Fig. 5.

6.1 Parallel Streamline Generation Performance

To test the performance of our parallel streamline generation algorithm with our data partitioning scheme, tests were conducted on *Surveyor*,

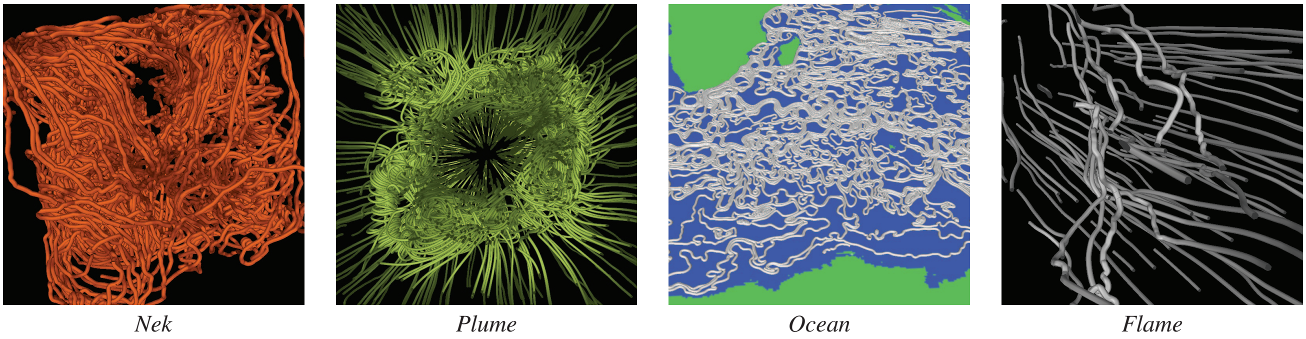
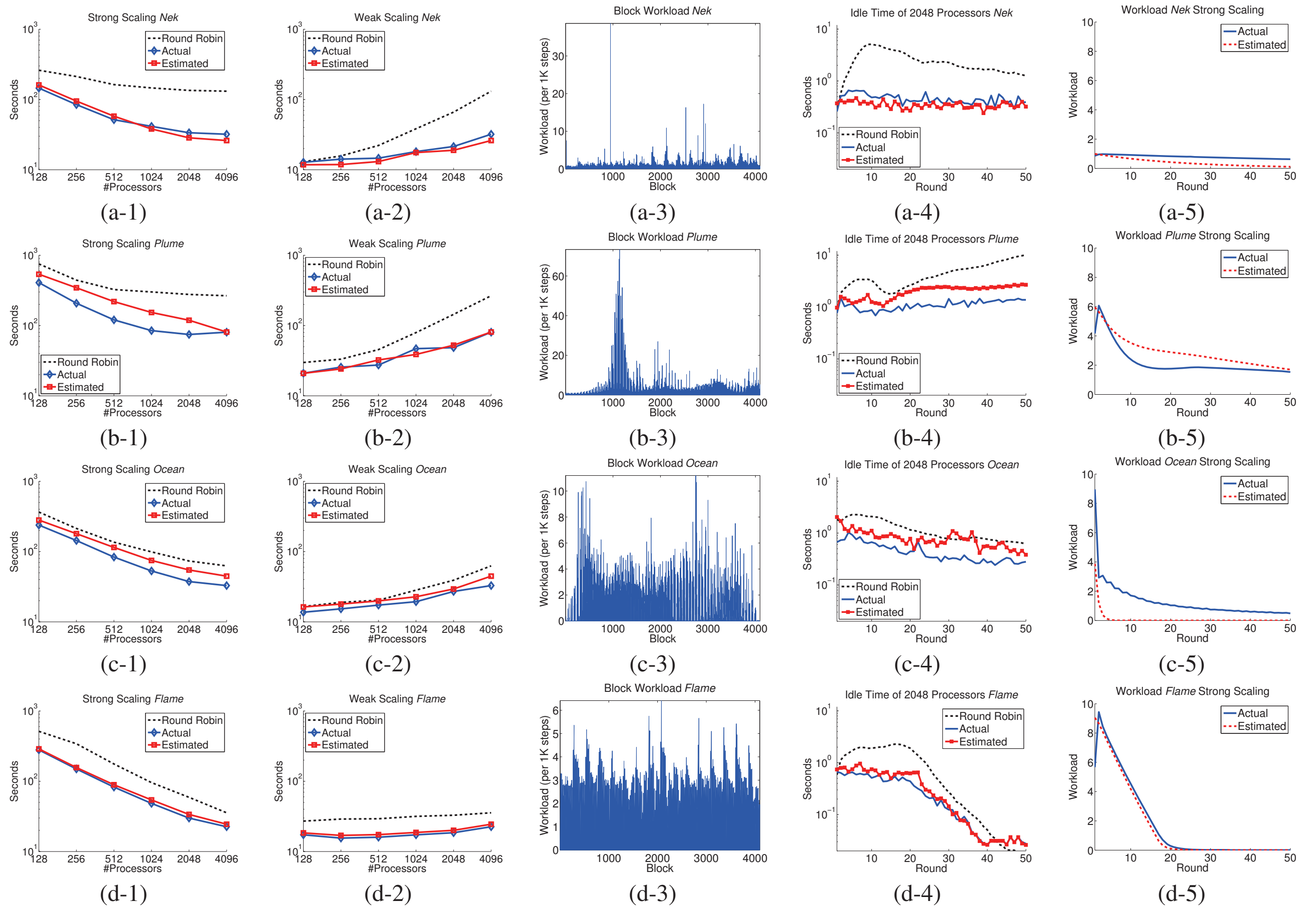


