Scientific Programming 18 (2010) 107–123
DOI 10.3233/SPR-2010-0301
IOS Press

Adjacency-based data reordering algorithm for acceleration of finite element computations

Min Zhou [a,*], Onkar Sahni [a], Mark S. Shephard [a], Christopher D. Carothers [b] and Kenneth E. Jansen [c]
[a] *SCOREC, Department of Mechanical, Aerospace, and Nuclear Engineering, Rensselaer Polytechnic Institute, Troy, New York, NY, USA*
[b] *Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York, NY, USA*
[c] *Aerospace Engineering Sciences, University of Colorado at Boulder, Boulder, CO, USA*

Abstract. Effective use of the processor memory hierarchy is an important issue in high performance computing. In this work, a part level mesh topological traversal algorithm is used to define a reordering of both mesh vertices and regions that increases the spatial locality of data and improves overall cache utilization during on processor finite element calculations. Examples based on adaptively created unstructured meshes are considered to demonstrate the effectiveness of the procedure in cases where the load per processing core is varied but balanced (e.g., elements are equally distributed across cores for a given partition). In one example, the effect of the current ajacency-based data reordering is studied for different phases of an implicit analysis including element-data blocking, element-level computations, sparse-matrix filling and equation solution. These results are compared to a case where reordering is applied to mesh vertices only. The computations are performed on various supercomputers including IBM Blue Gene (BG/L and BG/P), Cray XT (XT3 and XT5) and Sun Constellation Cluster. It is observed that reordering improves the per-core performance by up to 24% on Blue Gene/L and up to 40% on Cray XT5. The CrayPat hardware performance tool is used to measure the number of cache misses across each level of the memory hierarchy. It is determined that the measured decrease in L1, L2 and L3 cache misses when data reordering is used, closely accounts for the observed decrease in the overall execution time.

Keywords: Data reordering, cache penalty model, unstructured mesh, finite element analysis

1. Introduction

The finite element method is a standard analysis tool for solving complex sets of partial differential equations (PDEs) over general domains. In a large number of cases, the ability to solve problems of practical interest requires the use of many millions of spatial degrees of freedom and many thousands of implicit time steps. Problems of such size can only be solved on massively parallel computers but any performance improvements in on-processor calculations are desirable, particularly in situations with tight constraints on time to solution. One such example is patient-specific vascular surgical planning, where the solution times must be in minutes for problems with up to 100 million degrees of freedom in total. This paper presents an adjacency-based data reordering algorithm, which reorders both mesh vertices and regions to increase cache coherency during two primary stages of an implicit, finite element analysis, i.e., element formation and equation solution. The effective use of the memory hierarchy on processing cores in turn leads to improved flop rate (or floating point operations per second).

In calculations based on unstructured meshes, the irregular mesh structures and connections make the data storage order a strong contributor to the effective use of the memory hierarchy. Since the storage order of the data is defined by the labeling of the entities in the mesh, it is desirable to have the mesh entities labeled such that those that interact, or have sequential access during calculations, have labels that are as close to each other as possible. The historic way to address this issue is to employ the methods used in "bandwidth" and

*Corresponding author: Min Zhou, SCOREC, Department of Mechanical, Aerospace, and Nuclear Engineering, Rensselaer Polytechnic Institute, 110 8th St., Troy, New York, NY 12180, USA. E-mail: zhoum@scorec.rpi.edu.

"wavefront" minimization such as RCM [6], GPS [8, 9,5] and others. In previous work on edge-based unstructured mesh computations [4,16,17], the nodes are reordered using a bandwidth-minimization technique (RCM, recursive bisection, etc.) for optimal cache performance before subsequently renumbering the edges according to the nodes on each edge [16] or applying nodal reordering according to maximum connectivity to reduce indirect memory access [4,17] specifically for edge-based methods. Oliker et al. [18–20] compared effects of data reordering strategies, RCM and self-avoiding walk (SAW) developed in [11] for two-dimensional unstructured meshes. In their tests on smaller cache architectures, SAW is about twice as fast as RCM, while there is little difference in parallel performance between RCM and SAW on larger cache architectures. It is worth noting that SAW is not a technique designed specifically for vertex reordering and incurs a higher construction cost [20]. Another alternative approach is the permutation method of rows (and columns) of a matrix, which is presented in [21] to increase cache reuse in sparse-matrix and dense-vector product. This is the kernel of equation solution of our finite element analysis. In a more recent work, Yzelman et al. [33] introduced a data reordering algorithm for sparse matrix–vector multiplication by permuting rows and columns of the input matrix using a hypergraph-based matrix partitioning scheme. However, those algorithms do not help the other primary stage of our finite element analysis (element formation) and the overhead of the algorithm is generally high. Williams et al. [31] employed sparse cache blocking to specifically optimize sparse matrix–vector products. Data reordering algorithms for general unstructured meshes are described in [3]. In the first step they apply existing classic approaches (such as bandwidth reduction, the greedy method, etc.) to either vertices, edges, faces or regions. In the second step, they consistently renumber the other entities used in the solver. Mappings, such as a list of connected vertex pairs needs to be constructed to use the existing classic approaches in the first step, which can be expensive (see Section 3 for detailed discussions). Han et al. [10] introduced a data reordering (GPART) based on the graph structure within data access and Strout et al. [27] developed algorithm to increase spacial locality based on the hypergraph within data access, however both of them have to pay the overhead of constructing the graph/hypergraph at run time.

The examples considered in this paper are based on linear C^0 finite elements. Here, the only mesh entity labeling required is of the mesh regions and mesh vertices. The mesh vertices are the holders of the degrees of freedom that play the primary role in the equation solution stage. Thus, it is desirable to label vertices with coupled degrees of freedom such that they are close to each other. On the other hand, during element integration (equation formation) the ordering of both mesh regions and vertices is important since the process traverses the regions and accesses information related to the vertices of those regions. Note that in parallel finite element analysis, the ordering of mesh entities is required individually on parts making it easy to define. It is worth mentioning that the automatic unstructured mesh generators typically do not concern themselves with an ordering of entities since they typically do not know the details of the analysis algorithms. Furthermore, even if ordering was performed on the initial mesh, it would be disrupted by mesh adaptation procedures in their attempt to incrementally add and delete entities to achieve high local resolution meshes.

By taking advantage of the fact that a mesh topology is maintained during the pre-processing of the finite element analysis input, an algorithm that effectively reorders the mesh regions and vertices based on topological adjacency has been implemented. The mesh topology concept is discussed in Section 2. The reordering algorithm is described in Section 3 and results demonstrating its ability to improve flop rate are presented in Section 4.

2. Mesh representation and topological adjacency

The process of reordering a mesh is greatly aided by databases and functions that can answer basic queries about the topological adjacencies of a given mesh. In the studies described herein, we have made use of the Flexible distributed Mesh DataBase (FMDB) which is a distributed mesh data management system that is capable of shaping its data structure dynamically based on the user-requested mesh representation [25]. Before describing the algorithms it is useful to introduce some nomenclature.

2.1. Nomenclature

M	an abstract model of the mesh.
$\{M\{M^d\}\}$	a set of topological entities of dimension d in model M.
M_i^d	the ith entity of dimension d in model M. $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face and $d = 3$ for a region.

$\{M_i^d\{M^q\}\}$	a set of entities of dimension q in model M that are adjacent to M_i^d.
$M_i^d\{M^q\}_j$	the jth entity in the set of entities of dimension q in model M that are adjacent to M_i^d.
$\{M_i^d\{D\}\}$	a set of nodes (degrees of freedom) associated with mesh entity M_i^d.
$M_i^d\{D\}_j$	jth node (degree of freedom) associated with mesh entity M_i^d.
N_i	the number of entities of dimension i in the mesh. Shorthand for $\|\{M\{M^i\}\}\|$. N_0, N_1, N_2 and N_3 are the number of vertices, edges, faces and regions, respectively, in the mesh.
Examples	
$\{M\{M^2\}\}$	the set of all the faces in the mesh.
$\{M_3^1\{M^3\}\}$	the mesh regions adjacent to mesh edge M_3^1.
$M_1^3\{M^1\}_2$	the 2nd edge adjacent to mesh region M_1^3.

