

# Enabling High Performance Computational Science through Combinatorial Algorithms

Erik G. Boman<sup>1</sup>, Doruk Bozdag<sup>2</sup>, Umit V. Catalyurek<sup>2</sup>,  
Karen D. Devine<sup>1</sup>, Assefaw H. Gebremedhin<sup>3</sup>, Paul D. Hovland<sup>4</sup>,  
Alex Pothen<sup>3</sup>, and Michelle Mills Strout<sup>5</sup>

<sup>1</sup> Discrete Algorithms and Math Department, Sandia National Laboratories

<sup>2</sup> Biomedical Informatics, and Electrical and Computer Engineering, Ohio State University

<sup>3</sup> Computer Science and Center for Computational Science, Old Dominion University

<sup>4</sup> Mathematics and Computer Science Division, Argonne National Laboratory

<sup>5</sup> Computer Science, Colorado State University

E-mail: pothen@cs.odu.edu

**Abstract.** The Combinatorial Scientific Computing and Petascale Simulations (CSCAPES) Institute is developing algorithms and software for combinatorial problems that play an enabling role in scientific and engineering computations. Discrete algorithms will be increasingly critical for achieving high performance for irregular problems on petascale architectures. This paper describes recent contributions by researchers at the CSCAPES Institute in the areas of load balancing, parallel graph coloring, performance improvement, and parallel automatic differentiation.

## 1. Introduction

We describe recent research at the CSCAPES (pronounced “seascapes”) Institute, which was funded by the Department of Energy in 2006 in its SciDAC program to harness tera-scale and petascale performance for scientific simulations critical to DOE’s mission. The Institute will develop and deploy fundamental enabling technologies in high performance computing that employ algorithms from combinatorial or discrete mathematics. Combinatorial scientific computing (CSC) is the name for the broader inter-disciplinary field in which researchers identify combinatorial subproblems that arise in computational science and engineering, design graph and hypergraph algorithms to solve these subproblems, and create high performance software implementing the algorithms. CSC plays a crucial enabling role in applications requiring parallelization, differential equations, optimization, eigenvalue computations, management of large-scale data sets, etc. In this paper we consider four problems in CSC.

## 2. Partitioning and Load-balancing

Large-scale scientific simulations are typically run on high performance parallel computers with thousands of processors. Even desktop computers now have multiple cores, and petascale machines will consist of thousands of multi-core processors. An important task is then to distribute data and work of a large-scale computation among the processors to minimize total execution time. This problem is known as “load balancing” or “partitioning”. We assume that the “owner computes” strategy is used, which is common in large-scale computational science problems, since the data

sets are huge, and communication costs are relatively larger than computation costs on current distributed memory architectures. In this scheme, each data item is mapped to a unique processor (the owner), and the owner performs the computations that are associated with that data item. Most data items are mapped only to the owner, but items on the boundaries of a partitioned data structure could be mapped to more than one processor. Communication is required when a computation needs access to data items that reside on different processors.

There are two, often conflicting, goals in load balancing. First, the work that can be performed concurrently should be evenly distributed among the processors to avoid processors that finish early from having to wait for the slowest processor to finish its task. Second, communication between processors is relatively slow compared to computation, and so it should be as small as possible. The communication requirements are dictated by the data dependencies in the application at hand. These goals are conflicting since the first drives the data to be distributed among all the processors, while communication costs are lowest if the data resides on one processor.

The load balancing problem is pervasive in parallel scientific computing and important to many different types of applications, i.e., particle simulations (molecular dynamics, chemistry, biology), mesh-based PDE solvers (structural mechanics, chemical engineering, groundwater flow), and circuit simulation.