Fig. 5. Images of our test datasets.



(a-1) (a-2) (a-3) (a-4) (a-5)

(b-1) (b-2) (b-3) (b-4) (b-5)

(c-1) (c-2) (c-3) (c-4) (c-5)

(d-1) (d-2) (d-3) (d-4) (d-5)

Fig. 6. Performance tests. The rows from top to bottom represent the datasets *Nek*, *Plume*, *Ocean*, and *Flame*. The first and second column show the timing for strong scaling and weak scaling, respectively. The performance of round-robin partitioning (*Round Robin*), actual-workload-based partitioning (*Actual*), and estimated-workload-based partitioning (*Estimated*) is compared. The third column lists the average workload per block. The fourth column shows the average waiting time in logarithm scale per process in all rounds, which was measured when doing strong scaling tests for 2048 processes. The fifth column shows the average actual workload and the estimated workload in all rounds.

an IBM Blue Gene/P (BG/P) supercomputer at Argonne National Laboratory. This BG/P system includes 1024 compute nodes. Each node contains 4 cores and 2GB of system memory, for a maximum of 4096 cores.

Several timings tests were performed to study the performance gain of parallel streamline generation using our data partitioning scheme. For each test, partitions were generated by using our workload estimation and partitioning method. We compare our algorithm with a recent work published in 2011 by Peterka *et al.* [20], where a round-robin data assignment scheme is used, in which blocks are assigned to processes using a three-dimensional block-cyclic distribution. Similar to Peterka *et al.*'s system, our streamline generation was also based on a particle advection library called OSUFlow, which is also used in the package VAPOR from National Center for Atmospheric Research [5]. To study the efficacy of the workload estimation algorithm described in Section 5, we collected the actual workload in each of the blocks in every round and used those exact values to generate partitions using our partitioning algorithm. The results of using these partitions are compared with the results from our estimation method. We note that the purpose of collecting the actual workload is only for experimentation, since it requires the generation of all streamlines beforehand, which is obviously not a plausible solution. In the following, we differentiate the partitioning results generated by using the actual workload numbers and our estimated numbers with the terms *actual-*