2.2. Adjacency and distributed mesh representation

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d, adjacency, denoted by $\{M_i^d\{M^q\}\}$, returns all the mesh entities of dimension q, which are on the closure of the entity for a downward adjacency ($d > q$), or for which the entity is part of the closure for an upward adjacency ($d < q$).

There are many options in the design of the mesh data structure in terms of the entities and adjacencies stored. If a mesh representation stores all 0 to d level entities explicitly, it is a *full* representation, otherwise, it is referred to as a *reduced* representation. *Completeness of adjacency* indicates the ability of a mesh representation to provide any type of adjacencies requested without involving an operation dependent on the mesh size such as the global mesh search or mesh traversal. Regardless of whether the representation is full or reduced, if all adjacency information is obtainable in O(1) time, the representation is complete, otherwise, it is incomplete.

We assume full and complete mesh representations throughout this paper. Implementations with reduced complete representations using all the same overall algorithms are possible, with the addition of some complexities within the mesh database API functions [25].

A distributed mesh is a mesh divided into parts for distribution over a set of processors for specific reasons, for example, parallel computation. Each part is treated as a mesh with entities on that part and the addition of part boundaries to describe groups of mesh entities that are on inter-part boundaries. Mesh entities on inter-part boundaries are duplicated on parts where they are used in adjacency relations. Mesh entities that are not on any inter-part boundary exist on a single part. Figure 1 depicts a mesh that is distributed on 3 parts. Vertex M_1^0 is common to three parts and exists on each part, several mesh edges like M_j^1 are common to two parts. The dashed lines represent inter-part boundaries that consist of mesh vertices and edges (including mesh faces in 3D) duplicated on multiple parts.

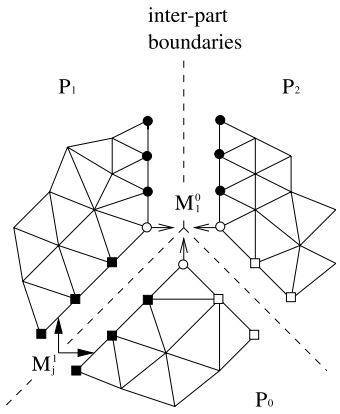


Fig. 1. Distributed mesh on three parts P_0, P_1 and P_2 [22].

The reordering algorithm described in Section 3 is developed on the basis of individual parts and by taking advantage of the complete adjacencies.

3. Algorithm of data reordering

As discussed in references [3,4,16–21], improving data locality in unstructured mesh calculations can have a substantial influence on the computational performance of the numerical operations performed. One can increase the data locality by accounting for the locality of the entities in the mesh, and the non-zero entries in the resulting finite element matrices. Since the ordering of the operations and information in the finite element matrices are derived from the labeling given to the mesh entities, one wants to define a labeling such that mesh entities considered to be near via

adjacency, and related to each other, are closely numbered.

The relationships between mesh entities dictate where terms are placed in sparse matrices and the order in which they are accessed during numerical operations. In the context of finite element analysis such information is easily identified based on the mesh adjacencies. For example, in the case of linear finite elements, the degrees of freedom are associated with vertices. The non-zero entities in the stiffness matrix for the equation associated with a given degree of freedom or a vertex are due to all of the vertices that are associated with elements that bound (sharing) the given vertex (i.e., $\{\{M_i^0\{M^3\}\}\{M^0\}\}$). Figure 2 shows an example, where a vertex, 29, has five elements around it and all the other vertices that bound these five elements are 1, 2, 12, 30 and 50. The entities in the stiffness matrix due to vertex 29, are shown in the matrix A. Matrix–vector products with this matrix will enjoy sequential access of the matrix entries (block compressed sparse row scheme is employed, which is naturally in row order), but the second half of each product pair will require entries from the product vector associated with these non-zeros as shown in the vector P. Note that the labels of mesh entities determine the entries in the vector that will be accessed one after another and thus, when the mesh entities labeled without consideration of order, the entries accessed subsequently may be placed far away in the vector.

$$A = \begin{pmatrix} & & & & \\ A_{29,1} & A_{29,2} & \cdots & A_{29,12} & \cdots \\ & & & & \end{pmatrix}$$

$$\begin{pmatrix} A_{1,29} & & & \\ A_{2,29} & & & \\ \vdots & & & \\ A_{12,29} & & & \\ \vdots & & & \\ A_{29,29} & A_{29,30} & \cdots & A_{29,50} \\ A_{30,29} & & & \\ \vdots & & & \\ A_{50,29} & & & \end{pmatrix},$$

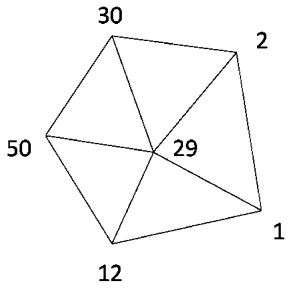


Fig. 2. All the vertices that bound the five elements around a given vertex 29.

$$P = \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_{12} \\ \vdots \\ P_{29} \\ P_{30} \\ \vdots \\ P_{50} \end{pmatrix}.$$

In the case where one maintains a complete set of mesh adjacencies, the determination of the vertices that are related to a given vertex can be found via constant time operations (i.e., O(1) operations). The first step determines the mesh entities defining the finite elements around a given vertex while the second step collects the set of vertices that bound those mesh entities. In the example shown in Fig. 3, it takes constant time to determine all the elements that M_1^0 bounds, which are elements $M_1^0\{M^2\}_1$, $M_1^0\{M^2\}_2$, $M_1^0\{M^2\}_3$, $M_1^0\{M^2\}_4$ and $M_1^0\{M^2\}_5$. For each of these five elements another constant time operation is required to determine all the bounding vertices, hence, constant time operations are used for collecting the set of vertices (combining M_2^0, M_3^0, M_4^0, M_5^0 and M_6^0) related to a given vertex M_1^0. For other numerical operations, and/or other equation discretization methods, different relationships between mesh entities are important. In each of those cases, the underlying discretization method determines such relationships which can be easily defined in terms of appropriate mesh adjacencies.

Classic approaches to define an effective labeling of the desired mesh entities rely on graph-based procedures [6,8,9], where the nodes in the graph are associated to the mesh entities of interest and graph edges are defined between graph nodes that are related to each other. Considering again the linear finite element example, there is a graph node for each mesh vertex

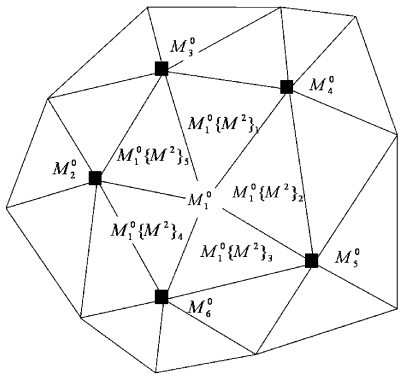


Fig. 3. Constant time operations are used to determine the adjacent vertices for a given vertex.

whereas graph edges are defined between nodes for which there will be a non-zero term in the resulting stiffness matrix. The key idea in graph-based labeling procedures [6,8,9] is that once a graph node is assigned a label, all the unlabeled graph nodes connected to it by edges are labeled as soon as possible. The primary differences in the various algorithms are the criteria used to decide the node to be labeled first and the following graph traversing mechanisms used in assigning labels to the unlabeled connected nodes. Typically one of the most time consuming, and most data intensive, aspects in the execution of these procedures is the definition of the graph from a given mesh. In the case where the starting data is the list of vertices for each element, this process requires traversing the elements, where a graph node is defined for each vertex of an element (if one does not exist already) and also graph edges are defined between graph nodes associated with the vertices of the element.

An alternative, but equivalent, approach to the classic graph-based labeling has been defined and is applied in the present work. This approach takes advantage of the fact that maintaining the adjacencies of a set of mesh entities in a complete representation. A complete mesh topology information is functionally equivalent to having a rich graph structure which houses any desired mesh entity relationship used in the classical method by simply stating the mesh entities and adjacencies to account for in the definition of the graph nodes and edges respectively. Equivalence is defined here in that once those mesh entities that will represent the graph nodes are known, the process of determining the graph edges between them due to mesh entity relationships can be performed in a number of O(1) operations per graph node [1]. Note that depending on the specific form of the complete mesh representation used, the number of operations can range from one to a few hundred, but each is of O(1) since it is not a function of the total number of entities in the mesh [2,25]. The mesh entity labeling algorithm presented here employs this equivalence to label the desired mesh entities based on a single traversal through selected mesh adjacencies which is equivalent to a specific graph traversal such as those used in the more classic procedures. In fact, by carefully defining the traversal process and the labeling order, these methods have been found to define orderings that, on average, produce slightly better matrix profiles for finite element meshes of any polynomial order when compared to Reverse Cuthill McKee and GPS.