### *2.1. Static Partitioning*

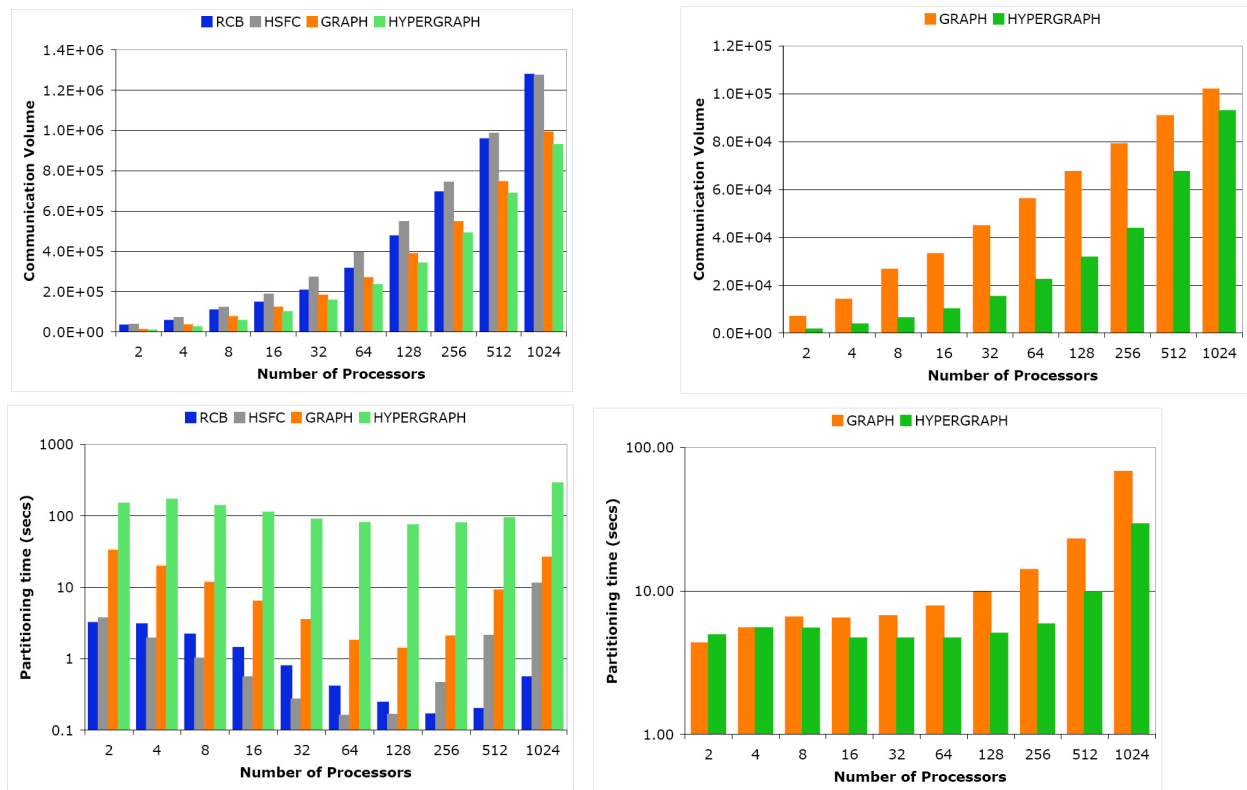
For scientific simulations where the computational tasks and the data dependences do not change much during the simulation, it is sufficient to partition the data once before the computation starts. We call this static partitioning.

Most partitioning algorithms fall into one of two categories: geometric or connectivity-based. Geometric methods are often preferred in some applications such as particle simulations, but can be used for any application where geometry is available. One popular geometric method is recursive coordinate bisection (RCB), where the domain is recursively subdivided into two halves by a cutting plane aligned with a coordinate axis. Another method is space-filling curve based partitioning, where the geometric domain is filled by a space-filling curve (Hilbert, Morton) and points are assigned to partitions according to their position along this curve.

Geometric partitions are generally fast to compute, but may result in high communication volumes. Graph partitioners, on the other hand, use connectivity information and often have low communication volume, but are more expensive to compute. Hypergraph partitioning is a generalization of graph partitioning that more accurately reflects the communication volume.

With these trade-offs, no single method is best in all cases. CSCAPES researchers have developed the Zoltan software toolkit, which contains a collection of partitioning (load balancing) methods. In this paper, we compare four algorithms from Zoltan on data from real applications. We present results from partitioning two such data sets: a mesh of an accelerator part from the Stanford Linear Accelerator Center (SLAC) with six million elements, and a circuit model from Sandia with 680 thousand matrix rows. We did not have any geometry for the circuit model, so could not use geometric methods. Our test platform was a large Dell cluster at Sandia (Thunderbird) with 8960 3.6 GHz processors and Infiniband interconnect.

We see from Figure 1 that as expected, the geometric methods (RCB and Hilbert Space-Filling Curve [HSFC]) are faster to compute but give relatively large communication volume. The partitioning itself was done in parallel, but the (weak) scalability results are not simple to interpret since the partitioning problem changes with number of processors. While hypergraph partitioning yields only slightly lower communication volume than graph partitioning on the mesh problem, it is substantially better for the more irregular circuit problem.