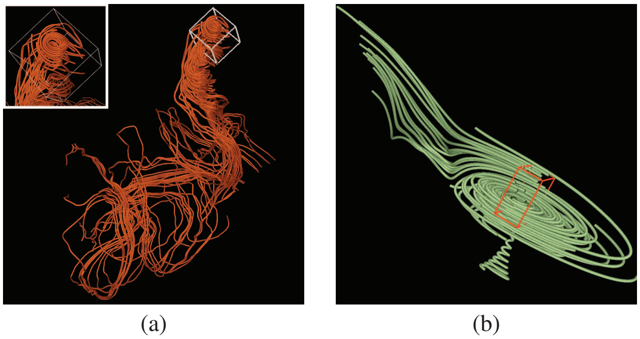


Fig. 7. The blocks with maximum average workload in *Nek* (a) and *Plume* (b), highlighted in the white boxes. (a): Streamlines originating from the busiest block in *Nek*. In the upper left corner, a close-up of the block is shown. (b): Streamlines terminating in the busiest block in *Plume*.

workload-based partitioning and *estimated-workload-based partitioning*, respectively.

We conducted both *strong scaling* and *weak scaling* tests involving all of our datasets, where strong scaling means the total amount of work remains the same as the process count increases, and weak scaling means the workload increases proportionally with the process count. For ideal parallel speedup, strong scaling results should show that the computation time decreases linearly as the process count increases, while weak scaling results should show the computation time remaining constant. In our strong scaling tests, a constant number of particles were used. In the weak scaling tests, more particles are generated proportionally as more processes are used. Table 1 lists the number of total particles for strong scaling and number of particles per process for weak scaling tests in our experiments. In all experiments, particles were allowed to go up to 50 rounds. For scaling tests, the particles were placed randomly throughout the domain, while the tests with uneven placed particles are discussed in Section 6.3. Each dataset was divided into 4K blocks, and processor counts of 128, 256, 512, 1024, 2048, 4096 were used. The selection of block size and the parameter λ in Equation 8, and how streamline generation time can be influenced by them, are discussed in more detail in Section 6.2.

Fig. 6 lists the results of all tests. The first two columns graph the total computation time, including communication, for each dataset. Fig. 6 (a-1) and (a-2) show the timings for the dataset *Nek*. It can be seen that as more processors were used, the amount of speedup using our algorithm compared to the round-robin scheme increased for both strong scaling and weak scaling tests. For strong scaling tests on 4K processors, both actual-workload-based partitioning and estimated-workload-based partitioning are 5 times faster than the round-robin method. Fig. 6 (b-1) and (b-2) list the timings for the *Plume* dataset, which also show a similar behavior when using more processors. Fig. 6 (c-1) and (c-2) show the timings for the dataset *Ocean*. Although using our partitioning method still leads to better performance, the performance gain versus round-robin flattened out at high processor counts. For this dataset, the percent speedup between using estimated-workload-based partitioning and round-robin ranged from 30% to 80% for strong scaling. For weak scaling tests, the performance numbers remain close when using less than 1K processors, but the difference increases to about 20% to 40% speedup over round-robin for larger processor counts. It can also be seen that partitioning based on the actual workload is faster than the partitions based on estimated workload, although estimated-workload-partitioning consistently performs better than round-robin. For the *Flame* dataset, the speedup compared to round-robin in strong scaling and weak scaling tests were about 40% to 120%, and 40% to 70%, respectively. Partitioning based on the actual workload led to only slightly better performance than that based on the estimated workload.