3.1. Linear finite element data access

In the current work based on linear finite elements, the mesh entities to be labeled are the mesh regions ($\{M\{M^3\}\}$) that represent the finite elements and the mesh vertices ($\{M\{M^0\}\}$) since they are the mesh entities that hold the degrees of freedom in the matrix equation system. The relabeling of both the regions and vertices is important in the system formation phase which has four major steps, (1) localization of the data from global to element level, (2) computation of the residual (right-hand side) vector and tangent (left-hand side) matrix at the element level, (3) globalization of the residual vector from the element to the global level and (4) filling of entries in a sparse-matrix. It is also important in the basic kernel of sparse matrix–vector products during equation solution stage based on iterative schemes. The pseudo code of these primary steps are demonstrated in Algorithm 1 and the notation used in the pseudo code are defined in Table 1 (note that *ylocal* and *yglobal* describe the solution variable which is taken to be a scalar at each vertex in the pseudo code but in many interesting problems is actually a vector which inflates all ranks involving solution or residual by one, and tangent matrices by two (e.g., *LHS*(*nnz_tot*) becomes *LHS*(*nnz_tot*, *nvar*, *nvar*) while *Resglobal*(N_3) becomes *Resglobal*(N_3, *nvar*)).

In the process of localization and globalization (see pseudo code 1 and 3 in Algorithm 1), the mesh regions are traversed, where each vertex of the region is mapped to a global degree of freedom (*gdof*) which is in the same ordering with the mesh vertices. Relabeling both the regions and vertices increases the cache hits since the vertices of adjacent regions will have global degrees of freedom that are close. Then, for a given vertex *gdof*, if *yglobal*(*gdof*) and *Xglobal*(*gdof*)

Algorithm 1 Pseudo code of localization, element level integrals, globalization, sparse-matrix filling and AP product

1. Pseudo code of Localization

for $(e = 1; e <= N_3; e++)$ **do**
  for $(j_{nen} = 1; j_{nen} <= nen; j_{nen}++)$ **do**
    $gdof = ien(e, j_{nen})$
    $ylocal(e, j_{nen}) = yglobal(gdof))$
    $Xlocal(e, j_{nen}, 1 : nsd) = Xglobal(gdof, 1 : nsd))$
  end for
end for

2. Pseudo code of Element Level Integrals

for $(e = 1; e <= N_3; e++)$ **do**
  compute $Reslocal(e, 1 : nen)$ **and** $xKebe(e, 1 : nen, 1 : nen)$ **based on** $Xlocal(e, 1 : nen, 1 : nsd)$ **and** $ylocal(e, 1 : nen)$
end for

3. Pseudo code of Globalization

$Resglobal(1 : N_0) = 0$
for $(e = 1; e <= N_3; e++)$ **do**
  for $(j_{nen} = 1; j_{nen} <= nen; j_{nen}++)$ **do**
    $gdof = ien(e, j_{nen})$
    $Resglobal(gdof) += Reslocal(e, j_{nen})$
  end for
end for

4. Pseudo code of Sparse-Matrix Filling

$LHS(1 : nnz\_tot, 1 : nen, 1 : nen) = 0$
for $(e = 1; e <= N_3; e++)$ **do**
  for $(j_{nen} = 1; j_{nen} <= nen; j_{nen}++)$ **do**
    $gdof = ien(e, j_{nen})$
    $c = col(gdof),\ n = col(gdof + 1) - 1$
    for $(k_{nen} = 1; k_{nen} <= nen; k_{nen}++)$ **do**
      $b = ien(e, k_{nen})$
      $nzg =$ **search for the match of** b **in array** row **confined to** $istart = c$ **and** $istop = n$
      $LHS(nzg) += xKebe(e, j_{nen}, k_{nen})$
    end for
  end for
end for

5. Pseudo code of Sparse AP product

$q(1 : N_0) = 0$
for $(i = 1; i <= N_0; i++)$ **do**
  for $(k = col(i); k <= col(i + 1) - 1; k++)$ **do**
    $j = row(k)$
    $q(i) += LHS(k) * P(j)$
  end for
end for

Table 1
Definitions of symbols in the pseudo code

nen:	Number of vertices per element
N_0:	Number of vertices in the mesh
N_3:	Number of regions in the mesh
nsd:	Number of space dimensions
gdof:	Global degree of freedom
e:	Element number
j_{nen} & k_{nen}:	Node number local to element
nnz_tot:	Total number of non-zeros in stiffness matrix
ien(N_3, *nen*):	Connectivity array (e and j_{nen} yield *gdof*)
ylocal(N_3, *nen*):	Local variable array
yglobal(N_0):	Global variable array
Xlocal((N_3, *nen*, *nsd*):	Local coordinate
Xglobal(N_0, *nsd*):	Global coordinate
Reslocal(N_3, *nen*):	Local residual array
Resglobal(N_0):	Global residual array
xKebe(N_3, *nen*, *nen*):	Local stiffness matrix
LHS(*nnz_tot*):	Global sparse-matrix
col($N_0 + 1$) & *row*(*nnz_tot*):	Auxiliary arrays

(in localization phase) or *Resglobal*(*gdof*) (in globalization phase) are in the cache, the vertices in the adjacent regions which are processed next are likely to be in the cache resulting in cache hit. In the process of the sparse-matrix filling, a mapping from the vertices of each region to the global degrees of freedom is created as well. The position of this global degree of freedom in the sparse-matrix is then searched. Therefore vertex and region relabeling improves the cache performance in the process of sparse-matrix filling. However, the relabeling has no effect on the element-level integrals (see pseudo code 2 in Algorithm 1), in which the memory hierarchy is already used efficiently due to the localization of the data. Previous work [3] reports high performance improvement (over a factor of two) in the equation formation phase, since without a localization procedure, region and vertex ordering has more influence on the stiffness matrix and residual computations.

In the solution phase of the finite element analysis, the resulting non-linear algebraic equations are linearized to yield a system of linear equations which are solved using iterative procedures (e.g., GMRES [23,26]). Only the vertex relabeling is important to the equation solution phase. The sparse matrix–vector product (see pseudo code 5 in Algorithm 1) work is proportional to the number of global degrees of freedom. The sparse storage structure naturally stores the *LHS* entries in the order they will be used (sequential by row) but proper vertex order ensures that the items which follow each other in the product vector P (list of j) are close in memory despite the indirect addressing of the sparse storage format. The last key kernel in the iterative solve is the vector dot product but this is not discussed here since it naturally enjoys sequential access for any ordering. Note that currently diagonal preconditioning is employed, which is not affected by data reordering.

3.2. Adjacency-based reordering algorithm

Algorithm 2 for element and nodal labeling given below considers degrees of freedom associated with mesh vertices, edges, faces and regions, and with the assumption that mesh regions are finite elements. The reordering process starts with the first vertex and labels it with N_0, due to the reverse ordering (see steps 1–2). Then it loops over adjacent edges of the current vertex, M_k^0 (step 9). For each adjacent edge M_i^1, the elements bounded by this edge are traversed and labeled (if they have not been labeled). If these elements have nodes which have not been processed, then those nodes are pushed into the queue *nodeList* (see steps 10–21). Then the faces bounded by edge M_i^1 are traversed as well. If there are unprocessed nodes associated with these faces, then those nodes get pushed into the queue, otherwise, the algorithm proceeds (steps 22–29). If the other vertex (M_l^0) of edge M_i^1 is not yet processed, then unprocessed nodes associated with M_i^1 and M_l^0 get pushed into the queue. Otherwise, the algorithm labels the unprocessed nodes associated with edge M_i^1 immediately (steps 30–44). The reordering algorithm continues until *nodeList* is empty.