**Figure 1.** Static partitioning results using Zoltan on a six million element mesh from SLAC (left) and a circuit matrix from Sandia (right). The top two subfigures show the communication volume, while the bottom two subfigures show partitioning time.

## 2.2. Repartitioning for Dynamic Load Balancing

In parallel adaptive computations, even if the original problem is well balanced, e.g., by the use of graph or hypergraph partitioning, the computation may become unbalanced over time due to the dynamic changes. A classic example is simulation based on adaptive mesh refinement, in which the computational mesh changes between time steps. The difference is often small, but over time, the cumulative change in the mesh becomes significant. An application may therefore periodically re-balance, that is, move data among processors to improve the load balance. This process is known as dynamic load balancing or repartitioning. It has multiple objectives with complicated trade-offs among them:

- (i) good load balance in the new data distribution;
- (ii) low communication cost within the application (as given by the new distribution);
- (iii) low data migration cost to move data from the old to the new distribution; and
- (iv) short repartitioning time.

Most geometric partitioners are naturally incremental, that is, a small perturbation in the data gives only a small change in the partitions. This property ensures low data migration cost.

For graph and hypergraph methods the picture is more complex. Much of the early work in load balancing focused on diffusive methods [3], where overloaded processors give work to neighboring processors that have lower than average loads. A quite different approach is to partition the new problem “from scratch” without accounting for existing partition assignments, and then try to remap partitions to minimize the migration cost. These two strategies have very

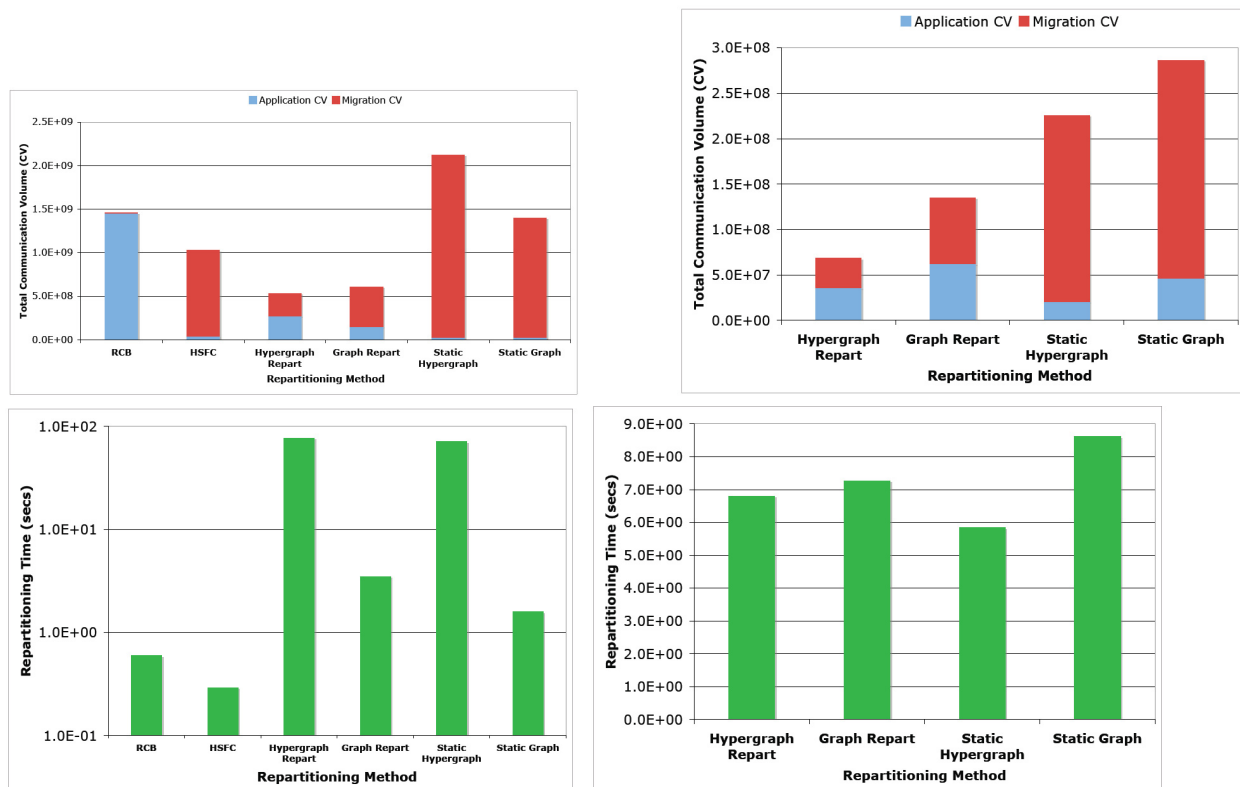
different properties. Diffusive schemes are fast and have low migration cost, but may incur high communication volume. Scratch-remap schemes give low communication volume but are slower and often have high migration cost.