From the first two columns of Fig. 6, it can be seen that the performance difference between our method and round-robin increases as the numbers of processors increase for the *Nek* and *Plume* datasets, while the performance difference between the two methods flattens out for *Ocean* and *Flame* at larger processor counts. To explain this behavior, it was found that the degree of speedup is related to the workload difference among the data blocks. The third column in Fig. 6 shows the average workload of all blocks for the different datasets, where all blocks are laid out along the x-axis, and the y-axis shows the average workload from all rounds. The distributions of workload in Fig. 6 (a-3) and (b-3) contain several peaks, implying that some blocks receive more work than other blocks, mostly due to the convergence of particles. For this case, our algorithm can properly replicate the data blocks and distribute the work more evenly, and hence performed better than the round-robin scheme. In contrast, the block workloads in Fig. 6 (c-3) and (d-3) are more uniform than that in Fig. 6 (a-3) and (b-3), which means that for *Ocean* and *Flame*, the blocks generally have more even workloads, which presented a simpler case and helped the round-robin scheme to narrow the performance gap between it and our algorithm. Nevertheless, our algorithm performs consistently better than round-robin.

Fig. 7 (a) displays the data block with highest average workload in *Nek*, and the streamlines that originate from this block. From these high curvature streamline segments, it can be seen that these streamlines take a large number of steps within this block, therefore causing a longer compute time for particle advection. Similarly, Fig. 7 (b) reveals the block with the highest average workload in *Plume*. The image clearly shows that this block contains the center of a circular sink. Therefore, more and more particles are gathered toward this block in the later rounds, thus causing higher block workload.

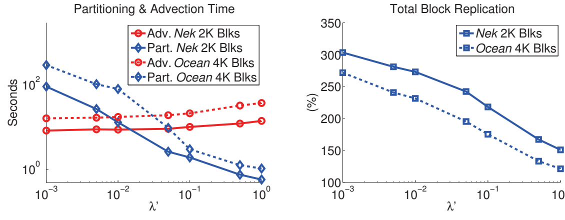
To further demonstrate that our partitioning algorithm produces load balanced partitions, we measured the process idle time in all rounds. The average idle time per process in all rounds is listed in the fourth column of Fig. 6. The results were collected from our strong scaling test using 2048 processes. By comparing Fig. 6 (a-4) and (b-4) with (c-4) and (d-4), it can be seen that when the round-robin partitioning scheme is used, the average waiting time for *Nek* and *Plume* is much higher than that of *Ocean* or *Flame*, which again confirms that the distribution of workload among blocks in the latter two datasets are more even. We can also see that estimated-workload-based partitioning almost always leads to smaller process idle time than round-robin. Smaller idle times indicate that the compute times are similar to each other, which proves that our algorithm produced more load balanced partitions. When using estimated-workload-based partitioning for *Ocean*, we observed a few rounds having higher wait time than round-robin, although the overall parallel performance of the entire run using our algorithm was still better.

When using estimated-workload-based partitioning for datasets *Plume* and *Ocean*, idle times are consistently longer and streamline generation is slower than partitioning based on the actual workload. Our hypothesis is that the amount of error of the workload estimation was higher for those datasets, which negatively impacted the performance. To verify this hypothesis, we computed the average actual workload and the average estimated workload over all rounds, which are plotted in the fifth column in Fig. 6. Comparing Fig. 6 (a-5) and (d-5), we can see that the difference between the actual workload and estimated one in Fig. 6 (b-5) and (c-5) are larger, which means that for *Plume* and *Ocean*, the estimation error is larger than the other two datasets, thus leading to less efficient partitions versus the partitions produced when using the actual workload.

6.2 Parameter Specification

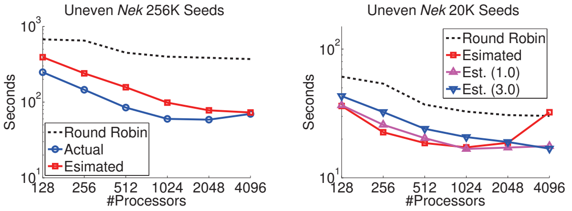
The parameter introduced in Equation 8, λ, represents a trade-off between the variance of the workload of the resulting partitions and the amount of block replication. Smaller values of λ leads to solutions with better load balancing, since it permits a higher degree of data replication, while larger values of λ has smaller memory overhead.