Though the current algorithm is developed and used for 3D unstructured meshes, for clarity, Fig. 4 uses a 2D example to explain the procedures for setting mesh entity labels. In the 2D context, mesh faces are the elements, which need to be labeled. The left picture of Fig. 4 illustrates the starting configuration of a mesh with 16 nodes and 5 faces, and the right one shows the new labels of the mesh nodes and elements. Table 2 demonstrates the status of the queue and the nodes labeled on each pass through. The reordering process starts with the first entity (node a) and labels it as 16, due to the reverse order. Then, it labels the adjacent element of node a as 5 (with circle in Fig. 4). b and c are two adjacent nodes of a through a mesh edge, which are not processed yet, therefore b and c are pushed into the queue. The process continues until the queue is

Algorithm 2 Pseudo code for nodal and element labeling

getstart(), take M_1^0 from the set $\{M\{M^0\}\}$, assign *labelnode* $= N_0$ and *labelelement* $= N_3$
push $M_1^0\{D\}_0$ back to *nodeList* and tag it to be in the *nodeList*
while *nodeList* is not empty **do**
  pop out the first node $M_k^d\{D\}_p$ from *nodeList*
  $M_k^d\{D\}_p \rightarrow$ *setlabel*(*labelnode* $--$)
  if M_k^d is not a vertex **then**
    continue
  else
    for $(i = 0; i < |\{M_k^0\{M^1\}\}|, i++)$ **do**
      take edge $M_i^1 = M_k^0\{M^1\}_i$
      for $(j = 0; j < |\{M_i^1\{M^3\}\}|, j++)$ **do**
        take element $M_j^3 = M_i^1\{M^3\}_j$
        if M_j^3 is not labeled **then**
          $M_j^3 \rightarrow$ *setlabel*(*labelelement* $--$)
        end if
        for $(p = 0; p < |\{M_j^3\{D\}\}|; p++)$ **do**
          if $M_j^3\{D\}_p$ is not labeled and $M_j^3\{D\}_p$ is not tagged to be in *nodeList* **then**
            push $M_j^3\{D\}_p$ back to *nodeList* and tag it to be in the *nodeList*
          end if
        end for
      end for
      for $(j = 0; j < |\{M_i^1\{M^2\}\}|, j++)$ **do**
        take face $M_j^2 = \{M_i^1\{M^2\}\}_j$
        for $(p = 0; p < |\{M_j^2\{D\}\}|; p++))$ **do**
          if $M_j^2\{D\}_p$ is not labeled and $M_j^2\{D\}_p$ is not tagged to be in *nodeList* **then**
            push $M_j^2\{D\}_p$ back to *nodeList* and tag it to be in the *nodeList*
          end if
        end for
      end for
      take the other vertex M_l^0 = $\{M_i^1\{M^0\}\}(M_k^0)$
      if $M_l^0\{D\}_0$ is not labeled and $M_l^0\{D\}_0$ is not tagged to be in *nodeList* **then**
        for $(p = 0; p < |\{M_i^1\{D\}\}|; p++)$ **do**
          if $M_i^1\{D\}_p$ is not labeled and $M_i^1\{D\}_p$ is not tagged to be in *nodeList* **then**
            push $M_i^1\{D\}_p$ back to *nodeList* and tag it to be in the *nodeList*
          end if
        end for
        push $M_l^0\{D\}_0$ back to *nodeList* and tag it to be in the list
      else
        for $(p = 0; p < |\{M_i^1\{D\}\}|; p++)$ **do**
          if $M_i^1\{D\}_p$ is not labeled and $M_i^1\{D\}_p$ is not tagged to be in *nodeList* **then**
            $M_i^1\{D\}_p \rightarrow$ *setlabel*(*labelnode* $--$)
          end if
        end for
      end if
    end for
  end if
end while

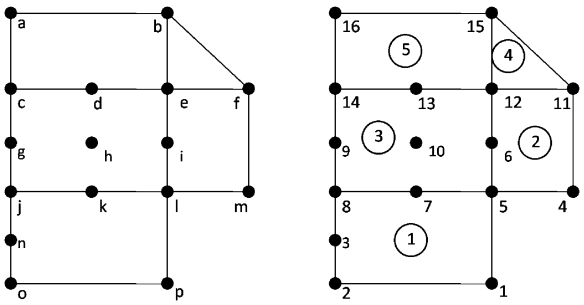


Fig. 4. A 2D example of nodal and element labeling.

Table 2
The status of the queue and the nodes labeled on each pass through

Queue	In process	Node label assigned
a	–	start of process-none assigned yet
b, c	a	16
c, e, f	b	15
e, f, h, g, j	c	14, 13 (d done since e already in queue)
f, h, g, j, i, l	e	12
h, g, j, i, l, m	f	11
g, j, i, l, m	h	10
j, i, l, m	g	9
i, l, m, n,o	j	8, 7 (k done since l already in queue)
l, m, n, o	i	6
m, n, o, p	l	5
n, o, p	m	4
o, p	n	3
p	o	2
–	p	1

empty. In the step 4, both nodes c and d are labeled when encountered since node e is already in queue, similarly in step 9.

After applying data reordering algorithm, the spatial data locality is increased. Figure 5 demonstrate the order of accessing the vector array P (array index j) in pseudo code 5 in Algorithm 1. This example considers a serial case with 54,195 non-zeros and 3977 vertices on a straight pipe. Without data reordering, the order to access vector array P is highly irregular, while the access is more sequential with the data reordering.

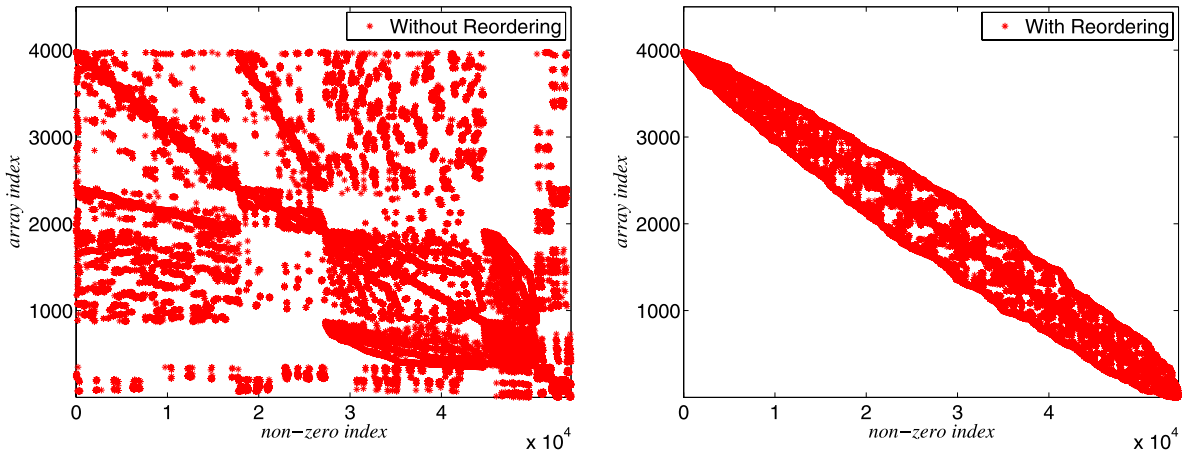
4. Results and discussions

In this section, two geometric models are used to study the performance improvement of the finite element analysis (FEA) solver (specifically PHASTA [14, 30] but referred to more generally as the FEA solver) by applying the adjacency-based data reordering algorithm presented in Section 3. The first example is a bifurcation pipe model which is shown in Fig. 6. The other example considers an abdominal aorta aneurysm (AAA) model (see Fig. 7). These two geometries and meshing approaches are used to study different aspects of the data ordering.