CSCAPES researchers have recently developed a new hypergraph-based repartitioning method [2]. The approach is to directly minimize the total execution time. The following model, from [16], is used:

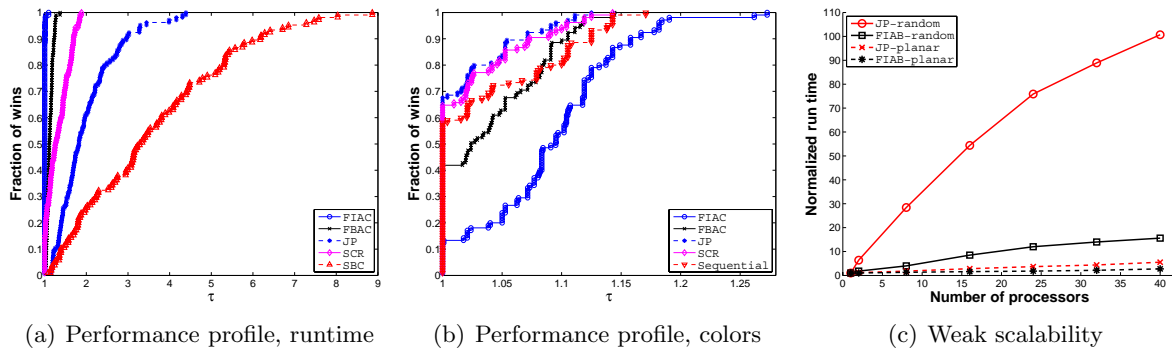
$$t_{tot} = \alpha(t_{comp} + t_{comm}) + t_{mig} + t_{repart},$$

where  $t_{comp}$  and  $t_{comm}$  denote computation and communication times within one iteration of the application, respectively,  $t_{mig}$  is the data migration time, and  $t_{repart}$  is the repartitioning time. The parameter  $\alpha$  indicates how many iterations (e.g., time steps in a simulation) the application performs between every load-balance operation. Since the goal of load balancing is to minimize the communication cost while maintaining well-balanced computational loads, we can safely assume that computation will be balanced and hence drop  $t_{comp}$  term. The repartitioning time is typically significantly smaller than  $\alpha(t_{comp} + t_{comm})$  due to fast state-of-the-art repartitioners, so we also ignore  $t_{repart}$ . Thus, the objective of our model is to minimize  $\alpha t_{comm} + t_{mig}$ .

The approach in [2] is to create a new hypergraph that exactly accounts for both the communication volume and the migration cost to move data. This is done by adding a new vertex to represent each partition and new (weighted) edges to model data migration cost. The augmented hypergraph can then be partitioned using existing algorithms and software.



**Figure 2.** Repartitioning results using Zoltan on 64 processors on a six million element mesh from SLAC (left) and a circuit matrix from Sandia (right). The top two subfigures show the total communication volume (application volume plus redistribution volume), while the bottom two subfigures show repartitioning time.



**Figure 3.** Subfigures (a) and (b) show performance profile plots, where the respective metrics are runtime and number of colors, of two variants of the parallelization framework, FIAC and FBAC, and three other, related algorithms, JP, SCR and SBC. The vertical axis in both (a) and (b) shows the fraction of the test cases for which an algorithm gave a solution within a factor  $\tau$  of the best solution rendered by any algorithm in the set of algorithms being compared. For example, the value on the vertical axis for  $\tau = 1$  for a specific algorithm  $A$  shows the fraction of the test-cases for which algorithm  $A$  gave the best result. Subfigure (c) shows *weak* scalability of the FIAB variant of the framework and the JP algorithm on random and planar graphs. An ideal plot here corresponds to a horizontal line.

Figure 2 shows results for the same two test problems as in the static case, but with one difference. To simulate an adaptive computation, the vertex weights were randomly increased in some parts of the graph. We assumed the application would run 100 iterations between each load balance. The top row shows communication volume, for which the blue part denotes application communication and the red part denotes migration cost. We see that RCB has the lowest migration cost but highest application communication volume. Overall, our new hypergraph repartitioner produced the lowest communication volume. The results show the time required by one partitioning step of the static methods, and the average time for each repartitioning step. It can be seen that the additional time required by the repartitioning methods over static partitioning is modest.