In Equation 8, since $g(P)$ is a sum of products of several workload ratios, and $f(P)$ measures the variance of workload in the compute processes, they usually have different scales. Therefore, the choice of λ needs to consider f's value range, such that λg will have a value



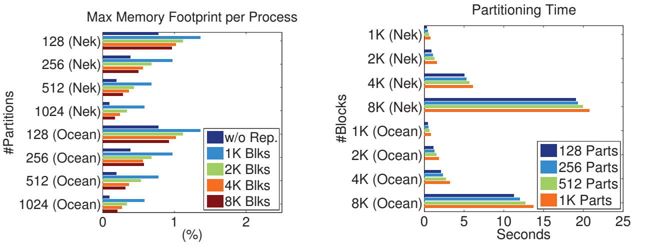
(a) (b)

Fig. 8. Impact of λ' on *Nek* (solid lines) with 2048 blocks and 1024 partitions and *Ocean* (dashed lines) with 4096 blocks and 1024 partitions. (a): Impact of λ' on performance. Note that partition creation times were performed on a sequential workstation, while particle advection times were gathered from parallel runs. (b): Effect of λ' on the total amount of block replication. The y-axis represents the ratio between the total number of blocks after partitioning and the original number of blocks.



(a) (b)

Fig. 10. Performance of uneven seeding when seeding the 40 busiest blocks in *Nek* with (a) 20K seeds and (b) 256K seeds.



(a) (b)

Fig. 9. Analysis of the partitioning algorithm for *Nek* and *Ocean*. (a) Maximum memory requirement per process for partitioning with replication. The x-axis is the maximum memory requirement for any one process, as a percentage of the original data size. (b) Time required to calculate partitions.

close enough to $f(P)$ so as to influence the final optimization outcome. We found that λ is usually data dependent, which makes it more difficult to choose a proper value without knowing the characteristics of the input data. To address this issue, we do not directly specify the value of λ. Instead, we specify another parameter, λ', which is independent of the scale of f. We then compute λ based on λ' and a sample of f, denoted as f_{sample}, as shown in Equation 9. We select f_{sample} using $f(P_0)$ where P_0 is the partitioning matrix that assigns the blocks to the compute processes using a round-robin scheme. We use this to get an estimate of the typical value of $f(P)$, since round-robin data assignment is one of many possible choices for data partitioning.

$$\lambda = \lambda' f_{sample} \tag{9}$$

We conducted tests with different values of λ', and measured the performance of both the optimization solver and the performance of the streamline generation using the resulting partitions. Fig. 8 (a) presents the performance of both the partitioning solver and corresponding particle advection using two datasets with varying values of λ'. The solver was tested on a machine equipped with two 2.27 GHz Xeon E5520 CPUs. Advection times were the result of parallel runs using the resulting partitions. We note that while our streamline generation algorithm is designed to run on large scale supercomputers, our data partitioning optimization problem currently is solved separately using a sequential workstation. Fig. 8 (b) shows the percentage of extra blocks after partitioning with replication. It can be seen that as λ' decreases, more blocks are replicated. As a result of these tests, λ' was empirically set to 0.1, since λ' values smaller than 0.1 resulted in the optimization solver becoming much slower, while the performance gains of the streamline generation computation diminishes.

The solution of our optimization model with different degrees of data replication was studied to see how it can influence the memory requirement for the compute processes. The maximum amount of memory that will be needed by one compute process was studied under different parameter settings. The datasets *Nek* and *Ocean* were divided into 1K, 2K, 4K, and 8K blocks, which were used to produce partitions for 128, 256, 512, and 1024 processes, using our partitioning algorithm. Fig. 9 (a) shows the amount of memory needed for one process, as compared to a partitioning that does not allow replication, where the y-axis shows the additional memory required in as a percentage of the original data size. It can be seen that as the dataset is divided into more blocks, the additional memory needed for partitions that allow data replication continues to decrease.