In the first study, the relative impact of the current data reordering algorithm on each phase of the FEA solver, including system formation (localization, globalization, element level integrals and sparse-matrix filling) and system solution is presented in Section 4.1. In Section 4.2, an 8.8M (where M denotes million) region mesh of the bifurcation pipe model is partitioned into different numbers of parts (from 128 up to 8192 parts), which allows a study of the effect of varying average work load per part in different partitions. These first two tests were performed on IBM Blue Gene/L at Rensselaer's Computational Center for Nanotechnology Innovations (CCNI). A similar study on an AAA model with a 133.6M region mesh is presented in Section 4.3. For this study, the computations are performed on various supercomputers, including IBM Blue Gene (BG/L and BG/P), Cray XT (XT3 and XT5) and Sun Constellation Cluster, to study the data reordering effect on various architectures. In Section 4.4, a hardware performance tool, CrayPat on Cray XT5 is used to predict the penalty due to cache misses, which confirms that the acceleration of FEA solver is due to the improved cache performance. In this test, the effect of data reordering algorithm is studied on cases with different load per part by fixing the number of processing cores and increasing the mesh size (i.e., the load per part changes from one case to another but in each case the load is balanced among parts). Note that one part per processing core is considered in this paper (i.e., the number of parts equals the number of processing cores) and the time usage of the FEA excludes the overhead of data reordering, which is negligible.

4.1. Data reordering relative effect

In this section, the effect of data reordering algorithm on each primary phase of FEA solver is studied on an 8.8M region mesh of the bifurcation pipe model (see Fig. 6). A parabolic steady inflow boundary condition is applied at the inlet of the bifurcation pipe model. The computation runs for 100 time steps with 3 nonlinear iterations per step using 128 CCNI BG/L cores. Table 3 provides the execution times for each primary phase of the finite element analysis, system formation (where localization and globalization, sparse-matrix



(a) Without data reordering (b) With data reordering

Fig. 5. Order to access vector array for computing AP product with and without data reordering algorithm.

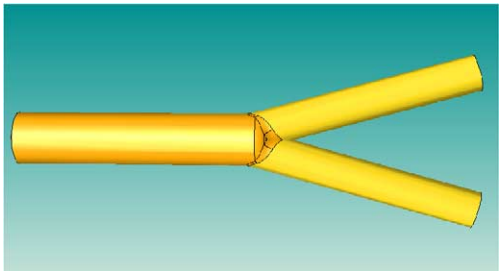


Fig. 6. Geometry model of a bifurcation pipe.

filling, element level integrals are the main components) and system solution. Furthermore, these times are compared between three strategies: without any reordering, reordering of vertices only and reordering of both vertices and regions. The values in the parentheses are the normalized time usage of each phase of FEA solver (normalized by total time usage of the analysis without any data reordering, which is 5422.8 s). It is clearly shown in the table that reordering, of both the vertices and elements, improves FEA solver performance the most among these strategies, which takes 81.7% of the original analysis time (the one without using data reordering), while reordering of only vertices improves the solver performance as well, which takes 83.6% of the original analysis time. Reordering both vertices and regions simultaneously improves the time usage of localization and globalization (from 1.1% to 0.5%), while (as expected) reordering of vertices only does not improve performance of localization and globalization steps. A similar observation can be made for the step of sparse-matrix filling. On the other hand, the data reordering algorithm has negligible effect on element-level integrals due to their use of localized data structures as mentioned in the previous section. Relabeling of vertices only improves the performance of the equation solution phase from 71.8% to 55.4%, but including region relabeling has negligible effect which is expected and also consistent with the discussion in Section 3. The relative improvement of each phase is obtained by comparing values in column 3 (reordering of vertices only) and 4 (reordering of vertices and elements) with column 2 (without data reordering). For example, relabeling of both vertices and elements improves localization and globalization by a factor of over two ($61.5/27.9 = 2.2$); a significant percentage gain but a small fraction of the total time. For the case of explicit time marching, data reordering accelerates system formation stage, however, this paper focuses on implicit analysis. Henceforth, overall performance improvement based on the total execution time of the analysis will be presented.

4.2. *The effect of load*

In these runs, the bifurcation pipe model is considered but now the mesh is partitioned into different numbers of parts. This allows us to study the effect of data reordering with different mesh sizes or load per processing core. To create different mesh partitions, the mesh database (FMDB) is interfaced with Zoltan library [7] where the graph based partitioning package ParMETIS [15] is used. Table 4 lists the average number of mesh regions and vertices per part for different partitions of an 8.8M element uniformly refined tetrahedral mesh. A wide range of average number of el-

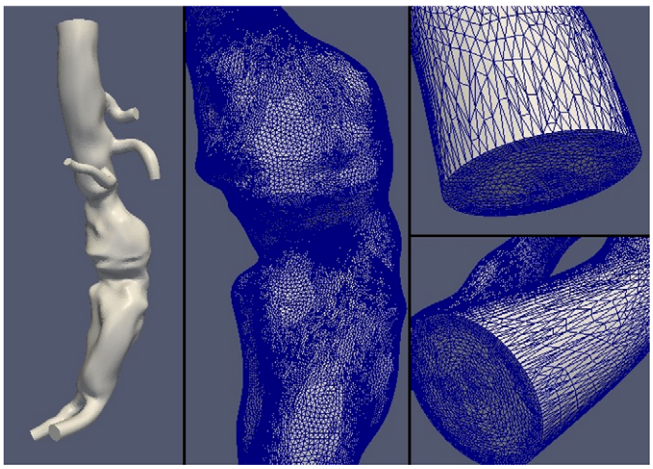


Fig. 7. Geometry and the mesh of AAA model.

Table 3
Data reordering effect on different phases of implicit FEA

Time in seconds (in %)	Without data reordering	Reordering of vertices only	Reordering of vertices and elements
Localization and globalization	61.5 (1.1%)	67.2 (1.2%)	27.9 (0.5%)
Sparse-matrix filling	163.8 (3.0%)	162.8 (3.1%)	132.2 (2.4%)
Element level integral	997.7 (18.4%)	998.6 (18.4%)	994.3 (18.3%)
Total of system formation	1520.4 (28.0%)	1524.1 (28.1%)	1429.9 (26.3%)
System solution	3894.9 (71.8%)	3005.7 (55.4%)	2992.8 (55.2%)
Total of analysis	5422.8 (100%)	4537.3 (83.6%)	4430.3 (81.7%)

Table 4
Average number of regions and vertices per part of an 8.8M region mesh of a bifurcation pipe model

Number of parts	128	256	512	1024	2048	4096	8192
Avg. rgn./part	68,856	34,463	17,231	8616	4308	2154	1077
Avg. vtx./part	14,020	7262	3780	1984	1050	560	306

ements per part is used, where it varies from as high as (about) hundred thousand (required for large simulations) on a partition with 128 parts to as few as several thousand (useful for time critical runs) on a partition with 8192 parts. It is worth noting that the average number of vertices per part does not decrease at the same rate as the regions when a fixed-size problem is spread into more and more parts. For example, 14,020 is the average number of mesh vertices per part in the partition with 128 parts and it is 306 in the 8192 part partition; which is about 39.7% higher relatively (219 mesh vertices will be ideal based on the average of 14,020 when going from 128 parts to 8192 parts). This occurs due to an increase in mesh vertices shared at inter-part boundaries. These tests are performed on IBM Blue Gene/L at CCNI.

The performance improvement ratio obtained due to data reordering for each partition is shown in Fig. 8. The time usage of FEA solver with and without using data reordering algorithm along with improvement is listed in Table 5. The algorithm has greater effect on heavily loaded processing cores, which can be easily observed in Fig. 8. This happens because in the run with heavily loaded parts (i.e., run on a lower core count) relatively more time is spent accessing the memory compared to the lightly loaded computation (higher core count). Also the larger the data a given part must hold, the smaller the percentage of total data will be resident in cache, so more misses are more likely when the data is randomly ordered and accessed. The data reordering algorithm attempts to produce a more uniform rate of accessing data from memory for all mesh sizes. This algorithm improves the FEA per-

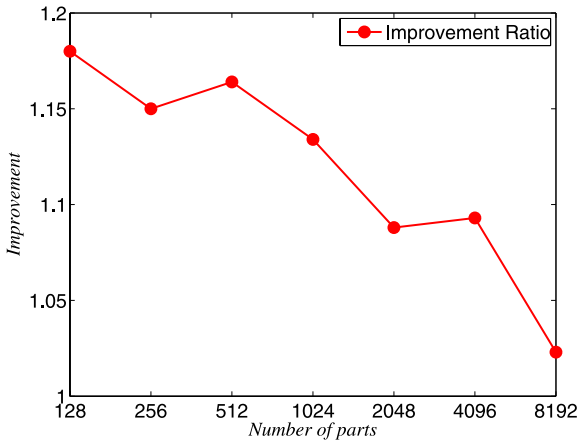


Fig. 8. FEA solver performance improvement due to current data reordering algorithm on a bifurcation pipe model on CCNI's BG/L.

formance by 18.0% for the run on 128 parts, while it improves 9.3% on the 4096 part run and 2.3% on the 8192 part run. Though the improvement diminishes as the load is reduced, data reordering reduced the computational time for all the test cases.