### 3. Parallel Graph Coloring and Applications

In parallel scientific computing applications, computational dependency is often modeled using a graph. A vertex coloring of the graph—an assignment of positive integers to vertices such that the values assigned to a pair of adjacent vertices are different—is then used as a routine to identify independent subtasks that can be performed concurrently. The number of colors used corresponds to the computational steps required and is therefore desired to be as small as possible. Examples where coloring is used for such a purpose include iterative solution of sparse linear systems [13], preconditioning [14], and sparse tiling [17]. The computational (model) graph in such contexts is often already distributed among processors and therefore the coloring needs to be performed in parallel. For large-scale problems, the alternative approach of gathering the graph on one processor for a subsequent sequential coloring (by the same processor) could be prohibitively time consuming or even infeasible due to memory constraints.

Theoretical results on graph coloring are highly pessimistic: even an approximate solution is known to be NP-hard to find. For the computational graphs mentioned earlier as well as many other graphs that arise in practice, however, *greedy*, linear-time, serial coloring heuristics give solutions of acceptable quality and are often preferable to slower, iterative heuristics that may use fewer colors. However, greedy coloring heuristics are inherently sequential, and thus their parallelization is a difficult endeavor.

Building upon experiences from a series of earlier efforts, CSCAPES researchers have recently

developed an efficient framework for parallelizing greedy coloring algorithms [1]. The basic features of the framework could be summarized as follows. Given a graph *partitioned* among the processors of a distributed-memory machine, each processor *speculatively* colors the vertices assigned to it in a series of rounds. Each round consists of a tentative coloring and a conflict detection phase. The coloring phase in a round is further broken down into *supersteps*. In each superstep, a processor first colors a pre-specified number  $s \gg 1$  of its assigned vertices sequentially, using color information available at the beginning of the superstep, and only thereafter exchanges recent color information with other, relevant processors. The rationale behind the infrequent, *coarse-grained* communication is the reduction of the associated communication cost. In the conflict-detection phase, each processor examines those of its vertices that are colored in the current round for consistency and identifies a set of vertices that needs to be recolored in the next round to resolve any detected conflicts. In the selection of vertices to be recolored, a *randomized* technique is employed in order to achieve a balanced workload distribution across processors. In particular, given a conflict-edge—an edge whose endpoints have received the same color—the endpoint to be recolored is chosen using a random function. The scheme terminates when no more conflicts remain to be resolved.

Bozdag et al [1] have implemented (using the message-passing library MPI) several variant algorithms derived from this framework and incorporated the codes into the Zoltan load-balancing and parallelization toolkit. The various implementations were experimentally analyzed so as to determine the best way in which various parameters of the framework need to be combined in order to reduce both runtime and number of colors. With this objective in mind, the authors attempt to answer the following questions. How large should the superstep size  $s$  be? Should the supersteps be run synchronously or asynchronously? Should interior vertices be colored before, after, or interleaved with boundary vertices? (A vertex is *interior* with respect to a given partition if all of its neighbors reside on the same processor as itself; it is *boundary* otherwise.) How should a processor choose a color for a vertex? Should inter-processor communication be customized or broadcast-based?

Experiments were carried out on two different platforms using large-size synthetic as well as real graphs drawn from various application areas. The computational results obtained suggest that, for large-size, *structured* graphs (i.e., graphs with partitions that have a small fraction of boundary vertices), a combination of parameters in which

- (i) a superstep size in the order of a thousand is used,
- (ii) supersteps are run asynchronously,
- (iii) each processor colors its assigned vertices in an order where interior vertices appear either strictly before or strictly after boundary vertices,
- (iv) a processor chooses a color for a vertex using a first-fit scheme (where the *smallest* available color is chosen at each coloring step), and
- (v) inter-processor communication is customized,

gives overall the best performance. Furthermore, the choice of the coloring order in (iii) offers a trade-off between number of colors and execution time: coloring interior vertices first gives a faster and slightly more scalable algorithm whereas an algorithm in which boundary vertices are colored first uses fewer colors. The number of colors used even when interior vertices are colored first is fairly close to the number used by a sequential greedy algorithm. For *unstructured* graphs (i.e., graphs where the best partitions have a majority of the vertices as boundary vertices), good performance is observed by using a superstep size close to a hundred in item (i), a broadcast based communication mode in item (v), and by keeping the remaining parameters as in the structured case. For most of the test graphs used in the experiments, the variants of the framework with the aforementioned combination of parameters converged rapidly (within at most six rounds) and yielded fairly good speedup with increasing number of processors.