The number of blocks also influences the time to solve the partitioning matrix, since the number of workload ratios, the unknown to solve, depends on the number of blocks and number of partitions. Fig. 9 (b) shows the performance of the optimization solver for the datasets *Nek* and *Ocean*. It can be seen that as the datasets are partitioned into more blocks, more time is needed to solve the partition matrix. While the time complexity of our optimization algorithm depends on the number of iterations to optimize the cost function, $h(P)$, in Equation 8, our observation was that the computation time was proportional to the squared of the number of blocks. It should be noted that currently our optimization algorithm is solved separately on a single machine. In the future, we would like to parallelize the optimization algorithm to improve the performance.

Fig. 9 (b) also shows that the number of partitions has a smaller influence on the time needed to solve the partitioning matrix, as compared to the number of blocks. This is because that multi-way partitioning is hierarchically solved via a sequence of 2-way partitioning. The first 2-way partitioning involves more blocks to be solved than later 2-way partitioning, and thus the majority of the computation time is spent solving the first one.

6.3 Performance for Uneven Seeding

In addition to placing seeds randomly over the entire domain, it is also important to consider how our algorithm performs when seeds are unevenly distributed. Often times seeds are strategically placed in order to reveal specific flow features. The definition of a feature, though, is application-dependent. Our strategy for uneven seeding is based on Fig. 7, where it can be observed that the busiest blocks contain more turbulent flow. Statistics collected from the original strong scaling tests were used to determine the 40 busiest blocks. Seeds were then placed only in those blocks.

The strong scaling results of uneven seeding are shown in Fig. 10, using 256K seeds and 20K seeds in *Nek*. By comparing Fig. 10 (a) with Fig. 6 (a-1), we can see that even if seeds are unevenly placed, our method still has better scalability than round-robin. In the case of 20K seeds, though, round-robin seems to scale just as well as our method. The red curve in Fig. 10 (b) shows the results of our method with λ' being set to 0.1. Initially, the wall time of our method is faster than round-robin, but beginning with 1K processes, the wall time begins to increase, and at 4K processes is actually greater than round-robin. The main reason for this decrease in performance is because even though better load balancing is being achieved from the data blocks being replicated, this data replication will cause extra communication. For example, sending a set of seeds to a block that is replicated 100 times will require 100 sends, whereas if that block is only replicated

10 times, then only 10 sends are needed. As fewer seeds are placed, streamline computation will take less time overall, and the communication overhead will begin to dominate the wall time. To lessen the communication overhead, we can change the parameter λ' to reduce the replication of data. Fig. 10 (b) shows the results when setting λ' to 1.0 and 3.0. It can be seen that as λ' increases, the wall time becomes slower for less than 1K processes, but scalability improves for more than 1K processes. Overall, our tests indicate that if a streamline run uses a large number of seeds and requires a large amount of computation, then our method will provide better performance than round-robin, even if the seed placement is uneven. On the other hand, if the number of seeds used is low, when adding in the preprocessing time, our method may not be beneficial.

6.4 Flow Graph Construction and Estimation

In addition to partitioning and particle advection, the computation time required for an entire streamline computation also includes workload estimation and flow graph construction. Workload estimation needs to be recomputed when a new set of seeds are specified. The compute time for this step, though, is generally very fast. For example, estimating the workload of 4K blocks with an initial 256K seed set requires roughly 1.5 seconds.

The preprocessing step of constructing the flow graph can be quite time consuming compared to the workload estimation step. Because the edge weight calculations for nodes in a flow graph can be independently computed, the graph construction can easily be performed in parallel. Our parallel implementation assigns an equal number of blocks to each process. Fig. 11 (a) shows the performance of computing the flow graph in parallel. It can be seen that while using more processes can accelerate the performance, the speedup obtained by using more than 512 processes becomes sub-linear. The reason behind the non-optimal scalability for larger process counts is because each block is assigned to only one process. Since some blocks will take more time than others, the wall clock time is bounded by the slowest processes.