4.3. The effect of different architectures

Another example considers a 133.6M element non-uniform, anisotropic, tetrahedral mesh created by applying mesh adaptation [24] on AAA model (see Fig. 7). A Womersley profile [32] is applied at the inlet and rigid vessel walls with no-slip boundary condition are considered. At outlets a lumped parameter boundary condition [28] is applied. The computation is run for 20 steps and the time usage for different partitions on different supercomputers with and without using data reordering algorithm is measured, see Table 6. The performance improvement ratio obtained by applying the current data reordering algorithm is shown in Fig. 9.

This result is consistent with the one shown in Section 4.2 in that the data reordering algorithm has more benefit in cases with heavily loaded processing cores (i.e., in runs on lower number of processors). It also suggests that the amount of performance gain that can be achieved by applying data reordering algorithm is dependent on the system architecture as well. In Table 7 we summarize the supercomputer systems considered in this study.

The data reordering algorithm has a more significant effect on the Blue Gene (up to 23.6% on BG/L and up to 21.9% on BG/P) due to a much smaller L2 cache, which is a prefetch buffer against L3. The L2 cache of Blue Gene (BG/L and BG/P) is only 2 KB in size compared with 1 MB in the case of Bigben and 512 KB in the case of Ranger and Kraken. The consequence of this dramatic difference in L2 cache size is that 4096 Ranger/Kraken processing cores have $4\text{K} \times 0.5\,\text{MB} = 2\,\text{GB}$ L2 cache in total, but same number of processing cores on Blue Gene have only $4\text{K} \times 2\,\text{KB} = 8\,\text{MB}$ L2 cache which is $1/256$ of the other systems. This cache size difference also explains why the performance improvement diminishes when reaching 4096 processing cores on Ranger, Bigben and Kraken, but still has over 20% performance improvement on Blue Gene systems. Because the L2 cache on the Blue Gene systems is prefetch driven, the reordering algorithm has a greater sustained positive impact on cache performance since it orders data items in such a way as to make it obvious to the prefetcher which data items to go after next.

4.4. Cache performance modeling

On the Cray XT5 (NICS Kraken), the hardware performance tool of CrayPat is used to predict the penalty due to cache misses. In these runs, the effect of data reordering on the cases with different load per part is studied by fixing the number of processing cores and increasing the mesh size. Note this study is complementary to the strong scaling study (fixed-size problem distributed into different number of processing cores) discussed earlier. Hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory, etc. The hardware performance, specifically cache performance, is studied in this section. In this study four meshes of different sizes are used, where 8.8M element mesh of a bifurcation pipe model (see Fig. 6) is adapted to obtain three additional meshes of different sizes that have 22M, 42M and 92M elements, respectively. Computation based on each mesh involves 5 time steps on 128 cores (16 nodes). Table 8 compares the time usage of FEA solver with and without using data reordering algorithm.

The NICS Kraken-XT5 system consists of compute nodes with 2.3 GHz quad-core AMD Opteron processors (Barcelona). It has 64 KB L1 cache per core for data (and same for instructions) with a 3 cycle latency, a 512 KB L2 cache per core with a 12 cycle latency [29], and 2 MB L3 cache shared among cores on a socket (or quad-core processor) with a 38 cycle latency [12]. Note that the 38 cycle latency is a reported average. It turns out that the L3 cache may have

Table 5
Time usage of FEA solver on a bifurcation pipe model on CCNI's BG/L along with performance improvement

Number of parts	128	256	512	1024	2048	4096	8192
Without reordering (s)	5422.8	2643.1	1316.3	679.0	351.7	201.3	136.5
With reordering (s)	4430.3	2246.6	1100.0	587.8	320.7	182.6	133.3
Improvement (%)	18.0	15.0	16.4	13.4	8.8	9.3	2.3

Table 6
Time usage of FEA solver on an AAA model with and without using data reordering algorithm on different supercomputers along with performance improvement

Number of parts		1024	2048	4096	8192
CCNI-BGL	Without (s)	492.96	246.50	123.88	61.89
IBM BG/L	With (s)	376.68	191.70	98.88	50.94
	Improvement	23.6%	22.2%	20.2%	17.7%
Intrepid, ANL	Without (s)	445.70	223.04	111.40	57.04
IBM BG/P	With (s)	348.08	177.88	90.89	46.91
	Improvement	21.9%	20.2%	18.4%	17.7%
Kraken, NICS	Without (s)	133.27	59.66	31.23	–
Cray XT5	With (s)	118.99	56.12	30.74	–
	Improvement	10.7%	5.9%	1.6%	–
Ranger, TACC	Without (s)	194.22	91.33	48.42	–
Sun Cluster	With (s)	166.33	84.19	47.55	–
	Improvement	14.4%	7.8%	1.8%	–
Bigben, PSC	Without (s)	155.53	64.51	30.03	–
Cray XT3	With (s)	132.31	57.65	28.92	–
	Improvement	14.9%	10.6%	3.7%	–

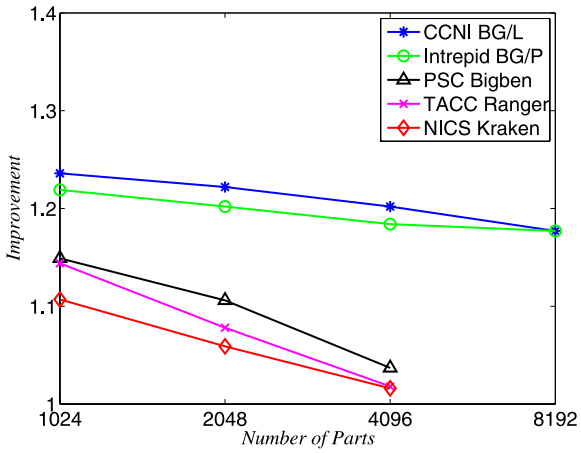


Fig. 9. FEA solver performance improvement using current data reordering algorithm on an AAA model on various supercomputers including IBM Blue Gene (CCNI-BGL and Intrepid-BGP), Cray XT (Bigben-XT3 and Kraken-XT5) and Sun Constellation Cluster (Ranger).

some additional latency for other reasons: (i) cache access between the four cores is doled out in a round-robin fashion, and (ii) FIFO buffers that sit in front of this cache in order to deal with cores running at what may be vastly different clock speeds. Additionally, the speed of the Barcelona's L3 cache varies between 1.8 and 2.0 GHz depending on the processor model [13].

The total penalty difference due to changes in cache misses could be computed by a simplified model as:

$$(\Delta L1\_DCM * L1\_Latency + \Delta L2\_DCM * L2\_Latency + \Delta L3\_CM * L3\_Latency)/Clock\_speed, \quad (1)$$

where $\Delta L1\_DCM$ is the change in L1 data cache misses, $\Delta L2\_DCM$ is the change in L2 data cache misses, and $\Delta L3\_CM$ is the change in L3 cache misses. The CrayPat report of cache misses is listed in Table 9 and the penalty differences due to cache misses are computed for the case with and without using the data reordering algorithm based on Eq. (1). Note, L1_DCA in the table refers to L1 data cache access. The penalty differences due to cache misses (last column of Table 9) are close to the differences measured in the execution time of the analysis with and without using data

Table 7
Summary of systems considered in this study. SPN refers to sockets per node, CPCN to cores per compute node and CPS to cores per socket. The tests reported in this paper use all the cores on each node. S refers to shared cache among cores on a socket while P refers to private cache on cores of a socket