The authors of [1] also compared the performance of the framework against their implementation of several other related parallel coloring algorithms, including the algorithm due to Jones and Plassmann (JP) [12] and the related variations of the framework in which conflicts after the first round are resolved sequentially (SCR) or boundary vertices are colored sequentially to begin with (SBC). Figure 3(a) and Figure 3(b) show *performance profile* plots of two of the best variants of the framework (FIAC and FBAC) and the algorithms JP, SCR, and SBC. See [5] for a discussion of performance profiles and their use as a generic tool for comparing a set of methods over a large set of test cases. In the plots in Figure 3, 21 test graphs drawn from 7 different application areas were used, and a “test instance” comprised of a tuple (graph, number-of-processors)  $(G, p)$ , where a value for  $p$  is drawn from the set  $\{8, 16, 24, 32, 40\}$ . The only difference between the FIAC and FBAC specializations of the framework is that in FIAC, interior vertices are colored before boundary vertices, whereas in FBAC, boundary vertices are colored before interior vertices. The rest of the letters in the acronym (F, A, and C) stand for *first-fit* coloring scheme, *asynchronous* supersteps, and *customized* inter-processor communication. The plots clearly show that overall FIAC and FBAC compare favorably against JP, SCR and SBC, and they in turn offer a trade-off between execution speed and quality of solution (number of colors). Note also that there is only marginal difference in quality among the various parallel algorithms and between the sequential and the parallel versions: the  $\tau$  values for more than 90% of the test cases range between 1 and 1.15, indicating that the worst algorithm uses only 15% more colors than the best algorithm.

Figure 3(c) shows weak-scalability results on the FIAB variant of the framework and the JP algorithm using two families of synthetic graphs, random and planar. These graphs are chosen for being representatives of two extreme ends: highly unstructured graphs with poor partition quality and highly structured graphs with high partition quality. More specifically, in each random graph almost every vertex in a partition produced by the graph partitioning tool Metis is found to be boundary, whereas in a similar situation in a planar graph, nearly every vertex is interior. As the last letter in FIAB indicates, a broadcast-based communication, as opposed to customized communication, is used since it gives better results for the unstructured graphs. In Figure 3(c) the size of a graph is increased in proportion to the number of processors, so an ideal plot is a horizontal line (slope 0). The figure shows that the performance of FIAB is close to ideal for both the random and the planar graphs, whereas the JP algorithm performs poorly especially on random graphs.

Future plans in CSCAPES include extending the parallelization framework to tera- and peta-scale machines and to other (specialized) coloring problems in the computation of sparse Jacobian and Hessian matrices using automatic differentiation; see [6, 7] for our work on sequential algorithms for such problems. We believe that the basic ingredients of this parallelization framework—*exploiting features of an initial data distribution, maximizing concurrency by tentatively tolerating inconsistencies and then detecting and resolving them, randomization, and coarse-grain communication*—are likely to be useful in the design of parallel algorithms for other graph problems such as matching. We plan to pursue this line in our future work within CSCAPES.

#### 4. Performance Improvement

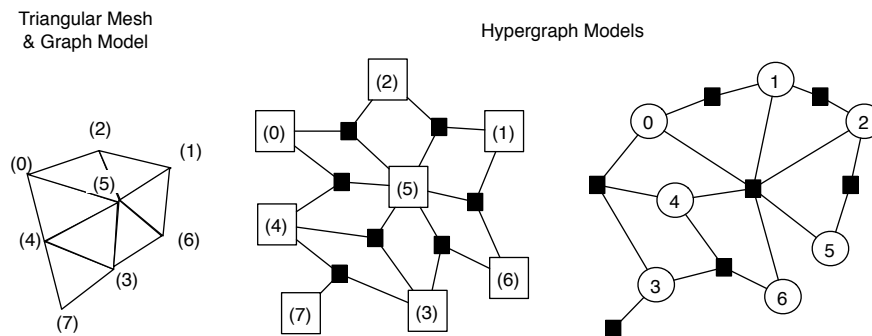
Achieving high performance on petascale architectures requires high single processor performance as well as obtaining good performance in parallel. A widely quoted estimate from researchers in DOE labs is that irregular scientific computing applications achieve less than 10% of peak performance on current microprocessors. This gap in performance between dense (and regular) computations and sparse (and irregular) computations has been called the “sparse matrix gap”. The sparse matrix gap can be attributed primarily to irregular memory references. The code segment in Fig. 4 contains a loop traversing triangular elements in a mesh. The indirect memory references such as `data[ n1[i] ]` can exhibit poor data locality and therefore cause performance problems. In the context of iterating over a triangular mesh, a data reordering involves reordering the entries in the `data` array and doing a corresponding update (called a pointer update [4]) to the values in the

```

// iterate over triangles
for (int i = 0; i < numTri; i++)
{
    // access data for each vertex
    ... data[ n1[i] ] ...
    ... data[ n2[i] ] ...
    ... data[ n3[i] ] ...
}