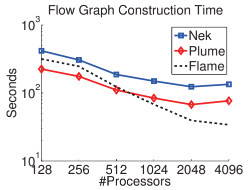


Fig. 11. Timing of Flow Graph Construction.

When adding the flow graph preprocessing time to the run time of our tests in Fig. 6, the resulting time will often be close to or be slower than the corresponding round-robin time. Fortunately, the flow graph only needs to be calculated once for a dataset, so the pre-processing time will be amortized over multiple runs.

In addition to the performance, the accuracy of the flow graph is also crucial. To decide the number of seeds per block for graph construction, we constructed flow graphs with 2.5K, 5K, 10K, and 50K seeds per block, and compared the weights of the same edge between the different versions to see how the number of seeds influenced the resulting edge weight. The average difference between different versions of the same edge is 0.0074. Considering that the possible range of an edge weight is [0, 1], edge weights remain relatively stable when more seeds are used for the flow graph.

6.5 Performance Overhead

Since our algorithm introduces extra stages, including flow graph construction, workload estimation, and partitioning, it is crucial to verify whether these additional stages will cause too much overhead for our method to be beneficial.

To analyze the time complexity of these stages, it should be noted that the performance of these extra stages is independent of the number of particles used at run time. For flow graph construction, its performance mainly relies on the number of blocks and the number of seeds used for preprocessing, not the number of seeds at run time. Therefore, the flow graph can be computed once and be re-used for an arbitrary number of seeds. Moreover, since flow graph construction can be accelerated by parallel processing, the overhead can be further reduced.

Both workload estimation and flow field partitioning need to be recomputed when given a new set of seeds. Their performance depends on the number of blocks and the number of rounds. For workload estimation, the time complexity is linearly proportional to the number of blocks. For flow field partitioning, from Fig 9 (b), we can observe that its time complexity is proportional to the square of the number of blocks. A possible explanation is that the optimization algorithm needs to approximate the Hessian matrix, whose size is the number of blocks squared.

Therefore, whether the overhead of partitioning matters depends on the total amount of streamline computation involved. If the number of particles is large, this overhead can be ignored; otherwise, simpler schemes such as round-robin might be preferred. Overall, our algorithm is better suited for large scale seeding, and can be applied to compute-intensive applications, such as the computation of Finite-time Lyapunov exponent (FTLE), which generates an excessive number of flow lines [9].

7 Limitations and Future Work

As our approach aims to load-balance streamline computation for steady flow fields, there exist several limitations, which will be addressed in future work. One limitation, found through our tests in Section 6.3, is that our method is better suited for computations involving a large number of seeds, and may not be worthwhile for runs with a small number of initial seeds.

Another major limitation is our streamline computation model. Since the current model is designed for steady flow, the blocks are statically assigned to processes in order to avoid extra I/O or communication overhead that could result if dynamic partitioning were used. However, for unsteady flow fields, it is generally unfeasible to load the entire data into memory, especially if there are many timesteps involved. Because of this, using dynamic partitioning may be a more natural choice for unsteady flow fields.

Another limitation is that the current model is designed for regular grids. Extending it to unstructured grids is a planned future work. To do so, the current flow graph model and workload estimation will probably have to be revised. Since the grid size in unstructured grids can vary, the flow graph will probably need to store extra information to consider size differences.