System name	CCNI-BGL@RPI	Intrepid@ALCF	Bigben@PSC	Ranger@TACC	Kraken@NICS
System type	IBM BG/L	IBM BG/P	Cray XT3	Sun Const.	Cray XT5
Proc. type	PPC 700 MHz	PPC 850 MHz	AMD 2.6 GHz	AMD 2.3 GHz	AMD 2.3 GHz
SPN	1	1	1	4	2
CPS	2	4	2	4	4
CPCN	2	4	2	16	8
L1 size	32 KB/core	32 KB/core	64 KB/core	64 K/core	64 KB/core
L2 size	2 KB/core	2 KB/core	1 MB/core (P)	512 KB/core	512 KB/core
L3 size	4 MB/socket (S)	8 MB/socket (S)	–	2 MB/socket (S)	2 MB/socket (S)

Table 8
Time usage of FEA solver on different meshes with and without using data reordering algorithm on NICS Kraken-XT5

Mesh size	8.8M	22M	42M	92M
Without (s)	69.80	196.72	444.08	1091.62
With (s)	64.34	153.71	307.41	654.46
Difference (s)	5.46	43.01	136.27	437.15

reordering algorithm shown in the last row of Table 8. For the case with 92M region mesh, the prediction of time difference due to cache miss penalty (464.13 s) is higher than the total difference observed in execution time (437.15 s) with and without data reordering. This is because the AMD Barcelona processor has an innovative prefetching algorithm that attempts to prefetch data in the L1 data cache and DRAM levels [29]. Due to such a data prefetching capability, some of the cache misses are hidden since the instruction pipeline will not stall on prefetched data that misses at any level in the cache. Thus, the time to service a cache miss is not incurred as a real execution time penalty since instruction processing continues during the processing of the cache miss. But at small mesh sizes, the amount of main memory used is much less and so there is not as much an opportunity to prefetch within a single iteration.

Additionally, Table 9 shows that the most dramatic improvement in memory hierarchy efficiency occurs in the L2 data cache. Here, it is observed that for the 42M region mesh case, L2 data cache misses are 311% higher without reordering than with reordering (e.g., 5.3 billion without reordering vs. 1.7 billion with reordering) and for the 92M region mesh case this trend increases to 366% more cache misses without reordering (e.g., 14.3 billion without reordering vs. 3.9 billion with reordering).

As mentioned before, these tests study the data reordering effect on different load per core cases by fixing the number of processing cores and increasing the mesh size. The data in Table 9 is normalized by the lightest loaded case (8.8M region mesh case), i.e., all the values are divided by the first row in Table 9. The normalized cache access and cache misses information is provided in Table 10.

As the mesh size increases (from 8.8M to 22M, 42M and 92M), the computational load per part increases to 2.5, 4.8 and 10.5 times, respectively. The L1 data cache access (L1_DCA) increases at the same ratio, which is expected. It also worth noting that, for a fixed-size problem on the same number of processing cores, L1_DCA is expected to be very close (comparing columns 2 and 6 in Table 9). The cache misses on each of the L1, L2 and L3 levels increase as the problem size increases as well but not in a proportional way due to fundamental differences in cache performance created by reordering the data. Specifically, comparing columns 3 and 7 we see that L1 cache misses increase at a faster rate than the mesh size growth when the data is not reordered. To see the difference in the growth rates more clearly we have rescaled each row of Table 10 by the first column and placed the results in parentheses (e.g., 1.3 indicates a 30% relative increase in cache misses for the largest mesh). The effect is even more dramatic on the L2 cache (170%) and L3 cache (100%) misses.

With data reordering, the increase in cache misses is close to the rate of the problem size increasing. The normalized L1_DCM by the relative problem size are 1.0, 1.0, 1.0 and 0.98, respectively with data reordering algorithm (column 7 of Table 10 in parentheses). This confirms the observation in Section 4.2 that the cases with heavily loaded cores spend disproportionally more time accessing the memory compared to the lightly loaded processors. L2 and L3 cache results are less uniform but still show substantially more stable

Table 9
Comparison of cache access and cache misses for different meshes of a bifurcation model on NICS Kraken-XT5 with and without using data reordering

Mesh	Without data reordering (bil.)				With data reordering (bil.)				Penalty diff. (s)
	L1_DCA	L1_DCM	L2_DCM	L3_CM	L1_DCA	L1_DCM	L2_DCM	L3_CM	
8.8M	32.9	3.1	0.5	1.8	32.9	2.5	0.3	1.6	5.13
22M	80.4	9.2	1.9	5.32	80.6	6.2	0.9	3.4	40.85
42M	152.7	18.9	5.3	13.5	153.1	11.9	1.7	7.9	120.43
92M	332.0	41.1	14.3	39.2	333.4	25.8	3.9	15.6	464.13

Table 10
Comparison of normalized cache access and cache misses for different meshes of a bifurcation model on NICS Kraken-XT5 with and without using data reordering algorithm

Mesh	Without data reordering				With data reordering			
	L1_DCA	L1_DCM	L2_DCM	L3_CM	L1_DCA	L1_DCM	L2_DCM	L3_CM
1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
2.5 (1.0)	2.4 (0.96)	3.0 (1.2)	3.8 (1.5)	3.0 (1.2)	2.4 (0.96)	2.5 (1.0)	3.0 (1.2)	2.1 (0.84)
4.8 (1.0)	4.6 (0.96)	6.1 (1.3)	10.6 (2.2)	7.5 (1.6)	4.7 (0.98)	4.8 (1.0)	5.7 (1.2)	4.9 (1.0)
10.5 (1.0)	10.1 (0.96)	13.4 (1.3)	28.6 (2.7)	21.8 (2.0)	10.1 (0.96)	10.3 (0.98)	13.0 (1.2)	9.75 (0.93)

Note: Case of 8.8M region mesh is taken as the base to normalize, i.e., normalized by the first row in Table 9. The information is normalized again (see numbers in the parentheses) by the relative work load which is in the first column (of a given row) of this table.

Table 11
Scaling factors (s_e) of FEA solver on different meshes with and without using data reordering algorithm on NICS Kraken-XT5

Mesh size	8.8M	22M	42M	92M
Without (s_e)	1.496	1.327	1.222	1.0
With (s_e)	0.973	1.018	0.972	1.0
Improvement	7.8%	21.9%	30.7%	40.0%

results as compared to cases without reordering. The current data reordering algorithm improves the cache performance in general and is reasonably successful at producing the same rate of data access from memory for all mesh sizes.

With this understanding of the cache performance, we have the opportunity to revisit the data from Table 8. Table 11 presents that data with rows 2 and 3 rescaled to give the time spent per element in the 92M element case relative to the time spend per element in the each case/column (e.g., $s_e = (t_{92}/92)/(t_i/n_i)$ where t_i is the time spent on the ith mesh (column) and n_i is the number of elements in the ith mesh). Without data reordering we appear to have super-linear speedup as we reduce the mesh size (better cache utilization due to smaller range of data). With data ordering we have a more scalable cache performance (much closer to 1). While we have given up the passive super-linear performance, the data ordering algorithm introduced here has replaced it with something that is uniformly better. Table 11 also shows the relative improvement in the fourth row (i.e., row 4 divided by row 2 of Table 8). Improvement is highest in the most heavily loaded case (i.e., nearly 40% in the 92M element case).

It is worth noting that the time spent in reordering the mesh for all of the above studies is negligible compared to the time spent in pre-processing the mesh (preparing boundary conditions, initial conditions and partitioning). This pre-processing time in turn is negligible compared to the analysis time for simulations of practical interest. A review of Algorithm 2 confirms that the reordering algorithm is O(n). We can further confirm this order and that the constant is small by considering a fairly large example that results in a negligible relative cost. For example, the pre-processing was performed on 512 processing cores on Intrepid for the case on an AAA model with 133.6M elements. The time on pre-processing was 190 s, while the time spent on the data reordering was 7 s which is less than one aggregate CPU. It takes 445.70 s to run 20 steps of finite element analysis on 1024 Intrepid cores and one cardiac cycle usually includes several thousand time steps, e.g., 5000. After applying data reordering, it takes 348.08 s for 20 steps. Data reordering saves aggregate 6942 CPU hours on the finite element analysis per cycle and it takes several cardiac cycles for finite element to converge before next mesh adaptation and pre-processing. This is a substantial savings in computational time.