```

**Figure 4.** Loop iterating over triangular elements.



**Figure 5.** Example triangular mesh. The middle subfigure shows a hypergraph where vertices correspond to nodes and hyperedges (the small black squares) represent triangle elements in the original mesh. The subfigure on the right is the dual hypergraph, where vertices correspond to triangle elements and hyperedges (the small black squares) represent nodes in the original mesh.

index arrays  $\mathbf{n1}$ ,  $\mathbf{n2}$  and  $\mathbf{n3}$ . An iteration reordering involves reordering the values in the  $\mathbf{n1}$ ,  $\mathbf{n2}$  and  $\mathbf{n3}$  arrays, and logically corresponds to reordering visits to triangles in the mesh.

Traditionally, reordering for data locality employs a graph model [9], where the nodes represents data items and there are edges between nodes whenever two data items are accessed within the same iteration of a loop. The subfigure on the left in Figure 5 shows an example triangular mesh. The graph model for this computation is the mesh itself.

In the CSCAPES project, we model the relationship between data and computation with hypergraph models, which include multiple nodes within each hyperedge. The middle subfigure in Figure 5 shows the hypergraph that models relationships between the nodes in the mesh. The square vertices with parenthesized numbers directly correspond to nodes in the mesh. All vertices adjacent to the same filled-in square are part of the same hyperedge, indicating that the corresponding nodes in the mesh are accessed together within one iteration of the loop. One advantage of the hypergraph model over the graph model is that the relationships between nodes/data are not broken down into pair-wise relationships, which causes some information to be lost. Another advantage of the hypergraph model is that there is a dual hypergraph model (the rightmost subfigure in Figure 5) where the nodes in the dual hypergraph correspond to iterations in the loop, and the hyperedges correspond to data being accessed. This enables the use of hypergraph reordering heuristics for the dual hypergraph as well, providing triangle/iteration reorderings.

We experimented with various graph and hypergraph-based reorderings on a mesh optimization application, FeasNewt, developed by Dr. Todd Munson at Argonne National Laboratory. Good reorderings of the mesh data structures enabled FeasNewt to run twice as fast relative to an ordering provided by the mesh generator. The goal of our future work is to select the optimal

```

k = processor_id
localprod = x_k

nr = lg(P)
% Up phase
for i = 0 ... nr-1
    if (k && (2^(i+1) - 1)) == (2^(i+1) - 1)

        receive prtlprod from processor k - 2^i
        localprod *= prtlprod
    else
        send localprod to processor k + 2^i

        goto downphase
    endif
endfor
% Down phase
for i = nr-1 ... 1
    if (k && (2^i - 1)) == (2^i - 1)
        send localprod to k + 2^(i-1)

    else if (k && (2^(i-1) - 1)) == (2^(i-1) - 1)
        receive prtlprod
        localprod *= prtlprod
    endif
endfor

k = processor_id
prod = x_k
deriv = 1
nr = lg(P)
% Up phase
for i = 0 ... nr-1
    if k && (2^(i+1) - 1) == (2^(i+1) - 1)
        send prod to processor k - 2^i
        receive nbrval[i] from processor k - 2^i
        prod *= nbrval[i]
    else
        send prod to processor k + 2^i
        receive nbrval[i] from processor k + 2^i
        goto downphase
    endif
endfor
% Down phase
for i = nr-1 ... 0
    if (k && (2^(i+1) - 1)) == (2^(i+1) - 1)
        send deriv to processor k - 2^i
        deriv *= nbrval[i]
    else if (k && (2^i - 1)) == (2^i - 1)
        receive deriv from processor k + 2^i
        deriv *= nbrval[i]
    endif
endfor

```

**Figure 6.** Binomial tree-based parallel prefix computation (left) and modified algorithm for computing partial derivatives of the product reduction (right).

reordering strategy based on the mesh and machine characteristics.