8 Conclusion

In this paper, we propose a workload model for load-balancing parallel streamline generation. Our model formulates the relationship between the block workload and the corresponding partitioning workload, allowing us to treat the partitioning as an optimization problem that can be solved by conventional optimization algorithms. Since our model requires the knowledge of the block workload, we also propose a graph-based workload estimation algorithm to predict the workload for each block. Testing results show that our partitioning algorithm can reduce the idle time of each process, leading to more efficient parallel streamline generation.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. We would like to also thank Aleks Obabko, Paul Fischer, and Tom Peterka of Argonne National Laboratory, Mathew Maltrud of Los Alamos National Laboratory, Ray Grout of the National Renewable Energy Laboratory, and Jackie Chen of Sandia National Laboratory for providing our test datasets. We would also like to thank Fabian Benitez-Quiroz for the discussion about the optimization algorithm. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This work was supported in part by NSF grant IIS-1017635, US Department of Energy DOE-SC0005036, Battelle Contract No. 137365, Los Alamos National Laboratory Contract No. 69552-001-08, and Department of Energy SciDAC grant DE-FC02-06ER25779.

References

[1] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, 1999.

[2] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy. Streamline integration using MPI-hybrid parallelism on large Multi-Core architecture. *IEEE Transactions on Visualization and Computer Graphics*, 99(PrePrints), 2010.

[3] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2:015001, 2009.

[4] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *PacficVis '08: Proceedings of the IEEE Pacific Visualization Symposium 2008*, pages 87–94, Mar. 2008.

[5] J. Clyne, P. Mininni, A. Norton, and M. Rast. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics*, 9, 2007.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[7] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *IPDPS '06: Proceedings of the International Parallel and Distributed Processing Symposium 2006*, pages 10 pp.+, 2006.

[8] T. F. Fric and A. Roshko. Vortical structure in the wake of a transverse jet. *Journal of Fluid Mechanics*, 279:1–47, 1994.

[9] H. G. Lagrangian structures and the rate of strain in a partition of two-dimensional turbulence. *Physics of Fluids*, 13(11):3365–3385, 2001.

[10] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *STOC '74: Proceedings of the ACM Symposium on Theory of computing 1974*, pages 47–63, 1974.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[12] R. W. Grout, A. Gruber, C. Yoo, and J. Chen. Direct numerical simulation of flame stabilization downstream of a transverse fuel jet in cross-flow. In *Proceedings of the Combustion Institute*, volume 33, pages 1629–1637, 2010.

[13] W. W. Hager and Y. Krylyuk. Graph partitioning and continuous quadratic programming. *SIAM Journal on Discrete Mathematics*, 12:500–523, Oct. 1999.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[15] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *SC '11: Proceedings of the ACM/IEEE Conference on Supercomputing 2011*, 2011. To appear.

[16] M. E. Maltrud and J. L. McClean. An eddy resolving global 1/10 ocean simulation. *Ocean Modelling*, 8(1–2):31, 2005.

[17] E. Merzari, W. Pointer, A. Obabko, and P. Fischer. On the numerical simulation of thermal striping in the upper plenum of a fast reactor. In *ICAPP '10: Proceedings of the International Congress on Advances in Nuclear Power Plants 2010*, 2010.

[18] J. C. Meza, R. A. Oliva, P. D. Hough, and P. J. Williams. Opt++: An object-oriented toolkit for nonlinear optimization. *ACM Transactions on Mathematical Software*, 33(2), June 2007.

[19] J. Nocedal. Updating Quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[20] T. Peterka, R. Ross, B. Nouanesengsey, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *IPDPS '11: Proceedings of IEEE International Parallel & Distributed Processing Symposium 2011*, 2011. To appear.

[21] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *SC '09: Proceedings of the ACM/IEEE Conference on Supercomputing 2009*, pages 16:1–16:12, 2009.

[22] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. In J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors, *Sourcebook of parallel computing*, chapter 18, pages 491–541. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Nov 2002.

[23] L. Xu and H.-W. Shen. Flow web: a graph based user interface for 3D flow field exploration. In *Proceedings of the IS&T/SPIE Visualization and Data 2010*, Jan. 2010.

[24] H. Yu, C. Wang, and K. L. Ma. Parallel hierarchical visualization of large time-varying 3D vector fields. In *SC 07: Proceedings of the ACM/IEEE Conference on Supercomputing 2007'*, 2007.