5. Conclusions

In this work, we have implemented and tested a data reordering algorithm in a finite element analysis flow solver that improves the flop rate by relabeling mesh regions and mesh vertices. Mesh entities are reordered such that those visited one after another are topologically close to each other. This algorithm improves the usage of the memory hierarchy during on processor calculations. Two examples are considered, one is a bifurcation pipe model and the other is an abdominal aortic aneurysm model. The following strategies are emphasized to demonstrate the usefulness of current data reordering algorithm:

- The first test was on an 8.8M region mesh of a bifurcation pipe model where the relative impact of the data reordering algorithm on each phase of FEA solver was studied. Results showed that the reordering of both the vertices and regions simultaneously is important in the equation formation phase while reordering of vertices improves performance in equation solution stage.
- In the second test, same 8.8M region mesh of a bifurcation pipe model was partitioned into different number of parts, which yields different average work load per part. This test indicates that the improvement due to the data reordering algorithm decays as the parts become lightly loaded.
- A similar test was performed on a much larger mesh (133.6M regions) on an AAA model considering different architectures including IBM Blue Gene (BG/L and BG/P), Cray XT (XT3 and XT5) and Sun Constellation Cluster. It additionally suggests that the amount of performance gain that can be achieved by applying data reordering algorithm is dependent on the system architecture as well.
- In the last test on Cray XT5, the data reordering effect was studied with different working load per part by fixing the number of processing cores and increasing the mesh size. The hardware performance tool of CrayPat was used to measure the cache misses. This data was combined with an estimate of the miss penalty, to model the acceleration achieved by the FEA solver with data reordering. It closely accounts for the observed decrease in the overall execution time.

Future work will include the data reordering algorithm using alternative mesh database descriptions (e.g., reduced complete representation) and alternative discretizations (e.g., higher order finite elements or various finite volume discretizations).

Acknowledgements

This work is supported in part by the US Department of Energy under Grant DE-FC02-06ER25769 and in part by the National Science Foundation under Grant 0749152. Computing resources used were provided by support from NSF through Teragrid resources including systems at TACC, PSC and NICS. Computing resources used at the Rensselaer Computational Center for Nanotechnology Innovations was funded by the State of New York, RPI and IBM. This research also used computing resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the US Department of Energy under Contract DE-AC02-06CH11357. The solution presented herein made use of the linear algebra solution library provided by Acusim Software Inc. and meshing and geometric modeling libraries by Simmetrix Inc. The authors would like to thank the following individuals for their kind assistance: Adam Todorski (CCNI, RPI), Karen Devine (SNL) and also David McWilliams (NICS) for guidance on using Craypat.

References

[1] M. Beall, An object-oriented framework for the reliable automated solution of problems in mathematical physics, PhD thesis, Rensselaer Polytechnic Institute, May 1999.

[2] M.W. Beall and M.S. Shephard, A general topology-based mesh data structure, *Int. J. Numer. Methods Engrg.* **40**(9) (1997), 1573–1596.

[3] D.A. Burgess and M.B. Giles, Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines, *Adv. Engrg. Soft.* **28** (1997), 189–201.

[4] A.L.G.A. Coutinho and M.A.D. Martins, Performance comparison of data-reordering algorithms for sparse matrix–vector multiplication in edge-based unstructured grid computations, *Int. J. Numer. Methods Engrg.* **66** (2006), 431–460.

[5] H.L. Crane Jr., N.E. Gibbs, W.G. Poole Jr. and P.K. Stockmeyer, Matrix bandwidth and profile reduction, *ACM Trans. Math. Soft.* **2** (1976), 375–377.

[6] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: *Proc. 24th Nat. Conf. ACM*, ACM, New York, NY, USA, 1969, pp. 157–172.

[7] K. Devine, E. Boman, R. Heaphy, B. Hendrickson and C. Vaughan, Zoltan data management services for parallel dynamic applications, *Comput. Sci. Eng.* **4**(2) (2002), 90–97.

[8] N.E. Gibbs, W.G. Poole Jr. and P.K. Stockmeyer, An algorithm for reducing the bandwidth and profile of sparse matrix, *SIAM J. Numer. Anal.* **13** (1976), 236–250.

[9] N.E. Gibbs, W.G. Poole Jr. and P.K. Stockmeyer, A comparison of several bandwidth and profile reduction algorithm, *ACM Trans. Math. Soft.* **2** (1976), 322–330.

[10] H. Han and C.W. Tseng, Exploiting locality for irregular scientific codes, *IEEE Trans. Parallel Distrib. Systems* **17** (2006), 606–618.

[11] G. Heber, R. Biswas and G.R. Gao, Self-avoiding walks over adaptive unstructured grids, *Concurrency Pract. Exper.* **12** (2000), 85–109.

[12] http://www.anandtech.com/.

[13] http://techreport.com/articles.x/13176/3.

[14] K.E. Jansen, C.H. Whiting and G.M. Hulbert, Generalized-α method for integrating the filtered Navier–Stokes equations with a stabilized finite element method, *Comput. Methods Appl. Mech. Engrg.* **190**(3,4) (2000), 305–319.

[15] G. Karypis and V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, in: *10th Intl. Parallel Processing Symposium*, IEEE Computer Society, Washington, DC, USA, 1996, pp. 314–319.

[16] R. Löhner, Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines, *Comput. Methods Appl. Mech. Engrg.* **163** (1998), 95–109.

[17] R. Löhner and M. Galle, Minimization of indirect addressing for edge-based field solvers, *Commun. Numer. Methods Engrg.* **18**(5) (2002), 335–343.

[18] L. Oliker, X. Li, P. Husbands and R. Biswas, Ordering schemes for sparse matrices using modern programming paradigms, LBNL47803, 2000.

[19] L. Oliker, X. Li, P. Husbands and R. Biswas, Parallel conjugate gradient: effects of ordering strategies, programming paradigms, and architectural platforms, LBNL45828, 2000.

[20] L. Oliker, X. Li, P. Husbands and R. Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, *SIAM Rev.* **44** (2002), 373–393.

[21] J.C. Pichel, D.E. Singh and J. Carretero, Reordering algorithms for increasing locality on multicore processors, in: *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, IEEE Computer Society, Washington, DC, USA, September 2008, pp. 123–130.

[22] J.F. Remacle, O. Klaas, J.E. Flaherty and M.S. Shephard, A parallel algorithm oriented mesh database, *Eng. Comput.* **18** (2002), 274–284.

[23] Y. Saad and M.H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Statist. Comput.* **7** (1986), 856–869.

[24] O. Sahni, Y. Müller, K.E. Jansen, M.S. Shephard and C.A. Taylor, Efficient anisotropic adaptive discretization of the cardiovascular system, *Comput. Methods Appl. Mech. Engrg.* **195** (2006), 5634–5655.

[25] E.S. Seol and M.S. Shephard, Efficient distributed mesh data structure for parallel automated adaptive analysis, *Eng. Comput.* **22**(3,4) (2006), 197–213.

[26] F. Shakib, T.J.R. Hughes and Z. Johan, A multi-element group preconditioned GMRES algorithm for nonsymmetric systems arising in finite element analysis, *Comput. Methods Appl. Mech. Engrg.* **75** (1989), 415–456.

[27] M.M. Strout and P.D. Hovland, Metrics and models for reordering transformations, in: *Proceedings of the 2nd ACM SIGPLAN Workshop on Memory System Performance (MSP)*, June 2005, ACM, New York, NY, USA, 2005, pp. 23–34.

[28] I.E. Vigon-Clementel, C.A. Figueroa, K.E. Jansen and C.A. Taylor, Outflow boundary conditions for three-dimensional finite element modeling of blood flow and pressure in arteries, *Comput. Mehtods Appl. Mech. Engrg.* **195** (2006), 3776–3796.

[29] B. Waldecker, Amd technical briefing, available at: http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/AMD_ ORNL_073007.pdf.

[30] C.H. Whiting and K.E. Jansen, A stabilized finite element method for the incompressible Navier–Stokes equations using a hierarchical basis, *Int. J. Numer. Meth. Fluids* **35** (2001), 93–116.

[31] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, Optimization of sparse matrix–vector multiplication on emerging multicore platforms, *Parallel Comput.* **35** (2009), 178–194.

[32] J. Womersley, Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known, *J. Physiol.* **127** (1955), 553–563.

[33] A.N. Yzelman and R.H. Bisseling, Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods, *SIAM J. Sci. Comput.* **31** (2009), 3128–3254.