## 5. Differentiation of Parallel Reduction Operations

Automatic differentiation (AD) is a means of developing code to compute the derivatives of complicated functions accurately and efficiently, without the difficulties associated with developing correct code by hand or using a computer algebra system [8]. The CSCAPES Institute will work on several combinatorial problems that arise efficient AD computations.

In many cases, AD can be applied to local computations without regard to parallel constructs [10]. However, in order to apply AD to parallel programs in a completely black box fashion [11], the derivatives of parallel reduction operations, including PRODUCT, SUM, MAX, and MIN need to be computed. We describe here an efficient algorithm for the first of these.

In [11], we proposed implementing the partial derivatives of  $y = \prod_{i=1}^P x_i$  using a pair of calls to MPI.Scan (parallel prefix computation), computing

$$P_k = \prod_{i=1}^k x_i \quad \text{and} \quad S_k = \prod_{j=k}^P x_j$$

on processor  $k$ . Then

$$\frac{\partial y}{\partial x_k} = P_{k-1} S_{k+1}.$$

A disadvantage of this algorithm is the requirement for  $4 \log_2 P$  communication phases if the parallel prefix computation is implemented using a conventional binomial tree algorithm. We propose a new algorithm, based on a modified parallel prefix algorithm, which reduces the

number of communication phases to  $2 \log_2 P$ . Figure 6 shows the basic parallel prefix binomial tree algorithm [15] and the modified algorithm for product derivatives. Since communication is expensive relative to computation on parallel machines, and this situation will be exacerbated on petascale machines, the halving of the communication costs in the product derivative algorithm is a significant improvement. We will implement this algorithm in future work.

### Acknowledgments

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U. S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. The work at Argonne was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract W-31-109-Eng-38. The CSCAPES Institute is supported by the U.S. Department of Energy's Office of Science through grant DE-FC-0206-ER-25774, as part of its SciDAC program.

### References

- [1] Bozdag D, Gebremedhin A, Manne F, Boman E and Catalyurek U A framework for scalable greedy coloring on distributed memory parallel computers. *Journal of Parallel and Distributed Computing*, 2007. Submitted.
- [2] Catalyurek U, Boman E, Devine K, Bozdag D, Heaphy R and Riesen L Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21th International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.
- [3] Cybenko G Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, **7** 279-301, 1989.
- [4] Ding C and Kennedy K Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 229-241, May 1-4, 1999.
- [5] Dolan E and Moré J Benchmarking optimization software with performance profiles. *Math. Prog.*, **91** 201-213, 2002.
- [6] Gebremedhin A, Manne F, and Pothen A What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, **47** 629-705, 2005.
- [7] Gebremedhin A, Tarafdar A, Manne F, and Pothen A New acyclic and star coloring algorithms with application to computing Hessians. *SIAM Journal on Scientific Computing*, **29** 1042-1072, 2007.
- [8] Griewank A *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.
- [9] Han H and Tseng C A comparison of locality transformations for irregular codes. In *Proceedings of the 5th International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, volume 1915 of *Lecture Notes in Computer Science*. Springer, 2000.
- [10] Hovland P, Norris B and Smith B Making automatic differentiation truly automatic: Coupling PETSc with ADIC. *Future Generation Computer Systems*, **21** 1426-1438, 2005.
- [11] Hovland P and Bischof C Automatic differentiation of message-passing parallel programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, Los Alamitos, CA, 1998. IEEE Computer Society Press.
- [12] Jones M and Plassmann P A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, **14** 654-669, 1993.
- [13] Jones M and Plassmann P Scalable iterative solution of sparse linear systems. *Parallel Computing*, **20** 753-773, 1994.
- [14] Saad Y ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, **17** 830-847, 1996.
- [15] Sanders P and Träff J-L Parallel prefix (scan) algorithms for MPI. In Mohr B, Träff J-L, Worringer J and Dongarra J, editors, *PVM/MPI*, volume 4192 of *Lecture Notes in Computer Science*, pages 49-57. Springer, 2006.
- [16] Schloegel K, Karypis G, and Kumar V A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing*, Dallas, 2000.
- [17] Strout M. M, Carter L, Ferrante J, Freeman J and Kreaseck B Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing*, College Park MD, 2002.